Intel® Itanium® Architecture Software Developer's Manual

Revision 2.3

Volume 1: Application Architecture

Intel® Itanium® Architecture Software Developer's Manual

Volume 1: Application Architecture

Revision 2.3

May 2010

Document Number: 245317

THIS DOCUMENT IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel® processors based on the Itanium architecture may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling1-800-548-4725, or by visiting Intel's website at http://www.intel.com.

Contents

Part I: Application Architecture Guide

Part II: Optimization Guide for the Intel® Itanium® Architecture

Figures

Part I: Application Architecture Guide

Part II: Optimization Guide for the Intel® Itanium® Architecture

Tables

Part I: Application Architecture Guide

Part II: Optimization Guide for the Intel® Itanium® Architecture

§

Part I: Application Architecture Guide

About this Manual 1

The Intel® Itanium® architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The Itanium architecture features a revolutionary 64-bit instruction set architecture (ISA), which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the Itanium architecture is IA-32 instruction set compatibility.

The *Intel® Itanium® Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of the features of the Itanium architecture to help them optimize code.

1.1 Overview of Volume 1: Application Architecture

This volume defines the Itanium application architecture, including application level resources, programming environment, and the IA-32 application interface. This volume also describes optimization techniques used to generate high performance software.

1.1.1 Part 1: Application Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual.Intel® Itanium® Architecture Software Developer's Manual*

Chapter 2, "Introduction to the Intel® Itanium® Architecture" provides an overview of the architecture.

Chapter 3, "Execution Environment" describes the Itanium register set used by applications and the memory organization models.

Chapter 4, "Application Programming Model" gives an overview of the behavior of Itanium application instructions (grouped into related functions).

Chapter 5, "Floating-point Programming Model" describes the Itanium floating-point architecture (including integer multiply).

Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium® System Environment" describes the operation of IA-32 instructions within the Itanium System Environment from the perspective of an application programmer.

1.1.2 Part 2: Optimization Guide for the Intel® Itanium® Architecture

Chapter 1, "About the Optimization Guide" gives an overview of the optimization guide.

Chapter 2, “Introduction to Programming for the Intel® Itanium® Architecture” provides an overview of the application programming environment for the Itanium architecture.

Chapter 3, “Memory Reference” discusses features and optimizations related to control and data speculation.

Chapter 4, “Predication, Control Flow, and Instruction Stream” describes optimization features related to predication, control flow, and branch hints.

Chapter 5, “Software Pipelining and Loop Support” provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 6, “Floating-point Applications” discusses current performance limitations in floating-point applications and features that address these limitations.

1.2 Overview of Volume 2: System Architecture

This volume defines the Itanium system architecture, including system level resources and programming state, interrupt model, and processor firmware interface. This volume also provides a useful system programmer's guide for writing high performance system software.

1.2.1 Part 1: System Architecture Guide

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Intel® Itanium® System Environment” introduces the environment designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications.

Chapter 3, “System State and Programming Model” describes the Itanium architectural state which is visible only to an operating system.

Chapter 4, “Addressing and Protection” defines the resources available to the operating system for virtual to physical address translation, virtual aliasing, physical addressing, and memory ordering.

Chapter 5, “Interruptions” describes all interruptions that can be generated by a processor based on the Itanium architecture.

Chapter 6, “Register Stack Engine” describes the architectural mechanism which automatically saves and restores the stacked subset (GR32 – GR 127) of the general register file.

Chapter 7, “Debugging and Performance Monitoring” is an overview of the performance monitoring and debugging resources that are available in the Itanium architecture.

Chapter 8, “Interruption Vector Descriptions” lists all interruption vectors.

Chapter 9, “IA-32 Interruption Vector Descriptions” lists IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment.

Chapter 10, “Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications” defines the operation of IA-32 instructions within the Itanium System Environment from the perspective of an Itanium architecture-based operating system.

Chapter 11, “Processor Abstraction Layer” describes the firmware layer which abstracts processor implementation-dependent features.

1.2.2 Part 2: System Programmer’s Guide

Chapter 1, “About the System Programmer’s Guide” gives an introduction to the second section of the system architecture guide.

Chapter 2, “MP Coherence and Synchronization” describes multiprocessing synchronization primitives and the Itanium memory ordering model.

Chapter 3, “Interruptions and Serialization” describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents and state. This chapter also describes system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 5, “Memory Management” introduces various memory management strategies.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

Chapter 9, “IA-32 Application Support” describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

Chapter 13, “Firmware Overview” introduces the firmware model, and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

Appendix A, “Code Examples” provides OS boot flow sample code.

1.3 Overview of Volume 3: Intel® Itanium® Instruction Set Reference

This volume is a comprehensive reference to the Itanium instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Instruction Reference” provides a detailed description of all Itanium instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “Pseudo-Code Functions” provides a table of pseudo-code functions which are used to define the behavior of the Itanium instructions.

Chapter 4, “Instruction Formats” describes the encoding and instruction format instructions.

Chapter 5, “Resource and Dependency Semantics” summarizes the dependency rules that are applicable when generating code for processors based on the Itanium architecture.

1.4 Overview of Volume 4: IA-32 Instruction Set Reference

This volume is a comprehensive reference to the IA-32 instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Base IA-32 Instruction Reference” provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-32 Intel® MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 4, “IA-32 SSE Instruction Reference” provides a detailed description of all IA-32 SSE instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.5 Terminology

The following definitions are for terms related to the Itanium architecture and will be used throughout this document:

Instruction Set Architecture (ISA) – Defines application and system level resources. These resources include instructions and registers.

Itanium Architecture – The new ISA with 64-bit instruction capabilities, new performance- enhancing features, and support for the IA-32 instruction set.

IA-32 Architecture – The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual***.

Itanium System Environment – The operating system environment that supports the execution of both IA-32 and Itanium architecture-based code.

Itanium Architecture-based Firmware – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

Processor Abstraction Layer (PAL) – The firmware layer which abstracts processor features that are implementation dependent.

System Abstraction Layer (SAL) – The firmware layer which abstracts system features that are implementation dependent.

1.6 Related Documents

The following documents can be downloaded at the Intel’s Developer Site at http://developer.intel.com:

- ***Dual-Core Update to the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization***– Document number 308065 provides model-specific information about the dual-core Itanium processors.
- ***Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*** – This document (Document number 251110) describes model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.
- ***Intel® Itanium® Processor Reference Manual for Software Development*** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.

- ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.
- ***Intel® Itanium® Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.

The following document can be downloaded at the Unified EFI Forum website at http://www.uefi.org:

- ***Unified Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware.

1.7 Revision History

Date of Revision	Revision Number	Description
March 2010	2.3	Added information about illegal virtualization optimization combinations and IIPA requirements. Added Resource Utilization Counter and PAL_VP_INFO. PAL_VP_INIT and VPD.vpr changes. New PAL_VPS_RESUME_HANDLER parameter to indicate RSE Current Frame Load Enable setting at the target instruction. PAL_VP_INIT_ENV implementation-specific configuration option. Minimum Virtual address increased to 54 bits. New PAL_MC_ERROR_INFO health indicator. New PAL_MC_ERROR_INJECT implementation-specific bit fields. MOV-to_SR.L reserved field checking. Added virtual machine disable. Added variable frequency mode additions to ACPI P-state description. Removed *pal_proc_vector* argument from PAL_VP_SAVE and PAL_VP_RESTORE. Added PAL_PROC_SET_FEATURES data speculation disable. Added Interruption Instruction Bundle registers. Min-state save area size change. PAL_MC_DYNAMIC_STATE changes. PAL_PROC_SET_FEATURES data poisoning promotion changes. ACPI P-state clarifications. Synchronization requirements for virtualization opcode optimization. New priority hint and multi-threading hint recommendations.

Date of Revision	Revision Number	Description
August 2005	2.2	Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 “Interruptions” and Section 11.2.2 PALE_RESET Exit State for details. Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10. Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of `ptc.l` instruction in Vol 3 for details. Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details. Introduced Unimplemented Instruction Address fault. Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details. Architecture extensions for processor virtualization. All instructions which must be last in an instruction group results in undefined behavior when this rule is violated. Added architectural sequence that guarantees increasing ITC and PMD values on successive reads. Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures. Allows IPI-redirection feature to be optional. Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block. Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details. “Bus parking” feature is now optional for PAL_BUS_GET_FEATURES. Introduced low-power synchronization primitive using `hint` instruction. FR32-127 is now preserved in PAL calling convention. New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges. Performance Monitor Data (PMD) registers are no longer sign-extended. New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details. Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure. Clarified PMU interrupts as edge-triggered. Modified ‘proc_number’ parameter in PAL_LOGICAL_TO_PHYSICAL procedure. Modified pal_copy_info alignment requirements. New bit in PAL_PROC_GET_FEATURES for variable P-state performance. Clarified descriptions for check_target_register and check_target_register_sof. Various fixes in dependency tables in Vol 3 Ch 5. Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5. Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3. Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.

Date of Revision	Revision Number	Description
August 2002	2.1	Added Predicate Behavior of `alloc` Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3). Added New `fc.i` Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3). Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2). Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2). Added Spontaneous NaT Generation on Speculative Load (`ld.s`) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3). Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2). Added Freeze Bit Functionality in Context Switching and Interrupt Generation Clarification (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Volume 2) Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2). Added capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Volume 2). Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2). Added Fault Handling Semantics for `lfetch.fault` Instruction (Section 2.2, Part I, Volume 3).
December 2001	2.0	Volume 1: Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1). IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3). Load instructions change (Section 4.4.1).

Date of Revision	Revision Number	Description
		Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Chapter 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4). Memory attribute transitions – added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation – changed behavior of DCR 'lc' bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2).
		Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3).

Date of Revision	Revision Number	Description
		Volume 2: Clarifications regarding “reserved” fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10). FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4). Clarification regarding ordering data dependency. Out-of-order IPI delivery is now allowed (Chapters 4 and 5). Content of EFLAG field changed in IIM (p. 9-24). PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11). PAL_CHECK processor state parameter changes (Chapter 11). PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11). PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11). PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11). PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11). PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11). Clarified memory ordering changes (Chapter 13). Clarification in dependence violation table (Appendix A).
		Volume 3: fmix instruction page figures corrected (Chapter 2). Clarification of “reserved” fields in ITIR (Chapters 2 and 3). Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/instruction group (Chapters 2 and 4). IA-32 JMPE instruction page typo fix (p. 5-238). Processor Serial Number feature removed (Chapter 5).
January 2000	1.0	Initial release of document.

§

Introduction to the Intel® Itanium® Architecture 2

The Itanium architecture was designed to overcome the performance limitations of traditional architectures and provide maximum headroom for the future. To achieve this, the Itanium architecture was designed with an array of innovative features to extract greater instruction level parallelism including speculation, predication, large register files, a register stack, advanced branch architecture, and many others. 64-bit memory addressability was added to meet the increasing large memory footprint requirements of data warehousing, e-business, and other high performance server applications. The Itanium architecture has an innovative floating-point architecture and other enhancements that support the high performance requirements of workstation applications such as digital content creation, design engineering, and scientific analysis.

The Itanium architecture also provides binary compatibility with the IA-32 instruction set. Processors based on the Itanium architecture can run IA-32 applications on an Itanium architecture-based operating system that supports execution of IA-32 applications. Such processors can run IA-32 application binaries on IA-32 legacy operating systems assuming the platform and firmware support exists in the system. The Itanium architecture also provides the capability to support mixed IA-32 and Itanium architecture-based code execution.

2.1 Operating Environments

The architectural model supports a mixture of IA-32 and Itanium architecture-based applications within a single Itanium architecture-based operating system. Table 2-1 defines the major supported operating environments.

Figure 2-1. System Environment

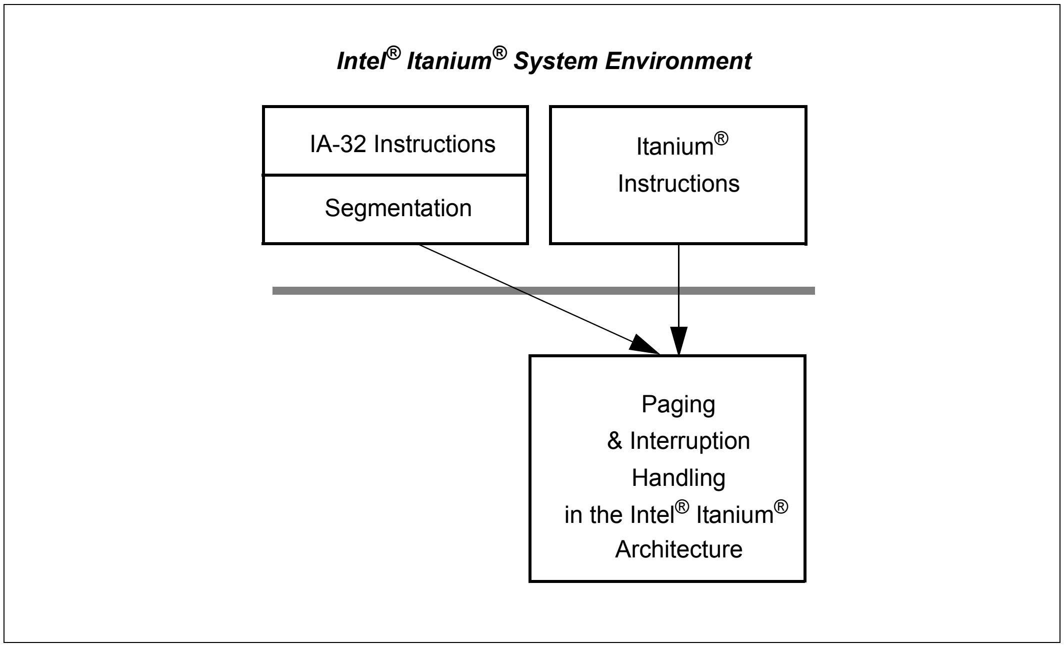

Intel® Itanium® System Environment
IA-32 Instructions
Segmentation
Itanium® Instructions
Paging & Interruption Handling in the Intel® Itanium® Architecture

Table 2-1. Major Operating Environments

System Environment	Application Environment	Usage
Itanium System Environment	IA-32 Protected Mode	IA-32 Protected Mode applications in the Intel® Itanium® System Environment.
	IA-32 Real Mode	IA-32 Real Mode applications in the Intel® Itanium® System Environment.
	IA-32 Virtual Mode	IA-32 Virtual 86 Mode applications in the Intel® Itanium® System Environment.
	Intel® Itanium® Instruction Set	Itanium architecture-based applications on Intel® Itanium architecture-based operating systems.

2.2 Instruction Set Transition Model Overview

Within the Itanium System Environment, the processor can execute either IA-32 or Itanium instructions at any time. Three special instructions and interruptions are defined to transition the processor between the IA-32 and the Itanium instruction set.

- `jmpe` (IA-32 instruction) Jump to an Itanium target instruction, and transition to the Itanium instruction set.
- `br.ia` (Itanium instruction) Branch to an IA-32 target instruction, and change the instruction set to IA-32.
- `rfi` (Itanium instruction) "Return from interruption" is defined to return to an IA-32 or Itanium instruction.
- Interrupts transition the processor to the Itanium instruction set for all interrupt conditions.

The `jmpe` and `br.ia` instructions provide a low overhead mechanism to transfer control between the instruction sets. These instructions are typically incorporated into "thunks" or "stubs" that implement the required call linkage and calling conventions to call dynamic or statically linked libraries. See Section 6.2.1, "Instruction Set Modes" for additional details.

2.3 Intel® Itanium® Instruction Set Features

Itanium architecture incorporates features which enable high sustained performance and remove barriers to further performance increases. The Itanium architecture is based on the following principles:

- Explicit parallelism
  - Mechanisms for synergy between the compiler and the processor
  - Massive resources to take advantage of instruction level parallelism
  - 128 integer and floating-point registers, 64 1-bit predicate registers, 8 branch registers
  - Support for many execution units and memory ports
- Features that enhance instruction level parallelism
  - Speculation (which minimizes memory latency impact).
  - Predication (which removes branches).
  - Software pipelining of loops with low overhead
  - Branch prediction to minimize the cost of branches
- Focused enhancements for improved software performance
  - Special support for software modularity
  - High performance floating-point architecture
  - Specific multimedia instructions

The following sections highlight these important features of the Itanium architecture.

2.4 Instruction Level Parallelism

Instruction Level Parallelism (ILP) is the ability to execute multiple instructions at the same time. The Itanium architecture allows issuing of independent instructions in bundles (three instructions per bundle) for parallel execution and can issue multiple bundles per clock. Supported by a large number of parallel resources such as large register files and multiple execution units, the Itanium architecture enables the compiler to manage work in progress and schedule simultaneous threads of computation.

The Itanium architecture incorporates mechanisms to take advantage of ILP. Compilers for traditional architectures are often limited in their ability to utilize speculative information because it cannot always be guaranteed to be correct. The Itanium architecture enables the compiler to exploit speculative information without sacrificing the correct execution of an application (see "Speculation" on page 1:16). In traditional architectures, procedure calls limit performance since registers need to be spilled and filled. The Itanium architecture enables procedures to communicate register usage to the processor. This allows the processor to schedule procedure register operations even when there is a low degree of ILP. See "Register Stack" on page 1:18.

2.5 Compiler to Processor Communication

The Itanium architecture provides mechanisms, such as instruction templates, branch hints, and cache hints to enable the compiler to communicate compile-time information to the processor. In addition, it allows compiled code to manage the processor hardware using runtime information. These communication mechanisms are vital in minimizing the performance penalties associated with branches and cache misses.

The cost of branches is minimized by permitting code to communicate branch information to the hardware in advance of the actual branch.

Every memory load and store in the Itanium architecture has a 2-bit cache hint field in which the compiler encodes its prediction of the spatial and/or temporal locality of the memory area being accessed. A processor based on the Itanium architecture can use this information to determine the placement of cache lines in the cache hierarchy to improve utilization. This is particularly important as the cost of cache misses is expected to increase.

2.6 Speculation

There are two types of speculation: control and data. In both control and data speculation, the compiler exposes ILP by issuing an operation early and removing the latency of this operation from critical path. The compiler will issue an operation speculatively if it is reasonably sure that the speculation will be beneficial. To be beneficial two conditions should hold: (1) it must be statistically frequent enough that the probability it will require recovery is small, and (2) issuing the operation early should expose further ILP-enhancing optimization. Speculation is one of the primary mechanisms for the compiler to exploit statistical ILP by overlapping, and therefore tolerating, the latencies of operations.

2.6.1 Control Speculation

Control speculation is the execution of an operation before the branch which guards it. Consider the code sequence below:

```
if (a>b) load(ld_addr1,target1)
else load(ld_addr2, target2)
```

If the operation `load(ld_addr1,target1)`were to be performed prior to the determination of `(a>b)`, then the operation would be control speculative with respect to the controlling condition `(a>b)`. Under normal execution, the operation `load(ld_addr1,target1)` may or may not execute. If the new control speculative load causes an exception, then the exception should only be serviced if `(a>b)` is true. When the compiler uses control speculation, it leaves a check operation at the original location. The check verifies whether an exception has occurred and if so it branches to recovery code. The code sequence above now translates into:

```
/* off critical path */
sload(ld_addr1,target1)
sload(ld_addr2,target2)

/* other operations including uses of target1/target2 */
if (a>b) scheck(target1,recovery_addr1)
else scheck(target2, recovery_addr2)
```

2.6.2 Data Speculation

Data speculation is the execution of a memory load prior to a store that preceded it and that may potentially alias with it. Data speculative loads are also referred to as "advanced loads." Consider the code sequence below:

```
store(st_addr,data)
load(ld_addr,target)
use(target)
```

The process of determining at compile time the relationship between memory addresses is called disambiguation. In the example above, if `ld_addr` and `st_addr` cannot be disambiguated, and if the load were to be performed prior to the store, then the load would be data speculative with respect to the store. If memory addresses overlap during execution, a data-speculative load issued before the store might return a different value than a regular load issued after the store. Therefore analogous to control speculation, when the compiler data speculates a load, it leaves a check instruction at the original location of the load. The check verifies whether an overlap has occurred and if so it branches to recovery code. The code sequence above now translates into:

```
/* off critical path */
aload(ld_addr,target)

/* other operations including uses of target */
store(st_addr,data)
acheck(target,recovery_addr)
use(target)
```

2.6.3 Predication

Predication is the conditional execution of instructions. Conditional execution is implemented through branches in traditional architectures. The Itanium architecture implements this function through the use of predicated instructions. Predication removes branches used for conditional execution resulting in larger basic blocks and the elimination of associated mispredict penalties.

To illustrate, an unpredicated instruction

```
r1 = r2 + r3
```

when predicated, would be of the form

```
if (p5) r1 = r2 + r3
```

In this example p5 is the controlling predicate that decides whether or not the instruction executes and updates state. If the predicate value is true, then the instruction updates state. Otherwise it generally behaves like a nop. Predicates are assigned values by compare instructions.

Predicated execution avoids branches, and simplifies compiler optimizations by converting a control dependency to a data dependency. Consider the original code:

```
if (a>b) c = c + 1
else d = d * e + f
```

The branch at (a>b) can be avoided by converting the code above to the predicated code:

```
pT, pF = compare(a>b)
if (pT) c = c + 1
if (pF) d = d * e + f
```

The predicate pT is set to 1 if the condition evaluates to true, and to 0 if the condition evaluates to false. The predicate pF is the complement of pT. The control dependency of the instructions c = c + 1 and d = d * e + f on the branch with the condition (a>b) is now converted into a data dependency on compare(a>b) through predicates pT and pF (the branch is eliminated). An added benefit is that the compiler can schedule the instructions under pT and pF to execute in parallel. It is also worth noting that there are several different types of compare instructions that write predicates in different manners including unconditional compares and parallel compares.

2.7 Register Stack

The Itanium architecture avoids the unnecessary spilling and filling of registers at procedure call and return interfaces through compiler-controlled renaming. At a call site, a new frame of registers is available to the called procedure without the need for register spill and fill (either by the caller or by the callee). Register access occurs by renaming the virtual register identifiers in the instructions through a base register into the physical registers. The callee can freely use available registers without having to spill and eventually restore the caller's registers. The callee executes an alloc instruction specifying the number of registers it expects to use in order to ensure that enough registers are available. If sufficient registers are not available (stack overflow), the alloc stalls the processor and spills the caller's registers until the requested number of registers are available.

At the return site, the base register is restored to the value that the caller was using to access registers prior to the call. Some of the caller's registers may have been spilled by the hardware and not yet restored. In this case (stack underflow), the return stalls the processor until the processor has restored an appropriate number of the caller's registers. The hardware can exploit the explicit register stack frame information to spill and fill registers from the register stack to memory at the best opportunity (independent of the calling and called procedures).

2.8 Branching

In addition to removing branches through the use of predication, several mechanisms are provided to decrease the branch misprediction rate and the cost of the remaining mispredicted branches. These mechanisms provide ways for the compiler to communicate information about branch conditions to the processor.

Branch predict instructions are provided which can be used to communicate an early indication of the target address and the location of the branch. The compiler will try to indicate whether a branch should be predicted dynamically or statically. The processor can use this information to initialize branch prediction structures, enabling good prediction even the first time a branch is encountered. This is beneficial for unconditional branches or in situations where the compiler has information about likely branch behavior.

For indirect branches, a branch register is used to hold the target address. Branch predict instructions provide an indication of which register will be used in situations when the target address can be computed early. A branch predict instruction can also signal that an indirect branch is a procedure return, enabling the efficient use of call/return stack prediction structures.

Special loop-closing branches are provided to accelerate counted loops and modulo-scheduled loops. These branches and their associated branch predict instructions provide information that allows for perfect prediction of loop termination, thereby eliminating costly mispredict penalties and a reduction of the loop overhead.

2.9 Register Rotation

Modulo scheduling of a loop is analogous to hardware pipelining of a functional unit since the next iteration of the loop starts before the previous iteration has finished. The iteration is split into stages similar to the stages of an execution pipeline. Modulo scheduling allows the compiler to execute loop iterations in parallel rather than sequentially. The concurrent execution of multiple iterations traditionally requires unrolling of the loop and software renaming of registers. The Itanium architecture allows the renaming of registers which provide every iteration with its own set of registers, avoiding the need for unrolling. This kind of register renaming is called register rotation. The result is that software pipelining can be applied to a much wider variety of loops – both small as well as large with significantly reduced overhead.

2.10 Floating-point Architecture

The Itanium architecture defines a floating-point architecture with full IEEE support for the single, double, and double-extended (80-bit) data types. Some extensions, such as a fused multiply and add operation, minimum and maximum functions, and a register file format with a larger range than the double-extended memory format, are also included. 128 floating-point registers are defined. Of these, 96 registers are rotating (not stacked) and can be used to modulo schedule loops compactly. Multiple floating-point status registers are provided for speculation.

The Itanium architecture has parallel FP instructions which operate on two 32-bit single precision numbers, resident in a single floating-point register, in parallel and independently. These instructions significantly increase the single precision floating-point computation throughput and enhance the performance of 3D intensive applications and games.

2.11 Multimedia Support

The Itanium architecture has multimedia instructions which treat the general registers as concatenations of eight 8-bit, four 16-bit, or two 32-bit elements. These instructions operate on each element in parallel, independent of the others. They are useful for creating high performance compression/decompression algorithms that are used by applications which have sound and video. Itanium multimedia instructions are semantically compatible with HP's MAX-2* multimedia technology and Intel's MMX and SSE technology instructions.

2.12 Intel® Itanium® System Architecture Features

2.12.1 Support for Multiple Address Space Operating Systems

Most contemporary commercial operating systems utilize a Multiple Address Space (MAS) model with the following characteristics:

Protection is enforced among processes by placing each process within a unique address space. Translation Lookaside Buffers (TLBs), which hold virtual to physical mappings, often need to be flushed on a process context switch.

Some memory areas may be shared among processes, e.g. kernel areas and shared libraries. Most operating systems assume at least one local and one global space.

To promote sharing of data between processes, MAS operating systems aggressively use virtual aliases to map physical memory locations into the address spaces of multiple processes. Virtual aliases create multiple TLB entries for the same physical data leading to reduced TLB efficiency.

The MAS model is supported by dividing the virtual address space into several regions. Region identifiers associated with each region are used to tag translations to a given address space. On a process switch, region identifiers uniquely identify the set of translations belonging to a process, thereby avoiding TLB flushes. Region identifiers also provide a unique intermediate virtual address that help avoid thrashing problems in virtual-indexed caches and TLBs. Regions provide efficient global/shared areas between processes, while reducing the occurrences of virtual aliasing.

2.12.2 Support for Single Address Space Operating Systems

A single address space (SAS) operating system style architecture is the basis for much of the current design work on future 64-bit operating systems. As operating systems (and other large, complex programs like databases) migrate from monolithic programs into cooperating subsystems, an SAS architecture becomes an important performance differentiation in future systems. The SAS or hybrid environments enable a more efficient use of hardware resources.

Common mechanisms are used in both SAS and MAS models such as page level access rights to enforce protection, although the reliance on the feature set will differ under each model. While most of the architected features are utilized in each model, protection keys exist to enable a single global address space operating environment.

2.12.3 System Performance and Scalability

Performance and scalability are achieved through a variety of features. Memory attributes, locking primitives, cache coherency, and memory ordering model work together to allow the efficient sharing of data in a multiprocessor environment. In addition, the Itanium architecture enables low latency fault, trap, and interrupt handlers along with light-weight domain crossings. Performance analysis is aided by the inclusion of several performance monitors, and mechanisms to support software profiling.

2.12.4 System Security and Supportability

Security and supportability result from a number of primitives which provide a very powerful runtime and debug environment. The protection model includes four protection rings and enables increased system integrity by offering a more sophisticated protection scheme than has generally been available. The machine check model allows detailed information to be provided describing the type of error involved and supports recovery for many types of errors. Several mechanisms are provided for debugging both system and application software.

2.13 Terminology

This following terms are used in the remainder of this document:

- **Itanium Instruction Set –** The Itanium architecture defines the 64-bit instruction set extensions to the IA-32 architecture.
- **IA-32 Architecture –** The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.
- **Itanium System Environment –** System environment that supports the execution of both IA-32 and Itanium architecture-based code.
- **Platform –** Application and operating system resources external to the processor such as: memory maps, external devices (e.g. DMA), keyboard controllers, buses (e.g. PCI), option cards, interrupt controllers, bridges, etc.
- **Itanium architecture-based Firmware –** The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).
- **Processor Abstraction Layer (PAL) –** The firmware layer which abstracts processor features that are implementation dependent.
- **System Abstraction Layer (SAL) –** The firmware layer which abstracts platform features that are implementation dependent.

§

Execution Environment 3

The architectural state consists of registers and memory. The results of instruction execution become architecturally visible according to a set of execution sequencing rules. This chapter describes the application architectural state and the rules for execution sequencing. See Chapter 6 for details on IA-32 instruction set execution.

3.1 Application Register State

The following is a list of the registers available to application programs (see Figure 3-1):

- **General Registers (GRs)** – General purpose 64-bit register file, GR0 - GR127. IA-32 integer and segment registers are contained in GR8 - GR31 when executing IA-32 instructions.
- **Floating-point Registers (FRs)** – Floating-point register file, FR0 - FR127. IA-32 floating-point and multi-media registers are contained in FR8 - FR31 when executing IA-32 instructions.
- **Predicate Registers (PRs)** – Single-bit registers, used in predication and branching, PR0 - PR63.
- **Branch Registers (BRs)** – Registers used in branching, BR0 - BR7.
- **Instruction Pointer (IP)** – Register which holds the bundle address of the currently executing instruction, or byte address of the currently executing IA-32 instruction.
- **Current Frame Marker (CFM)** – State that describes the current general register stack frame, and FR/PR rotation.
- **Application Registers (ARs)** – A collection of special-purpose registers.
- **Performance Monitor Data Registers (PMD)** – Data registers for performance monitor hardware.
- **User Mask (UM)** – A set of single-bit values used for alignment traps, performance monitors, and to monitor floating-point register usage.
- **Processor Identifiers (CPUID)** – Registers that describe processor implementation-dependent features.

IA-32 application register state is entirely contained within the larger Itanium application register set and is accessible by Itanium instructions. IA-32 instructions cannot access the Itanium register set. See Section 6.2.2, "IA-32 Application Register State Model" for details on IA-32 register assignments.

3.1.1 Reserved and Ignored Registers and Fields

Registers which are not defined are either reserved or ignored. An access to a **reserved register** raises an Illegal Operation fault. A read of an **ignored register** returns zero. Software may write any value to an ignored register and the hardware will ignore the value written. In variable-sized register sets, registers which are unimplemented in a particular processor are also reserved registers. An access to one of these unimplemented registers causes a Reserved Register/Field fault.

Within defined registers, fields which are not defined are either reserved or ignored. For **reserved fields**, hardware will always return a zero on a read. Software must always write zeros to these fields. Any attempt to write a non-zero value into a reserved field will raise a Reserved Register/Field fault. **Reserved** fields may have a possible future use.

For **ignored fields**, hardware will return a 0 on a read, unless noted otherwise. Software may write any value to these fields since the hardware will ignore any value written. Except where noted otherwise some IA-32 ignored fields may have a possible future use.

Table 3-1 summarizes how the processor treats reserved and ignored registers and fields.

Table 3-1. Reserved and Ignored Registers and Fields

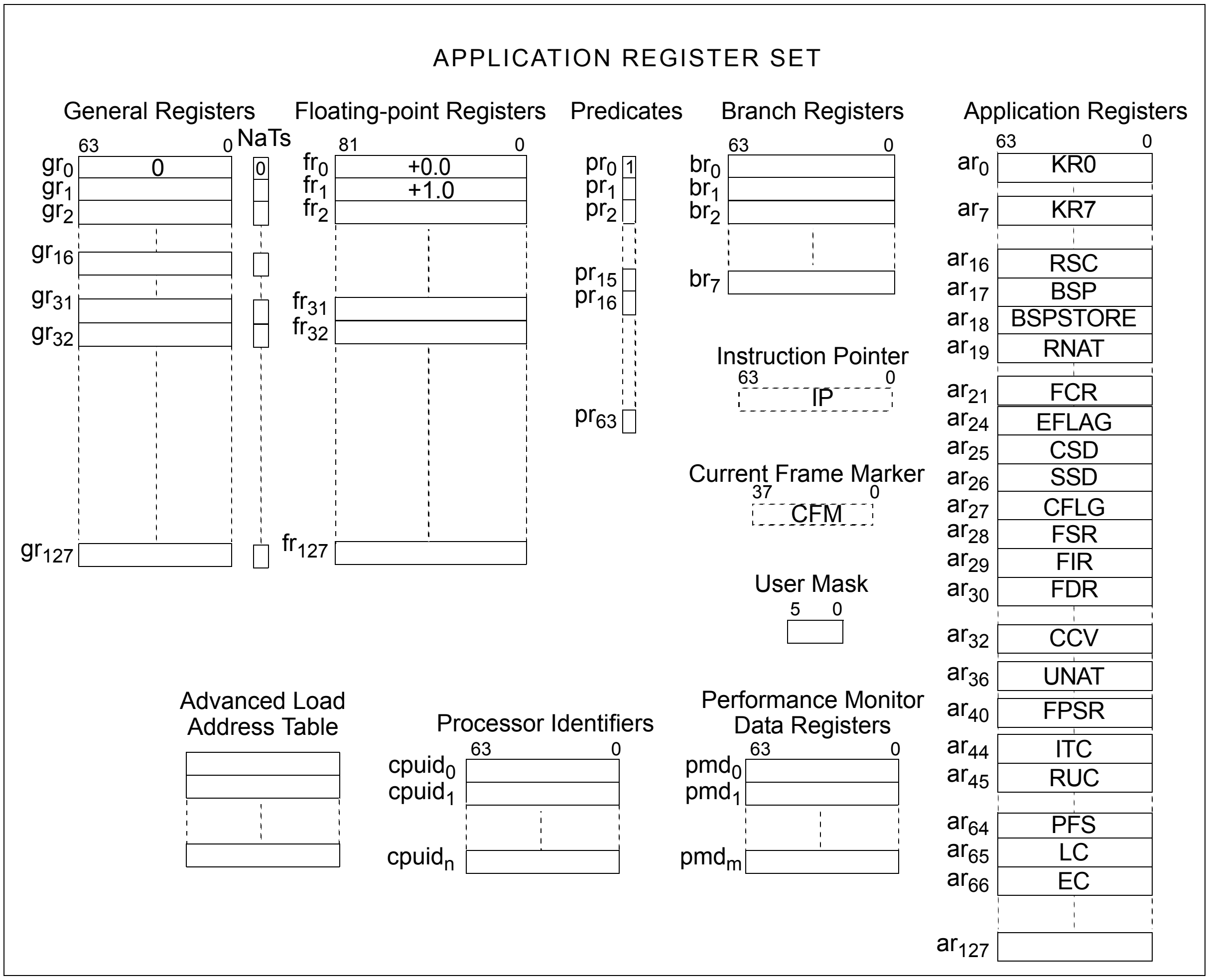
Type	Read	Write
Reserved register	Illegal Operation fault	Illegal Operation fault
Ignored register	0	Value written is discarded
Reserved field	0	Write of non-zero causes Reserved Reg/Field fault
Ignored field	0 (unless noted otherwise)	Value written is discarded

For defined fields in registers, values which are not defined are reserved. Software must always write defined values to these fields. Any attempt to write a **reserved value** will raise a Reserved Register/Field fault. Certain registers are **read-only registers**. A write to a read-only register raises an Illegal Operation fault.

When fields are marked as **reserved**, it is essential for compatibility with future processors that software treat these fields as having a future, though unknown effect. Software should follow these guidelines when dealing with **reserved** fields:

- Do not depend on the state of any reserved fields. Mask all reserved fields before testing.
- Do not depend on the state of any reserved fields when storing to memory or a register.
- Do not depend on the ability to retain information written into reserved or ignored fields.
- Where possible reload reserved or ignored fields with values previously returned from the same register, otherwise load zeros.

Figure 3-1. Application Register Model



3.1.2 General Registers

A set of 128 (64-bit) **general registers** provide the central resource for all integer and integer multimedia computation. They are numbered GR0 through GR127, and are available to all programs at all privilege levels. Each general register has 64 bits of normal data storage plus an additional bit, the **NaT** bit (Not a Thing), which is used to track deferred speculative exceptions.

The general registers are partitioned into two subsets. General registers 0 through 31 are termed the **static general registers**. Of these, GR0 is special in that it always reads as zero when sourced as an operand, and attempting to write to GR 0 causes an Illegal Operation fault. General registers 32 through 127 are termed the **stacked general registers**. The stacked registers are made available to a program by allocating a register stack frame consisting of a programmable number of local and output registers. See "Register Stack" on page 1:47 for a description. A portion of the stacked registers can be programmatically renamed to accelerate loops. See "Modulo-scheduled Loop Support" on page 1:75.

General registers 8 through 31 contain the IA-32 integer, segment selector and segment descriptor registers. See "IA-32 General Purpose Registers" on page 1:117 for details on IA-32 register assignments.

3.1.3 Floating-point Registers

A set of 128 (82-bit) **floating-point registers** are used for all floating-point computation. They are numbered FR0 through FR127, and are available to all programs at all privilege levels. The floating-point registers are partitioned into two subsets. Floating-point registers 0 through 31 are termed the **static floating-point registers**. Of these, FR0 and FR1 are special. FR0 always reads as +0.0 when sourced as an operand, and FR 1 always reads as +1.0. When either of these is used as a destination, a fault is raised. Deferred speculative exceptions are recorded with a special register value called **NaTVal (Not a Thing Value)**.

Floating-point registers 32 through 127 are termed the **rotating floating-point registers**. These registers can be programmatically renamed to accelerate loops. See "Modulo-scheduled Loop Support" on page 1:75.

Floating-point registers 8 through 31 contain the IA-32 floating-point and multi-media registers when executing IA-32 instructions. For details, see "IA-32 Floating-point Registers" on page 1:124.

3.1.4 Predicate Registers

A set of 64 (1-bit) **predicate registers** are used to hold the results of compare instructions. These registers are numbered PR0 through PR63, and are available to all programs at all privilege levels. These registers are used for conditional execution of instructions.

The predicate registers are partitioned into two subsets. Predicate registers 0 through 15 are termed the **static predicate registers**. Of these, PR0 always reads as '1' when sourced as an operand, and when used as a destination, the result is discarded. The static predicate registers are also used in conditional branching. See "Predication" on page 1:54.

Predicate registers 16 through 63 are termed the **rotating predicate registers**. These registers can be programmatically renamed to accelerate loops. See "Modulo-scheduled Loop Support" on page 1:75.

3.1.5 Branch Registers

A set of 8 (64-bit) **branch registers** are used to hold branching information. They are numbered BR 0 through BR 7, and are available to all programs at all privilege levels. The branch registers are used to specify the branch target addresses for indirect branches. For more information see "Branch Instructions" on page 1:74.

3.1.6 Instruction Pointer

The Instruction Pointer (IP) holds the address of the bundle which contains the current executing instruction. The IP can be read directly with a mov ip instruction. The IP cannot be directly written, but is incremented as instructions are executed, and can be set to a new value with a branch. Because instruction bundles are 16 bytes, and are 16-byte aligned, the least significant 4 bits of IP are always zero. See "Instruction Encoding Overview" on page 1:38. For IA-32 instruction set execution, IP holds the zero extended 32-bit virtual linear address of the currently executing IA-32 instruction. IA-32 instructions are byte-aligned, therefore the least significant 4 bits of IP are preserved for IA-32 instruction set execution. See "IA-32 Instruction Pointer" on page 1:117 for IA-32 instruction set execution details.

3.1.7 Current Frame Marker

Each general register stack frame is associated with a frame marker. The frame marker describes the state of the general register stack. The Current Frame Marker (CFM) holds the state of the current stack frame. The CFM cannot be directly read or written (see "Register Stack" on page 1:47).

The frame markers contain the sizes of the various portions of the stack frame, plus three Register Rename Base values (used in register rotation). The layout of the frame markers is shown in Figure 3-2 and the fields are described in Table 3-2.

On a call, the CFM is copied to the Previous Frame Marker field in the Previous Function State register (see Section 3.1.8.12, "Previous Function State (PFS – AR 64)"). A new value is written to the CFM, creating a new stack frame with no locals or rotating registers, but with a set of output registers which are the caller's output registers. Additionally, all Register Rename Base registers (RRBs) are set to 0. See "Modulo-scheduled Loop Support" on page 1:75.

Figure 3-2. Frame Marker Format

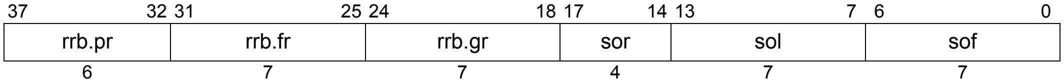


Table 3-2. Frame Marker Field Description

Field	Bits	Description
sof	6:0	Size of stack frame
sol	13:7	Size of locals portion of stack frame
sor	17:14	Size of rotating portion of stack frame (the number of rotating registers is 8 * sor)
rrb.gr	24:18	Register Rename Base for general registers
rrb.fr	31:25	Register Rename Base for floating-point registers
rrb.pr	37:32	Register Rename Base for predicate registers

3.1.8 Application Registers

The application register file includes special-purpose data registers and control registers for application-visible processor functions for both the IA-32 and Itanium instruction set architectures. These registers can be accessed by Itanium architecture-based applications (except where noted). Table 3-3 contains a list of the application registers.

Table 3-3. Application Registers

Register	Name	Description	Execution Unit Type
AR 0-7	KR 0-7[a]	Kernel Registers 0-7	M
AR 8-15		Reserved	
AR 16	RSC	Register Stack Configuration Register	
AR 17	BSP	Backing Store Pointer (read-only)	
AR 18	BSPSTORE	Backing Store Pointer for Memory Stores	
AR 19	RNAT	RSE NaT Collection Register	
AR 20		Reserved	
AR 21	FCR	IA-32 Floating-point Control Register	
AR 22 - AR 23		Reserved	
AR 24	EFLAG[b]	IA-32 EFLAG register	
AR 25	CSD	IA-32 Code Segment Descriptor / Compare and Store Data register	
AR 26	SSD	IA-32 Stack Segment Descriptor	
AR 27	CFLG[a]	IA-32 Combined CR0 and CR4 register	
AR 28	FSR	IA-32 Floating-point Status Register	
AR 29	FIR	IA-32 Floating-point Instruction Register	
AR 30	FDR	IA-32 Floating-point Data Register	
AR 31		Reserved	
AR 32	CCV	Compare and Exchange Compare Value Register	
AR 33 - AR 35		Reserved	
AR 36	UNAT	User NaT Collection Register	
AR 37 - AR 39		Reserved	
AR 40	FPSR	Floating-point Status Register	
AR 41 - AR 43		Reserved	
AR 44	ITC	Interval Time Counter	
AR 45	RUC	Resource Utilization Counter	
AR 46 - AR 47		Reserved	
AR 48 - AR 63		Ignored	M or I
AR 64	PFS	Previous Function State	I
AR 65	LC	Loop Count Register	
AR 66	EC	Epilog Count Register	
AR 67 - AR 111		Reserved	
AR 112 - AR 127		Ignored	M or I

[a] Writes to these registers when the privilege level is not zero result in a Privileged Register fault. Reads are always allowed.

[b] Some IA-32 EFLAG field writes are silently ignored if the privilege level is not zero. See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:243 for details.

Application registers can only be accessed by either a M or I execution unit. This is specified in the last column of the table. The ignored registers are for future backward-compatible extensions.

See Section 10.2, “System Register Model” on page 2:239 for the field definition of each IA-32 application register.

3.1.8.1 Kernel Registers (KR 0-7 – AR 0-7)

Eight user-visible 64-bit data kernel registers are provided to convey information from the operating system to the application. These registers can be read at any privilege level but are writable only at the most privileged level. KR0 - KR2 are also used to hold additional IA-32 register state when the IA-32 instruction set is executing. See Section 10.1, “Instruction Set Transitions” on page 2:239 for register details when calling IA-32 code.

3.1.8.2 Register Stack Configuration Register (RSC – AR 16)

The Register Stack Configuration (RSC) Register is a 64-bit register used to control the operation of the Register Stack Engine (RSE). Refer to Chapter 6, “Register Stack Engine” in Volume 2 for details. The RSC format is shown in Figure 3-3 and the field description is contained in Table 3-4. Instructions that modify the RSC can never set the privilege level field to a more privileged level than the currently executing process.

Figure 3-3. RSC Format

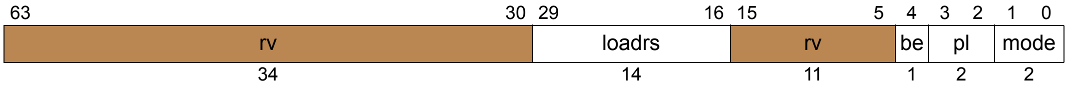


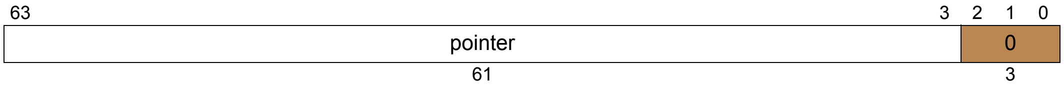
Table 3-4. RSC Field Description

<table>
<tr><th>Field</th><th>Bits</th><th colspan="4">Description</th></tr>
<tr><td rowspan="6">mode</td><td rowspan="6">1:0</td><td colspan="4">RSE mode – controls how aggressively the RSE saves and restores register frames. Eager and intensive settings are hints and can be implemented as lazy.</td></tr>
<tr><td>Bit Pattern</td><td>RSE Mode</td><td>Bit 1:
eager loads</td><td>Bit 0:
eager stores</td></tr>
<tr><td>00</td><td>enforced lazy</td><td>disabled</td><td>disabled</td></tr>
<tr><td>10</td><td>load intensive</td><td>enabled</td><td>disabled</td></tr>
<tr><td>01</td><td>store intensive</td><td>disabled</td><td>enabled</td></tr>
<tr><td>11</td><td>eager</td><td>enabled</td><td>enabled</td></tr>
<tr><td>pl</td><td>3:2</td><td colspan="4">RSE privilege level – loads and stores issued by the RSE are at this privilege level</td></tr>
<tr><td>be</td><td>4</td><td colspan="4">RSE endian mode – loads and stores issued by the RSE use this byte ordering (0: little endian; 1: big endian)</td></tr>
<tr><td>loadrs</td><td>29:16</td><td colspan="4">RSE load distance to tear point – value used in the `loadrs` instruction for synchronizing the RSE to a tear point</td></tr>
<tr><td>rv</td><td>15:5, 63:30</td><td colspan="4">Reserved</td></tr>
</table>

3.1.8.3 RSE Backing Store Pointer (BSP – AR 17)

The RSE Backing Store Pointer is a 64-bit read-only register (Figure 3-4). It holds the address of the location in memory which is the save location for GR 32 in the current stack frame. See Section 6.1, “RSE and Backing Store Overview” on page 2:133.

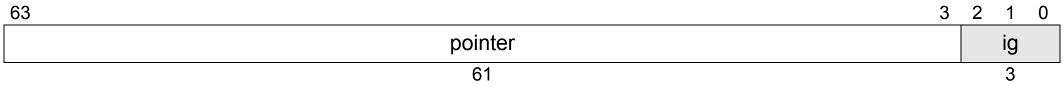
Figure 3-4. BSP Register Format



3.1.8.4 RSE Backing Store Pointer for Memory Stores (BSPSTORE – AR 18)

The RSE Backing Store Pointer for memory stores is a 64-bit register (Figure 3-5). It holds the address of the location in memory to which the RSE will spill the next value. See Section 6.1, "RSE and Backing Store Overview" on page 2:133.

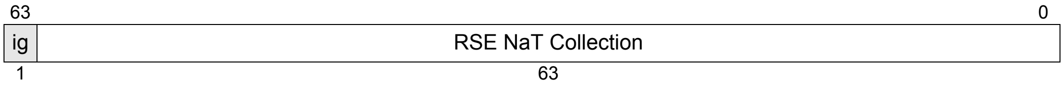
Figure 3-5. BSPSTORE Register Format



3.1.8.5 RSE NaT Collection Register (RNAT – AR 19)

The RSE NaT Collection Register is a 64-bit register (Figure 3-6) used by the RSE to temporarily hold NaT bits when it is spilling general registers. Bit 63 always reads as zero and ignores all writes. See Section 6.1, "RSE and Backing Store Overview" on page 2:133.

Figure 3-6. RNAT Register Format



3.1.8.6 Compare and Store Data register (CSD – AR 25)

The Compare and Store Data register is a 64-bit register that provides data to be stored by the Itanium `st16` and `cmp8xchg16` instructions, and receives data loaded by the Itanium `ld16` instruction.

For implementations that do not support the `ld16`, `st16` and `cmp8xchg16` instructions, bits 61:60 may be optionally implemented. This means that on move application register instructions the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads. For implementations that do support the `ld16`, `st16` and `cmp8xchg16` instructions, all bits of CSD are implemented.

For IA-32 execution, this register is the IA-32 Code Segment Descriptor. See Section 6.2.2.3, "IA-32 Segment Registers" on page 1:118.

3.1.8.7 Compare and Exchange Value Register (CCV – AR 32)

The Compare and Exchange Value Register is a 64-bit register that contains the compare value used as the third source operand in the Itanium `cmpxchg` instruction.

3.1.8.8 User NaT Collection Register (UNAT – AR 36)

The User NaT Collection Register is a 64-bit register used to temporarily hold NaT bits when saving and restoring general registers with the `ld8.fill` and `st8.spill` instructions.

3.1.8.9 Floating-point Status Register (FPSR – AR 40)

The floating-point status register (FPSR) controls traps, rounding mode, precision control, flags, and other control bits for Itanium floating-point instructions. FPSR does not control or reflect the status of IA-32 floating-point instructions. For more details on the FPSR, see "Floating-point Status Register" on page 1:88.

3.1.8.10 Interval Time Counter (ITC – AR 44)

The Interval Time Counter (ITC) is a 64-bit register which counts up at a fixed relationship to the input clock to the processor. The ITC may be clocked at a somewhat lower frequency than the instruction execution frequency. This clocking relationship is described in the PAL procedure PAL_FREQ_RATIOS on page 2:392. The ITC is guaranteed to be clocked at a constant rate, even if the instruction execution frequency may vary.

A sequence of reads of the ITC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads. Applications can directly sample the ITC for time-based calculations.

System software can secure the interval time counter from non-privileged access. When secured, a read of the ITC at any privilege level other than the most privileged causes a Privileged Register fault. The ITC can be written only at the most privileged level. The IA-32 Time Stamp Counter (TSC) is similar to ITC counter. ITC can directly be read by the IA-32 `rdtsc` (read time stamp counter) instruction. System software can secure the ITC from non-privileged IA-32 access. When secured, an IA-32 read of the ITC at any privilege level other than the most privileged raises an IA_32_Exception(GPfault).

3.1.8.11 Resource Utilization Counter (RUC – AR 45)

The Resource Utilization Counter (RUC) is a 64-bit register which counts up at a fixed relationship to the input clock to the processor, when the processor is active. RUC provides an estimate of the portion of resources used by a logical processor with respect to all resources provided by the underlying physical processor.

The Resource Utilization Counter (RUC) is a 64-bit register which provides an estimate of the portion of resources used by a logical processor with respect to all resources provided by the underlying physical processor.

In a given time interval, the difference in the RUC values for all of the logical processors on a given physical processor add up to the difference seen in the ITC on that physical processor for that same interval.

A sequence of reads of the RUC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads.

System software can secure the resource utilization counter from non-privileged access. When secured, a read of the RUC at any privilege level other than the most privileged causes a Privileged Register fault.

The RUC for a logical processor does not count when that logical processor is in LIGHT_HALT, unless all logical processors on a given physical processor are in LIGHT_HALT, in which case the last logical on a given physical processor to enter LIGHT_HALT has its RUC continue to count.

With processor virtualization, the RUC can be used to communicate the portion of resources used by a virtual processor. See Section 3.4, "Processor Virtualization" on page 2:44 and Section 11.7, "PAL Virtualization Support" on page 2:324 for details on virtual processors.

The RUC register is not supported on all processor implementations. Software can check CPUID register 4 to determine the availability of this feature. The RUC register is reserved when this feature is not supported.

3.1.8.12 Previous Function State (PFS – AR 64)

The Previous Function State register (PFS) contains multiple fields: Previous Frame Marker (pfm), Previous Epilog Count (pec), and Previous Privilege Level (ppl). Figure 3-7 diagrams the PFS format and Table 3-5 describes the PFS fields. These values are copied automatically on a call from the CFM register, Epilog Count Register (EC) and PSR.cpl (Current Privilege Level in the Processor Status Register) to accelerate procedure calling.

When a `br.call` or `brl.call` is executed, the CFM, EC, and PSR.cpl are copied to the PFS and the old contents of the PFS are discarded. When a `br.ret` is executed, the PFS is copied to the CFM and EC. PFS.ppl is copied to PSR.cpl, unless this action would increase the privilege level. For more details on the PSR see Chapter 3, "System State and Programming Model" in Volume 2.

The PFS.pfm has the same layout as the CFM (see Section 3.1.7, "Current Frame Marker"), and the PFS.pec has the same layout as the EC (see Section 3.1.8.14, "Epilog Count Register (EC – AR 66)").

Figure 3-7. PFS Format

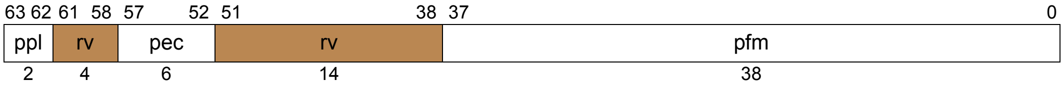


Table 3-5. PFS Field Description

Field	Bits	Description
pfm	37:0	Previous Frame Marker
pec	57:52	Previous Epilog Count
ppl	63:62	Previous Privilege Level
rv	51:38, 61:58	Reserved

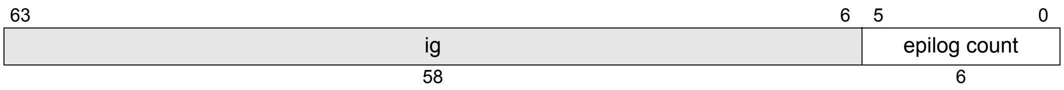
3.1.8.13 Loop Count Register (LC – AR 65)

The Loop Count register (LC) is a 64-bit register used in counted loops. LC is decremented by counted-loop-type branches.

3.1.8.14 Epilog Count Register (EC – AR 66)

The Epilog Count register (EC) is a 6-bit register used for counting the final (epilog) stages in modulo-scheduled loops. See "Modulo-scheduled Loop Support" on page 1:75. A diagram of the EC register is shown in Figure 3-8.

Figure 3-8. Epilog Count Register Format



3.1.9 Performance Monitor Data Registers (PMD)

A set of performance monitoring registers can be configured by privileged software to be accessible at all privilege levels. Performance monitor data can be directly sampled from within the application. The operating system is allowed to secure user-configured performance monitors. Secured performance counters return zeros when read, regardless of the current privilege level. The performance monitors can only be written at the most privileged level. Refer to Chapter 7, "Debugging and Performance Monitoring" in Volume 2 for details. Performance monitors can be used to gather performance information for the execution of both IA-32 and Itanium instruction sets.

3.1.10 User Mask (UM)

The user mask is a subset of the Processor Status Register and is accessible to application programs. The user mask controls memory access alignment, byte-ordering and user-configured performance monitors. It also records the modification state of floating-point registers. Figure 3-9 show the user mask format and Table 3-6 describes the user mask fields. For more details on the PSR refer to "Processor Status Register (PSR)" on page 2:23.

Figure 3-9. User Mask Format

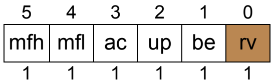


Table 3-6. User Mask Field Descriptions

Field	Bit	Description
rv	0	Reserved
be	1	Big-endian memory access enable (controls loads and stores but not RSE memory accesses) 0: accesses are done little-endian 1: accesses are done big-endian This bit is ignored for IA-32 data memory accesses. IA-32 data references are always performed little-endian.

Table 3-6. User Mask Field Descriptions (Continued)

Field	Bit	Description
up	2	User performance monitor enable (including IA-32) 0: user performance monitors are disabled 1: user performance monitors are enabled
ac	3	Alignment check for data memory references (including IA-32) 0: unaligned data memory references may cause an Unaligned Data Reference fault. 1: all unaligned data memory references cause an Unaligned Data Reference fault.
mfl	4	Lower (f2.. f31) floating-point registers written – This bit is set to one when an Intel® Itanium® instruction that uses register f2..f31 as a target register, completes. This bit is sticky and is only cleared by an explicit write of the user mask. See Section 3.3.2, “Processor Status Register (PSR)” for conditions when IA-32 instructions set this bit.
mfh	5	Upper (f32.. f127) floating-point registers written – This bit is set to one when an Intel® Itanium® instruction that uses register f32..f127 as a target register, completes. This bit is sticky and only cleared by an explicit write of the user mask. See Section 3.3.2, “Processor Status Register (PSR)” for conditions when IA-32 instructions set this bit.

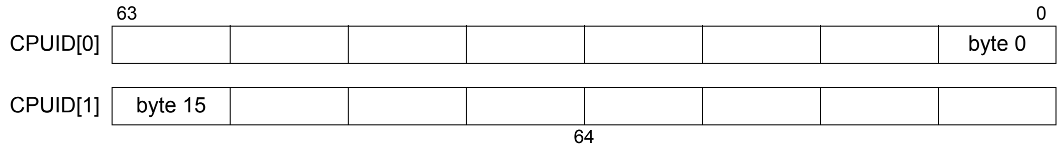
3.1.11 Processor Identification Registers

Application level processor identification information is available in a register file termed: CPUID. This register file is divided into a fixed region, registers 0 to 4, and a variable region, register 5 and above. The CPUID[3].number field indicates the maximum number of 8-byte registers containing processor specific information.

The CPUID registers are unprivileged and accessed using the indirect `mov` (from) instruction. All registers beyond register CPUID[3].number are reserved and raise a Reserved Register/Field fault if they are accessed. Writes are not permitted and no instruction exists for such an operation.

Vendor information is located in CPUID registers 0 and 1 and specify a vendor name, in ASCII, for the processor implementation (Figure 3-10). All bytes after the end of the string up to the 16th byte are zero. Earlier ASCII characters are placed in lower number register and lower numbered byte positions.

Figure 3-10. CPUID Registers 0 and 1 – Vendor Information



CPUID register 2 is an ignored register (reads from this register return zero).

CPUID register 3 contains several fields indicating version information related to the processor implementation. Figure 3-11 and Table 3-7 specify the definitions of each field.

Figure 3-11. CPUID Register 3 – Version Information

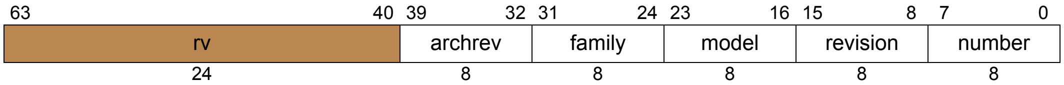


Table 3-7. CPUID Register 3 Fields

Field	Bits	Description
number	7:0	The index of the largest implemented CPUID register (one less than the number of implemented CPUID registers). This value will be at least 4.
revision	15:8	Processor revision number. An 8-bit value that represents the revision or stepping of this processor implementation within the processor model.
model	23:16	Processor model number. A unique 8-bit value representing the processor model within the processor family.
family	31:24	Processor family number. A unique 8-bit value representing the processor family.
archrev	39:32	Architecture revision. An 8-bit value that represents the architecture revision number that the processor implements.
rv	63:40	Reserved.

CPUID register 4 provides general application-level information about processor features. As shown in Figure 3-12, it is a set of flag bits used to indicate if a given feature is supported in the processor model. When a bit is one the feature is supported; when 0 the feature is not supported. The defined feature bits in the current architecture are listed in Table 3-8. As new features are added (or removed) from future processor models the presence (or removal) of new features will be indicated by new feature bits.

CPUID register 4 is logically split into two halves, both of which contain general feature and capability information but which have different usage models and access capabilities; this information reflects the status of any enabled or disabled features. Both the upper and lower halves of CPUID register 4 are accessible through the move indirect register instruction; depending on the implementation, the latency for this access can be long and this access method is not appropriate for low-latency code versioning using self-selection. In addition, the upper half of CPUID register 4 is also accessible using the test feature instruction; the latency for this access is comparable to that of the test bit instruction and this access method enables low-latency code versioning using self selection.

This register does not contain IA-32 instruction set features. IA-32 instruction set features can be acquired by the IA-32 cpuid instruction.

Figure 3-12. CPUID Register 4 – General Features/Capability Bits

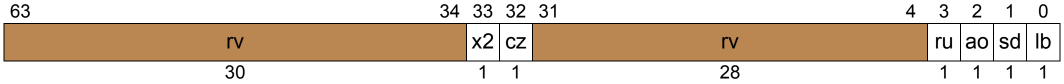


Table 3-8. CPUID Register 4 Fields

Field	Bits	Description
lb	0	Processor implements the long branch (`brl`) instructions.
sd	1	Processor implements spontaneous deferral (see Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:105).
ao	2	Processor implements 16-byte atomic operations (see “ld — Load”, “st — Store” and “cmpxchg — Compare and Exchange” instructions in Volume 3).
ru	3	Processor implements the Resource Utilization Counter (AR 45).
rv	31:4	Reserved.
cz	32	Processor implements the `clz` instruction (see “tf — Test Feature” instruction in Volume 3).

Table 3-8. CPUID Register 4 Fields (Continued)

Field	Bits	Description
x2	33	Processor implements `mpy4` and `mpyshl4` instructions (see "tf — Test Feature" instruction in Volume 3).
rv	63:34	Reserved.

3.2 Memory

This section describes an Itanium architecture-based application program's view of memory. This includes a description of how memory is accessed, for both 32-bit and 64-bit applications. The size and alignment of addressable units in memory is also given, along with a description of how byte ordering is handled.

The system view of memory and of virtual memory management is given in Chapter 4, "Addressing and Protection" in Volume 2 . The IA-32 instruction set view of memory and virtual memory management is defined in Section 10.6, "System Memory Model" on page 2:259.

3.2.1 Application Memory Addressing Model

Memory is byte addressable and is accessed with 64-bit pointers. A 32-bit pointer model without a hardware mode is supported architecturally. Pointers which are 32 bits in memory are loaded and manipulated in 64-bit registers. Software must explicitly convert 32-bit pointers into 64-bit pointers before use. For details on 32-bit addressing, refer to "32-bit Virtual Addressing" on page 2:71.

3.2.2 Addressable Units and Alignment

Memory can be addressed in units of 1, 2, 4, 8, 10 and 16 bytes.

It is recommended that all addressable units be stored on their naturally aligned boundaries. Hardware and/or operating system software may have support for unaligned accesses, possibly with some performance cost. 10-byte floating-point values should be stored on 16-byte aligned boundaries.

Bits within larger units are always numbered from 0 starting with the least-significant bit. Quantities loaded from memory to general registers are always placed in the least-significant portion of the register (loaded values are placed right justified in the target general register).

Instruction bundles (three instructions per bundle) are 16-byte units that are always aligned on 16-byte boundaries.

3.2.3 Byte Ordering

The UM.be bit in the User Mask controls whether loads and stores use little-endian or big-endian byte ordering for Itanium architecture-based code. When the UM.be bit is 0, larger-than-byte loads and stores are little endian (lower-addressed bytes in memory correspond to the lower-order bytes in the register). When the UM.be bit is 1, larger-than-byte loads and stores are big endian (lower-addressed bytes in memory correspond to the higher-order bytes in the register). Load byte and store byte are not affected by the UM.be bit. The UM.be bit does not affect instruction fetch, IA-32 references, or the RSE. Instructions are always accessed by the processor as little-endian units. When instructions are referenced as big-endian data, the instruction will appear reversed in a register.

Figure 3-13 shows various loads in little-endian format. Figure 3-14 shows various loads in big endian format. Stores are not shown but behave similarly.

Figure 3-13. Little-endian Loads

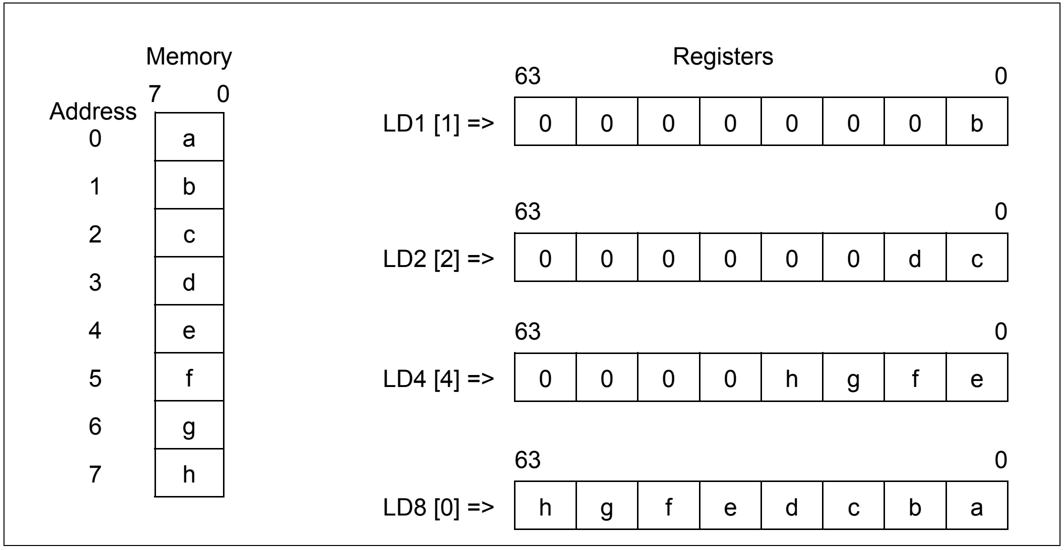
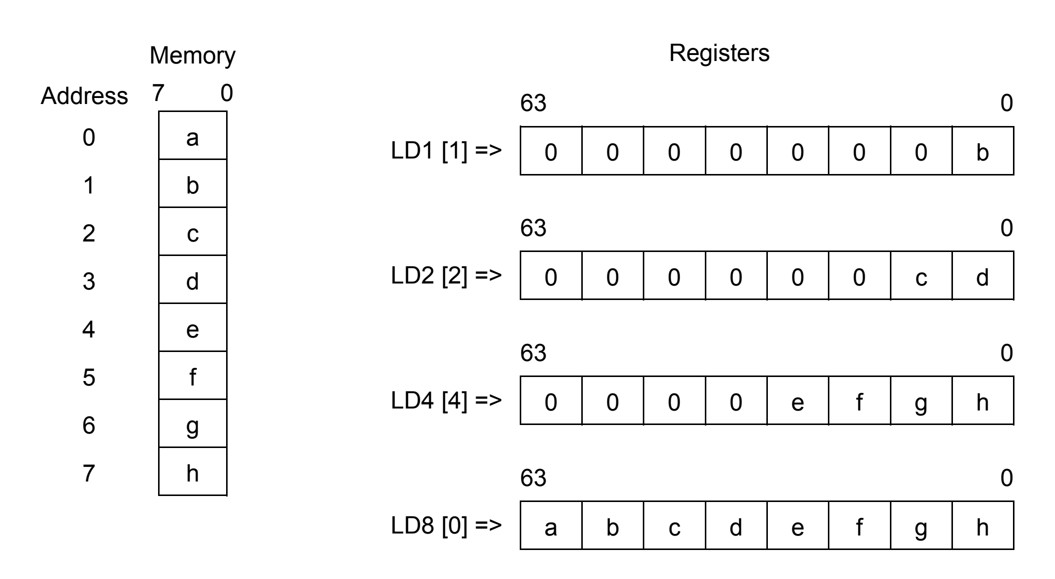

Memory
Address
7 0
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
Registers
63 0
LD1 [1] => 0 0 0 0 0 0 0 b
63 0
LD2 [2] => 0 0 0 0 0 0 d c
63 0
LD4 [4] => 0 0 0 0 h g f e
63 0
LD8 [0] => h g f e d c b a

Figure 3-14. Big-endian Loads


Memory
Address
7 0
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
Registers
63 0
LD1 [1] => 0 0 0 0 0 0 0 b
63 0
LD2 [2] => 0 0 0 0 0 0 c d
63 0
LD4 [4] => 0 0 0 0 e f g h
63 0
LD8 [0] => a b c d e f g h

3.3 Instruction Encoding Overview

Each instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 3-9 lists the instruction types and the execution unit type on which they are executed.

Table 3-9. Relationship between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 3-15.

Figure 3-15. Bundle Format

127 87	86 46	45 5	4 0
instruction slot 2	instruction slot 1	instruction slot 0	template
41	41	41	5

During execution, architectural **stops** in the program indicate to the hardware that one or more instructions before the stop may have certain kinds of resource dependencies with one or more instructions after the stop. A stop is present after each slot having a double line to the right of it in Table 3-10. For example, template 00 has no stops, while template 03 has a stop after slot 1 and another after slot 2.

In addition to the location of stops, the template field specifies the mapping of instruction slots to execution unit types. Not all possible mappings of instructions to units are available. Table 3-10 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle. Listed within each column is the execution unit type controlled by that instruction slot.

Table 3-10. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit ‖
02	M-unit	I-unit ‖	I-unit
03	M-unit	I-unit ‖	I-unit ‖
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a] ‖
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit ‖
0A	M-unit ‖	M-unit	I-unit

Table 3-10. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
0B	M-unit ‖	M-unit	I-unit ‖
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit ‖
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit ‖
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit ‖
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit ‖
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit ‖
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit ‖
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit ‖
1E			
1F			

a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode break.i and nop.i in addition to any X-unit instruction.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

3.4 Instruction Sequencing Considerations

Itanium architecture-based code consists of a sequence of instructions and stops packed in bundles. Instruction execution is ordered as follows:

- Bundles are ordered from lowest to highest memory address. Instructions in bundles with lower memory addresses are considered to precede instructions in bundles with higher memory addresses. The byte order of each bundle in memory is little-endian (the template field is contained in byte 0 of a bundle).
- Within a bundle, instructions are ordered from instruction slot 0 to instruction slot 2 as specified in Figure 3-15 on page 1:38.

Instruction execution consists of four phases:

1. Read the instruction from memory (*fetch*)
2. Read architectural state, if necessary (*read*)
3. Perform the specified operation (*execute*)

4. Update architectural state, if necessary (*update*).

An **instruction group** is a sequence of instructions starting at a given bundle address and slot number and including all instructions at sequentially increasing slot numbers and bundle addresses up to the first stop, taken branch, Break Instruction fault due to a `break.b`, or Illegal Operation fault due to a Reserved or Reserved if PR[qp] is one encoding in the B-type opcode space. For the instructions in an instruction group to have well-defined behavior, they must meet the ordering and dependency requirements described below.

For the purpose of clarification, the following do not end instruction groups:

- Break instructions other than `break.b` (`break.f`, `break.i`, `break.m`, `break.x`)
- Check instructions (`chk.s`, `chk.a`, `fchkf`)
- `rfi` instructions not followed by a stop
- `brl` instructions not followed by a stop
- Interruptions other than a Break Instruction fault due to a `break.b` or an Illegal Operation fault due to a Reserved or Reserved if PR[qp] is 1 encoding in the B-type opcode space

Thus, even if one of the above causes a change in control flow, the instructions at sequentially increasing addresses beyond the location of the change in control flow up to the next true end of the instruction group had the change of control flow not occurred, can still cause undefined values to be seen at the target of the change of control flow, if they cause a dependency violation. There are never, however, any dependencies between the instructions at the target of the change in control flow and those preceding the change in control flow, even for the above cases.

If the instructions in instruction groups meet the resource-dependency requirements, then the behavior of a program will be as though each individual instruction is sequenced through these phases in the order listed above. The order of a phase of a given instruction relative to any phase of a previous instruction is prescribed by the instruction sequencing rules below.

- There is no a priori relationship between the *fetch* of an instruction and the *read*, *execute*, or *update* of any dynamically previous instruction. The `sync.i` and `srlz.i` instructions can be used to enforce a sequential relationship between the *fetch* of all dynamically succeeding instructions and the *update* of all dynamically previous instructions.
- Between instruction groups, every instruction in a given instruction group will behave as though its read occurred after the update of all the instructions from the previous instruction group. All instructions are assumed to have unit latency. Instructions on opposing sides of a stop are architecturally considered to be separated by at least one unit of latency.

  Some system state updates require more stringent requirements than those described here. See Section 3.2, “Serialization” on page 2:17 for details.
- Within an instruction group, every instruction will behave as though its read of the memory and ALAT state occurred after the update of the memory and ALAT state of all prior instructions in that instruction group.
- Within an instruction group, every instruction will behave as though its read of the register state occurred before the update of the register state by any instruction (prior or later) in that instruction group, except as noted in the Register dependencies and Memory dependencies described below.

The ordering rules above form the context for register dependency restrictions, memory dependency restrictions and the order of exception reporting. These dependency restrictions apply only between instructions whose resource reads and writes are not dynamically disabled by predication.

- Register dependencies: Within an instruction group, read-after-write (RAW) and write-after-write (WAW) register dependencies are not allowed (except as noted in "RAW Dependency Special Cases" on page 1:42 and "WAW Dependency Special Cases" on page 1:43). Write-after-read (WAR) register dependencies are allowed (except as noted in "WAR Dependency Special Cases" on page 1:44).

  These dependency restrictions apply to both explicit register accesses (from the instruction's operands) and implicit register accesses (such as application and control registers implicitly accessed by certain instructions). Predicate register PR0 is excluded from these register dependency restrictions, since writes to PR0 are ignored and reads always return 1 (one).

  Some system state updates require more stringent requirements than those described here. See Section 3.2, "Serialization" on page 2:17 for details.
- Memory dependencies: Within an instruction group, RAW, WAW, and WAR memory dependencies and ALAT dependencies are allowed. A load will observe the results of the most recent store to the same memory address. In the event that multiple stores to the same address are present in the same instruction group, memory will contain the result of the latest store after execution of the instruction group. A store following a load to the same address will not affect the data loaded by the load. Advanced loads, check loads, advanced load checks, stores, and memory semaphore instructions implicitly access the ALAT. RAW, WAW, and WAR ALAT dependencies are allowed within an instruction group and behave as described for memory dependencies.

The net effect of the dependency restrictions stated above is that a processor may execute all (or any subset) of the instructions within a legal instruction group concurrently or serially with the end result being identical. If these dependency restrictions are not met, the behavior of the program is undefined (see "Undefined Behavior" on page 1:44).

Exceptions are reported in instruction order. The dependency restrictions apply independent of the presence or absence of exceptions — that is, restrictions must be satisfied whether or not an exception occurs within an instruction group. At the point of exception delivery for a correctly formed instruction group, all prior instructions will have completed their update of architectural state. All subsequent instructions will not have updated architectural state. If an instruction group violates a dependency requirement, then the update of architectural state before and after an exception is not guaranteed (the fault handler sees an undefined value on the registers involved in a dependency violation even if the exception occurs between the first and second instructions in the violation). In the event multiple exceptions occur while executing instructions from the same instruction group, the exception occurring on the earliest instruction will be reported.

The instruction sequencing resulting from the rules stated above is termed sequential execution.

The ordering rules and the dependency restrictions allow the processor to dynamically re-order instructions, execute instructions with non-unit latency, or even concurrently execute instructions on opposing sides of a stop or taken branch, provided that correct sequencing is enforced and the appearance of sequential execution is presented to the programmer.

IP is a special resource in that reads and writes of IP behave as though the instruction stream was being executed serially, rather than in parallel. RAW dependencies on IP are allowed, and the reader gets the IP of the bundle in which it is contained. So, each bundle being executed in parallel logically reads IP, increments it and writes it back. WAW is also allowed.

Ignored ARs are not exceptional for dependency checking purposes. RAW and WAW dependencies to ignored ARs are not allowed.

For more details on resource dependencies, see Chapter 5, "Resource and Dependency Semantics" in Volume 3.

3.4.1 RAW Dependency Special Cases

There are four special cases in which RAW register dependencies within an instruction group are permitted. These special cases are the `alloc` instruction, check load instructions, instructions that affect branching, and the `ld8.fill` and `st8.spill` instructions.

The `alloc` instruction implicitly writes the Current Frame Marker (CFM) which is implicitly read by all instructions accessing the stacked subset of the general register file. Instructions that access the stacked subset of the general register file may appear in the same instruction group as alloc and will see the stack frame specified by the `alloc`.

Note: Some instructions have RAW or WAW dependencies on resources other than CFM affected by `alloc` and are thus not allowed in the same instruction group after an `alloc`: `flushrs`, `loadrs`, move from AR[BSPSTORE], move from AR[RNAT], `br.cexit`, `br.ctop`, `br.wexit`, `br.wtop`, `br.call`, `brl.call`, `br.ia`, `br.ret`, `clrrrb`, `cover`, and `rfi`. See Chapter 5, "Resource and Dependency Semantics" in Volume 3 for details. Also note that `alloc` is required to be the first instruction in an instruction group.

A check load instruction may or may not perform a load since it is dependent upon its corresponding advanced load. If the check load misses the ALAT it will execute a load from memory. A check load and a subsequent instruction that reads the target of the check load may exist in the same instruction group. The dependent instruction will get the new value loaded by the check load.

A branch may read branch registers and may implicitly read predicate registers, the LC, EC, and PFS application registers, as well as CFM. Except for LC, EC and predicate registers, writes to any of these registers by a non-branch instruction will be visible to a subsequent branch in the same instruction group. Writes to predicate registers by any non-floating-point instruction will be visible to a subsequent branch in the same instruction group. RAW register dependencies within the same instruction group are not allowed for LC and EC. Dynamic RAW dependencies where the predicate writer is a floating-point instruction and the reader is a branch are also not allowed within the same instruction group. Branches `br.cond`, `br.call`, `brl.cond`, `brl.call`, `br.ret` and br.ia work like other instructions for the purposes of register dependency; i.e., if their qualifying predicate is 0, they are not considered readers or writers of other resources. Branches br.cloop, br.cexit, br.ctop, br.wexit, and br.wtop are exceptional in that they are always readers or writers of their resources, regardless of the value of their qualifying predicate. An indirect brp is considered a reader of the specified BR.

The ld8.fill and st8.spill instructions implicitly access the User NaT Collection application register (UNAT). For these instructions the restriction on dynamic RAW register dependencies with respect to UNAT applies at the bit level. These instructions may appear in the same instruction group provided they do not access the same bit of UNAT. RAW UNAT dependencies between ld8.fill or st8.spill instructions and mov ar= or mov =ar instructions accessing UNAT must not occur within the same instruction group.

For the purposes of resource dependencies, CFM is treated as a single resource.

3.4.2 WAW Dependency Special Cases

There are three special cases in which WAW register dependencies within an instruction group are permitted. The special cases are compare-type instructions, floating-point instructions, and the st8.spill instruction.

The set of compare-type instructions includes: cmp, cmp4, tbit, tnat, tf, fcmp, frsqrta, frcpa, and fclass. Compare-type instructions in the same instruction group may target the same predicate register provided:

- The compare-type instructions are either all AND-type compares or all OR-type compares (AND-type compares correspond to ".and" and ".andcm" completers; OR-type compares correspond to ".or" and ".orcm" completers), or
- The compare-type instructions all target PR0. All WAW dependencies for PR0 are allowed; the compares can be of any types and can be of differing types.

All other WAW dependencies within an instruction group are disallowed, including WAW register dependencies with move to PR instructions that access the same predicate registers as another writer.

Note: The move to PR instructions only writes those PRs indicated by its mask, but the move from PR instructions always reads all the predicate registers.

Floating-point instructions implicitly write the Floating-point Status Register (FPSR) and the Processor Status Register (PSR). Multiple floating-point instructions may appear in the same instruction group since the restriction on WAW register dependencies with respect to the FPSR and PSR do not apply. The state of FPSR and PSR after executing the instruction group will be the logical OR of all writes.

The st8.spill instruction implicitly writes the UNAT register. For this instruction the restriction on WAW register dependencies with respect to UNAT applies at the bit level. Multiple st8.spill instructions may appear in the same instruction group provided they do not write the same bit of UNAT. WAW register dependencies between st8.spill instructions and mov ar= instructions targeting UNAT must not occur within the same instruction group.

3.4.3 WAR Dependency Special Cases

The WAR dependency between the reading of predicate register 63 by any B-type instruction and the subsequent writing of predicate register 63 by a modulo-scheduled loop type branch (`br.ctop`, `br.cexit`, `br.wtop`, or `br.wexit`) without an intervening stop is not allowed. Otherwise, WAR dependencies within an instruction group are allowed.

3.4.4 Processor Behavior on Dependency Violations

If a program violates read-after-write, write-after-write or write-after-read resource dependency rules within an instruction group, then processor behavior is undefined. Constraints on undefined behavior are described in "Undefined Behavior" on page 1:44.

To help debug code that violates the architectural resource dependency rules, some processor implementations may provide dependency violation detection hardware that may cause an instruction group that contains an illegal dependency to take an Illegal Dependency fault (defined in Chapter 5, "Interruptions" in Volume 2). However, even in implementations that provide such checking, software can not assume the processor will catch all dependency violations or even catch the same violation every time it occurs.

However, all processor models that provide dependency violation detection hardware are required to satisfy the following dependency violation reporting constraints:

- All detected dependency violations must be reported as Illegal Dependency Faults (defined in Chapter 5, "Interruptions" in Volume 2). When an Illegal Dependency fault is taken, the value of the resource subject to the dependency violation is undefined. Undetected dependency violations cause undefined program behavior as described in "Undefined Behavior" on page 1:44.
- All detected read-after-write and write-after-write dependency violations must be delivered as Illegal Dependency Faults on the second operation, i.e. on the reader in the RAW case, and on second resource writer in the WAW case.
- All detected write-after-read dependency violations (on predicate register 63) must be delivered as Illegal Dependency faults on the second operation, the predicate writer.
- Illegal Dependency faults are delivered strictly in program order. If an interruption, branch or speculation check are taken between the first and the second operation of a dependency violation, then the Illegal Dependency fault is not taken.

Note: Since an instruction group starts at a given entry point (stop or target of a control flow transfer), instructions that precede the entry point are not considered part of the instruction group and must not take part in any dependency violation checking. For example, if an `rfi` is done to slot 1 of a bundle, the instruction in slot 0 and instructions in bundles with lower memory addresses are not part of the new instruction group, and must not take part in any dependency violation checking.

3.5 Undefined Behavior

Architecturally undefined behavior that applies to one or more instructions is listed below:

- RAW and WAW register dependencies within the same instruction group are disallowed except as noted in Section 3.4, "Instruction Sequencing Considerations" on page 1:39. Their behavior within an instruction group is undefined. Undefined behavior includes the possibility of an Illegal Operation fault.
- Reading a register outside of the defined general register stack frame boundaries (as determined by the most recent `alloc`, return, or call) will return an undefined result. All processors will not raise an interruption in this situation.

An undefined scenario is an event or sequence of events whose outcome is not defined in the architecture. For the behavior of Itanium instructions, refer to Chapter 2, "Instruction Reference" in Volume 3. For the behavior of IA32 instructions, refer to Volume 4: IA-32 Instruction Set Reference. Therefore, the result of an undefined scenario is strictly implementation dependent. User should not rely on these undefined behaviors for correct program behavior and compatibility across future implementations.

An undefined response (undefined behavior, undefined result) is subject to the following restrictions:

- It must not impede forward progress of the processor (i.e., the processor may not crash).
- It must not impede forward progress of other processors.
- It must not allow software to gain privileges not available at the current privilege level.
- It must not allow software to circumvent memory access rights.
- It must not modify state that cannot be modified by a defined response (e.g., a post-increment load instruction that generates an undefined response cannot modify any registers other than its target and address registers).
- It is subject to the same NaT/NaTVal propagation rules as a defined response.
- The processor may raise an Illegal Operation fault

§

Application Programming Model 4

This section describes the architectural functionality from the perspective of the application programmer. Itanium instructions are grouped into related functions and an overview of their behavior is given. Unless otherwise noted, all immediates are sign extended to 64 bits before use. The floating-point programming model is described separately in Chapter 5, "Floating-point Programming Model" in Volume 1. Refer to *Volume 3: Intel® Itanium® Instruction Set Reference* for detailed information on Itanium instructions.

The main features of the programming model covered here are:

- General Register Stack
- Integer Computation Instructions
- Compare Instructions and Predication
- Memory Access Instructions and Speculation
- Branch Instructions and Branch Prediction
- Multimedia Instructions
- Register File Transfer Instructions
- Character Strings and Population Count
- Privilege Level Transfer

4.1 Register Stack

As described in "General Registers" on page 1:25, the general register file is divided into static and stacked subsets. The static subset is visible to all procedures and consists of the 32 registers from GR 0 through GR 31. The stacked subset is local to each procedure and may vary in size from zero to 96 registers beginning at GR 32. The register stack mechanism is implemented by renaming register addresses as a side-effect of procedure calls and returns. The implementation of this rename mechanism is not otherwise visible to application programs. The register stack is disabled during IA-32 instruction set execution.

The static subset must be saved and restored at procedure boundaries according to software convention. The stacked subset is automatically saved and restored by the Register Stack Engine (RSE) without explicit software intervention (for details on the RSE see Chapter 6, "Register Stack Engine" in Volume 2). All other register files are visible to all procedures and must be saved/restored by software according to software convention.

4.1.1 Register Stack Operation

The registers in the stacked subset visible to a given procedure are called a register stack frame. The frame is further partitioned into two variable-size areas: the local area and the output area. Immediately after a call, the size of the local area of the newly activated frame is zero and the size of the output area is equal to the size of the caller's output area and overlays the caller's output area.

The local and output areas of a frame can be re-sized using the alloc instruction which specifies immediates that determine the size of frame (sof) and size of locals (sol).

Note: In the assembly language, alloc uses three immediate operands to determine the values of sol and sof: the size of inputs; the size of locals; and the size of outputs. The value of sol is determined by adding the size of inputs immediate and the size of locals immediate; the value of sof is determined by adding all three immediates.

The value of sof specifies the size of the entire stacked subset visible to the current procedure; the value of sol specifies the size of the local area. The size of the output area is determined by the difference between sof and sol. The values of these parameters for the currently active procedure are maintained in the Current Frame Marker (CFM).

Reading a stacked register outside the current frame will return an undefined result. Writing a stacked register outside the current frame will cause an Illegal Operation fault.

When a br.call or brl.call is executed, the CFM is copied to the Previous Frame Marker (PFM) field in the Previous Function State application register (PFS), and the callee's frame is created as follows:

- The stacked registers are renamed such that the first register in the caller's output area becomes GR 32 for the callee
- The size of the local area is set to zero
- The size of the callee's frame (sof_{b1}) is set to the size of the caller's output area (sof_a - sol_a)

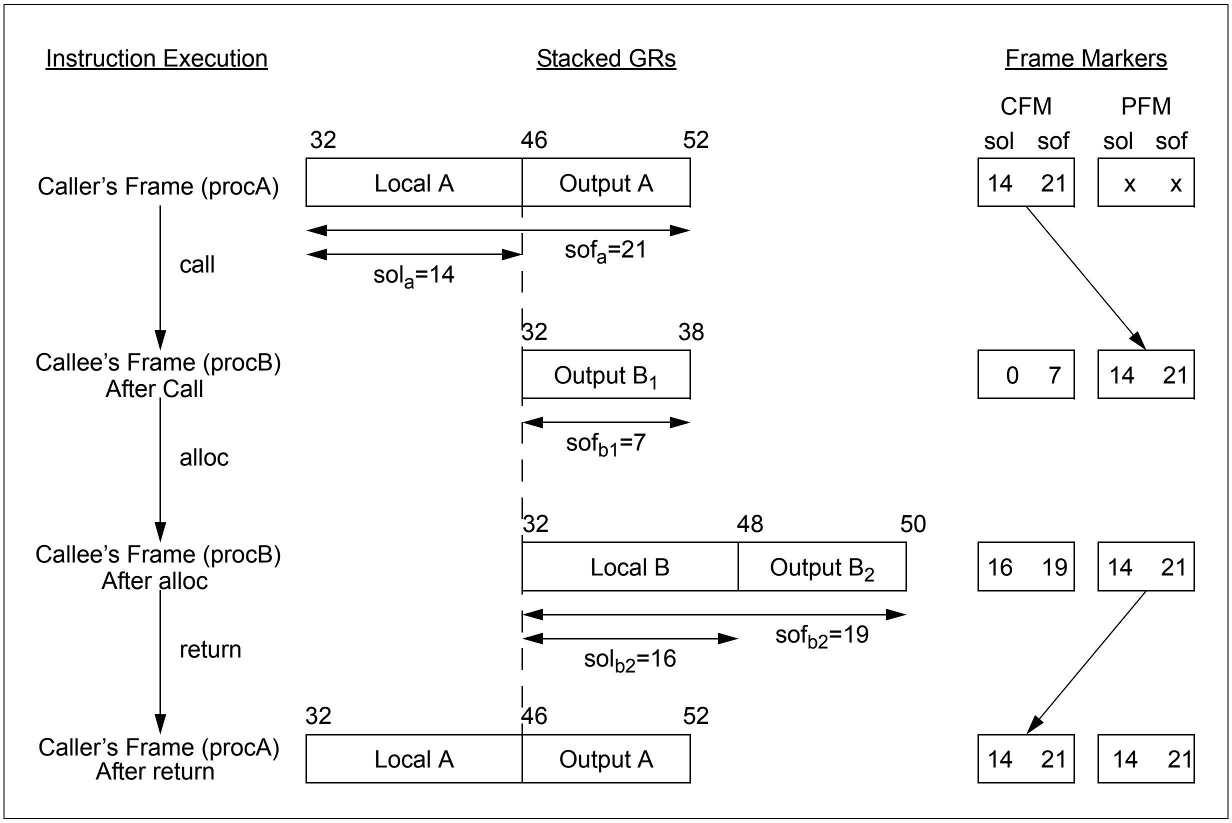
Values in the output area of the caller's register stack frame are visible to the callee. This overlap permits parameter and return value passing between procedures to take place entirely in registers.

Procedure frames may be dynamically re-sized by issuing an alloc instruction. An alloc instruction causes no renaming, but only changes the size of the register stack frame and the partitioning between local and output areas. Typically, when a procedure is called, it will allocate some number of local registers for its use (which will include the parameters passed to it in the caller's output registers), plus an output area (for passing parameters to procedures it will call). Newly allocated registers (including their NaT bits) have undefined values.

When a br.ret is executed, CFM is restored from PFM and the register renaming is restored to the caller's configuration. The PFM is procedure local state and must be saved and restored by non-leaf procedures. The CFM is not directly accessible in application programs and is updated only through the execution of calls, returns, alloc, cover, and clrrrb.

Figure 4-1 depicts the behavior of the register stack on a procedure call from procA (caller) to procB (callee). The state of the register stack is shown at four points: prior to the call, immediately following the call, after procB has executed an alloc, and after procB returns to procA.

Figure 4-1. Register Stack Behavior on Procedure Call and Return



The majority of application programs need only issue `alloc` instructions and save/restore PFM in order to effectively utilize the register stack. A detailed knowledge of the RSE (Register Stack Engine) is required only by certain specialized application software such as user-level thread packages, debuggers, etc. See Chapter 6, "Register Stack Engine" in Volume 2.

4.1.2 Register Stack Instructions

The `alloc` instruction is used to change the size of the current register stack frame. An `alloc` instruction must be the first instruction in an instruction group otherwise the results are undefined. An `alloc` instruction affects the register stack frame seen by all instructions in an instruction group, including the `alloc` itself. If the qualifying predicate for `alloc` is not PR0, an Illegal Operation fault is raised. An `alloc` does not affect the values or NaT bits of the allocated registers. When a register stack frame is expanded, newly allocated registers may have their NaT bit set.

In addition, there are three instructions which provide explicit control over the state of the register stack. These instructions are used in thread and context switching which necessitate a corresponding switch of the backing store for the register stack. See Chapter 6, "Register Stack Engine" in Volume 2 for details on explicit management of the RSE.

The flushrs instruction is used to force all previous stack frames out to backing store memory. It stalls instruction execution until all active frames in the physical register stack up to, but not including the current frame are spilled to the backing store by the RSE. A flushrs instruction must be the first instruction in an instruction group; otherwise, the results are undefined. A flushrs cannot be predicated.

The cover instruction creates a new frame of zero size (sof = sol = 0). The new frame is created above (not overlapping) the present frame. Both the local and output areas of the previous stack frame are automatically saved. A cover instruction must be the last instruction in an instruction group; otherwise, operation is undefined. A cover cannot be predicated.

The loadrs instruction ensures that the specified portion of the register stack is present in the physical registers. It stalls instruction execution until the number of bytes specified in the loadrs field of the RSC application register have been filled from the backing store by the RSE (starting from the current BSP). By specifying a zero value for RSC.loadrs, loadrs can be used to indicate that all stacked registers outside the current frame must be loaded from the backing store before being used. In addition, stacked registers outside the current frame (that have not been spilled by the RSE) will not be stored to the backing store. A loadrs instruction must be the first instruction in an instruction group otherwise the results are undefined. A loadrs cannot be predicated.

Table 4-1 lists the architectural visible state relating to the register stack. Table 4-2 summarizes the register stack management instructions. Call- and return-type branches, which affect the stack, are described in "Branch Instructions" on page 1:74.

Table 4-1. Architectural Visible State Related to the Register Stack

Register	Description
AR[PFS].pfm	Previous Frame Marker field
AR[RSC]	Register Stack Configuration application register
AR[BSP]	Backing store pointer application register
AR[BSPSTORE]	Backing store pointer application register for memory stores
AR[RNAT]	RSE NaT collection application register

Table 4-2. Register Stack Management Instructions

Mnemonic	Operation
alloc	Allocate register stack frame
flushrs	Flush register stack to backing store
loadrs	Load register stack from backing store
cover	Cover current stack frame

4.2 Integer Computation Instructions

The integer execution units provide a set of arithmetic, logical, shift and bit-field-manipulation instructions. Additionally, they provide a set of instructions to accelerate operations on 32-bit data and pointers.

Arithmetic, logical and 32-bit acceleration instructions can be executed on both I- and M-units

4.2.1 Arithmetic Instructions

Addition and subtraction (add, sub) are supported with regular two input forms and special three input forms. The three input addition form adds one to the sum of two input registers. The three input subtraction form subtracts one from the difference of two input registers. The three input forms share the same mnemonics as the two input forms and are specified by appending a "1" as a third source operand.

The immediate form of addition uses a register and a 14-bit immediate; the immediate form of subtraction uses a register and an 8-bit immediate. In both cases, the immediate is sign-extended before being added or subtracted. The immediate form is obtained simply by specifying an immediate rather than a register as the first operand. Also, addition can be performed between a register and a 22-bit immediate; however, the source register must be GR 0, 1, 2 or 3.

A shift left and add instruction (shladd) shifts one register operand to the left by 1 to 4 bits and adds the result to a second register operand.

32-bit multiplication is supported with the unsigned integer multiply (mpy4) instruction, which takes two 32-bit (unsigned) register operands and produces a 64-bit result. The unsigned integer shift left and multiply (mpyshl4) instruction provides a building block for doing 64-bit multiplication. It takes a 32-bit operand in the upper half of a first register, a 32-bit operand in the lower half of a second register, multiplies them, and places the least significant 32-bits of the product in the upper half of the result register, with zeros in the lower half.

Table 4-3 summarizes the integer arithmetic instructions.

Table 4-3. Integer Arithmetic Instructions

Mnemonic	Operation
add	Addition
add...,1	Three input addition
mpy4	Unsigned integer multiply
mpyshl4	Unsigned integer shift left and multiply
sub	Subtraction
sub...,1	Three input subtraction
shladd	Shift left and add

Note that an integer multiply instruction is defined which uses the floating-point registers. See "Integer Multiply and Add Instructions" on page 1:101 for details. Integer divide is performed in software similarly to floating-point divide.

4.2.2 Logical Instructions

Instructions to perform logical AND (and), OR (or), and exclusive OR (xor) between two registers or between a register and an immediate are defined. The andcm instruction performs a logical AND of a register or an immediate with the complement of another register. Table 4-4 summarizes the integer logical instructions.

Table 4-4. Integer Logical Instructions

Mnemonic	Operation
and	Logical and
or	Logical or
andcm	Logical and complement
xor	Logical exclusive or

4.2.3 32-bit Addresses and Integers

Support for 32-bit addresses is provided in the form of add instructions that perform region bit copying. This supports the virtual address translation model (see "32-bit Virtual Addressing" on page 2:71 for details). The add 32-bit pointer instruction (addp) adds two registers or a register and an immediate, zeroes the most significant 32-bits of the result, and copies bits 31:30 of the second source to bits 62:61 of the result. The shladdp instruction operates similarly but shifts the first source to the left by 1 to 4 bits before performing the add, and is provided only in the two-register form.

In addition, support for 32-bit integers is provided through 32-bit compare instructions and instructions to perform sign and zero extension. Compare instructions are described in "Compare Instructions and Predication" on page 1:54. The sign and zero extend (sxt, zxt) instructions take an 8-bit, 16-bit, or 32-bit value in a register, and produce a properly extended 64-bit result.

Table 4-5 summarizes 32-bit pointer and 32-bit integer instructions.

Table 4-5. 32-bit Pointer and 32-bit Integer Instructions

Mnemonic	Operation
addp	32-bit pointer addition
shladdp	Shift left and add 32-bit pointer
sxt	Sign extend
zxt	Zero extend

4.2.4 Bit Field and Shift Instructions

Four classes of instructions are defined for shifting and operating on bit fields within a general register: variable shifts, fixed shift-and-mask instructions, a 128-bit-input funnel shift, and special compare operations to test an individual bit within a general register. The compare instructions for testing a single bit (tbit), or for testing the NaT bit (tnat) are described in "Compare Instructions and Predication" on page 1:54.

The variable shift instructions shift the contents of a general register by an amount specified by another general register. The shift right signed (shr) and shift right unsigned (shr.u) instructions shift the contents of a register to the right with the vacated bit positions filled with the sign bit or zeroes respectively. The shift left (shl) instruction shifts the contents of a register to the left.

The fixed shift-and-mask instructions (extr, dep) are generalized forms of fixed shifts. The extract instruction (extr) copies an arbitrary bit field from a general register to the least-significant bits of the target register. The remaining bits of the target are written with either the sign of the bit field (extr) or with zero (extr.u). The length and starting position of the field are specified by two immediates. This is essentially a shift-right-and-mask operation. A simple right shift by a fixed amount can be specified by using shr with an immediate value for the shift amount. This is just an assembly pseudo-op for an extract instruction where the field to be extracted extends all the way to the left-most register bit.

The deposit instruction (dep) takes a field from either the least-significant bits of a general register, or from an immediate value of all zeroes or all ones, places it at an arbitrary position, and fills the result to the left and right of the field with either bits from a second general register (dep) or with zeroes (dep.z). The length and starting position of the field are specified by two immediates. This is essentially a shift-left-mask-merge operation. A simple left shift by a fixed amount can be specified by using shl with an immediate value for the shift amount. This is just an assembly pseudo-op for dep.z where the deposited field extends all the way to the left-most register bit.

The shift right pair (shrp) instruction performs a 128-bit-input funnel shift. It extracts an arbitrary 64-bit field from a 128-bit field formed by concatenating two source general registers. The starting position is specified by an immediate. This instruction can be used to accelerate the adjustment of unaligned data. A bit rotate operation can be performed by using shrp and specifying the same register for both operands.

Table 4-6 summarizes the bit field and shift instructions.

Table 4-6. Bit Field and Shift Instructions

Mnemonic	Operation
shr	Shift right signed
shr.u	Shift right unsigned
shl	Shift left
extr	Extract signed (shift right and mask)
extr.u	Extract unsigned (shift right and mask)
dep	Deposit (shift left, mask and merge)
dep.z	Deposit in zeroes (shift left and mask)
shrp	Shift right pair

4.2.5 Large Constants

A special instruction is defined for generating large constants (see Table 4-7). For constants up to 22 bits in size, the add instruction can be used, or the mov pseudo-op (pseudo-op of add with GR0, which always reads 0). For larger constants, the move long immediate instruction (movl) is defined to write a 64-bit immediate into a general register. This instruction occupies two instruction slots within the same bundle, and is the only such instruction.

Table 4-7. Instructions to Generate Large Constants

Mnemonic	Operation
mov	Move 22-bit immediate
movl	Move 64-bit immediate

4.3 Compare Instructions and Predication

A set of compare instructions provides the ability to test for various conditions and affect the dynamic execution of instructions. A compare instruction tests for a single specified condition and generates a boolean result. These results are written to predicate registers. The predicate registers can then be used to affect dynamic execution in two ways: as conditions for conditional branches, or as qualifying predicates for predication.

4.3.1 Predication

Predication is the conditional execution of instructions. The execution of most instructions is gated by a qualifying predicate. If the predicate is true, the instruction executes normally; if the predicate is false, the instruction does not modify architectural state (except for the unconditional type of compare instructions, floating-point approximation instructions and while-loop branches). Predicates are one-bit values and are stored in the predicate register file. A zero predicate is interpreted as false and a one predicate is interpreted as true (predicate register PR0 is hardwired to one).

A few instructions cannot be predicated. These instructions are: allocate stack frame (`alloc`), branch predict (`brp`), bank switch (`bsw`), clear rrb (`clrrrb`), cover stack frame (`cover`), enter privileged code (`epc`), flush register stack (`flushrs`), load register stack (`loadrs`), counted branches (`br.cloop`, `br.ctop`, `br.cexit`), and return from interruption (`rfi`).

4.3.2 Compare Instructions

Predicate registers are written by the following instructions: general register compare (`cmp`, `cmp4`), floating-point register compare (`fcmp`), test bit and test NaT (`tbit`, `tnat`), test feature (`tf`), floating-point class (`fclass`), and floating-point reciprocal approximation and reciprocal square root approximation (`frcpa`, `fprcpa`, `frsqrta`, `fprsqrta`). Most of these compare instructions (all but `frcpa`, `fprcpa`, `frsqrta` and `fprsqrta`) set two predicate registers based on the outcome of the comparison. The setting of the two target registers is described below in "Compare Types" on page 1:55. Compare instructions are summarized in Table 4-8.

Table 4-8. Compare Instructions

Mnemonic	Operation
`cmp, cmp4`	GR compare
`tbit`	Test bit in a GR
`tnat`	Test GR NaT bit
`tf`	Test feature
`fcmp`	FR compare
`fclass`	FR class
`frcpa, fprcpa`	Floating-point reciprocal approximation
`frsqrta, fprsqrta`	Floating-point reciprocal square root approximation

The 64-bit (cmp) and 32-bit (cmp4) compare instructions compare two registers, or a register and an immediate, for one of ten relations (e.g., >, <=). The compare instructions set two predicate targets according to the result. The cmp4 instruction compares the least-significant 32-bits of both sources (the most significant 32-bits are ignored).

The test bit (tbit) instruction sets two predicate registers according to the state of a single bit in a general register (the position of the bit is specified by an immediate). The test NaT (tnat) instruction sets two predicate registers according to the state of the NaT bit corresponding to a general register.

The test feature (tf) instruction sets two predicate registers according to whether or not the selected feature is implemented in the processor.

The fcmp instruction compares two floating-point registers and sets two predicate targets according to one of eight relations. The fclass instruction sets two predicate targets according to the classification of the number contained in the floating-point register source.

The frcpa, fprcpa, frsqrta and fprsqrta instructions set a single predicate target if their floating-point register sources are such that a valid approximation can be produced, otherwise the predicate target is cleared.

4.3.3 Compare Types

Compare instructions can have as many as five compare types: Normal, Unconditional, AND, OR, or DeMorgan. The type defines how the instruction writes its target predicate registers based on the outcome of the comparison and on the qualifying predicate. The description of these types is contained in Table 4-9. In the table, "qp" refers to the value of the qualifying predicate of the compare and "result" refers to the outcome of the compare relation (one if the compare relation is true and zero if the compare relation is false).

Table 4-9. Compare Type Function

Compare Type	Completer	Operation	
		First Predicate Target	Second Predicate Target
Normal	*none*	if (qp) {target = result}	if (qp) {target =!result}

Unconditional	unc	if (qp) {target = result}
		else {target = 0}
			if (qp) {target =!result}
			else {target = 0}

AND	and	if (qp &&!result) {target = 0}	if (qp &&!result) {target = 0}
	andcm	if (qp && result) {target = 0}	if (qp && result) {target = 0}
OR	or	if (qp && result) {target = 1}	if (qp && result) {target = 1}
	orcm	if (qp &&!result) {target = 1}	if (qp &&!result) {target = 1}
DeMorgan	or.andcm	if (qp && result) {target = 1}	if (qp && result) {target = 0}
	and.orcm	if (qp &&!result) {target = 0}	if (qp &&!result) {target = 1}

The Normal compare type simply writes the compare result to the first predicate target and the complement of the result to the second predicate target.

The Unconditional compare type behaves the same as the Normal type, except that if the qualifying predicate is 0, both predicate targets are written with 0. This can be thought of as an initialization of the predicate targets, combined with a Normal compare. Note that compare instructions with the Unconditional type modify architectural state when their qualifying predicate is false.

The AND, OR and DeMorgan types are termed "parallel" compare types because they allow multiple simultaneous compares (of the same type) to target a single predicate register. This provides the ability to compute a logical equation such as `p5 = (r4 == 0) || (r5 == r6)` in a single cycle (assuming p5 was initialized to 0 in an earlier cycle). The DeMorgan compare type is just a combination of an OR type to one predicate target and an AND type to the other predicate target. Multiple OR-type compares (including the OR part of the DeMorgan type) may specify the same predicate target in the same instruction group. Multiple AND-type compares (including the AND part of the DeMorgan type) may also specify the same predicate target in the same instruction group.

For all compare instructions (except for `tnat` and `fclass`), if one or both of the source registers contains a deferred exception token (NaT or NaTVal – see "Control Speculation" on page 1:60), the result of the compare is different. Both predicate targets are treated the same, and are either written to 0 or left unchanged. In combination with speculation, this allows predicated code to be turned off in the presence of a deferred exception. `fclass` behaves this way as well if NaTVal is not one of the classes being tested for. Table 4-10 describes the behavior.

Table 4-10. Compare Outcome with NaT Source Input

Compare Type	Operation
Normal	if (qp) {target = 0}
Unconditional	target = 0
AND	if (qp) {target = 0}
OR	(not written)
DeMorgan	(not written)

Only a subset of the compare types are provided for some of the compare instructions. Table 4-11 lists the compare types which are available for each of the instructions.

Table 4-11. Instructions and Compare Types Provided

Instruction	Relation	Types Provided
`cmp, cmp4`	a == b, a!= b, a > 0, a >= 0, a < 0, a <= 0, 0 > a, 0 >= a, 0 < a, 0 <= a	Normal, Unconditional, AND, OR, DeMorgan
	All other relations	Normal, Unconditional
`tbit, tnat, tf`	All	Normal, Unconditional, AND, OR, DeMorgan
`fcmp, fclass`	All	Normal, Unconditional

`frcpa, frsqrta,`
`fprcpa, fprsqrta`
	Not Applicable	Unconditional

4.3.4 Predicate Register Transfers

Instructions are provided to transfer between the predicate register file and a general register. These instructions operate in a "broadside" manner whereby multiple predicate registers are transferred in parallel, such that predicate register N is transferred to/from bit N of a general register.

The move to predicates instruction (`mov pr=`) loads multiple predicate registers from a general register according to a mask specified by an immediate. The mask contains one bit for each of PR 1 through PR 15 (PR 0 is hardwired to 1) and one bit for all of PR 16 through PR63 (the rotating predicates). A predicate register is written from the corresponding bit in a general register if the corresponding mask bit is 1; if the mask bit is 0 the predicate register is not modified.

The move to rotating predicates instruction (`mov pr.rot=`) copies 48 bits from an immediate value into the 48 rotating predicates (PR 16 through PR 63). The immediate value includes 28 bits, and is sign-extended. Thus PR 16 through PR 42 can be independently set to new values, and PR 43 through PR 63 are all set to either 0 or 1.

The move from predicates instruction (`mov =pr`) transfers the entire predicate register file into a general register target.

For all of these predicate register transfers, the predicate registers are accessed as though the register rename base (CFM.rrb.pr) were 0. Typically, therefore, software should clear CFM.rrb.pr before initializing rotating predicates.

4.4 Memory Access Instructions

Memory is accessed by simple load, store and semaphore instructions, which transfer data to and from general registers or floating-point registers. The memory address is specified by the contents of a general register.

Most load and store instructions can also specify base-address-register update. Base update adds either an immediate value or the contents of a general register to the address register, and places the result back in the address register. The update is done after the load or store operation, i.e., it is performed as an address post-increment.

For highest performance, data should be aligned on natural boundaries. Within a 4K-byte boundary, accesses misaligned with respect to their natural boundaries will always fault if UM.ac (alignment check bit in the User Mask register) is 1. If UM.ac is 0, then an unaligned access will succeed if it is supported by the implementation; otherwise it will cause an Unaligned Data Reference fault. Please see the processor-specific documentation for further information. All memory accesses that cross a 4K-byte boundary will cause an Unaligned Data Reference fault independent of UM.ac. Additionally, all semaphore instructions will cause an Unaligned Data Reference fault if the access is not aligned to its natural boundary, independent of UM.ac.

Accesses to memory quantities larger than a byte may be done in a big-endian or little-endian fashion. The byte ordering for all memory access instructions is determined by UM.be in the User Mask register. All IA-32 memory references are performed little-endian.

Load, store and semaphore instructions are summarized in Table 4-12 and the state related to memory reference instructions is summarized in Table 4-13.

Table 4-12. Memory Access Instructions

Mnemonic			Operation
General	Floating-point		
	Normal	Load Pair	
`ld`	`ldf`	`ldfp`	Load
`ld.s`	`ldf.s`	`ldfp.s`	Speculative load
`ld.a`	`ldf.a`	`ldfp.a`	Advanced load
`ld.sa`	`ldf.sa`	`ldfp.sa`	Speculative advanced load

`ld.c.nc, ld.c.clr`	`ldf.c.nc,`
	`ldf.c.clr`
		`ldfp.c.nc,`
		`ldfp.c.clr`
			Check load

`ld.c.clr.acq`			Ordered check load
`ld.acq`			Ordered load
`ld.bias`			Biased load
`ld.fill`	`ldf.fill`		Register Fill
`st`	`stf`		Store
`st.rel`			Ordered store
`st.spill`	`stf.spill`		Register Spill
`cmpxchg`			Compare and exchange
`xchg`			Exchange memory and GR
`fetchadd`			Fetch and add

Table 4-13. State Relating to Memory Access

Register	Function
UM.be	User mask byte ordering
UM.ac	User mask Unaligned Data Reference fault enable
UNAT	GR NaT collection
CCV	Compare and Exchange Compare Value application register
CSD	Compare and Store Data application register

4.4.1 Load Instructions

Load instructions transfer data from memory to a general register, a general register and the Compare and Store Data register (CSD), a floating-point register or a pair of floating-point registers.

For general register loads, access sizes of 1, 2, 4, 8, and 16 bytes are defined. For sizes less than eight bytes, the loaded value is zero extended to 64-bits. The 16-byte general-register load instructions load two adjacent 8-byte quantities into a general register and the CSD register. The 16-byte general-register load instructions cannot specify base register update.

For floating-point loads, the following access sizes are defined: single precision (4 bytes), double precision (8 bytes), double-extended precision (10 bytes), and integer/parallel FP (8 bytes). The value(s) loaded from memory are converted into floating-point register format (see "Memory Access Instructions" on page 1:91 for details).

The floating-point load pair instructions load two adjacent single precision (4 bytes each), double precision (8 bytes each), or integer/parallel FP (8 bytes each) numbers into two independent floating-point registers (see the `ldfp` instruction description for restrictions on target register specifiers). Floating-point load pair instructions can specify base register update, but only by an immediate value equal to double the data size.

Variants of both general and floating-point register loads are defined for supporting compiler-directed control and data speculation. These use the general register NaT bits and the ALAT. See "Control Speculation" on page 1:60 and "Data Speculation" on page 1:63.

Variants are also provided for controlling the memory/cache subsystem. An ordered load can be used to force ordering in memory accesses. See "Memory Access Ordering" on page 1:73. A biased load provides a hint to acquire exclusive ownership of the accessed line. See "Memory Hierarchy Control and Consistency" on page 1:69.

Special-purpose loads are defined for restoring register values that were spilled to memory. The `ld8.fill` instruction loads a general register and the corresponding NaT bit (defined for an 8-byte access only). The `ldf.fill` instruction loads a value in floating-point register format from memory without conversion (defined for 16-byte access only). See "Register Spill and Fill" on page 1:62.

4.4.2 Store Instructions

Store instructions transfer data from a general register, a general register and the CSD register, or floating-point register to memory. Store instructions are always non-speculative. Store instructions can specify base-address-register update, but only by an immediate value. A variant is also provided for controlling the memory/cache subsystem. An ordered store can be used to force ordering in memory accesses.

Both general and floating-point register stores are defined with the same access sizes as their load counterparts. The only exception is that there are no floating-point store pair instructions. The 16-byte general-register store instructions store two adjacent 8-byte quantities from a general register and the CSD register.

Special purpose stores are defined for spilling register values to memory. The `st8.spill` instruction stores a general register and the corresponding NaT bit (defined for 8-byte access only). This allows the result of a speculative calculation to be spilled to memory and restored. The `stf.spill` instruction stores a floating-point register in memory in the floating-point register format without conversion. This allows register spill and restore code to be written to be compatible with possible future extensions to the floating-point register format. The `stf.spill` instruction also does not fault if the register contains a NaTVal, and is defined for 16-byte access only. See "Register Spill and Fill" on page 1:62.

4.4.3 Semaphore Instructions

Semaphore instructions atomically load a general register from memory, perform an operation and then store a result to the same memory location. Semaphore instructions are always non-speculative. No base register update is provided.

Three types of atomic semaphore operations are defined: exchange (xchg); compare and exchange (cmpxchg); and fetch and add (fetchadd).

The xchg target is loaded with the zero-extended contents of the memory location addressed by the first source and then the second source is stored into the same memory location.

The cmpxchg target is loaded with the zero-extended contents of the memory location addressed by the first source; if the zero-extended value is equal to the contents of the Compare and Exchange Compare Value application register (CCV), then the second source is stored into the same memory location. The cmp8xchg16 instruction loads the target with 8 bytes from the memory location addressed by the first source; if this value is equal to the contents of the CCV register, then the second source and the CSD register are both stored into memory at the 16-byte-aligned address which contains the memory location loaded.

The fetchadd instruction specifies one general register source, one general register target, and an immediate. The fetchadd target is loaded with the zero-extended contents of the memory location addressed by the source and then the immediate is added to the loaded value and the result is stored into the same memory location.

4.4.4 Control Speculation

Special mechanisms are provided to allow for compiler-directed speculation. This speculation takes two forms, control speculation and data speculation, with a separate mechanism to support each. See also "Data Speculation" on page 1:63.

4.4.4.1 Control Speculation Concepts

Control speculation describes the compiler optimization where an instruction or a sequence of instructions is executed before it is known that the dynamic control flow of the program will actually reach the point in the program where the sequence of instructions is needed. This is done with instruction sequences that have long execution latencies. Starting the execution early allows the compiler to overlap the execution with other work, increasing the parallelism and decreasing overall execution time. The compiler performs this optimization when it determines that it is very likely that the dynamic control flow of the program will eventually require this calculation. In cases where the control flow is such that the calculation turns out not to be needed, its results are simply discarded (the results in processor registers are simply not used).

Since the speculative instruction sequence may not be required by the program, no exceptions encountered that would be visible to the program can be signalled until it is determined that the program's control flow does require the execution of this instruction sequence. For this reason, a mechanism is provided for recording the occurrence of an exception so that it can be signalled later if and when it is necessary. In such a situation, the exception is said to be deferred. When an exception is deferred by an instruction, a special token is written into the target register to indicate the existence of a deferred exception in the program.

Deferred exception tokens are represented differently in the general and floating-point register files. In general registers, an additional bit is defined for each register called the NaT bit (Not a Thing). Thus general registers are 65 bits wide. A NaT bit equal to 1 indicates that the register contains a deferred exception token, and that its 64-bit data portion contains an implementation-specific value that software cannot rely upon. In floating-point registers, a deferred exception is indicated by a specific pseudo-zero encoding called the NaTVal (see "Representation of Values in Floating-point Registers" on page 1:86 for details).

4.4.4.2 Control Speculation and Instructions

Instructions are divided into two categories: speculative (instructions which can be used speculatively) and non-speculative (instructions which cannot). Non-speculative instructions will raise exceptions if they occur and are therefore unsafe to schedule before they are known to be executed. Speculative instructions defer exceptions (they do not raise them) and are therefore safe to schedule before they are know to be executed.

Loads to general and floating-point registers have both non-speculative (`ld`, `ldf`, `ldfp`) and speculative (`ld.s`, `ldf.s`, `ldfp.s`) variants. Generally, all computation instructions which write their results to general or floating-point registers are speculative. Any instruction that modifies state other than a general or floating-point register is non-speculative, since there would be no way to represent the deferred exception (there are a few exceptions).

Deferred exception tokens propagate through the program in a dataflow manner. A speculative instruction that reads a register containing a deferred exception token will propagate a deferred exception token into its target. Thus a chain of instructions can be executed speculatively, and only the result register need be checked for a deferred exception token to determine whether any exceptions occurred.

At the point in the program when it is known that the result of a speculative calculation is needed, a speculation check (`chk.s`) instruction is used. This instruction tests for a deferred exception token. If none is found, then the speculative calculation was successful, and execution continues normally. If a deferred exception token is found, then the speculative calculation was unsuccessful and must be re-done. In this case, the `chk.s` instruction branches to a new address (specified by an immediate offset in the `chk.s` instruction). Software can use this mechanism to invoke code that contains a copy of the speculative calculation (but with non-speculative loads). Since it is now known that the calculation is required, any exceptions which now occur can be signalled and handled normally.

Since computational instructions do not generally cause exceptions, the only instructions which generate deferred exception tokens are speculative loads. (IEEE floating-point exceptions are handled specially through a set of alternate status fields. See "Floating-point Status Register" on page 1:88.) Other speculative instructions propagate deferred exception tokens, but do not generate them.

4.4.4.3 Control Speculation and Compares

As stated earlier, most instructions that write a register file other than the general registers or the floating-point registers are non-speculative. The compare (`cmp`, `cmp4`, `fcmp`), test bit (`tbit`), floating-point class (`fclass`), and floating-point approximation (`frcpa`, `frsqrta`) instructions are special cases. These instructions read general or floating-point registers and write one or two predicate registers.

For these instructions, if any source contains a deferred exception token, all predicate targets are either cleared or left unchanged, depending on the compare type (see Table 4-10 on page 1:56). Software can use this behavior to ensure that any dependent conditional branches are not taken and any dependent predicated instructions are nullified. See "Predication" on page 1:54.

Deferred exception tokens can also be tested for with certain compare instructions. The test NaT (`tnat`) instruction tests the NaT bit corresponding to the specified general register and writes two predicate results. The floating-point class (`fclass`) instruction can be used to test for a NaTVal in a floating-point register and write the result to two predicate registers. `fclass` does not clear both predicate targets in the presence of a NaTVal input if NaTVal is one of the classes being tested for.

4.4.4.4 Control Speculation without Recovery

A non-speculative instruction that reads a register containing a deferred exception token will raise a Register NaT Consumption fault. Such instructions can be thought of as performing a non-recoverable speculation check operation. In some compilation environments, it may be true that the only exceptions that are deferred are fatal errors. In such a program, if the result of a speculative calculation is checked and a deferred exception token is found, execution of the program is terminated. For such a program, the results of speculative calculations can be checked simply by using non-speculative instructions.

4.4.4.5 Operating System Control over Exception Deferral

An additional mechanism is defined that allows the operating system to control the exception behavior of speculative loads. The operating system has the option to select which exceptions are deferred automatically in hardware and which exceptions will be handled (and possibly deferred) by software. See Section 5.5.5, "Deferral of Speculative Load Faults" on page 2:105.

4.4.4.6 Register Spill and Fill

Special store and load instructions are provided for spilling a register to memory and preserving any deferred exception token, and for restoring a spilled register.

The spill and fill general register instructions (`st8.spill`, `ld8.fill`) are defined to save/restore a general register along with the corresponding NaT bit.

The `st8.spill` instruction writes a general register's NaT bit into the User NaT Collection application register (UNAT), and, if the NaT bit was 0, writes the register's 64-bit data portion to memory. If the register's NaT bit was 1, the UNAT is updated, but the memory update is implementation specific. As stated in Section 4.4.4.1, "Control Speculation Concepts", software cannot rely on the 64-bit data portion spilled to memory for a NaT'ed GR. Although guidance is given here for processor implementations, other allowed implementation strategies may be added in the future, and software should not rely on the implementation guidance.

Processor implementations (hardware and firmware) must consistently follow one of two spill behaviors (but software should not count on implementations being limited to these behaviors):

- The `st8.spill` may write a zero to the specified memory location, or
- The `st8.spill` may write the register's 64-bit data portion to memory, only if that implementation returns a zero into the target register of all NaTed speculative loads, and that implementation also guarantees that all NaT propagating instructions perform all computations as specified by the instruction pages.

Bits 8:3 of the memory address determine which bit in the UNAT register is written.

The `ld8.fill` instruction loads a general register from memory taking the corresponding NaT bit from the bit in the UNAT register addressed by bits 8:3 of the memory address. The UNAT register must be saved and restored by software. It is the responsibility of software to ensure that the contents of the UNAT register are correct while executing `st8.spill` and `ld8.fill` instructions.

The floating-point spill and fill instructions (`stf.spill`, `ldf.fill`) are defined to save/restore a floating-point register (saved as 16 bytes) without surfacing an exception if the FR contains a NaTVal (these instructions do not affect the UNAT register).

The general and floating-point spill/fill instructions allow spilling/filling of registers that are targets of a speculative instruction and may therefore contain a deferred exception token. Note also that transfers between the general and floating-point register files cause a conversion between the two deferred exception token formats.

Table 4-14 lists the state relating to control speculation. Table 4-15 summarizes the instructions related to control speculation.

Table 4-14. State Related to Control Speculation

Register	Description
NaT bits	65th bit associated with each GR indicating a deferred exception
NaTVal	Pseudo-Zero encoding for FR indicating a deferred exception
UNAT	User NaT collection application register

Table 4-15. Instructions Related to Control Speculation

Mnemonic	Operation
`ld.s, ldf.s, ldfp.s`	GR and FR speculative loads
`ld8.fill, ldf.fill`	Fill GR with NaT collection, fill FR
`st8.spill, stf.spill`	Spill GR with NaT collection, spill FR
`chk.s`	Test GR or FR for deferred exception token
`tnat`	Test GR NaT bit and set predicate

4.4.5 Data Speculation

Just as control speculative loads and checks allow the compiler to schedule instructions across control dependencies, data speculative loads and checks allow the compiler to schedule instructions across some types of ambiguous data dependencies. This section details the usage model and semantics of data speculation and related instructions.

4.4.5.1 Data Speculation Concepts

An ambiguous memory dependency is said to exist between a store (or any operation that may update memory state) and a load when it cannot be statically determined whether the load and store might access overlapping regions of memory. For convenience, a store that cannot be statically disambiguated relative to a particular load is said to be ambiguous relative to that load. In such cases, the compiler cannot change the order in which the load and store instructions were originally specified in the program. To overcome this scheduling limitation, a special kind of load instruction called an advanced load can be scheduled to execute earlier than one or more stores that are ambiguous relative to that load.

As with control speculation, the compiler can also speculate operations that are dependent upon the advanced load and later insert a check instruction that will determine whether the speculation was successful or not. For data speculation, the check can be placed anywhere the original non-data speculative load could have been scheduled.

Thus, a data-speculative sequence of instructions consists of an advanced load, zero or more instructions dependent on the value of that load, and a check instruction. This means that any sequence of stores followed by a load can be transformed into an advanced load followed by a sequence of stores followed by a check. The decision to perform such a transformation is highly dependent upon the likelihood and cost of recovering from an unsuccessful data speculation.

4.4.5.2 Data Speculation and Instructions

Advanced loads are available in integer (`ld.a`), floating-point (`ldf.a`), and floating-point pair (`ldfp.a`) forms. When an advanced load is executed, it allocates an entry in a structure called the Advanced Load Address Table (ALAT). Later, when a corresponding check instruction is executed, the presence of an entry indicates that the data speculation succeeded; otherwise, the speculation failed and one of two kinds of compiler-generated recovery is performed:

1. The check load instruction (`ld.c`, `ldf.c`, or `ldfp.c`) is used for recovery when the only instruction scheduled before a store that is ambiguous relative to the advanced load is the advanced load itself. The check load searches the ALAT for a matching entry. If found, the speculation was successful; if a matching entry was not found, the speculation was unsuccessful and the check load reloads the correct value from memory. Figure 4-2 shows this transformation.

Figure 4-2. Data Speculation Recovery Using ld.c

Before Data Speculation	After Data Speculation

`// Other instructions`
`st8        [r4] = r12`
`ld8        r6 = [r8];;`
`add        r5 = r6, r7;;`
`st8        [r18] = r5`
	`ld8.a        r6 = [r8];; // Advanced load`
	`// Other instructions`
	`st8          [r4] = r12`
	`ld8.c.clr    r6 = [r8] // Check load`
	`add          r5 = r6, r7;;`
	`st8          [r18] = r5`

2. The advanced load check (`chk.a`) is used when an advanced load and several instructions that depend on the loaded value are scheduled before a store that is ambiguous relative to the advanced load. The advanced load check works like the

speculation check (chk.s) in that, if the speculation was successful, execution continues inline and no recovery is necessary; if speculation was unsuccessful, the chk.a branches to compiler-generated recovery code. The recovery code contains instructions that will re-execute all the work that was dependent on the failed data speculative load up to the point of the check instruction. As with the check load, the success of a data speculation using an advanced load check is determined by searching the ALAT for a matching entry. This transformation is shown in Figure 4-3.

Figure 4-3. Data Speculation Recovery Using chk.a

Before Data Speculation	After Data Speculation

`// Other instructions`
`st8           [r4] = r12`
`ld8           r6 = [r8];;`
`add           r5 = r6, r7;;`
`st8           [r18] = r5`
	`ld8.a         r6 = [r8];;`
	`// Other instructions`
	`add           r5 = r6, r7;;`
	`// Other instructions`
	`st8           [r4] = r12`
	`chk.a.clr     r6, recover`
	
	`back:`
	`st8           [r18] = r5`
	
	`// Somewhere else in program`
	`recover:`
	`ld8           r6 = [r8];;`
	`add           r5 = r6, r7`
	`br            back`

Recovery code may use either a normal or advanced load to obtain the correct value for the failed advanced load. An advanced load is used only when it is advantageous to have an ALAT entry reallocated after a failed speculation. The last instruction in the recovery code should branch to the instruction following the chk.a.

4.4.5.3 Detailed Functionality of the ALAT and Related Instructions

The ALAT is the structure that holds the state necessary for advanced loads and checks to operate correctly. The ALAT is searched in two different ways: by physical addresses and by ALAT register tags. An ALAT register tag is a unique number derived from the physical target register number and type in conjunction with other implementation-specific state. Implementation-specific state might include register stack wraparound information to distinguish one instance of a physical register that may have been spilled by the RSE from the current instance of that register, thus avoiding the need to purge the ALAT on all register stack wraparounds.

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT values being preserved across an instruction set transition. On entry to IA-32 instruction set, existing entries in the ALAT are ignored.

4.4.5.3.1 Allocating and Checking ALAT Entries

Advanced loads perform the following actions:

1. The ALAT register tag for the advanced load is computed. (For ldfp.a, a tag is computed only for the first target register.)
2. If an entry with a matching ALAT register tag exists, it is removed.

3. A new entry is allocated in the ALAT which contains the new ALAT register tag, the load access size, and a tag derived from the physical memory address. The insertion of the new ALAT entry must occur no later in visibility order than the load of the data.
4. The value at the address specified in the advanced load is loaded into the target register and, if specified, the base register is updated and an implicit prefetch is performed.

Since the success of a check is determined by finding a matching register tag in the ALAT, both the `chk.a` and the target register of a `ld.c` must specify the same register as their corresponding advanced load. Additionally, the check load must use the same address and operand size as the corresponding advanced load; otherwise, the value written into the target register by the check load is undefined.

An advanced load check performs the following actions:

1. It looks for a matching ALAT entry and if found, falls through to the next instruction.
2. If no matching entry is found, the `chk.a` branches to the specified address.

An implementation may choose to implement a failing advanced load check directly as a branch or as a fault where the fault-handler emulates the branch. Although the expected mode of operation is for an implementation to detect matching entries in the ALAT during checks, an implementation may fail a check instruction even when an entry with a matching ALAT register tag exists. This will be a rare occurrence but software must not assume that the ALAT does not contain the entry.

A check load checks for a matching entry in the ALAT. If no matching entry is found, it reloads the value from memory and any faults that occur during the memory reference are raised. When a matching entry is found, there is flexibility in the actions that a processor can perform:

1. The implementation may choose to either leave the target register unchanged or to reload the value from memory.
2. If the implementation chooses to leave the target register unchanged and one or more exception conditions related to the data access or translation of the check load occurs, the implementation may choose to either raise the highest-priority of these faults or ignore them all and continue execution. The faults that can be ignored are those related to data access and translation (Data Nested TLB fault, Alternate Data TLB fault, VHPT Data fault, Data TLB fault, Data Page Not Present fault, Data NaT Page Consumption fault, Data Key Miss fault, Data Key Permission fault, Data Access Rights fault, Data Dirty Bit fault, Data Access Bit fault, Data Debug fault, Unaligned Data Reference fault, Unsupported Data Reference fault). See Table 5-6, "Interruption Priorities" on page 2:109.
3. If the implementation chooses to perform a reload, then any faults that occur because of the reload can not be ignored.
4. If the size, type, or address fields in the matching ALAT entry do not match that provided by a check load, the value returned by the check load is undefined. In such cases the implementation may choose to raise a fault or when the "no clear" variant of the check load is issued, an implementation may choose to update the address, size, or type fields of the matching ALAT entry or to leave the entry unchanged. The update of the ALAT entry must occur no later in visibility order

than the load of the data.

If the check load was an ordered check load (`ld.c.clr.acq`), then it is performed with the semantics of an ordered load (`ld.acq`). ALAT register tag lookups by advanced load checks and check loads are subject to memory ordering constraints as outlined in "Memory Access Ordering" on page 1:73.

In addition to the flexibility described above, the size, organization, matching algorithm, and replacement algorithm of the ALAT are implementation dependent. Thus, the success or failure of specific advanced loads and checks in a program may change: when the program is run on different processor implementations, within the execution of a single program on the same implementation, or between different runs on the same implementation.

4.4.5.3.2 Invalidating ALAT Entries

In addition to entries removed by advanced loads, ALAT entry invalidations can occur implicitly by events that alter memory state or explicitly by any of the following instructions: `ld.c.clr`, `ld.c.clr.acq`, `chk.a.clr`, `invala`, `invala.e`. Events that may implicitly invalidate ALAT entries include those that change memory state or memory translation state such as:

1. The execution of stores, semaphores, or `ptc.ga` on other processors in the coherence domain.
2. The execution of store or semaphore instructions issued on the local processor.
3. Platform-visible removal of a cache line from the processor's caches.

When one of these events occurs, hardware checks each memory region represented by an entry in the ALAT to see if it overlaps with the locations affected by the invalidation event. ALAT entries whose memory regions overlap with the invalidation event locations are removed. The invalidation of ALAT entries due to the execution of stores, semaphores or ptc.ga instructions must occur no later in visibility order than the store of the data or the TLB purge. Note that some invalidation events may require that multiple entries be removed from the ALAT. For example, the `ptc.ga` instruction is page aligned, thus a `ptc.ga` from another processor would require that hardware invalidate all ALAT entries related to that page. Stores due to RSE spills are not checked for ALAT invalidation and do not cause ALAT entries to be removed. See Section 6.9, "RSE and ALAT Interaction" on page 2:146. When an external agent can observe that the processor has removed a physical address range from its caches, then that address range is guaranteed to be invalidated from that processor's ALAT as well.

An implementation may invalidate entries over areas larger than explicitly required by a specific invalidation event, and more generally, to invalidate any ALAT entry at any time. For example, a `st1` only accesses one byte, but an implementation could choose to invalidate all ALAT entries whose memory region is in the same cache line. An implementation may also provide an ALAT with zero entries (i.e., all `ld.c`/`chk.a` instructions would act as if an ALAT miss had occurred).

Software is responsible for explicitly invalidating all affected ALAT entries whenever:

1. Software explicitly changes the virtual to physical register mapping on rotating registers that have been the target of advanced loads (`clrrrb`).
2. Software changes the virtual to physical memory mapping.

3. Software accesses the RSE backing store with advanced loads. See Section 6.9, "RSE and ALAT Interaction" on page 2:146 (since RSE stores do not invalidate ALAT entries).
4. Software explicitly changes the virtual to physical register mapping on stacked registers by switching the RSE backing stores. See Section 6.11.3, "Synchronous Backing Store Switch" on page 2:148.

4.4.5.4 Combining Control and Data Speculation

Control speculation and data speculation are not mutually exclusive; a given load may be both control and data speculative. Both control speculative (`ld.sa`, `ldf.sa`, `ldfp.sa`) and non-control speculative (`ld.a`, `ldf.a`, `ldfp.a`) variants of advanced loads are defined for general and floating-point registers. If a speculative advanced load generates a deferred exception token then:

1. Any existing ALAT entry with the same ALAT register tag is invalidated.
2. No new ALAT entry is allocated.
3. If the target of the load was a general-purpose register, its NaT bit is set.
4. If the target of the load was a floating-point register, then NaTVal is written to the target register.

If a speculative advanced load does not generate a deferred exception, then its behavior is the same as the corresponding non-control speculative advanced load.

Since there can be no matching entry in the ALAT after a deferred fault, a single advanced load check or check load is sufficient to check both for data speculation failures and to detect deferred exceptions.

4.4.5.5 Instruction Completers for ALAT Management

To help the compiler manage the allocation and deallocation of ALAT entries, two variants of advanced load checks and check loads are provided: variants with clear (`chk.a.clr`, `ld.c.clr`, `ld.c.clr.acq`, `ldf.c.clr`, `ldfp.c.clr`) and variants with no clear (`chk.a.nc`, `ld.c.nc`, `ldf.c.nc`, `ldfp.c.nc`).

The clear variants are used when the compiler knows that the ALAT entry will not be used again and wants the entry explicitly removed. This allows software to indicate when entries are unneeded, making it less likely that a useful entry will be unnecessarily forced out because all entries are currently allocated.

For the clear variants of check load, any ALAT entry with the same ALAT register tag is invalidated independently of whether the address or size fields of the check load and the corresponding advanced load match. For `chk.a.clr`, the entry is guaranteed to be invalidated only when the instruction falls through (the recovery code is not executed). Thus, a failing `chk.a.clr` may or may not clear any matching ALAT entries. In such cases, the recovery code must explicitly invalidate the entry in question if program correctness depends on the entry being absent after a failed `chk.a.clr`.

Non-clear variants of both kinds of data speculation checks act as a hint to the processor that an existing entry should be maintained in the ALAT or that a new entry should be allocated when a matching ALAT entry doesn't exist. Such variants can be used within loops to check advanced loads which were presumed loop-invariant and moved out of the loop by the compiler. This behavior ensures that if the check load fails on one iteration, then the check load will not necessarily fail on all subsequent iterations. Whenever a new entry is inserted into the ALAT or when the contents of an entry are updated, the information written into the ALAT only uses information from the check load and does not use any residual information from a prior entry. The non-clear variant of `chk.a`, `chk.a.nc`, does not allocate entries and the ‘`nc`’ completer acts as a hint to the processor that the entry should not be cleared.

Table 4-16 and Table 4-17 summarize state and instructions relating to data speculation.

Table 4-16. State Relating to Data Speculation

Structure	Function
ALAT	Advanced load address table

Table 4-17. Instructions Relating to Data Speculation

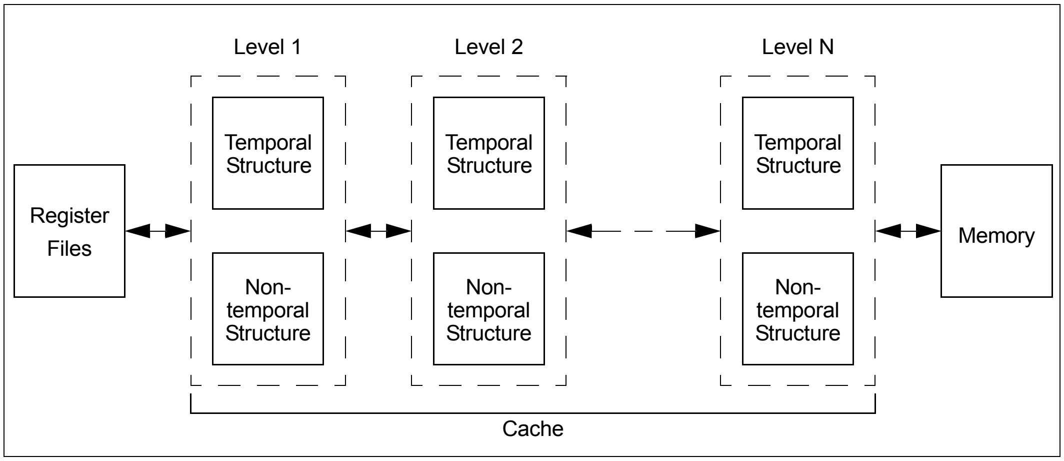
Mnemonic	Operation
`ld.a, ldf.a, ldfp.a`	GR and FR advanced load
`st, st.rel, st.spill, stf, stf.spill`	GR and FR store
`cmpxchg, fetchadd, xchg`	GR semaphore
`ld.c.clr, ld.c.clr.acq, ldf.c.clr, ldfp.c.clr`	GR and FR check load, clear on ALAT hit
`ld.c.nc, ldf.c.nc, ldfp.c.nc`	GR and FR check load, re-allocate on ALAT miss
`ld.sa, ldf.sa, ldfp.sa`	GR and FR speculative advanced load
`chk.a.clr, chk.a.nc`	GR and FR advanced load check
`invala`	Invalidate all ALAT entries
`invala.e`	Invalidate individual ALAT entry for GR or FR

4.4.6 Memory Hierarchy Control and Consistency

4.4.6.1 Hierarchy Control and Hints

Memory access instructions are defined to specify whether the data being accessed possesses temporal locality. In addition, memory access instructions can specify which levels of the memory hierarchy are affected by the access. This leads to an architectural view of the memory hierarchy depicted in Figure 4-1 composed of zero or more levels of cache between the register files and memory where each level may consist of two parallel structures: a temporal structure and a non-temporal structure. Note that this view applies to data accesses and not instruction accesses.

Figure 4-1. Memory Hierarchy



The temporal structures cache memory accessed with temporal locality; the non-temporal structures cache memory accessed without temporal locality. Both structures assume that memory accesses possess spatial locality. The existence of separate temporal and non-temporal structures, as well as the number of levels of cache, is implementation dependent. Please see the processor-specific documentation for further information.

Three mechanisms are defined for allocation control: locality hints; explicit prefetch; and implicit prefetch. Locality hints are specified by load, store, and explicit prefetch (`lfetch`) instructions. A locality hint specifies a hierarchy level (e.g., 1, 2, all). An access that is temporal with respect to a given hierarchy level is treated as temporal with respect to all lower (higher numbered) levels. An access that is non-temporal with respect to a given hierarchy level is treated as temporal with respect to all lower levels. Finding a cache line closer in the hierarchy than specified in the hint does not demote the line. This enables the precise management of lines using `lfetch` and then subsequent uses by`.nta` loads and stores to retain that level in the hierarchy. For example, specifying the`.nt2` hint by a prefetch indicates that the data should be cached at level 3. Subsequent loads and stores can specify`.nta` and have the data remain at level 3.

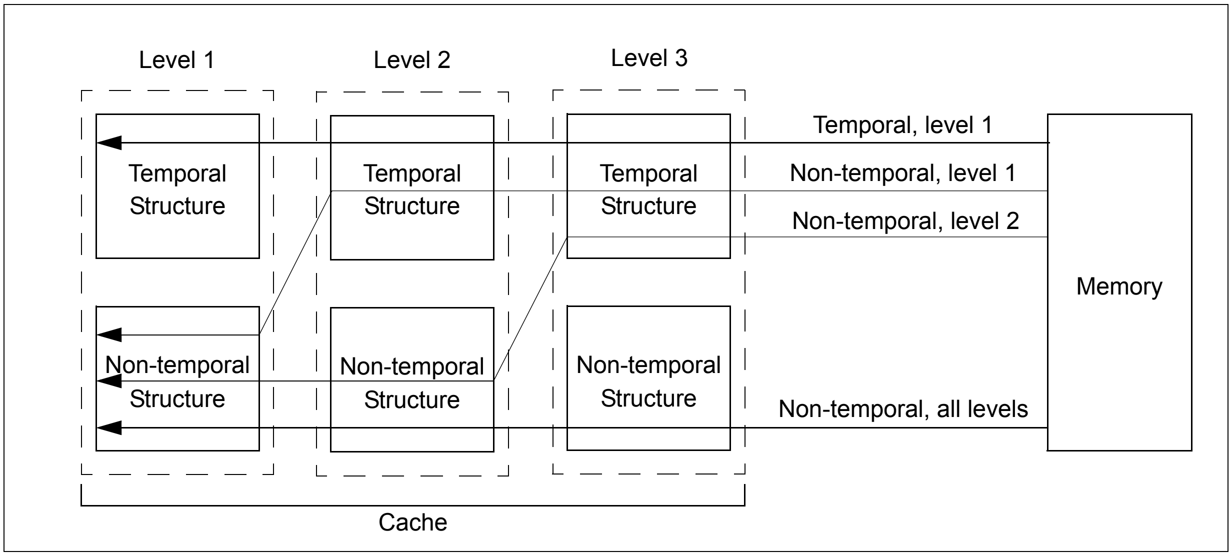
Locality hints do not affect the functional behavior of the program and may be ignored by the implementation. The locality hints available to loads, stores, and explicit prefetch instructions are given in Table 4-18. Instruction accesses are considered to possess both temporal and spatial locality with respect to level 1.

Table 4-18. Locality Hints Specified by Each Instruction Class

Mnemonic	Locality Hint	Instruction Type		
		Load	Store	lfetch, lfetch.fault
`none`	Temporal, level 1	x	x	x
nt1	Non-temporal, level 1	x		x
`nt2`	Non-temporal, level 2			x
`nta`	Non-temporal, all levels	x	x	x

Each locality hint implies a particular allocation path in the memory hierarchy. The allocation paths corresponding to the locality hints are depicted in Figure 4-2. The allocation path specifies the structures in which the line containing the data being referenced would best be allocated. If the line is already at the same or higher level in the hierarchy no movement occurs. Hinting that a datum should be cached in a temporal structure indicates that it is likely to be read in the near future. Hinting that a datum should not be cached in a temporal structure indicates that it is not likely to be read in the near future. For stores, the `.nta` completer also hints that the store may be part of a set of streaming stores that would likely overwrite the entire cache line without any data in that line first being read, enabling the processor to avoid fetching the data.

Figure 4-2. Allocation Paths Supported in the Memory Hierarchy



Explicit prefetch is defined in the form of the line prefetch instruction (`lfetch`, `lfetch.fault`). The lfetch instructions moves the line containing the addressed byte to a location in the memory hierarchy specified by the locality hint. If the line is already at the same or higher level in the hierarchy, no movement occurs. Both immediate and register post-increment are defined for `lfetch` and `lfetch.fault`. The `lfetch` instruction does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation. The `lfetch.fault` instruction affects the memory hierarchy in exactly the same way as `lfetch` but takes exceptions as if it were a 1-byte load instruction.

Implicit prefetch is based on the address post-increment of loads, stores, `lfetch` and `lfetch.fault`. The line containing the post-incremented address is moved in the memory hierarchy based on the locality hint of the originating load, store, `lfetch` or `lfetch.fault`. If the line is already at the same or higher level in the hierarchy then no movement occurs. Implicit prefetch does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation.

Another form of hint that can be provided on loads is the `ld.bias` load type. This is a hint to the implementation to acquire exclusive ownership of the line containing the addressed data. The bias hint does not affect program functionality and may be ignored by the implementation.

The following instructions are defined for flush control: flush cache (`fc`, `fc.i`) and flush write buffers (`fwb`). The `fc` instruction invalidates the cache line in all levels of the memory hierarchy above memory. If the cache line is not consistent with memory, then it is copied into memory before invalidation. The `fc.i` instruction ensures the data cache line associated with an address is coherent with the instruction caches. The `fc.i` instruction is not required to invalidate the targeted cache line nor write the targeted cache line back to memory if it is inconsistent with memory, but may do so if this is required to make the targeted cache line coherent with the instruction caches. The `fwb` instruction provides a hint to flush all pending buffered writes to memory (no indication of completion occurs).

Table 4-19 summarizes the memory hierarchy control instructions and hint mechanisms.

Table 4-19. Memory Hierarchy Control Instructions and Hint Mechanisms

Mnemonic	Operation
`.nt1` and`.nta` completer on loads	Load usage hints
`.nta` completer on stores	Store usage hints
Prefetch line at post-increment address on loads and stores	Prefetch hint
`lfetch, lfetch.fault` with `.nt1`,`.nt2`, and `.nta` hints	Prefetch line
`fc, fc.i`	Flush cache
`fwb`	Flush write buffers

4.4.6.2 Memory Consistency

In the Itanium architecture, instruction accesses made by a processor are not coherent with respect to instruction and/or data accesses made by any other processor, nor are instruction accesses made by a processor coherent with respect to data accesses made by that same processor. Therefore, hardware is not required to keep a processor's instruction caches consistent with respect to any processor's data caches, including that processor's own data caches; nor is hardware required to keep a processor's instruction caches consistent with respect to any other processor's instruction caches. Data accesses from different processors in the same coherence domain are coherent with respect to each other; this consistency is provided by the hardware. Data accesses from the same processor are subject to data dependency rules; see "Memory Access Ordering" below.

The mechanism(s) by which coherence is maintained is implementation dependent. Separate or unified structures for caching data and instructions are not architecturally visible. Within this context there are two categories of data memory hierarchy control: allocation and flush. Allocation refers to movement towards the processor in the hierarchy (lower numbered levels) and flush refers to movement away from the processor in the hierarchy (higher numbered levels). Allocation and flush occur in line-sized units; the minimum architecturally visible line size is 32 bytes (aligned on a 32-byte boundary). The line size in an implementation may be smaller in which case the implementation will need to move multiple lines for each allocation and flush event. An implementation may allocate and flush in units larger than 32 bytes.

In order to guarantee that a write from a given processor becomes visible to the instruction stream of that same, and other, processors, the affected line(s) must be made coherent with instruction caches. Software may use the `fc.i` instruction for this purpose. Memory updates by DMA devices are coherent with respect to instruction and data accesses of processors. The consistency between instruction and data caches of processors with respect to memory updates by DMA devices is provided by the hardware. In case a program modifies its own instructions, the `sync.i` and `srlz.i` instructions are used to ensure that prior coherency actions are observed by a given point in the program. Refer to the description `sync.i` on page 3:259 in *Volume 3: Intel® Itanium® Instruction Set Reference* for an example of self-modifying code.

4.4.7 Memory Access Ordering

Memory data access ordering must satisfy read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) data dependencies to the same memory location. In addition, memory writes and flushes must observe control dependencies. Except for these restrictions, reads, writes, and flushes may occur in an order different from the specified program order. Note that no ordering exists between instruction accesses and data accesses or between any two instruction accesses. The mechanisms described below are defined to enforce a particular memory access order. In the following discussion, the terms "previous" and "subsequent" are used to refer to the program specified order. The term "visible" is used to refer to all architecturally visible effects of performing a memory access (at a minimum this involves reading or writing memory).

Memory accesses follow one of four memory ordering semantics: unordered, release, acquire or fence. Unordered data accesses may become visible in any order. Release data accesses guarantee that all previous data accesses are made visible prior to being made visible themselves. Acquire data accesses guarantee that they are made visible prior to all subsequent data accesses. Fence operations combine the release and acquire semantics into a bi-directional fence, i.e., they guarantee that all previous data accesses are made visible prior to any subsequent data accesses being made visible.

Explicit memory ordering takes the form of a set of instructions: ordered load and ordered check load (`ld.acq`, `ld.c.clr.acq`), ordered store (`st.rel`), semaphores (`cmpxchg`, `xchg`, `fetchadd`), and memory fence (`mf`). The `ld.acq` and `ld.c.clr.acq` instructions follow acquire semantics. The `st.rel` follows release semantics. The `mf` instruction is a fence operation. The `xchg`, `fetchadd.acq`, and `cmpxchg.acq` instructions have acquire semantics. The `cmpxchg.rel`, and `fetchadd.rel` instructions have release semantics. The semaphore instructions also have implicit ordering. If there is a write, it will always follow the read. In addition, the read and write will be performed atomically with no intervening accesses to the same memory region.

Table 4-20 illustrates the ordering interactions between memory accesses with different ordering semantics. "O" indicates that the first and second reference are performed in order with respect to each other. A "-" indicates that no ordering is implied other than data dependencies (and control dependencies for writes and flushes).

Table 4-20. Memory Ordering Rules

First Reference	Second Reference			
	Fence	Acquire	Release	Unordered
fence	O	O	O	O
acquire	O	O	O	O
release	O	–	O	–
unordered	O	–	O	–

Table 4-21 summarizes memory ordering instructions related to cacheable memory. For definitions of the ordering rules related to non-cacheable memory, cache synchronization, and privileged instructions, refer to Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82.

Table 4-21. Memory Ordering Instructions

Mnemonic	Operation
`ld.acq, ld.c.clr.acq`	Ordered load and ordered check load
`st.rel`	Ordered store
`xchg`	Exchange memory and general register
`cmpxchg.acq, cmpxchg.rel`	Conditional exchange of memory and general register
`fetchadd.acq,fetchadd.rel`	Add immediate to memory
`mf`	Memory ordering fence

4.5 Branch Instructions

Branch instructions effect a transfer of control flow to a new address. Branch targets are bundle-aligned, which means control is always passed to the first instruction slot of the target bundle (slot 0). Branch instructions are not required to be the last instruction in an instruction group. In fact, an instruction group can contain arbitrarily many branches (provided that the normal RAW and WAW dependency requirements are met). If a branch is taken, only instructions up to the taken branch will be executed. After a taken branch, the next instruction executed will be at the target of the branch.

There are three categories of branches: IP-relative branches, long branches, and indirect branches. IP-relative branches specify their target with a signed 21-bit displacement, which is added to the IP of the bundle containing the branch to give the address of the target bundle. The displacement allows a branch reach of ±16MBytes. Long branches are IP-relative with a 60-bit displacement, allowing the target to be anywhere in the 64-bit address space. Because of the long immediate, long branches occupy two instruction slots. Indirect branches use the branch registers to specify the target address.

There are several branch types, as shown in Table 4-22. The conditional branch `br.cond` or `br` is a branch which is taken if the specified predicate is 1, and not-taken otherwise. The conditional call branch `br.call` does the same thing, and in addition, writes a link address to a specified branch register and adjusts the general register stack (see "Register Stack" on page 1:47). The conditional return `br.ret` does the same thing as an indirect conditional branch, plus it adjusts the general register stack. Unconditional branches, calls and returns are executed by specifying PR 0 (which is always 1) as the predicate for the branch instruction. The long branches, `brl.cond` or `brl`, and `brl.call` are identical to `br.cond` or `br`, and `br.call`, respectively, except for their longer displacement.

Table 4-22. Branch Types

Mnemonic	Function	Branch Condition	Target Address
`br.cond or br`	Conditional branch	Qualifying predicate	IP-rel or Indirect
`br.call`	Conditional procedure call	Qualifying predicate	IP-rel or Indirect
`br.ret`	Conditional procedure return	Qualifying predicate	Indirect

Table 4-22. Branch Types (Continued)

Mnemonic	Function	Branch Condition	Target Address
`br.ia`	Invoke the IA-32 instruction set	Unconditional	Indirect
`br.cloop`	Counted loop branch	Loop count	IP-rel
`br.ctop, br.cexit`	Modulo-scheduled counted loop	Loop count and Epilog count	IP-rel
`br.wtop, br.wexit`	Modulo-scheduled while loop	Qualifying predicate and Epilog count	IP-rel
`brl.cond or brl`	Long conditional branch	Qualifying predicate	IP-rel
`brl.call`	Long conditional procedure call	Qualifying predicate	IP-rel

The counted loop type (`br.cloop`) uses the Loop Count (LC) application register. If LC is non-zero then it is decremented and the branch is taken. If LC is zero, the branch falls through. The modulo-scheduled loop type branches (`br.ctop`, `br.cexit`, `br.wtop`, `br.wexit`) are described in "Modulo-scheduled Loop Support" on page 1:75. The loop type branches (`br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, `br.wexit`) are allowed only in slot 2 of a bundle. A loop type branch executed in slot 0 or 1 will cause an Illegal Operation fault.

Instructions are provided to move data between branch registers and general registers (`mov =br`, `mov br=`). Table 4-23 and Table 4-24 summarize state and instructions relating to branching.

Table 4-23. State Relating to Branching

Register	Function
BRs	Branch registers
PRs	Predicate registers
CFM	Current Frame Marker
PFS	Previous Function State application register
LC	Loop Count application register
EC	Epilog Count application register

Table 4-24. Instructions Relating to Branching

Mnemonic	Operation
`br`	Branch
`brl`	Long branch
`brp`	Provide early hint information about a future branch
`mov =br`	Move from BR to GR
`mov br=`	Move from GR to BR

4.5.1 Modulo-scheduled Loop Support

Support for software-pipelined loops is provided through rotating registers and loop branch types. Software pipelining of a loop is analogous to hardware pipelining of a functional unit. The loop body is partitioned into multiple "stages" with zero or more instructions in each stage. Modulo-scheduled loops have three phases: prolog, kernel, and epilog. During the prolog phase, new loop iterations are started each time around (filling the software pipeline). During the kernel phase, the pipeline is full. A new loop iteration is started, and another is finished each time around. During the epilog phase, no new iterations are started, but previous iterations are completed (draining the software pipeline).

A predicate is assigned to each stage to control the activation of the instructions in that stage (this predicate is called the "stage predicate"). To support the pipelining effect of stage predicates and registers in a software-pipelined loop, a fixed sized area of the predicate and floating-point register files (PR16-PR63 and FR32-FR127), and a programmable sized area of the general register file, are defined to "rotate." The size of the rotating area in the general register file is determined by an immediate in the `alloc` instruction. This immediate must be either zero or a multiple of 8. The general register rotating area is defined to start at GR32 and overlay the local and output areas, depending on their relative sizes. The stage predicates are allocated in the rotating area of the predicate register file. For counted loops, PR16 is architecturally defined to be the first stage predicate with subsequent stage predicates extending to higher predicate register numbers. For while loops, the first stage predicate may be any rotating predicate with subsequent stage predicates extending to higher predicate register numbers. Software is required to initialize the stage (rotating) predicates prior to entering the loop. An alloc instruction may not change the size of the rotating portion of the register stack frame unless all rotating register bases (rrb's) in the CFM are zero. All rrb's can be set to zero with the `clrrrb` instruction. The `clrrrb.pr` form can be used to clear just the rrb for the predicate registers. The `clrrrb` instruction must be the last instruction in an instruction group.

Rotation by one register position occurs when a software-pipelined loop type branch is executed. Registers are rotated towards larger register numbers in a wraparound fashion. For example, the value in register X will be located in register X+1 after one rotation. If X is the highest addressed rotating register its value will wrap to the lowest addressed rotating register. Rotation is implemented by renaming register numbers based on the value of a rotating register base (rrb) contained in CFM. An independent rrb is defined for each of the three rotating register files: CFM.rrb.gr for the general registers, CFM.rrb.fr for the floating-point registers, and CFM.rrb.pr for the predicate registers. General registers only rotate when the size of the rotating region is not equal to zero. Floating-point and predicate registers always rotate. When rotation occurs, two or all three rrb's are decremented in unison. Each rrb is decremented modulo the size of their respective rotating regions (e.g., 96 for rrb.fr). The operation of the rotating register rename mechanism is not otherwise visible to software. The instructions that modify the rrb's are listed in Table 4-25.

Table 4-25. Instructions that Modify RRBs

Mnemonic	Operation
`clrrrb`	Clears all rrb's
`clrrrb.pr`	Clears rrb.pr
`br.call, brl.call`	Clears all rrb's
`cover`	Clears all rrb's
`br.ret`	Restores CFM.rrb's from PFM.rrb's
`rfi`	Restores CFM.rrb's from IFM.rrb's if IFM.v==1

`br.ctop, br.cexit,`
`br.wtop, and br.wexit`
	Decrements all rrb's

There are two categories of software-pipelined loop branch types: counted and while. Both categories have two forms: top and exit. The "top" variant is used when the loop decision is located at the bottom of the loop body. A taken branch will continue the loop while a not-taken branch will exit the loop. The "exit" variant is used when the loop decision is located somewhere other than the bottom of the loop. A not-taken branch will continue the loop and a taken branch will exit the loop. The "exit" variant is also used at intermediate points in an unrolled pipelined loop.

The branch condition of a counted loop branch is determined by the specific counted loop type (ctop or cexit), the value of the loop count application register (LC), and the value of the epilog count application register (EC). Note that the counted loop branches do not use a qualifying predicate. LC is initialized to one less than the number of iterations for the counted loop and EC is initialized to the number of stages into which the loop body has been partitioned. While LC is greater than zero, the branch direction will continue the loop, LC will be decremented, registers will be rotated (rrb's are decremented), and PR 16 will be set to 1 after rotation. (For each of the loop-type branches, PR 63 is written by the branch, and after rotation this value will be in PR 16.)

Execution of a counted loop branch with LC equal to zero signals the start of the epilog. While in the epilog and while EC is greater than one, the branch direction will continue the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. Execution of a counted loop branch with LC equal to zero and EC equal to one signals the end of the loop; the branch direction will exit the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. A counted loop type branch executed with both LC and EC equal to zero will have a branch direction to exit the loop. LC, EC, and the rrb's will not be modified (no rotation) and PR 63 will be set to 0. LC and EC equal to zero can occur in some types of optimized, unrolled software-pipelined loops if the target of a cexit branch is set to the next sequential bundle and the loop trip count is not evenly divisible by the unroll amount.

The direction of a while loop branch is determined by the specific while loop type (wtop or wexit), the value of the qualifying predicate, and the value of EC. The while loop branches do not use LC. While the qualifying predicate is one, the branch direction will continue the loop, registers will be rotated, and PR 16 will be set to 0 after rotation. While the qualifying predicate is zero and EC is greater than one, the branch direction will continue the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. The qualifying predicate is one during the kernel and zero during the epilog. During the prolog, the qualifying predicate may be zero or one depending upon the scheme used to program the pipelined while loop. Execution of a while loop branch with qualifying predicate equal to zero and EC equal to one signals the end of the loop; the branch direction will exit the loop, EC will be decremented, registers will be rotated, and PR 16 will be set to 0 after rotation. A while loop branch executed with a zero qualifying predicate and with EC equal to zero has a branch direction to exit the loop. EC and the rrb's will not be modified (no rotation) and PR 63 will be set to 0.

For while loops, the initialization of EC depends upon the scheme used to program the pipelined while loop. Often, the first valid condition for the while loop branch is not computed until several stages into the prolog. Therefore, software pipelines for while loops often have several speculative prolog stages. During these stages, the qualifying predicate can be set to zero or one depending upon the scheme used to program the loop. If the qualifying predicate is one throughout the prolog, EC will be decremented only during the epilog phase and is initialized to one more than the number of epilog stages. If the qualifying predicate is zero during the speculative stages of the prolog, EC will be decremented during this part of the prolog, and the initialization value for EC is increased accordingly.

4.5.2 Branch Prediction Hints

Information about branch behavior can be provided to the processor to improve branch prediction. This information can be encoded in two ways: with branch hints as part of a branch instruction (referred to as hints), and with separate Branch Predict instructions (`brp`) where the entire instruction is hint information. Hints and `brp` instructions do not affect the functional behavior of the program and may be ignored by the processor.

Branch instructions can provide three types of hints:

- **Whether prediction strategy:** This describes (for COND, CALL and RET type branches) how the processor should predict the branch condition. (For the loop type branches, prediction is based on LC and EC.) The suggested strategies that can be hinted are shown in Table 4-26.

Table 4-26. Whether Prediction Hint on Branches

Completer	Strategy	Operation
`spnt`	Static Not-Taken	Ignore this branch, do not allocate prediction resources for this branch.
`sptk`	Static Taken	Always predict taken, do not allocate prediction resources for this branch.
`dpnt`	Dynamic Not-Taken	Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict not-taken.
`dptk`	Dynamic Taken	Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict taken.

- **Sequential prefetch:** This indicates how much code the processor should prefetch at the branch target (shown in Table 4-27). Please see the processor-specific documentation for further information.

Table 4-27. Sequential Prefetch Hint on Branches

Completer	Sequential Prefetch Hint	Operation
`few`	Prefetch few lines	When prefetching code at the branch target, stop prefetching after a few (implementation-dependent number of) lines.
`many`	Prefetch many lines	When prefetching code at the branch target, prefetch more lines (also an implementation-dependent number).

- **Predictor deallocation:** This provides re-use information to allow the hardware to better manage branch prediction resources. Normally, prediction resources keep track of the most-recently executed branches. However, sometimes the most-recently executed branch is not useful to remember, either because it will not be re-visited any time soon or because a hint instruction will re-supply the information prior to re-visiting the branch. In such cases, this hint can be used to free up the prediction resources.

Table 4-28. Predictor Deallocation Hint

Completer	Operation
none	Don’t deallocate
`clr`	Deallocate branch information

4.5.3 Branch Predict Instructions

Branch predict instructions are entire instructions whose only purpose is to provide early information about future branches. Branch predict instructions provide the following pieces of information:

- **Location of the branch:** A displacement in the `brp` instruction added to the IP of the bundle containing the `brp` instruction gives the IP of the bundle containing the future branch.
- **Target of the branch:** IP-relative `brp` instructions specify the target of the future branch with a 21-bit displacement (just like in branches). The displacement plus the IP of the bundle containing the `brp` instruction gives the target address. Indirect `brp` instructions specify the branch register which will be used by the future branch.
- **Branch importance:** This hint indicates to hardware that it should employ a very fast (but small) prediction structure for this branch (useful on tight loops).
- **Whether prediction strategy:** Same as the strategy hint on branches, except that the available hints are slightly different. Static not-taken is not provided (it’s not useful to provide early indication of such branches), and only one form of Dynamic prediction is provided. Instead, two strategies are included to indicate that the branch will be a “positive” (CLOOP, CTOP, WTOP) or “negative” (CEXIT, WEXIT) loop-type.

The move to branch register instruction can also provide this same hint information, simplifying the setup for a hinted indirect branch.

4.6 Multimedia Instructions

Multimedia instructions (see Table 4-29) treat the general registers as concatenations of eight 8-bit, four 16-bit, or two 32-bit elements. They operate on each element independently and in parallel. The elements are always aligned on their natural boundaries within a general register. Most multimedia instructions are defined to operate on multiple element sizes. Three classes of multimedia instructions are defined: arithmetic, shift and data arrangement.

4.6.1 Parallel Arithmetic

There are three forms of parallel addition and subtraction: modulo (`padd`, `psub`), signed saturation (`padd.sss`, `psub.sss`), and unsigned saturation (`padd.uuu`, `padd.uus`, `psub.uuu`, `psub.uus`). The modulo forms have the result wraparound the largest or smallest representable value in the range of the result element. In the saturating forms, results larger than the largest representable value of the range of the result element, or smaller than the smallest representable value of the range, are clamped to the largest or smallest value in the range of the result element respectively. The signed saturation form treats both sources as signed and clamps the result to the limits of a signed range. The unsigned saturation form treats one source as unsigned and clamps the result to the limits of an unsigned range. Two variants are defined that treat the second source as either signed (`.uus`) or unsigned (`.uuu`).

The parallel average instruction (`pavg`, `pavg.raz`) adds corresponding elements from each source and right shifts each result by one bit. In the simple form of the instruction, the carry out of the most-significant bit of each sum is written into the most significant bit of the result element. In the round-away-from-zero form, a 1 is added to each sum before shifting. The parallel average subtract instruction (`pavgsub`) performs a similar operation on the difference of the sources.

The parallel shift left and add instruction (`pshladd`) performs a left shift on the elements of the first source and then adds them to the corresponding elements from the second source. Signed saturation is performed on both the shift and the add operations. The parallel shift right and add instruction (`pshradd`) is similar to `pshladd`. Both of these instructions are defined for 2-byte elements only.

The parallel compare instruction (`pcmp`) compares the corresponding elements of both sources and writes all ones (if true) or all zeroes (if false) into the corresponding elements of the target according to one of two relations (== or >).

The parallel multiply right instruction (`pmpy.r`) multiplies the corresponding two even-numbered signed 2-byte elements of both sources and writes the results into two 4-byte elements in the target. The `pmpy.l` instruction performs a similar operation on odd-numbered 2-byte elements. The parallel multiply and shift right instruction (`pmpyshr`, `pmpyshr.u`) multiplies the corresponding 2-byte elements of both sources producing four 4-byte results. The 4-byte results are shifted right by 0, 7, 15, or 16 bits as specified by the instruction. The least-significant 2 bytes of the 4-byte shifted results are then stored in the target register.

The parallel sum of absolute difference instruction (`psad`) accumulates the absolute difference of corresponding 1-byte elements and writes the result in the target.

The parallel minimum (`pmin.u`, `pmin`) and the parallel maximum (`pmax.u`, `pmax`) instructions deliver the minimum or maximum, respectively, of the corresponding 1-byte or 2-byte elements in the target. The 1-byte elements are treated as unsigned values and the 2-byte elements are treated as signed values.

Table 4-29. Parallel Arithmetic Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
`padd`	Parallel modulo addition	x	x	x
`padd.sss`	Parallel addition with signed saturation	x	x	
`padd.uuu`, `padd.uus`	Parallel addition with unsigned saturation	x	x	
`psub`	Parallel modulo subtraction	x	x	x
`psub.sss`	Parallel subtraction with signed saturation	x	x	
`psub.uuu`, `psub.uus`	Parallel subtraction with unsigned saturation	x	x	
`pavg`	Parallel arithmetic average	x	x	
`pavg.raz`	Parallel arithmetic average with round away from zero	x	x	
`pavgsub`	Parallel average of a difference	x	x	

Table 4-29. Parallel Arithmetic Instructions (Continued)

Mnemonic	Operation	1-byte	2-byte	4-byte
pshladd	Parallel shift left and add with signed saturation		x	
pshradd	Parallel shift right and add with signed saturation		x	
pcmp	Parallel compare	x	x	x
pmpy.l	Parallel signed multiply of odd elements			x
pmpy.r	Parallel signed multiply of even elements			x
pmpyshr	Parallel signed multiply and shift right		x	
pmpyshr.u	Parallel unsigned multiply and shift right		x	
psad	Parallel sum of absolute difference	x		
pmin	Parallel minimum	x	x	
pmax	Parallel maximum	x	x	

4.6.2 Parallel Shifts

The parallel shift left instruction (pshl) individually shifts each element of the first source by a count contained in either a general register or an immediate. The parallel shift right instruction (pshr) performs an individual arithmetic right shift of each element of one source by a count contained in either a general register or an immediate. The pshr.u instruction performs an unsigned right shift. Table 4-30 summarizes the types of parallel shift instructions.

Table 4-30. Parallel Shift Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
pshl	Parallel shift left		x	x
pshr	Parallel signed shift right		x	x
pshr.u	Parallel unsigned shift right		x	x

4.6.3 Data Arrangement

The mix right instruction (mix.r) interleaves the even-numbered elements from both sources into the target. The mix left instruction (mix.l) interleaves the odd-numbered elements. The unpack low instruction (unpack.l) interleaves the elements in the least-significant 4 bytes of each source into the target register. The unpack high instruction (unpack.h) interleaves elements from the most significant 4 bytes. The pack instructions (pack.sss, pack.uss) convert from 32-bit or 16-bit elements to 16-bit or 8-bit elements respectively. The least-significant half of larger elements in both sources are extracted and written into smaller elements in the target register. The pack.sss instruction treats the extracted elements as signed values and performs signed saturation on them. The pack.uss instruction performs unsigned saturation. The mux instruction (mux) copies individual 2-byte or 1-byte elements in the source to arbitrary positions in the target according to a specified function. For 2-byte elements, an 8-bit immediate allows all possible permutations to be specified. For 1-byte elements the copy function is selected from one of five possibilities (reverse, mix, shuffle, alternate, broadcast). Table 4-31 describes the various types of parallel data arrangement instructions.

Table 4-31. Parallel Data Arrangement Instructions

Mnemonic	Operation	1-byte	2-byte	4-byte
`mix.l`	Interleave odd elements from both sources	x	x	x
`mix.r`	Interleave even elements from both sources	x	x	x
`mux`	Arbitrary copy of individual source elements	x	x	
`pack.sss`	Convert from larger to smaller elements with signed saturation		x	x
`pack.uss`	Convert from larger to smaller elements with unsigned saturation		x	
`unpack.l`	Interleave least-significant elements from both sources	x	x	x
`unpack.h`	Interleave most significant elements from both sources	x	x	x

4.7 Register File Transfers

Table 4-32 shows the instructions defined to move values between the general register file and the floating-point, branch, predicate, performance monitor, processor identification, and application register files. Several of the transfer instructions share the same mnemonic (`mov`). The value of the operand identifies which register file is accessed.

Table 4-32. Register File Transfer Instructions

Mnemonic	Operation
`getf.exp, getf.sig`	Move FR exponent or significand to GR
`getf.s, getf.d`	Move single/double precision memory format from FR to GR
`setf.s, setf.d`	Move single/double precision memory format from GR to FR
`setf.exp, setf.sig`	Move from GR to FR exponent or significand
`mov =br`	Move from BR to GR
`mov br=`	Move from GR to BR
`mov =pr`	Move from predicates to GR
`mov pr=, mov pr.rot=`	Move from GR to predicates
`mov ar=`	Move from GR to AR
`mov =ar`	Move from AR to GR
`mov =psr.um`	Move from user mask to GR
`mov psr.um=`	Move from GR to user mask
`sum, rum`	Set and reset user mask
`mov =pmd[...]`	Move from performance monitor data register to GR
`mov =cpuid[...]`	Move from processor identification register to GR
`mov =ip`	Move from Instruction Pointer

Memory access instructions only target or source the general and floating-point register files. It is necessary to use the general register file as an intermediary for transfers between memory and all other register files except the floating-point register file.

Two classes of move are defined between the general registers and the floating-point registers. The first type moves the significand or the sign/exponent (`getf.sig`, `setf.sig`, `getf.exp`, `setf.exp`). The second type moves entire single or double precision numbers (`getf.s`, `setf.s`, `getf.d`, `setf.d`). These instructions also perform a conversion between the deferred exception token formats.

Instructions are provided to transfer between the branch registers and the general registers. The move to branch register instruction can also optionally include branch hints. See "Branch Prediction Hints" on page 1:78.

Instructions are defined to transfer between the predicate register file and a general register. These instructions operate in a "broadside" manner whereby multiple predicate registers are transferred in parallel (predicate register N is transferred to and from bit N of a general register). The move to predicate instruction (`mov pr=`) transfers a general register to multiple predicate registers according to a mask specified by an immediate. The mask contains one bit for each of the static predicate registers (PR 1 through PR 15 – PR 0 is hardwired to 1) and one bit for all of the rotating predicates (PR 16 through PR63). A predicate register is written from the corresponding bit in a general register if the corresponding mask bit is set. If the mask bit is clear then the predicate register is not modified. The rotating predicates are transferred as if CFM.rrb.pr were zero. The actual value in CFM.rrb.pr is ignored and remains unchanged. The move from predicate instruction (`mov =pr`) transfers the entire predicate register file into a general register target.

In addition, instructions are defined to move values between the general register file and the user mask (`mov psr.um=` and `mov =psr.um`). The `sum` and `rum` instructions set and reset the user mask. The user mask is the non-privileged subset of the Process Status Register (PSR).

The `mov =pmd[]` instruction is defined to move from a performance monitor data (PMD) register to a general register. If the operating system has not enabled reading of performance monitor data registers in user level then all zeroes are returned. The `mov =cpuid[]` instruction is defined to move from a processor identification register to a general register.

The `mov =ip` instruction is provided for copying the current value of the instruction pointer (IP) into a general register.

4.8 Character and Bit Strings

A small set of special instructions accelerate operations on character and bit-field data.

4.8.1 Character Strings

The compute zero index instructions (`czx.l`, `czx.r`) treat the general register source as either eight 1-byte or four 2-byte elements and write the general register target with the index of the first zero element found. If there are no zero elements in the source, the target is written with a constant one higher than the largest possible index (8 for the 1-byte form, 4 for the 2-byte form). The `czx.l` instruction scans the source from left to right with the left-most element having an index of zero. The `czx.r` instruction scans from right to left with the right-most element having an index of zero. Table 4-33 summarizes the compute zero index instructions.

Table 4-33. String Support Instructions

Mnemonic	Operation	1-byte	2-byte
czx.l	Locate first zero element, left to right	x	x
czx.r	Locate first zero element, right to left	x	x

4.8.2 Bit Strings

The population count instruction (popcnt) writes the number of bits that have a value of 1 in the source register into the target register. The count leading zeros instruction (clz) writes the number of leading zero bits in the source register into the target register; coupled with complement, clz can also perform count leading ones functionality as well.

Table 4-34. Bit Support Instructions

Mnemonic	Operation
popcnt	Count number of ones in source register
clz	Count number of leading zeros in source register

4.9 Privilege Level Transfer

Three instructions may cause a privilege level change: break (break), enter privileged code (epc) and branch return (br.ret). The break instruction is defined to cause a Break Instruction fault which can be used to transfer privilege levels. The break instruction contains an immediate which is made available to a dedicated fault handler. The epc instruction increases the privilege level without causing an interruption or a control flow transfer. The new privilege level is specified by the TLB entry for the page containing the epc, if virtual address translation for instruction fetches is enabled. If the privilege level specified by PFS.ppl (in the Previous Function State application register) is lower than the current privilege level (as specified by PSR.cpl in the Processor Status Register) epc raises an Illegal Operation fault. The br.ret instruction is defined to demote the privilege level if PFS.ppl is lower than PSR.cpl. A br.ret will never increase privilege level.

§

Floating-point Programming Model 5

The floating-point architecture is fully compliant with the ANSI/IEEE Standard for Binary Floating-Point Arithmetic (Std. 754-1985). There is full IEEE support for single, double, and double-extended real formats. The two IEEE methods for controlling rounding precision are supported. The first method converts results to the double-extended exponent range. The second method converts results to the destination precision. Some IEEE extensions such as fused multiply and add, minimum and maximum operations, and a register format with a larger range than the minimum double-extended format are also included.

5.1 Data Types and Formats

Six data types are supported directly: single, double, double-extended real (IEEE real types); 64-bit signed integer, 64-bit unsigned integer, and the 82-bit floating-point register format. A "Parallel FP" format where a pair of IEEE single precision values occupy a floating-point register's significand is also supported. A seventh data type, IEEE-style quad-precision, is supported by software routines. A future architecture extension may include additional support for the quad-precision real type.

5.1.1 Real Types

The parameters for the supported IEEE real types are summarized in Table 5-1.

Table 5-1. IEEE Real-type Properties

	Single	Double	Double-Extended	Quad-Precision
IEEE Real-Type Parameters				
Sign	+ or –	+ or –	+ or –	+ or –
E_{max}	+127	+1023	+16383	+16383
E_{min}	–126	–1022	–16382	–16382
Exponent bias	+127	+1023	+16383	+16383
Precision (bits)	24	53	64	113
IEEE Memory Formats				
Total memory format width (bits)	32	64	80	128
Sign field width (bits)	1	1	1	1
Exponent field width (bits)	8	11	15	15
Significand field width (bits)	23	52	64	112

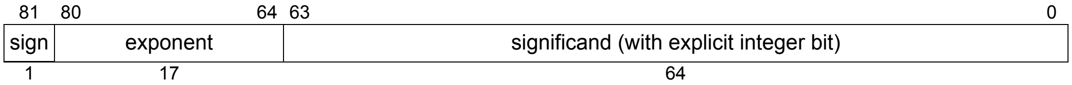
5.1.2 Floating-point Register Format

Data contained in the floating-point registers can be either integer or real type. The format of data in the floating-point registers is designed to accommodate both of these types with no loss of information.

Real numbers reside in 82-bit floating-point registers in a three-field binary format (see Figure 5-1). The three fields are:

- The 64-bit **significand** field, $b_{63}. b_{62}b_{61} .. b_1b_0$, contains the number's significant digits. This field is composed of an explicit integer bit (significand{63}), and 63 bits of fraction (significand{62:0}).
- The 17-bit **exponent** field locates the binary point within or beyond the significant digits (i.e., it determines the number's magnitude). The exponent field is biased by 65535 (0xFFFF). An exponent field of all ones is used to encode the special values for IEEE signed infinity and NaNs. An exponent field of all zeros and a significand field of all zeros is used to encode the special values for IEEE signed zeros. An exponent field of all zeros and a non-zero significand field encodes the double-extended real denormals and double-extended real pseudo-denormals.
- The 1-bit **sign** field indicates whether the number is positive (sign=0) or negative (sign=1).

Figure 5-1. Floating-point Register Format



The value of a finite floating-point number, encoded with non-zero exponent field, can be calculated using the expression:

$$(-1)^{(sign)} * 2^{(exponent - 65535)} * (significand\{63\}.significand\{62:0\}_2)$$

The value of a finite floating-point number, encoded with zero exponent field, can be calculated using the expression:

$$(-1)^{(sign)} * 2^{(-16382)} * (significand\{63\}.significand\{62:0\}_2)$$

Integers (64-bit signed/unsigned) and Parallel FP numbers reside in the 64-bit significand field. In their canonical form, the exponent field is set to 0x1003E (biased 63) and the sign field is set to 0.

5.1.3 Representation of Values in Floating-point Registers

The floating-point register encodings are grouped into classes and subclasses and listed below in Table 5-2 (shaded encodings are unsupported). The last two table entries contain the values of the constant floating-point registers, FR 0 and FR 1. The constant value in FR 1 does not change for the parallel single precision instructions or for the integer multiply accumulate instruction.

Table 5-2. Floating-point Register Encodings

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (Explicit Integer Bit is Shown) (64-bits)
NaNs	0/1	0x1FFFF	1.000...01 through 1.111...11
Quiet NaNs	0/1	0x1FFFF	1.100...00 through 1.111...11
Quiet NaN Indefinite[a]	1	0x1FFFF	1.100...00
Signaling NaNs	0/1	0x1FFFF	1.000...01 through 1.011...11
Infinity	0/1	0x1FFFF	1.000...00

Table 5-2. Floating-point Register Encodings (Continued)

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (Explicit Integer Bit is Shown) (64-bits)
Pseudo-NaNs	0/1	0x1FFFF	0.000...01 through 0.111...11
Pseudo-Infinity	0/1	0x1FFFF	0.000...00
Normalized Numbers (Floating-point Register Format Normals)	0/1	0x00001 through 0x1FFFE	1.000...00 through 1.111...11
Integers or Parallel FP (large unsigned or negative signed integers)	0	0x1003E	1.000...00 through 1.111...11
Integer Indefinite[b]	0	0x1003E	1.000...00
IEEE Single Real Normals	0/1	0x0FF81 through 0x1007E	1.000...00...(40)0s through 1.111...11...(40)0s
IEEE Double Real Normals	0/1	0x0FC01 through 0x103FE	1.000...00...(11)0s through 1.111...11...(11)0s
IEEE Double-Extended Real Normals	0/1	0x0C001 through 0x13FFE	1.000...00 through 1.111...11
Normal numbers with the same value as Double-Extended Real Pseudo-Denormals	0/1	0x0C001	1.000...00 through 1.111...11
IA-32 Stack Single Real Normals (produced when the computation model is IA-32 Stack Single)	0/1	0x0C001 through 0x13FFE	1.000...00...(40)0s through 1.111...11...(40)0s
IA-32 Stack Double Real Normals (produced when the computation model is IA-32 Stack Double)	0/1	0x0C001 through 0x13FFE	1.000...00...(11)0s through 1.111...11...(11)0s
Unnormalized Numbers (Floating-point Register Format unnormalized numbers)	0/1	0x00000	0.000...01 through 1.111...11
		0x00001 through 0x1FFFE	0.000...01 through 0.111...11
		0x00001 through 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00
Integers or Parallel FP (positive signed/unsigned integers)	0	0x1003E	0.000...00 through 0.111...11
IEEE Single Real Denormals	0/1	0x0FF81	0.000...01...(40)0s through 0.111...11...(40)0s
IEEE Double Real Denormals	0/1	0x0FC01	0.000...01...(11)0s through 0.111...11...(11)0s
Register Format Denormals	0/1	0x00001	0.000...01 through 0.111...11
Unnormal numbers with the same value as IEEE Double-Extended Real Denormals	0/1	0x0C001	0.000...01 through 0.111...11
IEEE Double-Extended Real Denormals	0/1	0x00000	0.000...01 through 0.111...11
IA-32 Stack Single Real Denormals (produced when computation model is IA-32 Stack Single)	0/1	0x00000	0.000...01...(40)0s through 0.111...11...(40)0s

Table 5-2. Floating-point Register Encodings (Continued)

Class or Subclass	Sign (1 bit)	Biased Exponent (17-bits)	Significand i.bb...bb (Explicit Integer Bit is Shown) (64-bits)
IA-32 Stack Double Real Denormals (produced when computation model is IA-32 Stack Double)	0/1	0x00000	0.000...01...(11)0s through 0.111...11...(11)0s
Double-Extended Real Pseudo-Denormals (IA-32 stack and memory format)	0/1	0x00000	1.000...00 through 1.111...11
Pseudo-Zeros	0/1	0x00001 through 0x1FFFD	0.000...00
	1	0x1FFFE	0.000...00
NaTVal[c]	0	0x1FFFE	0.000...00
Zero	0/1	0x00000	0.000...00
FR 0 (positive zero)	0	0x00000	0.000...00
FR 1 (positive one)	0	0x0FFFF	1.000...00

[a] Created by a masked real invalid operation.

[b] Created by a masked integer invalid operation.

[c] Created by an unsuccessful speculative memory operation.

All register encodings are allowed as inputs to arithmetic operations. The result of an arithmetic operation is always the most normalized register format representation of the computed value, with the exponent range limited from Emin to Emax of the destination type, and the significand precision limited to the number of precision bits of the destination type. Computed values, such as zeros, infinities, and NaNs that are outside these bounds are represented by the corresponding unique register format encoding. Double-extended real denormal results are mapped to the register format exponent of 0x00000 (instead of 0x0C001). Unsupported encodings (Pseudo-NaNs and Pseudo-Infinities), Pseudo-zeros and Double-extended Real Pseudo-denormals are never produced as a result of an arithmetic operation.

Arithmetic on pseudo-zeros operates exactly as an equivalently signed zero, with one exception. Pseudo-zero multiplied by infinity returns the correctly signed infinity instead of an Invalid Operation Floating-Point Exception fault (and QNaN). Also, pseudo-zeros are classified as unnormalized numbers, not zeros.

5.2 Floating-point Status Register

The Floating-Point Status Register (FPSR) contains the dynamic control and status information for floating-point operations. There is one main set of control and status information (FPSR.sf0), and three alternate sets (FPSR.sf1, FPSR.sf2, FPSR.sf3). The FPSR layout is shown in Figure 5-2 and its fields are defined in Table 5-3. Table 5-4 gives the FPSR's status field description and Figure 5-3 shows their layout.

Figure 5-2. Floating-point Status Register Format

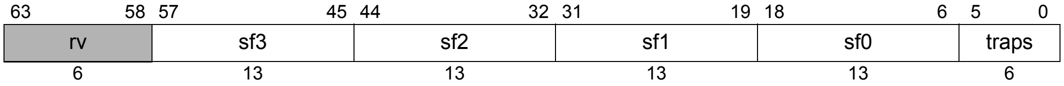


Table 5-3. Floating-point Status Register Field Description

Field	Bits	Description
traps.vd	0	Invalid Operation Floating-Point Exception fault (IEEE Trap) disabled when this bit is set
traps.dd	1	Denormal/Unnormal Operand Floating-Point Exception fault disabled when this bit is set
traps.zd	2	Zero Divide Floating-Point Exception fault (IEEE Trap) disabled when this bit is set
traps.od	3	Overflow Floating-Point Exception trap (IEEE Trap) disabled when this bit is set
traps.ud	4	Underflow Floating-Point Exception trap (IEEE Trap) disabled when this bit is set
traps.id	5	Inexact Floating-Point Exception trap (IEEE Trap) disabled when this bit is set
sf0	18:6	Main status field
sf1	31:19	Alternate status field 1
sf2	44:32	Alternate status field 2
sf3	57:45	Alternate status field 3
rv	63:58	Reserved

Figure 5-3. Floating-point Status Field Format

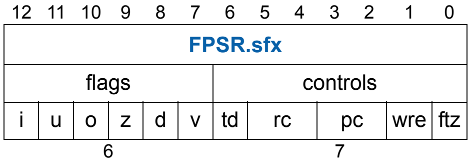


Table 5-4. Floating-point Status Register’s Status Field Description

Field	Bits	Description
ftz	0	Flush-to-Zero mode
wre	1	Widest range exponent (see Table 5-6)
pc	3:2	Precision control (see Table 5-6)
rc	5:4	Rounding control (see Table 5-5)
td	6	Traps disabled[a]
v	7	Invalid Operation (IEEE Flag)
d	8	Denormal/Unnormal Operand
z	9	Zero Divide (IEEE Flag)
o	10	Overflow (IEEE Flag)
u	11	Underflow (IEEE Flag)
i	12	Inexact (IEEE Flag)

[a] td is a reserved bit in the main status field, FPSR.sf0

The Denormal/Unnormal Operand status flag is an IEEE-style sticky flag that is set if the value is used in an arithmetic instruction and in an arithmetic calculation; e.g. unorm*NaN doesn’t set this flag. As depicted in Table 5-2 on page 1:86, canonical single/double/double-extended denormal, double-extended pseudo-denormal and register format denormal encodings are a subset of the floating-point register format unnormalized numbers.

Note: The Floating-Point Exception fault/trap occurs only if an enabled floating-point exception occurs during the processing of the instruction. Hence, setting a flag bit of a status field to 1 in software will not cause an interruption. The status fields flags are merely indications of the occurrence of floating-point exceptions.

Flush-to-Zero (FTZ) mode causes results which encounter “tininess” (see “Definition of Tininess, Inexact and Underflow” on page 1:106) to be truncated to the correctly signed zero. Flush-to-Zero mode can be enabled only if Underflow is disabled. If Underflow is enabled then it takes priority and Flush-to-Zero mode is ignored. Note that the software exception handler could examine the Flush-to-Zero mode bit and choose to emulate the Flush-to-Zero operation when an enabled Underflow exception arises. The FPSR.sf*x*.u and FPSR.sf*x*.i bits will be set to 1 when a result is flushed to the correctly signed zero because of Flush-to-Zero mode. If enabled, an inexact result exception is signaled.

A floating-point result is rounded based on the instruction’s.*pc* completer and the status field’s *wre*, *pc*, and *rc* control fields. The result’s significand precision and exponent range are determined as described in Table 5-6, “Floating-point Computation Model Control Definitions” on page 1:90. If the result isn’t exact, FPSR.sf*x*.*rc* specifies the rounding direction (see Table 5-5).

Table 5-5. Floating-point Rounding Control Definitions

	Nearest (or even)	- Infinity (down)	+ Infinity (up)	Zero (truncate/chop)
FPSR.sf*x*.rc	00	01	10	11

Table 5-6. Floating-point Computation Model Control Definitions

Computation Model Control Fields			Computation Model Selected		
Instruction’s.*pc* Completer	FPSR.sfx’s Dynamic *pc* Field	FPSR.sfx’s Dynamic *wre* Field	Significand Precision	Exponent Range	Computational Style
.s	ignored	0	24 bits	8 bits	IEEE real single
.d	ignored	0	53 bits	11 bits	IEEE real double
.s	ignored	1	24 bits	17 bits	Register format range, single precision
.d	ignored	1	53 bits	17 bits	Register format range, double precision
none	00	0	24 bits	15 bits	IA-32 stack single
none	01	0	N.A.	N.A.	Reserved
none	10	0	53 bits	15 bits	IA-32 stack double
none	11	0	64 bits	15 bits	IA-32 double-extended
none	00	1	24 bits	17 bits	Register format range, single precision
none	01	1	N.A.	N.A.	Reserved
none	10	1	53 bits	17 bits	Register format range, double precision
none	11	1	64 bits	17 bits	Register format range, double-extended precision
not applicable[a]	ignored	ignored	24 bits	8 bits	A pair of IEEE real singles
not applicable[b]	ignored	ignored	64 bits	17 bits	Register format range, double-extended precision

[a] For parallel FP instructions which have no.*pc* completer (e.g., fpma).

[b] For non-parallel FP instructions which have no.*pc* completer (e.g., frcpa).

The trap disable (sf*x*.td) control bit allows one to easily set up a local IEEE exception trap default environment. If FPSR.sf*x*.td is clear (enabled), the FPSR.traps bits are used. If FPSR.sf*x*.td is set, the FPSR.traps bits are treated as if they are all set (disabled). Note that FPSR.sf0.td is a reserved field which returns 0 when read.

5.3 Floating-point Instructions

This section describes the floating-point instructions. Refer to *Volume 3: Intel® Itanium® Instruction Set Reference* for a detailed description.

5.3.1 Memory Access Instructions

There are floating-point load and store instructions for the single, double, double-extended floating-point real data types, and the Parallel FP or signed/unsigned integer data type. The addressing modes for floating-point load and store instructions are the same as for integer load and store instructions, except for floating-point load pair instructions which can have an implicit base-register post increment. The memory hint options for floating-point load and store instructions are the same as those for integer load and store instructions. (See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69.) Table 5-7 lists the types of floating-point load and store instructions. The floating-point load pair instructions require the two target registers to be odd/even or even/odd. See "ldfp — Floating-point Load Pair" on page 3:161. The floating-point store instructions (`stfs`, `stfd`, `stfe`) require the value in the floating-point register to have the same type as the store for the format conversion to be correct.

Table 5-7. Floating-point Memory Access Instructions

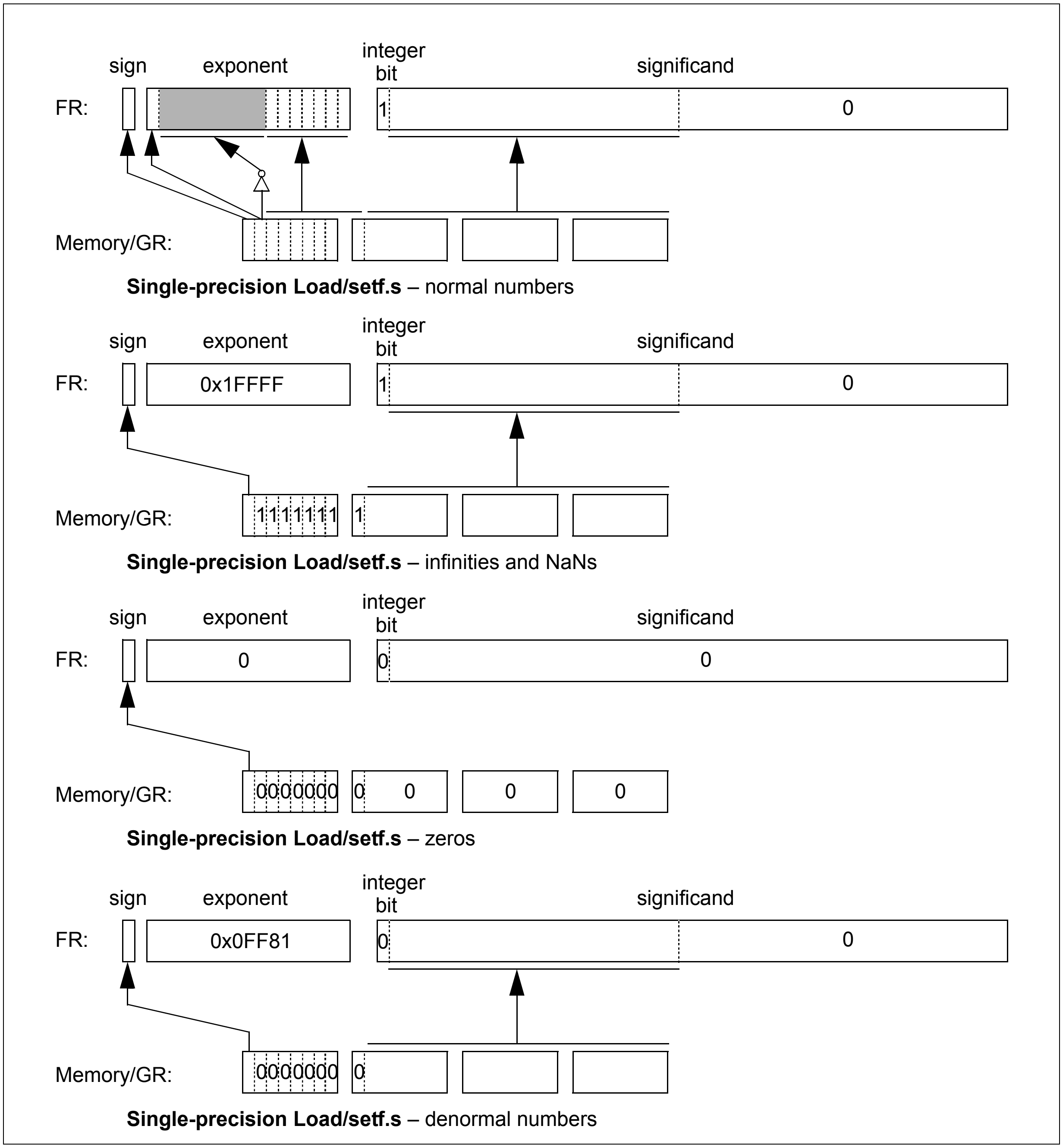
Operations	Load to FR	Load Pair to FR	Store from FR
Single	`ldfs`	`ldfps`	`stfs`
Integer/Parallel FP	`ldf8`	`ldfp8`	`stf8`
Double	`ldfd`	`ldfpd`	`stfd`
Double-extended	`ldfe`		`stfe`
Spill/fill	`ldf.fill`		`stf.spill`

Unsuccessful speculative loads write a NaTVal into the destination register or registers (see Section 4.4.4, "Control Speculation"). Storing a NaTVal to memory will cause a Register NaT Consumption fault, except for the spill instruction (`stf.spill`).

Saving and restoring floating-point registers is accomplished by the spill and fill instructions (`stf.spill`, `ldf.fill`) using a 16-byte memory container. These are the only instructions that can be used for saving and restoring the actual register contents since they do not fault on NaTVal. They save and restore all types (single, double, double-extended, register format and integer or Parallel FP) and will ensure compatibility with possible future architecture extensions.

Figure 5-4, Figure 5-5, Figure 5-6, Figure 5-7, Figure 5-8 and Figure 5-9 describe how single precision, double precision, double-extended precision, integer/parallel FP, and spill/fill data is translated during transfers between floating-point registers and memory.

Figure 5-4. Memory to Floating-point Register Data Translation – Single Precision



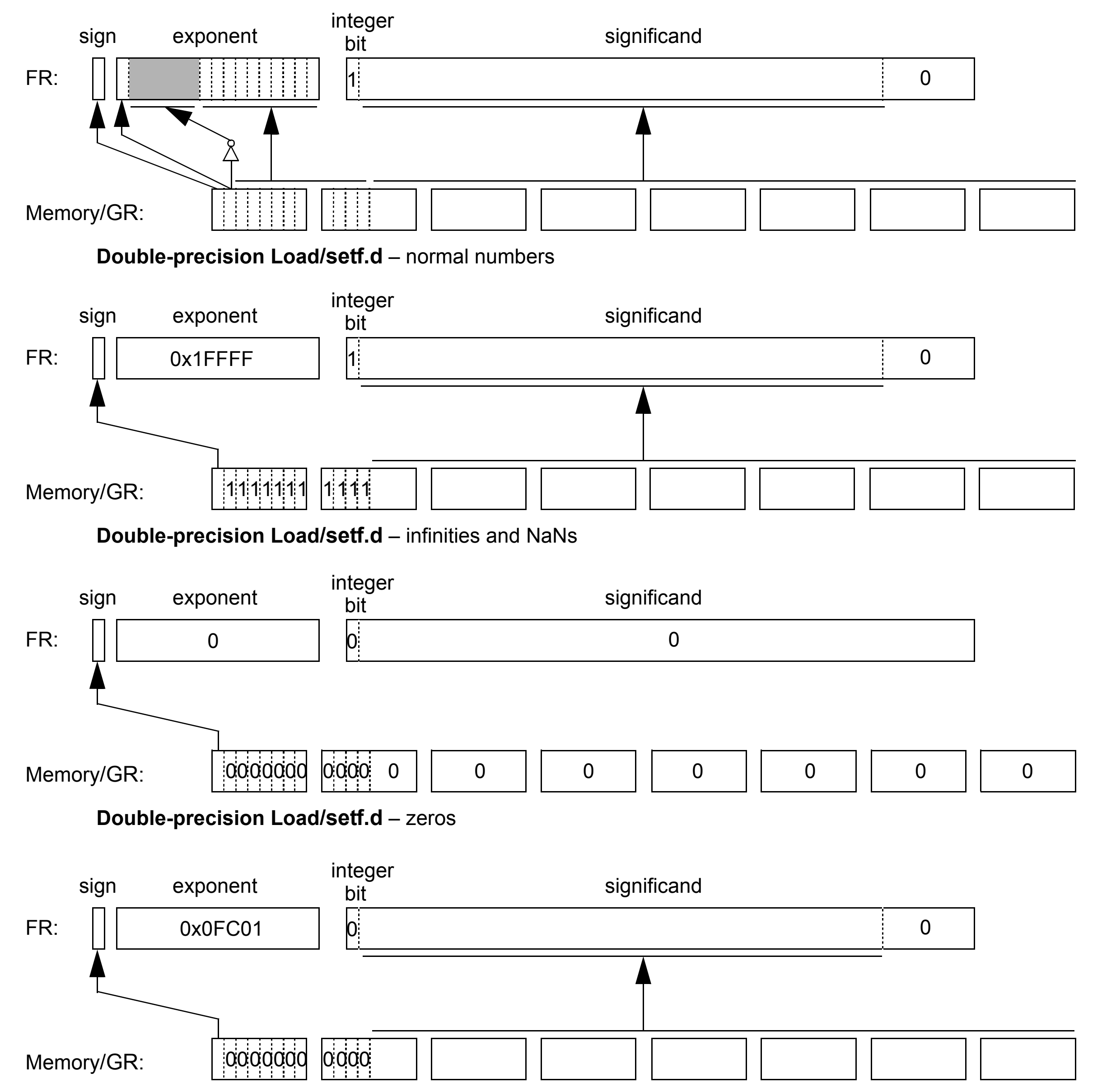
Single-precision Load/setf.s – normal numbers

Single-precision Load/setf.s – infinities and NaNs

Single-precision Load/setf.s – zeros

Single-precision Load/setf.s – denormal numbers

Figure 5-5. Memory to Floating-point Register Data Translation – Double Precision



Double-precision Load/setf.d – normal numbers

Double-precision Load/setf.d – infinities and NaNs

Double-precision Load/setf.d – zeros

Double-precision Load/setf.d – denormal numbers

Figure 5-6. Memory to Floating-point Register Data Translation – Double Extended, Integer, Parallel FP and Fill

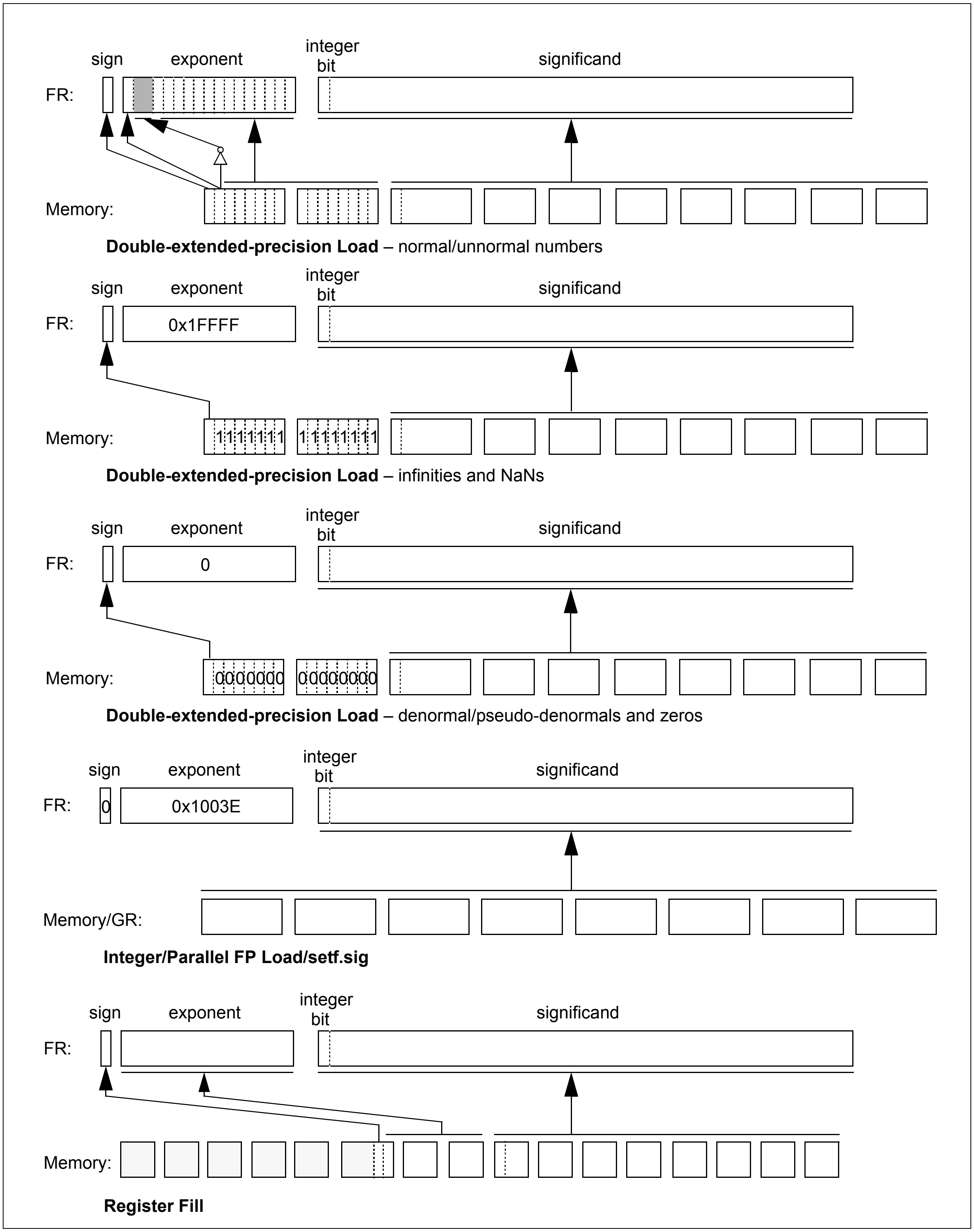


Figure 5-7. Floating-point Register to Memory Data Translation – Single Precision

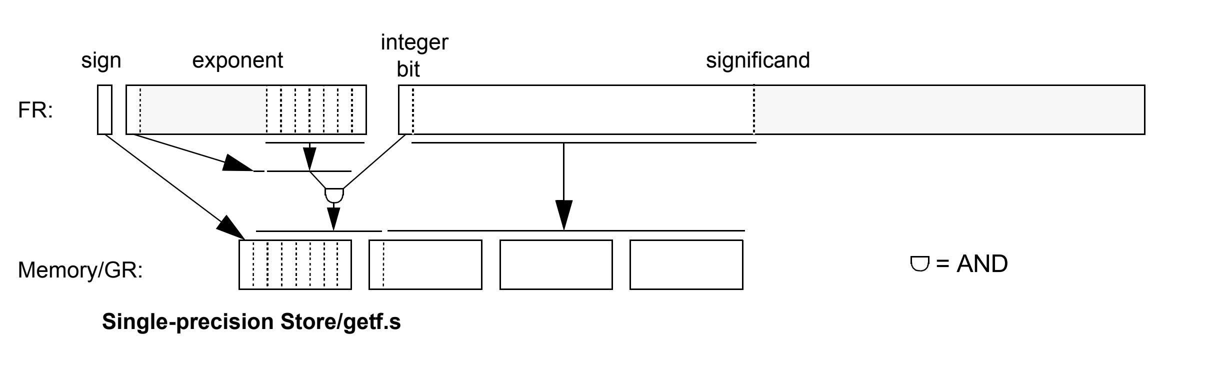


Figure 5-8. Floating-point Register to Memory Data Translation – Double Precision

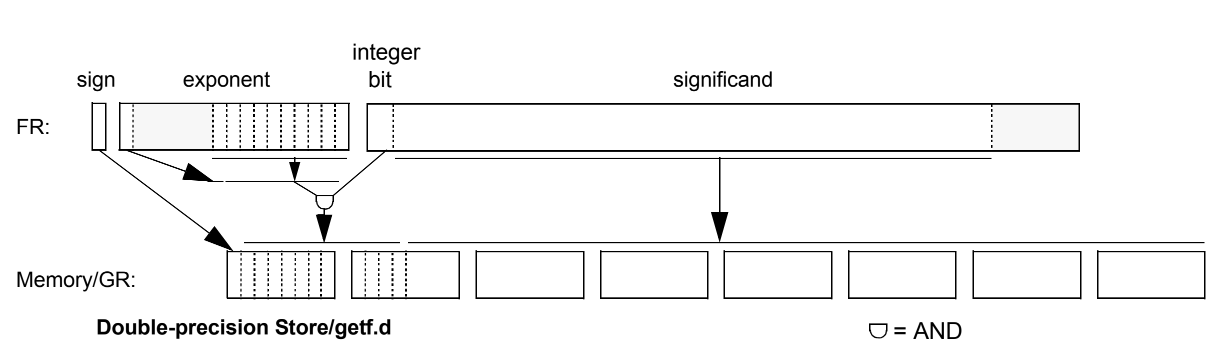
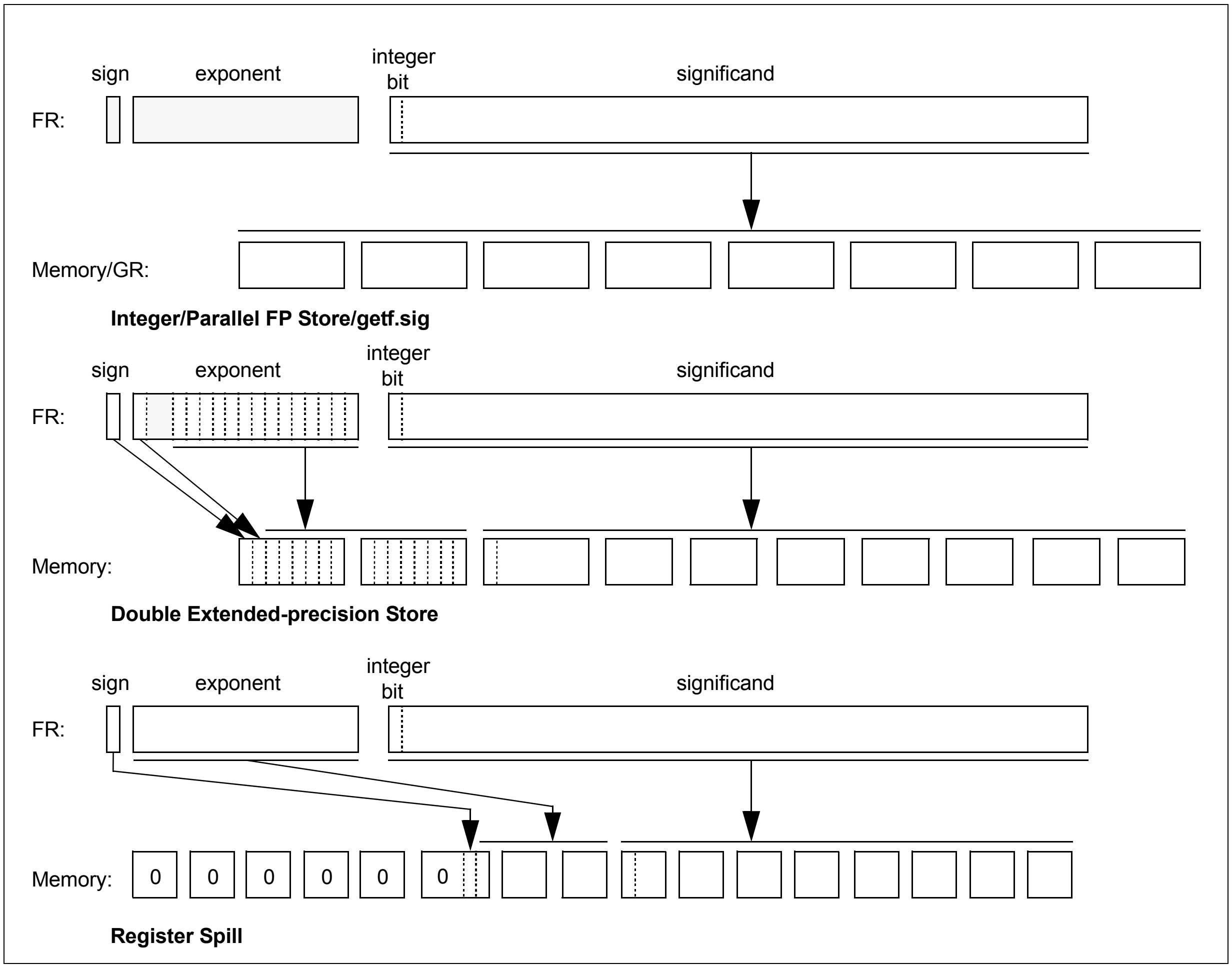
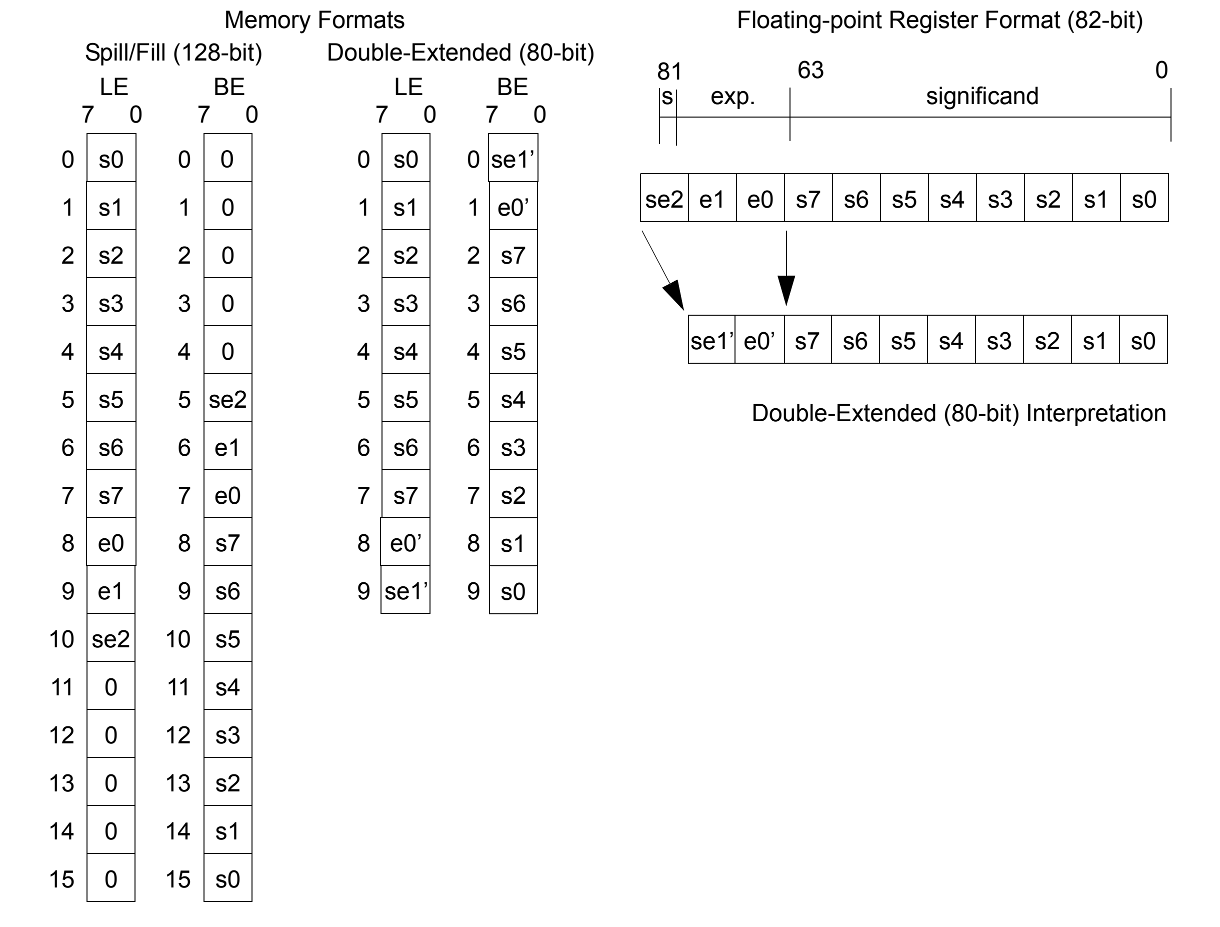


Figure 5-9. Floating-point Register to Memory Data Translation – Double Extended, Integer, Parallel FP and Spill



Both little-endian and big-endian byte ordering is supported on floating-point loads and stores. For both single and double memory formats, the byte ordering is identical to the 32-bit and 64-bit integer data types (see Section 3.2.3, "Byte Ordering"). The byte-ordering for the spill/fill memory and double-extended formats is shown in Figure 5-10.

Figure 5-10. Spill/Fill and Double-extended (80-bit) Floating-point Memory Formats



Memory Formats

	Spill/Fill (128-bit) LE 7 0	Spill/Fill (128-bit) BE 7 0	Double-Extended (80-bit) LE 7 0	Double-Extended (80-bit) BE 7 0
0	s0	0	s0	se1’
1	s1	0	s1	e0’
2	s2	0	s2	s7
3	s3	0	s3	s6
4	s4	0	s4	s5
5	s5	se2	s5	s4
6	s6	e1	s6	s3
7	s7	e0	s7	s2
8	e0	s7	e0’	s1
9	e1	s6	se1’	s0
10	se2	s5		
11	0	s4		
12	0	s3		
13	0	s2		
14	0	s1		
15	0	s0		

Floating-point Register Format (82-bit)
81
63
0
s
exp.
significand
se2 e1 e0 s7 s6 s5 s4 s3 s2 s1 s0
se1’ e0’ s7 s6 s5 s4 s3 s2 s1 s0
Double-Extended (80-bit) Interpretation

5.3.2 Floating-point Register to/from General Register Transfer Instructions

The `setf` and `getf` instructions (see Table 5-8) transfer data between floating-point registers (FR) and general registers (GR). These instructions will translate a general register NaT to/from a floating-point register NaTVal. For all other operands, the `.s` and `.d` variants of the `setf` and `getf` instructions translate to/from FR as per Figure 5-4, Figure 5-5, Figure 5-7 and Figure 5-8. The memory representation is read from or written to the GR. The `.exp` and `.sig` variants of the `setf` and `getf` instructions operate on the sign/exponent and significand portions of a floating-point register, respectively, and their translation formats are described in Table 5-9 and Table 5-10.

Table 5-8. Floating-point Register Transfer Instructions

Operations	GR to FR	FR to GR
Single	`setf.s`	`getf.s`
Double	`setf.d`	`getf.d`
Sign and Exponent	`setf.exp`	`getf.exp`
Significand/Integer	`setf.sig`	`getf.sig`

Table 5-9. General Register (Integer) to Floating-point Register Data Translation (setf)

	General Register		Floating-Point Register (.sig)			Floating-Point Register (.exp)		
Class	**NaT**	**Integer**	**Sign**	**Exponent**	**Significand**	**Sign**	**Exponent**	**Significand**
NaT	1	ignore	NaTVal			NaTVal		
integers	0	000...00 through 111...11	0	0x1003E	integer	integer{17}	integer{16:0}	0x8000000000000000

Table 5-10. Floating-point Register to General Register (Integer) Data Translation (getf)

	Floating-Point Register			General Register (.sig)		General Register (.exp)	
Class	**Sign**	**Exponent**	**Significand**	**NaT**	**Integer**	**NaT**	**Integer**
NaTVal	0	0x1FFFE	0.000...00	1	0x0000000000000000	1	0x1FFFE
integers or parallel FP	0	0x1003E	0.000...00 through 1.111...11	0	significand	0	0x1003E
other	any	any	any	0	significand	0	((sign<<17) | exponent)

5.3.3 Arithmetic Instructions

All arithmetic floating-point instructions, except fcvt.xf (which is always exact), have a.*sf* specifier. This indicates which of the four FPSR's status fields will both control and record the status of the execution of the instruction (see Table 5-11). The status field specifies: enabled exceptions, rounding mode, exponent width, precision control, and which status field's flags to update. See "Floating-point Status Register" on page 1:88.

Table 5-11. Floating-point Instruction Status Field Specifier Definition

.*sf* Specifier	.s0	.s1	.s2	.s3
Status field	FPSR.sf0	FPSR.sf1	FPSR.sf2	FPSR.sf3

Most arithmetic floating-point instructions can specify the precision and range of the result. The precision is determined either statically using a.*pc* completer or dynamically using the.*pc* field of the FPSR status field. The range is determined similarly except the.*wre* field of the FPSR status field is also used. Normal (non Parallel FP) arithmetic instructions that do not have a.*pc* completer use the floating-point register format precision and range. See Table 5-6 for details.

Table 5-12 lists the arithmetic floating-point instructions and Table 5-13 lists the arithmetic pseudo-operation definitions.

Table 5-12. Arithmetic Floating-point Instructions

Operation	Normal FP Mnemonic(s)	Parallel FP Mnemonic(s)
Floating-point multiply and add	fma.*pc*.*sf*	fpma.*sf*
Floating-point multiply and subtract	fms.*pc*.*sf*	fpms.*sf*
Floating-point negate multiply and add	fnma.*pc*.*sf*	fpnma.*sf*
Floating-point reciprocal approximation	frcpa.*sf*	fprcpa.*sf*
Floating-point reciprocal square root approximation	frsqrta.*sf*	fprsqrta.*sf*
Floating-point compare	fcmp.*frel*.*fctype*.*sf*	fpcmp.*frel*.*sf*

Table 5-12. Arithmetic Floating-point Instructions (Continued)

Floating-point minimum	fmin.*sf*	fpmin.*sf*
Floating-point maximum	fmax.*sf*	fpmax.*sf*
Floating-point absolute minimum	famin.*sf*	fpamin.*sf*
Floating-point absolute maximum	famax.*sf*	fpamax.*sf*
Convert floating-point to signed integer	fcvt.fx.*sf* fcvt.fx.trunc.*sf*	fpcvt.fx.*sf* fpcvt.fx.trunc.*sf*
Convert floating-point to unsigned integer	fcvt.fxu.*sf* fcvt.fxu.trunc.*sf*	fpcvt.fxu.*sf* fpcvt.fxu.trunc.*sf*
Convert signed integer to floating-point	fcvt.xf	N.A.

Table 5-13. Arithmetic Floating-point Pseudo-operations

Operation	Mnemonic	Operation Used
Floating-point multiplication (IEEE) Parallel FP multiplication	fmpy.*pc*.*sf* fpmpy.*sf*	fma, using FR 0 for addend fpma, using FR 0 for addend
Floating-point negate multiplication (IEEE) Parallel FP negate multiplication	fnmpy.*pc*.*sf* fpnmpy.*sf*	fnma, using FR 0 for addend fpnma, using FR 0 for addend
Floating-point addition (IEEE)	fadd.*pc*.*sf*	fma, using FR 1 for multiplicand
Floating-point subtraction (IEEE)	fsub.*pc*.*sf*	fms, using FR 1 for multiplicand
Floating-point normalization	fnorm.*pc*.*sf*	fma, using FR 1 for multiplicand and FR 0 for addend
Convert unsigned integer to floating-point	fcvt.xuf.pc.*sf*	fma, using FR 1 for multiplicand and FR 0 for addend

There are no pseudo-operations for Parallel FP addition, subtraction, negation or normalization since FR 1 does not contain a packed pair of single precision 1.0 values. A parallel FP addition can be performed by first forming a pair of 1.0 values in a register (using the fpack instruction) and then using the fpma instruction. Similarly, an integer add operation can be generated by first forming an integer 1 in a floating-point register (using the fcvt.fx instruction) and then using the xma instruction.

The fmpy pseudo-operation delivers the IEEE compliant result by rounding the product and without performing the addition inherent in the fma. An fma with the addend specified as a register other than FR 0, and containing the value +0.0, will not deliver the IEEE compliant multiply result in some cases.

5.3.4 Non-arithmetic Instructions

The non-arithmetic floating-point instructions always use the floating-point register (82-bit) precision since they do not have a.*pc* completer nor a.*sf* specifier.

The fclass instruction is used to classify the contents of a floating-point register. The fmerge instruction is used to merge data from two floating-point registers into one floating-point register. The fmix, fsxt, fpack, and fswap instructions are used to manipulate the Parallel FP data in the floating-point significand. The fand, fandcm, for, and fxor instructions are used to perform logical operations on the floating-point significand. The fselect instruction is used for conditional selects.

The `fneg` pseudo-operation (see Table 5-15) simply reverses the sign bit of the operand and is therefore not equivalent to the IEEE negation operation. For the IEEE negation operation, an `fnma` using FR 1 as the multiplicand and FR 0 as the addend must be used.

Table 5-14 lists the non-arithmetic floating-point instructions and Table 5-15 lists the non-arithmetic pseudo-operation definitions.

Table 5-14. Non-arithmetic Floating-point Instructions

Operation	Mnemonic(s)
Floating-point classify	`fclass.`*`fcrel.fctype`*
Floating-point merge sign Parallel FP merge sign	`fmerge.s` `fpmerge.s`
Floating-point merge negative sign Parallel FP merge negative sign	`fmerge.ns` `fpmerge.ns`
Floating-point merge sign and exponent Parallel FP merge sign and exponent	`fmerge.se` `fpmerge.se`
Floating-point mix left	`fmix.l`
Floating-point mix right	`fmix.r`
Floating-point mix left-right	`fmix.lr`
Floating-point sign-extend left	`fsxt.l`
Floating-point sign-extend right	`fsxt.r`
Floating-point pack	`fpack`
Floating-point swap	`fswap`
Floating-point swap and negate left	`fswap.nl`
Floating-point swap and negate right	`fswap.nr`
Floating-point And	`fand`
Floating-point And Complement	`fandcm`
Floating-point Or	`for`
Floating-point Xor	`fxor`
Floating-point Select	`fselect`

Table 5-15. Non-arithmetic Floating-point Pseudo-operations

Operation	Mnemonic	Operation Used
Floating-point absolute value Parallel FP absolute value	`fabs` `fpabs`	`fmerge.s`, with sign from FR 0 `fpmerge.s`, with sign from FR 0
Floating-point negate Parallel FP negate	`fneg` `fpneg`	`fmerge.ns` `fpmerge.ns`
Floating-point negate absolute value Parallel FP negate absolute value	`fnegabs` `fpnegabs`	`fmerge.ns`, with sign from FR 0 `fpmerge.ns`, with sign from FR 0

5.3.5 Floating-point Status Register (FPSR) Status Field Instructions

Speculation of floating-point operations requires that the status flags be stored temporarily in one of the alternate status fields (not FPSR.sf0). After a speculative execution chain has been committed, a `fchkf` instruction can be used to update the main status field flags (FPSR.sf0.flags). This operation will preserve the correctness of the IEEE flags. The `fchkf` instruction does this by comparing the flags of the status field with the FPSR.sf0.flags and FPSR.traps. If the flags of the alternate status field indicate the occurrence of an event that corresponds to an enabled floating-point exception in FPSR.traps, or an event that is not already registered in the FPSR.sf0.flags (i.e., the flag for that event in FPSR.sf0.flags is clear), then the `fchkf` instruction branches to recovery code. If neither of these cases arise then the `fchkf` instruction does nothing.

The `fsetc` instruction allows bit-wise modification of a status field's control bits. The FPSR.sf0.controls are ANDed with a 7-bit immediate and-mask and ORed with a 7-bit immediate or-mask to produce the control bits for the status field. The `fclrf` instruction clears all of the status field's flags to zero.

Table 5-16. FPSR Status Field Instructions

Operation	Mnemonic(s)
Floating-point check flags	`fchkf.`*sf*
Floating-point clear flags	`fclrf.`*sf*
Floating-point set controls	`fsetc.`*sf*

5.3.6 Integer Multiply and Add Instructions

Integer (fixed-point) multiply is executed in the floating-point unit using the three-operand `xma` instructions. The operands and result of these instructions are floating-point registers. The `xma` instructions ignore the sign and exponent fields of the floating-point register, except for a NaTVal check. The product of two 64-bit source significands is added to the third 64-bit significand (zero extended) to produce a 128-bit result. The low and high versions of the instruction select the appropriate low/high 64-bits of the 128-bit result, respectively, and write it into the destination register as a canonical integer. The signed and unsigned versions of the instructions treat the input multiplicands as signed and unsigned 64-bit integers respectively.

Table 5-17. Integer Multiply and Add Instructions

Integer Multiply and Add	Low	High
Signed	`xma.l`	`xma.h`
Unsigned	`xma.lu (pseudo-op)`	`xma.hu`

5.4 Additional IEEE Considerations

This section describes the support of the IEEE standard in the areas where specific details are left open to implementation.

5.4.1 Floating-point Interruptions

Floating-point interruptions are precise. The exception reporting and handling occurs on the instruction which causes the interruption. There are three floating-point interruptions: Disabled Floating-Point Register fault, Floating-Point Exception fault, and Floating-Point Exception trap (see Chapter 5, "Interruptions" in Volume 2 for more details).

Exceptions are processed according to a predetermined precedence. Precedence in exception handling means that higher-priority exceptions are flagged first and results are delivered according to the requirements of that exception. Lower-priority exceptions are not flagged even if they occur. For example, dividing an SNaN by zero causes an invalid operation exception (due to the SNaN) and not a zero-divide exception; the exception disabled result is the quieted version of the SNaN, not infinity. However, an IEEE Inexact Floating-Point Exception trap can accompany an IEEE Underflow or Overflow Floating-Point Exception trap.

For instructions that access the floating-point register file, the Disabled Floating-point Register fault has the highest priority.

5.4.1.1 Disabled Floating-point Register Fault

Two bits in the PSR, PSR.dfl and PSR.dfh, (see Section 3.3.2, "Processor Status Register (PSR)" on page 2:23) can be used by an operating system to enable or disable access to two subsets of floating-point registers: FR 2 to FR 31, and FR 32 to FR 127, respectively. The Disabled Floating-Point Register fault occurs when an access (read or write) is made to a FR which has been disabled. Operating systems can use this fault to identify a task as integer or floating-point and optimize the default set of registers which get saved on a task switch. If a mainly integer task is able to use only FR 2 to FR 32 for executing integer multiply and divide operations, then context switch time may be reduced by disabling access to the high floating-point registers.

5.4.1.2 Floating-point Exception Fault

A Floating-Point Exception fault occurs if one of the following four circumstances arises:

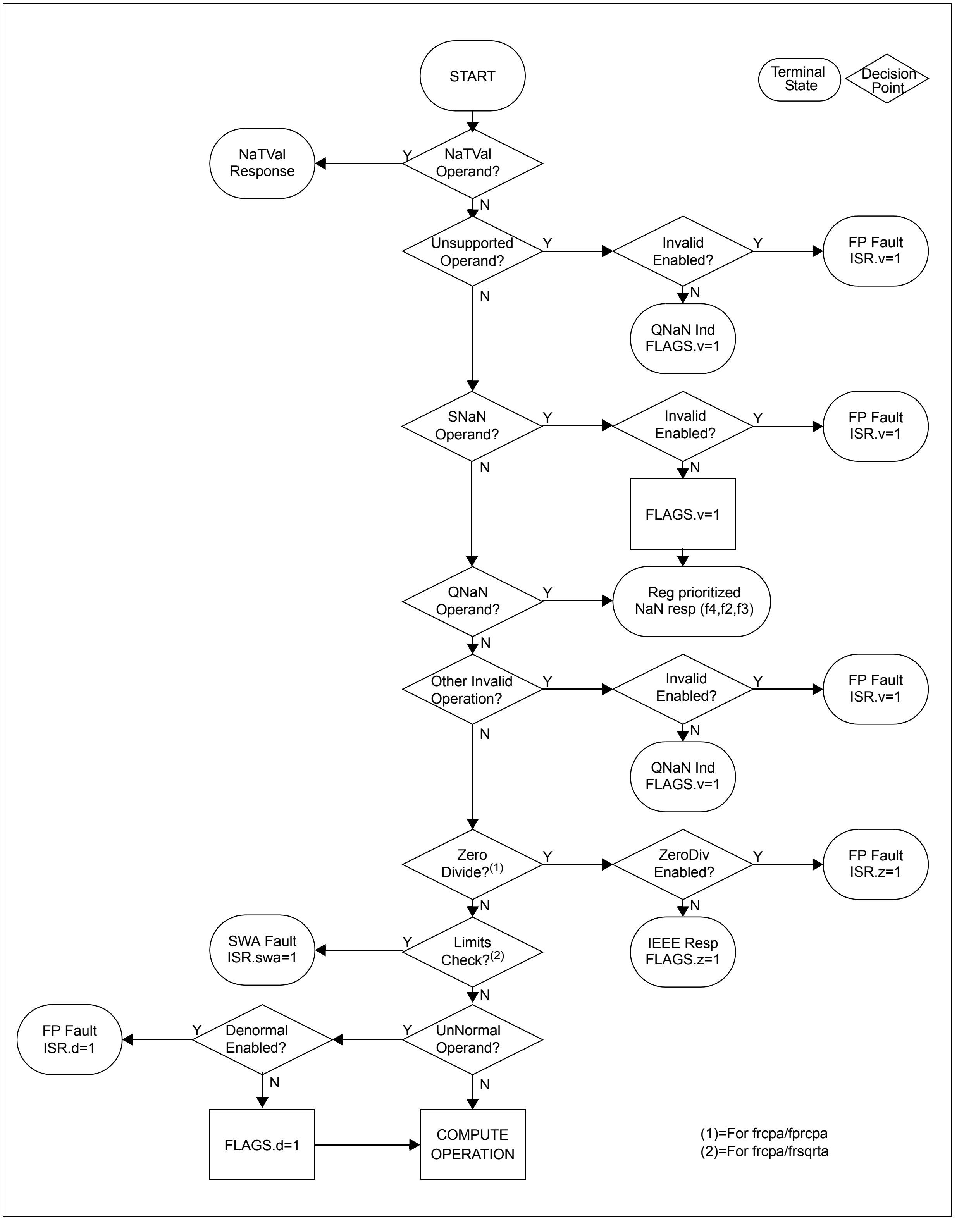
1. The processor requests system software assistance to complete the operation, via the Software Assist fault
2. The IEEE Invalid Operation trap is enabled and this condition occurs
3. The IEEE Zero Divide trap is enabled and this condition occurs
4. The Denormal/Unnormal Operand trap is enabled and an unnormalized operand (denormals are represented as unnormalized numbers in the register file) is encountered by a floating-point arithmetic instruction

If a Floating-Point Exception fault occurs, the only indication of which fault occurred is in the ISR.code. The appropriate status flags are not updated in the FPSR.

There is no requirement that the Software Assist Floating-Point Exception fault ever be signaled (except for certain operands in the `frcpa` and the `frsqrta` instructions), nor is there a mode to force its use. If there is no input NaTVal operand, a processor implementation may signal a Software Assist Floating-Point Exception fault at any time during the operation. In order to ensure maximum floating-point performance, most implementations will not use this exception except in difficult situations such as operations consuming denormal numbers.

The precedence among Floating-point Exception faults for arithmetic operations is depicted in Figure 5-11.

Figure 5-11. Floating-point Exception Fault Prioritization



5.4.1.3 Floating-point Exception Trap

A Floating-point Exception trap occurs if one of the following four circumstances arises:

1. The processor requests system software assistance to complete the operation, via the Software Assist trap
2. The IEEE Overflow trap is enabled and an overflow occurs
3. The IEEE Underflow trap is enabled and an underflow occurs
4. The IEEE Inexact trap is enabled and an inexact result occurs

When an overflow, underflow, or inexact result occurs, the appropriate status flags are updated in the FPSR. If enabled, a Floating-Point Exception trap occurs, and an indication of which enabled trap occurred is stored in ISR.code and the fpa bit in ISR.code (ISR{14}) is set as described in the next paragraph.

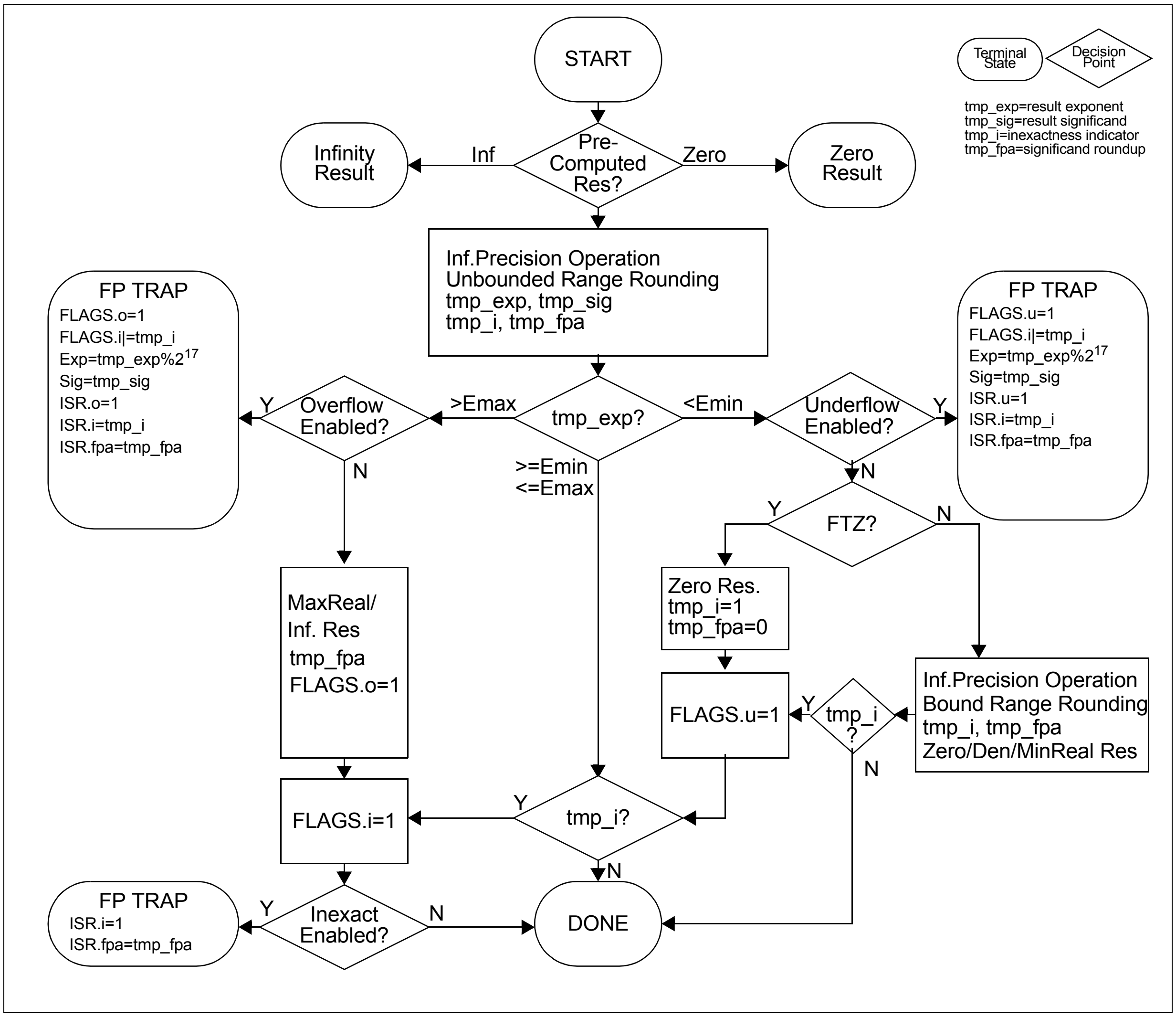
ISR.fpa is set to 1 when the magnitude of the delivered result is greater than the magnitude of the infinitely precise result. It is set to 0 otherwise. The magnitude of the delivered result may be greater if:

- The significand is incremented during rounding, or
- A larger pre-determined value (e.g., infinity) is substituted for the computed result (e.g., when overflow is disabled).

There is no requirement that the Software Assist Floating-Point Exception trap ever be signaled, nor is there a mode to force its use. In order to ensure maximum floating-point performance, most implementations will not use this exception except in difficult situations, such as operations creating denormal numbers. The occurrence of a Software Assist trap is indicated when a trap bit is set in ISR.code, but that trap is disabled. The destination register contains the trap enabled response for that trap.

The precedence among Floating-point Exception traps for arithmetic operations is depicted in Figure 5-12.

Figure 5-12. Floating-point Exception Trap Prioritization



5.4.2 Definition of Overflow

The overflow exception can occur whenever the rounded true result would exceed, in magnitude, the largest finite number in the destination format.

The IEEE Overflow Floating-Point Exception trap disabled response for all normal and Parallel-FP arithmetic instructions is to either return an infinity or the correctly signed maximum finite value for the destination precision. This depends on the rounding mode, the sign of the result, and the operation. An inexact result exception is signaled.

The IEEE Overflow Floating-Point Exception trap enabled response for all normal arithmetic instructions is to return the true biased exponent value MOD 2^{17} and for all Parallel-FP arithmetic instructions is to return the true biased exponent value MOD 2^{8}. The value’s significand is rounded to the specified precision and written to the destination register. If the rounded value is different from the infinitely-precise value, then inexactness is signaled. If the significand was rounded by adding a one to its least significant bit, then bit `fpa` in ISR.code is set to 1. Finally, an interruption due to a Floating-Point Exception trap will occur.

Note that when rounding to single, double, or double-extended real, the overflow trap enabled response for normal (non Parallel FP) arithmetic instructions is not guaranteed to be in the range of a valid single, double, or double-extended real quantity, because it is in 17-bit exponent format.

5.4.3 Definition of Tininess, Inexact and Underflow

Tininess is detected after rounding, and is said to occur when a non-zero result (computed as though the exponent range were unbounded) would lie strictly between $+2^{Emin}$ and -2^{Emin}. See Table 5-1 for the values of Emin for each real type. Creation of a tiny result may cause an exception later (such as overflow upon division because it is so small).

Inexactness is said to occur when the result differs from what would have been computed if both the exponent range and precision were unbounded.

How tininess and inexactness trigger the underflow exception depends on whether the Underflow Floating-Point Exception trap is disabled or enabled. If the trap is disabled then the underflow exception is signaled when the result is both tiny and inexact. If the trap is enabled then the underflow exception is signaled when the result is tiny, regardless of inexactness. Note that in the event that the Underflow Floating-Point Exception trap is disabled and tininess but not inexactness occurs, then neither underflow nor inexactness is signaled, and the result is a denormal.

The IEEE Underflow Floating-Point Exception trap disabled response for all normal and Parallel-FP arithmetic instructions is to denormalize the infinitely precise result and then round it to the destination precision. The result may be a denormal, zero, or a normal. The inexact exception is signaled when appropriate.

The IEEE Underflow Floating-Point Exception trap enabled response for all normal arithmetic instructions is to return the true biased exponent value MOD $2^{17}$and for all Parallel-FP arithmetic instructions is to return the true biased exponent value MOD 2^{8}. The significand is rounded to the specified precision and written to the destination register independent of the possibility of the exponent calculation requiring a borrow. If the rounded value is different from the infinitely-precise value, then inexactness is signaled. If the significand was rounded by adding a one to its least significant bit, then bit `fpa` in ISR.code is set to 1. Finally, an interruption due to a Floating-Point Exception trap will occur.

Note: When rounding to single, double, or double-extended real, the underflow trap enabled response for normal (non Parallel FP) arithmetic instructions is not guaranteed to be in the range of a valid single, double, or double-extended real quantity, because it is in 17-bit exponent format.

When Flush-to-Zero mode is enabled, the behavior for tiny results is different. If an instruction would deliver a tiny result, a correctly signed zero is delivered instead and the appropriate FPSR.sf*x*.u and FPSR.sf*x*.i bits are set. This mode may improve the performance on implementations that do not implement denormal handling in hardware. When the Flush-to-Zero mode is enabled, floating-point exception software assist traps will not occur when producing tiny results.

5.4.4 Integer Invalid Operations

Floating-point to integer conversions which are invalid (in the IEEE sense) signal an Invalid Operation Floating-Point Exception fault. If the IEEE Invalid Operation trap is disabled, then the largest magnitude negative integer is the result, even for unsigned integer operations.

5.4.5 Definition of Arithmetic Operations

Arithmetic operations are those that compute on the operands by treating each operand's encoding as a value, whereas non-arithmetic operations perform bit manipulations on the input operands without regard to the value represented by the encoding (except for NaTVal detection). Non-arithmetic instructions do not cause Floating-point Exception faults or traps, but can cause the Disabled Floating-point Register fault.

5.4.6 Definition and Propagation of NaNs

Signaling NaNs have a zero in the most significant fractional bit of the significand. Quiet NaNs have a one in the most significant fractional bit of the significand. This definition of signaling and quiet NaNs easily preserves "NaNness" when converting between different precisions. When propagating NaNs in operations that have more than one NaN operand, the result NaN is chosen from one of the operand NaNs in the following priority based on register encoding fields: first *f4,* then *f2*, and lastly *f3*.

5.4.7 IEEE Standard Mandated Operations Deferred to Software

The following IEEE mandated operations will be implemented in software:

- String to floating-point conversion
- Floating-point to string conversion
- Divide (with help from `frcpa` or `fprcpa` instruction)
- Square root (with help from `frsqrta` or `fprsqrta` instruction)
- Remainder (with help from `frcpa` or `fprcpa` instruction)
- Floating-point to integer valued floating-point conversion
- Correctly wrapping the exponent for single, double, and double-extended overflow and underflow values, as recommended by the IEEE standard

5.4.8 Additions beyond the IEEE Standard

- The fused multiply and add (`fma`, `fms`, `fnma`, `fpma`, `fpms`, `fpnma`) operations enable efficient software divide, square root, and remainder algorithms.
- The extended range of the 17-bit exponent in the register format allows simplified implementation of many basic numeric algorithms by the careful numeric programmer.

- The NaTVal is a natural extension of the IEEE concept of NaNs. It is used to support speculative execution.
- Flush-to-Zero mode is an industry standard addition.
- The minimum and maximum instructions allow the efficient execution of the common Fortran Intrinsic Functions: MIN(), MAX(), AMIN(), AMAX(); and C language idioms such as a<b?a:b.
- All mixed precision operations are allowed. The IEEE standard suggests that implementations allow lower precision operands to produce higher precision results; this is supported. The IEEE standard also suggests that implementations not allow higher precision operands to produce lower precision results; this suggestion is not followed. When computations with higher precision operands produce values beyond the destination precision range, the information provided in the ISR.code allows the true result to be unambiguously determined by software. The correct wrapping count and the appropriate bias amount can also be computed.
- An IEEE style quad-precision real type that is supported in software.

§

IA-32 Application Execution Model in an Intel® Itanium® System Environment 6

IA-32 application execution on Itanium-based systems may be supported with IA-32 Execution Layer, an OS-based optimizing binary translator, or processor hardware-based execution. The implementation of IA-32 application execution on a platform is transparent to IA-32 applications and does not require any application modification.

6.1 IA-32 Execution Layer

IA-32 Execution Layer provides operating systems with optimizing dynamic binary translation to accelerate legacy IA-32 application performance relative to hardware-based execution. When installed, IA-32 Execution Layer supersedes hardware-based execution of IA-32 applications.

The operating system loads IA-32 Execution Layer into user space, where it executes using application virtual space and privilege level. IA-32 Execution Layer uses the native OS for acquiring system resources (memory, synchronization objects, etc.), executing 32-bit system calls issued by the IA-32 application, signal handling, exceptions, and other system notifications.

IA-32 Execution Layer supports user-mode, 32-bit-flat-protected applications. Consistent with Itanium-based operating systems that support legacy IA-32 applications, 16-bit applications and applications containing 32-bit device drivers are not supported.

6.2 Hardware-based IA-32 Application Execution

This section describes the IA-32 execution model from the perspective of an application programmer using the Itanium architecture, interfacing with IA-32 code, while operating in the Itanium System Environment. The main features covered are:

- IA-32 integer, segment, floating-point, MMX technology, and SSE register state mappings
- Instruction set transitions
- IA-32 memory and addressing model overview

This section does not cover the details of IA-32 application programming model, IA-32 instructions and registers. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details regarding IA-32 application programming model.

The Itanium architecture can support 16-bit Real Mode, 16-bit VM86, and 16-bit/32-bit Protected Mode IA-32 applications in the context of an Itanium architecture-based operating system. Whether an IA-32 application is actually supported on specific operating systems is determined by the infrastructure provided by that specific operating system.

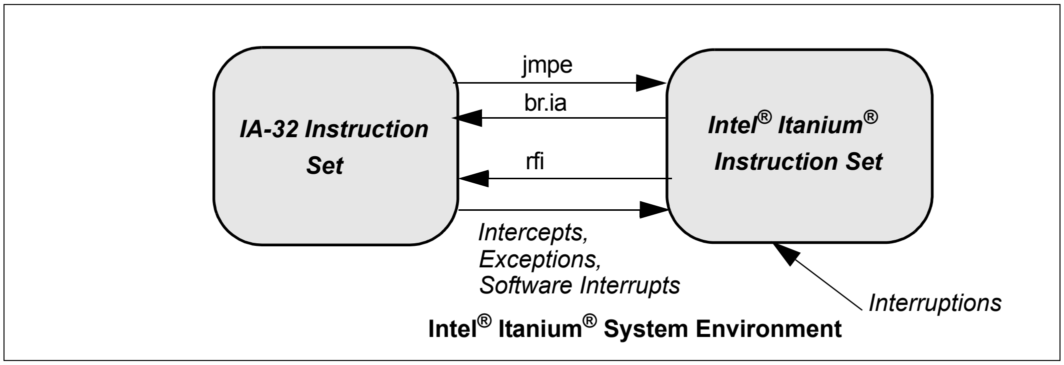
6.2.1 Instruction Set Modes

The processor can be executing either IA-32 or Itanium instructions at any point in time. PSR.is (defined in Section 3.3.2, "Processor Status Register (PSR)" on page 2:23) specifies the currently executing instruction set, where 1 indicates IA-32 instructions are executing, and 0 indicates Itanium instructions are executing. Three special instructions and interruptions are defined to transition the processor between the IA-32 and the Itanium instruction sets as shown in Figure 6-1.

- `jmpe` (IA-32 instruction) Jump to an Itanium target instruction, and transition to the Itanium instruction set.
- `br.ia` (Itanium instruction) Branch to an IA-32 target instruction, and change the instruction set to IA-32.
- `rfi` (Itanium instruction) "Return from interruption" is defined to return to either an IA-32 or Itanium instruction when resuming from an interruption.
- Interruptions transition the processor to the Itanium instruction set for all interruption conditions.

The `jmpe` and `br.ia` instructions provide a low overhead mechanism to transfer control between the instruction sets. These primitives typically are incorporated into "thunks" or "stubs" that implement the required call linkage and calling conventions to call dynamic or statically linked libraries.

Figure 6-1. Instruction Set Transition Model


jmpe
br.ia
IA-32 Instruction Set
rfi
Intel® Itanium® Instruction Set
Intercepts, Exceptions, Software Interrupts
Interruptions
Intel® Itanium® System Environment

6.2.1.1 Instruction Set Execution in the Intel® Itanium® Architecture

While the processor executes from the Itanium instruction set (PSR.is is 0):

- Itanium instructions are fetched, decoded and executed by the processor.
- Itanium instructions can access the entire Itanium and IA-32 application register state. This includes IA-32 segment descriptors, selectors, general registers, physical floating-point registers, MMX technology registers, and SSE registers. See

Section 6.2.2, “IA-32 Application Register State Model” for a description of the register state mapping.

- Segmentation is disabled. No segmentation protection checks are applied nor are segment bases added to compute virtual addresses. All computed addresses are virtual addresses.
- 2^{64} virtual addresses can be generated and memory management is used for all memory and I/O references.

6.2.1.2 IA-32 Instruction Set Execution

While the processor is executing the IA-32 instruction set (PSR.is is 1) within the Itanium System Environment, the IA-32 application architecture as defined by the Pentium III processor is used, namely:

- IA-32 16/32-bit application level, MMX technology, and SSE instructions are fetched, decoded, and executed by the processor. Instructions are confined to 32/16-bit operations.
- Only IA-32 application level register state is visible (i.e. IA-32 general registers, MMX technology, and SSE registers, selectors, EFLAGS, FP registers and FP control registers). Itanium application and control register state is not visible, e.g. branch, predicate, application, control, debug, test, and performance monitor registers.
- IA-32, Real Mode, VM86 and Protected Mode segmentation is in effect. Segment protection checks are applied and virtual addresses generated according to IA-32 segmentation rules. GDT and LDT segments are defined to support IA-32 segmented applications. Segmented 16- and 32-bit code is fully supported.
- Virtual addresses are confined to the lower 4G bytes of virtual region 0. Itanium architecture memory management is used to translate virtual to physical addresses for all IA-32 instruction set memory and I/O Port references.
- Instruction and Data memory references are forced to be little-endian. Memory ordering uses the Pentium III processor memory ordering model.
- IA-32 operating system resources; IA-32 paging, MTRRs, IDT, control registers, debug registers and privileged instructions are superseded by resources defined in the Itanium architecture. All accesses to these resources result in an interception fault.

6.2.1.3 Instruction Set Transitions

The following section summarizes behavior for each instruction set transition. Detailed instruction description on `jmpe` (IA-32 instruction) and `br.ia` (Itanium instruction) should be consulted for details.

Operating systems can disable instruction set transitions (`jmpe` and `br.ia`) by setting PSR.di to one. If PSR.di is one, execution of `jmpe` or `br.ia` results in a Disabled Instruction Set Transition Fault. System level instruction set transitions due to either `rfi` or an interruption ignore the state of PSR.di (defined in Section 3.3.2, “Processor Status Register (PSR)” on page 2:23).

6.2.1.3.1 JMPE Instruction

`jmpe reg16/32; jmpe disp16/32` is used to jump and transfer control to the Itanium instruction set. There are two forms; register indirect and absolute. The absolute form computes the Itanium target virtual address as follows:

```
IP{31:0} =disp16/32 + CSD.base
IP{63:32} = 0
```

The indirect form reads a 16/32-bit register location and then computes the Itanium target address as follows:

```
IP{31:0} = [reg16/32] + CSD.base
IP{63:32} = 0
```

`jmpe` targets are forced to be 16-byte aligned, and are constrained to the lower 4G-bytes of the 64-bit virtual address space due to limited IA-32 addressability. If there are any pending IA-32 numeric exceptions, `jmpe` is nullified, and an IA-32 floating-point exception fault is generated.

Transitions into the Itanium instruction set do not change the privilege level of the processor.

6.2.1.3.2 Branch to IA Instruction

The `br.ia` instruction is used to unconditionally branch to the IA-32 instruction set. IA-32 targets are specified by a 32-bit virtual address target (not an effective address). The IA-32 virtual address is truncated to 32-bits. The `br.ia` branch hints should always be set to predicted static taken. The processor transitions to the IA-32 instruction set as follows:

```
IP{31:0} = BR[b]{31:0}
IP{63:32} = 0
EIP{31:0} = IP{31:0} - CSD.base
```

Transitions into the IA-32 instruction set do not change the privilege level of the processor.

Software should ensure the code segment descriptor and selector are properly loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA-32 GPFault(0) exception is reported on the target IA-32 instruction.

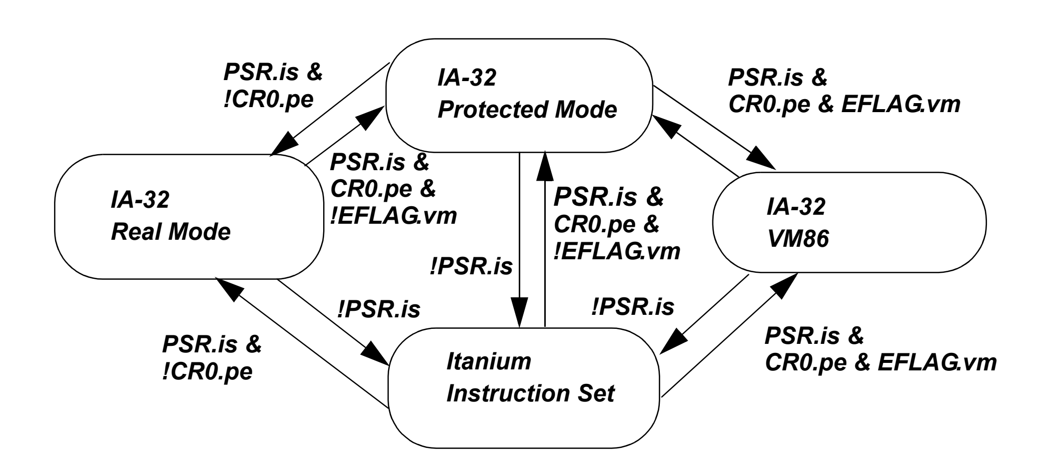
The processor does not ensure Itanium instruction set generated writes into the IA-32 instruction stream are observed by the processor. For details, see "Self Modifying Code" on page 1:132. Before entering the IA-32 instruction set, Itanium architecture-based software must ensure all prior register stack frames have been flushed to memory. All registers left in the current and prior register stack frames are left in an undefined state after IA-32 instruction set execution. Software can not rely on the value of these registers across an instruction set transition. For details, see "Register Stack Engine" on page 1:133.

6.2.1.4 IA-32 Operating Mode Transitions

As described in "IA-32 Instruction Set Execution" on page 1:111, `jmpe`, `br.ia`, and `rfi` instructions and interruptions can transition the processor between the two instruction set modes. Transitions are allowed between the Itanium architecture and all major IA-32 modes. As shown in Figure 6-1, `br.ia` and `rfi` will transition the processor from the Itanium instruction set into IA-32 VM86, Real Mode or Protected Mode. While `jmpe` and interruptions will transition the processor from either IA-32 VM86, Real Mode or

Protected Mode into the Itanium instruction set. Mode transitions between IA-32 Real Mode, Protected Mode and VM86 definitions are the same as those defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

Figure 6-1. Instruction Set Mode Transitions



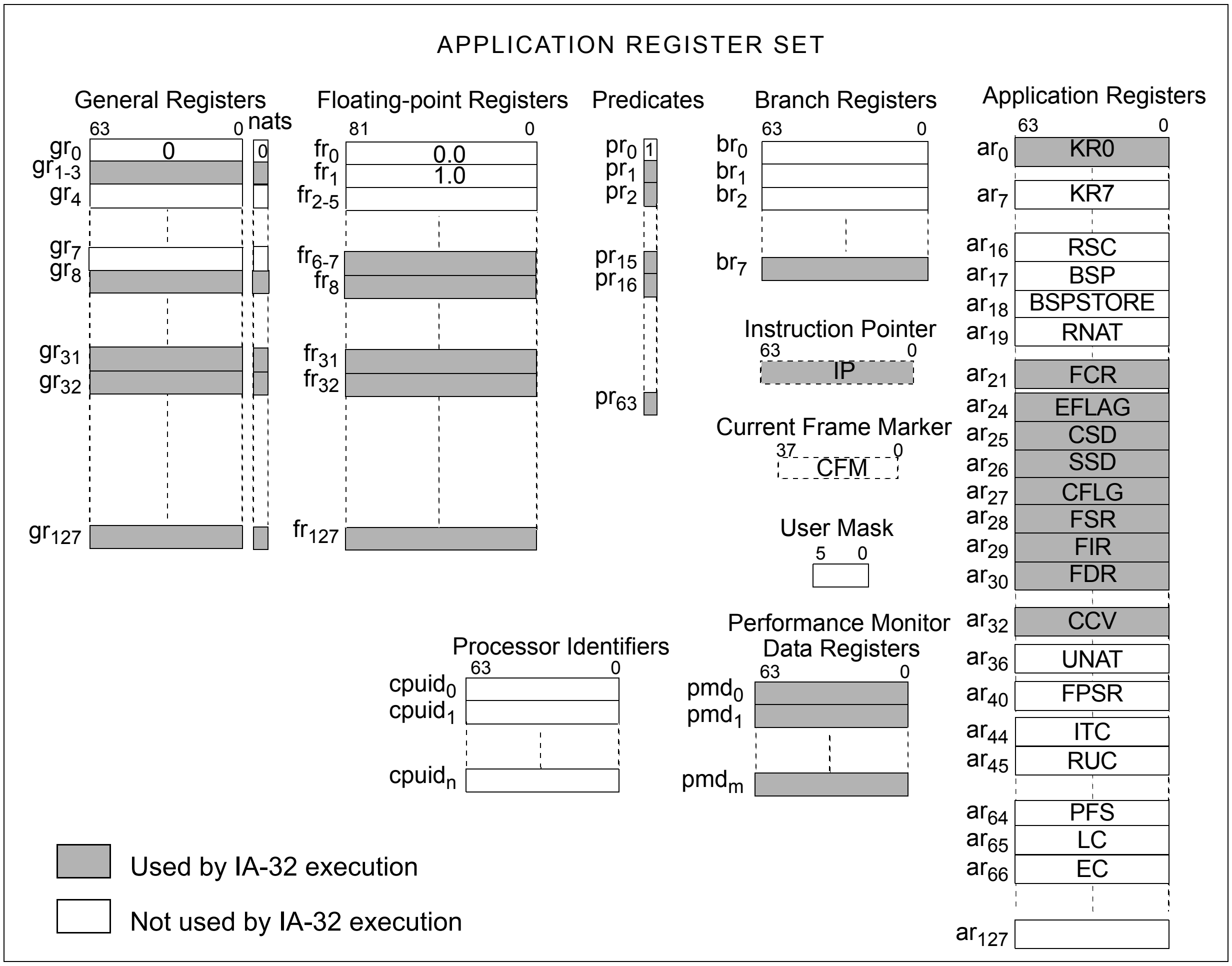
Itanium architecture-based interface code is responsible for setting up and loading a consistent Protected Mode, Real Mode, or VM86 environment (e.g. loading segment selectors and descriptors, etc.) as defined in "Segment Descriptor and Environment Integrity" on page 1:119. The processor applies additional segment descriptor checks to ensure operations are performed in a consistent manner.

6.2.2 IA-32 Application Register State Model

As shown in Figure 6-2 and Table 6-1, IA-32 general purpose registers, segment selectors, and segment descriptors, are mapped into the lower 32-bits of Itanium general purpose registers GR8 to GR31. The floating-point register stack, MMX technology, and SSE registers are mapped on Itanium floating-point registers FR8 to FR31.

To promote straight-forward parameter passing, integer and IEEE floating-point register and memory data types are binary compatible between both IA-32 and Itanium instruction sets.

Figure 6-2. IA-32 Application Register Model



Some Itanium registers are modified to an undefined state by hardware as a side-effect during IA-32 instruction set execution as noted in Table 6-1 and Figure 6-2. Generally, Itanium system state is not affected by IA-32 instruction set execution. Itanium architecture-based code can reference all registers (including IA-32), while IA-32 instruction set references are confined to the IA-32 visible application register state.

Registers are assigned the following conventions during transitions between IA-32 and Itanium instruction sets.

- **IA-32 state**: The register contains an IA-32 register during IA-32 instruction set execution. Expected IA-32 values should be loaded before switching to the IA-32 instruction set. After completion of IA-32 instructions, these registers contain the results of the execution of IA-32 instructions. These registers may contain any value during Itanium instruction execution according to Itanium software conventions. Software should follow IA-32 and Itanium calling conventions for these registers.
- **Undefined**: Registers marked as undefined may be used as scratch areas for execution of IA-32 instructions by the processor and are not ensured to be preserved across instruction set transitions.

- **Shared**: Shared registers contain values that have similar functionality in either instruction set. For example, the stack pointer (ESP) and instruction pointer (IP) are shared.
- **Unmodified**: These registers are not altered by IA-32 execution. Itanium architecture-based code can rely on these values not being modified during IA-32 instruction set execution. The register will have the same contents when entering the IA-32 instruction set and when exiting the IA-32 instruction set.

Table 6-1. IA-32 Application Register Mapping

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
General Purpose Integer Registers				
GR0				constant 0
GR1-3		undefined[f]		scratch for IA-32 execution
GR4-7		unmodified		Intel® Itanium® preserved registers
GR8	EAX	IA-32 state	32[a]	IA-32 general purpose registers
GR9	ECX			
GR10	EDX			
GR11	EBX			
GR12	ESP			
GR13	EBP			
GR14	ESI			
GR15	EDI			
GR16{15:0}	DS		64	IA-32 selectors
GR16{31:16}	ES			
GR16{47:32}	FS			
GR16{63:48}	GS			
GR17{15:0}	CS			
GR17{31:16}	SS			
GR17{47:32}	LDT			
GR17{63:48}	TSS			
GR18-23		undefined[f]		scratch for IA-32 execution
GR24	ESD	IA-32 state	64	IA-32 segment descriptors (register format)[b]
GR25-26		undefined[f]		scratch for IA-32 execution
GR27	DSD	IA-32 state	64	IA-32 segment descriptors (register format)[b]
GR28	FSD			
GR29	GSD			
GR30	LDTD[c]			
GR31	GDTD			
GR32-127		undefined[d]		IA-32 code execution space
Process Environment				
IP	IP	shared	64	shared IA-32 and Intel® Itanium® virtual Instruction Pointer
Floating-point Registers				
FR0				constant +0.0
FR1				constant +1.0
FR2-5		unmodified		Intel® Itanium® preserved registers
FR6-7		undefined		IA-32 code execution space

Table 6-1. IA-32 Application Register Mapping (Continued)

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
FR8	MM0/FP0	IA-32 state	64/80	IA-32 Intel MMX technology registers (aliased on 64-bit FP mantissa) IA-32 FP registers (physical registers mapping)[e]
FR9	MM1/ FP1			
FR10	MM2/FP2			
FR11	MM3/FP3			
FR12	MM4/FP4			
FR13	MM5/FP5			
FR14	MM6/FP6			
FR15	MM7/FP7			

FR16-17	XMM0	IA-32 state	64	IA-32 SSE registers
				low order 64-bits of XMM0 are mapped to FR16{63:0}
				high order 64-bits of XMM0 are mapped to FR17{63:0}

FR18-19	XMM1			
FR20-21	XMM2			
FR22-23	XMM3			
FR24-25	XMM4			
FR26-27	XMM5			
FR28-29	XMM6			
FR30-31	XMM7			
FR32-127		undefined[f]		IA-32 code execution space
Predicate Registers				
PR0				constant 1
PR1-63		undefined[f]		IA-32 code execution space
Branch Registers				
BR0-5		unmodified		Intel® Itanium® preserved registers
BR6-7		undefined		IA-32 code execution space
Application Registers				
RSC		unmodified		not used for IA-32 execution Intel® Itanium® preserved registers
BSP				
BSPSTORE				
RNAT				
CCV		undefined[f]	64	IA-32 code execution space
UNAT		unmodified		not used for IA-32 execution, Intel® Itanium® preserved register
FPSR.sf0		unmodified		Intel® Itanium® numeric status and controls register
FPSR.sf1,2,3		undefined[f]		IA-32 code execution space.
FSR	FSW,FTW, MXCSR	IA-32 state	64	IA-32 numeric status and tag word and SSE status
FCR	FCW, MXCSR		64	IA-32 numeric and SSE control
FIR	FOP, FIP, FCS		64	IA-32 x87 numeric environment opcode, code selector and IP
FDR	FEA, FDS		64	IA-32 x87 numeric environment data selector and offset
ITC	TSC	shared	64	shared IA-32 time stamp counter (TSC) and Intel® Itanium® Interval Timer
RUC		unmodified	64	RUC continues to count while in IA-32 execution mode

Table 6-1. IA-32 Application Register Mapping (Continued)

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
PFS		unmodified		not used for IA-32 code execution, Prior EC is preserved in PFM Intel® Itanium® preserved registers
LC				
EC				
EFLAG	EFLAG	IA-32 state	32	IA-32 System/Arithmetic flags, writes of some bits condition by CPL and EFLAG.iopl.
CSD	CSD		64	IA-32 code segment (register format)[b]
SSD	SSD			IA-32 stack segment (register format)[b]
CFLG	CR0/CR4		64	IA-32 control flags CR0=CFLG{31:0}, CR4=CFLG{63:32}, writable at CPL=0 only.

a. On transitions into the IA-32 instruction set the upper 32-bits are ignored. On exit the upper 32-bits are sign extended from bit 31.

b. Segment descriptor formats differ from the iA-32 memory format, see "IA-32 Segment Registers" on page 1:118 for details. Modification of a selector or descriptor does not set the access/busy bit in memory.

c. The GDT/LDT descriptors are NOT protected from modification by Itanium architecture-based user level code

d. All registers in the current and prior registers frames are left in an undefined state after IA-32 execution. Software must preserve these values before entering the IA-32 instruction set.

e. IA-32 floating-point register mappings are physical and do not reflect the IA-32 top of stack value.

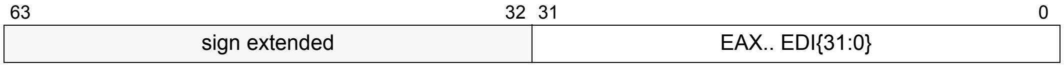
f. These registers are used by the processor and may be left an undefined state following IA-32 instruction set execution. Software should preserve required values before entering IA-32 code.

6.2.2.1 IA-32 General Purpose Registers

Integer registers are mapped into the lower 32-bits of Itanium general registers GR8 to GR15. Values in the upper 32-bits of GR8 to GR15 are ignored on entry to IA-32 execution. After the IA-32 instruction set completes execution, the upper 32-bits of GR8 - GR15 are sign-extended from bit 31.

Based on IA-32 and Itanium calling conventions, the required IA-32 state must be loaded in memory or registers by Itanium architecture-based code before entering the IA-32 instruction set.

Figure 6-3. IA-32 General Registers (GR8 to GR15)



6.2.2.2 IA-32 Instruction Pointer

The processor maintains two instruction pointers for IA-32 instruction set references, EIP (32-bit effective address) and IP (a 64-bit virtual address equivalent to the Itanium instruction set IP). IP is generated by adding the code segment base to EIP and zero extending to 64-bits. IP should not be confused with the 16-bit effective address instruction pointer of the 8086. EIP is an offset within the current code segment, while IP is a 64-bit virtual pointer shared with the Itanium instruction set. The following relationship is defined between EIP and IP while executing IA-32 instructions.

```
IP{63:32} = 0;
IP{31:0} = EIP{31:0} + CSD.Base;
```

EIP is added to the code segment base and zero extended into a 64-bit virtual address on every IA-32 instruction fetch. If during an IA-32 instruction fetch, EIP exceeds the code segment limit, a GPFault is generated on the referencing instruction. Effective instruction addresses (sequential values or jump targets) above 4G-bytes are truncated to 32 bits, resulting in a 4-G byte wraparound condition.

6.2.2.3 IA-32 Segment Registers

IA-32 segment selectors and descriptors are mapped to GR16 - GR29 and AR25 - AR26. Descriptors are maintained in an unscrambled format shown in Figure 6-5. This format differs from the IA-32 scrambled memory descriptor format. The unscrambled register format is designed to support fast conversion of IA-32 segmented 16/32-bit pointers into virtual addresses by Itanium architecture-based code. IA-32 segment register load instructions unscramble the GDT/LDT memory format into the descriptor register format on a segment register load. Itanium architecture-based software can also directly load descriptor registers provided they are properly unscrambled by software. When Itanium architecture-based software loads these registers, no data integrity checks are performed at that time if illegal values are loaded in any fields. For a complete definition of all bit fields and field semantics refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

Figure 6-4. IA-32 Segment Register Selector Format

63 48	47 32	31 16	15 0	
GS	FS	ES	DS	GR16
TSS	LDT	SS	CS	GR17

Figure 6-5. IA-32 Code/Data Segment Register Descriptor Format

63	62	61	60	59	58 57	56	55 52	51 32	31 0
g	d/b	ig	av	p	dpl	s	type	lim{19:0}	base{31:0}

Table 6-2. IA-32 Segment Register Fields

Field	Bits	Description
selector	15:0	Segment Selector value, see the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for bit definition.
base	31:0	Segment Base value. This value when zero extended to 64-bits, points to the start of the segment in the 64-bit virtual address space for IA-32 instruction set memory references.
lim	51:32	Segment Limit. Contains the maximum effective address value within the segment for expand up segments for IA-32 instruction set memory references. For expand down segments, limit defines the minimum effective address within the segment. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details and segment limit fault conditions. The segment limit is scaled by (lim << 12) | 0xFFF if the segment's g-bit is 1.
type	55:52	Type identifier for data/code segments, including the Access bit (bit 52). See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for encodings and definition.
s	56	Non System Segment. If 1, a data segment, if 0 a system segment.
dpl	58:57	Descriptor Privilege Level. The DPL is checked for memory access permission for IA-32 instruction set memory references.
p	59	Segment Present bit. If 0, and a IA-32 memory reference uses this segment an IA_32_Exception(GPFault) is generated for data segments (CS, DS, ES, FS, GS) and an IA_32_Exception(StackFault) for SS.

Table 6-2. IA-32 Segment Register Fields (Continued)

Field	Bits	Description
av	60	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field is available for IA-32 software use and there will be no future use for this field. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.
ig	61	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by IA-32 software. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.
d/b	62	Segment Size. If 0, IA-32 instruction set effective addresses within the segment are truncated to 16-bits. Otherwise, effective addresses are 32-bits. The code segment's d/b-bit also controls the default operand size for IA-32 instructions. If 1, the default operand size is 32-bits, otherwise 16-bits.
g	63	Segment Limit Granularity. If 1, scales the segment limit by lim=(lim<<12) | 0xFFF for IA-32 instruction set memory references. This field is ignored for Intel® Itanium® instruction set memory references.

6.2.2.3.1 Data and Code Segments

On the transition into IA-32 code, the IA-32 segment descriptor and selector registers (GDT, LDT, DS, ES, CS, SS, FS and GS) must be initialized by Itanium architecture-based code to the required values based on IA-32 and Itanium calling conventions and the segmentation model used.

Itanium architecture-based code may manually load a descriptor with an 8-byte fetch from the LDT/GDT, unscramble the descriptor and write the segment base, limit and attribute. Alternately, Itanium architecture-based software can switch to the IA-32 instruction set and perform the required segment load with an IA-32 Mov Sreg instruction. If Itanium architecture-based code explicitly loads the segment descriptors, it is responsible for the integrity of the segment descriptor.

The processor does not ensure coherency between descriptors in memory and the descriptor registers, nor does the processor set segment access bits in the LDT/GDT if segment registers are loaded by Itanium instructions.

6.2.2.3.2 Segment Descriptor and Environment Integrity

For IA-32 instruction set execution, most segment protection checks are applied by the processor when the segment descriptor is loaded by IA-32 instructions into a segment register. However, segment descriptor loads from the Itanium instruction set into the general purpose register file perform no such protection checks, nor are segment Access-bits updated by the processor.

If Itanium architecture-based software directly loads a descriptor, it is responsible for the validity of the descriptor, and ensuring integrity of the IA-32 Protected Mode, Real Mode or VM86 environments. Table 6-3 defines software guidelines for establishing the initial IA-32 environment. The processor checks the integrity of the IA-32 environment as defined in "IA-32 Environment Runtime Integrity Checks" on page 1:122. On the transitions between IA-32 and Itanium architecture-based code, the processor does NOT alter the base, limit or attribute values of any segment descriptor, nor is there a change in privilege level.

Table 6-3. IA-32 Environment Initial Register State

Register	Field	Real Mode	Protected Mode	VM86 Mode
PSR	**cpl**	**0**	**Privilege Level**	**3**
EFLAG	**vm**	**0**	**0**	**1**
CR0	**pe**	**0**	**1**	**1**
CS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl[c]	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	16-bit
	type	data rd/wr, expand up	execute	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
SS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	PSR.cpl (0)	PSR.cpl	PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit size	16-bit
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	p-bit	1	1	1
	a-bit	1	1	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
DS, ES, FS, GS	selector	base >> 4[a]	selector	base >> 4
	base	selector << 4[b]	base	selector << 4
	dpl	dpl >= PSR.cpl (0)	dpl >= PSR.cpl	dpl >= PSR.cpl (3)
	d-bit	16-bit[d]	16/32-bit	0
	type	data rd/wr, expand up	data types	data rd/wr, expand up
	s-bit	1	1	1
	a-bit	1	1	1
	p-bit	1	1/0[f]	1
	g-bit/limit	0xFFFF[e]	limit	0xFFFF
LDT,GDT, TSS	selector	N/A	selector	
	base		base	
	dpl		dpl >= PSR.cpl	
	d-bit		0	
	type		ldt/gdt/tss types	
	s-bit		0	
	p-bit		1	
	a-bit		1	
	g-bit/limit		limit	

a. Selectors should be set to 16*base for normal RM 64KB operation.
b. Segment base should be set to selector/16 for normal RM 64KB operation.
c. Unless a conforming code segment is specified
d. Segment size should be set to 16-bits for normal RM 64KB operation.
e. Segment limit should be set to 0xFFFF for normal RM 64KB operation.
f. For valid segments the p-bit should be set to 1, for null segments the p-bit should be set to 0.

6.2.2.3.2.1 Protected Mode

Itanium architecture-based software should follow these rules for setting up the segment descriptors for Protected Mode environment before entering the IA-32 instruction set:

- Itanium architecture-based software should ensure the stack segment descriptor register's DPL==PSR.cpl.
- For DSD, ESD, FSD and GSD segment descriptor registers, Itanium architecture-based software should ensure DPL>=PSR.cpl.
- For CSD segment descriptor register, Itanium architecture-based software should ensure DPL==PSR.cpl (except for conforming code segments).
- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the a-bit of all segment descriptor registers are set to 1.
- Software should ensure the p-bit is set to 1 for all valid data segments and to 0 for all NULL data segments.

6.2.2.3.2.2 VM86

Itanium architecture-based software should follow these rules when setting up segment descriptors for the VM86 environment before entering the IA-32 instruction set:

- PSR.cpl must be 3 (or IPSR.cpl must be 3 for `rfi`).
- Itanium architecture-based software should ensure the stack segment descriptor register's DPL==PSR.cpl==3 and set to 16-bit, data read/write, expand up.
- For CSD, DSD, ESD, FSD and GSD segment descriptor registers, Itanium architecture-based software should ensure DPL==3, the segment is set to 16-bit, data read/write, expand up.
- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the P-bit and A-bit of all segment descriptor registers is one.
- Software should ensure that the relationship Base = Selector*16, is maintained for all DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor registers, otherwise processor operation is unpredictable.
- Software should ensure that the DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor register's limit value is set to 0xFFFF, otherwise spurious segment limit faults (GPFault or Stack Faults) may be generated.
- Itanium architecture-based software should ensure all segment descriptor registers are data read/write, including the code segment. The processor will ignore execute permission faults.

6.2.2.3.2.3 Real Mode

Itanium architecture-based software should follow these rules when setting up segment descriptors for the Real Mode environments before entering the IA-32 instruction set, otherwise software operation is unpredictable.

- Itanium architecture-based software should ensure PSR.cpl is 0
- Itanium architecture-based software should ensure the stack segment descriptor register's DPL is 0.

- Software should ensure that all code, stack and data segment descriptor registers do not contain encodings for any system segments.
- Software should ensure the P-bit and A-bit of all segment descriptor registers is one.
- For normal real mode 64K operations, software should ensure that the relationship Base = Selector*16, is maintained for all DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor registers.
- For normal real mode 64K operations, software should ensure that the DSD, CSD, ESD, SSD, FSD, and GSD segment descriptor register’s limit value is set to 0xFFFF and the segment size is set to 16-bit (64K)
- Itanium architecture-based software should ensure all segment descriptor registers indicate readable, writable, including the code segment for normal Real Mode operation.

6.2.2.3.3 IA-32 Environment Runtime Integrity Checks

Processors in the Itanium processor family perform additional runtime checks to verify the integrity of the IA-32 environments. These checks are in addition to the runtime checks defined on IA-32 processors and are high-lighted in Table 6-4. Existing IA-32 runtime checks are listed but not highlighted. Descriptor fields not listed in the table are not checked. As defined in the table, runtime checks are performed either on IA-32 instruction code fetches or on an IA-32 data memory reference to one of the specified segment registers. These runtime checks are not performed during transitions from the Itanium instruction set to the IA-32 instruction set.

Table 6-4. IA-32 Environment Runtime Integrity Checks

Reference	Resource	Real Mode	Protected Mode	VM86Mode	Fault
all code fetches	PSR.cpl	is not 0	ignored	is not 3	Code Fetch Fault (GPFault(0))[a]
	EFLAG.vmCFLG.pe	EFLAG.vm is 1 and CFLG.pe is 0			
	EFLAG.vif EFLAG.vip	EFLAG.vip & EFLAG.vif & CFLG.pe & PSR.cpl==3 & (CFLG.pvi | (EFLAG.vm & CFLG.vme))			
all code fetches CS	dpl	ignored		dpl is not 3	Code Fetch Fault (GPFault(0))
	d-bit			is not 16-bit	
	type	ignored (can be exec or data)			
		GPFault if data expand down			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data memory references to SS	dpl	dpl!=PSR.cpl			Stack Fault
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		read and not readable, write and not writeable			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			

Table 6-4. IA-32 Environment Runtime Integrity Checks (Continued)

Reference	Resource	Real Mode	Protected Mode	VM86Mode	Fault
data memory references to DS, ES, FS and GS	dpl	ignored			GPFault(0)
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		read and not readable, write and not writeable			
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
data memory references to CS	dpl	ignored			GPFault(0)
	d-bit	ignored		is not 16-bit	
	type	ignored		data expand down	
		rd/wr checks are ignored	rd and not readable, wr and not writeable	rd/wr checks are ignored	
	s, p, a-bits	are not 1			
	g-bit/limit	segment limit violation			
memory references to LDT,GDT, TSS	dpl	ignored			GPFault (Selector/0)[b]
	type	ignored			
	s-bit	is not 0			
	a, d-bits	ignored			
	p-bit	is not 1			
	g-bit/limit	segment limit violation			

a. Code Fetch Faults are delivered as higher priority GPFault(0).

b. The GP Fault error code is the selector value if the reference is to GDT or LDT. Otherwise the error code is zero.

6.2.2.4 IA-32 Application EFLAG Register

The EFLAG (AR24) register is made up of two major components, user arithmetic flags (CF, PF, AF, ZF, SF, OF, and ID) and system control flags (TF, IF, IOPL, NT, RF, VM, AC, VIF, VIP). None of the arithmetic or system flags affect Itanium instruction execution. See Table 6-5, "IA-32 EFLAGS Register Fields" on page 1:124 for the behavior on IA-32 and Itanium instruction reads/writes to this application register. For details on system flags in the IA-32 EFLAGS register, see Section 10.3.2, "IA-32 System EFLAG Register" on page 2:243.

Figure 6-1. IA-32 EFLAG Register (AR24)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved (set to 0)										id	vip	vif	ac	vm	rf	0	nt	iopl		of	df	if	tf	sf	zf	0	af	0	pf	1	cf
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																															

The arithmetic flags are used by the IA-32 instruction set to reflect the status of IA-32 operations, control IA-32 string operations, and control branch conditions for IA-32 instructions. These flags are ignored by Itanium instructions. Flags ID, OF, DF, SF, ZF, AF, PF and CF are defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

Table 6-5. IA-32 EFLAGS Register Fields

EFLAG[a]	Bits	Description
cf	0	IA-32 Carry Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
	1	Ignored – For IA-32 instructions, writes are ignored, reads return one. For Itanium instructions, the implementation can either ignore writes and return one on reads; or write the value, and return the last value written on reads.
	3,5, 15	Ignored – For IA-32 instructions, writes are ignored, reads return zero. For Itanium instructions, the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads.
pf	2	IA-32 Parity Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
af	4	IA-32 Aux Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
zf	6	IA-32 Zero Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
sf	7	IA-32 Sign Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
tf	8	See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:243.
if	9	
df	10	IA-32 Direction Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
of	11	IA-32 Overflow Flag. See the ***Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*** for details.
iopl	13:12	See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:243.
nt	14	
rf	16	
vm	17	
ac	18	
vif	19	
vip	20	
id	21	
	63:22	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

[a] On entry into the IA-32 instruction set all bits may be read by subsequent IA-32 instructions, after exit from the IA-32 instruction set these bits represent the results of all prior IA-32 instructions. None of the EFLAG bits alter the behavior of Itanium instruction set execution.

6.2.2.5 IA-32 Floating-point Registers

IA-32 floating-point register stack, numeric controls and environment are mapped into the Itanium floating-point registers FR8 - FR15 and the application register name space as shown in Table 6-6.

Table 6-6. IA-32 Floating-point Register Mappings

Intel® Itanium® Reg	IA-32 Reg	Size (bits)	Description
FR8	ST[(TOS + N)==0]	80	IA-32 numeric register stack Accesses to FR8 - FR15 by Intel® Itanium® instructions ignore the IA-32 TOS adjustment IA-32 accesses use the TOS adjustment for a given register N
FR9	ST[(TOS + N)==1]		
FR10	ST[(TOS + N)==2]		
FR11	ST[(TOS + N)==3]		
FR12	ST[(TOS + N)==4]		
FR13	ST[(TOS + N)==5]		
FR14	ST[(TOS + N)==6]		
FR15	ST[(TOS + N)==7]		
FCR (AR21)	FCW, MXCSR	64	IA-32 numeric and SSE control register
FSR (AR28)	FSW,FTW, MXCSR	64	IA-32 numeric and SSE status and tag word
FIR (AR29)	FOP, FCS, FIP	64	IA-32 numeric instruction pointer
FDR (AR30)	FDS, FEA	48	IA-32 numeric data pointer

6.2.2.5.1 IA-32 Floating-point Stack

IA-32 floating-point registers are defined as follows:

- IA-32 numeric register stack is mapped to FR8 - FR15, using the Intel 8087 80-bit IEEE floating-point format.
- For IA-32 instruction set references, floating-point registers are logically mapped into FR8 - FR15 based on the IA-32 top-of-stack (TOS) pointer held in FCR.top. FR8 represents a physical register after the TOS adjustment and is not necessarily the top of the logical floating-point register stack.
- For Itanium instruction set references, the floating-point register numbers are physical and not a function of the numeric TOS pointer, e.g. references to FR8 always return the value in physical register FR8 regardless of the TOS value. Itanium architecture-based software cannot necessarily assume that FR8 contains the IA-32 logical register ST(0). It is highly recommended that typically IA-32 calling conventions be used which pass floating-point values through memory.

6.2.2.5.2 Special Cases

For IA-32 floating-point instructions, loading a single or double denormal results in a normalized double-extended value placed in the target floating-point register. For Itanium instructions, loading a single or double denormal results in an un-normalized denormal value placed in the target floating-point register. There are two canonical exponent values in the Itanium architecture which indicate single precision and double precision denormals.

When transferring floating-point values from Itanium to IA-32 instructions, it is highly recommended that typical IA-32 calling conventions be followed which pass floating-point values through the memory stack. If software does pass floating-point values from IA-32 to Itanium architecture-based code via the floating-point registers, software must ensure the following:

- Single or double precision Itanium denormals must be converted into a normalized double extended precision value expected by IA-32 instructions. Software can convert Itanium denormals by multiplying by 1.0 in double extended precision (`fma.sfx fr = fr, f1, f0`). If an illegal single or double precision denormal is encountered in IA-32 floating-point operations, an IA-32 Exception (FPError Invalid Operand) fault is generated.

- Floating-point values must be within the range of the IA-32 80-bit (15-bit exponent) double extended precision format. The Itanium architecture uses 82 bits (17-bit widest range exponent) for intermediate calculations. Software must ensure all floating-point register values passed to IA-32 instructions are representable in double extended precision 80-bit format, otherwise processor operation is model specific and undefined. Undefined behavior can include but is not limited to: the generation of an IA_32_Exception (FPError Invalid Operation) fault when used by an IA-32 floating-point instruction, rounding of out-of-range values to zero/denormal/infinity and possible IA_32_Exception (FPError Overflow/Underflow) faults, or float-point register(s) containing out of range values silently converted to QNAN or SNAN (conversion could occur during entry to the IA-32 instruction set or on use by an IA-32 floating-point instruction). Software can ensure all passed floating-point register values are within range by multiplying by 1.0 in double extended precision format (with widest range exponent disabled) by using `fma.sfx fr = fr, f1, f0`.
- Floating-point NaTVal values must not be propagated into IA-32 floating-point instructions, otherwise processor operation is model specific and undefined. Processors may silently convert floating-point register(s) containing NaTVal to a SNAN (during entry to the IA-32 instruction set or on a consuming IA-32 floating-point instruction). Dependent IA-32 floating-point instructions that directly or indirectly consume a propagated NaTVal register will either propagate the NaTVal indication or generate an IA_32_Exception (FPError Invalid Operand) fault. Whether a processor generates the fault or propagates the NaTVal is model specific. In no case will the processor allow a NaTVal register to be used without either propagating the NaTVal or generating an IA_32_Exception (FPError Invalid Operand) fault.

Note: It is not possible for IA-32 code to read a NaTVal from a memory location with an IA-32 floating-point load instruction, since a NatVal cannot be expressed by a 80-bit double extended precision number.

It is highly recommended that floating-point values be passed on the memory stack per typical IA-32 calling conventions to avoid numeric problems with NatVal and Itanium denormals.

6.2.2.5.3 IA-32 Floating-point Control Registers

FPSR controls Itanium floating-point instructions control and status bits. FPSR does not control IA-32 floating-point instructions or reflect the status of IA-32 floating-point instructions. IA-32 floating-point and SSE instructions have separate control and status registers, namely FCR (floating-point control register) and FSR (floating-point status register).

FCR contains the IA-32 FCW bits and all SSE control bits as shown in Figure 6-1.

FSR contains the IA-32 floating-point status flags FSW, FTW, and SSE status fields as shown in Figure 6-2. The Tag fields indicate whether the corresponding IA-32 logical floating-point register is empty. Tag encodings for zero and special conditions such as Nan, Infinity or Denormal of each IA-32 logical floating-point register are not supported. However, IA-32 instruction set reads of FTW compute the additional special conditions of each IA-32 floating-point register. Itanium architecture-based code can issue a floating-point classify operation to determine the disposition of each IA-32 floating-point register.

FCR and FSR collectively hold all IA-32 floating-point control, status and tag information. IA-32 instructions that are updated and controlled by MXSCR, FCW, FSW and FTAG effectively update FSR and are controlled by FSR. IA-32 reads/writes of MXCSR, FSW, FCW and FTW return the same information as reads/writes of FSR and FCR by Itanium instructions.

Software must ensure that FCR and FSR are properly loaded for IA-32 numeric execution before entering the IA-32 instruction set. For Itanium instructions accessing ignored fields, the implementation can either ignore writes and return the specified constant on reads, or write the value and return the last value written on reads. For Itanium instructions accessing reserved fields, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

Figure 6-1. IA-32 Floating-point Control Register (FCR)

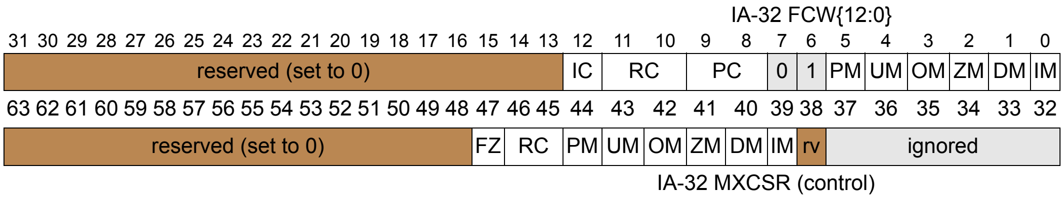


Figure 6-2. IA-32 Floating-point Status Register (FSR)

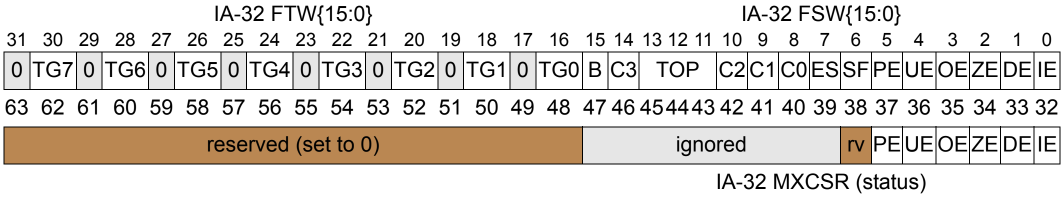


Table 6-7. IA-32 Floating-point Status Register Mapping (FSR)

IA-32 State	Intel® Itanium® State	Bits	IA-32 Usage	Usage in the Intel® Itanium® Architecture
FSW, FTW, MXCSR state in the FSR Register				

Table 6-7. IA-32 Floating-point Status Register Mapping (FSR)

IA-32 State	Intel® Itanium® State	Bits	IA-32 Usage	Usage in the Intel® Itanium® Architecture
FSW.ie	FSR.ie	0	Invalid operation Exception	None of these bits reflect the status of Intel® Itanium® floating-point execution. See the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual*** for IA-32 numeric flag details
FSW.de	FSR.de	1	Denormalized operand Exception	
FSW.ze	FSR.ze	2	Zero divide Exception	
FSW.oe	FSR.oe	3	Overflow Exception	
FSW.ue	FSR.ue	4	Underflow Exception	
FSW.pe	FSR.pe	5	Precision Exception	
FSW.sf	FSR.sf	6	Stack Fault	
FSW.es	FSR.es[a]	7	Error Summary	
FSW.c3:0	FSR.c3:0	8:10,14	Numeric Condition codes	
FSW.top	FSR.top	11:13	Top of IA-32 numeric stack	
FSW.b	FSR.b	15	IA-32 FPU Busy always equals state of FSW.ES	
FTW	FSR.tg {7:0}[b]	16,18,20,22 ,24,26,28,30	Numeric Tags 0-NotEmpty, 1-Empty[c]	
zeros		17,19,21,23,25, 27,29,31, 39:47	Ignored – Writes are ignored, reads return zero	
MXCSR.ie	FSR.ie	32	SSE Invalid operation Exception	Does not reflect the status of Intel® Itanium® floating-point execution. See ***Intel® 64 and IA-32 Architectures Software Developer’s Manual*** for details.
MXCSR.de	FSR.de	33	SSE Denormalized operand Exception	
MXCSR.ze	FSR.ze	34	SSE Zero divide Exception	
MXCSR.oe	FSR.oe	35	SSE Overflow Exception	
MXCSR.ue	FSR.ue	36	SSE Underflow Exception	
MXCSR.pe	FSR.pe	37	SSE Precision Exception	
reserved		38, 48:63	Reserved	
ignored		39:47	Ignored – Writes are ignored, reads return zero	

a. Exception Summary bit, see Section 6.2.2.5.4, “IA-32 Floating-point Environment” for details

b. Tag encodings indicate whether each IA-32 numeric register contains an zero, NaN, Infinity or Denormal are not supported by reads of FSR by Itanium instructions. IA-32 instruction set reads of the FTW field do return zero, Nan, Infinity and Denormal classifications.

c. All MMX technology instructions set all Numeric Tags to 0 = NotEmpty. However, MMX technology instruction EMMS sets all Numeric Tags to 1 = Empty.

6.2.2.5.4 IA-32 Floating-point Environment

To support the Intel 8087 delayed numeric exception model, FSR, FDR and FIR contain pending information related to the numeric exception. FDR contains the operand’s effective address and segment selector. FIR contains the numeric instruction’s effective address, code segment selector, and opcode bits. FSR summaries the type of numeric exception in the IE, DE, ZE, OE, UE, PE, SF and ES-bits. The ES-bit summarizes the IA-32 floating-point exception status as follows:

- When FSR.es is read by Itanium architecture-based code, the value returned is either a summary of any unmasked pending exceptions contained in the FSR, IE, DE, ZE, OE, UE, and PE bits or it may be the value that was last written into the register depending on the implementation.

- When FSR.es is set to 1 by Itanium architecture-based code, delayed IA-32 numeric exceptions are generated on the next IA-32 floating-point instruction, regardless of numeric exception information written into FSR bits; IE, DE, ZE, OE, UE, and PE.
- When FSR.es is written with inconsistent state with respect to the FSR bits (IE, DE, ZE, OE, and PE), subsequent numeric exceptions may report inconsistent floating-point status bits.

For Itanium instructions, the implementation can either raise Reserved Register/Field faults on non-zero writes to the reserved fields, or write the value and return the last value written on reads. FSR, FDR, and FIR must be preserved across a context switch to generate and accurately report numeric exceptions.

Figure 6-1. Floating-point Data Register (FDR)

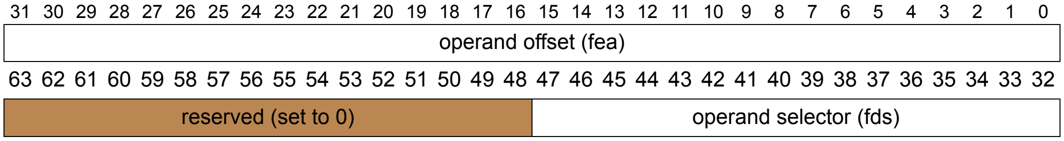
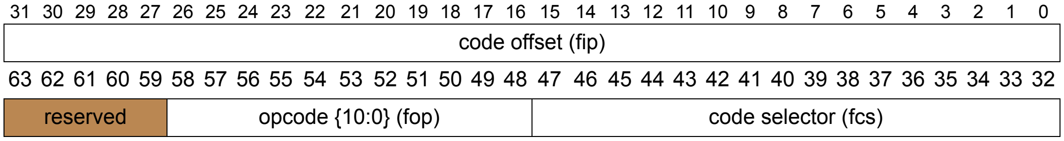


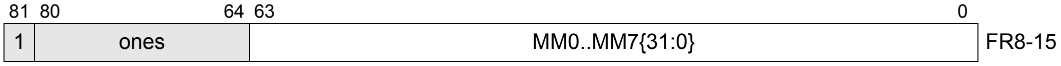
Figure 6-2. Floating-point Instruction Register (FIR)



6.2.2.6 IA-32 Intel® MMX™ Technology Registers

The eight IA-32 Intel MMX technology registers are mapped on the eight Itanium floating-point registers FR8 - FR15 where MM0 is mapped to FR8 and MM7 is mapped to FR15. The MMX technology register mapping for the IA-32 floating-point stack view is dependent on the floating-point IA-32 Top-of-Stack value.

Figure 6-3. IA-32 Intel® MMX™ Technology Registers (MM0 to MM7)



- When a value is written to an MMX technology register using an IA-32 MMX technology instruction:
  - The exponent field of the corresponding floating-point register (bits 80-64) and the sign bit (bit 81) are set to all ones.
  - The mantissa (bits 63-0) is set to the MMX technology data value.
- When a value is read from an MMX technology register by an IA-32 MMX technology instruction:
  - The exponent field of the corresponding floating-point register (bits 80-64) and its sign bit (bit 81) are ignored, including any NaTVal encodings.

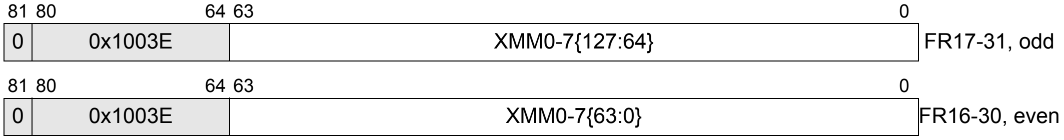
As a result of this mapping, the mantissa of a floating-point value written by either IA-32 or Itanium floating-point instructions will also appear in an IA-32 MMX technology register. An IA-32 MMX technology register will also appear in one of the eight mapped floating-point register's mantissa field.

To avoid performance degradation, software programmers are strongly recommended not to intermix IA-32 floating and IA-32 MMX technology instructions. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for MMX technology coding guidelines for details.

6.2.2.7 IA-32 SSE Registers

The eight 128-bit IA-32 SSE registers (XMM0-7) are mapped on sixteen physical Itanium floating-point register pairs FR16 - FR31. The low order 64-bits of XMM0 are mapped to FR16{63:0}, and the high order 64-bits of XMM0 are mapped to FR17{63:0}.

Figure 6-4. SSE Registers (XMM0-XMM7)

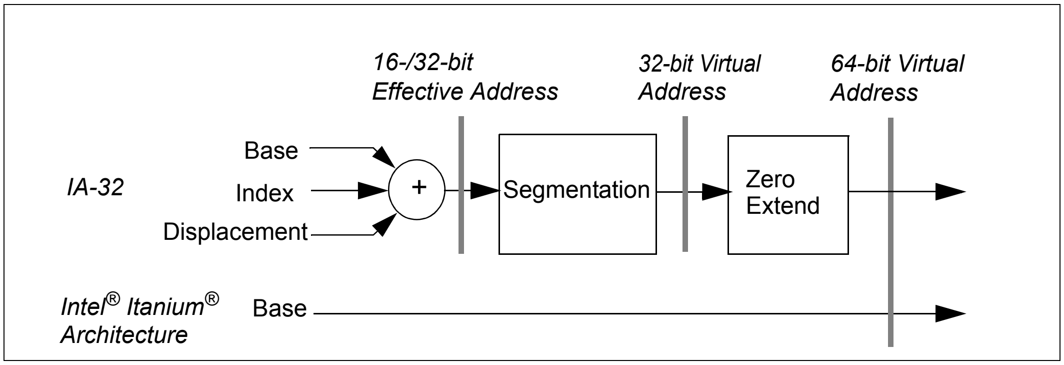


- When a value is written to an SSE register using IA-32 SSE instructions:
  - The exponent field of the corresponding Itanium floating-point register (bits 80-64) is set to 0x1003E and the sign bit (bit 81) is set to 0.
  - The mantissa (bits 63-0) is set to the XMM data value bits{63:0} for even registers and bits{127:64} for odd registers.
- When a SSE register is read using IA-32 SSE instructions:
  - The exponent field of the corresponding Itanium floating-point register (bits 80-64) and the sign bit (bit 81) are ignored, including any NaTVal encodings.

6.2.3 Memory Model Overview

Virtual addresses within either the Itanium or IA-32 instruction set are defined to address the same physical memory location. Itanium instructions directly generate 64-bit virtual addresses. IA-32 instructions generate 16- or 32-bit effective addresses that are then converted into 32-bit virtual addresses by IA-32 segmentation. 32-bit virtual addresses are then converted into 64-bit virtual addresses by zero extending to 64-bits. Zero extension places all IA-32 memory references in the lower 4G-bytes of the 64-bit virtual address space within virtual region 0. Virtual addresses generated by either instruction set are then translated into physical addresses using memory management mechanisms defined in Chapter 4, "Addressing and Protection" in Volume 2.

Figure 6-5. Memory Addressing Model



6.2.3.1 Memory Endianess

Memory integer and floating-point (IEEE) data types are binary compatible between the IA-32 and Itanium instruction sets. Itanium architecture-based applications and operating systems that interact with IA-32 code should use "little-endian" accesses to ensure that memory formats are the same. All IA-32 instruction data and instruction memory references are forced to "little-endian."

6.2.3.2 IA-32 Segmentation

Segmentation is not used for Itanium instruction set memory references. Segmentation is performed on IA-32 instruction set memory references based on the state of EFLAG.vm and CFLG.pe. Either Real Mode, VM86, or Protected Mode segmentation rules are followed as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual****,* specifically:

- **IA-32 Data 16/32-bit Effective Addresses:** 16 or 32-bit effective addresses are generated, based on CSD.d, SSD.b and prefix overrides, by the addition of a base register, scaled index register and 16/32-bit displacement value. Starting effective addresses (first byte of multi-byte operands) larger than 16 or 32 bits are truncated to 16 or 32-bits. Ending (last byte of multi-byte operands) 16-bit effective addresses can extend above the 64K byte boundary, however, ending 32-bit effective addresses are truncated to 32-bits and do not extend above the 4G-byte effective address boundary. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for complete details on wrap conditions.
- **IA-32 Code 16/32-bit Effective Addresses:** 16 or 32-bit EIP, based on CSD.d, is used as the effective address. Starting EIP values (first byte of multi-byte instruction) larger than 16 or 32 bits are truncated to 16 or 32-bits. Ending (last byte of multi-byte instruction) 16-bit effective addresses can extend above the 64K byte boundary, however, ending 32-bit EIP values are truncated to 32-bits and do not extend above the 4G-byte effective address boundary.
- **IA-32 32-bit Virtual Address Generation:** The resultant 16 or 32-bit effective address is mapped into the 32-bit virtual address space by the addition of a segment base. Full segment protection and limit checks are verified as specified by the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** and additional checks as specified in this section. Starting 32-bit virtual addresses are truncated to 32-bits after the addition of the segment base. Ending virtual address

(last byte of a multiple byte operand or instruction) is truncated (wrapped) at the 4G-byte virtual boundary

- **IA-32 64-bit Address Generation:** The resultant 32-bit virtual address is converted into a 64-bit virtual address by zero extending to 64-bits, this places all IA-32 instruction set memory references within the first 4G-bytes of the 64-bit virtual address space within virtual region 0.

If IA-32 code is utilizing a flat segmented model (segment bases are set to zero) then IA-32 and Itanium architecture-based code can freely exchange pointers after a pointer has been zero extended to 64-bits. For segmented IA-32 code, effective address pointers must be first transformed into a virtual address before they are shared with Itanium architecture-based code.

6.2.3.3 Self Modifying Code

While operating in the IA-32 instruction set, self modifying code and instruction cache coherency (coherency with respect to the local processor's data cache) is supported for all IA-32 programs. Self modifying code detection is directly supported at the same level of compatibility as the Pentium processor*.* Software must insert an IA-32 branch instruction between the store operation and the instruction modified for the updated instruction bytes to be recognized.

It is undefined whether the processor will detect a IA-32 self modifying code event for the following conditions; 1) PSR.dt or PSR.it is 0, or 2) there are virtual aliases to different physical addresses between the instruction and data TLBs. To ensure self modifying code works correctly for IA-32 applications, the operating system must ensure that there are no virtual aliases to different physical addresses between the instruction and data TLBs.

When switching from the Itanium instruction set to the IA-32 instruction set, and while executing Itanium instructions, self modifying code and instruction cache coherency are not directly supported by the processor hardware. Specifically, if a modification is made to IA-32 instructions by Itanium instructions, Itanium architecture-based code must explicitly synchronize the instruction caches with the code sequence defined in "Memory Consistency" on page 1:72. Otherwise the modification may or may not be observed by subsequent IA-32 instructions.

When switching from the IA-32 to the Itanium instruction sets, modification of the local instruction cache contents by IA-32 instructions is detected by the processor hardware. The processor ensures that the instruction cache is made coherent with respect to the modification and all subsequent Itanium instruction fetches see the modification.

6.2.3.4 Memory Ordering Interactions

IA-32 instructions are mapped into the Itanium memory ordering model as follows:

- All IA-32 stores have ***release*** semantics
- All IA-32 loads have ***acquire*** semantics
- All IA-32 read-modify-write or lock instructions have ***release*** and ***acquire*** semantics (fully fenced).

Instruction set transitions do not automatically fence memory data references. To ensure proper ordering software needs to take into account the following ordering rules.

Transitions from Itanium instruction set to IA-32 instruction set

- All data dependencies are honored, IA-32 loads see the results of all prior Itanium stores
- IA-32 stores (*release*) can not pass any prior Itanium load or store
- IA-32 loads (*acquire*) can pass prior Itanium unordered loads or any prior Itanium store to a different address. Itanium architecture-based software can prevent IA-32 loads from passing prior Itanium loads and stores by issuing an *acquire* operation (or `mf`) before the instruction set transition.

Transitions from IA-32 instruction set to Itanium instruction set

- All data dependencies are honored, Itanium loads see the results of all prior IA-32 stores
- Itanium stores or loads can not pass prior IA-32 loads (*acquire*)
- Itanium unordered stores or any Itanium load can pass prior IA-32 stores (*release*) to a different address. Itanium architecture-based software can prevent Itanium loads and stores from passing prior IA-32 stores by issuing a *release* operation (or `mf`) after the instruction set transition.

6.2.4 IA-32 Usage of Intel® Itanium® Registers

This section lists software considerations for the Itanium general and floating-point registers, and the ALAT when interacting with IA-32 code.

6.2.4.1 Register Stack Engine

Software must ensure that all dirty registers in the register stack have been flushed to the backing store using a `flushrs` instruction before starting IA-32 execution via either the `br.ia` or `rfi`. Any dirty registers left in the current and prior register stack frames are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition.

Once IA-32 instruction set execution is entered, the RSE is effectively disabled, regardless of any RSE control register enabling conditions.

After exiting the IA-32 instruction set due to a `jmpe` instruction or interruption, all stacked registers are marked as invalid and the number of clean registers is set to zero.

6.2.4.2 ALAT

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software cannot rely on ALAT state being preserved across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. For details on the ALAT, refer to Section 4.4.5.2, “Data Speculation and Instructions” on page 1:64.

6.2.4.3 NaT/NaTVal Response for IA-32 Instructions

If Itanium architecture-based code sets a NaT condition in the integer registers or a NaTVal condition in a floating-point register, MMX technology, or SSE register before switching to the IA-32 instruction set the following conditions can arise:

- When the IA-32 instruction set is entered, NaT values must not be contained in any register defined to contain IA-32 state, otherwise processor operation is model specific and undefined. Processors may generate a NaT Register Consumption Abort on any IA-32 instruction at any time (including the first IA-32 instruction) for all IA-32 integer, MMX technology, SSE, or FP instructions regardless of whether not that instruction directly (or indirectly) references a register containing a NaT. NaT Register Consumption aborts encountered during IA-32 execution may terminate IA-32 instructions in the middle of execution with architectural state already modified.
- Floating-point NaTVal values must not be propagated into IA-32 floating-point instructions, otherwise processor operation is model specific and undefined. Processors may convert floating-point register(s) containing NaTVal to a SNAN (during entry to the IA-32 instruction set or on a consuming IA-32 floating-point instruction). Dependent IA-32 floating-point instructions that directly or indirectly consume a propagated NaTVal register will either propagate the NaTVal indication or generate an IA_32_Exception (FPError Invalid Operand) fault. Whether a processor generates the fault or propagates the NaTVal is model specific. In no case will the processor allow a NaTVal register to be used without either propagating the NaTVal or generating an IA_32_Exception (FPError Invalid Operand) fault.

Note: It is not possible for IA-32 code to read a NaTVal from a memory location with an IA-32 floating-point load instruction since a NaTVal cannot be expressed by a 80-bit double extended precision number. It is highly recommended that floating-point values be passed on the memory stack per typical IA-32 calling conventions to avoid problems with NatVal and Itanium denormals.

- IA-32 SSE instructions that directly or indirectly consume a register containing a NaTVal encoding, will ignore the NaTVal encoding and interpret the register’s mantissa field as a legal data value.
- IA-32 MMX technology instructions that directly or indirectly consume a register containing a NaTVal encoding, will ignore the NaTVal encoding and interpret the register’s mantissa field as a legal data value.

Software should not rely on the behavior of NaT or NaTVal during IA-32 instruction execution, or propagate NaT or NaTVal into IA-32 instructions.

§

Part II: Optimization Guide for the Intel® Itanium® Architecture

About the Optimization Guide 1

The second portion of this document explains in detail optimization techniques associated with the Itanium instruction set. It is intended for those interested in furthering their understanding of application architecture features and optimization techniques that benefit application performance. Intel and the industry are developing compilers to take advantage of these techniques. Application developers are not advised to use this as a guide to assembly language programming for the Itanium architecture.

Note: To demonstrate techniques, this guide contains code examples that are not targeted towards a specific processor based on the Itanium architecture, but rather a hypothetical implementation. For these code examples, ALU operations are assumed to take one cycle and loads take two cycles to return from first level cache and that there are two load/store execution units and four ALUs. Other latencies and execution unit details are described as needed

1.1 Overview of the Optimization Guide

Chapter 2, "Introduction to Programming for the Intel® Itanium® Architecture" provides an overview of the application programming environment.

Chapter 3, "Memory Reference" discusses features and optimizations related to control and data speculation.

Chapter 4, "Predication, Control Flow, and Instruction Stream" describes optimization features related to predication, control flow, and branch hints.

Chapter 5, "Software Pipelining and Loop Support" provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 6, "Floating-point Applications" discusses current performance limitations in floating- point applications and features that address these limitations.

§

Introduction to Programming for the Intel® Itanium® Architecture 2

2.1 Overview

The Itanium instruction set is designed to allow the compiler to communicate information to the processor to manage resource characteristics such as instruction latency, issue width, and functional unit assignment. Although such resources can be statically scheduled, the Itanium architecture does not require that code be written for a specific microarchitecture implementation in order to be functional.

The Itanium architecture includes a complete instruction set with new features designed to:

- Increase instruction-level parallelism (ILP).
- Better manage memory latencies.
- Improve branch handling and management of branch resources.
- Reduce procedure call overhead.

The architecture also enables high floating-point performance and provides direct support for multimedia applications.

Complete descriptions of the syntax and semantics of Itanium instructions can be found in Volume 3: Intel® Itanium® Instruction Set Reference. Though this chapter provides a high level introduction to application level programming, it assumes prior experience with assembly language programming as well as some familiarity with the Itanium application architecture. Optimization is explored in other chapters of this guide.

2.2 Registers

The architecture defines 128 general purpose registers, 128 floating-point registers, 64 predicate registers, and up to 128 special purpose registers. The large number of architectural registers enable multiple computations to be performed without having to frequently spill and fill intermediate data to memory.

There are 128, 64-bit **general purpose registers** (`r0-r127`) that are used to hold values for integer and multimedia computations. Each of the 128 registers has one additional NaT (Not a Thing) bit which is used to indicate whether the value stored in the register is valid. Execution of Itanium speculative instructions can result in a register’s NaT bit being set. Register `r0` is read-only and contains a value of zero (0). Attempting to write to `r0` will cause a fault.

There are 128, 82-bit **floating-point registers** (`f0–f127`) that are used for floating-point computations. The first two registers, `f0` and `f1`, are read-only and read as +0.0 and +1.0, respectively. Instructions that write to `f0` or `f1` will fault.

There are 64, one-bit **predicate registers** (`p0–p63`) that control conditional execution of instructions and conditional branches. The first register, `p0`, is read-only and always reads true (1). The results of instructions that write to `p0` are discarded.

There are 8, 64-bit **branch registers** (`b0–b7`) that are used to specify the target addresses of indirect branches.

There is space for up to 128 **application registers** (`ar0-ar127`) that support various functions. Many of these register slots are reserved for future use. Some application registers have assembler aliases. For example, `ar66` is the Epilogue Counter and is called `ar.ec`.

The **instruction pointer** is a 64-bit register that points to the currently executing instruction bundle.

2.3 Using Intel® Itanium® Instructions

Itanium instructions are grouped into 128-bit *bundles* of three instructions. Each instruction occupies the first, second, or third *slot* of a bundle. Instruction format, expression of parallelism, and bundle specification are described below.

2.3.1 Format

A basic Itanium instruction has the following syntax:

[*qp*] *mnemonic*[.*comp*] *dest*=*srcs*

Where:

qp Specifies a qualifying predicate register. The value of the qualifying predicate determines whether the results of the instruction are committed in hardware or discarded. When the value of the predicate register is true (1), the instruction executes, its results are committed, and any exceptions that occur are handled as usual. When the value is false (0), the results are not committed and no exceptions are raised. Most Itanium instructions can be accompanied by a qualifying predicate.

mnemonic Specifies a name that uniquely identifies an Itanium instruction.

comp Specifies one or more instruction completers. Completers indicate optional variations on a base instruction mnemonic. Completers follow the mnemonic and are separated by periods.

dest Represents the destination operand(s), which is typically the result value(s) produced by an instruction.

srcs Represents the source operands. Most Itanium instructions have at least two input source operands.

2.3.2 Expressing Parallelism

The Itanium architecture requires the compiler or assembly writer to explicitly indicate groups of instructions, called *instruction groups*, that have no register read after write (RAW) or write after write (WAW) register dependencies. Instruction groups are delimited by *stops* in the assembly source code. Since instruction groups have no RAW or WAW register dependencies, they can be issued without hardware checks for register dependencies between instructions. Both of the examples below show two instruction groups separated by stops (indicated by double semicolons):

```
ld8 r1=[r5] ;; // First group
add r3=r1,r4   // Second group
```

A more complex example with multiple register flow dependencies is shown below:

```
ld8 r1=[r5]    // First group
sub r6=r8,r9 ;;// First group
add r3=r1,r4   // Second group
st8 [r6]=r12   // Second group
```

All instructions in a single instruction group may not necessarily issue in parallel because specific implementations may not have sufficient resources to issue all instructions in an instruction group.

2.3.3 Bundles and Templates

In assembly code, each 128-bit bundle is enclosed in curly braces and contains a template specification and three instructions. Thus, a stop may be specified at the end of any bundle or in the middle of a bundle by using one of two special template types that implicitly include mid-bundle stops.

Each instruction in a bundle is 41-bits long. Five other bits are used by a template-type specification. Bundle templates enable processors based on the Itanium architecture to dispatch instructions with simple instruction decoding, and stops enable explicit specification of parallelism.

There are five slot types (M, I, F, B, and L), six instruction types (M, I, A, F, B, L), and 12 basic template types (MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB, MFB). Each basic template type has two versions: one with a stop after the third slot and one without. Instructions must be placed in slots corresponding to their instruction types based on the template specification, except for A-type instructions that can go in either I or M slots. For example, a template specification of `.MII` means that of the three instructions in a bundle, the first is a memory (`M`) or A-type instruction, and the next two are ALU integer (`I`) or A-type instructions:

```
{ .mii
ld4  r28=[r8]  //  Load a 4-byte value
add r9=2,r1    //  2+r1 and put in r9
add  r30=1,r1  //  1+r1 and put in r30
}
```

For readability, most code examples in this book do not specify templates or braces.

Note: Bundle boundaries have no direct correlation with instruction group boundaries as instruction groups can extend over an arbitrary number of bundles. Instruction groups begin and end where stops are set in assembly code, and dynamically whenever a branch is taken or a stop is encountered.

2.4 Memory Access and Speculation

The Itanium architecture provides memory access only through register load and store instructions and special semaphore instructions. The architecture also provides extensive support for hiding memory latency via programmer-controlled speculation.

2.4.1 Functionality

Data and instructions are referenced by 64-bit addresses. Instructions are stored in memory in little endian byte order, in which the *least* significant byte appears in the lowest addressed byte of a memory location. For data, modes for both big and little endian byte order are supported and can be controlled by a bit in the User Mask Register.

Integer loads of one, two, and four bytes are zero-extended, since all 64 bits of each register are always written. Integer stores write one, two, four, or eight bytes of registers to memory as specified.

2.4.2 Speculation

Speculation allows a programmer to break data or control dependencies that would normally limit code motion. The two kinds of speculation are called control speculation and data speculation. This section summarizes speculation in the Itanium architecture. See Chapter 3, “Memory Reference” for more detailed descriptions of speculative instruction behavior and application.

2.4.3 Control Speculation

Control speculation allows loads and their dependent uses to be safely moved above branches. Support for this is enabled by special NaT bits that are attached to integer registers and by special NatVal values for floating-point registers. When a speculative load causes an exception, it is not immediately raised. Instead, the NaT bit is set on the destination register (or NatVal is written into the floating-point register). Subsequent speculative instructions that use a register with a set NaT bit propagate the setting until a non-speculative instruction checks for or raises the deferred exception.

For example, in the absence of other information, the compiler for a typical RISC architecture cannot safely move the load above the branch in the sequence below:

```
(p1) br.cond.dptk L1   // Cycle 0
     ld8 r3=[r5];;     // Cycle 1
     shr r7=r3,r87     // Cycle 3
```

Supposing that the latency of a load is 2 cycles, the shift right (`shr`) instruction will stall for 1. However, by using the speculative loads and checks provided in the Itanium architecture, two cycles can be saved by rewriting the above code as shown below:

```
 ld8.s r3=[r5]         // Earlier cycle
 // Other instructions

(p1) br.cond.dptk L1;; // Cycle 0
     chk.s r3,recovery // Cycle 1
     shr r7=r3,r87     // Cycle 1
```

This code assumes `r5` is ready when accessed and that there are sufficient instructions to fill the latency between the `ld8.s` and the `chk.s`.

2.4.4 Data Speculation

Data speculation allows loads to be moved above possibly conflicting memory references. *Advanced loads* exclusively refer to data speculative loads. Review the order of loads and stores in this assembly sequence:

```
st8 [r55]=r45  // Cycle 0
ld8 r3=[r5] ;; // Cycle 0
shr r7=r3,r87  // Cycle 2
```

The Itanium architecture allows the programmer to move the load above the store even if it is not known whether the load and the store reference overlapping memory locations. This is accomplished using special advanced load and check instructions:

```
ld8.a r3=[r5]  // Advanced load
// Other instructions

st8 [r55]=r45  // Cycle 0
ld8.c r3=[r5]  // Cycle 0 - check
shr r7=r3,r87  // Cycle 0
```

Note: The `shr` instruction in this schedule could issue in cycle 0 if there were no conflicts between the advanced load and intervening stores. If there were a conflict, the check load instruction (`ld8.c`) would detect the conflict and reissue the load.

2.5 Predication

Predication is the conditional execution of an instruction based on a qualifying predicate. A qualifying predicate is a predicate register whose value determines whether the processor commits the results computed by an instruction.

The values of predicate registers are set by the results of instructions such as compare (`cmp`) and test bit (`tbit`). When the value of a qualifying predicate associated with an instruction is true (1), the processor executes the instruction, and instruction results are committed. When the value is false (0), the processor discards any results and raises no exceptions. Consider the following C code:

```
if (a) {
    b = c + d;
}
if (e) {
    h = i + j;
}
```

This code can be implemented in the Itanium architecture using qualifying predicates so that branches are removed. The pseudo-code shown below implements the C expressions without branches:

```
cmp.ne p1,p2=a,r0      // p1 <- a!= 0
cmp.ne  p3,p4=e,r0 ;;  // p3 <- e != 0
(p1)add b=c,d          // If a!= 0 then add
(p3)sub h=i,j          // If e!= 0 then sub
```

See Chapter 4, "Predication, Control Flow, and Instruction Stream" for detailed discussion of predication. There are a few special cases where predicated instructions read or write architectural resources regardless of their qualifying predicate.

2.6 Architectural Support for Procedure Calls

Calling conventions normally require callee and caller saved registers which can incur significant overhead during procedure calls and returns. To address this problem, a subset of the Itanium general registers are organized as a logically infinite set of stack frames that are allocated from a finite pool of physical registers.

2.6.1 Stacked Registers

Registers r0 through r31 are called global or static registers and are not part of the stacked registers. The stacked registers are numbered r32 up to a user-configurable maximum of r127.

A called procedure specifies the size of its new stack frame using the alloc instruction. The procedure can use this instruction to allocate up to 96 registers per frame shared amongst input, output, and local values. When a call is made, the output registers of the calling procedure are overlapped with the input registers of the called procedure, thus allowing parameters to be passed with no register copying or spilling.

The hardware renames physical registers so that the stacked registers are always referenced in a procedure starting at r32.

2.6.2 Register Stack Engine

Management of the register stack is handled by a hardware mechanism called the Register Stack Engine (RSE). The RSE moves the contents of physical registers between the general register file and memory without explicit program intervention. This provides a programming model that looks like an unlimited physical register stack to compilers; however, saving and restoring of registers by the RSE may be costly, so compilers should still attempt to minimize register usage.

2.7 Branches and Hints

Since branches have a major impact on program performance, the Itanium architecture includes features to improve their performance by:

- Using predication to reduce the number of branches in the code. This improves instruction fetching because there are fewer control flow changes, decreases the number of branch mispredicts since there are fewer branches, and it increases the branch prediction hit rates since there is less competition for prediction resources.
- Providing software hints for branches to improve hardware use of prediction and prefetching resources.
- Supplying explicit support for software pipelining of loops and exit prediction of counted loops.

2.7.1 Branch Instructions

Branching in the Itanium architecture is largely expressed the same way as on other microprocessors. The major difference is that branch triggers are controlled by predicates rather than conditions encoded in branch instructions. The architecture also provides a rich set of hints to control branch prediction strategy, prefetching, and specific branch types like loops, exits, and branches associated with software pipelining. Targets for indirect branches are placed in branch registers prior to branch instructions.

2.7.2 Loops and Software Pipelining

Compilers sometimes try to improve the performance of loops by using unrolling. However, unrolling is not effective on all loops for the following reasons:

- Unrolling may not fully exploit the parallelism available.
- Unrolling is tailored for a statically defined number of loop iterations.
- Unrolling can increase code size.

To maintain the advantages of loop unrolling while overcoming these limitations, the Itanium architecture provides architectural support for software pipelining. Software pipelining enables the compiler to interleave the execution of several loop iterations without having to unroll a loop. Software pipelining is performed using:

- Loop-branch instructions.
- `LC` and `EC` application registers.
- Rotating registers and loop stage predicates.
- Branch hints that can assign a special prediction mechanism to important branches.

In addition to software pipelined *while* and *counted* loops, the architecture provides particular support for simple counted loops using the `br.cloop` instruction. The `cloop` branch instruction uses the 64-bit Loop Count (`LC`) application register rather than a qualifying predicate to determine the branch exit condition.

For a complete discussion of software pipelining support, see Chapter 5, “Software Pipelining and Loop Support.”

2.7.3 Rotating Registers

Rotating registers enable succinct implementation of software pipelining with predication. Rotating registers are rotated by one register position each time one of the special loop branches is executed. Thus, after one rotation, the content of register `X` will be found in register `X+1` and the value of the highest numbered rotating register will be found in r32. The size of the rotating region of general registers can be any multiple of 8 and is selected by a field in the alloc instruction. The predicate and floating-point registers can also be rotated but the number of rotating registers is not programmable: predicate registers p16 through p63 are rotated, and floating-point registers f32 through f127 are rotated.

2.8 Summary

The Itanium architecture provides features that reduce the effects of traditional microarchitectural performance barriers by enabling:

- Improved ILP with a large number of registers and software scheduling of instruction groups and bundles.
- Better branch handling through predication.
- Reduced overhead for procedure calls through the register stack mechanism.
- Streamlined loop handling through hardware support of software pipelined loops.
- Support for hiding memory latency using speculation.

§

Memory Reference 3

3.1 Overview

Memory latency is a major factor in determining the performance of integer applications. In order to help reduce the effects of memory latency, the Itanium architecture explicitly supports software pipelining, large register files, and compiler-controlled speculation. This chapter discusses features and optimizations related to compiler-controlled speculation. See Chapter 5, “Software Pipelining and Loop Support” for a complete description of how to use software pipelining.

The early sections of this chapter review non-speculative load and store in the Itanium architecture, and general concepts and terminology related to data dependencies. The concept of speculation is then introduced, followed by discussions and examples of how speculation is used. The remainder of this chapter describes several important optimizations related to memory access and instruction scheduling.

3.2 Non-speculative Memory References

The Itanium architecture supports non-speculative loads and stores, as well as explicit memory hint instructions.

3.2.1 Stores to Memory

Itanium integer store instructions can write either 1, 2, 4, or 8 bytes and 4, 8, or 10 bytes for floating-point stores. For example, a `st4` instruction will write the first four bytes of a register to memory.

Although the Itanium architecture uses a little endian memory byte order by default, software can change the byte order by setting the big endian (be) bit of the user mask (UM).

3.2.2 Loads from Memory

Itanium integer load instructions can read either 1, 2, 4, or 8 bytes from memory depending on the type of load issued. Loads of 1, 2, or 4 bytes of data are zero-extended to 64-bits prior to being written into their target registers.

Although loads are provided for various data types, the basic data type is the quadword (8 bytes). Apart from a few exceptions, all integer operations are on quadword data. This can be particularly important when dealing with signed integers and 32-bit addresses, or any addresses that are shorter than 64 bits.

3.2.3 Data Prefetch Hint

The lfetch instruction requests that lines be moved between different levels of the memory hierarchy. Like all hint instructions defined in the Itanium architecture, lfetch has no effect on program correctness, and any microarchitecture implementation may choose to ignore it.

3.3 Instruction Dependencies

Data and control dependencies are fundamental factors in optimization and instruction scheduling. Such dependencies can prevent a compiler from scheduling instructions in an order that would yield shorter critical paths and better resource usage since they restrict the placement of instructions relative to other instructions on which they are dependent.

In general, memory references are the major source of control and data dependencies that cannot be broken due to getting a wrong answer (if a data dependency is broken) or raising a fault that should not be raised (if a control dependency is broken). This section describes:

- Background material on memory reference dependencies.
- Descriptions of how dependencies constrain code scheduling on traditional architectures.

Section 3.4 describes memory reference features defined in the Itanium architecture that increase the number of dependencies that can be removed by a compiler.

3.3.1 Control Dependencies

An instruction is *control dependent* on a branch if the direction taken by the branch affects whether the instruction is executed. In the code below, the load instruction is control dependent on the branch:

```
(p1)br.cond some_label
ld8 r4=[r5]
```

The following sections provide overviews of control dependencies and their effects on optimization.

3.3.1.1 Instruction Scheduling and Control Dependencies

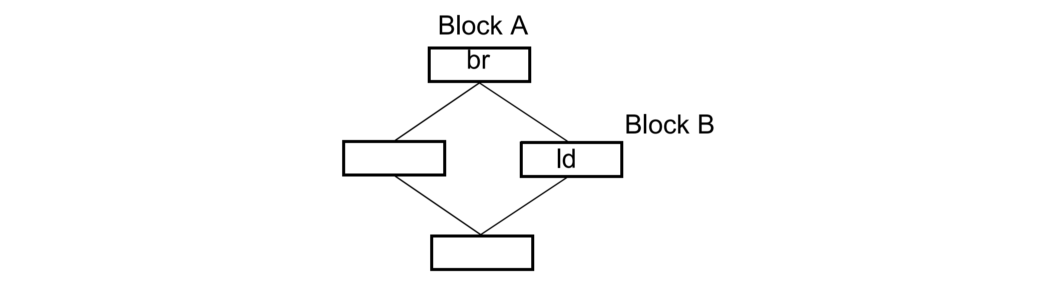
The code below contains a control dependency at the branch instruction:

```
        add        r7=r6,1            // Cycle 0
        add        r13=r25,r27
        cmp.eq     p1,p2=r12,r23
(p1)    br.cond    some_label ;;

        ld4        r2=[r3];;          // Cycle 1
        sub        r4=r2,r11          // Cycle 3
```

A compiler cannot safely move the load instruction before the branch unless it can guarantee that the moved load will not cause a fatal program fault or otherwise corrupt program state. Since the load cannot be moved upward, the schedule cannot be improved using normal code motion.

Thus, the branch creates a barrier to instructions whose execution depends upon it. In Figure 3-1, the load in block B cannot be moved up because of a conditional branch at the end of block A.

Figure 3-1. Control Dependency Preventing Code Motion



3.3.2 Data Dependencies

A data dependency exists between an instruction that accesses a register or memory location and another instruction that alters the same register or location.

3.3.2.1 Basics of Data Dependency

The following basic terms describe data dependencies between instructions:

- Write-after-write (WAW)

  A dependency between two instructions that write to the same register or memory location.

- Write-after-read (WAR)

  A dependency between two instructions in which an instruction reads a register or memory location that a subsequent instruction writes.

- Read-after-write (RAW)

  A dependency between two instructions in which an instruction writes to a register or memory location that is read by a subsequent instruction.

- Ambiguous memory dependencies

  Dependencies between a load and a store, or between two stores where it cannot be determined if the involved instructions access overlapping memory locations. Ambiguous memory references include possible WAW, WAR, or RAW dependencies.

- Independent memory references

  References by two or more memory instructions that are known not to have conflicting memory accesses.

3.3.2.2 Data Dependency in the Intel® Itanium® Architecture

The Itanium architecture requires the programmer to insert stops between RAW and WAW *register* dependencies to ensure correct code results. For example, in the code below, the add instruction computes a value in r4 needed by the sub instruction:

```
add     r4=r5,r6 ;;// Instruction group 1
sub     r7=r4,r9   // Instruction group 2
```

The stop after the add instruction terminates one instruction group so that the sub instruction can legally read r4.

On the other hand, implementations based on the Itanium architecture are required to observe *memory*-based dependencies within an instruction group. In a single instruction group, a program can contain memory-based data dependent instructions and hardware will produce the same results as if the instructions were executed sequentially and in program order. The pseudo-code below demonstrates a memory dependency that will be observed by hardware:

```
mov     r16=1
mov     r17=2 ;;
st8     [r15]=r16
st8     [r14]=r17;;
```

If the address in r14 is equal to the address in r15, uni-processor hardware guarantees that the memory location will contain the value in r17 (2). The following RAW dependency is also legal in the same instruction group even if software is unable to determine if r1 and r2 overlap:

```
st8     [r1]=x
ld4     y=[r2]
```

3.3.2.3 Instruction Scheduling and Data Dependencies

The dependency rules are sufficient to generate correct code, but to generate efficient code, the compiler must take into account the latencies of instructions. For example, the generic implementation has a two cycle latency to the first level data cache. In the code below, the stop maintains correct ordering, but a use of r2 is scheduled only one cycle after its load:

```
add     r7=r6,1           // Cycle 0
add     r13=r25,r27
cmp.eq  p1,p2=r12,r23;;

add     r11=r13,r29       // Cycle 1
ld4     r2=[r3];;

sub     r4=r2,r11         // Cycle 3
```

Since the latency of a load is two cycles, the `sub` instruction will stall until cycle three. To avoid a stall, the compiler can move the load earlier in the schedule so that the machine can perform useful work each cycle:

```
ld4     r2=[r3]             // Cycle 0
add     r7=r6,1
add     r13=r25,r27
cmp.eq  p1,p2=r12,r23;;

add     r11=r13,r29;;       // Cycle 1

sub     r4=r2,r11           // Cycle 2
```

In this code, there are enough independent instructions to move the load earlier in the schedule to make better use of the functional units and reduce execution time by one cycle.

Now suppose that the original code sequence contained an ambiguous memory dependency between a store instruction and the load instruction:

```
add     r7=r6,1             // Cycle 0
add     r13=r25,r27
cmp.ne  p1,p2=r12,r23;;

st4     [r29]=r13           // Cycle 1
ld4     r2=[r3];;

sub     r4=r2,r11           // Cycle 3
```

In this case, the load cannot be moved past the store due to the memory dependency. Stores will cause data dependencies if they cannot be disambiguated from loads or other stores.

In the absence of other architectural support, stores can prevent moving loads and their dependent instructions: The following C language statements could not be reordered unless `ptr1` and `ptr2` were statically known to point to independent memory locations:

```
*ptr1 = 6;
x = *ptr2;
```

3.4 Using Speculation in the Intel® Itanium® Architecture to Overcome Dependencies

Both data and control dependencies constrain optimization of program code. The Itanium architecture provides support for two basic techniques used to overcome dependencies:

- **Data speculation**: Allow a load and possibly its uses to be moved across ambiguous memory writes.
- **Control speculation**: Allows a load and possibly its uses to be moved across a branch on which the load is control dependent.

These techniques are used to hide load latencies and reduce execution time.

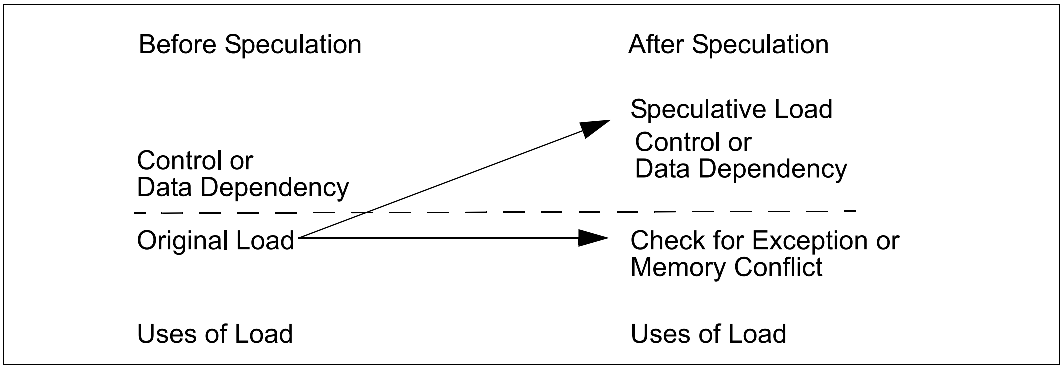
3.4.1 Speculation Model in the Intel® Itanium® Architecture

The limitations imposed by dependencies on instruction scheduling can be solved by separating the loading of data from the exception handling or the acknowledgment of data conflicts. The Itanium architecture supports special speculative versions of instructions to accomplish this:

- Control speculative load instructions defer exceptions.
- Data speculative load instructions save address information.
- Special check instructions check for exceptions or data conflicts.

An Itanium speculative load can be moved above a dependency barrier (shown as a dashed line) as shown in Figure 3-2.

Figure 3-2. Speculation Model in the Intel® Itanium® Architecture



The check detects a deferred exception or a conflict with an intervening store and provides a mechanism to recover from failed speculation. With this support, speculative loads and their uses can be scheduled earlier than non-speculative instructions. As a result, the memory latencies of these loads can be hidden more easily than for non-speculative loads.

3.4.2 Using Data Speculation in the Intel® Itanium® Architecture

Data speculation in the Itanium architecture uses a special load instruction (`ld.a`) called an *advanced load* instruction and an associated check instruction (`chk.a` or `ld.c`) to validate data-speculated results.

When the `ld.a` instruction is executed, an entry is allocated in a hardware structure called the Advanced Load Address Table (ALAT). The ALAT is indexed by physical register number and records the load address, the type of the load, and the size of the load.

A check instruction must be executed before the result of an advanced load can be used by any non-speculative instruction. The check instruction must specify the same register number as the corresponding advanced load.

When a check instruction is executed, the ALAT is searched for an entry with the same target physical register number and type. If an entry is found, execution continues normally with the next instruction.

If no matching entry is found, the speculative results need to be recomputed:

- Use a `chk.a` if a load and some of its uses are speculated. The `chk.a` jumps to compiler-generated recovery code to re-execute the load and dependent instructions.
- Use a `ld.c` if no uses of the load are speculated. The `ld.c` reissues the load.

Entries are removed from the ALAT due to:

- Stores that write to addresses overlapping with ALAT entries.
- Other advanced loads that target the same physical registers as ALAT entries.
- Implementation-defined hardware or operating system conditions needed to maintain correctness.
- Limitations of the capacity, associativity, and matching algorithm used for a given implementation of the ALAT.

3.4.2.1 Advanced Load Example

Advanced loads can reduce the critical path of a sequence of instructions. In the code below, a load and store may access conflicting memory addresses:

```
st8     [r4]=r12       // Cycle 0: ambiguous store
ld8     r6=[r8];;      // Cycle 0: load to advance
add     r5=r6,r7;;     // Cycle 2
st8     [r18]=r5       // Cycle 3
```

On the generic machine model, the code above would execute in four cycles, but it can be rewritten using an advanced load and check:

```
ld8.a   r6=[r8]        // Cycle -2 or earlier

// Other instructions

st8     [r4]=r12       // Cycle 0: ambiguous store
ld8.c   r6=[r8]        // Cycle 0: check load
add     r5=r6,r7;;     // Cycle 0
st8     [r18]=r5       // Cycle 1
```

The original load has been turned into a check load, and an advanced load has been scheduled above the ambiguous store. If the speculation succeeds, the execution time of the remaining non-speculative code is reduced because the latency of the advanced load is hidden.

3.4.2.2 Recovery Code Example

Consider again the non-speculative code from the last section:

```
st8     [r4]=r12       // Cycle 0: ambiguous store
ld8     r6=[r8];;      // Cycle 0: load to advance
add     r5=r6,r7;;     // Cycle 2
st8     [r18]=r5       // Cycle 3
```

The compiler could move up not only the load, but also one or more of its uses. This transformation uses a `chk.a` rather than a `ld.c` instruction to validate the advanced load. Using the same example code sequence but now advancing the `add` as well as the `ld8` results in:

```
        ld8.a   r6=[r8];;   // Cycle -3

        // other instructions

        add     r5=r6,r7    // Cycle -1: add that uses r6

        // Other instructions

        st8     [r4]=r12    // Cycle 0
        chk.a   r6,recover // Cycle 0: check
back: // Return point from jump to recover
        st8     [r18]=r5    // Cycle 0
```

Recovery code must also be generated:

```
recover:
        ld8     r6=[r8] ;; // Reload r6 from [r8]
        add     r5=r6,r7    // Re-execute the add
        br      back        // Jump back to main code
```

If the speculation fails, the check instruction branches to the label `recover` where the speculated code is re-executed. If the speculation succeeds, execution time of the transformed code is three cycles less than the original code.

3.4.2.3 Terminology Review

Terms related to speculation, such as *advanced loads* and *check loads*, have well-defined meanings in the Itanium architecture. The terms below were introduced in the preceding sections:

- Data speculative load

  A speculative load that is statically scheduled prior to one or more stores upon which it may be dependent. The data speculative load instruction is `ld.a`.

- Advanced load

  A data speculative load.

- Check load

  An instruction that checks whether a corresponding advanced load needs to be re-executed and does so if required. The check load instruction is `ld.c`.

- Advanced load check

  An instruction that takes a register number and an offset to a set of compiler-generated instructions to re-execute speculated instructions when necessary. The advanced load check instruction is `chk.a`.

- Recovery code

  Program code that is branched to by a speculation check. Recovery code repeats a load and chain of dependent instructions to recover from a speculation failure.

3.4.3 Using Control Speculation in the Intel® Itanium® Architecture

The check to determine if control speculation was successful is similar to that for data speculation.

3.4.3.1 The NaT Bit

The Not A Thing (NaT) bit is an extra bit on each of the general registers. A register NaT bit indicates whether the content of a register is valid. If the NaT bit is set to one, the register contains a deferred exception token due to an earlier speculation fault. In a floating-point register, the presence of a special value called the NaTVal signals a deferred exception.

During a control speculative load, the NaT bit on the destination register of the load may be set if an exception occurs and it is deferred. The exact set of events and exceptions that cause an exception to be deferred (thus causing the NaT bit to be set), depends in part upon operating system policy. When a speculative instruction reads a source register that has its NaT bit set, NaT bits of the target registers of that instruction are also set. That is, NaT bits are propagated through dependent computations.

3.4.3.2 Control Speculation Example

When a control speculative load is scheduled, the compiler must insert a speculative check, `chk.s`, along all paths on which results of the speculative load are consumed. If a non-speculative instruction (other than a `chk.s`) reads a register with its NaT bit set, a NaT consumption fault occurs, and the operating system will terminate the program.

The code sequence below illustrates a basic use of control speculation:

```
(p1)    br.cond some_label      // Cycle 0
        ld8     r1=[r5];;       // Cycle 1
        add     r2=r1,r3        // Cycle 3
```

This code can be rewritten using a control speculative load and check. The check can be placed in the same basic block as the original load:

```
        ld8.s   r1=[r5];;       // Cycle -2

        // Other instructions

(p1)    br.cond some_label      // Cycle 0
        chk.s   r1,recovery     // Cycle 0
        add     r2=r1,r3        // Cycle 0
```

Until a speculation check is reached dynamically, the results of the control speculative chain of instructions cannot be stored to memory or otherwise accessed non-speculatively without the possibility of a fault. If a speculation check is executed and the NaT bit on the checked register is set, the processor will branch to recovery code pointed to by the check instruction.

It is also possible to test for the presence of set NaT bits and NaTVals using the test NaT (`tnat`) and floating-point class (`fclass`) instructions.

Although every speculative computation needs to be checked, this does not mean that every speculative load requires its own `chk.s`. Speculative checks can be optimized by taking advantage of the propagation of NaT bits through registers as described in Section 3.5.6.

3.4.3.3 Spills, Fills and the UNAT Register

Saving and restoring of registers that may have set NaT bits is enabled by `st8.spill` and `ld8.fill` instructions and the User NaT Collection application register (UNAT).

The "spill general register and NaT" instruction, `st8.spill`, saves eight bytes of a general register to memory and writes its NaT bit into the UNAT. Bits 8:3 of the memory address of the store determine which UNAT bit is written with the register NaT value. The "fill general register" instruction, `ld8.fill`, reads eight bytes from memory into a general register and sets the register NaT bit according to the value in the UNAT. Software is responsible for saving and restoring the UNAT contents to ensure correct spilling and filling of NaT bits.

The corresponding floating-point instructions, `stf.spill` and `ldf.fill`, save and restore floating-point registers in floating-point register format without surfacing exceptions due to NaTVals.

3.4.3.4 Terminology Review

The terms below are related to control speculation:

- Control speculative load

  A speculative load that is scheduled prior to an earlier controlling branch. References to "speculative loads" without qualifiers generally refer to control speculative loads and not data speculative loads. Loads using the `ld.s` instruction are control speculative loads.

- Speculation check

  An instruction that checks whether a speculative instruction has deferred an exception. Speculation check instructions include labels that point to compiler-generated recovery code. The speculation check instruction is `chk.s`.

- Recovery code

  Code executed to recover from a speculation failure. Control speculative recovery code is analogous to data speculative recovery code.

3.4.4 Combining Data and Control Speculation

A load that is both data and control speculative is called a *speculative advanced load*. The `ld.sa` instruction performs all the operations of both a speculative load and an advanced load. An ALAT entry will not be allocated if this type of load generates a deferred exception token, so an advanced load check instruction (`chk.a`) is sufficient to check for both interference from subsequent stores and for deferred exceptions.

3.5 Optimization of Memory References

Speculation can increase parallelism and help to hide latency by enabling more code motion than can be performed on traditional architectures. Speculation can increase the application of traditional loop optimizations such as invariant code motion and common subexpression elimination. The Itanium architecture also offers post-increment loads and stores that improve instruction throughput without increasing code size.

Memory reference optimization should take several factors into account including:

- Difference between the execution costs of speculative and non-speculative code.
- Code size.
- Interference probabilities and properties of the ALAT (for data speculation).

The remainder of this chapter discusses these factors and optimizations relating to memory accesses.

3.5.1 Speculation Considerations

The use of data speculation requires more attention than the use of control speculation. In part this is due to the fact that one control speculative load cannot inadvertently cause another control speculative load to fail. Such an effect is possible with data speculative loads since the ALAT has limited capacity and the replacement policy of ALAT entries is implementation dependent. For example, if an advanced load is issued and there are no unused ALAT entries, the hardware may choose to invalidate an existing entry to make room for a new one.

Moreover, exceptions associated with control speculative calculations are uncommon in correct code since they are related to events such as page faults and TLB misses. However, excessive control speculation can be expensive as associated instructions fill issue slots.

Although the static critical path of a program may be reduced by the use of data speculation, the following factors contribute to the benefit/dynamic cost of data speculation:

- The probability that an intervening store will interfere with an advanced load.
- The cost of recovering from a failed advanced load.
- The specific microarchitectural implementation of the ALAT: its size, associativity, and matching algorithm.

Determining interference probabilities can be difficult, but dynamic memory profiling can help to predict how often ambiguous loads and stores will conflict.

When using advanced loads, there should be case-by-case consideration as to whether advancing only a load and using a `ld.c` might be preferable to advancing both a load and its uses, which would require the use of the potentially more expensive `chk.a`.

Even when recovery code is not executed, its presence extends the lifetimes of registers used in data and control speculation, thus increasing register pressure and possibly the cost of register movement by the Register Stack Engine (RSE). See Section 3.5.3 for information on considerations for recovery code placement.

3.5.2 Data Interference

Data references with *low* interference probabilities and *high* path probabilities can make the best use of data speculation. In the pseudo-code below, assume the probabilities that the stores to `*p1` and `*p2` conflict with `var` are independent.

```
*p1 =      /* Prob interference = 0.30 */
. . .
*p2 =      /* Prob interference = 0.40 */
. . .
= var      /* Load to be advanced */
```

If the compiler advances the load from var above the stores to pointers `p1` and `p2`, then:

```
Prob that stores to p1 or p2 interfere with var
    = 1.0 - (Prob p1 will not interfere with var *
            Prob p2 will not interfere with var)
    = 1.0 - (0.70 * 0.60)
    = 0.58
```

Given the interference probabilities above, there is a 58% probability at least one of `p1` and `p2` will interfere with a load from `var` if it is advanced above both of them. A compiler can use traditional heuristics concerning data interference and interprocedural memory access information to estimate these probabilities.

When advancing loads past function calls, the following should be considered:

- If a called function has many stores in it, it is more likely that actual or aliased ALAT conflicts will occur.
- If other advanced loads are executed during the function call, it is possible that their physical register numbers will either be identical or conflict with ALAT entries allocated from calls in parent functions.
- If it is unknown whether a large number of advanced loads will be executed by the called routines, then the possibility that the capacity of that ALAT may be exceeded must be considered.

3.5.3 Optimizing Code Size

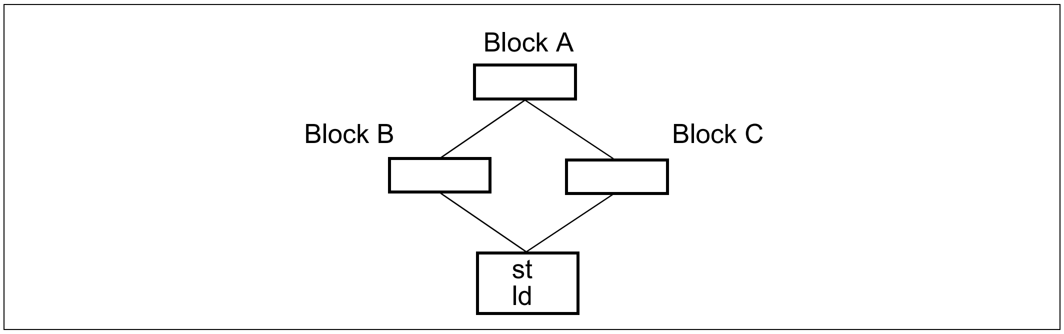
Part of the decision of when to speculate should involve consideration of any possible increases in code size. *Such consideration is not particular to speculation, but to any transformations that cause code to be duplicated, such as loop unrolling, procedure inlining, or tail duplication.* Techniques to minimize code growth are discussed later in this section.

In general, control speculation increases the dynamic code size of a program since some of the speculated instructions are executed and their results are never used. Recovery code associated with control speculation primarily contributes to the static size of the binary since it is likely to be placed out-of-line and not brought into cache until a speculative computation fails (uncommon for control speculation).

Data speculation has a similar effect on code size except that it is less likely to compute values that are never used since most non-control speculative data speculative loads will have their results checked. Also, since control speculative loads only fail in uncommon situations such as deferred data related faults (depending on operating system configuration), while data speculative loads can fail due to ALAT conflicts, actual memory conflicts, or aliasing in the ALAT, the decision as to where to place recovery code for advanced loads is more difficult than for control speculation and should be based on the expected conflict rate for each load.

As a general rule, efficient compilers will attempt to minimize code growth related to speculation. As an example, moving a load above the join of two paths may require duplication of speculative code on every path. The flow graph depicted in Figure 3-3 and the explanation shows how this could arise.

Figure 3-3. Minimizing Code Size During Speculation



If the compiler or programmer advanced the load up to block B from its original non-speculative position, all speculative code would need to be duplicated in both blocks B and C. This duplicated code might be able to occupy NOP slots that already exist. But if space for the code is not already available, it might be preferable to advance the load to block A since only one copy would be required in this case.

3.5.4 Using Post-increment Loads and Stores

Post-increment loads and stores can improve performance by combining two operations in a single instruction. Although the text in this section mentions only post-increment loads, most of the information applies to stores as well.

Post-increment loads are issued on M-units and can increment their address register by either an immediate value or by the contents of a general register. The following pseudo-code that performs two loads:

```
ld8     r2=[r1]
add     r1=1,r1 ;;
ld8     r3=[r1]
```

can be rewritten using a post-increment load:

```
ld8     r2=[r1],1 ;;
ld8     r3=[r1]
```

Post-increment loads may not offer direct savings in dependency path height, but they are important when calculating addresses that feed subsequent loads:

- A post-increment load avoids code size expansion by combining two instructions into one.
- Adds can be issued on either I-units or M-units. When a program combines an add with a load, an I-unit or M-unit resource remains available that otherwise would have been consumed. Thus, throughput of dependent adds and loads can be doubled by using post-increment loads.

A disadvantage of post-increment loads is that they create new dependencies between post-increment loads and the operations that use the post-increment values. In some cases, the compiler may wish to separate post-increment loads into their component instructions to improve the overall schedule. Alternatively, the compiler could wait until after instruction scheduling and then opportunistically find places where post-increment loads could be substituted for separate load and add instructions.

3.5.5 Loop Optimization

In cyclic code, speculation can extend the use of classical loop optimizations like invariant code motion. Examine this pseudo-code:

```
while (cond) {
    c = a + b; // Probably loop invariant
    *ptr++ = c;// May point to a or b
}
```

The variables a and b are probably loop invariant; however, the compiler must assume the stores to *ptr will overwrite the values of a and b unless analysis can guarantee that this can never happen. The use of advanced loads and checks allows code that is likely to be invariant to be removed from a loop, even when a pointer cannot be disambiguated:

```
        ld4.a   r1 = [&a]
        ld4.a   r2 = [&b]
        add     r3 = r1,r2  // Move computation out of loop
        while (cond) {
            chk.a.nc r1, recover1
    L1:     chk.a.nc r2, recover2
    L2:     *p++ = r3
        }
```

At the end of the module:

```
recover1:  // Recover from failed load of a
        ld4.a   r1 = [&a]
        add     r3 = r1, r2
        br.sptk L1          // Unconditional branch

recover2:  // Recover from failed load of b
        ld4.a   r2 = [&b]
        add     r3 = r1, r2
        br.sptk L2          // Unconditional branch
```

Using speculation in this loop hides the latency of the calculation of c whenever the speculated code is successful.

Since checks have both a clear (clr) and no clear (nc) form, the programmer must decide which to use. This example shows that when checks are moved out of loops, the no clear version should be used. This is because the clear (clr) version will cause the corresponding ALAT entry to be removed (which would cause the next check to that register to fail).

3.5.6 Minimizing Check Code

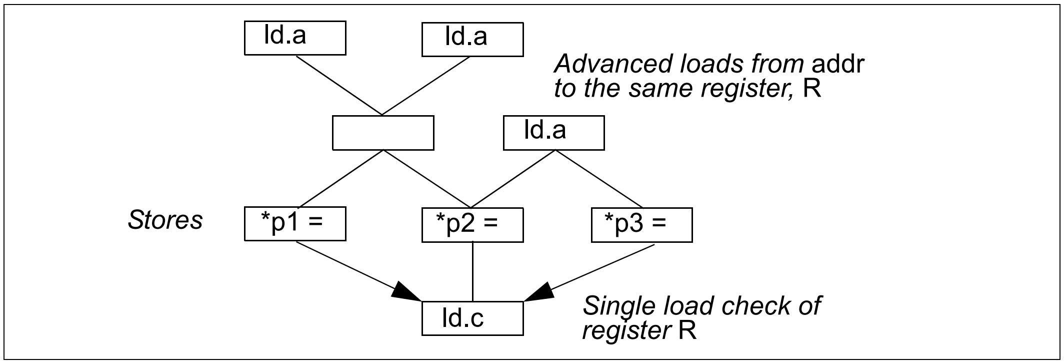
Checks of speculative loads can sometimes be combined to reduce code size. The propagation of NaT bits and NaTVals via speculative instructions can permit a single check of a speculative result to replace multiple intermediate checks. The code below demonstrates this optimization potential:

```
ld4.s   r1=[r10]    // Speculatively load to r1
ld4.s   r2=[r20]    // Speculatively load to r2
add     r3=r1,r2;;  // Add two speculative values

// Other instructions

chk.s   r3,imm21    // Check for NaT bit in r3
st4     [r30]=r1    // Store r1
st4     [r40]=r2    // Store r2
st4     [r50]=r3    // Store r3
```

Only the result register, `r3`, needs to be checked before stores of any of `r1`, `r2`, or `r3`. If a NaT bit were set at the time of the control speculative loads of `r1` or `r2`, the NaT bit would have been propagated to `r3` from `r1` or `r2` via the `add` instruction.

Another way to reduce the amount of check code is to use control flow analysis to avoid issuing extra `ld.c` or `ld.a` instructions. For example, the compiler can schedule a single check where it is known to be reached by all copies of the advanced load. The portion of a flow graph shown in Figure 3-4 demonstrates where this technique might be applied.

Figure 3-4. Using a Single Check for Three Advanced Loads



A single check in the lowermost block shown for all of the advanced loads is correct if both of these conditions are met:

- The lowermost block post-dominates all of the blocks with advanced loads from location `addr`.
- The lowermost block precedes any uses of the advanced loads from `addr`.

3.6 Summary

The examples in this chapter show where the Itanium architecture can take advantage of existing techniques like dynamic profiling and disambiguation. Special architectural support allows implementation of speculation in common scenarios in which it would normally not be allowed. Speculation, in turn, increases ILP by making greater code motion possible, thus enhancing traditional optimizations such as those involving loops.

Even though the speculation model can be applied in many different situations, careful cost and benefit analysis is needed to insure best performance.

§

Predication, Control Flow, and Instruction Stream 4

4.1 Overview

This chapter is divided into three sections that describe optimizations related to predication, control flow, and branch hints as follows:

- The **predication** section describes if-conversion, predicate usage, and code scheduling to reduce the affects of branching.
- The **control flow optimization** section describes optimizations that collapse and converge control flow by using parallel compares, multiway branches, and multiple register writers under predicate.
- The **branch and prefetch hints** section describes how hints are used to improve branch and prefetch performance.

4.2 Predication

Predication allows the compiler to convert control dependencies into data dependencies. This section describes several sources of branch-related performance considerations, followed by a summary of predication mechanism, followed by a series of descriptions of optimizations and techniques based on predication.

4.2.1 Performance Costs of Branches

Branches can decrease application performance by consuming hardware resources for prediction at execution time and by restricting instruction scheduling freedom during compilation.

4.2.1.1 Prediction Resources

Branch prediction resources include branch target buffers, branch prediction tables, and the logic used to control these resources. The number of branches that can accurately be predicted is limited by the size of the buffers on the processor, and such buffers tend to be small relative to the total number of branches executed in a program.

This limitation means that branch intensive code may have a large portion of its execution time spent due to contention for prediction resources. Furthermore, even though the size of the predictors is a primary factor in determining branch prediction performance, some branches are best predicted with different types of predictors. For example, some branches are best predicted statically while others are more suitably predicted dynamically. Of those predicted dynamically, some are of greater importance than others, such as loop branches.

Since the cost of a misprediction is generally proportional to pipeline length, good branch prediction is essential for processors with long instruction pipelines. Thus, optimizing the use of prediction resources can significantly improve the overall performance of an application.

Suppose, for instance, that the conditional in the code below is mispredicted 30% of the time and branch mispredictions incur a ten cycle penalty. On average, the mispredicted branch will add three cycles to each execution of the code sequence (30% * 10 cycles):

```
if (r1)
    r2 = r3 + r4;
else
    r7 = r6 - r5;
```

Equivalent Itanium architecture-based code that has not been optimized is shown below. It requires five instructions including two branches and executes in two cycles, not including potential misprediction or taken-branch penalty cycles:

```
        cmp.eq  p1,p2=r1,r0     // Cycle 0
(p1)    br.cond else_clause             // Cycle 0
        add     r2=r3,r4        // Cycle 1
        br      end_if          // Cycle 1
else_clause:
        sub     r7=r6,r5        // Cycle 1
end_if:
```

Using the information above, this code will take five cycles to execute on average even thought the critical path is only two cycles long (2 cycles + (30% * 10 cycles) = 5). If the branch misprediction penalty could be eliminated (either by reducing contention for resources or by removing the branch itself), performance of the code sequence would improve by a factor of two.

4.2.1.2 Instruction Scheduling

Branches limit the ability of the compiler to move instructions that alter memory state or that can raise exceptions, because instructions in a program are control dependent on all lexically enclosing branches. In addition to the control dependencies, compound conditionals can take several cycles to compute and may themselves require intermediate branches in languages like C that require short-circuit evaluation.

Control speculation is the primary mechanism used to perform global code motion for Itanium architecture-based compilers. However, when an instruction does not have a speculative form or the instruction could potentially corrupt memory state, control speculation may be insufficient to allow code motion. Thus, techniques that allow greater freedom in code motion or eliminate branches can improve the compiler’s ability to schedule instructions.

4.2.2 Predication in the Intel® Itanium® Architecture

Now that the performance implications of branching have been described, this section overviews predication in the Itanium architecture – the primary mechanism used by optimizations described in this section.

Almost all Itanium instructions can be tagged with a guarding predicate. If the value of the guarding predicate is false at execution time, then the predicated instruction's architectural updates are suppressed, and the instruction behaves like a nop. If the predicate is true, then the instruction behaves as if it were unpredicated. There are a small number of instructions such as unconditional compares and floating-point square-root and reciprocal approximate instructions whose qualifying predicate do not operate as described above. See *Part I:, "Application Architecture Guide"* for additional information.

The following sequence shows a set of predicated instructions:

```
(p1)    add     r1=r2,r3
(p2)    ld8     r5=[r7]
(p3)    chk.s   r4,recovery
```

To set the value of a predict register, the architecture provides compare and test instructions such as those as shown below.

```
        cmp.eq  p1,p2=r5,r6
        tbit    p3,p4=r6,5
```

Additionally, a predicate almost always requires a stop to separate its producing instruction and its use:

```
        cmp.eq  p1,p2=r1,r2;;
(p1)    add     r1=r2,r3
```

The only exception to this rule involves an integer compare or test instruction that sets a predicate that is used as the condition for a subsequent branch instruction:

```
        cmp.eq  p1,p2=r1,r2   // No stop required
(p1)    br.cond some_target
```

4.2.3 Optimizing Program Performance Using Predication

This section describes predication-related optimizations, their use, and basic performance analysis techniques. Following are descriptions of optimizations including if-conversion, misprediction elimination, off-path predication, upward code motion, and downward code motion.

4.2.3.1 Applying if-Conversion

One of the most important optimizations enabled by predication is the complete removal of branches from some program sequences. Without predication, the pseudo-code below would require a branch instruction to conditionally jump around the if-block code:

```
if (r4) {
    add  r1=r2,r3
    ld8  r6=[r5]
}
```

Using predication, the sequence can be written without a branch:

```
        cmp.ne  p1,p0=r4,0 ;;// Set predicate reg
(p1)    add     r1=r2,r3
(p1)    ld8     r6=[r5]
```

The process of predicating instructions in conditional blocks and removing branches is referred to as *if-conversion*. Once if-conversion has been performed, instructions can be scheduled more freely because there are fewer branches to limit code motion, and there are fewer branches competing for issue slots.

In addition to removing branches, this transformation will make dynamic instruction fetching more efficient since there are fewer possibilities for control flow changes. Under more complex circumstances, several branches can be removed. The following C code sequence:

```
if (r1)
    r2 = r3 + r4;
else
    r7 = r6 - r5;
```

can be rewritten in Itanium architecture-based assembly code without branches as:

```
        cmp.ne  p1,p2 = r1,0;;
(p1)    add     r2 = r3,r4
(p2)    sub     r7 = r6,r5
```

Since instructions from opposite sides of the conditional are predicated with complementary predicates they are guaranteed not to conflict, hence the compiler has more freedom when scheduling to make the best use of hardware resources. The compiler could also try to schedule these statements with earlier or later code since several branches and labels have been removed as part of if-conversion.

Since the branches have been removed, no branch misprediction is possible and there will be no pipeline bubbles due to taken branches. Such effects are significant in many large applications, and these transformations can greatly reduce branch-induced stalls or flushes in the pipeline.

Thus, comparing the cost of the code above with the non-predicated version above shows that:

- Non-predicated code consumes: 2 cycles + (30% * 10 cycles) = 5 cycles.
- Predicated code consumes: 2 cycles.

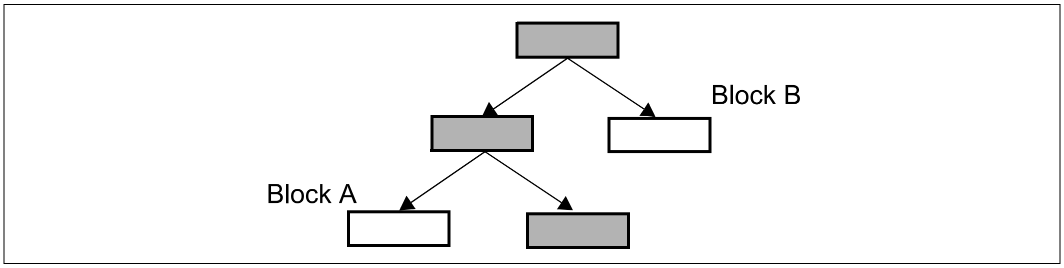
In this case, predication saves an average of three cycles.

4.2.3.2 Off-path Predication

If a compiler has dynamic profile information, it is possible to form an instruction schedule based on the control flow path that is most likely to execute – this path is called the main trace. In some cases, execution paths not on the main trace are still executed frequently, and thus it may be beneficial to use predication to minimize their critical paths as well.

The main trace of a flow graph is highlighted in Figure 4-1. Although blocks A and B are not on the main trace, suppose they are executed a significant number of times.

Figure 4-1. Flow Graph Illustrating Opportunities for Off-path Predication



If some of the instructions in block A or block B can be included in the main trace without increasing its critical path, then techniques of upward code motion can be applied to reduce the critical path through blocks A and B when they are taken. An example of how to use predication to implement upward code motion is given in the next section.

4.2.3.3 Upward Code Motion

When traditional control speculation is inadequate, it may still be possible to predicate an instruction and move it up or down in the schedule to reduce dependency height. This is possible because predicating an instruction replaces a control dependency with a data dependency. If the data dependency is less constraining than the control dependency, such a transformation may improve the instruction schedule.

Given the Itanium architecture-based assembly sequence below, the store instruction cannot be moved above the enclosing conditional instruction because it could cause an address fault or other exception, depending upon the branch direction:

```
(p1)    br.cond some_label      // Cycle 0
        st4     [r34] = r23     // Cycle 1
        ld4     r5 = [r56]      // Cycle 1
        ld4     r6 = [r57]      // Cycle 2:no cycle 1 M’s
```

One reason why it might be desirable to move the store instruction up is to allow loads below it to move up.

Note: Ambiguous stores are barriers beyond which normal loads cannot move. In this case, moving the store also frees up an M-unit slot. To rewrite the code so that the store comes before the branch, p2 has been assigned the complement of p1:

```
(p2)    st4     [r34] = r23     // Cycle 0
(p2)    ld4     r5 = [r56]      // Cycle 0
(p1)    br.cond some_label      // Cycle 0
        ld4     r6 = [r57]      // Cycle 1
```

Since the store is now predicated, no faults or exceptions are possible when the branch is taken, and memory state is only updated if and when the original home block of the store is entered. Once the store is moved, it is also possible to move the load instruction without having to use advanced or speculative loads (as long as r5 is not live on the taken branch path).

4.2.3.4 Downward Code Motion

As with upward code motion, downward code motion is normally difficult in the presence of stores. The next example shows how code can be moved downward past a label, a transformation that is often unsafe without predication:

```
        ld8     r56 = [r45];;  // Cycle 0: load
        st4     [r23] = r56;;  // Cycle 2: store
label_A:
        add     ...            // Cycle 3
        add     ...
        add     ...
        add     ...;;
```

In the code above, suppose the latency between the load and the store is two clocks. Assuming the load instruction cannot be moved upward due to other dependencies, the only way to schedule the instructions so that the load latency is covered is to move the store downward past the label.

The following code demonstrates the overall idea of using predicates to enable downward code motion. In actual compiler-generated code, the predicates that are explicitly computed in this example might already be available in predicate registers and not require extra instructions.

```
        // Point which “dominates” label_A
        cmp.ne p1,p0 = r0,r0   // Initialize p1 to false

        // Other instructions

        cmp.eq  p1,p0 = r0,r0  // Initialize p1 to true
        ld8     r56=[r45];;    // Cycle 0
label_A:
        add     ...            // Cycle 1
        add     ...
        add     ...
        add     ...;;
(p1)    st4      [r23]=r56     // Cycle 2
```

Here, downward code motion saves one cycle. There are examples of more sophisticated situations involving cyclic scheduling, other store-constrained code motion, or pulling code from outside loops into them, but they are not described here.

4.2.3.5 Cache Pollution Reduction

Loads and stores with predicates that are false at runtime are generally likely not to cause any cache lines to be removed, replaced, or brought in. Also, no extra instructions or recovery code are required as would be necessary for control or data speculation. Therefore, when the use of predication yields the same critical path length as data or control speculation, it is almost always preferable to use predication.

4.2.4 Predication Considerations

Even though predication can have a variety of beneficial effects, there are several cases where the use of predication should be carefully considered. Such cases are usually associated with execution paths that have unbalanced total latencies or over-usage of a particular resource such as those associated with memory operations.

4.2.4.1 Unbalanced Execution Paths

The simple conditional below has an unbalanced flow-dependency height. Suppose that non-predicated assembly for this sequence takes two clocks for the if-block and approximately 18 clocks if we assume a setf takes 8 clocks, a getf takes 2 clocks, and an xma takes 6 clocks:

```
if (r4)     // 2 clocks
    r3 = r2 + r1;
else        // 18 clocks
    r3 = r2 * r1;
f (r3);     // An integer use of r3
```

If-converted Itanium architecture-based code is shown below. The cycle numbers shown depend upon the values of p1 and p2 and assume the latencies shown:

```
        // Issue cycle if p2 is:TrueFalse
        cmp.ne  p1,p2=r4,r0;;  //  0   0
(p1)    add     r3=r2,r1       //  1   1
(p2)    setf    f1=r1          //  1   1
(p2)    setf    f2=r2;;        //  1   1
(p2)    xma.l   f3=f1,f2,f0;;  //  9   2
(p2)    getf    r3=f3;;        //  15  3
(p2)    use of r3              //  17  4
```

This code takes 18 cycles to complete if p2 is true and five cycles if p2 is false. When analyzing such cases, consider execution weights, branch misprediction probabilities, and prediction costs along each path.

In the three scenarios presented below, assume a branch misprediction costs ten cycles. No instruction cache or taken-branch penalties are considered.

4.2.4.2 Case 1

Suppose the if-clause is executed 50% of the time and the branch is never mispredicted. The average number of clocks for:

- Unpredicated code is: (2 cycles * 50%) + (18 cycles * 50%) = 10 clocks
- Predicated code is: (5 cycles * 50%) + (18 cycles * 50%) = 11.5 clocks

In this case, if-conversion would *increase* the cost of executing the code.

4.2.4.3 Case 2

Suppose the if-clause is executed 70% of the time and the branch mispredicts 10% if the time with mispredicts costing 10 clocks. The average number of clocks for:

- Unpredicated code is:

  (2 cycles * 70%) + (18 cycles * 30%) + (10 cycles * 10%) = 7.8 clocks

- Predicated code is:

  (5 cycles * 70%) + (18 cycles * 30%) = 8.9 clocks

In this case, if-conversion still would *increase* the cost of executing the code.

4.2.4.4 Case 3

Suppose the if-clause is executed 30% of the time and the branch mispredicts 30% of the time. The average number of clocks for:

- Unpredicated code is:

  (2 cycles * 30%) + (18 cycles * 70%) + (10 cycles * 30%) = 16.2 clocks

- Predicated code is:

  (5 cycles * 30%) + (18 cycles * 70%) = 14.1 clocks

In this case, if-conversion would *decrease* the execution cost by more than two clocks, on average.

4.2.4.5 Overlapping Resource Usage

Before performing if-conversion, the programmer must consider the execution resources consumed by predicated blocks in addition to considering flow-dependency height. The *resource availability height* of a set of instructions is the minimum number of cycles taken considering only the execution resources required to execute them.

The code below is derived from an if-then-else statement. Given the generic machine model that has only two load/store (M) units. If a compiler predicates and combines these two blocks, then the resource availability height through the block will be four clocks since that is the minimum amount of time necessary to issue eight memory operations:

```
then_clause:
        ld      r1=[r21]    // Cycle 0
        ld      r2=[r22]    // Cycle 0
        st      [r32]=r3    // Cycle 1
        st      [r33]=r4 ;; // Cycle 1
        br      end_if
else_clause:
        ld      r3=[r23]    // Cycle 0
        ld      r4=[r24]    // Cycle 0
        st      [r34]=r5    // Cycle 1
        st      [r35]=r6 ;; // Cycle 1
end_if:
```

As with the example in the previous section, assuming various misprediction rates and taken branch penalties changes the decision as to when to predicate and when not to predicate. One case is illustrated below.

4.2.4.6 Case 1

Suppose the branch condition mispredicts 10% of the time and that the predicated code takes four clocks to execute. The average number of clocks for:

- Non-predicated code is: (10 cycles * 10%) + 2 cycles = 3 cycles
- Predicated code is: 4 cycles

Predicating this code would *increase* execution time even though the flow dependency heights of the branch paths are equal.

4.2.5 Guidelines for Removing Branches

The following if-conversion guidelines apply to cases where only local behavior of the code and its execution profile are known:

1. The flow dependency and resource availability heights of both paths must be considered when deciding whether to predicate or not.
2. If if-conversion increases the length of *any control path* through the original code sequence, careful analysis using profile or misprediction data must be performed to ensure that execution time of the converted code is equivalent to or better than unpredicated code.
3. If if-conversion removes a branch that is mispredicted a significant percentage of the time, the transformation frequently pays off even if the blocks are significantly unbalanced since mispredictions are very expensive.
4. If the flow-dependeny heights of the paths being if-converted are nearly equal and there are sufficient resources to execute both streams simultaneously, if-conversion is often advantageous.

Although these guidelines are useful for optimizing segments of code, the behavior of some programs is limited by non-local effects such as overall branch behavior, sensitivity to code size, percentage of time spent servicing branch mispredictions, etc. In these situations, the decision to use if-convert or perform other speculative transformation becomes more involved.

4.3 Control Flow Optimizations

A common occurrence in programs is for several control flows to converge at one point or for multiple control flows to start from one point. In the first case, multiple flows of control are often computing the value of the same variable or register and the join point represents the point at which the program needs to select the correct value before proceeding. In the second case, multiple flows may begin at a point where several independent paths are taken based on a set of conditions.

In addition to these multiway joins and branches, the computation of complex compound conditions normally requires a tree-like computation to reduce several conditions into one. The Itanium architecture provides special instructions that allow such conditions to be computed in fewer tree levels.

A third control-flow related optimization uses predication to improve instruction fetching by if-conversion to generate straight-line sequences that can be efficiently fetched. The use and optimization of these cases is described in the remainder of this section.

4.3.1 Reducing Critical Path with Parallel Compares

The computation of the compound branch condition shown below requires several instructions on processors without special instructions:

```
if ( rA || rB || rC || rD ) {
    /* If-block instructions */
}
/* after if-block */
```

The pseudo-code below, shows one possible solution uses a sequence of branches:

```
        cmp.ne  p1,p0 = rA,0
        cmp.ne  p2,p0 = rB,0
(p1)    br.cond if_block
(p2)    br.cond if_block
        cmp.ne  p3,p0 = rC,0
        cmp.ne  p4,p0 = rD,0
(p3)    br.cond if_block
(p4)    br.cond if_block
        // after if-block
```

On many implementations based on the Itanium architecture, this sequence is likely to require at least two cycles to execute if all the conditions are false, plus the possibility of more cycles due to one or more branch mispredictions. Another possible sequence computes an or-tree reduction:

```
        or      r1 = rA,rB
        or      r2 = rC,rD;;
        or      r3 = r1,r2;;
        cmp.ne  p1,p2 = r3,0
(p1)    br      if_block
```

This solution requires three cycles to compute the branch condition which can then be used to branch to the if-block.

Note: It is also possible to predicate the if-block using `p1` to avoid branch mispredictions.

To reduce the cost of compound conditionals, the Itanium architecture has special *parallel compare* instructions to optimize expressions that have `and` and `or` operations. These compare instructions are special in that multiple `and`/`or` compare instructions are allowed to target the same predicate within a single instruction group. This feature allows the possibility that a compound conditional can be resolved in a single cycle.

For this usage model to work properly, the architecture requires that the programmer ensure that during any given execution of the code, that all instructions that target a given predicate register must either:

- Write the same value (0 or 1) or
- Do not write the target register at all.

This usage model means that sometimes a parallel compare may not update the value of its target registers and thus, unlike normal compares, the predicates used in parallel compares must be initialized prior to the parallel compare. Please see *Part I:, "Application Architecture Guide"* for full information on the operation of parallel compares.

Initialization code must be placed in an instruction group prior to the parallel compare. However, since the initialization code has no dependencies on prior values, it can generally be scheduled without contributing to the critical path of the code.

The instructions below shows how to generate code for the example above using parallel compares:

```
       cmp.ne     p1,p0 = r0,r0;; // initialize p1 to 0
       cmp.ne.or  p1,p0 = rA,r0
       cmp.ne.or  p1,p0 = rB,r0
       cmp.ne.or  p1,p0 = rC,r0
       cmp.ne.or  p1,p0 = rD,r0
(p1)   br.cond    if_block
```

It is also possible to use p1 to predicate the if-block in-line to avoid a possible misprediction. More complex conditional expressions can also be generated with parallel compares:

```
if ((rA < 0) && (rB == -15) && (rC > 0))
    /* If-block instructions */
```

The assembly pseudo-code below shows a possible sequence for the C code above:

```
cmp.eq      p1,p0=r0,r0;; // initialize p1 to 1
cmp.ne.and  p1,p0=rB,-15
cmp.ge.and  p1,p0=rA,r0
cmp.le.and  p1,p0=rC,r0
```

When used correctly, and or compares write both target predicates with the same value or do not write the target predicate at all. Another variation on parallel compare usage is where both the if and else part of a complex conditional are needed:

```
if ( rA == 0 || rB == 10 )
    r1 = r2 + r3;
else
    r4 = r5 - r6;
```

Parallel compares have an andcm variant that computes both the predicate and its complement simultaneously.

```
       cmp.ne         p1,p2 = r0,r0;; // initialize p1,p2
       cmp.eq.or.andcmp1,p2 = rA,r0
       cmp.eq.or.andcmp1,p2 = rB,10;;
(p1)   add            r1=r2,r3
(p2)   sub            r4=r5,r6
```

Clearly, these instructions can be used in other combinations to create more complex conditions.

4.3.2 Reducing Critical Path with Multiway Branches

While there are no special instructions to support branches with multiple conditions and multiple targets, the Itanium architecture has implicit support by allowing multiple consecutive B-slot instructions within an instruction group.

An example uses a basic block with four possible successors. The following Itanium architecture-based multi-target branch code uses a BBB bundle template and can branch to either block B, block C, block D, or fall through to block A:

```
label_AA:
        ... // Instructions in block AA
{ .bbb
(p1)    br.cond label_B
(p2)    br.cond label_C
(p3)    br.cond label_D
}
        // Fall through to A
label_A:
        ... // Instructions in block A
```

The ordering of branches is important for program correctness unless all branches are mutually exclusive, in which case the compiler can choose any ordering desired.

4.3.3 Selecting Multiple Values for One Variable or Register with Predication

A common occurrence in programs is for a set of paths that compute different values for the same variable to join and then continue. A variant of this is when separate paths need to compute separate results but could otherwise use the same registers since the paths are known to be complementary. The use of predication can optimize these cases.

4.3.3.1 Selecting One of Several Values

When several control paths that each compute a different value of a single variable meet, a sequence of conditionals is usually required to select which value will be used to update the variable. The use of predication can efficiently implement this code without branches:

```
switch (rW)
case 1:
    rA = rB + rC;
    break;
case 2:
    rA = rE + rF;
    break;
case 3:
    rA = rH - rI;
    break;
```

The entire switch-block above can be executed in a single cycle using predication if all of the predicates have been computed earlier. Assume that if `rW` equals 1, 2, or 3, then one of `p1`, `p2`, or `p3` is true, respectively:

```
(p1)    add     rA=rB,rC
(p2)    add     rA=rE,rF
(p3)    sub     rA=rH,rI;;
```

Without this predication capability, numerous branches or conditional move operations would be needed to collapse these values.

The Itanium architecture allows multiple instructions to target the same register in the same clock provided that only one of the instructions writing the target register is predicated true in that clock. Similar capabilities exist for writing predicate registers, as discussed in Section 4.3.1.

4.3.3.2 Reducing Register Usage

In some instances it is possible to use the same register for two separate computations in the presence of predication. This technique is similar to the technique for allowing multiple writers to store a value into the same register, although it is a register allocation optimization rather than a critical path issue.

After if-conversion, it is particularly common for sequences of instructions to be predicated with complementary predicates. The contrived sequence below shows instructions predicated by `p1` and `p2`, which are known by the compiler to be complementary:

```
(p1)    add     r1=r2,r3
(p2)    sub     r5=r4,r56
(p1)    ld8     r7=[r2]
(p2)    ld8     r9=[r6];;
(p1)    a use of r1
(p2)    a use of r5
(p1)    a use of r7
(p2)    a use of r9
```

Assuming registers `r1`, `r5`, `r7`, and `r9` are used for compiler temporaries, each of which is live only until its next use, the preceding code segment can be rewritten as:

```
(p1)    add     r1=r2,r3
(p2)    sub     r1=r4,r56   // Reuse r1
(p1)    ld8     r7=[r2]
(p2)    ld8     r7=[r6];;   // Reuse r7
(p1)    a use of r1
(p2)    a use of r1
(p1)    a use of r7
(p2)    a use of r7
```

The new sequence uses two fewer registers. With the 128 registers defined in the architecture, this may not seem essential, but reducing register use can still reduce program and register stack engine spills and fills that can be common in codes with high instruction-level parallelism.

4.3.4 Improving Instruction Stream Fetching

Instructions flow through the pipeline most efficiently when they are executed in large blocks with no taken branches. Whenever the instruction pointer needs to be changed, the hardware may have to insert bubbles into the pipeline either while the target prediction is taking place or because the target address is not computed until later in the pipeline.

By using predication to reduce the number of control flow changes, the fetching efficiency will generally improve. The only case where predication is likely to reduce instruction cache efficiency is when there is a large increase in the number of instructions fetched which are subsequently predicated off. Such a situation uses instruction cache space for instructions that compute no useful results.

4.3.4.1 Instruction Stream Alignment

For many processors, when a program branches to a new location, instruction fetching is performed on instruction cache lines. If the target of the branch does not start on a cache line boundary, then fetching from that target will likely not retrieve an entire cache line. This problem can be avoided if a programmer aligns instruction groups that cross more than one bundle so that the instruction groups do not span cache line boundaries. However, padding all labels would cause an unacceptable increase in code size. A more practical approach aligns only tops of loops and commonly entered basic blocks when the first instruction group extends across more than one bundle. That is, if both of the following conditions are true at some label L, then padding previous instruction groups so that `L` is aligned on a cache line boundary is recommended:

- The label is commonly branched to from out-of-line. Examples include tops of loops and commonly executed else clauses.
- The instruction group starting at label `L` extends across more than one bundle.

To illustrate, assume code at label `L` in the segment below is not cache-aligned and that a cache boundary occurs between the two bundles. If a program were to branch to `L`, then execution may split issue after the third add instruction even though there are no resource oversubscriptions or stops:

```
L:
{ .mii
        add     r1=r2,r3
        add     r4=r5,r6
        add     r7=r8,r9
}
{ .mfb
        ld8     r14=[r56] ;;
        nop.f
        nop.b
}
```

On the other hand, if `L` were aligned on an even-numbered bundle, then all four instructions at `L` could issue in one cycle.

4.4 Branch and Prefetch Hints

Branch and prefetch hints are architecturally defined to allow the compiler or hand coder to provide extra information to the hardware. Compared to hardware, the compiler has more time, looks at a wider instruction window (including the source), and performs more analysis. Transfer of this knowledge to the processor can help to reduce penalties associated with I-cache accesses and branch prediction.

Two types of branch-related hints are defined by the Itanium architecture: branch prediction hints and instruction prefetch hints. Branch prediction hints let the compiler recommend the resources (if any) that should be used to dynamically predict specific branches. With prefetch hints, the compiler can indicate the areas of the code that should be prefetched to reduce demand I-cache misses.

Hints can be specified as completers on branch (`br`) and move to branch register (abbreviated mov2br in this text since the actual mnemonic is `mov br=xx`). The hints on branch instructions are the easiest to use since the instruction already exists and the hint completer just has to be specified. mov2br instructions are used for indirect branches. The exact interpretation of these hints is implementation specific although the general behavior of hints is expected to be similar between processor generations.

It is also possible to re-write the hint fields on branches later using a binary rewriting tools. This can occur statically or at execution time based on profile data without changing the correctness of the program. This technique allows static hints to be tailored for usage patterns that may not be fully known at compilation time or when the binaries are first distributed.

4.5 Hints for Controlling Multi-threading

Some processors support multi-threading; that is, they support the simultaneous execution of multiple threads (multiple logical processors) through a common set of execution resources (data paths, functional units, TLBs, etc.). Functionally, each of these hardware threads fully implements the Itanium architecture; therefore, software need not be aware of multi-threading nor do anything special to support it. From performance standpoint, there are a few circumstances where it may be beneficial for software to provide information about its future resource requirements, which can be done with the `hint` instruction. Such a hint could allow the processor to optimize resource allocation among the hardware threads.

Note that, although not all implementations support all types of `hint` instruction, those that do not support them execute the hint instruction as a nop, and hence there is little penalty for software to provide these hints.

4.5.1 Wait Loops

Say a thread is waiting for another software thread to complete a task and, during that time, doesn't expect to need significant processor resources but would like to receive its fair share of resources once the task is complete. In such a situation, the waiting thread can communicate this information to the processor as a hint. This encourages the processor to allocate more processor resources to other threads of execution while this thread is waiting.

Typically, the completion signal in question is a store, by some other software thread, to a particular memory location. For example, a software thread may be waiting to acquire a spinlock and may have little work to do until such time as it is able to acquire the lock. A store to the spinlock in question may be an indication that the lock is now available for this software thread to acquire.

This scenario can be hinted to the processor by executing an advanced load (`ld.a` or `ld.sa`) to the address that this software thread is waiting on, and then by executing a `hint @pause` instruction (in a subsequent instruction group). This encourages the processor to devote more resources to other threads, yet if an entry is invalidated from this thread's ALAT, normal processor resource allocation is resumed for this thread.

Resource allocation within the processor eventually reverts to a fair allocation, so there's no need for software to hint that it is no longer in a wait loop. Conversely, while software is in such a wait loop, it would be best to re-execute the `hint @pause` as part of that loop, to continue to assert the hint for as long as that thread is waiting.

Note that if there is some high likelihood that the ALAT may contain a large number of valid entries upon entering into a wait loop, there may be some advantage to removing these (e.g., with an `invala` instruction) prior to executing the advanced load to the address to be waited on. This may reduce the restoration of resource allocation to this thread in cases where ALAT entries get invalidated other than the one for the address being waited on, hence providing more processor resources to other threads.

4.5.2 Idle Loops

Another situation where a software thread expects not to need significant processor resources for the next little while is when the software thread is executing an OS-kernel idle loop. It can provide this information to the processor also by executing a `hint @pause` instruction. This encourages the processor to allocate more processor resources to other threads of execution for the next while.

Resource allocation within the processor eventually reverts to a fair allocation, so there's no need for software to hint that it is no longer in an idle loop. Conversely, while software is in such an idle loop, it would be best to re-execute the `hint @pause` as part of that loop, to continue to assert the hint for as long as that thread is idle.

Note that if there is some high likelihood that the ALAT may contain a large number of valid entries upon entering into an idle loop, there may be some advantage to removing these (e.g., with an `invala` instruction) prior to entering the idle loop. This may reduce the restoration of resource allocation to this thread in cases where these ALAT entries get invalidated, hence providing more processor resources to other threads.

4.5.3 Critical Sections

The opposite case exists if software expects that, given extra resources for the next period of time, overall system performance and throughput would be optimized. For example, this software thread may be about to acquire a highly contested spinlock and enter a critical section of code, and expeditious progress through that critical section and the resultant speedy release of the spinlock may disproportionately benefit overall system performance and throughput.

This scenario can be hinted to the processor by executing a `hint @priority` instruction. This encourages the processor to devote more processor resources to this thread (at the expense of other threads) for some period of time.

Resource allocation within the processor eventually reverts to a fair allocation, so there's no need for software to hint that it is no longer in a critical section. Processors that support this hint also ensure that it cannot be abused to affect overall longer-term fairness of processor resource allocation.

4.6 Summary

This chapter has presented a wide variety of topics related to optimizing control flow including predication, branch architecture, multiway branches, parallel compares, instruction stream alignment, and branch hints. Although such topics could have been presented in separate chapters, the interplay between the features is best understood by their effects on each other.

Predication and its interplay on scheduling region formation is central to the performance of the Itanium architecture. Unfortunately, discussion of compiler algorithms of this nature are far beyond the scope of this document.

§

Software Pipelining and Loop Support 5

5.1 Overview

The Itanium architecture provides extensive support for software-pipelined loops, including register rotation, special loop branches, and application registers. When combined with predication and support for speculation, these features help to reduce code expansion, path length, and branch mispredictions for loops that can be software pipelined.

The beginning of this chapter reviews basic loop terminology and instructions, and describes the problems that arise when optimizing loops in the absence of architectural support. Specific loop support features of the Itanium architecture are then introduced. The remainder of this chapter describes the programming and optimization of various type of loops.

5.2 Loop Terminology and Basic Loop Support

Loops can be categorized into two types: counted and while. In counted loops, the loop condition is based on the value of a loop counter and the trip count can be computed prior to starting the loop. In while loops, the loop condition is a more general calculation (not a simple count) and the trip count is unknown. Both types are directly supported in the architecture.

The Itanium architecture improves the performance of conventional counted loops by providing a special counted loop branch (the `br.cloop` instruction) and the Loop Count application register (`LC`). The `br.cloop` instruction does not have a branch predicate. Instead, the branching decision is based on the value of the `LC` register. If the `LC` register is greater than zero, it is decremented and the `br.cloop` branch is taken.

5.3 Optimization of Loops

In many loops, there are not enough independent instructions within a single iteration to hide execution latency and make full use of the functional units. For example, in the loop body below, there is very little ILP:

```
L1:
        ld4     r4 = [r5],4;;  // Cycle 0 load postinc 4
        add     r7 = r4,r9;;   // Cycle 2
        st4     [r6] = r7,4    // Cycle 3 store postinc 4
        br.cloopL1;;           // Cycle 3
```

In this code, all the instructions from iteration X are executed before iteration X+1 is started. Assuming that the store from iteration X and the load from iteration X+1 are independent memory references, utilization of the functional units could be improved by moving independent instructions from iteration X+1 to iteration X, effectively overlapping iteration X with iteration X+1.

This section describes two general methods for overlapping loop iterations, both of which result in code expansion on traditional architectures. The code expansion problem is addressed by loop support features in the Itanium architecture that are explored later in this chapter. The loop above will be used as a running example in the next few sections.

5.3.1 Loop Unrolling

Loop unrolling is a technique that seeks to increase the available instruction level parallelism by making and scheduling multiple copies of the loop body together. The registers in each copy of the loop body are given different names to avoid unnecessary WAW and WAR data dependencies. The code below shows the loop from our example on page 1:181 after unrolling twice (total of two copies of the original loop body) and instruction scheduling, assuming two memory ports and a two cycle latency for loads. For simplicity, assume that the loop trip count is a constant N that is a multiple of two, so that no exit branch is required after the first copy of the loop body:

```
L1:
        ld4     r4 = [r5],4;;       // Cycle 0
        ld4     r14 = [r5],4;;      // Cycle 1
        add     r7 = r4,r9;;        // Cycle 2
        add     r17 = r14,r9        // Cycle 3
        st4     [r6] = r7,4;;       // Cycle 3
        st4     [r6] = r17,4        // Cycle 4
        br.cloopL1;;                // Cycle 4
```

The above code does not expose as much ILP as possible. The two loads are serialized because they both use and update r5. Similarly the two stores both use and update r6. A variable which is incremented (or decremented) once each iteration by the same amount is called an induction variable. The single induction variable r5 (and similarly r6) can be expanded into two registers as shown in the code below:

```
        add     r15 = 4,r5
        add     r16 = 4,r6;;
L1:     ld4     r4 = [r5],8         // Cycle 0
        ld4     r14 = [r15],8;;     // Cycle 0
        add     r7 = r4,r9          // Cycle 2
        add     r17 = r14,r9;;      // Cycle 2
        st4     [r6] r7,8           // Cycle 3
        st4     [r16] = r17,8       // Cycle 3
        br.cloopL1;;                // Cycle 3
```

Compared to the original loop on page 1:181, twice as many functional units are utilized and the code size is twice as large. However, no instructions are issued in cycle 1 and the functional units are still under utilized in the remaining cycles. The utilization can be increased by unrolling the loop more times, but at the cost of further code expansion. The loop below is unrolled four times (assuming the trip count is multiple of four):

```
        add     r15 = 4,r5
        add     r25 = 8,r5
        add     r35 = 12,r5
        add     r16 = 4,r6
        add     r26 = 8,r6
        add     r36 = 12,r6;;
L1:     ld4     r4 = [r5],16       // Cycle 0
        ld4     r14 = [r15],16;;   // Cycle 0
        ld4     r24 = [r25],16     // Cycle 1
        ld4     r34 = [r35],16;;   // Cycle 1
        add     r7 = r4,r9         // Cycle 2
        add     r17 = r14,r9;;     // Cycle 2
        st4     [r6] = r7,16       // Cycle 3
        st4     [r16] = r17,16     // Cycle 3
        add     r27 = r24,r9       // Cycle 3
        add     r37 = r34,r9;;     // Cycle 3
        st4     [r26] = r27,16     // Cycle 4
        st4     [r36] = r37,16     // Cycle 4
        br.cloop L1;;              // Cycle 4
```

The two memory ports are now utilized in every cycle except cycle 2. Four iterations are now executed in five cycles verses the two iterations in four cycles for the previous version of the loop.

5.3.2 Software Pipelining

Software pipelining is a technique that seeks to overlap loop iterations in a manner that is analogous to hardware pipelining of a functional unit. Each iteration is partitioned into stages with zero or more instructions in each stage. A conceptual view of a single pipelined iteration of the loop from page 1:181 in which each stage is one cycle long is shown below:

```
stage 1:ld4  r4 = [r5],4
stage 2:---                // empty stage
stage 3:add  r7 = r4,r9
stage 4:st4  [r6] = r7,4
```

The following is a conceptual view of five pipelined iterations:

```
 1   2   3   4   5           Cycle
----------------------------------------------------
ld4                          X
    ld4                     X+1
add     ld4                 X+2
st4 add     ld4             X+3
    st4 add     ld4         X+4
        st4 add             X+5
            st4 add         X+6
                st4         X+7
```

The number of cycles between the start of successive iterations is called the initiation interval (II). In the above example, the II is one. Each stage of a pipelined iteration is II cycles long. Most of the examples in this chapter utilize modulo scheduling, which is a particular form of software pipelining in which the II is a constant and every iteration of the loop has the same schedule. It is likely that software pipelining algorithms other than modulo scheduling could benefit from the loop support features. Therefore the examples in this chapter are discussed in terms of software pipelining rather than modulo scheduling.

Software pipelined loops have three phases: prolog, kernel, and epilog, as shown below:

```
 1   2   3   4   5            Phase
-------------------------------------------------------
ld4
    ld4                        Prolog
add     ld4
-------------------------------------------------------
st4 add     ld4                Kernel
    st4 add     ld4
---------------------------------------------------------
        st4 add
            st4 add            Epilog
                st4
```

During the prolog phase, a new loop iteration is started every II cycles (every cycle for the above example) to fill the pipeline. During the first cycle of the prolog, stage 1 of the first iteration executes. During the second cycle, stage 1 of the second iteration and stage 2 of the first iteration execute, etc. By the start of the kernel phase, the pipeline is full. Stage 1 of the fourth iteration, stage 2 of the third iteration, stage 3 of the second iteration, and stage 4 of the first iteration execute. During the kernel phase, a new loop iteration is started, and another is completed every II cycles. During the epilog phase, no new iterations are started, but the iterations already in progress are completed, draining the pipeline. In the above example, iterations 3-5 are completed during the epilog phase.

The software pipeline is coded as a loop that is very different from the original source code loop. To avoid confusion when discussing loops and loop iterations, we use the term *source loop* and *source iteration* to refer back to the original source code loop, and the term *kernel loop* and *kernel iteration* to refer to the loop that implements the software pipeline.

In the above example, the load from the second source iteration is issued before result of the first load is consumed. Thus, in many cases, loads from successive iterations of the loop must target different registers to avoid overwriting existing live values. In traditional architectures, this requires unrolling of the kernel loop and software renaming of the registers, resulting in code expansion. Furthermore, in traditional architectures, separate blocks of code are generated for the prolog, kernel, and epilog phases, resulting in additional code expansion.

5.4 Loop Support Features in the Intel® Itanium® Architecture

The code expansion that results from loop optimizations (such as software pipelining and loop unrolling) on traditional architectures can increase the number of instruction cache misses, thus reducing overall performance. The loop support features in the

Itanium architecture allow some loops to be software pipelined without code expansion. Register rotation provides a renaming mechanism that reduces the need for loop unrolling and software renaming of registers. Special software pipelined loop branches support register rotation and, combined with predication, reduce the need to generate separate blocks of code for the prolog and epilog phases.

5.4.1 Register Rotation

Register rotation renames registers by adding the register number to the value of a register rename base (rrb) register contained in the CFM. The rrb register is decremented when certain special software pipelined loop branches are executed at the end of each kernel iteration. Decrementing the rrb register makes the value in register X appear to move to register X+1. If X is the highest numbered rotating register, its value wraps to the lowest numbered rotating register.

A fixed-sized area of the predicate and floating-point register files (`p16`-`p63` and `f32`-`f127`), and a programmable-sized area of the general register file are defined to rotate. The size of the rotating area in the general register file is determined by an immediate in the `alloc` instruction and must be either zero or a multiple of 8, up to a maximum of 96 registers. The lowest numbered rotating register in the general register file is `r32`. An rrb register is provided for each of the three rotating register files: `CFM.rrb.gr` for the general registers; `CFM.rrb.fr` for the floating-point registers; `CFM.rrb.pr` for the predicate registers. The software pipelined loop branches decrement all the rrb registers simultaneously.

Below is an example of register rotation. The `swp_branch` pseudo-instruction represents a software pipelined loop branch:

```
L1:     ld4     r35 = [r4],4        // post increment by 4
        st4     [r5] = r37,4        // post increment by 4
        swp_branchL1 ;;
```

The value that the load writes to `r35` is read by the store two kernel iterations (and two rotations) later as `r37`. In the meantime, two more instances of the load are executed. Because of register rotation, those instances write their result to different registers and do not modify the value needed by the store.

The rotation of predicate registers serves two purposes. The first is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of the pipeline. To do this, a programmer assigns a predicate to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called the *stage predicate*. For counted loops, `p16` is architecturally defined to be the predicate for the first stage, `p17` is defined to be the predicate for the second stage, etc. A conceptual view of a pipelined source iteration of the example counted loop on page 1:181 is shown below. Each stage is one cycle long and the stage predicates are shown:

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---              // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

A register rotation takes place at the end of each stage (when the software-pipelined loop branch is executed in the kernel loop). Thus a 1 written to `p16` enables the first stage and then is rotated to `p17` at the end of the first stage to enable the second stage for the same source iteration. Each one written to p16 sequentially enables all the stages for a new source iteration. This behavior is used to enable or disable the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases as described in the next section.

5.4.2 Note on Initializing Rotating Predicates

In this chapter, the instruction mov pr.rot = immed is used to initialize rotating predicates. This instruction ignores the value of CFM.rrb.pr. Thus, the examples in this chapter are written assuming that CFM.rrb.pr is always zero prior to the initialization of predicate registers using mov pr.rot = immed.

5.4.3 Software-pipelined Loop Branches

The special software-pipelined loop branches allow the compiler to generate very compact code for software-pipelined loops by supporting register rotation and by controlling the filling and draining of the software pipeline during the prolog and epilog phases. Generally speaking, each time a software-pipelined loop branch is executed, the following actions take place:

1. A decision is made on whether or not to continue kernel loop execution.
2. p16 is set to a value to control execution of the stages of the software pipeline (p63 is written by the branch, and after rotation this value will be in p16).
3. The registers are rotated (rrb registers are decremented).
4. The Loop Count (LC) and/or the Epilog Count (EC) application registers are selectively decremented.

There are two types of software-pipelined loop branches: counted and while.

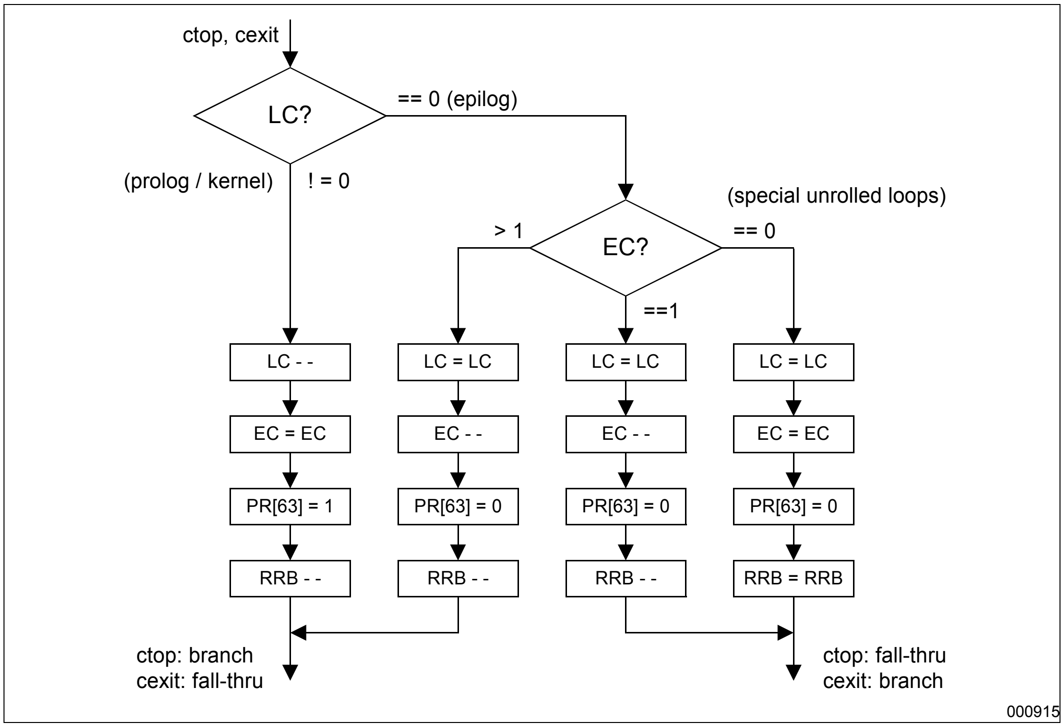
5.4.3.1 Counted Loop Branches

Figure 5-1 shows a flowchart for modulo-scheduled counted loop branches.

During the prolog and kernel phase, a decision to continue kernel loop execution means that a new source iteration is started. Register rotation must occur so that the new source iteration does not overwrite registers that are in use by prior source iterations that are still in the pipeline. p16 is set to 1 to enable the stages of the new source iteration. LC is decremented to update the count of remaining source iterations. EC is not modified.

During the epilog phase, the decision to continue loop execution means that the software pipeline has not yet been fully drained and execution of the source iterations in progress must continue. Register rotation must continue because the remaining source iterations are still writing results and the consumers of the results expect rotation to occur. p16 is now set to 0 because there are no more new source iterations and the instructions that correspond to non-existent source iterations must be disabled. EC contains the count of the remaining execution stages for the last source iteration and is decremented during the epilog. For most loops, when a software pipelined loop branch is executed with EC equal to 1, it indicates that the pipeline has been drained and a decision is made to exit the loop. The special case in which a software-pipelined loop branch is executed with EC equal to 0 can occur in unrolled software-pipelined loops if the target of the cexit branch is set to the next sequential bundle.

Figure 5-1. ctop and cexit Execution Flow



There are two types of software-pipelined loop branches for counted loops. br.ctop is taken when a decision to continue kernel loop execution is made, and is not taken otherwise. It is used when the loop execution decision is located at the bottom of the loop. br.cexit is not taken when a decision to continue kernel loop execution is made, and is taken otherwise. It is used when the loop execution decision is located somewhere other than the bottom of the loop.

5.4.3.2 Counted Loop Example

A conceptual view of a pipelined iteration of the example counted loop on page 1:181 with II equal to one is shown below:

```
stage 1:(p16)  ld4 r4 = [r5],4
stage 2:(p17)  ---             // empty stage
stage 3:(p18)  add r7 = r4,r9
stage 4:(p19)  st4 [r6] = r7,4
```

To generate an efficient pipeline, the compiler must take into account the latencies of instructions and the available functional units. For this example, the load latency is two and the load and add are scheduled two cycles apart. The pipeline below is coded assuming there are two memory ports and the loop count is 200.

Note: Rotating GRs have now been included in the code (the code directly preceding did not). Also, induction variables that are post incremented must be allocated to the static portion of the register file:

```
    mov    lc = 199               // LC =loop count - 1
    mov    ec = 4                 // EC =epilog stages + 1
    mov    pr.rot = 1<<16;;       // PR16 = 1, rest = 0
L1:
    (p16)  ld4     r32  = [r5],4          // Cycle 0
    (p18)  add     r35  = r34,r9          // Cycle 0
    (p19)  st4     [r6] = r36,4           // Cycle 0
           br.ctop L1;;                   // Cycle 0
```

The memory ports are fully utilized. Table 5-1 shows a trace of the execution of this loop.

Table 5-1. ctop Loop Trace

Cycle	Port/Instructions				State before br.ctop					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
…	…	…	…	…	…	…	…	…	…	…
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
…	…	…	…	…	…	…	…	…	…	…
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
…					0	0	0	0	0	0

In cycle 3, the kernel phase is entered and the fourth iteration of the kernel loop executes the `ld4`, `add`, and `st4` from the fourth, second, and first source iterations respectively. By cycle 200, all 200 loads have been executed, and the epilog phase is entered. When the `br.ctop` is executed in cycle 202, `EC` is equal to 1. `EC` is decremented, the registers are rotated one last time, and execution falls out of the kernel loop.

Note: After this final rotation, `EC` and the stage predicates (`p16` - `p19`) are 0.

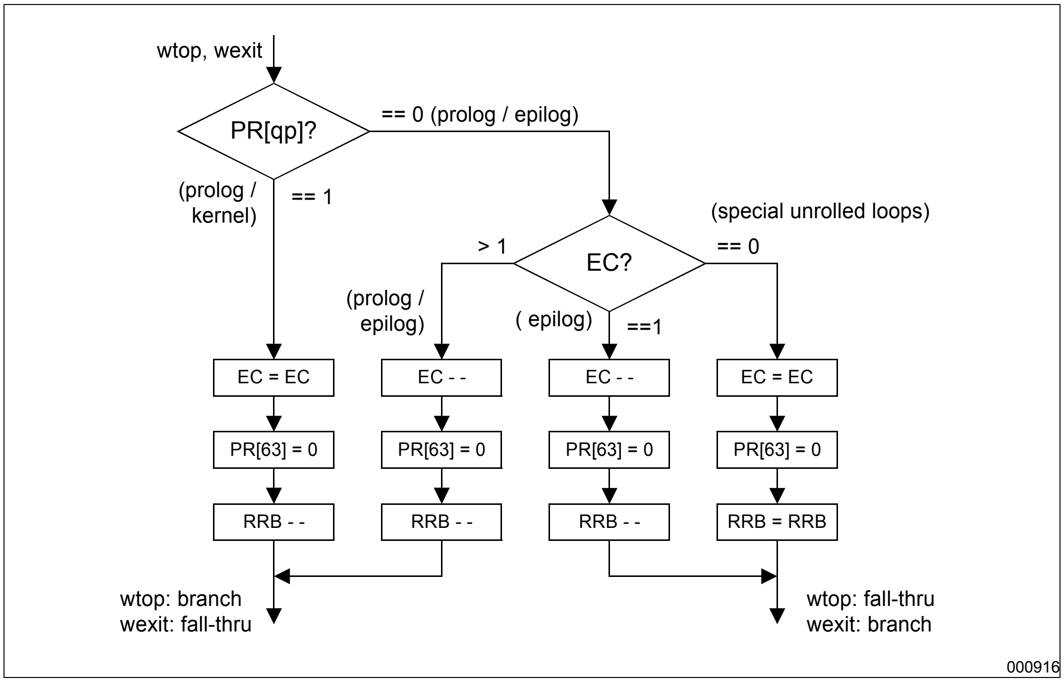
It is desirable to allocate variables that are loop variant to the rotating portion of the register file whenever possible to preserve space in the static portion for loop invariant variables. Induction variables that are post incremented must be allocated to the static portion of the register file.

5.4.3.3 While Loop Branches

Figure 5-2 shows the flowchart for while loop branches.

There are a few differences in the operation of the while loop branch compared to the counted loop branch. The while loop branch does not access `LC` — a branch predicate determines the behavior of this branch instead. During the kernel and epilog phases, the branch predicate is one and zero respectively. During the prolog phase, the branch predicate may be either zero or one depending on the scheme used to program the while loop. Also, `p16` is always set to zero after rotation. The reasons for these differences are related to the nature of while loops and will be explained in more depth with an example in a later section.

Figure 5-2. wtop and wexit Execution Flow



5.4.4 Terminology Review

The terms below were introduced in the preceding sections:

Initiation Interval (II)
The number of cycles between the start of successive source iterations in a software pipelined loop. Each stage of the pipeline is II cycles long.

Prolog The first phase of a software-pipelined loop, in which the pipeline is filled.

Kernel The second phase of a software-pipelined loop, in which the pipeline is full.

Epilog The third phase of a software-pipelined loop, in which the pipeline is drained.

Source Iteration
An iteration of the original source code loop.

Kernel Iteration
An iteration of the loop that implements the software pipeline.

Register Rotation
A form of register renaming that is visible to software. Registers are renamed with respect to a register rename base that is decremented.

Induction Variable

Value that is incremented (or decremented) once per source iteration by the same amount.

5.5 Optimization of Loops in the Intel® Itanium® Architecture

Register rotation, predication, and the software pipelined loop branches allow the generation of compact, yet highly parallel code. Speculation can further increase loop performance by removing dependency barriers that limit the throughput of software pipelined loops. Register rotation removes the requirement that kernel loops be unrolled to allow software renaming of the registers. However in some cases performance can be increased by unrolling the source loop prior to software pipelining, or by generating explicit prolog and/or epilog blocks. The remainder of this chapter discusses loop optimizations.

5.5.1 While Loops

The programming scheme for while loops depends upon the structure of the loop. This section discusses do-while loops, in which the loop condition is computed at the bottom of the loop. Optimizing compilers often transform while loops (where the condition is computed at the top of the loop) into do-while loops by moving the condition computation to the bottom of the loop and placing a copy of the condition computation prior to the loop to reduce the number of branches in the loop. The remainder of this section refers to such loops simply as while loops. Below is a simple while loop:

```
L1:     ld4     r4 = [r5],4;;       // Cycle 0
        st4     [r6] = r4,4         // Cycle 2
        cmp.ne  p1,p0 = r4,r0       // Cycle 2
(p1)    br      L1;;                // Cycle 2
```

A conceptual view of a pipelined iteration of this loop with II equal to one is shown below:

```
stage 1:ld4        r4  = [r5],4
stage 2:---                        // empty stage
stage 3:st4        [r6]= r4,4
        cmp.ne.unc p1,p0 = r4,r0
(p1)    br         L1
```

The following is a conceptual view of four overlapped source iterations assuming the load and store are independent memory references. The store, compare, and branch instructions in stage two are represented by the pseudo-instruction `scb`:

```
 1   2   3   4     Cycle
-------------------------------------------------------
ld4                 X
    ld4.s          X+1
scb     ld4.s      X+2
    scb     ld4.s  X+3
        scb        X+4
            scb    X+5
```

Notice that the load for the second source iteration is executed before the compare and branch of the first source iteration. That is, the load (and the update of r5) is speculative. The loop condition is not computed until cycle X+2, but in order to maximize the use of resources, it is desirable to start the second source iteration at cycle X+1.   Without the support for control speculation in the Itanium architecture, the second source iteration could not be started until cycle X+3.

The computation of the loop condition for while loops is very different from that of counted loops. In counted loops, it is possible to compute the loop condition in one cycle using a counted loop branch. This is what a br.ctop instruction does when it sets p16. In while loops, a compare must compute the loop condition and set the stage predicates. The stages prior to the one containing the compare are called the *speculative stages* of the pipeline, because it is not possible for the compare to completely control the execution of these stages. Therefore, the stage predicate set by the compare is used (after rotation) to control the first non-speculative stage of the pipeline.

The pipelined version of the while loop on page 1:190 is shown below.   A check for the speculative load is included:

```
        mov     ec = 2
        mov     pr.rot = 1 << 16;;      // PR16 = 1, rest = 0
L1:
        ld4.s   r32 = [r5],4            // Cycle 0
(p18)   chk.s   r34, recovery           // Cycle 0
(p18)   cmp.ne  p17,p0 = r34,r0         // Cycle 0
(p18)   st4     [r6] = r34,4            // Cycle 0
(p17)   br.wtop.sptkL1;;                // Cycle 0
L2:
```

To explain why the kernel loop is programmed the way it is, it is helpful to examine a trace of the execution of the loop (assume there are 200 source iterations) shown in Table 5-2.

There is no stage predicate assigned to the load because it is speculative. The compare sets p17. This is the branch predicate for the current iteration and, after rotation, the stage predicate for the first non-speculative stage (stage three) of the next source iteration. During the prolog, the compare cannot produce its first valid result until cycle two. The initialization of the predicates provides a pipeline that disables the compare until the first source iteration reaches stage two in cycle two.   At that point the compare starts generating stage predicates to control the non-speculative stages of the pipeline. Notice that the compare is conditional. If it were unconditional, it would always write a zero to p17 and the pipeline would not get started correctly.

Table 5-2. wtop Loop Trace

Cycle	Port/Instructions					State before br.wtop			
	M	I	I	M	B	p16	p17	p18	EC
0	ld4.s				br.wtop	1	0	0	2
1	ld4.s				br.wtop	0	1	0	1
2	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1

Table 5-2. wtop Loop Trace

Cycle	Port/Instructions					State before br.wtop			
	M	I	I	M	B	p16	p17	p18	EC
3	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
100	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
…	…	…		…	…	…	…	…	…
199	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
200	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
201	ld4.s	cmp	chk	st4	br.wtop	0	0	1	1
						0	0	0	0

The executions of `br.wtop` in the first two cycles of the prolog do not correspond to any of the source iterations. Their purpose is simply to continue the kernel loop until the first valid loop condition can be produced. In cycle one, the branch predicate `p17` is one. For this programming scheme, the branch predicate of the `br.wtop` is always a one during the last speculative stage of the first source iteration. During all the prior stages, the branch predicate is zero. If the branch predicate is zero, the `br.wtop` continues the kernel loop only if `EC` is greater than one. It also decrements `EC`. Thus `EC` must be initialized to (# epilog stages + # speculative pipeline stages). In the above example, this is 0 + 2 = 2.

In cycle 201, the compare for the 200th source iteration is executed. Since this is the final source iteration, the result of the compare is a zero and `p17` is unmodified. The zero that was rotated into `p17` from `p16` causes the `br.wtop` to fall through to the loop exit. `EC` is decremented and the registers are rotated one last time.

In the above example, there are no epilog stages. As soon as the branch predicate becomes zero, the kernel loop is exited.

5.5.2 Loops with Predicated Instructions

Instructions that already have predicates in the source loop are not assigned stage predicates. They continue to be controlled by compare instructions in the loop body. For example, the following loop contains predicated instructions:

```
L1:     ldfs    f4 = [r5],4
        ldfs    f9 = [r8],4;;
        fcmp.ge.unc p1,p2 = f4,f9;;
 (p1)   stfs    [r9] = f4, 4
 (p2)   stfs    [r9] = f9, 4
        br.cloopL1 ;;
```

Below is a possible pipeline with an II of 2, assuming a floating-point load latency of 9 cycles:

```
stage 1:
(p16)   ldfs    f4 = [r5],4
(p16)   ldfs    f9 = [r8],4;;
            ---                // empty cycle
stage 2-4:  ---                // empty stages
stage 5:    ---                // empty cycle
(p20)   fcmp.ge.unc p1,p2 = f4,f9;;
stage 6:    ---                // empty cycle
(p1)    stfs    [r9] = f4, 4
(p2)    stfs    [r9] = f9, 4
```

The following is the code to implement the pipeline:

```
        mov     lc = 199           // LC = loop count - 1
        mov     ec = 6             // EC = epilog stages + 1
        mov     pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)   ldfs    f32 = [r5],4
(p16)   ldfs    f38 = [r8],4;;
(p32)   stfs    [r9] = f37, 4
(p20)   fcmp.ge.uncp31,p32 = f36,f42
(p33)   stfs    [r9] = f43, 4
L2:     br.ctop.sptkL1;;
```

5.5.3 Multiple-exit Loops

All of the example loops discussed so far have a single exit at the bottom of the loop. The loop below contains multiple exits — an exit at the bottom associated with the loop closing branch, and an early exit in the middle:

```
L1:     ld4     r4 = [r5],4;;
        ld4     r9 = [r4];;
        cmp.eq.unc p1,p0 = r9,r7
(p1)    br.cond  exit              // early exit
        add     r8 = -1,r8;;
        cmp.ge.unc p3,p0 = r8,r0
(p3)    br.cond L1;;
```

Loops with multiple exits require special care to ensure that the pipeline is correctly drained when the early exit is taken.There are two ways to generate a pipelined version of the above loop: (1) convert it to a single exit loop, or (2) pipeline it with the multiple exits explicitly present.

5.5.3.1 Converting Multiple Exit Loops to Single Exit Loops

The first is to transform the multiple exit loop into a single exit loop. In the source loop, execution of the add, the second compare and the second branch is guarded by the first branch. The loop can be transformed into a single exit loop by using predicates to guard the execution of these instructions and moving the early exit branch out of the loop as shown below:

```
L1:     ld4    r4 = [r5],4;;
        ld4    r9 = [r4];;
        cmp.eq.uncp1,p2 = r9,r7
        add    r8 = -1,r8;;
(p2)    cmp.ge.unc p3,p0 = r8,r0
(p3)    br.cond L1;;
(p1)    br.cond exit      // early exit if p1 is 1
```

The computation of `p3` determines if either exit of the source loop would have been taken. If `p3` is zero, the loop is exited and `p1` is used to determine which exit was actually taken. The add is executed speculatively (it is not guarded by `p2`) to keep the dependency from the `cmp.eq` to the `add` from limiting the II. It is assumed that either `r8` is not live out at the early exit or that compensation code is added at the target of the early exit. The pipeline for this loop is shown below with the stage predicate assignments but no other rotating register allocation. The compare and the branch at the end of stage 4 are not assigned stage predicates because they already have qualifying predicates in the source loop:

```
stage 1:ld4.s  r4 = [r5],4;;     // II = 2
        ---                   // empty cycle
stage 2:---                   // empty cycle
        ld4.s  r9 = [r4];;
stage 3:---                   // empty stage
stage 4:
(p19)   add    r8 = -1,r8
(p19)   cmp.eq.uncp1,p2 = r9,r7;;
(p2)    cmp.ge.uncp3,p0 = r8,r0
(p3)    br.cond L1;;
```

The code to implement this pipeline is shown below complete with the `chk` instruction:

```
        mov    ec = 3
        mov    pr.rot = 1 << 16;;    // PR16 = 1, rest = 0
L1:     ld4.s  r32 = [r5],4          // Cycle 0
(p19)   chk.s  r36, recovery         // Cycle 0
(p19)   add    r8  = -1,r8           // Cycle 0
(p19)   cmp.eq.unc p31,p32 = r36,r7;; // Cycle 0
        ld4.s  r34 = [r33]           // Cycle 1
(p32)   cmp.ge  p18,p0 = r8,r0       // Cycle 1
L2:
(p18)   br.wtop.sptk L1;;            // Cycle 1
(p32)   br.cond exit                 // early exit if p32 is 1
```

Note: When the loop is exited, one final rotation occurs, rotating the value in `p31` to `p32`. Thus, `p32` is used as the branch predicate for the early exit branch.

5.5.3.2 Pipelining with Explicit Multiple Exits

The second approach is to combine the last three instructions in the loop into a br.cloop instruction and then pipeline the loop. The pipeline using this approach is shown below:

```
stage 1:    ld4.s r4 = [r5],4;;        // II = 1
stage 4:    ld4.s r9 = [r4];;
stage 6:    cmp.eq.unc p1,p0 = r9,r7
(p1)        br.cond  exit
            br.cloop L1;;
```

There are five speculative stages in this pipeline because a non-speculative decision to initiate another loop iteration cannot be made until the br.cond and br.cloop are executed in stage 6. The code to implement this pipeline is shown below assuming a trip count of 200:

```
        mov     lc = 204
        mov     ec = 1
        mov     pr.rot = 1 << 16;;     // PR16 = 1, rest = 0
L1:
        ld4.s   r32 = [r5],4           // Cycle 0
(p21)   chk.s   r38, recovery          // Cycle 0
(p21)   cmp.eq.uncp1,p0 = r38,r7       // Cycle 0
        ld4.s   r36 = [r35]            // Cycle 0
(p1)    br.cond exit                   // Cycle 0
L2:     br.ctop.sptkL1;                // Cycle 0
```

When the kernel loop is exited at either the br.cond or the br.ctop, the last source iteration is complete. Thus, EC is initialized to 1 and there is no explicit epilog block generated for the early exit. The LC register is initialized to five more than 199 because there are five speculative stages. The purpose of the first five executions of br.ctop is simply to keep the loop going until the first valid branch predicate is generated for the br.cond. During each of these executions, LC is decremented, so five must be added to the LC initialization amount to compensate.

A smaller II is achieved with the second approach. This pipelined code will also work if LC is initialized to 199 and EC is initialized to 6. However, if the early exit is taken, LC will have been decremented too many times and will need to be adjusted if it is used at the target of the early exit. If there is any epilog when the early exit is taken, that epilog must be explicit.

5.5.4 Software Pipelining Considerations

There may be instances where it may not be desirable to pipeline a loop. Software pipelining increases the throughput of iterations, but may increase the time required to complete a single iteration. As a result, loops with very small trip counts may experience decreased performance when pipelined. For example, consider the following loop:

```
L1:         ld4         r4 = [r5],4             // Cycle 0
            ld4         r7 = [r8],4;;           // Cycle 0
            st4         [r6] = r4,4             // Cycle 2
            st4         [r9] = r7,4             // Cycle 2
            br.cloop    L1;;                    // Cycle 2
```

The following is a possible pipeline with an II of 2:

```
stage 1:    ld4         r4 = [r5],4             // Cycle 0
            ld4         r7 = [r8],4;;           // Cycle 0
            ---                                 // empty cycle
stage 2:    ---                                 // empty cycle
            st4         [r6] = r4,4             // Cycle 3
            st4         [r9] = r7,4;;           // Cycle 3
```

In the source loop, one iteration is completed every three cycles. In the software pipelined loop, it takes four cycles to complete the first iteration. Thereafter, iterations are completed every two cycles. If the trip count is two, the execution time of both versions of the loop is the same, six cycles. If the average trip count of the loop is less than two, the software pipelined version of the loop is slower than the source loop.

In addition, it may not be desirable to pipeline a floating-point loop that contains a function call. The number of floating-point registers used by the loop is not known until after the loop is pipelined. After pipelining, it may be difficult to find empty slots for the instructions needed to save and restore the caller-saved floating-point registers across the function call.

5.5.5 Software Pipelining and Advanced Loads

Advanced loads allow some code that is likely to be invariant to be removed from loops, thus reducing the resource requirements of the loop. Use of advanced loads also can reduce the critical path through the iterations, allowing a smaller II to be achieved. See Chapter 3, “Memory Reference” for more information on advanced loads. However, caution must be exercised when using advanced loads with register rotation. For this discussion, we assume an ALAT with 32 entries.

5.5.5.1 Capacity Limitations

An advanced load with a destination that is a rotating register targets a different physical register and allocates a new ALAT entry for each kernel iteration.   For example, the simple loop below replaces 32 ALAT entries in 32 iterations:

```
L1:
(p16)   ld4.a   r32 = [r8]
(p47)   ld4.c   r63 = [r8]
        br.ctop L1;;
```

To avoid unnecessary ALAT misses, the check load or advanced load check must be executed before a later advanced load causes a replacement of the entry being checked. In the simple loop above, the unnecessary ALAT misses do not occur because the check load is done within 31 iterations of the advanced load. In the example below, an ALAT miss is encountered for every check load because the advanced load replaces an entry just before the corresponding check load is executed:

```
L1:
(p16)   ld4.a   r32 = [r8]
(p48)   ld4.c   r64 = [r8]
        br.ctop L1;;
```

5.5.5.2 Conflicts in the ALAT

Using an advanced load to remove a likely invariant load from a loop while advancing another load inside the loop results in poor performance if the latter load targets a rotating register. The advanced load that targets the rotating register will eventually invalidate the ALAT entry for the loop invariant load. Thereafter, every execution of the check load for the loop invariant load will cause an ALAT miss.

When more than one advanced load in the loop targets a rotating register, the registers must be assigned and the register lifetimes controlled so that the check load for a particular advanced load X is executed before any of the other advanced loads can invalidate the entry allocated by load X. For example, the following loop successfully targets rotating registers with two advanced loads without any ALAT misses because the two advanced load – check load pairs never create more than 32 simultaneously live ALAT entries:

```
L1:
(p16)   ld4.a   r32 = [r8]
(p31)   ld4.c   r47 = [r8]
(p16)   ld4.a   r48 = [r9]
(p31)   ld4.c   r63 = [r9]
        br.ctop L1;;
```

When the code cannot be arranged to avoid ALAT misses, it may be best to assign static registers to the destinations of the advanced loads and unroll the loop to explicitly rename the destinations of the advanced loads where necessary. The following example shows how to unroll the loop to avoid the use of rotating registers. The loop has an II equal to 1 and the check load is executed one cycle (and one rotation) after the advanced load:

```
L1:
(p16)   ld4.a   r33 = [r8]
(p17)   ld4.c   r34 = [r8]
        br.ctop L1;;
```

Static registers can be assigned to the destinations of the loads if the loop is unrolled twice:

```
L1:
(p16)   ld4.a   r3 = [r8]
(p17)   ld4.c   r4 = [r8]
        br.cexit L2;;
(p16)   ld4.a   r4 = [r8]
(p17)   ld4.c   r3 = [r8]
        br.ctop L1;;
L2:     //
```

Rotating registers could still be used for the values that are not generated by advanced loads. The effect of this unrolling on instruction cache performance must be considered as part of the cost of advancing a load.

5.5.6 Loop Unrolling Prior to Software Pipelining

In some cases, higher performance can be achieved by unrolling the loop prior to software pipelining. Loops that are resource constrained can be improved by unrolling such that the limiting resource is more fully utilized. In the following example if we assume the target processor has only two memory units, the loop performance is bound by the number of memory units:

```
L1:     ld4         r4 = [r5],4             // Cycle 0
        ld4         r9 = [r8],4;;           // Cycle 0
        add         r7 = r4,r9;;            // Cycle 2
        st4         [r6] = r7,4             // Cycle 3
        br.cloop    L1;;                    // Cycle 3
```

A pipelined version of this loop must have an II of at least two because there are three memory instructions, but only two memory units. If the loop is unrolled twice prior to software pipelining and assuming the store is independent of the loads, an II of 3 can be achieved for the new loop. This is an effective II of 1.5 for the original source loop. Below is a possible pipeline for the unrolled loop:

```
stage 1:
(p16)      ld4     r4 = [r5],8        // odd iteration
(p16)      ld4     r9 = [r8],8;;      // odd iteration
stage 2:
(p16)      ld4     r14 = [r15],8      // even iteration
(p16)      ld4     r19 = [r18],8;;    // even iteration
           // ---     empty cycle
stage 3:(p18) add  r7 = r4,r9         // odd iteration
(p17)      add     r17 = r14,r19;;    // even iteration
stage 4:   // ---     empty cycle
(p19)      st4     [r6]  = r7,8       // odd iteration
(p18)      st4     [r16] = r17,8;;    // even iteration
```

The unrolled loop contains two copies of the source loop body, one that corresponds to the odd source iterations and one that corresponds to the even source iterations. The assignment of stage predicates must take this into account. Recall that each one written to `p16` sequentially enables all the stages for a new source iteration. During stage one of the above pipeline, the stage predicate for the odd iteration is in `p16`. The stage predicate for the even iteration does not exist yet. During stage two of the above pipeline, the stage predicate for the odd iteration is in `p17` and the new stage predicate for the even iteration is in `p16`. Thus within the same pipeline stage, if the stage predicate for the odd iteration is in predicate register X, the stage predicate for the even iteration is in predicate register X-1. The pseudo-code to implement this pipeline assuming an unknown trip count is shown below:

```
        add     r15 = r5,4
        add     r18 = r8,4
        mov     lc = r2            // LC = loop count - 1
        mov     ec = 4             // EC = epilog stages + 1
        mov     pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)   ld4     r33 = [r5],8       // Cycle 0 odd iteration
(p18)   add     r39 = r35,r38      // Cycle 0 odd iteration
(p17)   add     r38 = r34,r37      // Cycle 0 even iteration
(p16)   ld4     r36 = [r8],8       // Cycle 0 odd iteration
        br.cexit.spnt L3;;         // Cycle 0
(p16)   ld4     r33 = [r15],8      // Cycle 1 even iteration
(p16)   ld4     r36 = [r18],8;;    // Cycle 1 even iteration
(p19)   st4     [r6]  = r40,8      // Cycle 2 odd iteration
(p18)   st4     [r16] = r39,8      // Cycle 2 even iteration
L2:     br.ctop.sptk L1;;          // Cycle 2
L3:
```

Notice that the stages are not equal in length. Stages 1 and 3 are one cycle each, and stages 2 and 4 are two cycles each. Also, the length of the epilog phase varies with the trip count.   If the trip count is odd, the number of epilog stages is three, starting after the br.cexit and ending at the `br.ctop`. If the trip count is even, the number of epilog stages is two, starting after the `br.ctop` and ending at the `br.ctop`. The `EC` must be set to account for the maximum number of epilog stages. Thus for this example, `EC` is initialized to four. When the trip count is even, one extra epilog stage is executed and `br.exit L3` is taken. All of the stage predicates used during the extra epilog stages are equal to 0, so nothing is executed.

The extra epilog stage for even trip counts can be eliminated by setting the target of the `br.cexit` branch to the next sequential bundle and initializing `EC` to three as shown below:

```
        add     r15 = r5,4
        add     r18 = r8,4
        mov     lc = r2            // LC = loop count - 1
        mov     ec = 3             // EC = epilog stages + 1
        mov     pr.rot=1<<16;;     // PR16 = 1, rest = 0
L1:
(p16)   ld4     r33 = [r5],8       // Cycle 0 odd iteration
(p18)   add     r39 = r35,r38      // Cycle 0 odd iteration
(p17)   add     r38 = r34,r37      // Cycle 0 even iteration
(p16)   ld4     r36 = [r8],8       // Cycle 0 odd iteration
        br.cexit.spnt L4;;         // Cycle 0
L4:
(p16)   ld4     r33 = [r15],8      // Cycle 1 even iteration
(p16)   ld4     r36 = [r18],8;;    // Cycle 1 even iteration
(p19)   st4     [r6]  = r40,8      // Cycle 2 odd iteration
(p18)   st4     [r16] = r39,8      // Cycle 2 even iteration
L2:     br.ctop.sptk L1;;          // Cycle 2
L3:
```

If the loop trip count is even, two epilog stages are executed and the kernel loop is exited at the br.ctop. If the trip count is odd, the first two epilog stages are executed and then the br.cexit branch is taken. Because the target of the br.cexit branch is the next sequential bundle (L4), a third epilog stage is executed before the kernel loop is exited at the br.ctop. This optimization saves one stage at the end of the loop when the trip count is even, and is beneficial for short trip count loops.

Although unrolling can be beneficial, there are a few considerations before trying to unroll and software pipeline. Unrolling reduces the trip count of the loop that is given to the pipeliner, and thus may make pipelining of the loop undesirable since low trip count loops sometimes run faster unpipelined. Unrolling also increases the code size, which may adversely affect instruction cache performance. Unrolling is most beneficial for small loops because the potential performance degradation due to under utilized resources is greater and the effect of unrolling on the instruction cache performance is smaller compared to large loops.

5.5.7 Implementing Reductions

In the following example, a sum of products is accumulated in register f7:

```
        mov         f7 = 0;;            // initialize sum
L1:     ldfs        f4 = [r5],4
        ldfs        f9 = [r8],4;;
        fma         f7 = f4,f9,f7;;     // accumulate
        br.cloop    L1 ;;
```

The performance is bound by the latency of the fma instruction which we assume is 5 cycles for these examples. A pipelined version of this loop must have an II of at least five because the fma latency is five.   By making use of register rotation, the loop can be transformed into the one below.

Note that the loop has not yet been pipelined. The register rotation and special loop branches are being used to enable an optimization prior to software pipelining.

```
        mov     lc = 199            // LC = loop count - 1
        mov     ec = 1              // Not pipelined, so no epilog
        mov     f33 = 0             // initialize 5 sums
        mov     f34 = 0
        mov     f35 = 0
        mov     f36 = 0
        mov     f37 = 0;;
L1:     ldfs    f4 = [r5],4
        ldfs    f9 = [r8],4;;
        fma     f32 = f4,f9,f37;;   // accumulate
        br.ctop L1 ;;

        fadd    f10 = f33,f34       // add sums
        fadd    f11 = f35,f36;;
        fadd    f12 = f10,f11;;
        fadd    f7 = f12,f37
```

This loop maintains five independent sums in registers f33-f37. The fma instruction in iteration X produces a result that is used by the fma instruction in iteration X+5. Iterations X through X+4 are independent, allowing an II of one to be achieved. The code for a pipelined version of the loop assuming two memory ports and a nine cycle latency for a floating-point load is shown below:

```
       mov            lc = 199           // LC = loop count - 1
       mov            ec = 10            // EC = epilog stages + 1
       mov            pr.rot=1<<16       // PR16 = 1, rest = 0
       mov            f33 = 0            // initialize sums
       mov            f34 = 0
       mov            f35 = 0
       mov            f36 = 0
       mov            f37 = 0
L1:
(p16)  ldfs           f50 = [r5],4       // Cycle 0
(p16)  ldfs           f60 = [r8],4       // Cycle 0
(p25)  fma            f41 = f59,f69,f46  // Cycle 0
       br.ctop.sptk   L1;;               // Cycle 0
       fadd           f10 = f42,f43      // add sums
       fadd           f11 = f44,f45 ;;
       fadd           f12 = f10,f11 ;;
       fadd           f7 = f12,f46
```

5.5.8 Explicit Prolog and Epilog

In some cases, an explicit prolog is necessary for code correctness. This can occur in cases where a speculative instruction generates a value that is live across source iterations. Consider the following loop:

```
       ld4     r3 = [r5] ;;
L1:
       ld4     r6 = [r8],4        // Cycle 0
       ld4     r5 = [r9],4 ;;     // Cycle 0
       add     r7 = r3,r6 ;;      // Cycle 2
       ld4     r3 = [r5]          // Cycle 3
       and     r10 = 3,r7;;       // Cycle 3
       cmp.ne  p1,p0=r10,r11      // Cycle 4
(p1)   br.cond L1 ;;              // Cycle 4
```

The following is a possible pipeline for the loop:

```
stage 1:          ld4.s   r6 = [r8],4    // II = 2
                  ld4.s   r5 = [r9],4 ;;
                  ---                    // empty cycle
stage 2:          ---                    // empty cycle
                  ld4.s   r36 = [r5]
                  add     r7 = r37,r6 ;;
stage 3:   (p18)  and     r10 = 3,r7 ;;
           (p18)  cmp.ne  p1,p0 = r10,r11
           (p1)   br.wtop L1 ;;
```

Note that, in the code above, the `ld4` and the `add` instructions in stage 2 have been reordered. Register rotation has been used to eliminate the WAR register dependency from the `add` to the `ld4`. The first two stages are speculative. The code to implement the pipeline is shown below:

```
        ld4        r36 = [r5]
        mov        ec = 2
        mov        pr.rot = 1 << 16 ;;    // PR16 = 1, rest = 0
L1:     ld4.s      r32 = [r8],4           // Cycle 0
        ld4.s      r34 = [r9],4           // Cycle 0
(p18)   and        r40 = 3,r39 ;;         // Cycle 0
        ld4.s      r36 = [r35]            // Cycle 1
        add        r38 = r37,r33          // Cycle 1
(p18)   chk.s      r40, recovery          // Cycle 1
(p18)   cmp.ne     p17,p0 = r40,r11       // Cycle 1
(p17)   br.wtop    L1 ;;                  // Cycle 1
```

The problem with this pipelined loop is that the value written to `r36` prior to the loop is overwritten before it is used by the `add`. The value is overwritten by the load into `r36` in the first kernel iteration. This load is in the second stage of the pipeline, but cannot be controlled during the first kernel iteration because it is speculative and does not have a stage predicate. This problem can be solved by peeling off one iteration of the kernel and excluding from that copy any instructions that are not in the first stage of the pipeline as shown below.

Note that the destination register numbers for the instructions in the explicit prolog have been increased by one. This is to account for the fact that there is no rotation at the end of the peeled kernel iteration.

```
        ld4        r37 = [r5]
        mov        ec = 1
        mov        pr.rot = 1<<17;;       // PR17 = 1, rest = 0
        ld4        r33 = [r8],4
        ld4        r35 = [r9],4
L1:     ld4.s      r32 = [r8],4           // Cycle 0
        ld4.s      r34 = [r9],4           // Cycle 0
(p18)   and        r40 = 3,r39;;          // Cycle 0
        ld4.s      r36 = [r35]            // Cycle 1
        add        r38 = r37,r33          // Cycle 1
(p18)   chk.s      r40, recovery          // Cycle 1
(p18)   cmp.ne     p17,p0 = r40,r11       // Cycle 1
(p17)   br.wtop    L1 ;;                  // Cycle 1
```

In some cases, higher performance can be achieved by generating separate blocks of code for all or part of the prolog and/or epilog phase. It is clear from the execution trace of the pipelined counted loop from page 1:188 that the functional units are under-utilized during the prolog and epilog phases. Part of the prolog and epilog could be peeled off and merged with the code preceding and following the loop. The following is a pipelined version of that counted loop with an explicit prolog and epilog:

```
        mov     lc = 196
        mov     ec = 1
prolog:
        ld4     r35 = [r5],4;;     // Cycle 0
        ld4     r34 = [r5],4 ;;    // Cycle 1
        ld4     r33 = [r5],4       // Cycle 2
        add     r36 = r35,r9 ;;    // Cycle 2
L1:
        ld4     r32 = [r5],4
        add     r35 = r34,r9
        st4     [r6] = r36,4
L2:     br.ctop L1 ;;
epilog:
        add     r35  = r34,r9      // Cycle 0
        st4     [r6] = r36,4 ;;    // Cycle 0
        add     r34 = r33,r9       // Cycle 1
        st4     [r6] = r35,4 ;;    // Cycle 1
        st4     [r6] = r34,4       // Cycle 2
```

The entire prolog (first three iterations of the kernel loop) and epilog (last three iterations) have been peeled off. No attempt has been made to reschedule the peeled instructions. The stage predicates have been removed from the instructions since they are not required for controlling the prolog and epilog phases. Removing them from the prolog makes the prolog instructions independent of the rotating predicates and eliminates the need for software-pipelined loop branches between prolog stages. Thus the entire prolog is independent of the initialization of LC and EC that precede it. The register numbers in the prolog and epilog have been adjusted to account for the lack of rotation between stages during those phases.

Note: This code assumes that the trip count of the source loop is at least four. If the minimum trip count is unknown at compile time, then a runtime check of the trip count must be added before the prolog. If the trip count is less than four, then control branches to a copy of the original loop.

If this pipelined loop is nested inside an outer loop, there exists a further optimization opportunity. The outer loop could be rotated such that the kernel loop is at the top followed by the epilog for the current outer loop iteration and the prolog for the next outer loop iteration. A copy of the prolog would also be added prior to the outer loop.

Note: From the earlier trace of the counted loop execution, the functional unit usage of the prolog and epilog are complimentary such that they could be very nicely overlapped.

The drawback of creating an explicit prolog or epilog is code expansion.

5.5.9 Redundant Load Elimination in Loops

Unrolling of a loop is sometimes necessary to remove copy operations created by loop optimizations. The following is an example of redundant load elimination. In the code below, each iteration loads two values, one of which has already been loaded by the previous source iteration:

```
        add         r8 = r5,4;;
L1:     ld4         r4 = [r5],4         // a[i]
        ld4         r9 = [r8],4 ;;      // a[i+1]
        add         r7 = r4,r9 ;;
        st4         [r6] = r7,4
        br.cloop    L1 ;;
```

The redundant load can be eliminated by adding a copy of the first load prior to the loop and changing the load to a copy (`mov`):

```
        add         r8 = r5,4
        ld4         r9 = [r5],4;;       // a[i]
L1:     mov         r4 = r9             // a[i] = previous a[i+1]
        ld4         r9 = [r8],4 ;;      // a[i+1]
        add         r7 = r4,r9 ;;
        st4         [r6] = r7,4
        br.cloop    L1 ;;
```

In traditional architectures, the `mov` instruction can only be removed by unrolling the loop twice. One instruction is removed from the loop at the cost of two times code expansion. The register rotation feature in the Itanium architecture can be used to eliminate the `mov` instruction without unrolling the loop:

```
        add     r8 = r5,4
        ld4     r33 = [r5],4;;      // a[i]
L1:     ld4     r32 = [r8],4 ;;     // a[i+1]
        add     r7 = r33,r32 ;;
        st4     [r6] = r7,4
        br.ctop L1 ;;
```

5.6 Summary

The examples in this chapter show how features in the Itanium architecture can be used to optimize loops without the code expansion required with traditional architectures. Register rotation, predication, and the software-pipelined loop branches all contribute to this capability. Control speculation increases the overlap of the iterations of while loops. Data speculation increases the overlap of iterations of loops that have loads and stores that cannot be disambiguated.

§

Floating-point Applications 6

6.1 Overview

The Itanium floating-point architecture is fully ANSI/IEEE-754 standard compliant and provides performance enhancing features such as the fused multiply accumulate instruction, the large floating-point register file (with static and rotating sections), the extended range register file data representation, the multiple independent floating-point status fields, and the high bandwidth memory access instructions that enable the creation of compact, high performance, floating-point application code.

The beginning of this chapter reviews some specific performance limitations that are common in floating-point intensive application codes. Later, architectural features that address these limitations are presented with illustrative code examples. The remainder of this chapter highlights the optimization of some commonly used kernels using these features.

6.2 FP Application Performance Limiters

Floating-point applications are characterized by a predominance of loops. Some loops compute complex calculations on regularly structured data, others simply copy data from one place to another, while others perform gather/scatter-type operations that simultaneously compute and rearrange data. The following sections describe code characteristics that limit performance and how they affect these different kinds of loops.

6.2.1 Execution Latency

Loops often contain recurrence relationships. Consider the tri-diagonal elimination kernel from the Livermore Fortran Kernel suite.

```
DO 5 i = 2, N
   5X[i] = Z[i] * (Y[i] - X[i-1])
```

The dependency between X[i] and X[i-1] limits the iteration time of the loop to be the sum of the latency of the subtract and the multiply. The available parallelism can be increased by unrolling the loop and can be exploited by replicating computation, however the fundamental limitation of the data dependency remains.

Sometimes, even if the loop is vectorizable and can be software pipelined, the iteration time of the loop is limited by the execution latency of the hardware that executes the code. A simple vector divide (shown below) is a typical example:

```
DO 1 I = 1, N
   1X[i] = Y[i] / Z[i]
```

Since typical modern microprocessors contain a non-pipelined floating-point unit, the iteration time of the loop is the latency of the divide which can be tens of clocks.

6.2.2 Execution Bandwidth

When sufficient ILP exists and can be exploited, the performance limitation is the availability of the execution resources – or the execution bandwidth of the machine. Consider the dense matrix multiply kernel from the BLAS3 library.

```
DO 1 i = 1, N
    DO 1 j = 1, P
        DO 1 k = 1, M
1               C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

Common techniques of loop interchange, loop unrolling, and unroll-and-jam, can be used to increase the available ILP in the inner loop. When this is done, the inner loop contains an abundance of independent floating-point computations with a relatively small number of memory operations. The performance constraint is then largely the floating-point execution bandwidth of the machine (assuming sufficient registers are available to hold the accumulators – `C[i,j]` and the intermediate computations).

6.2.3 Memory Latency

While cycle time disparity between the processor and memory creates a general memory latency problem for most codes, there are a few special conditions in floating-point codes that exacerbate its impact.

One such condition is the use of indirect addressing. Gather/scatter codes in general and sparse matrix vector multiply code (below) in particular are good examples.

```
DO 1 ROW = 1, N
    R[ROW] = 0.0d0
    DO 1 I = ROWEND(ROW-1)+1, ROWEND(ROW)
1       R[ROW] = R[ROW] + A[I] * X[COL[I]]
```

The memory latency of the access of `COL[I]` is exposed, since it is used to index into the vector `X`. The access of the element of `X`, the computation of the product, and the summation of the product on `R[ROW]` are all dependent on the memory latency of the access of `COL[I]`.

Another common condition in floating-point codes where memory latency impact is exacerbated is the presence of ambiguous memory dependencies. Consider the incomplete Cholesky conjugate gradient excerpt kernel, again from the Livermore Fortran Kernel suite.

```
    II      = n
    IPNTP   = 0
222 IPNT = IPNTP
    IPNTP   = IPNTP + II
    II      = II/2
    I       = IPNTP + 1
cdir$ ivdep
    DO 2 K = IPNT+2, IPNTP, 2
        I       = I+1
        2      X[I] = X[K] - V[K] * X[K-1] - V[K-1] * X[K+1]
    IF (II .GT. 1) GO TO 222
```

The DO-loop involves an update of X at the index I using X at the indices K, K+1, K-1. Since it is difficult for the compiler to establish whether these indices overlap, the loads of X[K], X[K+1] or X[K-1] for the next iteration cannot be scheduled until the store of X[I] of the current iteration. This exposes the memory latency of access of these operands.

6.2.4 Memory Bandwidth

Floating-point loops are often limited by the rate at which the machine can deliver the operands of the computation. The DAXPY kernel from the BLAS1 library is a typical example:

```
DO 1 I = 1, N
1      Y[I] = Y[I] + A * X[I]
```

The computation requires loading two operands (X[I] and Y[I]) and storing one result (Y[I]) for each floating-point multiply and add operation. If the data arrays (X and Y) are not in cache, then the performance of this loop on most modern microprocessors would be limited by the available memory bandwidth on the machine.

6.3 Floating-point Features in the Intel® Itanium® Architecture

This section highlights architectural features that reduce the impact of the performance limiters described in Section 6.2 using illustrative examples.

6.3.1 Large and Wide Floating-point Register Set

As machine cycle times are reduced, the latency in cycles of the execution units generally increases. As latency increases, register pressure due to multiple operations in-flight also increases. Furthermore as multiple execution units are added, the register pressure increases similarly since even more instructions can be in-flight at any one time.

The Itanium architecture provides 128 directly addressable floating-point registers to enable data reuse and to reduce the number of load/store operations required due to an insufficient number of registers. This reduction in the number of loads and stores can increase performance by changing a computation from being memory operation (MOP) limited to being floating-point operation (FLOP) limited. Consider the dense matrix multiply code below:

```
DO 1 i = 1, N
    DO 1 j = 1, P
        DO 1 k = 1, M
1           C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

In the inner loop (k), two loads are required for every multiply and add operation. The MOP:FLOP ratio is therefore 1:1.

```
L1: ldfd       f5  = [r5], 8      // Load A[i,k]
    ldfd       f6  = [r6], 8      // Load B[k,j]
    fma.d.s0   f7  = f5, f6, f7   // *,+ to C[i,j]
    br.cloop   L1
```

Here, three registers are required to hold the operands (f5, f6) and the accumulator (f7). By recognizing the reuse of A[i,k] for different B[k,j] as j is varied, and the reuse of B[k,j] for different A[i,k] as i is varied, the computation can be restructured as:

```
DO 1 i = 1, N, 2
    DO 1 j = 1, P, 2
        DO 1 k = 1, M
            C[i  ,j  ] = C[i  ,j  ]
                        + A[i  ,k]*B[k,j  ]
            C[i+1,j  ] = C[i+1,j  ]
                        + A[i+1,k]*B[k,j  ]
            C[i  ,j+1] = C[i  ,j+1]
                        + A[i  ,k]*B[k,j+1]
1           C[i+1,j+1] = C[i+1,j+1]
                        + A[i+1,k]*B[k,j+1]
```

Now, for every 4 loads, 4 multiplies and adds can be performed, thus changing the MOP:FLOP ratio to 1:2. However, 8 registers are now required: 4 for the accumulators and 4 for the operands.

```
    add        r6 = r5, 8
    add        r8 = r7, 8
L1: ldfd       f5 = [r5], 16          // Load A[i,k]
    ldfd       f6 = [r6], 16          // Load A[i+1,k]
    ldfd       f7 = [r7], 16          // Load B[k,j]
    ldfd       f8 = [r8], 16          // Load B[k,j+1]
    fma.s0     f9 = f5, f7, f9        // *,+ on C[i,j]
    fma.s0     f10 = f6, f7, f10      // *,+ on C[i+1,j]
    fma.s0     f11 = f5, f8, f11      // *,+ on C[i,j+1]
    fma.s0     f12 = f6, f8, f12      // *,+ on C[i+1,j+1]
    br.cloop   L1
```

With 128 available registers, the outer loops of i and j could be unrolled by 8 each so that 64 multiplies and adds can be performed by loading just 16 operands.

The floating-point register file is divided into two regions: a static region (f0-f31) and a rotating region (f32-f127). The register rotation provides the automatic register renaming required to create compact kernel-only software-pipelined code. Register rotation also enables scheduling software pipelined code with an initiation interval that is less than the longest latency operation. For e.g. consider the simple vector add loop shown below:

```
DO 1 i = 1, N
1       A[i] = B[i] + C[i]
```

The basic inner loop is:

```
L1: ldf        f5 = [r5], 8       // Load B[i]
    ldf        f6 = [r6], 8       // Load C[i]
    fadd       f7 = f5, f6        // Add operands
    stf        [r7] = f7, 8       // Store A[i]
    br.cloop   L1
```

If we suppose the minimum floating-point load latency is 9 clocks, and 2 memory operations can be issued per clock, the above loop has to be unrolled by at least six if there is no register rotation.

```
       add     r8 = r7, 8
 L1:
 (p18)  stf     [r7] = f25, 16     // Cycle 17,26...
 (p18)  stf     [r8] = f26, 16     // Cycle 17,26...
 (p17)  fadd    f25 = f5, f15      // Cycle 8,17,26...
 (p16)  ldf     f5 = [r5], 8       // Cycle 0,9,18...
 (p16)  ldf     f15 = [r6], 8      // Cycle 0,9,18...
 (p17)  fadd    f26 = f6, f16;;    // Cycle 9,18,27 ...
 (p16)  ldf     f6 = [r5], 8       // Cycle 1,10,19 ...
 (p16)  ldf     f16 = [r6], 8      // Cycle 1,10,19 ...
 (p18)  stf     [r7] = f27, 16     // Cycle 20,29 ...
 (p18)  stf     [r8] = f28, 16     // Cycle 20,29 ...
 (p17)  fadd    f27 = f7, f17 ;;   // Cycle 11,20 ...
 (p16)  ldf     f7 = [r5], 8       // Cycle 3,12,21 ...
 (p16)  ldf     f17 = [r6], 8      // Cycle 3,12,21 ...
 (p17)  fadd    f28 = f8, f18 ;;   // Cycle 12,21 ...
 (p16)  ldf     f8 = [r5], 8       // Cycle 4,13,22 ...
 (p16)  ldf     f18 = [r6], 8      // Cycle 4,13,22 ...
 (p18)  stf     [r7] = f29, 16     // Cycle 23,32 ...
 (p18)  stf     [r8] = f30, 16     // Cycle 23,32 ...
 (p16)  fadd    f29 = f9, f19 ;;   // Cycle 14,23 ...
 (p16)  ldf     f9 = [r5], 8       // Cycle 6,15,24 ...
 (p16)  ldf     f19 = [r6], 8      // Cycle 6,15,24 ...
 (p16)  fadd    f30 = f10, f20 ;;  // Cycle 15,24 ...
 (p16)  ldf     f10 = [r5], 8      // Cycle 7,16,25 ...
 (p16)  ldf     f20 = [r6], 8      // Cycle 7,16,25 ...
        br.ctop L1 ;;
```

However, with register rotation, the same loop can be scheduled with an initiation interval of just 2 clocks without unrolling (and 1.5 clocks if unrolled by 2):

```
 L1:
 (p24)  stf     [r7] = f57, 8      // Cycle 15,17...
 (p21)  fadd    f57  = f37, f47    // Cycle 9,11,13...
 (p16)  ldf     f32  = [r5], 8     // Cycle 0,2,4,6...
 (p16)  ldf     f42  = [r6], 8     // Cycle 0,2,4,6...
        br.ctop L1;;
```

It is thus often advantageous to modulo schedule and then unroll (if required). Please see Chapter 5, "Software Pipelining and Loop Support" for details on how to rewrite loops using this transformation.

6.3.1.1 Notes on FP Precision

The floating-point registers are 82 bits wide with 17 bits for exponent range, 64 bits for significand precision and 1 sign bit. During computation, the result range and precision is determined by the computational model chosen by the user. The computational model is indicated either statically in the instruction encoding, or dynamically via the precision control (PC) and widest-range-exponent (WRE) bits in the floating-point status register. Using an appropriate computational model, the user can minimize the error accumulation in the computation. In the above matrix multiply example, if the multiply and add computations are performed in full register file range and precision, the results (in accumulators) can hold 64 bits of precision and up to 17 bits of range for inputs that might be single precision numbers. With the rounding performed at the 64th precision bit (instead of the 24th for single precision) a smaller error is accumulated with each multiply and add. Furthermore, with 17 bits of range (instead of 8 bits for single precision) large positive and negative products can be added to the accumulator without overflow or underflow. In addition to providing more accurate results the extra range and precision can often enhance the performance of iterative computations that are required to be performed until convergence (as indicated by an error bound) is reached.

6.3.2 Multiply-Add Instruction

The Itanium architecture defines the fused multiply-add (`fma`) as the basic floating-point computation, since it forms the core of many computations (linear algebra, series expansion, etc.) and its latency in hardware is typically less than the sum of the latencies of an individual multiply operation (with rounding) implementation and an individual add operation (with rounding) implementation.

In computational loops that have a loop carried dependency and whose speed is often determined by the latency of the floating-point computation rather than the peak computational rate, the multiply-add operation can often be used advantageously. Consider the Livermore FORTRAN Kernel 9 – General Linear Recurrence Equations:

```
DO 191 k= 1,n
    B5(k+KB5I)= SA(k) + STB5 * SB(k)
    STB5= B5(k+KB5I) - STB5
191CONTINUE
```

Since there is a true data dependency between the two statements on variable `B5(k+KB5I))` and a loop-carried dependency on variable `STB5`, the loop number of clocks per iteration is entirely determined by the latency of the floating-point operations. In the absence of an `fma` type operation, and assuming that the individual multiply and add latencies are 5 clocks each and the loads are 8 cycles, the loop would be:

```
L1:
(p16)   ldf     f32 = [r5], 8          // Load SA(k)
(p16)   ldf     f42  = [r6], 8         // Load SB(k)
(p17)   fmul    f5  = f7, f43;;        // tmp,Clk 0,15 ...
(p17)   fadd    f6  = f33, f5 ;;       // B5,Clk 5,20 ...
(p17)   stf     [r7] = f6, 8           // Store B5
(p17)   fsub    f7  = f6, f7           // STB5,Clk 10,25 ..
        br.ctop L1 ;;
```

With an `fma`, the overall latency of the chain of operations decreases and assuming a 5 cycle `fma`, the loop iteration speed is now 10 clocks (as opposed to 15 clocks above).

```
L1:
(p16)   ldf     f32  = [r5], 8         // Load SA(k)
(p16)   ldf     f42 = [r6], 8          // Load SB(k)
(p17)   fma     f6  = f7, f43, f33;;   // B5,Clk 0,10 ...
(p17)   stf     [r7] = f6, 8           // Store B5
(p17)   fsub    f7  = f6, f7           // STB5,Clk 5,15 ..
        br.ctop L1 ;;
```

The fused multiply-add operation also offers the advantage of a single rounding error for the pair of computations which is valuable when trying to compute small differences of large numbers.

6.3.3 Software Divide/Square Root Sequence

To perform division or square root operations on the Itanium architecture, a software-based sequence of operations is used. The sequence consists of obtaining an initial guess (using `frcpa`/`frsqrta` instruction) and then refining the guess by performing Newton-Raphson iterations until the error is sufficiently small so that it may not affect the rounding of the result. Examples of double precision divide and square root sequences, optimized for latency and throughput, are provided below.

Note: For reduced precision, square and divide sequences can be completed with even fewer instructions.

6.3.3.1 Double Precision – Divide

Divide (Max Throughput) (10 Instructions, 8 Groups)	Divide (Min Latency) (13 Instructions, 7 Groups)

`     frcpa.s0 f8,p6 = f6,f7 ;;`
`(p6) fnma.s1 f9 = f7,f8,f1 ;;`
`(p6) fma.s1 f8 = f9,f8,f8`
`(p6) fma.s1 f9 = f9,f9,f0 ;;`
`(p6) fma.s1  f8 = f9 ,f8,f8`
`(p6) fma.s1 f9 = f9,f9,f0 ;;`
`(p6) fma.s1 f8 = f9,f8,f8 ;;`
`(p6) fma.d.s1 f9 = f6,f8,f0 ;;`
`(p6) fnma.d.s1 f6 = f7,f9,f6 ;;`
`(p6) fma.d.s0 f8 = f6,f8,f9`
	`     frcpa.s0 f8,p6 = f6,f7 ;;`
	`(p6) fma.s1 f9 = f6,f8,f0`
	`(p6) fnma.s1 f10 = f7,f8,f1 ;;`
	`(p6) fma.s1 f9 = f10,f9,f9`
	`(p6) fma.s1 f11 = f10,f10,f0`
	`(p6) fma.s1 f8 = f10,f8,f8 ;;`
	`(p6) fma.s1 f9 = f11,f9,f9`
	`(p6) fma.s1 f10 = f11,f11,f0`
	`(p6) fma.s1 f8 = f11,f8,f8 ;;`
	`(p6) fma.d.s1 f9 = f10,f9,f9`
	`(p6) fma.s1 f8 = f10,f8,f8 ;;`
	`(p6) fnma.d.s1 f6 = f7,f9,f6 ;;`
	`(p6) fma.d.s0 f8 = f6,f8,f9`

6.3.3.2 Double Precision – Square Root

Square Root (Max Throughput)[a] (14 Instructions, 10 Groups)	Square Root (Min Latency)[b] (17 Instructions, 9 Groups)

`     frsqrta.s0 f7,p6=f6 ;;`
`(p6) fma.s1 f8=f10,f7,f0`
`(p6) fma.s1 f7=f6,f7,f0 ;;`
`(p6) fnma.s1 f9=f7,f8,f10 ;;`
`(p6) fma.s1 f8=f9,f8,f8`
`(p6) fma.s1 f7=f9,f7,f7 ;;`
`(p6) fnma.s1 f9=f7,f8,f10 ;;`
`(p6) fma.s1 f8=f9,f8,f8`
`(p6) fma.s1 f7=f9,f7,f7 ;;`
`(p6) fnma.s1 f9=f7,f8,f10 ;;`
`(p6) fma.s1 f8=f9,f8,f8`
`(p6) fma.d.s1 f7=f9,f7,f7 ;;`
`(p6) fnma.s1 f9=f7,f7,f6 ;;`
`(p6) fma.d.s0 f7=f9,f8,f7 ;;`
	`     frsqrta.s0 f7,p6=f6 ;;`
	`(p6) fma.s1 f8=f9,f7,f0`
	`(p6) fma.s1 f7=f6,f7,f0 ;;`
	`(p6) fnma.s1 f9=f7,f8,f9 ;;`
	`(p6) fma.s1 f10=f11,f9,f10`
	`(p6) fma.s1 f11=f9,f9,f0`
	`(p6) fma.s1 f12=f13,f9,f12 ;;`
	`(p6) fma.s1 f10=f11,f10,f9`
	`(p6) fma.s1 f11=f11,f11,f0`
	`(p6) fma.s1 f9=f9,f12,f14 ;;`
	`(p6) fma.s1 f12=f10,f7,f7`
	`(p6) fma.s1 f7=f7,f11,f0`
	`(p6) fma.s1 f10=f11,f9,f10 ;;`
	`(p6) fma.d.s1 f7=f9,f7,f12`
	`(p6) fma.s1 f8=f10,f8,f8 ;;`
	`(p6) fnma.s1 f9=f7,f7,f6 ;;`
	`(p6) fma.d.s0 f7=f9,f8,f7 ;;`

[a] The following value is assumed preset: f10=1/2.

[b] The following values are assumed preset: f9=1/2, f10=3/2, f11=5/2, f12=63/8, f13=231/16, f14=35/8.

For divide, the first instruction (frcpa) provides an approximation (good to 8 bits) of the reciprocal of f7 and sets the predicate (p6) to 1, if the ratio f6/f7 can be obtained using the prescribed Newton-Raphson iterations. If, however, the ratio f6/f7 is special (finite/0, finite/infinite, etc) the final result of f6/f7 is provided in f8 and the predicate (p6) is cleared. For certain boundary conditions (when the operand values (f6 and f7) are well outside the single precision, double precision and even double-extended precision ranges) frcpa will cause a software assist fault and the software handler will produce the ratio f6/f7 and return it in f8 and clear the predicate (p6).

The multiple status fields provided in the FPSR are used in these sequences. S0 is the main (architectural) status field and it is written to by the first operation (frcpa) to signal any faults (V, Z, D), and by the last operation to signal any traps. The conditions of all intermediate operations are ignored by writing them to S1. Thus these sequences not only obtain the correct IEEE 754 specified result (in f8) but the flags are also set (in S0) as per the standard's requirements. If the divide is part of a speculative chain of operations that is using S2 as its status field, then S0 should be replaced with S2 in these sequences. S1 can still be used by the intermediate operations of all the divide sequences (i.e. those that target S0, S2, or S3) since its flags are all discarded.

When divide and square-root operations appear in vectorizable loops, it is often very advantageous to have these operations be performed in software rather than hardware. In software, these operations can be pipelined and the overall throughput be improved, whereas in hardware these operations are typically not pipelineable.

Another significant advantage of the software-based divide/square-root computations is that the accuracy of the result can be controlled by the user and can be traded off for speed. This trade-off is often used in graphics codes where the divide accuracy of about 14-bits suffices and the sequence can be shorter than that used for single or double precision.

6.3.4 Computational Models

The Itanium architecture offers complete user control of the computational model. The user can select the result's precision and range, the rounding mode, and the IEEE trap response. Appropriately selecting the computational model can result in code that has greater accuracy, higher performance, or both.

The register file format is uniform for the three memory data types – single, double and double-extended. Since all the computations are performed on registers (regardless of the data type of its content) operands of different types can be easily combined. Also since the conversion from the memory type to the register file format is done on loads automatically no extra operations are required to perform the format conversion.

The C syntax semantics is also easily emulated. Loads convert all input operands into the register file format automatically. Data operands of different types, now residing in register file format can be operated upon and all intermediate results coerced to double precision by statically indicating the result precision in the instruction encoding. The computation leading to the final result can specify the result precision and range (statically in the instruction encoding for single and double precision, and dynamically in the status field bits for double-extended precision). Compliance to the IA-32 FP computational style (range=extended, precision=single/double/extended) can also achieved using the status field bits.

6.3.5 Multiple Status Fields

The FPSR is divided into one main (architectural) status field and three additional identical status fields. These additional status fields could be used to performance advantage.

First, divide and square-root sequences (described in Section 6.3.3) contain operations that might cause intermediate results to overflow/underflow or be inexact even if the final result may not. In order to maintain correct IEEE flag status the status flags of these computations need to be discarded. One of these additional status fields (typically status field 1) can be used to discard these flags.

Second, speculating floating-point operations requires maintaining the status flags of the speculated operations distinct from the architectural status flags until the speculated operations are committed to architectural state (if they ever are). One of these additional status fields (typically status fields 2 or 3) can be used for this purpose.

Consider the Livermore FORTRAN kernel 16 – Monte Carlo Search

```
DO 470 k= 1,n
    k2= k2+1
    j4= j2+k+k
    j5= ZONE(j4)
    IF( j5-n      ) 420,475,450
415 IF( j5-n+II   ) 430,425,425
420 IF( j5-n+LB   ) 435,415,415
425 IF( PLAN(j5)-R) 445,480,440
430 IF( PLAN(j5)-S) 445,480,440
435 IF( PLAN(j5)-T) 445,480,440
440 IF( ZONE(j4-1)) 455,485,470
445 IF( ZONE(j4-1)) 470,485,455
450 k3= k3+1
    IF( D(j5)-(D(j5-1)*(T-D(j5-2))**2
    ,   +(S-D(j5-3))**2
    ,   +(R-D(j5-4))**2)) 445,480,440
455 m= m+1
    IF( m-ZONE(1) ) 465,465,460
460 m= 1
465 IF( i1-m) 410,480,410
470 CONTINUE
475 CONTINUE
480 CONTINUE
485 CONTINUE
```

Profiling indicates that the conditional after statement 450 is most frequently executed. It is therefore advantageous to speculatively execute the computation in the conditional while the conditionals in 415...445 are being evaluated. In the event that any of the conditionals in 415...445 cause the control to be moved on beyond 450 the results (and flags) of the speculatively computed operations (of the conditional after statement 450) can be discarded.

The availability of multiple additional status fields can allow a user to maintain multiple computational environments and to dynamically select among them on an operation by operation basis. One such use is in the implementation of interval arithmetic code where each primitive operation is required to be computed in two different rounding modes to determine the interval of the result.

6.3.6 Other Features

The Itanium architecture offers a number of other architectural constructs to enhance the performance of different computational situations.

6.3.6.1 Operand Screening Support

Operand screening is often a required or useful step prior to a computation. The operand may be screened to ensure that it is in a valid range (e.g. finite positive or zero input to square-root; non-zero divisor for divide) or it may be screened to take an early out – the result of the computation is predetermined or could be computed more efficiently in another way. The `fclass` instruction can be used to classify the input operand to either be or not be a part of a set of classes. Consider the following code used for screening invalid operands for square-root computation:

```
IF (A.EQ. NATVAL OR
    A.EQ. SNAN OR A.EQ. QNAN OR
    A.EQ. NEG_INF OR A.EQ. POS_INF OR
    A.LT. 0.0D0) THEN
    WRITE (*, “INVALID INPUT OPERAND”)
ELSE
    WRITE (*, “SQUARE-ROOT = “, SQRT(A))
ENDIF
```

The above conditional can be determined by two fclass instructions as indicated below:

```
        fclass.m    p1, p2 = f2, 0x1E3;;   // Detect NaTVal, NaN, +Inf or -Inf
(p2)    fclass.m    p1, p2 = f2, 0x01A     // Detect -Norm or -Unorm
```

The resultant complimentary predicates (p1 and p2) can be used to control the `ELSE` and `THEN` statements respectively.

6.3.6.2 Min/Max/AMin/AMax

The Itanium architecture provides direct instruction level support for the FORTRAN intrinsic `MIN(a,b)` or the equivalent C idiom: `a<b? a: b` and the FORTRAN intrinsic `MAX(b, a)` or the equivalent C idiom: `a<b? b: a`. These instructions can enhance performance by avoiding the function call overhead in FORTRAN, and by reducing the critical path in C. The instructions are designed to mimic the C statement behavior so that they can be generated by the compiler. They are also not commutative. By appropriately selecting the input operand order, the user can either ignore or catch NaNs.

Consider the problem of finding the minimum value in an array (similar to the Livermore FORTRAN kernel 24):

```
XMIN = X(1)
DO 24  k= 2,n
24 IF(X(k) .LT. XMIN)  XMIN = X(k)
```

Since NaNs are unordered, comparison with NaNs (including LT) will return false. Hence if the above code is implemented as:

```
    ldf        f5 = [r5], 8;;
L1: ldf        f6 = [r5], 8
    fmin       f5 = f6, f5
    br.cloop   L1 ;;
```

NaNs in the array (X) will be ignored.

If the value in the array X (loaded in f6) is a NaN, the new minimum value (in f5) will remain unchanged, since the NaN will fail the.LT. comparison and fmin will return the second argument – in this case the old minimum value in f5.

However, if the code is implemented as:

```
    ldf        f5 = [r5], 8;;
L1: ldf        f6 = [r5], 8
    fmin       f5 = f5, f6
    br.cloop   L1 ;;
```

NaNs in the array (X) will reset the minimum value.

Now, if the value in the array X (loaded in f6) is a NaN, the new minimum value (in f5) will be set to the NaN, since the NaN will fail the.LT. comparison and fmin will return the second argument – in this case the NaN in f6. In the next iteration, the new array value (loaded in f6) will become the new minimum.

famin/famax perform the comparison on the absolute value of the input operands (i.e. they ignore the sign bit) but otherwise operate in the same (non-commutative) way as the fmin/fmax instructions.

6.3.6.3 Integer/Floating-point Conversion

Unsigned integers are converted to their equivalently valued floating-point representations by simply moving the integer to the significand field of the floating-point register using the setf.sig instruction. The resulting floating-point value would be in its unnormal representation (unless the unsigned integer was greater than 2^{63}).

Conversions from signed integers to floating-point and from floating-point to signed or unsigned integers are accomplished by fcvt.xf and fcvt.fx/fcvt.fxu instructions respectively. However, since signed integers are converted directly to their canonical floating-point representations, they do not need to be normalized after conversion.

6.3.6.4 FP Subfield Handling

It is sometimes useful to assemble a floating-point value from its constituent fields. Multiplication and division of floating-point values by powers of two, for example, can be easily accomplished by appropriately adjusting the exponent. The Itanium architecture provides instructions that allow moving floating-point fields between the integer and floating-point register files. Division of a floating-point number by 2.0 is accomplished as follows:

```
getf.exp    r5  = f5            // Move S+Exp to int
add         r5  = r5, -1        // Sub 1 from Exp
setf.exp    f6  = r5            // Move S+Exp to FP
fmerge.se   f5  = f6, f5        // Merge S+E w/ Mant
```

Floating-point values can also be constructed from fields from different floating-point registers.

6.3.7 Memory Access Control

Recognizing the trend of growing memory access latency, and the implementation costs of high bandwidth, the Itanium architecture incorporates many architectural features to help manage the memory hierarchy and increase performance. As described in Section 6.2, memory latency and bandwidth are significant performance limiters in floating-point applications. The architecture offers features to address both these limitations.

In order to enhance the core bandwidth to the floating-point register file, the architecture defines load-pair instructions. In order to mitigate the memory latency, explicit and implicit data prefetch instructions are defined. In order to maximize the utilization of caches, the architecture defines locality attributes as part of memory access instructions to help control the allocation (and de-allocation) of data in the caches. For instances where the instruction bandwidth may become a performance limiter, the architecture defines machine hints to trigger relevant instruction prefetches.

6.3.7.1 Load-pair Instructions

The floating-point load pair instructions enable loading two contiguous values in memory to two independent floating-point registers. The target registers are required to be odd and even physical registers so that the machine can utilize just one access port to accomplish the register update.

Note: The odd/even pair restriction is on physical register numbers, not logical register numbers. A programming violation of this rule will cause an illegal operation fault.

For example, suppose a machine that can issue 2 FP instructions per cycle, provides sufficient bandwidth from the second level cache (L2) to sustain 2 load-pairs every cycle. Then loops that require up to 2 data elements (of 8 bytes each) per floating-point instruction can run at peak speeds when the data is resident in L2. A common example of such a case is a simple double precision dot product – DDOT:

```
DO 1 I = 1, N
1 C = C + A(I) * B(I)
```

The inner loop consists of two loads (for `A` and `B`) and a multiply-add (to accumulate the product on C). The loop would run at the latency of the fma due to the recurrence on C. In order to break the recurrence on C, the loop is typically unrolled and multiple partial accumulators are used.

```
DO 1 I = 1, N, 8
    C1 = C1 + A[I] * B[I]
    C2 = C2 + A[I+1] * B[I+1]
    C3 = C3 + A[I+2] * B[I+2]
    C4 = C4 + A[I+3] * B[I+3]
    C5 = C5 + A[I+4] * B[I+4]
    C6 = C6 + A[I+5] * B[I+5]
    C7 = C7 + A[I+6] * B[I+6]
1 C8 = C8 + A[I+7] * B[I+7]
C = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8
```

If normal (non-double pair) loads are used, the inner loop would consist of 16 loads and 8 fmas. If we assume the machine has two memory ports, this loop would be limited by the availability of M slots and run at a peak rate of 1 clock per iteration. However, if this loop is rewritten using 8 load-pairs (for `A[I]`, `A[I+1]` and `B[I]`, `B[I+1]` and `A[I+2]`, `A[I+3]` and `B[I+2]`, `B[I+3]` and so on) and 8 fmas this loop could run at a peak rate of 2 iterations per clock (or just 0.5 clocks per iteration) with just two M-units.

6.3.7.2 Data Prefetch

`lfetch` allows the advance prefetching of a line (defined as 32 bytes or more) of data into the cache from memory. Allocation hints can be used to indicate the nature of the locality of the subsequent accesses on that data and to indicate which level of cache that data needs to be promoted to.

While regular loads can also be used to achieve the effect of data prefetching, (if the load target is never used) lfetches can more effectively reduce the memory latency without using floating-point registers as targets of the data being prefetched. Furthermore `lfetch` allows prefetching the data to different levels of caches.

6.3.7.3 Allocation Control

Since data accesses have different locality attributes (temporal/non-temporal, spatial/non-spatial), The Itanium architecture allows annotating the data accesses (loads/stores) to reflect these attributes. Based on these annotations, the implementation can better manage the storage of the data in the caches.

Temporal and Non-temporal hints are defined. These attributes are applicable to the various cache levels. (Only two cache levels are architecturally identified). The non-temporal hint is best used for data that typically has no reuse with respect to that level of cache. The temporal hint is used for all other data (that has reuse).

6.4 Summary

This chapter describes the limiting factors for many scientific and floating-point applications: memory latency and bandwidth, functional unit latency, and number of available functional units. It also describes the important features of floating-point support in the Itanium architecture beyond the software-pipelining support described in Chapter 5, “Software Pipelining and Loop Support” that help to overcome some of these performance limiters. Architectural support for speculation, rounding, and precision control are also described.

Examples in the chapter include how to implement floating-point division and square root, common scientific computations such as reductions, use of features such as the `fma` instruction, and various Livermore kernels.

§

Intel® Itanium® Architecture

Software Developer's Manual

Revision 2.3

Volume 2: System Architecture

Intel® Itanium® Architecture Software Developer’s Manual

Volume 2: System Architecture

Revision 2.3

May 2010

Document Number: 245318

Contents

Part I: System Architecture Guide

Part II: System Programmer's Guide

Figures

Part I: System Architecture Guide

Part II: System Programmer’s Guide

Tables

Part I: System Architecture Guide

Part II: System Programmer's Guide

§

Part I: System Architecture Guide

About this Manual 1

The Intel® Itanium® architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The Itanium architecture features a revolutionary 64-bit instruction set architecture (ISA) which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the Itanium architecture is IA-32 instruction set compatibility.

The *Intel® Itanium® Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of the features of the Itanium architecture to help them optimize code.

1.1 Overview of Volume 1: Application Architecture

This volume defines the Itanium application architecture, including application level resources, programming environment, and the IA-32 application interface. This volume also describes optimization techniques used to generate high performance software.

1.1.1 Part 1: Application Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

Chapter 2, "Introduction to the Intel® Itanium® Architecture" provides an overview of the architecture.

Chapter 3, "Execution Environment" describes the Itanium register set used by applications and the memory organization models.

Chapter 4, "Application Programming Model" gives an overview of the behavior of Itanium application instructions (grouped into related functions).

Chapter 5, "Floating-point Programming Model" describes the Itanium floating-point architecture (including integer multiply).

Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium® System Environment" describes the operation of IA-32 instructions within the Itanium System Environment from the perspective of an application programmer.

1.1.2 Part 2: Optimization Guide for the Intel® Itanium® Architecture

Chapter 1, "About the Optimization Guide" gives an overview of the optimization guide.

Chapter 2, “Introduction to Programming for the Intel® Itanium® Architecture” provides an overview of the application programming environment for the Itanium architecture.

Chapter 3, “Memory Reference” discusses features and optimizations related to control and data speculation.

Chapter 4, “Predication, Control Flow, and Instruction Stream” describes optimization features related to predication, control flow, and branch hints.

Chapter 5, “Software Pipelining and Loop Support” provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 6, “Floating-point Applications” discusses current performance limitations in floating-point applications and features that address these limitations.

1.2 Overview of Volume 2: System Architecture

This volume defines the Itanium system architecture, including system level resources and programming state, interrupt model, and processor firmware interface. This volume also provides a useful system programmer's guide for writing high performance system software.

1.2.1 Part 1: System Architecture Guide

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Intel® Itanium® System Environment” introduces the environment designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications.

Chapter 3, “System State and Programming Model” describes the Itanium architectural state which is visible only to an operating system.

Chapter 4, “Addressing and Protection” defines the resources available to the operating system for virtual to physical address translation, virtual aliasing, physical addressing, and memory ordering.

Chapter 5, “Interruptions” describes all interruptions that can be generated by a processor based on the Itanium architecture.

Chapter 6, “Register Stack Engine” describes the architectural mechanism which automatically saves and restores the stacked subset (GR32 – GR 127) of the general register file.

Chapter 7, “Debugging and Performance Monitoring” is an overview of the performance monitoring and debugging resources that are available in the Itanium architecture.

Chapter 8, “Interruption Vector Descriptions” lists all interruption vectors.

Chapter 9, “IA-32 Interruption Vector Descriptions” lists IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment.

Chapter 10, “Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications” defines the operation of IA-32 instructions within the Itanium System Environment from the perspective of an Itanium architecture-based operating system.

Chapter 11, “Processor Abstraction Layer” describes the firmware layer which abstracts processor implementation-dependent features.

1.2.2 Part 2: System Programmer’s Guide

Chapter 1, “About the System Programmer’s Guide” gives an introduction to the second section of the system architecture guide.

Chapter 2, “MP Coherence and Synchronization” describes multiprocessing synchronization primitives and the Itanium memory ordering model.

Chapter 3, “Interruptions and Serialization” describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents and state. This chapter also describes system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 5, “Memory Management” introduces various memory management strategies.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

Chapter 9, “IA-32 Application Support” describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

Chapter 13, “Firmware Overview” introduces the firmware model, and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

Appendix A, “Code Examples” provides OS boot flow sample code.

1.3 Overview of Volume 3: Intel® Itanium® Instruction Set Reference

This volume is a comprehensive reference to the Itanium instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Instruction Reference” provides a detailed description of all Itanium instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “Pseudo-Code Functions” provides a table of pseudo-code functions which are used to define the behavior of the Itanium instructions.

Chapter 4, “Instruction Formats” describes the encoding and instruction format instructions.

Chapter 5, “Resource and Dependency Semantics” summarizes the dependency rules that are applicable when generating code for processors based on the Itanium architecture.

1.4 Overview of Volume 4: IA-32 Instruction Set Reference

This volume is a comprehensive reference to the IA-32 instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Base IA-32 Instruction Reference” provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-32 Intel® MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 4, “IA-32 SSE Instruction Reference” provides a detailed description of all IA-32 SSE instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.5 Terminology

The following definitions are for terms related to the Itanium architecture and will be used throughout this document:

Instruction Set Architecture (ISA) – Defines application and system level resources. These resources include instructions and registers.

Itanium Architecture – The new ISA with 64-bit instruction capabilities, new performance- enhancing features, and support for the IA-32 instruction set.

IA-32 Architecture – The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual***.

Itanium System Environment – The operating system environment that supports the execution of both IA-32 and Itanium architecture-based code.

Itanium Architecture-based Firmware – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

Processor Abstraction Layer (PAL) – The firmware layer which abstracts processor features that are implementation dependent.

System Abstraction Layer (SAL) – The firmware layer which abstracts system features that are implementation dependent.

1.6 Related Documents

The following documents can be downloaded at the Intel’s Developer Site at http://developer.intel.com:

- ***Dual-Core Update to the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization***– Document number 308065 provides model-specific information about the dual-core Itanium processors.
- ***Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*** – This document (Document number 251110) describes model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.
- ***Intel® Itanium® Processor Reference Manual for Software Development*** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.

- ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.
- ***Intel® Itanium® Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.

The following document can be downloaded at the Unified EFI Forum website at http://www.uefi.org:

- ***Unified Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware.

1.7 Revision History

Date of Revision	Revision Number	Description
March 2010	2.3	Added information about illegal virtualization optimization combinations and IIPA requirements. Added Resource Utilization Counter and PAL_VP_INFO. PAL_VP_INIT and VPD.vpr changes. New PAL_VPS_RESUME_HANDLER parameter to indicate RSE Current Frame Load Enable setting at the target instruction. PAL_VP_INIT_ENV implementation-specific configuration option. Minimum Virtual address increased to 54 bits. New PAL_MC_ERROR_INFO health indicator. New PAL_MC_ERROR_INJECT implementation-specific bit fields. MOV-to_SR.L reserved field checking. Added virtual machine disable. Added variable frequency mode additions to ACPI P-state description. Removed *pal_proc_vector* argument from PAL_VP_SAVE and PAL_VP_RESTORE. Added PAL_PROC_SET_FEATURES data speculation disable. Added Interruption Instruction Bundle registers. Min-state save area size change. PAL_MC_DYNAMIC_STATE changes. PAL_PROC_SET_FEATURES data poisoning promotion changes. ACPI P-state clarifications. Synchronization requirements for virtualization opcode optimization. New priority hint and multi-threading hint recommendations.

Date of Revision	Revision Number	Description
August 2005	2.2	Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 “Interruptions” and Section 11.2.2 PALE_RESET Exit State for details. Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10. Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of `ptc.l` instruction in Vol 3 for details. Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details. Introduced Unimplemented Instruction Address fault. Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details. Architecture extensions for processor virtualization. All instructions which must be last in an instruction group results in undefined behavior when this rule is violated. Added architectural sequence that guarantees increasing ITC and PMD values on successive reads. Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures. Allows IPI-redirection feature to be optional. Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block. Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details. “Bus parking” feature is now optional for PAL_BUS_GET_FEATURES. Introduced low-power synchronization primitive using `hint` instruction. FR32-127 is now preserved in PAL calling convention. New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges. Performance Monitor Data (PMD) registers are no longer sign-extended. New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details. Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure. Clarified PMU interrupts as edge-triggered. Modified ‘proc_number’ parameter in PAL_LOGICAL_TO_PHYSICAL procedure. Modified pal_copy_info alignment requirements. New bit in PAL_PROC_GET_FEATURES for variable P-state performance. Clarified descriptions for check_target_register and check_target_register_sof. Various fixes in dependency tables in Vol 3 Ch 5. Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5. Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3. Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.

Date of Revision	Revision Number	Description
August 2002	2.1	Added Predicate Behavior of alloc Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3). Added New fc.i Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3). Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2). Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2). Added Spontaneous NaT Generation on Speculative Load (ld.s) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3). Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2). Added Freeze Bit Functionality in Context Switching and Interrupt Generation Clarification (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Volume 2) Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2). Added capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Volume 2). Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2). Added Fault Handling Semantics for lfetch.fault Instruction (Section 2.2, Part I, Volume 3).
December 2001	2.0	Volume 1: Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1). IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3). Load instructions change (Section 4.4.1).

Date of Revision	Revision Number	Description
		Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Chapter 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4). Memory attribute transitions – added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation – changed behavior of DCR 'lc' bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2).
		Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3).

Date of Revision	Revision Number	Description
		Volume 2: Clarifications regarding “reserved” fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10). FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4). Clarification regarding ordering data dependency. Out-of-order IPI delivery is now allowed (Chapters 4 and 5). Content of EFLAG field changed in IIM (p. 9-24). PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11). PAL_CHECK processor state parameter changes (Chapter 11). PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11). PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11). PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11). PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11). PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11). Clarified memory ordering changes (Chapter 13). Clarification in dependence violation table (Appendix A).
		Volume 3: fmix instruction page figures corrected (Chapter 2). Clarification of “reserved” fields in ITIR (Chapters 2 and 3). Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/ instruction group (Chapters 2 and 4). IA-32 JMPE instruction page typo fix (p. 5-238). Processor Serial Number feature removed (Chapter 5).
January 2000	1.0	Initial release of document.

§

Intel® Itanium® System Environment 2

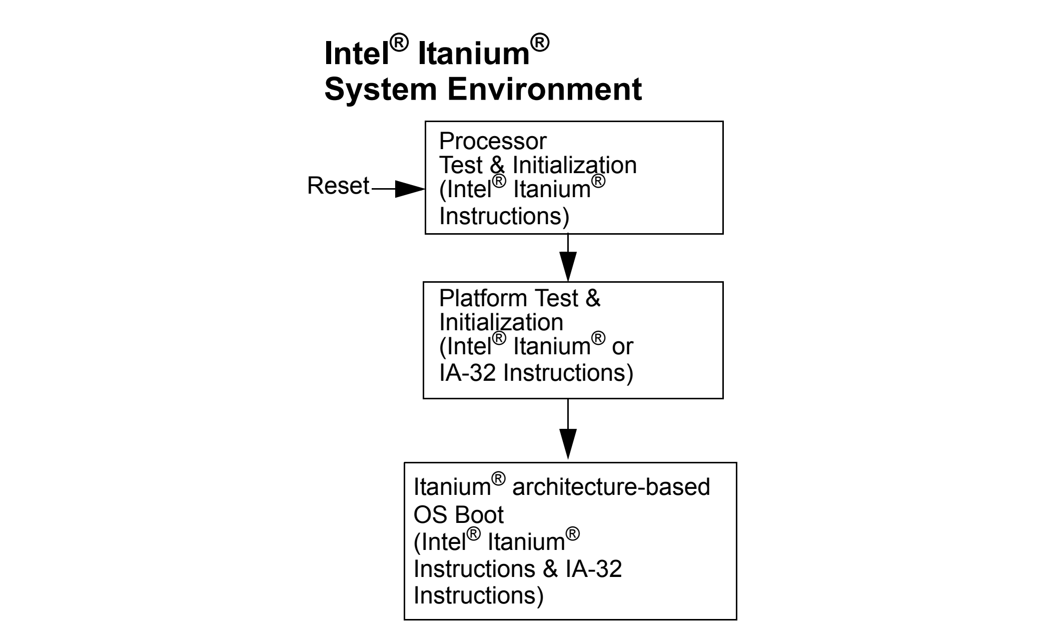
As described in Section 2.1, "Operating Environments" on page 1:13, the Itanium System Environment supports Itanium architecture-based operating systems. The architectural model also supports a mixture of IA-32 and Itanium architecture-based application code within an Itanium architecture-based operating system.

The system environment determines the set of processor system resources seen by the operating system. These resources include: virtual memory management, physical memory attributes, external interrupt mechanisms, exception and interrupt delivery, machine check architectures, debug, performance monitoring, control registers, and the set of privileged instructions.

2.1 Processor Boot Sequence

Figure 2-1 shows the defined boot sequence. Unlike IA-32 processors, which power up in 32-bit Real Mode, processors in the Itanium processor family power up in the Itanium System Environment running Itanium architecture-based code. Processor initialization, testing, memory, and platform initialization/testing are performed by processor firmware. Mechanisms are provided to execute Real Mode IA-32 boot BIOSs and device drivers during the boot sequence.

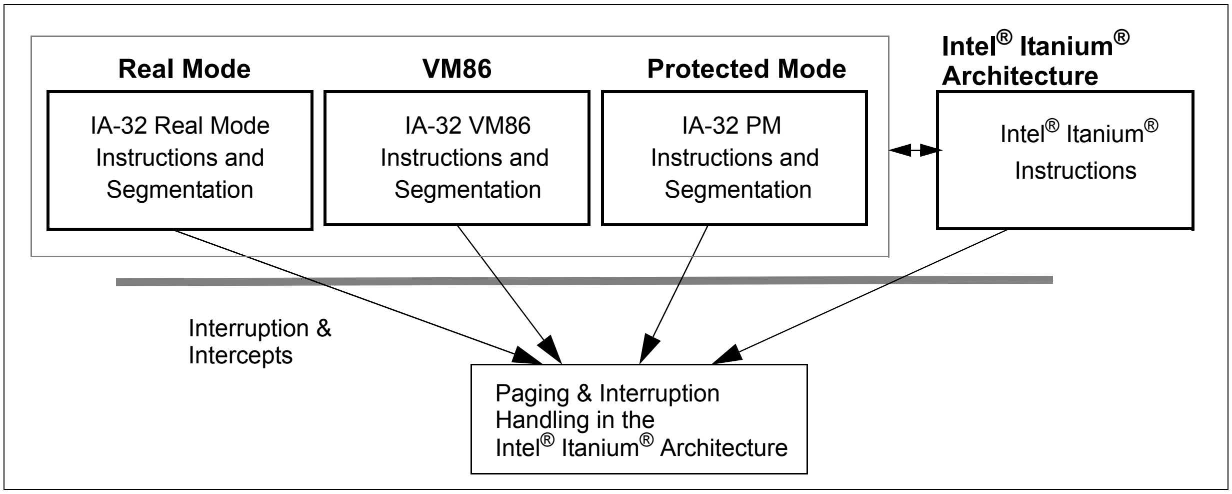
Figure 2-1. System Environment Boot Flow


Intel® Itanium® System Environment
Reset
Processor Test & Initialization (Intel® Itanium® Instructions)
Platform Test & Initialization (Intel® Itanium® or IA-32 Instructions)
Itanium® architecture-based OS Boot (Intel® Itanium® Instructions & IA-32 Instructions)

2.2 Intel® Itanium® System Environment Overview

The Itanium System Environment is designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications. IA-32 applications can interact with Itanium architecture-based operating systems, applications and libraries within this environment. Both IA-32 application level code and Itanium instructions can be executed by the operating system and user level software. The entire machine state, including the IA-32 general registers and floating-point registers, segment selectors and descriptors is accessible to Itanium architecture-based code. As shown in Figure 2-2, all major IA-32 operating modes are fully supported.

Figure 2-2. Intel® Itanium® System Environment


Real Mode
VM86
Protected Mode
Intel® Itanium® Architecture
IA-32 Real Mode Instructions and Segmentation
IA-32 VM86 Instructions and Segmentation
IA-32 PM Instructions and Segmentation
Intel® Itanium® Instructions
Interruption & Intercepts
Paging & Interruption Handling in the Intel® Itanium® Architecture

In the Itanium system environment, Itanium architecture operating system resources supersede all IA-32 system resources. Specifically, the IA-32 defined set of control, test, debug, machine check registers, privilege instructions, and virtual paging algorithms are replaced by the Itanium architecture system resources. When IA-32 code is running on an Itanium architecture-based operating system, the processor directly executes all performance critical but non-sensitive IA-32 application level instructions. Accesses to sensitive system resources (interrupt flags, control registers, TLBs, etc.) are intercepted into the Itanium architecture-based operating system. Using this set of intervention hooks, an Itanium architecture-based operating system can emulate or virtualize an IA-32 system resource for an IA-32 application, OS, or device driver.

The Itanium system architecture features are presented in the following chapters:

- Chapter 3, "System State and Programming Model" describes system resources.
- Chapter 4, "Addressing and Protection" describes the virtual memory architecture.
- Chapter 5, "Interruptions" defines the interrupt and exception architecture.
- Chapter 6, "Register Stack Engine" describes the register stack engine.
- Chapter 7, "Debugging and Performance Monitoring" describes debug and performance monitoring hooks.
- Chapter 8, "Interruption Vector Descriptions" describes interruption handler entry points.

Additional support for IA-32 applications in the Itanium system environment is defined by chapters:

- Chapter 9 describes IA-32 interruption handler entry points.
- Chapter 10, “Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications”describes how IA-32 applications interact with Itanium architecture-based operating systems.

§

System State and Programming Model 3

This chapter describes the architectural state visible only to an operating system and defines system state programming models. It covers the functional descriptions of all the system state registers, descriptions of individual fields in each register, and their serialization requirements. The virtual and physical memory management details are described in Chapter 4, "Addressing and Protection." Interruptions are described in Chapter 5, "Interruptions."

Note: Unless otherwise noted, references to "interruption" in this chapter refer to IVA-based interruptions. See "Interruption Definitions" on page 2:95.

3.1 Privilege Levels

Four privilege levels, numbered from 0 to 3, are provided to control access to system instructions, system registers and system memory areas. Level 0 is the most privileged and level 3 the least privileged. Application instructions and registers can be accessed at any privilege level. System instructions and registers defined in this chapter can only be accessed at privilege level 0; otherwise, a Privilege Operation fault is raised. The processor maintains a Current Privilege Level (CPL) in the cpl field of the Processor Status Register (PSR). CPL can only be modified by controlled entry and exit points managed by the operating system. Virtual memory protection mechanisms control memory accesses based on the Privilege Level (PL) of the virtual page and the CPL.

3.2 Serialization

For all application and system level resources, apart from the control register file, the processor ensures values written to a register are observed by instructions in subsequent instruction groups. This is termed **data dependency**. For example, writes to general registers, floating-point and application registers are observed by subsequent reads of the same register. (See "Control Registers" on page 2:29 for control register serialization requirements.) For modifications of application level resources with side effects, the side effects are ensured by the processor to be observed by subsequent instruction groups. This is termed **implicit serialization**. Application registers (ARs), with the exception of the Interval Time Counter, the User Mask, when modified by `sum`, `rum`, and mov to psr.um, and the Current Frame Marker (CFM), are implicitly serialized. PMD registers have special serialization requirements as described in "Generic Performance Counter Registers" on page 2:156. All other application-level resources (GRs, FRs, PRs, BRs, IP, CPUID) have no side effects and so need not be serialized.

To avoid serialization overhead in privileged operating system code, system register resources are not implicitly serialized. The processor does not ensure modification of registers with side effects are observed by subsequent instruction groups. For system register resources other than control registers, the processor ensures data dependencies are honored (reads see the results of prior writes to the same register). See Section 3.3.3, "Control Registers" and Table 3-3 on page 2:29 for control register serialization requirements. This approach simplifies hardware and allows for more efficient software operations. For example, during a low level context switch where there is no immediate use of loaded system registers, these registers can be loaded without any serialization overhead. To ensure side effects are observed before a dependent instruction is fetched or executed, two serialization operations are provided: **instruction serialization** and **data serialization**.

3.2.1 Instruction Serialization

Instruction serialization ensures that modifications to processor resources are observed before subsequent instruction group fetches are re-initiated. Software must use an instruction serialization operation before any instruction group that is dependent upon the modified system resource. Resource side effects may be observed at any point before the explicit serialization operation.

Modification of the following system resources (if the modification affects instruction fetching) require instruction serialization: RR, PKR, ITR, ITC, IBR, PMC, PMD, PSR bits as defined in "Processor Status Register (PSR)" on page 2:23 and Control Registers as defined in "Control Registers" on page 2:29.

The instructions Return from Interruption (`rfi`) and Instruction Serialize (`srlz.i`) perform explicit instruction serialization.

An interruption performs an implicit instruction serialization operation, so the first instruction group in the interruption handler will observe the serialized state.

```
Instruction Serialization Example:

mov ibr[reg]= reg    // move to instruction debug register
;;                   // end of instruction group
srlz.i               // ensure subsequent instruction fetches observe
                     // modification
;;                   // end of instruction group
inst                 // dependent instruction
```

Note: The serializing instruction, the instruction to be serialized, and any operations dependent on the serialization must be in three separate instruction groups.

3.2.2 Data Serialization

Data serialization ensures that modifications to processor resources affecting both execution and data memory accesses are observed. Software must issue a data serialize operation prior to the instruction dependent upon the modified resource. Data serialization can be issued within the same instruction group as the dependent instruction. Resource side effects may be observed at any point before the explicit serialization operation.

Modification of the following system resources require data serialization: RR, PKR, RUC, DTR, DTC, DBR, PMC, PMD, PSR bits as defined in "Processor Status Register (PSR)" on page 2:23 and Control Registers as defined in "Control Registers" on page 2:29.

The control registers are different from the general registers and other registers. Most control registers require an explicit data serialization between the writing of a control register and the reading of that same control register. (See Table 3-3 on page 2:29 for serialization requirements for specific control registers.)

The Data Serialize (`srlz.d`) instruction performs explicit data serialization. Instruction serialization operations (`rfi`, `srlz.i`, and interruptions) also perform a data serialization operation.

```
Data Serialization Example:

mov rr[reg] = reg    //move into region register
;;                   //end of instruction group
srlz.d               //serialize region register modification
ld                   //perform a dependent load
```

The serializing instruction and the instruction to be serialized (the one writing the resource) must be in two different instruction groups. Operations dependent on the serialization and the serialization can be in the same instruction group, but the `srlz` instruction must be before the dependent instruction slot.

3.2.3 Definition of In-flight Resources

When the value of a resource that requires an explicit instruction or data serialization is changed by one or more writers, that resource is said to be **in-flight** until the required serialization is performed. There can be multiple in-flight values if multiple writers have occurred since the last serialization.

An instruction that reads an in-flight resource will see one of the in-flight values or the state prior to any of the unserialized writers. However, whether such a reader sees the original or one of the in-flight values is not predictable.

For a reader of an in-flight resource, this definition includes (but is not limited to) the following possible outcomes:

- The reader of an in-flight resource may see the most-recently-serialized value or any of the in-flight values each time it is executed – seeing the value from a particular writer one time does not guarantee that the same writer’s value will be seen by that reader the next time.
- Multiple readers of an in-flight resource may see different values – each may see the most-recently-serialized value or any of the in-flight values, independent of what other readers may see.
- If a single execution of an instruction reads an in-flight resource more than once during its execution, each read may see a different value.

Thus, the only way to guarantee that the latest value is seen by a reader is to perform the required serialization.

3.3 System State

The architecture provides a rich set of system register resources for process control, interruptions handling, protection, debugging, and performance monitoring. This section gives an overview of these resources.

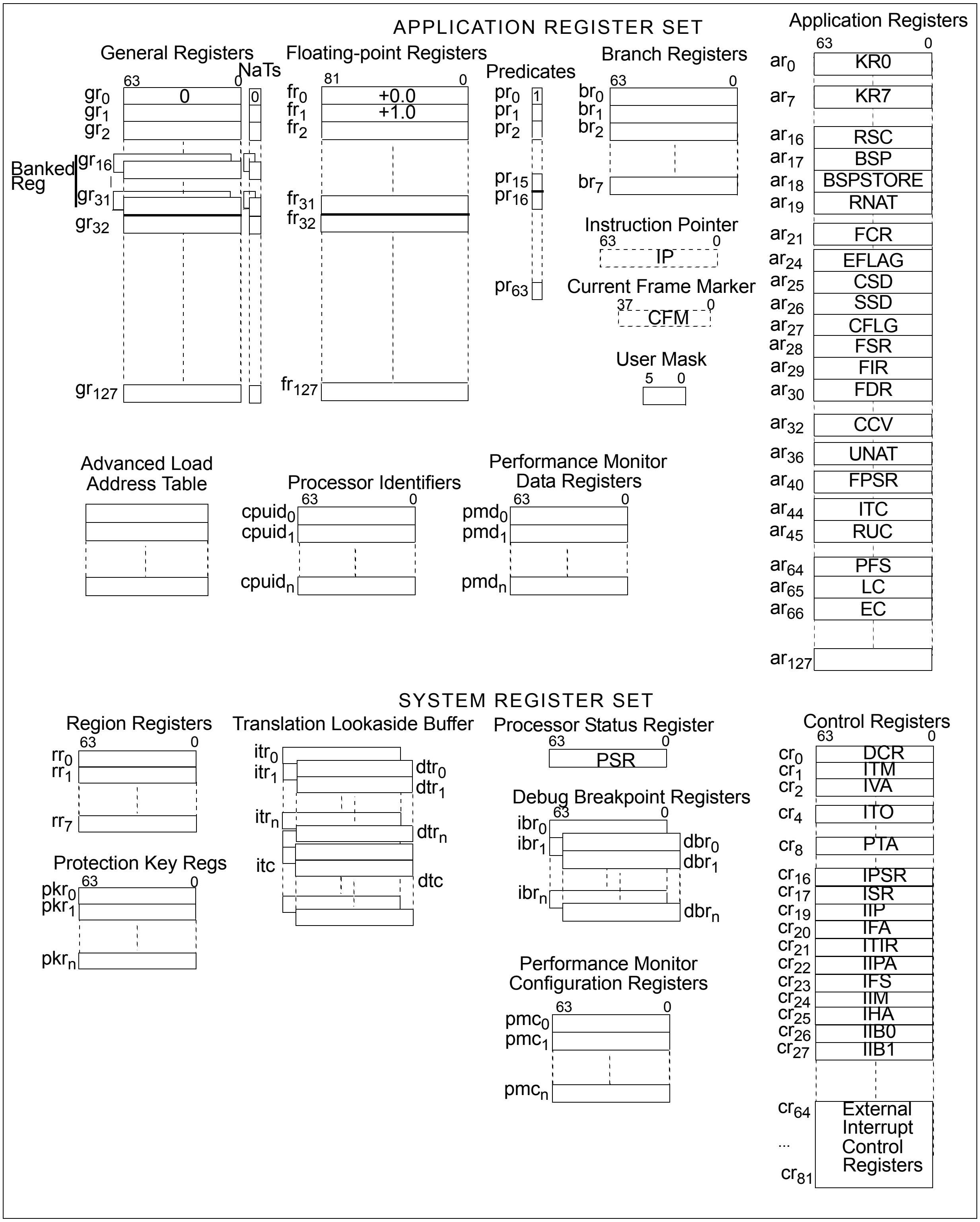
3.3.1 System State Overview

Figure 3-1 shows the set of all defined privileged system register resources. Application state as defined in "Application Register State" on page 1:23 is also accessible.

- **Processor Status Register (PSR)** – 64-bit register that maintains control information for the currently running process. See "Processor Status Register (PSR)" on page 2:23 for complete details.
- **Control Registers (CR)** – This register name space contains several 64-bit registers that capture the state of the processor on an interruption, enable system-wide features, and specify global processor parameters for interruptions and memory management. See "Control Registers" on page 2:29 for complete information.
- **Interrupt Registers** – These registers provide the capability of masking external interrupts, reading external interrupt vector numbers, programming vector numbers for internal processor asynchronous events and external interrupt sources. For complete information, see "Interrupts" on page 2:114.
- **Interval Timer Facilities** – A 64-bit interval timer is provided for privileged and non-privileged use and as a time base for performance measurements. Timing facilities are defined in detail in "Interval Time Counter and Match Register (ITC – AR44 and ITM – CR1)" on page 2:32.
- **Resource Utilization Facility** – A 64-bit resource utilization counter is provided for privileged and non-privileged use. This counts the number of Interval Timer cycles consumed by this logical processor. See Section 3.1.8.11, "Resource Utilization Counter (RUC – AR 45)" on page 1:31.
- **Debug Breakpoint Registers (DBR/IBR)** – 64-bit Data and 64-bit Instruction Breakpoint Register pairs (DBR, IBR) can be programmed to fault on reference to a range of virtual and physical addresses generated by either Itanium or IA-32 instructions. See "Debugging" on page 2:151 for details. The minimum number of DBR register pairs and IBR register pairs is 4 in any implementation. On some implementations, a hardware debugger may use two or more of these register pairs for its own use; see "Data and Instruction Breakpoint Registers" on page 2:152 for details.
- **Performance Monitor Configuration/Data Registers (PMC/PMD)** – Multiple performance monitors can be programmed to measure a wide range of user, operating system, or processor performance values. Performance monitors can be programmed to measure performance values from either IA-32 or Itanium instructions. Performance monitors are defined in "Performance Monitoring" on page 2:155. The minimum number of generic PMC/PMD register pairs in any implementation is 4.
- **Banked General Registers** – A set of 16 banked 64-bit general purpose registers, GR 16-GR 31, are available as temporary storage and register context when operating in low level interruption code. See "Banked General Registers" on page 2:42 for complete details.

- **Region Registers (RR)** – Eight 64-bit region registers specify the identifiers and preferred page sizes for multiple virtual address spaces. Refer to "Region Registers (RR)" on page 2:58 for complete information.
- **Protection Key Registers (PKR)** – At least sixteen 64-bit protection key registers contain protection keys and read, write, execute permissions for virtual memory protection domains. Please see the processor-specific documentation for further information on the number of Protection Key Registers implemented on the Itanium processor. Refer to "Protection Keys" on page 2:59 for details.
- **Translation Lookaside Buffer (TLB)** – Holds recently used virtual to physical address mappings. The TLB is divided into Instruction (ITLB), Data (DTLB), Translation Registers (TR) and Translation Cache (TC) sections. See "Translation Lookaside Buffer (TLB)" on page 2:47 for complete details. Translation Registers are software managed portions of the TLB and the Translation Cache section of the TLB is directly managed by the processor.

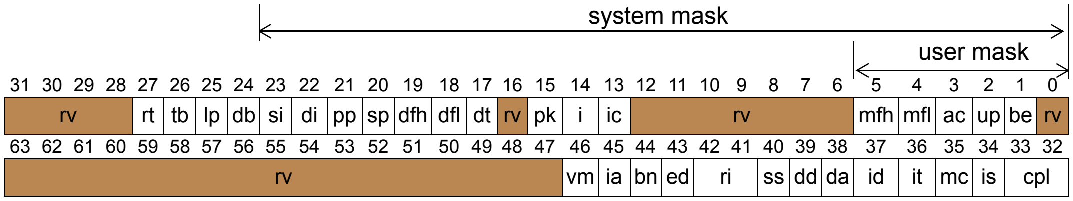
Figure 3-1. System Register Model



3.3.2 Processor Status Register (PSR)

The PSR maintains the current execution environment. The PSR is divided into four overlapping sections (See Figure 3-2): user mask bits (PSR{5:0}), system mask bits (PSR{23:0}), the lower half (PSR{31:0}), and the entire PSR (PSR{63:0}). PSR fields are defined in Table 3-2 along with serialization requirements for modification of each field and the state of the field after an interruption.

Figure 3-2. Processor Status Register (PSR)



The PSR instructions and their serialization requirements are defined in Table 3-1. These instructions explicitly read or write portions of the PSR. Other instructions also read and write portions of the PSR as described in Table 3-2 and Table 5-2.

Table 3-1. Processor Status Register Instructions

Mnemonic	Description	Operation	Instr. Type	Serialization Required
`sum` *imm*	Set user mask from immediate	PSR{5:0} ← PSR{5:0} | *imm*	M	implicit
`rum` *imm*	Reset user mask from immediate	PSR{5:0} ← PSR{5:0} & ~*imm*	M	implicit
`mov psr.um =` r_2	Move to user mask	PSR{5:0} ← GR[r_2]	M	implicit
`mov` r_1 `= psr.um`	Move from user mask	GR[r_1] ←PSR{5:0}	M	none
`ssm` *imm*	Set system mask from immediate	PSR{23:0} ← PSR{23:0} | *imm*	M	data/inst[a]
`rsm` *imm*	Reset system mask from immediate	PSR{23:0} ← PSR{23:0} &~*imm*	M	data/inst[a]
`mov psr.l =` r_2	Move to lower PSR	PSR{31:0} ← GR[r_2]	M	data/inst[a]
`mov` r_1 `= psr`	Move from PSR	GR[r_1] ←PSR{36:35,31:0}[b]	M	none
`bsw.0, bsw.1`	Bank switch	PSR{44} ← 0 or 1	B	implicit
`vmsw.0, vmsw.1`	Virtual machine switch	PSR{46} ← 0 or 1	B	implicit
`rfi`	Return From Interruption	PSR{63:0} ← IPSR	B	implicit

a. Based upon the resource being serialized, use data or instruction serialization.
b. All other bits of the PSR read as zero.

The user mask, PSR{5:0}, can be set and cleared by the Set User Mask (`sum`), Reset User Mask (`rum`) and Move to User Mask (`mov psr.um=`) instructions at any privilege level. For user mask modifications by `sum`, `rum` and `mov`, the processor ensures all side effects are observed before subsequent instruction groups.

The system mask, PSR{23:0}, can be set and cleared by the Set System Mask (`ssm`) and Reset System Mask (`rsm`) instructions. Software must issue the appropriate serialization operation before dependent instructions. The system mask instructions are privileged.

The lower half of the PSR, PSR{31:0}, can be written with the Move to Lower PSR (`mov psr.l=`) instruction. Software must issue the appropriate serialization operation before dependent instructions. The Move to Lower PSR instruction is privileged.

The PSR can be read with the Move from PSR (`mov =psr`) instruction. Only PSR{36:35} and PSR{31:0} are written to the target register by Move from PSR. PSR{63:37} and PSR{34:32} can only be read after an interruption by reading the state in IPSR. The entire PSR is updated from IPSR by the Return from Interruption (`rfi`) instruction. An `rfi` also implicitly serializes the PSR. Both Move from PSR and Return from Interruption are privileged.

Table 3-2. Processor Status Register Fields

Field	Bits	Description	Interruption State	Serialization Required
User Mask = PSR{5:0}				
rv	0	reserved		
be	1	Big-Endian – When 1, data memory references are big-endian. When 0, data memory references are little endian. This bit is ignored for IA-32 data references, which are always performed little-endian. Instruction fetches are always performed little endian.	DCR.be	data[a]
up	2	User Performance monitor enable – When 1, performance monitors configured as user monitors are enabled to count events (including IA-32). When 0, user configured monitors are disabled. See “Performance Monitoring” on page 2:155 for details.	unchanged	data[a] inst[b]
ac	3	Alignment Check – When 1, all unaligned data memory references result in an Unaligned Data Reference fault. When 0, unaligned data memory references may or may not result in a Unaligned Data Reference fault. See “Memory Datum Alignment and Atomicity” on page 2:93 for details. Unaligned semaphore references also result in a Unaligned Data Reference fault, regardless of the state of PSR.ac. For IA-32 instructions, if PSR.ac is 1 an unaligned IA-32 data memory reference raises an IA_32_Exception(AlignmentCheck) fault. When 0, additional IA-32 control bits as defined in Section 10.6.7, “Memory Alignment” also generate alignment checks.	0	data[a]
mfl	4	Lower (f2 .. f31) floating-point registers written – This bit is set to one when an Intel Itanium instruction completes that uses register f2..f31 as a target register. This bit is sticky and only cleared by an explicit write of the user mask. When leaving the IA-32 instruction set, PSR.mfl is set to 1 if PSR.dfl is 0, otherwise PSR.mfl is unmodified.	unchanged	data[a]

Table 3-2. Processor Status Register Fields (Continued)

Field	Bits	Description	Interruption State	Serialization Required
mfh	5	Upper (f32 .. f127) floating-point registers written – This bit is set to one when an Intel Itanium instruction completes that uses register f32..f127 as a target register. This bit is sticky and only cleared by an explicit write of the user mask. PSR.mfh is unmodified by IA-32 instruction set execution.	unchanged	data[a]
System Mask = PSR{23:0}				
ic	13	Interruption Collection – When 1 and an interruption occurs, the current state of the processor is loaded in IIP, IPSR, IIM and IFS; and additional registers defined in "Interruption Vector Descriptions" on page 2:165. When 0, IIP, IPSR, IIM and IFS are not modified on an interruption (see Table 8-1, "Writing of Interruption Resources by Vector" on page 2:166 for details). When 0, speculative load exceptions result in deferred exception behavior, regardless of the state of the DCR and ITLB deferral bits. Processor operation is undefined if PSR.ic is 0 and a transition is made to execute IA-32 code.	0	inst/data[c]
i	14	Interrupt Bit – When 1 and executing Intel Itanium instructions, unmasked pending external interrupts will interrupt the processor by transferring control to the external interrupt handler. When 0, pending external interrupts do not interrupt the processor. The effect of clearing PSR.i via Reset System Mask (`rsm`) instructions is observed by the next instruction. Toggling PSR.i from one to zero via Move to PSR.l requires data serialization. When executing IA-32 instructions, external interrupts are enabled if PSR.i and (CFLG.if is 0 or EFLAG.if is 1). NMI interrupts are enabled if PSR.i is 1 regardless of EFLAG.if.	0	clear: implicit serialization set: data[d]
pk	15	Protection Key enable – When 1 and PSR.it is 1, instruction references (including IA-32) check for valid protection keys. When 1 and PSR.dt is 1, data references (including IA-32) check for valid protection keys. When 1 and PSR.rt is 1, protection key checks are enabled for register stack references. When 0, neither instruction, data, nor register stack references are checked for valid protection keys. When PSR.dt, PSR.rt or PSR.it are 0, PSR.pk is ignored for the corresponding reference.	unchanged	inst/data[e]
rv	12:6, 16	reserved		
dt	17	Data address Translation – When 1, virtual data addresses are translated and access rights checked. When 0, data accesses use physical addressing. PSR.dt must be 1 when entering IA-32 code, otherwise processor operation is undefined.	unchanged/0[j]	inst/data[c]
dfl	18	Disabled Floating-point Low register set – When 1, a read or write access to f2 through f31 results in a Disabled Floating-Point Register fault. When 1, all IA-32 FP, Intel SSE and Intel MMX technology instructions raise a Disabled FP Register fault (regardless whether the instruction actually references f2-31).	0	data

Table 3-2. Processor Status Register Fields (Continued)

Field	Bits	Description	Interruption State	Serialization Required
dfh	19	Disabled Floating-point High register set – When 1, a read or write access to f32 through f127 results in a Disabled Floating-Point Register fault. When 1, a Disabled FP Register fault is raised on the first IA-32 target instruction following a `br.ia` or `rfi`, regardless whether f32-127 are referenced.	0	data
sp	20	Secure Performance monitors – Controls the ability of non-privileged code (including IA-32 code) to read non-privileged performance monitors. See Table 7-5 on page 2:158 for values returned by PMD read instructions. Also, when 0, PSR.up can be modified by user mask instructions; otherwise, PSR.up is unchanged by user mask instructions. When 1 or CFLG.pce is 0, non-privileged IA-32 performance monitor reads (via `rdpmc`) raise an IA_32_Exception(GPFault).	0	data
pp	21	Privileged Performance monitor enable – When 1, monitors configured as privileged monitors are enabled to count events (including IA-32 events). When 0, privileged monitors are disabled. See "Performance Monitoring" on page 2:155 for details.	DCR.pp	inst/data[e]
di	22	Disable Instruction set transition – When 1, attempts to switch instruction sets via the IA-32 `jmpe` or `br.ia` instructions results in a Disabled Instruction Set Transition fault. This bit doesn't restrict instruction set transitions due to interruptions or `rfi`.	0	data
si	23	Secure Interval timer – When 1, the Interval Time Counter (ITC) register and the Resource Utilization Counter (RUC) are readable only by privileged code; non-privileged reads result in a Privileged Register fault. When 0, ITC and RUC are readable at any privilege level. System software can secure the ITC from non-privileged IA-32 access by setting either PSR.si or CFLG.tsd to 1. When secured, an IA-32 rdtsc (read time stamp counter) instruction at any privilege level other than the most privileged raises an IA_32_Exception(GPfault)	0	data
PSR.l = PSR{31:0}				
db	24	Debug Breakpoint fault – When 1, data and instruction address breakpoints are enabled and can cause an Data/Instruction Debug fault. When 1, IA-32 instruction address breakpoints are enabled and can cause an IA_32_Exception(Debug) fault.When 1, IA-32 data address breakpoints are enabled and can cause an IA_32_Exception(Debug) Trap.When 0, address breakpoint faults and traps are disabled.	0	inst/data[e]
lp	25	Lower Privilege transfer trap – When 1, a Lower Privilege Transfer trap occurs whenever a taken branch lowers the current privilege level (numerically increases). This bit is ignored during IA-32 instruction set execution.	0	data

Table 3-2. Processor Status Register Fields (Continued)

Field	Bits	Description	Interruption State	Serialization Required
tb	26	Taken Branch trap – When 1, the successful completion of a taken branch results in a Taken Branch trap. `rfi` and interruptions can not raise a Taken Branch trap. When 1, successful completion of a taken IA-32 branch results in an IA_32_Exception(Debug) trap.	0	data
rt	27	Register stack Translation – When 1, register stack accesses are translated and access rights are checked. When 0, register stack accesses use physical addressing. PSR.dt is ignored for register stack accesses. The register stack engine must be in enforced lazy mode (RSC.mode = 00) when modifying this bit; otherwise, processor behavior is undefined. During IA-32 instruction execution this bit is ignored and the register stack is disabled.	unchanged	data
rv	31:28	reserved		
PSR{63:0}				
cpl[f]	33:32	Current Privilege Level –The current privilege level of the processor (including IA-32). Controls accessibility to system registers, instructions and virtual memory pages. A value of 0 is most privileged, a value of 3 is least privileged. Written by the `rfi`, `epc`, and `br.ret` instructions. PSR.cpl is unchanged by the `jmpe` and `br.ia` instructions. PSR.cpl cannot be updated by any IA-32 instructions.	0	rfi[g]
is	34	Instruction Set – When 0, Intel Itanium instructions are executing. When 1, IA-32 instructions are executing. Written by the `rfi` and `br.ia` instructions and the IA-32 `jmpe` instruction.	0	rfi[g], br.ia[h]
mc	35	Machine Check abort mask – When 1, machine check aborts are masked. When 0, machine check aborts can be delivered (including IA-32 instruction set execution). Processor operation is undefined if PSR.mc is 1 and a transition is made to execute IA-32 code.	unchanged/1[i]	rfi[g]
it	36	Instruction address Translation – When 1, virtual instruction addresses are translated and access rights checked. When 0, instruction accesses use physical addressing. PSR.it must be 1 when entering IA-32 code, otherwise processor operation is undefined.	unchanged/0[j]	rfi[g]
id	37	Instruction Debug fault disable – When 1, Instruction Debug faults are disabled on the first restart instruction in the current bundle.[k] When PSR.id is 1 or EFLAG.rf is 1, IA-32 instruction debug faults are disabled for one IA-32 instruction. PSR.id and EFLAG.rf are set to 0 after the successful execution of each IA-32 instruction.	0	rfi[g]
da	38	Disable Data Access and Dirty-bit faults – When 1, Data Access and Dirty-Bit faults are disabled on the first restart instruction in the current bundle or for the first mandatory RSE reference following the `rfi`.[k] IA-32 Access/Dirty-bit faults are not affected by PSR.da.[l]	0	rfi[g]
dd	39	Data Debug fault disable – When 1, Data Debug faults are disabled on the first restart instruction in the current bundle or for the first mandatory RSE reference.[k] IA-32 Data Debug traps are not affected by PSR.dd.[l]	0	rfi[g]

Table 3-2. Processor Status Register Fields (Continued)

Field	Bits	Description	Interruption State	Serialization Required
ss	40	Single Step enable – When 1, a Single Step trap occurs following the successful execution of the first restart instruction in the current bundle. Instruction slots 0, 1, and 2 can be single stepped. When 1 or EFLAG.tf is 1, an IA_32_Exception(Debug) trap is taken after each IA-32 instruction.	0	rfi[g]
ri	42:41	Restart Instruction – Set on an interruption, indicating the next instruction in the bundle to be executed. When the next instruction is the L+X instruction of an MLX, this field is set to the value 1. When restarting instructions with `rfi`, this field in IPSR specifies which instruction(s) in the bundle are restarted. The specified and subsequent instructions are restarted, all instructions prior to the restart point are ignored. 0 – restart execution at instruction slot 0 1 – restart execution at instruction slot 1 2 – restart execution at instruction slot 2 3 – reserved Except at an interruption and for the first restart instruction following an `rfi`, the value of this field is undefined. This field is set to 0 after any interruption from the IA-32 instruction set and is ignored when IA-32 instructions are restarted.	instruction pointer	rfi[g]
ed	43	Exception Deferral – When 1, if the first restart instruction in the current bundle is a speculative load, the operation is forced to indicate a deferred exception by setting the load target register to NaT or NaTVal. No memory references are performed, however any address post increments are performed. If the operation is a speculative advanced load, the ALAT entry corresponding to the load address and target register is purged. If the operation is an `lfetch` instruction, memory promotion is not performed, however any address post increments are performed. When 0, exception deferral is not forced on restarted speculative loads. If the first restart instruction is not a speculative load or `lfetch` instruction, this bit is ignored.[kl]	0	rfi[g]
bn	44	register Bank – When 1, registers GR16 to GR31 for bank 1 are accessible. When 0, registers GR16 to GR31 for bank 0 are accessible. Written by `rfi` and `bsw` instructions.	0	implicit[m]
ia	45	Disable Instruction Access-bit faults – When 1, Instruction Access-Bit faults are disabled on the first restart instruction in the current bundle.[k] IA-32 Access-bit faults are not affected by PSR.ia.[l]	0	rfi[g]
vm	46	Virtual Machine – When 1, an attempt to execute certain instructions results in a Virtualization fault. Implementation of this bit is optional. If the bit is not implemented, it is treated as a reserved bit. Written by the `rfi` and `vmsw` instructions.	0	rfi, vmsw: implicit[n]
rv	63:47	reserved		

a. User mask bits are implicitly serialized if accessed via user mask instructions; `sum`, `rum`, and move to User Mask. If modified with system mask instructions; `rsm`, `ssm` and move to PSR.l, software must explicitly serialize to ensure side effects are observed before dependent instructions.
b. User mask modification serialization is implicit only for monitoring data execution events. Software should issue instruction serialization operations before monitoring instruction events to achieve better accuracy.
c. Requires instruction serialization to guarantee that VHPT walks initiated on behalf of an instruction reference observe the new value of this bit. Otherwise, data serialization is sufficient to guarantee that the new value is observed.
d. The effect of masking external interrupts with `rsm` is observed by the next instruction. However, the processor does not ensure unmasking interruptions with ssm is immediately observed. Software can issue a data serialization operation to ensure the effects of setting PSR.i are observed before a given point in program execution.
e. Requires instruction or data serialization, based on whether the dependent “use” is an instruction fetch access or data access.
f. CPL can be modified due to interruptions, Return From Interruption (`rfi`), Enter Privilege Code (`epc`), and Branch Return (`br.ret`) instructions.
g. Can only be modified by the Return From Interruption (`rfi`) instruction. `rfi` performs an explicit instruction and data serialization operation.
h. Modification of the PSR.is bit by a `br.ia` instruction set is implicitly instruction serialized.
i. PSR.mc is set to 1 after a machine check abort or INIT; otherwise, unmodified on interruptions.
j. After an interruption this bit is normally unchanged, however after a PAL-based interruption this bit is set to 0.
k. This bit is set to 0 after the successful execution of each instruction in a bundle except for `rfi` which may set it to 1.
l. This bit is ignored when restarting IA-32 instructions and set to zero when `br.ia` or `rfi` successfully complete and before the first IA-32 instruction starts execution.
m. After an interruption, `rfi`, or `bsw` the processor ensures register accesses are made to the new register bank. For interruptions, `rfi` and `bsw`, the processor ensures all register accesses and outstanding loads prior to the bank switch operate on the prior register bank.
n. Can be modified by the Return From Interruption (`rfi`) and Virtual Machine Switch (`vmsw`) instructions. `rfi` performs an explicit instruction and data serialization operation. Modification of PSR.vm bit by the `vmsw` instruction is implicitly serialized.

3.3.3 Control Registers

Table 3-3 defines all registers in the control register name space along with serialization requirements to ensure side effects are observed by subsequent instructions. However, reads of a control register must be data serialized with prior writes to the same register. The serialization required column only refers to the side effects of the data value.

Writes to read-only registers (IVR, IRR0-3) result in an Illegal Operation fault, accesses to reserved registers result in a Illegal Operation fault. Accesses can only be performed by `mov` to/from instructions defined in Table 3-4 at privilege level 0; otherwise, a Privileged Operation fault is raised.

Table 3-3. Control Registers

	Register	Name	Description	Serialization Required
Global Control Registers	CR0	DCR	Default Control Register	inst/data
	CR1	ITM	Interval Timer Match register	data[a]
	CR2	IVA	Interruption Vector Address	inst[a]
	CR3		reserved	
	CR4	ITO	Interval Timer Offset Register	data[a]
	CR5-7		reserved	
	CR8	PTA	Page Table Address	inst/data[b]
	CR9-15		reserved	

Table 3-3. Control Registers (Continued)

	Register	Name	Description	Serialization Required
Interruption Control Registers	CR16	IPSR	Interruption Processor Status Register	implied[d]
	CR17	ISR	Interruption Status Register	implied[c]
	CR18		reserved	
	CR19	IIP	Interruption Instruction Pointer	implied[d]
	CR20	IFA	Interruption Faulting Address	implied[d]
	CR21	ITIR	Interruption TLB Insertion Register	implied[d]
	CR22	IIPA	Interruption Instruction Previous Address	implied[c]
	CR23	IFS	Interruption Function State	implied[d,e]
	CR24	IIM	Interruption Immediate register	implied[c]
	CR25	IHA	Interruption Hash Address	implied[c]
	CR26	IIB0	Interruption Instruction Bundle 0	implied[c]
	CR27	IIB1	Interruption Instruction Bundle 1	implied[c]
Reserved	CR28-63		reserved	
Interrupt Control Registers	CR64	LID	Local Interrupt ID	data[a]
	CR65	IVR	External Interrupt Vector Register (read only)	data[a]
	CR66	TPR	Task Priority Register	data[a]
	CR67	EOI	End Of External Interrupt	data[a]
	CR68	IRR0	External Interrupt Request Register 0 (read only)	data[a]
	CR69	IRR1	External Interrupt Request Register 1 (read only)	data[a]
	CR70	IRR2	External Interrupt Request Register 2 (read only)	data[a]
	CR71	IRR3	External Interrupt Request Register 3 (read only)	data[a]
	CR72	ITV	Interval Timer Vector	data[a]
	CR73	PMV	Performance Monitoring Vector	data[a]
	CR74	CMCV	Corrected Machine Check Vector	data[a]
	CR75-79		reserved	reserved
	CR80	LRR0	Local Redirection Register 0	data[a]
	CR81	LRR1	Local Redirection Register 1	data[a]
Reserved	CR82-127		reserved	reserved

a. Serialization is needed to ensure external interrupt masking, new interval timer match values or new interruption table addresses are observed before a given point in program execution.

b. Serialization is needed to ensure new values in PTA are visible to the hardware Virtual Hash Page Table (VHPT) walker before a dependent instruction fetch or data access.

c. These registers are modified by the processor on an interruption or by an explicit move to these registers. There are no side effects when written.

d. These registers are implied operands to the rfi and/or TLB insert instructions. The processor ensures writes in previous instruction groups are observed by rfi and/or TLB insert instructions in subsequent instruction groups. These registers are also modified by the processor on an interruption, subsequent reads return the results of the interruption. There are no other side effects.

e. IFS written by a `cover` instruction followed by a move-from IFS is implicitly serialized.

Table 3-4. Control Register Instructions

Mnemonic	Description	Operation	Format
`mov` cr_3 = r_2	Move to control register	CR[r_3] ← GR[r_2]	M
`mov` r_1 = cr_3	Move from control register	GR[r_1] ← CR[r_3]	M

Table 3-4. Control Register Instructions (Continued)

Mnemonic	Description	Operation	Format
`srlz.i, rfi`	Serialize instruction references	Ensure side effects are observed by the instruction fetch stream	M
`srlz.d`	Serialize data references	Ensure side effects are observed by the execute and data streams	M

3.3.4 Global Control Registers

3.3.4.1 Default Control Register (DCR – CR0)

The DCR specifies default parameters for PSR values on interruption, some additional global controls, and whether speculative load faults can be deferred. Figure 3-3 and Table 3-5 define and describe the DCR fields.

Figure 3-3. Default Control Register (DCR – CR0)

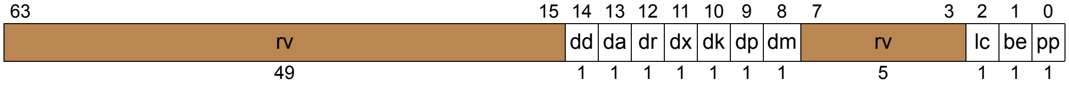


Table 3-5. Default Control Register Fields

Field	Bit	Description	Serialization Required
pp	0	Privileged Performance monitor default – On interruption, DCR.pp is loaded into PSR.pp.	data
be	1	Big-Endian default – When 1, Virtual Hash Page Table (VHPT) walker accesses are performed big-endian; otherwise, little-endian. On interruption, DCR.be is loaded into PSR.be.	inst
lc	2	IA-32 Lock Check enable – When 1, and an IA-32 atomic memory reference is defined as requiring a read-modify-write operation external to the processor under an external bus lock, an IA_32_Intercept(Lock) is raised. (IA-32 atomic memory references are defined to require an external bus lock for atomicity when the memory transaction is made to non-write-back memory or are unaligned across an implementation-specific non-supported alignment boundary.) When 0, and an IA-32 atomic memory reference is defined as requiring a read-modify-write operation external to the processor under external bus lock, the processor may either execute the transaction as a series of non-atomic transactions or perform the transaction with an external bus lock, depending on the processor implementation. Intel Itanium semaphore accesses ignore this bit. All unaligned Intel Itanium semaphore references generate an Unaligned Data Reference fault. All aligned Intel Itanium semaphore references made to memory that is neither write-back cacheable nor a NaTPage result in an Unsupported Data Reference fault.	data
dm	8	Defer TLB Miss faults only (VHPT data, Data TLB, and Alternate Data TLB faults) – When 1, and a TLB miss is deferred, lower priority Debug faults may still be delivered. A TLB miss fault, deferred or not, precludes concurrent Page not Present, Key Miss, Key Permission, Access Rights, or Access Bit faults. This bit is ignored by IA-32 instructions.	data
dp	9	Defer Page not Present faults only – When 1, and a Page not Present fault is deferred, lower priority Debug faults may still be delivered. A Page not Present fault, deferred or not, precludes concurrent Key Miss, Key Permission, Access Rights, or Access Bit faults. This bit is ignored by IA-32 instructions.	data

Table 3-5. Default Control Register Fields (Continued)

Field	Bit	Description	Serialization Required
dk	10	Defer Key Miss faults only – When 1, and a Key Miss fault is deferred, lower priority Access Bit, Access Rights or Debug faults may still be delivered. A Key Miss fault, deferred or not, precludes concurrent Key Permission faults. This bit is ignored by IA-32 instructions.	data
dx	11	Defer Key Permission faults only – When 1, and a Key Permission fault is deferred, lower priority Access Bit, Access Rights or Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
dr	12	Defer Access Rights faults only – When 1, and an Access Rights fault is deferred, lower priority Access Bit or Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
da	13	Defer Access Bit faults only – When 1, and an Access Bit fault is deferred, lower priority Debug faults may still be delivered. This bit is ignored by IA-32 instructions.	data
dd	14	Defer Debug faults – When 1, Data Debug faults on speculative loads are deferred. This bit is ignored by IA-32 instructions.	data
rv	7:3, 63:15	reserved	reserved

For the DCR exception deferral bits, when the bit is 1, and a speculative load results in the specified fault condition, and the speculative load’s code page exception deferral bit (ITLB.ed) is 1, the exception is deferred by setting the speculative load target register to NaT or NaTVal. Otherwise, the specified fault is taken on the speculative load. For a description of faults on speculative loads see “Deferral of Speculative Load Faults” on page 2:105.

Since DCR.be also controls byte ordering of VHPT references that are the result of instruction misses, DCR.be requires instruction serialization. Other DCR bits require data serialization only.

3.3.4.2 Interval Time Counter and Match Register (ITC – AR44 and ITM – CR1)

The Interval Time Counter (ITC) and Interval Timer Match (ITM) register support elapsed time notification, see Figure 3-4 and Figure 3-5.

Figure 3-4. Interval Time Counter (ITC – AR44)

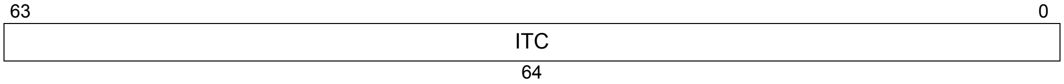
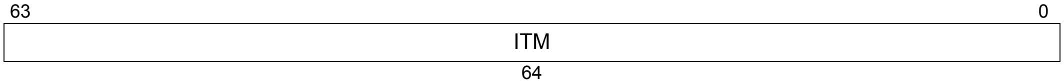


Figure 3-5. Interval Timer Match Register (ITM – CR1)



The ITC is a free-running 64-bit counter that counts up at a fixed relationship to the input clock to the processor. The ITC may be clocked at a somewhat lower frequency than the instruction execution frequency. This clocking relationship is described in the PAL procedure PAL_FREQ_RATIOS on page 2:393. The ITC is guaranteed to be clocked at a constant rate, even if the instruction execution frequency may vary. The ITC counting rate is not affected by power management mechanisms.

A sequence of reads of the ITC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads. Applications can directly sample the ITC for time-based calculations.

A 64-bit overflow condition can occur without notification. The ITC can be read at any privilege level if PSR.si is zero. The timer can be secured from non-privileged access by setting PSR.si to one. When secured, a read of the ITC by non-privileged code results in a Privileged Register fault. Writes to the ITC can only be performed at privilege level 0; otherwise, a Privileged Register fault is raised.

The IA-32 Time Stamp Counter (TSC) is similar to ITC. The ITC can be read by the IA-32 `rdtsc` (read time stamp counter) instruction. System software can secure the ITC from non-privileged IA-32 access by setting either PSR.si or CFLG.tsd to 1. When secured, an IA-32 read of the ITC at any privilege level other than the most privileged raises an IA_32_Exception(GPfault).

When the value in the ITC is equal to the value in the ITM an Interval Timer Interrupt is raised. Once the interruption is taken by the processor and serviced by software, the ITC may not necessarily be equal to the ITM. The ITM is accessible only at privilege level 0; otherwise, a Privileged Operation fault is raised.

The interval counter can be written, for initialization purposes, by privileged code. The ITC is not architecturally guaranteed to be synchronized with any other processor’s interval time counter in an multiprocessor system, nor is it synchronized with the wall clock. Software must calibrate interval timer ticks to wall clock time and periodically adjust for drift. In a multiprocessor system, a processor's ITC is not architecturally guaranteed to be clocked synchronously with the ITC's on other processors, and may not be clocked at the same nominal clock rate as ITC's on other processors. The platform firmware provides information on the clocking of processors in a multiprocessor system.

Modification of the ITC or ITM is not necessarily serialized with respect to instruction execution. Software can issue a data serialization operation to ensure the ITC or ITM updates and possible side effects are observed by a given point in program execution. Software must accept a level of sampling error when reading the interval timer due to various machine stall conditions, interruptions, bus contention effects, etc. Please see the processor-specific documentation for further information on the level of sampling error of the Itanium processor.

3.3.4.3 Resource Utilization Counter (RUC – AR45)

The Resource Utilization Counter (RUC) is a 64-bit counter that counts up at a fixed relationship to the input clock to the processor, when the processor is active. Processors may be inactive due to hardware multi-threading. Virtual processors may be inactive when not scheduled to run by the VMM. (See Section 11.7, “PAL Virtualization Support” on page 2:324 for details on virtual processors.)

The RUC is clocked such that, in a given time interval, the difference in the RUC values for all of the logical or virtual processors on a given physical processor add up to approximately the difference seen in the ITC on that physical processor for that same interval.

A sequence of reads of the RUC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads. Applications can directly sample the RUC for active-running-time calculations.

A 64-bit overflow condition can occur without notification. The RUC can be read at any privilege level if PSR.si is zero. The timer can be secured from non-privileged access by setting PSR.si to one. When secured, a read of the RUC by non-privileged code results in a Privileged Register fault. Writes to the RUC can only be performed at privilege level 0; otherwise, a Privileged Register fault is raised.

Modification of the RUC is not necessarily serialized with respect to instruction execution. Software can issue a data serialization operation to ensure the RUC updates are observed by a given point in program execution. Software must accept a level of sampling error when reading the resource utilization counter due to various machine stall conditions, interruptions, bus contention effects, etc. Please see the processor-specific documentation for further information on the level of sampling error of the Itanium processor.

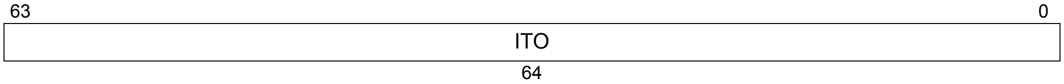
RUC should only be written by Virtual Machine Monitors; other Operating Systems should not write to RUC, but should only read it.

The RUC register is not supported on all processor implementations. Software can check CPUID register 4 to determine the availability of this feature. The RUC register is reserved when this feature is not supported.

3.3.4.4 Interval Timer Offset (ITO – CR4)

The Interval Timer Offset (ITO) register allows virtual machine monitors to specify an offset to the Interval Timer Counter (ITC) for the virtual processor. The layout of the register is shown in Figure 3-6. For details of the usage of this register in virtual environment, please refer to Section 11.7.4.1.3, "Guest MOV-from-AR.ITC Optimization" on page 2:337.

Figure 3-6. Interval Timer Offset Register (ITO – CR4)



The ITO register has no effects on instruction execution when PSR.vm is 0.

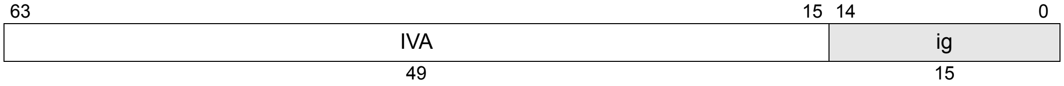
The ITO register does not affect the generation of interval timer interrupts, discussed in Section 3.3.4.2, "Interval Time Counter and Match Register (ITC – AR44 and ITM – CR1)".

The ITO register is not supported on all processor implementations. Software can call either PAL_PROC_GET_FEATURES or PAL_VP_ENV_INFO to determine the availability of this feature. The ITO register is reserved when this feature is not supported.

3.3.4.5 Interruption Vector Address (IVA – CR2)

The IVA specifies the location of the interruption vector table in the virtual address space, or the physical address space if PSR.it is 0, see Figure 3-7. The size of the vector table is 32K bytes and is 32K byte aligned. The lower 15 bits of the IVA are ignored when written, reads return zeros. All upper 49 address bits of IVA must be implemented regardless of the size of the physical and virtual address space. If an unimplemented virtual or physical address (see "Unimplemented Address Bits" on page 2:73) is loaded into IVA, and an interruption occurs, processor behavior is unpredictable. See "IVA-based Interruption Vectors" on page 2:113 for a description of an interruption table layout.

Figure 3-7. Interruption Vector Address (IVA – CR2)



3.3.4.6 Page Table Address (PTA – CR8)

The PTA anchors the Virtual Hash Page Table (VHPT) in the virtual address space. See "Virtual Hash Page Table (VHPT)" on page 2:61 for a complete definition of the VHPT. Operating systems must ensure that the table is aligned on a natural boundary; otherwise, processor operation is undefined. See Figure 3-8 and Table 3-6 for the PTA field definitions.

Figure 3-8. Page Table Address (PTA – CR8)

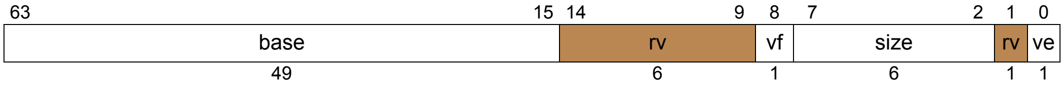


Table 3-6. Page Table Address Fields

Field	Bits	Description
ve	0	VHPT Enable – When 1, the processor is enabled to walk the VHPT.
size	7:2	VHPT Size – VHPT table size in power of 2 increments, table size is 2^{size} bytes. Size generates a mask that is logically AND'ed with the result of the VHPT hash function. Minimum VHPT table size is 32K bytes; otherwise, a Reserved Register/Field fault is raised (see "Virtual Hash Page Table (VHPT)" on page 2:61). The maximum size is 2^{61} bytes for long format VHPTs, and 2^{52} bytes for short format VHPTs.
vf	8	VHPT Format – When 0, 8-byte short format entries are used, when 1, 32-byte long format entries are used.
base	63:15	VHPT Base virtual address – Defines the starting virtual address of the VHPT table. Base is logically OR'ed with the hash index produced by the VHPT hash function when referencing the VHPT. Base must be on 2^{size} boundary otherwise processor operation is undefined. All base address bits of PTA must be implemented regardless of the size of the physical and virtual address space. If an unimplemented virtual address (see "Unimplemented Address Bits" on page 2:73) is used by the processor as a page table base, all VHPT walks generate an Instruction/Data TLB miss (see "Translation Searching" on page 2:69).
rv	1, 14:9	reserved

3.3.5 Interruption Control Registers

Registers CR16 - CR27 record information at the time of an interruption (including from the IA-32 instruction set) and are used by handlers to process the interruption.

The interruption control registers can only be read or written while PSR.ic is 0; otherwise, an Illegal Operation fault is raised. These registers are only guaranteed to retain their values when PSR.ic is 0. When PSR.ic is 1, the processor does not preserve their contents.

The contents of the interruption control registers are defined only when the PSR.ic bit is cleared by an interruption. If the PSR.ic bit is explicitly cleared (e.g., by using `rsm`, or mov to PSR), then the contents of these registers are undefined. If the PSR.ic bit is explicitly set (e.g., by using `ssm`, or mov to PSR), then the contents of these registers are undefined until the PSR.ic bit has been serialized and an interruption occurs.

IIPA has special behavior in case of an `rfi` to a fault. Refer to "Interruption Instruction Previous Address (IIPA – CR22)" on page 2:40.

3.3.5.1 Interruption Processor Status Register (IPSR – CR16)

On an interruption and if PSR.ic is 1, the IPSR receives the value of the PSR. The IPSR, IIP and IFS are used to restore processor state on a Return From Interruption (`rfi`). The IPSR has the same format as PSR, see "Processor Status Register (PSR)" on page 2:23 for details.

3.3.5.2 Interruption Status Register (ISR – CR17)

The ISR receives information related to the nature of the interruption, and is written by the processor on all interruption events regardless of the state of PSR.ic, except for Data Nested TLB faults. The ISR contains information about the excepting instruction and its properties such as whether it was doing a read, write, execute, speculative, or non-access operation, see Figure 3-9 and Table 3-7. Multiple bits may be concurrently set in the ISR, for example, a faulting semaphore operation will set both ISR.r and ISR.w, and faults on speculative loads will set ISR.sp and ISR.r. Additional fault- or trap-specific information is available in ISR.code and ISR.vector. Refer to Section 8.2, "ISR Settings" for complete definition of the ISR field settings.

Figure 3-9. Interruption Status Register (ISR – CR17)

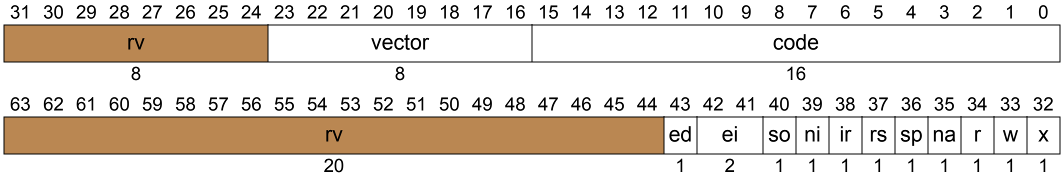


Table 3-7. Interruption Status Register Fields

Field	Bits	Description
code	15:0	Interruption Code – 16 bit code providing additional information specific to the current interruption. For IA-32 specific exceptions and software interrupts, contains the IA-32 interruption error code or zero.
vector	23:16	IA-32 exception/interception vector number. For IA-32 exceptions and software interrupts, contains the IA-32 vector number (e.g., GPFault has a vector number of 13). See Chapter 9, "IA-32 Interruption Vector Descriptions" for details.
x	32	Execute exception – Interruption is associated with an instruction fetch (including IA-32).
w	33	Write exception – Interruption is associated with a write operation. Both ISR.r and ISR.w are set for IA-32 read-modify-write instructions.
r	34	Read exception – Interruption is associated with a read operation. Both ISR.r and ISR.w are set for IA-32 read-modify-write instructions.
na	35	Non-access exception – See Section 5.5.2, "Non-access Instructions and Interruptions" on page 2:103. This bit is always 0 for interruptions taken in the IA-32 instruction set.
sp	36	Speculative load exception – Interruption is associated with a speculative load instruction. This bit is always 0 for interruptions taken in the IA-32 instruction set.
rs	37	Register Stack – Interruption is associated with a mandatory RSE fill or spill. This bit is always 0 for interruptions taken in the IA-32 instruction set.
ir	38	Incomplete Register frame – The current register frame is incomplete when the interruption occurred. This bit is always 0 for interruptions taken in the IA-32 instruction set.
ni	39	Nested Interruption – Indicates that PSR.ic was 0 or in-flight when the interruption occurred. This bit is always 0 for interruptions taken in the IA-32 instruction set.
so	40	IA-32 Supervisor Override – Indicates the fault occurred during an IA-32 instruction set supervisor override condition (the processor was performing a data memory accesses to the IDT, GDT, LDT or TSS segments) or an IA-32 data memory access at a privilege level of zero. This bit is always 0 for interruptions taken while executing Intel Itanium instructions.
ei	42:41	Excepting Instruction – 0 – exception due to instruction in slot 0 1 – exception due to instruction in slot 1 2 – exception due to instruction in slot 2 For faults and external interrupts, ISR.ei is equal to IPSR.ri. For traps, ISR.ei defines the slot of the excepting instruction. Traps on the L+X instruction of an MLX set ISR.ei to 2. This field is always 0 for interruptions taken in the IA-32 instruction set.
ed	43	Exception Deferral – this bit is set to the value of the TLB exception deferral bit (TLB.ed) for the instruction page containing the faulting instruction. If a translation does not exist or instruction translation is disabled, or if the interruption is caused by a mandatory RSE spill or fill, ISR.ed is set to 0. This bit is always 0 for interruptions taken in the IA-32 instruction set.
rv	31:24, 63:44	reserved

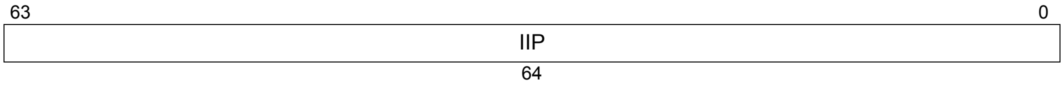
3.3.5.3 Interruption Instruction Bundle Pointer (IIP – CR19)

On an interruption and if PSR.ic is 1, the IIP receives the value of IP. IIP contains the virtual address (or physical if instruction translations are disabled) of the next instruction bundle or the IA-32 instruction to be executed upon return from the interruption. For IA-32 instruction addresses, IIP is zero extended to 64-bits and specifies a byte granular address. For traps and interrupts, IIP points to the next instruction to execute. For faults, IIP points to the faulting instruction. As shown in

Figure 3-10, all 64-bits of the IIP must be implemented regardless of the size of the physical and virtual address space supported by the processor model (see "Unimplemented Address Bits" on page 2:73). IIP also receives byte-aligned IA-32 instruction pointers. The IIP, IPSR and IFS are used to restore processor state on a Return From Interruption instruction (`rfi`). See "Interruption Vector Descriptions" on page 2:165 for usages of the IIP.

An `rfi` to Itanium architecture-based code (IPSR.is is 0) ignores IIP{3:0}, an `rfi` to IA-32 code (IPSR.is is 1) ignores IIP{63:32}. Ignored bits are assumed to be zero.

Figure 3-10. Interruption Instruction Bundle Pointer (IIP – CR19)



Control transfers to unimplemented addresses (see "Unimplemented Address Bits" on page 2:73) result in an Unimplemented Instruction Address trap or fault. When the trap or fault is delivered, IIP is written as follows:

- If the trap is taken for an unimplemented virtual address, IIP is written in one of two ways, depending on the implementation: 1) IIP may be written with the implemented virtual address bits IP{63:61} and IP{IMPL_VA_MSB:0} only. Bits IIP{60:IMPL_VA_MSB+1} are set to IP{IMPL_VA_MSB}, i.e., sign-extended. 2) IIP may be written with the full, unimplemented virtual address from IP.
- If the trap is taken for an unimplemented physical address, IIP is written in one of two ways, depending on the implementation: 1) IIP may be written with the physical addressing memory attribute bit IP{63} and the implemented physical address bits IP{IMPL_PA_MSB:0} only. Bits IIP{62:IMPL_PA_MSB+1} are set to 0. 2) IIP may be written with the full, unimplemented physical address from IP.

When an `rfi` is executed with an unimplemented address in IIP (an unimplemented virtual address if IPSR.it is 1, or an unimplemented physical address if IPSR.it is 0), and an Unimplemented Instruction Address trap is taken, an implementation may optionally leave IIP unchanged (preserving the unimplemented address in IIP).

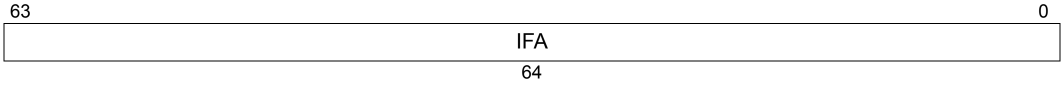
Note: Since IP{3:0} are always 0 when executing Itanium architecture-based code, IIP{3:0} will always be 0 when any interruption is taken from Itanium architecture-based code, with the exception of an Unimplemented Instruction Address trap on an `rfi`, where IIP may optionally be preserved as whatever value it held before executing the `rfi`.

3.3.5.4 Interruption Faulting Address (IFA – CR20)

On an interruption and if PSR.ic is 1, the IFA receives the virtual address (or physical address if translations are disabled) that raised a fault. IFA reports the faulting address for both instruction and data memory accesses (including IA-32). For faulting data references (including IA-32), IFA points to the first byte of the faulting data memory operand. IFA reports a byte granular address. For faulting instruction references (including IA-32), IFA contains the 16-byte aligned bundle address (IFA{3:0} are zero) of the faulting instruction. For faulting IA-32 instructions, IIP points to the first byte of the IA-32 instruction, and is byte granular. In the event of an IA-32 instruction spanning a virtual page boundary, IA-32 instruction fetch faults are reported as either (1) for faults on the first page, IFA is set to the bundle address (IFA{3:0}=0) of the faulting instruction and IIP points to the first byte of the faulting instruction, or (2) for faults on the second page, IFA contains the bundle address of the second virtual page and IIP points to the first byte of the faulting IA-32 instruction.

The IFA also specifies a translation's virtual address when a translation entry is inserted into the instruction or data TLB. See "Interruption Vector Descriptions" on page 2:165 and "Translation Insertion Format" on page 2:53 for usages of the IFA. As shown in Figure 3-11, all 64-bits of the IFA must be implemented regardless of the size of the virtual and physical space supported by the processor model (see "Unimplemented Address Bits" on page 2:73). In some implementations, a mov to IFA instruction may raise an Unimplemented Data Address fault if an unimplemented virtual address is used.

Figure 3-11. Interruption Faulting Address (IFA – CR20)



3.3.5.5 Interruption TLB Insertion Register (ITIR – CR21)

The ITIR receives default translation information from the referenced virtual region register on a virtual address translation fault. See "Interruption Vector Descriptions" on page 2:165 for the fault conditions that set the ITIR. The ITIR provides additional virtual address translation parameters on an insertion into the instruction or data TLB. See "Translation Instructions" on page 2:60 for ITIR usage information. Figure 3-12 and Table 3-8 define the ITIR fields.

Figure 3-12. Interruption TLB Insertion Register (ITIR)

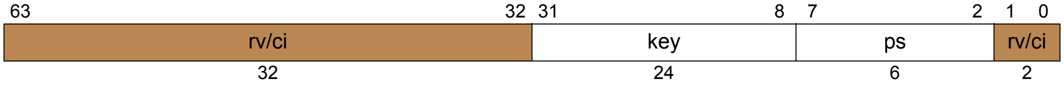


Table 3-8. ITIR Fields

Field	Bits	Description
rv/ci	63:32, 1:0	Reserved / Check on Insert – On a read these fields may return zeros or the value last written to them. If a non-zero value is written, a Reserved Register/Field fault may be raised on the mov to ITIR instruction. If not, a subsequent TLB insert will raise a Reserved Register Field fault depending on other parameters to the insert. See "Translation Insertion Format" on page 2:53. On an instruction or data translation fault, these fields are set to zero.
ps	7:2	Page Size – On a TLB insert, specifies the size of the virtual to physical address mapping. If an unsupported page size is written, a Reserved Register/Field fault may be raised on the mov to ITIR instruction. If not, a subsequent TLB insert will raise a Reserved Register/Field fault. See "Translation Insertion Format" on page 2:53. On an instruction or data translation fault, this field is set to the accessed region's page size (RR.ps).
key	31:8	Protection Key – On a TLB insert specifies a protection key that uniquely tags translations to a protection domain. If non-zero values are written to unimplemented protection key bits, a Reserved Register/Field fault may be raised on the mov to ITIR instruction. If not, a subsequent TLB insert will raise a Reserved Register/Field fault depending on other parameters to the insert. See "Translation Insertion Format" on page 2:53. On an instruction or data translation fault, this field is set to the accessed Region Identifier (RR.rid).

3.3.5.6 Interruption Instruction Previous Address (IIPA – CR22)

For Itanium instructions, IIPA records the last successfully executed instruction bundle address. For IA-32 instructions, IIPA records the byte granular virtual instruction address zero extended to 64-bits of the faulting or trapping IA-32 instruction. In the case of a fault, IIPA does not report the address of the last successfully executed IA-32 instruction, but rather the address of the faulting IA-32 instruction. IIPA preserves bits 3:0 for byte aligned IA-32 instruction addresses.

The IIPA can be used by software to locate the address of the instruction bundle or IA-32 instruction that raised a trap or the instruction executed prior to a fault or interruption. In the case of a branch related trap, IIPA points to the instruction bundle which contained the branch instruction that raised the trap, while IIP points to the target of the branch.

When an instruction successfully executes without a fault, and the PSR.ic bit was 1 prior to instruction execution, it becomes the "last successfully executed instruction." On interruptions, IIPA contains the address of the last successfully executed instruction bundle or IA-32 instruction, if PSR.ic was 1 prior to the interruption. Note that execution of an `rfi` instruction with PSR.ic equal to 0, but which sets PSR.ic to 1 does not update IIPA, since PSR.ic was zero prior to instruction execution.

When PSR.ic is one, accesses to IIPA cause an Illegal Operation fault. When PSR.ic is zero, IIPA is not updated by hardware and can be read and written by software. This permits low-level code to preserve IIPA across interruptions.

If the PSR.ic bit is explicitly cleared, e.g., by using `rsm`, then the contents of IIPA are undefined. Only when the PSR.ic bit is cleared by an interruption is the value of IIPA defined. It may point at the instruction which caused a trap, or at the instruction just prior to a faulting instruction, at an earlier instruction that became defined by some prior interruption, or by a move to IIPA instruction when PSR.ic was zero.

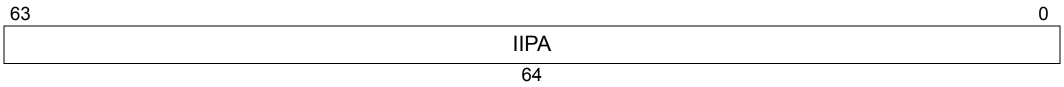
If the PSR.ic bit is explicitly set, e.g., by using `ssm`, then the contents of IIPA are undefined until the PSR.ic bit has been serialized and an interruption occurs.

During instruction set transitions the following boundary cases exist:

- On faults taken on the first IA-32 instruction after a `br.ia` or `rfi`, IIPA records the faulting IA-32 instruction address.
- On `br.ia` traps, IIPA records the address of the trapping instruction bundle.
- On faults taken on the first Itanium instruction after leaving the IA-32 instruction set, due to a `jmpe` or interruption, IIPA contains the address of the `jmpe` instruction or the interrupted IA-32 instruction.
- On `jmpe` Data Debug, Single Step and Taken Branch traps, IIPA contains the address of the `jmpe` instruction.

As shown in Figure 3-13, all 64-bits of the IIPA must be implemented regardless of the size of the physical and virtual address space supported by the processor model (see "Unimplemented Address Bits" on page 2:73).

Figure 3-13. Interruption Instruction Previous Address (IIPA – CR22)



3.3.5.7 Interruption Function State (IFS – CR23)

The IFS register is used to reload the current register stack frame (CFM) on a Return From Interruption (`rfi`). If the IFS is accessed while PSR.ic is 1, an Illegal Operation fault is raised. The IFS can only be accessed at privilege level 0; otherwise, a Privileged Operation fault is raised. The IFS.v bit is cleared on interruption if PSR.ic is 1. All other fields are undefined after an interruption. If PSR.ic is 0, the `cover` instruction copies CFM to IFS.ifm and sets IFS.v to 1. See Figure 3-14 and Table 3-9 for the IFS field definitions.

Figure 3-14. Interruption Function State (IFS – CR23)

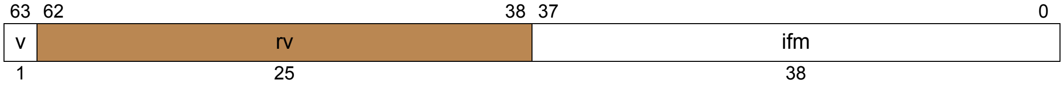


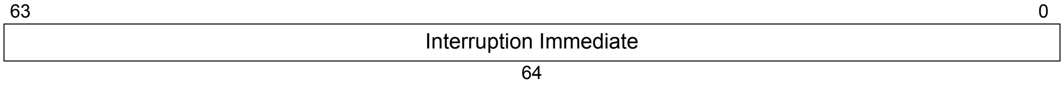
Table 3-9. Interruption Function State Fields

Field	Bits	Description
ifm	37:0	Interruption Frame Marker
v	63	Valid bit, cleared to 0 on interruption if PSR.ic is 1.
rv	62:38	reserved

3.3.5.8 Interruption Immediate (IIM – CR24)

If PSR.ic is 1, the IIM (Figure 3-15) records the zero-extended immediate field encoded in `chk.a`, `chk.s`, `fchkf` or `break` instruction faults. The `break.b` instruction always writes a zero value and ignores its immediate field. The IA_32_Intercept vector writes all 64-bits of IIM to indicate the cause of the intercept. See Table 8-1 on page 2:166 for the value of IIM in other situations. For the purpose of resource dependency, IIM is written as a result of the fault, not by the instruction itself.

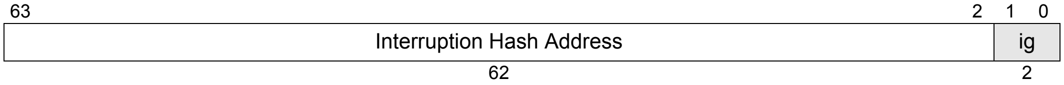
Figure 3-15. Interruption Immediate (IIM – CR24)



3.3.5.9 Interruption Hash Address (IHA – CR25)

The IHA (Figure 3-16) is loaded with the address of the Virtual Hash Page Table (VHPT) entry the processor referenced or would have referenced to resolve a translation fault. The IHA is written on interruptions by the processor when PSR.ic is 1. Refer to "VHPT Hashing" on page 2:65 for complete details. See Table 8-1 on page 2:166 for the value of IHA in other situations. All upper 62 address bits of IHA must be implemented regardless of the size of the virtual address space supported by the processor model (see "Unimplemented Address Bits" on page 2:73). The virtual address written to IHA by the processor is guaranteed to be an implemented virtual addresses on all processor models; however, if the address referenced by the VHPT is an unimplemented virtual address, the value of IHA is undefined.

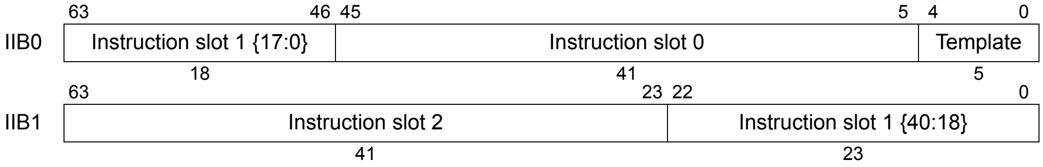
Figure 3-16. Interruption Hash Address (IHA – CR25)



3.3.5.10 Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)

On an interruption and if PSR.ic is 1, the IIB registers receive the 16-byte instruction bundle corresponding to the interruption. The bundle reported in the IIB registers is the bundle exactly as it was fetched for execution of the instruction which raised the interruption. Figure 3-17 shows the format of the IIB0 and IIB1 registers. For details on instruction bundle format, see Section 3.3, "Instruction Encoding Overview" on page 1:38.

Figure 3-17.Interruption Instruction Bundle Registers (IIB0-1, – CR26, 27)



If the interruption is a fault, the IIB registers record the instruction bundle pointed to by IIP. If the interruption is a trap, the IIB registers record the instruction bundle pointed to by IIPA.

The IIB registers only provide valid interruption bundle information on certain IVA-based faults and traps. Please refer to Table 8-1, "Writing of Interruption Resources by Vector" on page 2:166 and corresponding interruption vector pages in Section 8.3, "Interruption Vector Definition" on page 2:166 for information on which faults and traps these registers are valid. For faults and traps that indicate IIB is not valid, updates to the register may occur, but the information is undefined.

For IA-32 interruptions, instruction bundle information is not provided and the values in IIB registers are undefined.

The IIB registers are not supported on all processor implementations. Software can call PAL_PROC_GET_FEATURES to determine the availability of this feature, see "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details. The IIB registers are reserved when this feature is not supported.

3.3.6 External Interrupt Control Registers

The external interrupt control registers (CR64-81) are defined in "External Interrupt Control Registers" on page 2:121. They are used to prioritize and deliver external interrupts, send inter-processor interrupts to other processors and assign interrupt vectors for locally generated processor interrupts.

3.3.7 Banked General Registers

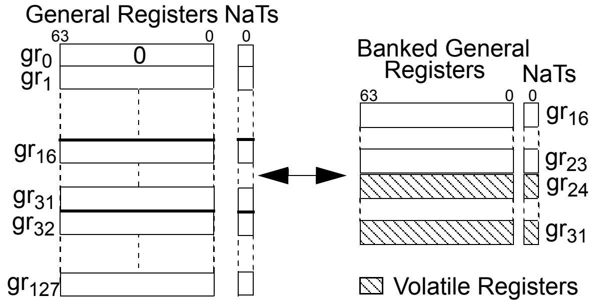
Banked general registers (see Figure 3-18) provide immediate register context for low-level interruption handlers (e.g., speculation and TLB miss handlers). Upon interruption, the processor switches 16 general purpose registers (GR16 to GR31) to register bank 0, register bank 1 contents are preserved.

When PSR.bn is 1, bank 1 for registers GR16 to GR31 is selected; when 0, bank 0 for registers GR16 to GR31 is selected. Banks are switched in the following cases:

- An interruption selects bank 0,
- `rfi` switches to the bank specified by IPSR.bn, or
- `bsw` switches to the specified bank.

On an interruption or bank switch, the processor ensures all prior register accesses (reads and writes) are performed to the prior register bank. Data values in banked registers are preserved across bank switches and both banks maintain NaT values when loaded from general registers. Registers from both banks cannot be addressed at the same time. However, non-banked general registers (GR0-15, and GR32-127) are accessible regardless of the state of PSR.bn.

Figure 3-18. Banked General Registers



Whether the ALAT register target tracking mechanism (see "Data Speculation" on page 1:63) distinguishes between the two register banks is implementation dependent; from the ALAT's perspective, GR16 in bank 0 may be the same register as GR16 in bank 1 in some implementations.

Operating systems should ensure that IA-32 and Itanium architecture-based application code is executed within register bank 1. If IA-32 or Itanium architecture-based application code executes out of register bank 0, the application register state (including IA-32) will be lost on any interruption. During interruption processing the operating system uses register bank 0 as the initial working register context.

Usage of these additional registers is determined by software conventions. However, registers GR24 to GR31, of bank 0, are not preserved when PSR.ic is 1; operating system code can not rely on register values being preserved unless PSR.ic is 0. While PSR.ic is 1, processor-specific firmware may use these registers for machine check or firmware interruption handling at any point regardless of the state of PSR.i. If PSR.ic is 0, GR24 to GR31 can be used as scratch registers for low-level interruption handlers. Registers GR16 to GR23 are always preserved; operating system code can rely on the values being preserved.

3.4 Processor Virtualization

Processors in the Itanium Processor Family may optionally implement a mechanism to support processor virtualization. This includes an additional PSR.vm bit (see Section 3.3.2, "Processor Status Register (PSR)"), which, when 1, causes certain instructions to take a Virtualization fault (see Section 5.6, "Interruption Priorities" and "Virtualization vector (0x6100)" on page 2:209).

The set of instructions which are virtualized by PSR.vm are listed in Table 3-10 below.

Table 3-10. Virtualized Instructions

Class	Virtualized Instructions
All privileged instructions	`itc.i, itc.d, itr.i, itr.d, ptc.l, ptc.g, ptc.ga, ptc.e, ptr, tak, tpa, mov rr, mov pkr, mov cr, mov ibr, mov dbr, mov pmc, mov to pmd, ssm, rsm, mov psr, rfi, bsw`
Some non-privileged instructions (virtualized at all privilege levels)	`thash, ttag, mov from cpuid, probe`[a]
Some non-privileged instructions (virtualized at privilege level 0)	`cover, probe`[a]
Reading AR[ITC] or AR[RUC] with PSR.si==1 (virtualized at all privilege levels)	`mov from ar.itc, mov from ar.ruc`
Instructions which write privileged registers	`mov to ar.itc, mov to ar.ruc`

[a] Virtualization of the `probe` instruction is configurable, see Section 11.7.4.2.8, "Probe Instruction Virtualization" on page 2:344 for details.

Processors which support processor virtualization must provide an implementation-dependent mechanism for disabling the `vmsw` instruction. When enabled, the `vmsw` instruction functions as described on the `vmsw` instruction page. When disabled, the `vmsw` instruction always raises a Virtualization fault when executed at the most privileged level.

Processors which support processor virtualization may provide an implementation-dependent mechanism to disable virtual machine features, see "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details.

Processor virtualization is largely invisible to system software, and therefore its effects on virtualized instructions are not discussed in this document, except on the instruction description pages themselves.

§

Addressing and Protection 4

This chapter defines operating system resources to translate 64-bit virtual addresses into physical addresses, 32-bit virtual addressing, virtual aliasing, physical addressing, memory ordering and properties of physical memory. Register state defined to support virtual memory management is defined in Chapter 3, while Chapter 5 provides complete information on virtual memory faults.

Note: Unless otherwise noted, references to "interruption" in this chapter refer to IVA-based interruptions. See "Interruption Definitions" on page 2:95.

The following key features are supported by the virtual memory model.

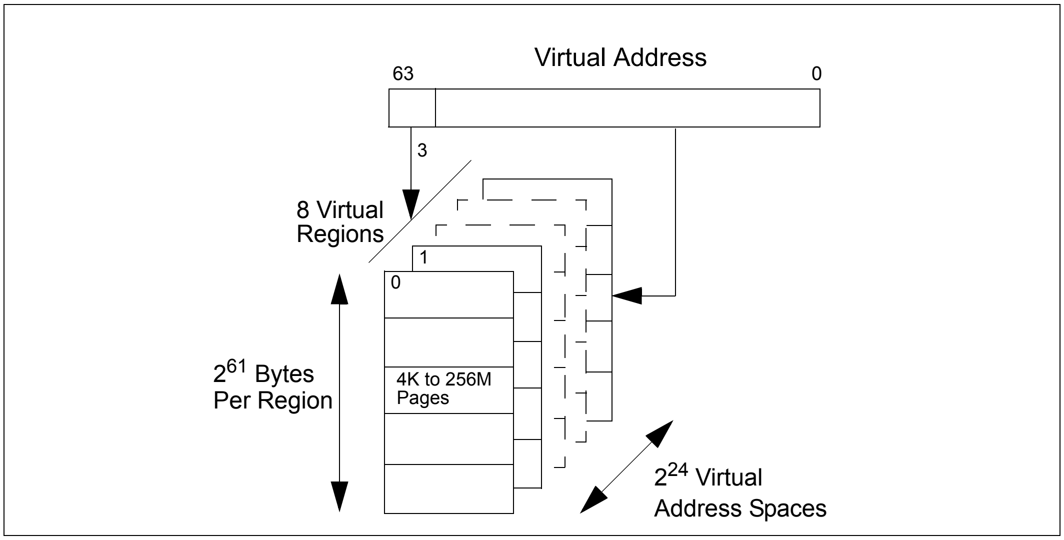
- Virtual Regions are defined to support contemporary operating system Multiple Address Space (MAS) models of placing each process within a unique address space. Region identifiers uniquely tag virtual address mappings to a given process.
- Protection Domain mechanisms support the Single Address Space (SAS) model, where processes co-exist within the same virtual address space.
- Translation Lookaside Buffer (TLB) structures are defined to support high-performance paged virtual memory systems. Software TLB fill and protection handlers are utilized to defer translation policies and protection algorithms to the operating system.
- A Virtual Hash Page Table (VHPT) is designed to augment the performance of the TLB. The VHPT is an extension of the processor's TLB that resides in memory and can be automatically searched by the processor. A particular operating system page table format is not dictated. However, the VHPT is designed to mesh with two common translation structures: the virtual linear page table and hashed page table. Enabling of the VHPT and the size of the VHPT are completely under software control.
- Sparse 64-bit virtual addressing is supported by providing for large translation arrays (including multiple levels of hierarchy similar to a cache hierarchy), efficient translation miss handling support, multiple page sizes, pinned translations, and mechanisms to promote sharing of TLB and page table resources.

4.1 Virtual Addressing

As seen by Itanium architecture-based application programs, the virtual addressing model is fundamentally a 64-bit flat linear virtual address space. 64-bit general registers are used as pointers into this address space. IA-32 32-bit virtual linear addresses are zero extended into the 64-bit virtual address space.

As shown in Figure 4-1, the 64-bit virtual address space is divided into eight 2^{61} byte virtual regions. The region is selected by the upper 3-bits of the virtual address. Associated with each virtual region is a region register that specifies a 24-bit region identifier (unique address space number) for the region. Eight out of the possible 2^{24} virtual address spaces are concurrently accessible via the 8 region registers. The region identifier can be considered the high order address bits of a large 85-bit global address space for a single address space model, or as a unique ID for a multiple address space model.

Figure 4-1. Virtual Address Spaces



By assigning sequential region identifiers, regions can be coalesced to produce larger 62-, 63- or 64-bit spaces. For example, an operating system could implement a 62-bit region for process private data, 62-bit region for I/O, and a 63-bit region for globally shared data. Default page sizes and translation policies can be assigned to each virtual region.

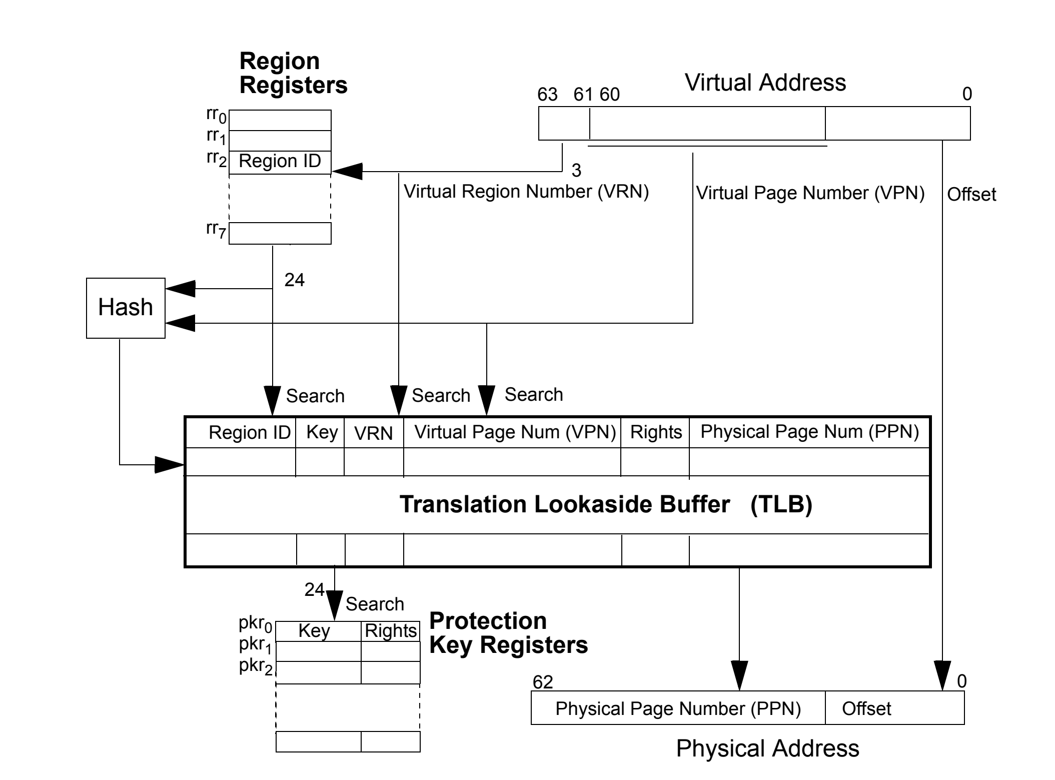
Figure 4-2 shows the process of mapping a virtual address into a physical address. Each virtual address is composed of three fields: the Virtual Region Number, the Virtual Page Number, and the page offset. The upper 3-bits select the Virtual Region Number (VRN). The least-significant bits form the page offset. The Virtual Page Number (VPN) consists of the remaining bits. The VRN bits are not included in the VPN. The page offset bits are passed through the translation process unmodified. Exact bit positions for the page offset and VPN bits vary depending on the page size used in the virtual mapping.

On a memory reference (any reference other than an insert or purge), the VRN bits select a Region Identifier (RID) from 1 of the 8 region registers, the TLB is then searched for a translation entry with a matching VPN and RID value. The VRN may optionally be used when searching for a matching translation on memory references (references other than inserts and purges – see Section 4.1.1.4, "Purge Behavior of TLB Inserts and Purges"). If a matching translation entry is found, the entry's physical page number (PPN) is concatenated with the page offset bits to form the physical address. Matching translations are qualified by page-granular privilege level access right checks and optional protection domain checks by verifying the translation's key is contained within a set of protection key registers and read, write, execute permissions are granted.

If the required translation is not resident in the TLB, the processor may optionally search the VHPT structure in memory for the required translation and install the entry into the TLB. If the required entry cannot be found in the TLB and/or VHPT, the processor raises a TLB Miss fault to request that the operating system supply the translation. After the operating system installs the translation in the TLB and/or VHPT, the faulting instruction can be restarted and execution resumed.

Virtual addressing for instruction references are enabled when PSR.it is 1, data references when PSR.dt is 1, and register stack accesses when PSR.rt is 1.

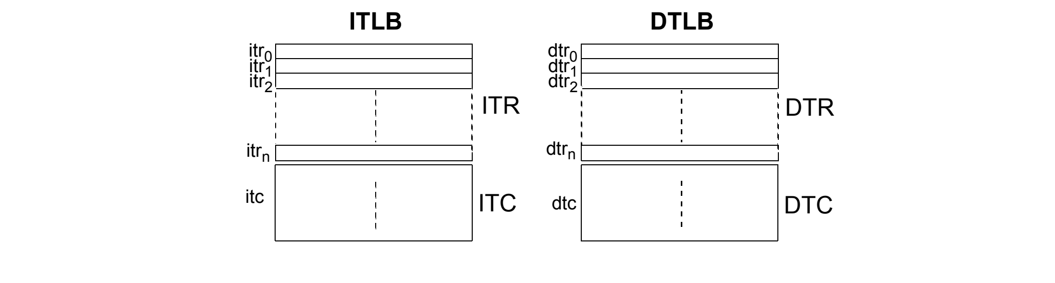
Figure 4-2. Conceptual Virtual Address Translation for References



4.1.1 Translation Lookaside Buffer (TLB)

The processor maintains two architectural TLBs as shown in Figure 4-3, the Instruction TLB (ITLB) and Data TLB (DTLB). Each TLB services translation requests for instruction and data memory references (including IA-32), respectively. The Data TLB also services translation requests for references by the RSE and the VHPT walker. The TLBs are further divided into two sub-sections; Translation Registers (TR) and Translation Cache (TC).

Figure 4-3. TLB Organization



In the remainder of this document, the term TLB refers to the combined instruction, data, translation register, and translation cache structures.

The TLB is a local processor resource; installation of a translation or local processor purges do not affect other processor's TLBs. Global TLB purges are provided to purge translations from all processors within a TLB coherence domain in a multiprocessor system.

4.1.1.1 Translation Registers (TR)

The Translation Register (TR) section of the TLB is a fully-associative array defined to hold translations that software directly manages. Software can explicitly insert a translation into a TR by specifying a register slot number. Translations are removed from the TRs by specifying a virtual address, page size and a region identifier. Translation registers allow the operating system to "pin" critical virtual memory translations in the TLB. Examples include I/O spaces, kernel memory areas, frame buffers, page tables, sensitive interruption code, etc. Instruction fetches for interruption handlers are performed using virtual addresses; therefore, virtual address ranges containing software translation miss routines and critical interruption sequences should be pinned or else additional TLB faults may occur. Other virtual mappings may be pinned for performance reasons.

Entries are placed into a specific TR slot with the Insert Translation Register (`itr`) instruction. Once a translation is inserted, the processor will not replace the translation to make room for other translations. Local translations can only be removed by software issuing the Purge Translation Register (`ptr`) instruction.

TR inserts and purges may cause other TR and/or TC entries to be removed (refer to Section 4.1.1.4, "Purge Behavior of TLB Inserts and Purges" for details). Prior to inserting a TR entry, software must ensure that no overlapping translation exists in any TR (including the one being written); otherwise, a Machine Check abort may be raised, or the processor may exhibit other undefined behavior. Translation register entries may be removed by the processor due to hardware or software errors. In the presence of an error, the processor can remove TR entries; notification is raised via a Machine Check abort.

There are at least 8 instruction and 8 data TR slots implemented on all processor models. Please see the processor-specific documentation for further information on the number of translation registers implemented on the Itanium processor. Translation registers support all implemented page sizes and must be implemented in a single-level fully-associative array. Any register slot can be used to specify any virtual address mapping. Translation registers are not directly readable.

In some processor models, translation registers are physically implemented as a subsection of the translation cache array. Valid TR slots are ignored for purposes of processor replacement on an insertion into the TC. However, invalid TR slots (unused slots) may be used as TC entries by the processor. As a result, software inserts into previously invalid TR entries may invalidate a TC entry in that slot.

Implementations may also place a floating boundary between TR and TC entries within the same structure where any entry above the boundary is considered a TC and any entry below the boundary a TR. To maximize TC resources, software should allocate contiguous translation registers starting at slot 0 and continuing upwards.

4.1.1.2 Translation Cache (TC)

The Translation Cache (TC) is an implementation-specific structure defined to hold the large working set of dynamic translations for memory references (including IA-32). Please see the processor-specific documentation for further information on Itanium processor TC implementation details. The processor directly controls the replacement policy of all TC entries.

Entries are installed by software into the translation cache with the Insert Data Translation Cache (`itc.d`) and Insert Instruction Translation Cache (`itc.i`) instructions. The Purge Translation Cache Local (`ptc.l`) instruction purges all ITC/DTC entries in the local processor that match the specified virtual address range and region identifier. Purges of all ITC/DTC entries matching a specified virtual address range and region identifier among all processors in a TLB coherence domain can be globally performed with the Purge Translation Cache Global (`ptc.g`, `ptc.ga`) instruction. The TLB coherence domain covers at least the processors on the same local bus on which the purge was broadcast. Propagation between multiple TLB coherence domains is platform dependent. Software must handle the case where a purge does not propagate to all processors in a multiprocessor system. Translation cache purges do not invalidate TR entries.

All the entries in a local processor's ITC and DTC can be purged of all entries with a sequence of Purge Translation Cache Entry (`ptc.e`) instructions. A `ptc.e` does not propagate to other processors.

In all processor models, the translation cache has at least 1 instruction and 1 data entry in addition to the specified 8 instruction and 8 data translation registers. Implementations are free to implement translation cache arrays of larger sizes. Implementations may also choose to implement additional hierarchies for increased performance. At least one translation cache level is required to support all implemented page sizes. Additional hierarchy levels may or may not be performance optimized for the preferred page size specified by the virtual region, may be set-associative or fully associative, and may support a limited set of page sizes. Please see the processor-specific documentation for further information on the Itanium processor implementation details of the translation cache.

The translation cache is managed by both software and hardware. In general, software cannot assume any entry installed will remain, nor assume the lifetime of any entry since replacement algorithms are implementation specific. The processor may discard or replace a translation at any point in time for any reason (subject to the forward progress rules below). TC purges may remove more entries than explicitly requested. In the presence of a processor hardware error, the processor may remove TC entries and optionally raise a Corrected Machine Check Interrupt.

In order to ensure forward progress for Itanium architecture-based code, the following rules must be observed by the processor and software.

- Software may insert multiple translation cache entries per TLB fault, provided that only the last installed translation is required for forward progress.
- The processor may occasionally invalidate the last TC entry inserted. The processor must eventually guarantee visibility of the last inserted TC entry to all references while PSR.ic is zero. The processor must eventually guarantee visibility of the last inserted TC entry until an `rfi` sets PSR.ic to 1 and at least one instruction is executed with PSR.ic equal to 1, and completes without a fault or interrupt. The last

inserted TC entry may be occasionally removed before this point, and software must be prepared to re-insert the TC entry on a subsequent fault. For example, eager or mandatory RSE activity, speculative VHPT walks, or other interruptions of the restart instruction may displace the software-inserted TC entry, but when software later re-inserts the same TC entry, the processor must eventually complete the restart instruction to ensure forward progress, even if that restart instruction takes other faults which must be handled before it can complete. If PSR.ic is set to 1 by instructions other than `rfi`, the processor does not guarantee forward progress.

- If software inserts an entry into the TLB with an overlapping entry (same or larger size) in the VHPT, and if the VHPT walker is enabled, forward progress is not guaranteed. See "VHPT Searching" on page 2:62.
- Software may only make references to memory with physical addresses or with virtual addresses which are mapped with TRs, or to addresses mapped by the just-inserted translation, between the insertion of a TC entry, and the execution of the instruction with PSR.ic equal to 1 which is dependent on that entry for forward progress. Software may also make repeated attempts to execute the same instruction with PSR.ic equal to 1. If software makes any other memory references than these, the processor does not guarantee forward progress.
- Software must not defeat forward progress by consistently displacing a required TC entry through a global or local translation cache purge.

IA-32 code has more stringent forward progress rules that must be observed by the processor and software. IA-32 forward progress rules are defined in Section 10.6.3, "IA-32 TLB Forward Progress Requirements" on page 2:261.

The translation cache can be used to cache TR entries if the TC maintains the instruction vs. data distinction that is required of the TRs. A data reference cannot be satisfied by a TC entry that is a cache of an instruction TR entry, nor can an instruction reference be satisfied by a TC entry that is a cache of a data TR entry. This approach can be useful in a multi-level TLB implementation.

4.1.1.3 Unified Translation Lookaside Buffers

Some processor models may merge the ITC and DTC into a unified translation cache. The minimum number of unified entries is 2 (1 for instruction, and 1 for data). Processors may service instruction fetch memory references with TC entries originally installed into the DTC and service data memory references with translations originally installed in the ITC. To ensure consistent operation across processor implementations, software is recommended to not install different translations into the ITC or DTC for the same virtual region and virtual address. ITC inserts may remove DTC entries. DTC inserts may remove ITC entries. TC purges remove ITC and DTC entries.

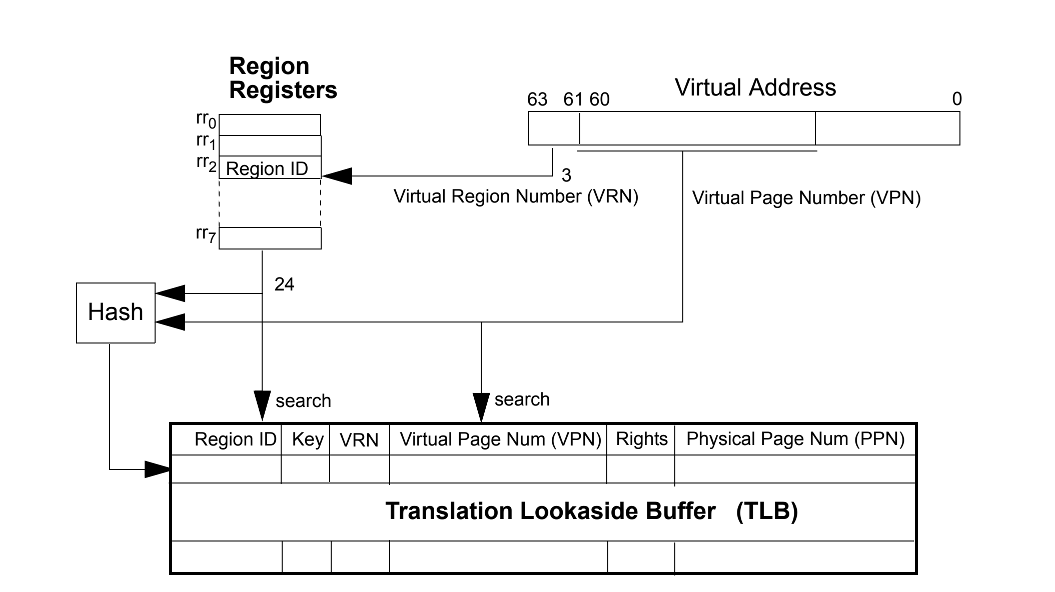
Instruction and data translation registers cannot be unified. DTR entries cannot be used by instruction references and ITR entries cannot be used by data references. ITR inserts and purges do not remove DTR entries. DTR inserts and purges do not remove ITR entries.

4.1.1.4 Purge Behavior of TLB Inserts and Purges

Translations contained in the translation caches (TC) and translation registers (TR) are maintained in a consistent state by ensuring that TLB insertions remove existing overlapping entries before new TR or TC entries are installed. Similarly, TLB purges that partially or fully overlap with existing translations may remove all overlapping entries. In this context, “overlap” refers to two translations with the same region identifier (but not necessarily identical virtual region numbers), and with partially or fully overlapping virtual address ranges (determined by the virtual address and the page size). Examples are: two 4K-byte pages at the same virtual address, or an 8K-byte page at virtual address 0x2000 and a 4K-byte page at 0x3000.

As described in Section 4.1, “Virtual Addressing” on page 2:45, each TLB may contain a VRN field, and virtual address bits {63:61} may be used as part of the match for memory references (references other than inserts and purges). This binding of a translation to the VRN implies that a lookup of a given virtual address (region identifier/VPN pair) in either the translation cache or translation registers may result in a TLB miss if a memory reference is made through a different VRN (even if the region identifiers in the two region registers are identical). Some processor models may also omit the VRN field of the TLB, causing the TLB search on memory references to find an entry independent of VRN bits. However, all processor models are required, during translation cache purge and insert operations, to purge all possible translations matching the region identifier and virtual address regardless of the specified VRN.

Figure 4-4. Conceptual Virtual Address Searching for Inserts and Purges



A processor may overpurge translation cache entries; i.e., it may purge a larger virtual address range than required by the overlap. Since page sizes are powers of 2 in size and aligned on that same power of 2 boundary, purged entries can either be a superset of, identical to, or a subset of the specified purge range.

Table 4-1 define the purge behavior of different TLB insert and purge instructions. Table 4-2 describes the purge behavior for VHPT inserts.

Note: Please refer to Table 4-1 for footnotes in Table 4-2.

Table 4-1. Purge Behavior of TLB Inserts and Purges

Case	Insert?	Purge?	Machine Check?
`it[cr].[id]` overlaps [ID]TC[a]	Must[b]	Must[c]	Must not[d]
`it[cr].[id]` overlaps [DI]TC[e]	Must	May[f]	Must not
`it[cr].[id]` overlaps [ID]TR	May[g]	May	Must[h]
`it[cr].[id]` overlaps [DI]TR	Must	Must not[i]	Must not
`ptc.l` overlaps [ID]TC	N/A	Must	Must not
`ptc.l` overlaps [ID]TR		May	Must
`ptc.g` (local) overlaps [ID]TC[j]		Must	Must not
`ptc.g` (local) overlaps [ID]TR		May	Must
`ptc.g` (remote) overlaps [ID]TC		Must	Must not
`ptc.g` (remote) overlaps [ID]TR		Must not	Must not
`ptc.e` overlaps [ID]TC		Must	Must not
`ptc.e` overlaps [ID]TR		Must not	Must not
`ptr.[id]` overlaps [ID]TC		Must	Must not
`ptr.[id]` overlaps [DI]TC		May	Must not
`ptr.[id]` overlaps [ID]TR		Must	Must not
`ptr.[id]` overlaps [DI]TR		Must not	Must not

a. Bracketed notation is intended to specify TC and TR overlaps in the same stream, e.g. `itc.i` and ITC.

b. Must Insert: requires that the translation specified by the operation is inserted into a TC or TR as appropriate. For `itc` and VHPT walker inserts, there is no guarantee to software that the entry will exist in the future, with the exception of the relevant forward-progress requirements specified in Section 4.1.1.2, "Translation Cache (TC)".

c. Must Purge: requires that all partially or fully overlapped translations are removed prior to the insert or purge operation.

d. Must not Machine Check: indicates that a processor does not cause a Machine Check abort as a result of the operation.

e. Bracketed notation is intended to specify TC and TR overlaps in the opposite stream, e.g. `itc.i` and DTC.

f. May Purge: indicates that a processor may remove partially or fully overlapped translations prior to the insert or purge operation. However, software must not rely on the purge.

g. May Insert: indicates that the translation specified by the operation may be inserted into a TC. However, software must not rely on the insert.

h. Must Machine Check: indicates that a processor will cause a Machine Check abort if an attempt is made to insert or purge a partially or fully overlapped translation. The Machine Check abort may not be delivered synchronously with the TLB insert or purge operation itself, but is guaranteed to be delivered, at the latest, on a subsequent instruction serialization operation.

i. Must not Purge: the processor does not remove (or check for) partially or fully overlapped translations prior to the insert or purge operation. Software can rely on this behavior.

j. `ptc.g` (and `ptc.ga`): two forms of global TLB purges are distinguished: local and remote. The local form indicates that the `ptc.g` or `ptc.ga` was initiated on the local processor. The remote form indicates that this is an incoming TLB shoot-down from a remote processor.

Table 4-2. Purge behavior of VHPT Inserts

Case	VRN bits used for TLB searching on VHPT insert						VRN bits not used for TLB searching on VHPT insert		
	VRN Match			No VRN Match					
	Insert?	Purge?	Machine Check?	Insert?	Purge?	Machine Check?	Insert?	Purge?	Machine Check?
[ID]VHPT overlaps [ID]TC[a]	Must[b]	Must[c]	Must not[d]	Must	May	Must not	Must	Must	Must not
[ID]VHPT overlaps [DI]TC[e]	Must	May[f]	Must not	Must	May	Must not	Must	May	Must not
[ID]VHPT overlaps [ID]TR	May[g]	May	Must[h]	May	Must not[i]	May	May	Must not	Must
[ID]VHPT overlaps [DI]TR	Must	Must not	Must not	Must	Must not	Must not	Must	Must not	Must not

The VHPT walker's inserts into the TC follow purge-before-insert rules similar to those for software inserts. VHPT walker inserts into the DTC behave similar to `itc.d`; VHPT walker inserts into the ITC behave similar to `itc.i`. If an instruction reference results in a VHPT walk that misses in the data TLB, the DTC insert for the translation for the VHPT acts similar to an `itc.d`.

As described in Section 4.1, "Virtual Addressing" on page 2:45, processors may optionally use VRN bits when searching for a matching translation for a memory reference (references other than inserts and purges). In processors which do use VRN bits for such searches, VHPT inserts optionally may also use VRN bits in searching for overlapping entries. Thus, if a VHPT insertion overlaps a translation in the TC, but the VRN of the address being inserted does not match the VRN of the existing TC translation, the purge of the existing TC entry is optional. If a VHPT insertion overlaps a translation in a TR, but the VRN of the address being inserted does not match the VRN of the TR translation, the VHPT insertion is allowed, and a machine check is optional. In processors which do not use VRN bits when searching for a matching translation for a memory reference, the behavior of VHPT inserts is identical to that of software inserts (see Table 4-1, "Purge Behavior of TLB Inserts and Purges" on page 2:52).

If a VHPT insert overlaps with an existing TR entry and the VRN of the insertion matches the VRN of the existing TR entry (for example, if the translation being inserted is for a large page which overlaps with a small page translation in the TR), the VHPT insertion can be done, but a machine check must be raised. Software must not create overlapping translations in the VHPT that are larger than a currently existing TR translation. The behavior of VHPT inserts is summarized in Table 4-2.

4.1.1.5 Translation Insertion Format

Figure 4-5 shows the register interface to insert entries into the TLB. TLB insertions are performed by issuing the Insert Translation Cache (`itc.d`, `itc.i`) and Insert Translation Registers (`itr.d`, `itr.i`) instructions. The first 64-bit field containing the physical address, attributes and permissions is supplied by a general purpose register operand. Additional protection key and page size information is supplied by the Interruption TLB Insertion Register (ITIR). The Interruption Faulting Address register (IFA) specifies the virtual address for instruction and data TLB inserts. ITIR and IFA are defined in "Control Registers" on page 2:29. The upper 3 bits of IFA (VRN bits{63:61}) select a virtual region register that supplies the RID field for the TLB entry. The RID of the selected region is tagged to the translation as it is inserted into the TLB.

Reserved fields or encodings are checked as follows:

- The GR[r] value is checked when a TLB insert instruction is executed, and if reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the TLB insert instruction. If GR[r]{0} is zero (not-present Translation Insertion Format), the rest of GR[r] is ignored.
- The RR[vrn] value is checked when a mov to RR instruction is executed, and if reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the mov to RR instruction.
- The ITIR value is checked either when a mov to ITIR instruction is executed, or when a TLB insert instruction is executed, depending on the processor implementation. If reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the mov to ITIR or TLB insert instruction. In implementations where ITIR is checked on a TLB insert instruction, ITIR{63:32} and ITIR{31:8} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
- The IFA value is checked either when a mov to IFA instruction is executed, or when a TLB insert instruction is executed, depending on the processor implementation. If an unimplemented virtual address is used, an Unimplemented Data Address fault is raised on the mov to IFA or TLB insert instruction.

Software must issue an instruction serialization operation to ensure installs into the ITLB are observed by dependent instruction fetches and a data serialization operation to ensure installs into the DTLB are observed by dependent memory data references.

Figure 4-5. Translation Insertion Format

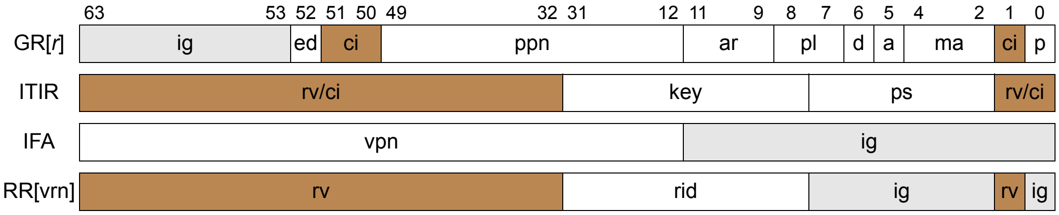


Table 4-3 describes all the translation interface fields.

Table 4-3. Translation Interface Fields

TLB Field	Source Field	Description
ci	GR[*r*]{1,51:50}	Checked on Insert – Checked on a TLB insert instruction. If reserved fields or encodings are used, a Reserved Register/Field fault is raised on the TLB insert instruction.
rv/ci	ITIR{1:0,63:32}	Reserved/Checked on Insert – Depending on implementation, may be reserved (checked on a mov to ITIR instruction) or checked on a TLB insert instruction. If reserved fields or encodings are used, a Reserved Register/Field fault is raised on the mov to ITIR or TLB insert instruction. In implementations where ITIR is checked on a TLB insert instruction, ITIR{63:32} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
rv	RR[vrn]{1,63:32}	Reserved – Checked on a mov to RR instruction. If reserved fields or encodings are used, a Reserved Register/Field fault is raised on the mov to RR instruction.

Table 4-3. Translation Interface Fields (Continued)

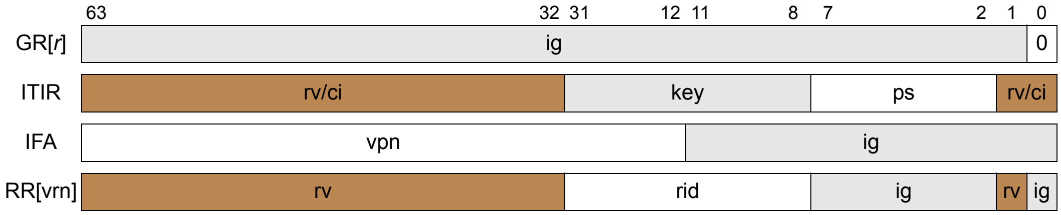
TLB Field	Source Field	Description
p	GR[*r*]{0}	Present bit – When 0, references using this translation cause an Instruction or Data Page Not Present fault. Most other fields are ignored by the processor, see Figure 4-6 for details. This bit is typically used to indicate that the mapped physical page is not resident in physical memory. The present bit is not a valid bit. For each TLB entry, the processor maintains an additional hidden valid bit indicating if the entry is enabled for matching.
ma	GR[*r*]{4:2}	Memory Attribute – describes the cacheability, coherency, write-policy and speculative attributes of the mapped physical page. See "Memory Attributes" on page 2:75 for details.
a	GR[*r*]{5}	Accessed Bit – When 0 and PSR.da is 0, data references to the page cause a Data Access Bit fault. When 0 and PSR.ia is 0, instruction references to the page cause an Instruction Access Bit fault. When 0, IA-32 references to the page cause an Instruction or Data Access Bit fault. This bit can trigger a fault on reference for tracing or debugging purposes. The processor does not update the Accessed bit on a reference.
d	GR[*r*]{6}	Dirty Bit – When 0 and PSR.da is 0, Intel Itanium store or semaphore references to the page cause a Data Dirty Bit fault. When 0, IA-32 store or semaphore references to the page cause a Data Dirty Bit fault. The processor does not update the Dirty bit on a store or semaphore reference.
pl	GR[*r*]{8:7}	Privilege Level – Specifies the privilege level or promotion level of the page. See "Page Access Rights" on page 2:56 for complete details.
ar	GR[*r*]{11:9}	Access Rights – page granular read, write and execute permissions and privilege controls. See "Page Access Rights" on page 2:56 for details.
ppn	GR[*r*]{49:12}	Physical Page Number – Most significant bits of the mapped physical address. Depending on the page size used in the mapping, some of the least significant PPN bits are ignored.

ig	GR[*r*]{63:53}
	IFA{11:0},
	RR[vrn]{0,7:2}
		available – Software can use these fields for operating system defined parameters. These bits are ignored when inserted into the TLB by the processor.

ed	GR[*r*]{52}	Exception Deferral – For a speculative load that results in an exception, the speculative load's instruction page TLB.ed bit is one of the conditions which determines whether the exception must be deferred. See "Deferral of Speculative Load Faults" on page 2:105 for complete details. This bit is ignored in the data TLB for data memory references and for IA-32 memory references.
ps	ITIR{7:2}	Page Size – Page size of the mapping. For page sizes larger than 4K bytes the low-order bits of PPN and VPN are ignored. Page sizes are defined as 2^{ps} bytes. See "Page Sizes" on page 2:57 for a list of supported page sizes.
key	ITIR{31:8}	Protection Key – Uniquely tags the translation to a protection domain. If a translation's Key is not found in the Protection Key Registers (PKRs), access is denied and a Data or Instruction Key Miss fault is raised. See "Protection Keys" on page 2:59 for complete details. In implementations where ITIR is checked on a TLB insert instruction, ITIR{31:8} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
vpn	IFA{63:12}	Virtual Page Number – Depending on a translation's page size, some of the least-significant VPN bits specified are ignored in the translation process. VPN{63:61} (VRN) selects the region register.
rid	RR[VRN].rid	Virtual Region Identifier – On TLB inserts the Region Identifier selected by VPN{63:61} (VRN) is used as additional match bits for subsequent accesses and purges (much like vpn bits).

The format in Figure 4-6 is defined for not-present translations (P-bit is zero).

Figure 4-6. Translation Insertion Format – Not Present



4.1.1.6 Page Access Rights

Page granular access controls use 4 levels of privilege. Privilege level 0 is the most privileged and has access to all privileged instructions; privilege level 3 is least privileged. Access (including IA-32) to a page is determined by the TLB.ar and TLB.pl fields, and by the privilege level of the access, as defined in Table 4-4. RSE fills and spills obtain their privilege level from RSC.pl; all other accesses (including IA-32) obtain their privilege level from PSR.cpl. Within each cell, "–" means no access, "R" means read access, "W" means write access, "X" means execute access, and "Pn" means promote PSR.cpl to privilege level "n" when an Enter Privileged Code (`epc`) instruction is executed.

Table 4-4. Page Access Rights

TLB.ar	TLB.pl	Privilege Level[a]				Description
		3	2	1	0	
0	3	R	R	R	R	read only
	2	–	R	R	R	
	1	–	–	R	R	
	0	–	–	–	R	
1	3	RX	RX	RX	RX	read, execute
	2	–	RX	RX	RX	
	1	–	–	RX	RX	
	0	–	–	–	RX	
2	3	RW	RW	RW	RW	read, write
	2	–	RW	RW	RW	
	1	–	–	RW	RW	
	0	–	–	–	RW	
3	3	RWX	RWX	RWX	RWX	read, write, execute
	2	–	RWX	RWX	RWX	
	1	–	–	RWX	RWX	
	0	–	–	–	RWX	
4	3	R	RW	RW	RW	read only / read, write
	2	–	R	RW	RW	
	1	–	–	R	RW	
	0	–	–	–	RW	
5	3	RX	RX	RX	RWX	read, execute / read, write, exec
	2	–	RX	RX	RWX	
	1	–	–	RX	RWX	
	0	–	–	–	RWX	

Table 4-4. Page Access Rights (Continued)

TLB.ar	TLB.pl	Privilege Level[a]				Description
		3	2	1	0	
6	3	RWX	RW	RW	RW	read, write, execute / read, write
	2	–	RWX	RW	RW	
	1	–	–	RWX	RW	
	0	–	–	–	RW	
7	3	X	X	X	RX	exec, promote[b] / read, execute
	2	XP2	X	X	RX	
	1	XP1	XP1	X	RX	
	0	XP0	XP0	XP0	RX	

[a] RSC.pl, for RSE fills and spills; PSR.cpl for all other accesses.
[b] User execute only pages can be enforced by setting PL to 3.

Software can verify page level permissions by the `probe` (regular_form `probe` or `probe.fault`) instruction, which checks accessibility to a given virtual page by verifying privilege levels, page level read and write permission, and protection key read and write permission.

Execute-only pages (TLB.ar 7) can be used to promote the privilege level on entry into the operating system. User level code would typically branch into a promotion page (controlled by the operating system) and execute the Enter Privileged Code (`epc`) instruction. When `epc` successfully promotes, the next instruction group is executed at the target privilege level specified by the promotion page. A procedure return branch type (`br.ret`) can demote the current privilege level.

4.1.1.7 Page Sizes

A range of page sizes are supported to assist software in mapping system resources and improve TLB/VHPT utilization. Typically, operating systems will select a small range of fixed page sizes to implement virtual memory algorithms. Larger pages may be statically allocated. For example, large areas of the virtual address space may be reserved for operating system kernels, frame buffers, or memory-mapped I/O regions. Software may also elect to pin these translations, by placing them in the translation registers.

Table 4-5 lists insertable and purgeable page sizes that are supported by all processor models. Insertable page sizes can be specified in the translation cache, the translation registers, the region registers and the VHPT. Insertable page sizes can also be used as parameters to TLB purge instructions (`ptc.l`, `ptc.g`, `ptc.ga` or `ptr`). Page sizes that are purgeable only may only be used as parameters to TLB purge instructions.

Processors may also support additional insertable and purgeable page sizes. Please see the processor-specific documentation for further information on the page sizes supported by the Itanium processor.

Table 4-5. Architected Page Sizes

	Page Sizes										
	4k	8k	16k	64k	256k	1M	4M	16M	64M	256M	4G
Insertable	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	-
Purgeable	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes

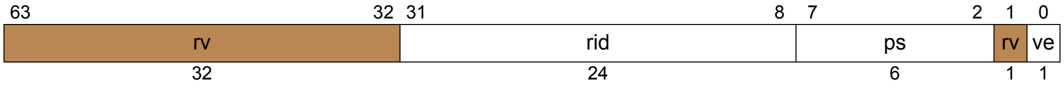
Page sizes are encoded in translation entries and region registers as a 6-bit encoded page size field. Each field specifies a mapping size of 2^N bytes, thus a value of 12 represents a 4K-byte page. If unimplemented page sizes are specified to an `itc`, `itr` or `mov` to region register instruction, a Reserved Register/Field fault is raised. If unimplemented page sizes are specified for a TLB purge instruction an implementation may raise a Machine Check abort, may under-purge translations up to ignoring the request, or may over-purge translations up to removal of all entries from the translation cache. If unimplemented page sizes are specified by a `ptc.g` or `ptc.ga` broadcast from another processor, an implementation may under-purge translations up to ignoring the request, or may over-purge translations up to removal of all entries from the translation cache. However, it must not raise a Machine Check abort.

Virtual and physical pages are aligned on the natural boundary of the page. For example, 4K-byte pages are aligned on 4K-byte boundaries, and 4 M-byte pages on 4 M-byte boundaries.

4.1.2 Region Registers (RR)

Associated with each of the 8 virtual regions is a privileged Region Register (RR). Each register contains a Region Identifier (RID) along with several other region attributes, see Figure 4-7. The values placed in the region register by the operating system can be viewed as a collection of process address space identifiers.

Figure 4-7. Region Register Format



Regions support multiple address space operating systems by avoiding the need to flush the TLB on a context switch. Sharing between processes is promoted by mapping common global or shared region identifiers into the region register working set of multiple processes. All IA-32 memory references are through region register 0.

Table 4-6 describes the region register fields. Region Identifier (rid) bits 0 through 17 must be implemented on all processor models. Some processor models may implement additional bits. Additional implemented bits must be contiguous and start at bit 18. Unimplemented bits are reserved. Please see the processor-specific documentation for further information on the size of the Region Identifier implemented on the Itanium processor.

Table 4-6. Region Register Fields

Field	Bits	Description
rv	1,63:32	reserved
ve	0	VHPT Walker Enable – When 1, the VHPT walker is enabled for the region. When 0, disabled.

Table 4-6. Region Register Fields (Continued)

Field	Bits	Description
ps	7:2	Preferred page Size – Selects the virtual address bits used in hash functions for set-associative TLBs or the VHPT. Encoded as 2^{ps} bytes. The processor may make significant performance optimizations for the specified preferred page size for the region.[a]
rid	31:8	Region Identifier – During TLB inserts, the region identifier from the select region register is used to tag translations to a specific address space. During TLB/VHPT lookups, the region identifier is used to match translations and to distribute hash indexes among VHPT and TLB sets.

[a] For more details on the usage of this field, See “VHPT Hashing” on page 2:65.

Software must issue an instruction serialization operation to ensure writes into the region registers are observed by dependent instruction fetches and issue a data serialization operation for dependent memory data references.

4.1.3 Protection Keys

Protection Keys provide a method to restrict permission by tagging each virtual page with a unique protection domain identifier. The Protection Key Registers (PKR) represent a register cache of all protection keys required by a process. The operating system is responsible for management and replacement polices of the protection key cache. Before a memory access (including IA-32) is permitted, the processor compares a translation’s key value against all keys contained in the PKRs. If a matching key is not found, the processor raises a Key Miss fault. If a matching Key is found, access to the page is qualified by additional read, write and execute protection checks specified by the matching protection key register. If these checks fail, a Key Permission fault is raised. Upon receipt of a Key Miss or Key Permission fault, software can implement the desired security policy for the protection domain. Figure 4-8 and Table 4-7 describe the protection key register format and protection key register fields.

Figure 4-8. Protection Key Register Format

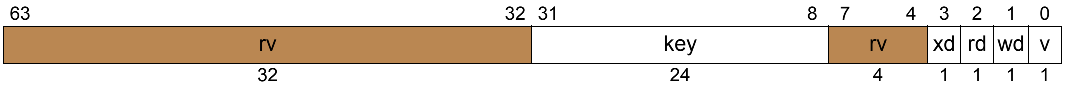


Table 4-7. Protection Register Fields

Field	Bits	Description
v	0	Valid – When 1, the Protection Register entry is valid and is checked by the processor when performing protection checks. When 0, the entry is ignored.
wd	1	Write Disable – When 1, write permission is denied to translations in the protection domain.
rd	2	Read Disable – When 1, read permission is denied to translations in the protection domain.
xd	3	Execute Disable – When 1, execute permission is denied to translations in the protection domain.
key	31:8	Protection Key – uniquely tags translation to a given protection domain.
rv	7:4,63:32	reserved

Processor models have at least 16 protection key registers, and at least 18-bits of protection key. Some processor models may implement additional protection key registers and protection key bits. Unimplemented bits and registers are reserved. Key registers have at least as many implemented key bits as region registers have rid bits. Additional implemented bits must be contiguous and start at bit 18. Please see the processor-specific documentation for further information on the number of protection key registers and protection key bits implemented on the Itanium processor.

Software must issue an instruction serialization operation to ensure writes into the protection key registers are observed by dependent instruction fetches and a data serialization operation for dependent memory data references.

The processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers during the move to PKR instruction. If a valid matching key is found in any PKR register, the processor invalidates the matching PKR register by setting PKR.v to zero, before performing the write of the new PKR register. The other fields in any matching PKR remain unchanged when it is invalidated.

Key Miss and Permission faults are only raised when memory translations are enabled (PSR.dt is 1 for data references, PSR.it is 1 for instruction references, PSR.rt is 1 for register stack references), and protection key checking is enabled (PSR.pk is one).

Data TLB protection keys can be acquired with the Translation Access Key (`tak`) instruction. Instruction TLB key values are not directly readable. To acquire instruction key values software should make provisions to read memory structures.

4.1.4 Translation Instructions

Table 4-8 lists translation instructions used to manage translations. Region registers, protection key registers and the TLBs are accessed indirectly; the register number is determined by the contents of a general register.

The processor does not ensure that modification of the translation resources is observed by subsequent instruction fetches or data memory references. Software must issue an instruction serialization operation before any dependent instruction fetch and a data serialization operation before any dependent data memory reference.

Table 4-8. Translation Instructions

Mnemonic	Description	Operation	Instr. Type	Serialization Requirement
`mov rr[`r_3`] = `r_2	Move to region register	RR[GR[r_3]] = GR[r_2]	M	data/inst
`mov `r_1` = rr[`r_3`]`	Move from region register	GR[r_1] = RR[GR[r_3]]	M	none
`mov pkr[`r_3`] = `r_2	Move to protection key register	PKR[GR[r_3]] = GR[r_2]	M	data/inst
`mov `r_1` = pkr[`r_3`]`	Move from protection key register	GR[r_1] = PKR[GR[r_3]]	M	none
`itc.i `r_3	Insert instruction translation cache	ITC = GR[r_3], IFA, ITIR	M	inst

Table 4-8. Translation Instructions (Continued)

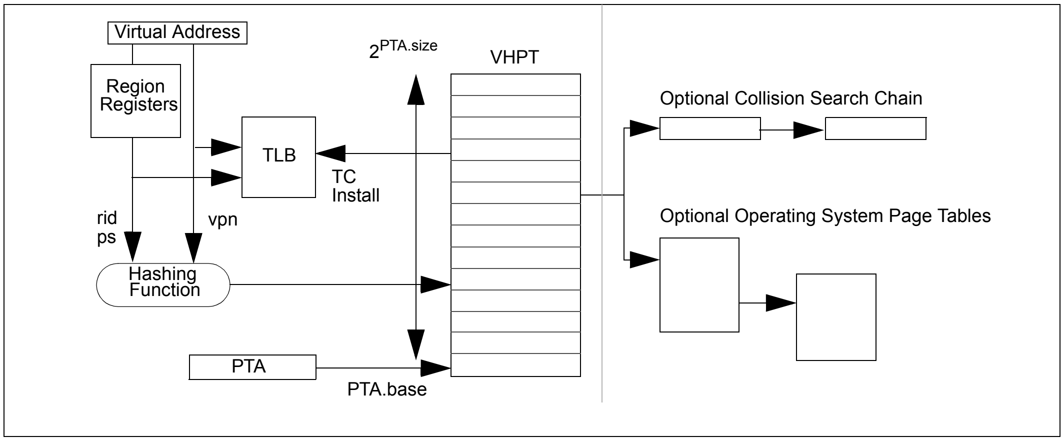
Mnemonic	Description	Operation	Instr. Type	Serialization Requirement
itc.d r_3	Insert data translation cache	DTC = GR[r_3], IFA, ITIR	M	data
itr.i itr[r_2] = r_3	Insert instruction translation register	ITR[GR[r_2]] = GR[r_3], IFA, ITIR	M	inst
itr.d dtr[r_2] = r_3	Insert data translation register	DTR[GR[r_2]] = GR[r_3], IFA, ITIR	M	data
probe r_1 = r_3, r_2	Probe data TLB for translation		M	none
probe.fault r_3, imm_2	Probe data TLB for translation		M	none
ptc.l r_3, r_2	Purge a translation from local processor instruction and data translation cache		M	data/inst
ptc.g r_3, r_2	Globally purge a translation from multiple processor's instruction and data translation caches		M	data/inst
ptc.ga r_3, r_2	Globally purge a translation from multiple processor's instruction and data translation caches and remove matching entries from multiple processor's ALATs		M	data/inst
ptc.e r_3	Purge local instruction and data translation cache of all entries		M	data/inst
ptr.i r_3, r_2	Purge instruction translation registers		M	inst
ptr.d r_3, r_2	Purge data translation registers		M	data
tak r_1 = r_3	Obtain data TLB entry protection key		M	none
thash r_1 = r_3	Generate translation's VHPT hash address		M	none
ttag r_1 = r_3	Generate translation tag for VHPT		M	none
tpa r_1 = r_3	Translate a virtual address to a physical address		M	none

4.1.5 Virtual Hash Page Table (VHPT)

The VHPT is an extension of the TLB hierarchy designed to enhance virtual address translation performance. The processor's VHPT walker can optionally be configured to search the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker provides significant performance enhancements by reducing the rate of flushing the processor's pipelines due to a TLB Miss fault, and by providing speculative translation fills concurrent to other processor operations.

The VHPT, resides in the virtual memory space and is configurable as either the primary page table of the operating system or as a single large translation cache in memory (see Figure 4-9). Since the VHPT resides in the virtual address space, an additional TLB miss can be raised when the VHPT is referenced. This property allows the VHPT to also be used as a linear page table.

Figure 4-9. Virtual Hash Page Table (VHPT)



The processor does not manage the VHPT or perform any writes into the table. Software is responsible for insertion of entries into the VHPT (including replacement algorithms), dirty/access bit updates, invalidation due to purges and coherency in a multiprocessor system. The processor does not ensure the TLBs are coherent with the VHPT memory image.

If software needs to control the entries inserted into the TLB more explicitly, or programs the VHPT with differing mappings for the same virtual address range, it may need to take additional action to ensure forward progress. See "VHPT Searching" on page 2:62.

4.1.5.1 VHPT Configuration

The Page Table Address (PTA) register determines whether the processor is enabled to walk the VHPT, anchors the VHPT in the virtual address space, and controls VHPT size and configuration information. The VHPT can be configured as either a per-region virtual linear page table structure (8-byte short format) or as a single large hash page table (32-byte long format). No mixing of formats is allowed within the VHPT.

To implement a per-region linear page table structure an operating system would typically map the leaf page table nodes with small backing virtual translations. The size of the table is expanded to include all possible virtual mappings, effectively creating a large per-region flat page table within the virtual address space.

To implement a single large hash page table, the entire VHPT is typically mapped with a single large pinned virtual translation placed in the translation registers and the size of the table is reduced such that only a subset of all virtual mappings can be resident within the table. Operating systems can tune the size of the hash page table based on the size of physical memory and operating system performance requirements.

4.1.5.2 VHPT Searching

When enabled, the processor's VHPT walker searches the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker checks only the specific VHPT entry addressed by the short- or the long-format hash function, as selected by PTA.vf. If additional TLB misses are encountered during the VHPT access, a VHPT Translation fault is raised. If the region-based short-format VHPT entry contains no reserved bits or encodings, it is installed into the TLB, and the processor again attempts to translate the failed instruction or data reference. If the long-format VHPT entry's tag specifies the correct region identifier and virtual address, and the entry contains no reserved bits or encodings, it is installed into the TLB, and the processor again attempts to translate the failed instruction or data reference. Otherwise the processor raises a TLB Miss fault. The translation is installed into the TLB even if its VHPT entry is marked as not present (p=0). Software may optionally search additional VHPT collision chains (associativities) or search for translations within the operating system's primary page tables. Performance is optimized by placing frequently referenced translations within the VHPT structure directly searched by the processor.

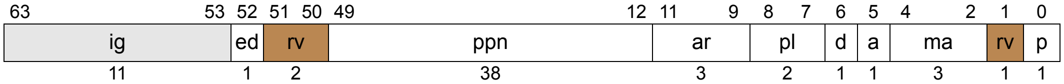
The VHPT walker is optional on a given processor model. Software can neither assume the presence of a VHPT walker, nor that the VHPT walker will find a translation in the VHPT. The VHPT walker can abort a search at any time for implementation-specific reasons, even if the required translation entry is in the VHPT. Operating systems must regard the VHPT walker strictly as a performance optimization and must be prepared to handle TLB misses if the walker fails.

VHPT walks may be done speculatively by the processor's VHPT walker. Additionally, VHPT walks triggered by non-speculatively-executed instructions are not required to be done in program order. Therefore, if the walker is enabled and if the VHPT contains multiple entries that map the same virtual address range, software must set up these entries such that any of them can be used in the translation of any part of this virtual address range. Additionally, if software inserts a translation into the TLB which is needed for forward progress, and this translation has a smaller page size than the translation which would have been inserted on a VHPT walk for the same address, then software may need to disable the VHPT walker in order to ensure forward progress, since this inserted translation may be displaced by a VHPT walk before it can be used.

4.1.5.3 Region-based VHPT Short Format

The region-based VHPT short format shown in Figure 4-10 uses 8-byte VHPT entries to support a per-region linear page table configuration. To use the short-format VHPT, PTA.vf must be set to 0.

Figure 4-10. VHPT Short Format



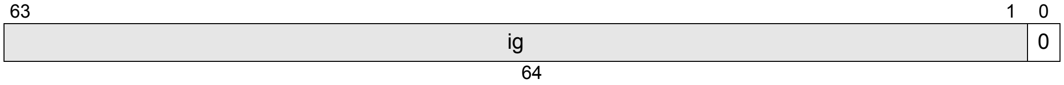
See "Translation Insertion Format" on page 2:53 for a description of all fields. The VHPT walker provides the following default values when entries are installed into the TLB.

- Virtual Page Number – implied by the position of the entry in the VHPT. The hashed short-format entry is considered to be the matching translation.
- Region Identifiers are not specified in the short format. To ensure uniqueness, software must provide unique VHPT mappings per region. Region identifiers obtained from the referenced region register are tagged with the translation when inserted into the TLB.
- Page Size – specified by the accessed region's preferred page size (RR[VA{63:61}].ps)

- Protection Key – specified by the accessed region identifier value (RR[VA{63:61}].rid). As a result, all implementations must ensure that the number of implemented key bits is greater than or equal to the number of implemented region identifier bits.

If a translation is marked as not present, ignored fields are usable by software as noted in Figure 4-11.

Figure 4-11. VHPT Not-present Short Format



4.1.5.4 VHPT Long Format

The long-format VHPT uses 32-byte VHPT entries to support a single large virtual hash page table. To use the long-format VHPT, PTA.vf must be set to 1. The long format is a superset of the TLB insertion format, as noted in Figure 4-12, and specifies full translation information (including protection keys and page sizes). Additional fields are defined in Table 4-9. The long format is typically used to build the hash page table configuration.

Figure 4-12. VHPT Long Format

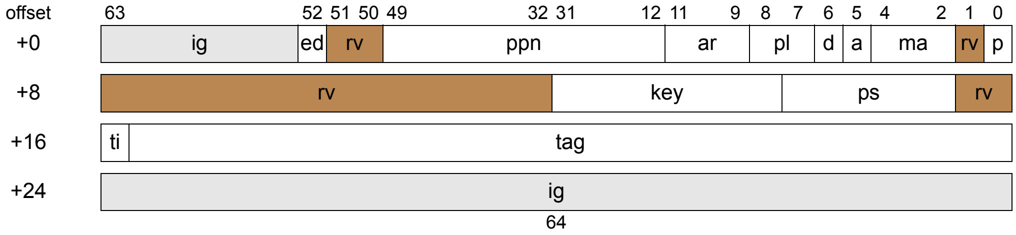
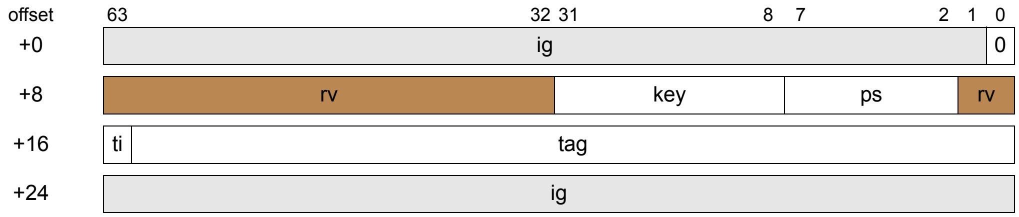


Table 4-9. VHPT Long-format Fields

Field	Offset	Description
tag	+16	Translation Tag – The tag, in conjunction with the VHPT hash index, is used to uniquely identify the translation. Tags are computed by hashing the virtual page number and the region identifier. See "VHPT Hashing" on page 2:65 for details on tag and hash index generation.
ti	+16	Tag Invalid Bit – If one, this bit of the tag indicates an invalid tag. On all processor implementations, the VHPT walker and the `ttag` instruction generate tags with the ti bit equal to 0. A VHPT entry with the ti bit equal to one will never be inserted into the processor's TLBs. Software can use the ti bit to invalidate long-format VHPT entries in memory.
ig	+24	available – field for software use, ignored by the processor. Operating systems may store any value, such as a link address to extend collision chains on a hash collision.

If a translation is marked as not present, ignored fields are usable by software as noted in Figure 4-13. Also, in some implementations, +8{63:32} and +8{31:8} may be ignored as well.

Figure 4-13. VHPT Not-present Long Format



For multiprocessor systems, atomic updates of long-format VHPT entries may be ensured by software as follows:

- Before making multiple non-atomic updates to a VHPT entry in memory, software is required to set its ti bit to one.
- After making multiple non-atomic updates to a VHPT entry in memory, software may clear its ti bit to zero to re-enable tag matches.

The updates to the VHPT entry in memory must be constrained to be observable only after the store that sets the ti bit to one is observable. This can be accomplished with a `mf` instruction, or by performing the updates to the VHPT entry with release stores. Similarly, the clearing of the ti bit must be constrained to be observable only after all of the updates to the VHPT entry are observable. This can be accomplished with a `mf` instruction, or by performing the clear of the ti bit with a release store.

4.1.6 VHPT Hashing

The processor provides two methods for software to determine a VHPT entry's address: the Translation Hash (`thash`) instruction, and the Interruption Hash Address (IHA) register defined on page 2:41. The virtual address of the VHPT entry is placed in the IHA register when a VHPT Translation or TLB fault is delivered. In the long format, IHA can be used as a starting address to scan additional collision chains (associativities) defined by the operating system or to perform a search in software. The `thash` instruction is used to generate a VHPT entry's address outside of interruption handlers and provides the same hash function that is used to calculate IHA.

`thash` produces a VHPT entry's address for a given virtual address and region identifier, depending on the setting of the PTA.vf bit. When PTA.vf=0, `thash` returns the region-based short-format index as defined in "Region-based VHPT Short-format Index" on page 2:65. When PTA.vf=1, `thash` returns the long-format hash as defined in "Long-format VHPT Hash" on page 2:66. The `ttag` instruction is only useful for long-format hashing, and generates a 64-bit ti/tag identifier that the processor's VHPT walker will check when it looks up a given virtual address and region identifier. Software should use the `ttag` instruction, and either the `thash` instruction or the IHA register when forming translation tags and hash addresses for the long-format VHPT. These resources encapsulate the implementation-specific long-format hashing functionality and improve performance.

4.1.6.1 Region-based VHPT Short-format Index

In the region-based short format, the linear page table for each region resides in the referenced region itself. As a result, the short-format VHPT consists of separate per-region page tables, which are anchored in each region by PTA{60:15}. For regions in which the VHPT is enabled, the operating system is required to maintain a per-region linear page table. As defined in Figure 4-14, the VHPT walker uses the virtual address, the region's preferred page size, and the PTA.size field to compute a linear index into the short-format VHPT.

Figure 4-14. Region-based VHPT Short-format Index Function

```
Mask = (1 << PTA.size) - 1;
VHPT_Offset = (VA{IMPL_VA_MSB:0} u>> RR[VA{63:61}].ps) << 3;
VHPT_Addr = (VA{63:61} << 61) |
    (((PTA{60:15} & ~Mask{60:15}) | (VHPT_Offset{60:15} &
       Mask{60:15})) << 15) |
    VHPT_Offset{14:0};
```

The size of the short-format VHPT (PTA.size) defines the size of the mapped virtual address space. The maximum architectural table size in the short format is 2^{52} bytes per region. To map an entire region (2^{61} bytes) using 4Kbyte pages, $2^{(61-12)} = 2^{49}$ pages must be mappable. A short-format VHPT entry is 8 bytes = 2^{3} bytes large. As a result, the maximum table size is $2^{(61-12+3)} = 2^{52}$ bytes per region. If the short format is used to map an address space smaller than 2^{61}, a smaller short-format table (PTA.size<52) can be used. Mapping of an address space of 2^{n} with 4KByte pages requires a minimum PTA.size of (n-9).

In the short format, the `thash` instruction returns the region-based short-format index defined in Figure 4-14. The `ttag` instruction is not used with the short format. VHPT translation and TLB miss faults write the IHA register with the region-based short-format index defined in Figure 4-14.

4.1.6.2 Long-format VHPT Hash

The long-format VHPT is a single large contiguous hash table that resides in the region defined by PTA.base. As defined in Figure 4-15, the VHPT walker uses the virtual address, the region identifier, the region's preferred page size, and the PTA.size field to compute a hash index into the long-format VHPT. PTA{63:15} defines the base address and the region of the long-format VHPT. PTA.size reflects the size of the hash table, and is typically set to a number significantly smaller than 2^{64}; the exact number is based on operating system performance requirements.

Figure 4-15. VHPT Long-format Hash Function

```
Mask = (1 << PTA.size) - 1;
HPN = VA{IMPL_VA_MSB:0} u>> RR[VA{63:61}].ps;
Hash_Index = tlb_vhpt_hash_long(HPN,RR[VA{63:61}].rid);
// model-specific hash function
VHPT_Offset = Hash_Index << 5;
VHPT_Addr = (PTA{63:61} << 61) |
    (((PTA{60:15} & ~Mask{60:15}) | (VHPT_Offset{60:15}
    & Mask{60:15})) << 15) | VHPT_Offset{14:0};
```

The long-format hash function (*tlb_vhpt_hash_long*) and long-format tag generation function are implementation specific. However, on all processor models the hash and tag functions must exclude the virtual region number (virtual address bits VA{63:61}) from the hash and tag computations. This ensures that a unique 85-bit global virtual address hashes to the same VHPT hash address, regardless of which region the address is mapped to. All processor implementations guarantee that the most significant bit of the tag (ti bit) is zero for all valid tags. The hash index and tag together must uniquely identify a translation. The processor must ensure that the indices into the hashed table, the region's preferred page size, and the tag specified in an indexed entry can be used in a reverse hash function to uniquely regenerate the region identifier and virtual address used to generate the index and tag. This must be possible for all supported page sizes, implemented virtual addresses and legal values of region identifiers. A hash function is reversible if using the hash result and all but one input produces the missing input as the result of the reverse hash function. The easiest hash function and reverse hash function is a simple XOR of bits. To ensure uniqueness, software must follow these rules:

1. Software must use only one preferred page size for each unique region identifier at any given time; otherwise, processor operation is undefined.
2. All tags for translations within a given region must be created with the preferred page size assigned to the region; otherwise, processor operation is undefined.
3. Software is not allowed to have pages in the VHPT that are smaller than the preferred page size for the region; otherwise, processor operation is undefined. Software can specify a page with a page size larger than the preferred page size in the VHPT, but tag values for the entries representing that page size must be generated using the preferred page size assigned to that region.
4. To reuse a region identifier with a different preferred page size, software must first ensure that the VHPT contains no insertable translations for that rid, purge all translations for that rid from all processors that may have used it, and then update the region register with the new preferred page size.

4.1.7 VHPT Environment

The processor's VHPT walker can optionally be configured to search the VHPT for a translation after a failed instruction or data TLB search. The VHPT walker is enabled for different types of references under the following conditions:

- Data and non-access references (including IA-32): PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1.
- Instruction fetches (including IA-32): PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1, and PSR.it=1, and PSR.ic=1.
- RSE references: PTA.ve=1, and RR[VA{63:61}].ve=1, and PSR.dt=1, and PSR.rt=1.

If the walker is not enabled, and an attempt is made to reference the VHPT, an Alternate Instruction/Data TLB Miss fault is raised. The remainder of this section assumes that the VHPT is enabled.

Region registers must support all implemented page sizes so software can use IHA, `thash` and `ttag` to manage the VHPT. `thash` and `ttag` are defined to operate on all page sizes supported by the translation cache, regardless of the VHPT walker's supported page sizes. The PTA register must be implemented on processor models that do not implement a VHPT walker. Software must ensure PTA is initialized and serialized before issuing `ttag`, `thash`, before enabling the VHPT walker or issuing a reference that may cause a VHPT walk. The minimum VHPT size is 32KBytes (PTA.size=15), and operating systems must ensure that the VHPT is aligned on the natural boundary of the structure; otherwise, processor operation is undefined. For example, a 64K-byte table must be aligned on a 64K-byte boundary.

VHPT walker references to the VHPT are performed at privilege level 0, regardless of the state of PSR.cpl. VHPT byte ordering is determined by the state of DCR.be. When DCR.be=1, VHPT walker references are performed using big-endian memory formats; otherwise, VHPT walker references are little-endian. A long-format VHPT reference is matched against the data break-point registers as a 32-byte reference.

The VHPT is accessed by the processor only if the VHPT is virtually mapped into cacheable memory areas. The walker may access the VHPT speculatively, i.e., references may be performed that are not required by an in-order execution of the program. Any VHPT or TLB faults encountered during a VHPT walker's search are not reported until the faulting translation is required by an in-order execution of the program. If the VHPT is mapped into non-cacheable memory areas the VHPT is not referenced, and all TLB misses result in an Instruction/Data TLB Miss fault.

The VHPT walker will abort the search and deliver an Instruction/Data TLB Miss fault if an attempt is made to install translations that have reserved bits or encodings, or if the translation mapping the VHPT would have taken one of the following faults: Data Page Not Present, Data NaT Page Consumption, Data Key Miss, Data Key Permission, Data Access Bit, or Data Debug. The VHPT walker may abort a search and deliver an Instruction/Data TLB Miss fault at any time for implementation-specific reasons.

The processor's VHPT walker is required to read and insert VHPT entries from memory atomically (an 8-byte atomic read-and-insert for short format, and a 32-byte atomic read-and-insert for long format). Some implementation strategies for achieving this atomicity are as follows:

- If the walker performs its VHPT read with multiple cache accesses which are not done as an atomic unit, and if an update to part of the entry that is being installed is made in-between these multiple reads, the walker must abort the insert and deliver an Instruction/Data TLB Miss.
- If the walker performs its VHPT read and the insertion of the entry into the TLB as separate actions, and not as an atomic unit, and if an update to part of the entry that is being installed is made in-between the read and the insert, the walker must either abort the insert and deliver an Instruction/Data TLB Miss, or ignore the update and install the complete old entry.
- If the purge address range of a TLB purge operation (`ptc.l`, `ptc.e`, local or remote `ptc.g` or `ptc.ga`, `ptr.i`, or `ptr.d`) overlaps the virtual address the walker is attempting to insert, then the walker must either abort the insert and deliver an Instruction/Data TLB Miss, or delay the purge operation until after the walker either completes the insertion or aborts the walk.

The RSE can only raise a VHPT fault on a mandatory RSE spill/fill operation as defined for successful execution of an `alloc`, `loadrs`, `flushrs`, `br.ret` or `rfi` instruction. Eager RSE operations may generate speculative VHPT walks provided encountered faults are not reported.

Data TLB Miss faults encountered during a VHPT walk are permitted and, when PSR.ic=1, are converted into a VHPT Translation fault as defined in the next section.

4.1.8 Translation Searching

The general sequence of searching the TLB and VHPT is shown in Figure 4-16. On a failed TLB search, if the VHPT walker is disabled for the referenced region an Alternate Instruction/Data TLB Miss fault is raised. If the VHPT walker is enabled for the referenced region, the VHPT is accessed to locate the missing translation. See "VHPT Environment" on page 2:67. If additional TLB misses are encountered during the VHPT walker's references, a VHPT Translation fault is raised. If the VHPT walker does not find the required translation in the VHPT or the search is aborted, an Instruction/Data TLB Miss fault is raised. Otherwise the entry is loaded into the ITC or DTC. Provided the above fault conditions are not detected, the processor may load the entry into the ITC or DTC even if an in-order execution of the program did not require the translation.

See Table 4-1, "Purge Behavior of TLB Inserts and Purges," on page 2:52 for the purge behavior of VHPT walker inserts.

After the translation entry is loaded, additional TLB faults are checked; these include in priority order: Page Not Present, NaT page Consumption, Key Miss, Key Permission, Access Rights, Access Bit, and Dirty Bit faults. Table 4-10 describes the TLB and VHPT walker related faults.

On a failed TLB/VHPT search, the processor loads interruption registers and translation defaults as defined in "Interruption Vector Descriptions" on page 2:165 defining the parameters of the translation fault. Provided the operating system accepts the defaults provided, only the physical address portion of a TLB entry need be provided on a TLB insert.

Figure 4-16. TLB/VHPT Search

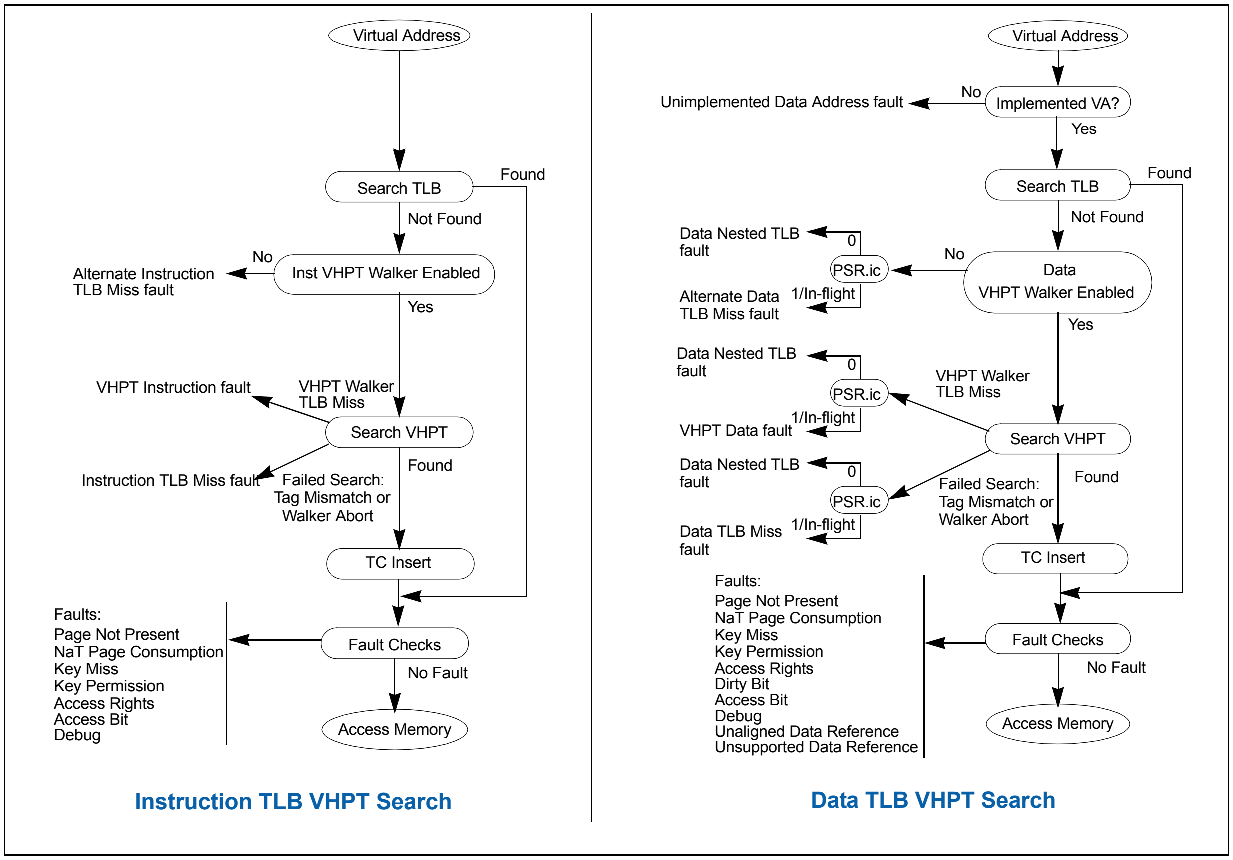


Table 4-10. TLB and VHPT Search Faults

Fault	Description
VHPT Instruction/Data	Raised if there is an additional TLB miss when the VHPT walker attempts to access the VHPT. Typically used to construct leaf table mappings for linear page table configurations.
Alternate Instruction/Data TLB Miss	Raised when the VHPT walker is not enabled and an instruction or data reference causes a TLB miss. For example, the VHPT walker can be disabled within a given virtual region so region-specific translation algorithms can be utilized.

Table 4-10. TLB and VHPT Search Faults (Continued)

Fault	Description
Instruction/Data TLB Miss	Raised when the VHPT walker is enabled, but the processor: • Cannot locate the required VHPT entry, or • The processor aborts the VHPT search for implementation-specific reasons, or • The VHPT walker is not implemented, or • The referenced region specifies a non-supported VHPT preferred page size, or • Reserved fields or unimplemented PPN bits are used in the translation, or • The hash address falls into unimplemented virtual address space, or • The hash address matches a data debug register. Instruction/Data TLB Miss handlers are essentially software walkers of the VHPT.
Data Nested TLB	Raised when a Data TLB Miss, Alternate Data TLB Miss, or VHPT Data Translation fault occurs and PSR.ic is 0 and not in-flight (e.g., fault within a TLB miss handler). Data Nested TLB faults enable software to avoid overheads for potential data TLB Miss faults.
Instruction/Data Page Not Present	The referenced translation's P-bit is 0.
Instruction/Data NaT Page Consumption	A non-speculative load, store, mandatory RSE load/store, execution on, or semaphore operation accesses a page marked with the physical memory attribute NaTPage. See "Not a Thing Attribute (NaTPage)" on page 2:86 for details.
Instruction/Data Key Miss	The referenced translation's permission key is not present in the set of valid protection key registers.
Instruction/Data Key Permission	The referenced translation is denied read, write, execute permissions by the matching protection key registers.
Instruction/Data Access Rights	Page granular read, write, execute and privilege level accesses are denied.
Data Dirty Bit	The referenced translation's Dirty bit is 0 on a store or semaphore operation.
Instruction/Data Access Bit	The referenced translation's Access bit is 0.

4.1.9 32-bit Virtual Addressing

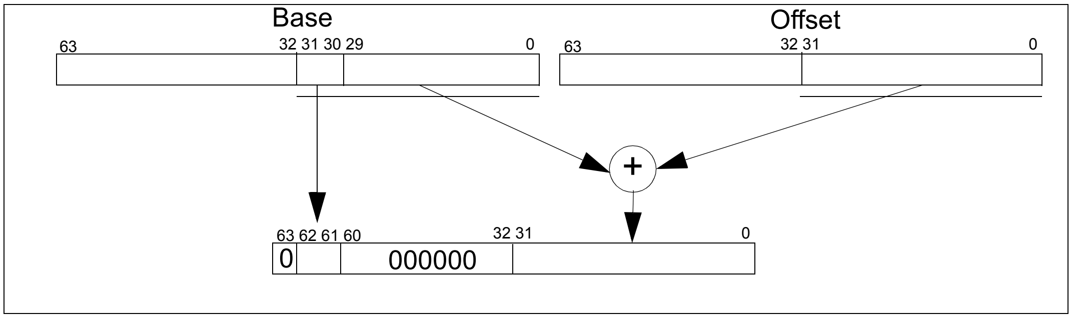
32-bit virtual data addressing is supported in the Itanium instruction set architecture by three models: zero-extension, sign-extension, and pointer "swizzling." IA-32 memory references use the zero-extension model, all IA-32 32-bit virtual linear addresses are zero extended into the 64-bit virtual address space.

The zero-extension model performs address computations with the `add` and `shladd` instructions while software ensures that the upper 32-bits are always zeros. This model constrains 32-bit virtual addressing to virtual region zero. In this model, regions 1 to 7 are accessible only by 64-bit addressing.

In the sign-extension model, software ensures that the upper 32-bits of a virtual address are always equal to bit 31. Address computations use the `add`, `shladd`, and `sxt` instructions. This model splits the 32 bit address space into two halves that are spread into 2^{31} bytes of virtual regions 0 and 7 within the 64-bit virtual address space. In this model, regions 2 to 6 are accessible only by 64-bit addressing.

The pointer "swizzling" model performs address computations with the `addp4`, and `shladdp4` instructions. These instructions generate a 32-bit address within the 64-bit virtual address space as shown in Figure 4-17. The 32-bit virtual address space is divided into 4 sections that are spread into 2^{30} bytes of virtual regions 0 to 3 within the 64-bit virtual address space. In this model, regions 4 to 7 are accessible only by 64-bit addressing.

Figure 4-17. 32-bit Address Generation using addp4



In the pointer "swizzling" model, mappings within each region do not necessarily start at offset zero, since the upper 2-bits of a 32-bit address serve both as the virtual region number and an offset within each region. Virtual address bits{62:61} do not participate in the address addition, therefore some regions may be effectively larger than 2^{30} bytes due to the addition of a 32-bit offset and lack of a carry into bits{62:61}. Note that the conversion is non-destructive: a converted 64-bit pointer can be used as a 32-bit pointer. Flat 31 or 32 bit address spaces can be constructed by assigning the same region identifier to contiguous region registers. Branches into another 2^{30}-byte region are performed by first calculating the target address in the 32-bit virtual space and then converting to a 64-bit pointer by `addp4`. Otherwise, branch targets will extend above the 2^{30} byte boundary within the originating region.

4.1.10 Virtual Aliasing

Virtual aliasing (two or more virtual pages mapped to the same physical page) is functionally supported for memory references (including IA-32), however performance may be degraded on some processor models where the distance between virtual aliases is less than 1 MB. To avoid any possible performance degradation, software is advised to use aliases whose virtual addresses differ by an integer multiple of 1 MB. The processor ensures cache coherency and data dependencies in the presence of an alias. Stores using a virtual alias followed by a load with another alias to the same physical location see the effects of prior stores to the same physical memory location.

To support advanced loads in the presence of a virtual alias, the processor ensures that the Advanced Load Address Table (ALAT) is resolved using physical addresses and is coherent with physical memory. For details, please refer to "Detailed Functionality of the ALAT and Related Instructions" on page 1:65.

4.2 Physical Addressing

Objects in memory and I/O occupy a common 63-bit physical address space that is accessed using byte addresses. Accesses to physical memory and I/O may be performed via virtual addresses mapped to the 63-bit physical address space or by direct physical addressing. Current page table formats allow for mapping virtual addresses into 50 bits of physical address space (on processor implementations that support this many physical address bits). Future extensions to the page table formats will allow larger mappings, up to the full 63 bits of physical address space.

Physical addressing for instruction references (including IA-32) is enabled when PSR.it is 0, data references (including IA-32) when PSR.dt is 0, and register stack references when PSR.rt is 0.

While software views the physical addressing as being 63-bits, implementations may implement between 32 and 63 physical address bits. All processor models must implement a contiguous set of physical address bits starting at bit 32 and continuing upwards. Please see the processor-specific documentation for further information on the number of physical address bits implemented on the Itanium processor. Implementations must validate that memory references are performed to implemented physical address bits. Instruction references to unimplemented physical addresses result either in an Unimplemented Instruction Address trap on the last valid instruction, or in an Unimplemented Instruction Address fault on the instruction fetch of the unimplemented address. Data references to unimplemented physical addresses result in an Unimplemented Data Address fault. Memory references to unpopulated address ranges result in an asynchronous Machine Check abort, when the platform signals a transaction time-out. Exact machine check behavior is model specific.

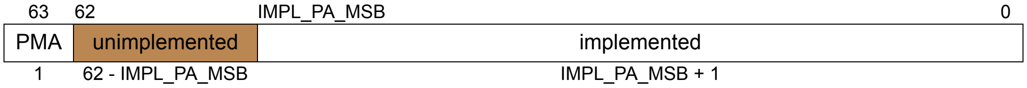
4.3 Unimplemented Address Bits

Based on the processor model, some physical and/or virtual address bits may not be implemented. Regardless of the number of implemented address bits, all general purpose, branch, control and application registers implement all 64 register bits on all processors. Similarly, regardless of the number of implemented address bits, data and instruction breakpoint registers must implement all 64 address bits and all 56 mask bits on all processors.

4.3.1 Unimplemented Physical Address Bits

As shown in Figure 4-18, a 64-bit physical address consists of three fields: physical memory attribute (PMA), unimplemented and implemented bits.

Figure 4-18. Physical Address Bit Fields



All processor models implement at least 32 physical address bits, bits 0 to 31, plus the physical memory attribute bit. Additional implemented physical bits must be contiguous starting at bit 32. IMPL_PA_MSB is the implementation-specific position of the most significant implemented physical address bit. In a processor that implements all physical address bits, IMPL_PA_MSB is 62. Please see the processor-specific documentation for further information on the number of physical address bits implemented on the Itanium processor.

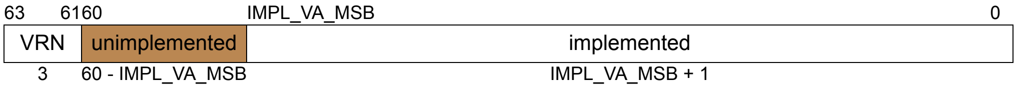
If unimplemented physical address bits are set by software, an Unimplemented Data Address fault is raised during the TLB insert instructions (`itc`, `itr`). Inserts performed by the VHPT walker, as noted in "VHPT Hashing" on page 2:65, abort the VHPT search if unimplemented or reserved fields are used. For translations marked as Not-Present (TLB.p is 0), the processor does not check the validity of PPN and some reserved bits as noted in Figure 4-6.

When a processor model does not implement all physical address bits, the missing bits are defined to be zero. Physical addresses in which bits PA{62:min(IMPL_PA_MSB+1,62)} are not zero are considered "unimplemented" physical addresses on that processor model. Physical addresses are checked for correctness on use by ensuring that PA{62:min(IMPL_PA_MSB+1,62)} bits are zero.

4.3.2 Unimplemented Virtual Address Bits

As shown in Figure 4-19, a 64-bit virtual address consists of three fields: virtual region number (VRN), unimplemented and implemented bits.

Figure 4-19. Virtual Address Bit Fields



All processor models provide three VRN bits in VA{63:61}. IMPL_VA_MSB is the implementation-specific bit position of the most significant implemented virtual address bit. In addition to the three VRN bits, all processor models implement at least 54 virtual address bits; i.e., the smallest IMPL_VA_MSB is 53. In a processor that implements all 64 virtual address bits IMPL_VA_MSB is 60. Please see the processor-specific documentation for further information on the number of virtual address bits implemented on the Itanium processor.

If the PSR.vm bit is implemented, and if PSR.vm is 1, then virtual addresses are treated as though one additional virtual address bit were unimplemented. If the PSR.vm bit is implemented, at least 55 virtual address bits must be implemented.

When a processor model does not implement all virtual address bits, the missing bits are defined to be a sign-extension of VA{IMPL_VA_MSB}. Virtual addresses in which bits VA{60:min(IMPL_VA_MSB+1,60)} do not match VA{IMPL_VA_MSB} are considered "unimplemented" virtual addresses on that processor model. Virtual addresses are checked for correctness on use by ensuring that VA{60:min(IMPL_VA_MSB+1,60)} bits are identical to VA{IMPL_VA_MSB}.

4.3.3 Instruction Behavior with Unimplemented Addresses

The use of an unimplemented address affects instruction execution as described in the bullet list below. If instruction address translation is enabled, an "unimplemented address" refers to an unimplemented virtual address. If instruction address translation is disabled, an "unimplemented address" refers to an unimplemented physical address.

- Non-speculative memory references (non-speculative loads, stores, and semaphores), the following non-access references: `fc`, `fc.i`, `tpa`, `lfetch.fault`, and `probe.fault`, and mandatory RSE operations to unimplemented addresses result in an Unimplemented Data Address fault.
- Virtual addresses used by instruction and data TLB purge/insert operations are checked, and if the base address (register r3 of the purge, IFA for inserts) targets an unimplemented virtual address, a Unimplemented Data Address fault is raised. The page size of the insert or purge is ignored.
- Speculative loads from unimplemented addresses always return a NaT bit in the target register.
- A regular_form `probe` instruction to an unimplemented address returns zero in the target register.
- A `tak` instruction to an unimplemented address returns one in the target register.
- A non-faulting `lfetch` to an unimplemented address is silently ignored.
- Eager RSE operations to unimplemented addresses do not fault.
- Execution of a taken branch, taken `chk`, or an `rfi` to an unimplemented address, or execution of a non-branching slot 2 instruction in a bundle at the upper edge of the implemented address space (where the next sequential bundle address would be an unimplemented address) results either in an Unimplemented Instruction Address trap on the branch, `chk`, `rfi` or non-branching slot 2 instruction, or in an Unimplemented Instruction Address fault on the fetch of the unimplemented address.
- When `ptc.g` or `ptc.ga` operations place a virtual address on the bus, the virtual address is sign-extended to a full 64-bit format. If an incoming `ptc.g` or `ptc.ga` presents a virtual address base that targets an unimplemented virtual address, the upper (unimplemented) virtual address bits are dropped, and the purge is performed with the truncated address.
- The behavior of executing `vmsw.1` in a bundle whose address will become unimplemented after PSR.vm is set to 1 is undefined.

4.4 Memory Attributes

When virtual addressing is enabled, memory attributes defining the speculative, cacheability and write-policies of the virtually mapped physical page are defined by the TLB. When physical addressing is enabled, memory attributes are supplied as described in "Physical Addressing Memory Attributes" on page 2:76.

4.4.1 Virtual Addressing Memory Attributes

For virtual memory references, the memory attribute field of each virtual translation describes physical memory properties as shown in Table 4-11.

Table 4-11. Virtual Addressing Memory Attribute Encodings

Attribute	Mnemonic	ma	Cacheability	Write Policy	Speculation	Coherent[a] with Respect to
Write Back	WB	000	Cacheable	Write back	Non-sequential & speculative	WB, WBL
Write Coalescing	WC	110	Uncacheable	Coalescing		Not MP coherent[b]
Uncacheable	UC	100		Non-coalescing	Sequential & non-speculative	UC, UCE
Uncacheable Exported	UCE	101				
Reserved[c]		001				
Reserved		010 011				
NaTPage	NaTPage	111	Cacheable	N/A	Speculative	N/A

a. The Coherency column in this table refers to multiprocessor coherence on normal, side-effect free memory. The data dependency rules defined in "Memory Access Ordering" on page 1:73 ensure uni-processor coherence for the memory attributes listed in each row.

b. WC is not MP coherent w.r.t. any memory attribute, but is uni-processor coherent w.r.t. itself.

c. This memory attribute is reserved for Software use.

The attribute UCE is identical to UC except when executing an `fetchadd` instruction. UCE enables the exporting of the `fetchadd` instruction outside the processor. Support for UCE is model-specific; see "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86 for details.

Insert TLB instructions (`itc`, `itr`) that attempt to insert reserved memory attributes (Table 4-11) into the TLB raise Reserved Register/Field faults. External system operation is undefined if software inserts a memory attribute supported by the processor but not supported by the external system.

If software modifies the memory attributes for a page, it must follow the attribute transition requirements in Section 4.4.11, "Memory Attribute Transition" on page 2:88.

It is recommended that processor models report a Machine Check abort if the following memory attribute aliasing is detected:

- Cache hit on an uncacheable page, other than as the target of a local or remote flush cache (`fc`, `fc.i`) instruction (see "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86).

4.4.2 Physical Addressing Memory Attributes

The selection of memory attributes for physical addressing is selected by bit 63 of the address contained in the address base register as shown in Figure 4-20 and Table 4-12.

Figure 4-20. Physical Addressing Memory

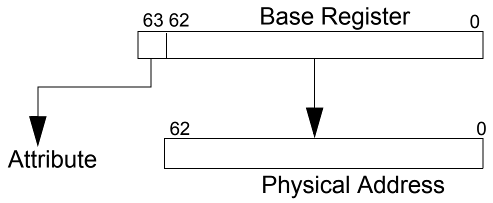


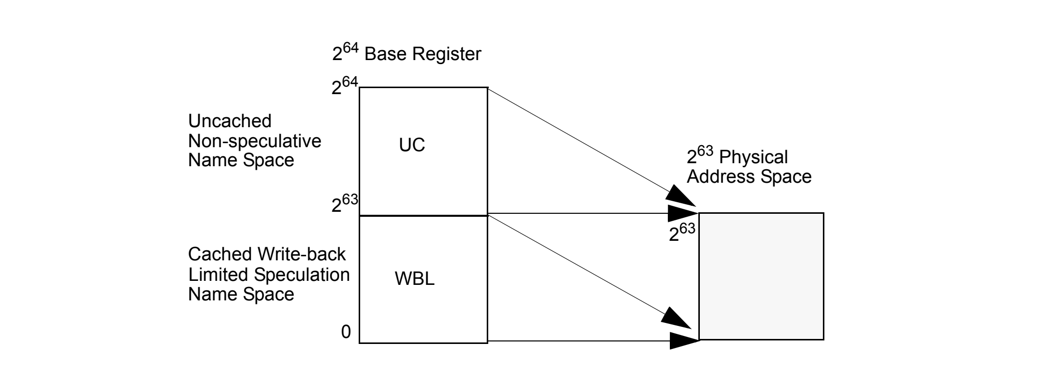
Table 4-12. Physical Addressing Memory Attribute Encodings

Bit{63}	Mnemonic	Cacheability	Write Policy	Speculation	Coherent[a] with respect to
0	WBL	Cacheable	Write Back	Non-sequential & limited speculation	WBL, WB
1	UC	Uncached	Non-coalescing	Sequential & non-speculative	UC, UCE

[a] Coherency here refers to multiprocessor coherence on normal, side-effect free memory.

See "Speculation Attributes" on page 2:79 for a description of physical addressing limited speculation. Bit{63} is discarded when forming the physical address, effectively creating a write-back name space and an uncached name space as shown in Figure 4-21.

Figure 4-21. Addressing Memory Attributes



Software must use the correct name space when using physical addressing; otherwise, I/O devices with side-effects may be accessed speculatively. Physical addressing accesses are ordered only if ordered loads or ordered stores are used. Otherwise, physical addressing memory references are unordered.

4.4.3 Cacheability and Coherency Attribute

A page can be either **cacheable** or **uncacheable**. If a page is marked cacheable, the processor is permitted to allocate a local copy of the corresponding physical memory in all levels of the processor memory/cache hierarchy. Allocation may be modified by the cache control hints of memory reference instructions.

A page which is cached is coherent with memory; i.e., the processor and memory system ensure that there is a consistent view of memory from each processor. Processors support multiprocessor cache coherence based on physical addresses between all processors in the coherence domain (tightly coupled multiprocessors). Coherency is supported in the presence of virtual aliases, although software is recommended to use aliases which are an integer multiple of 1 MB apart to avoid any possible performance degradation.

Processors are not required to maintain coherency between processor local instruction and data caches for Itanium architecture-based code; i.e., locally initiated Itanium stores may not be observed by the local instruction cache. Processors are required to maintain coherency between processor local instruction and data caches for IA-32 code. Instruction caches are also not required to be coherent with multiprocessor Itanium instruction set originated memory references. Instruction caches are required to be coherent with multiprocessor IA-32 instruction set originated memory references. The processor must ensure that transactions from other I/O agents (such as DMA) are physically coherent with the instruction and data cache.

For non-cacheable references the processor provides no coherency mechanisms; the memory system must ensure that a consistent view of memory is seen by each processor. See "Coalescing Attribute" on page 2:78 for a description of coherency for the coalescing memory attribute.

4.4.4 Cache Write Policy Attribute

Write-back cacheable pages need only modify the processor's copy of the physical memory location; written data need only be passed to the memory system when the processor's copy is displaced, or a Flush Cache (`fc`) instruction is issued to flush a virtual address. A cache line can only be written back to memory if a store, semaphore (successful or not), the `ld.bias`, a mandatory RSE store, or a `.excl` hinted lfetch instruction targeting that line has executed without a fault. These events enable write-backs. A synchronized `fc` instruction disables subsequent write-backs (after the line has been flushed).

As described in "Invalidating ALAT Entries" on page 1:67, platform visible removal of cache lines from a processor's caches (e.g., cache line write-backs or platform visible replacements) cause the corresponding ALAT entries to be invalidated.

4.4.5 Coalescing Attribute

For uncacheable pages, the **coalescing** attribute informs the processor that multiple stores to this page may be collected in a coalescing buffer and issued later as a single larger merged transaction. The processor may accumulate stores for an indefinite period of time. Multiple pending loads may also be coalesced into a single larger transaction which is placed in a coalescing buffer. Coalescing is a performance hint for the processor; a processor may or may not implement coalescing.

A processor with multiple coalescing buffers must provide a flush policy that flushes buffers at roughly equal rate even if some buffers are only partially full. The processor may make coalesced buffer flushes visible in any order. Furthermore, individual bytes within a single coalesced buffer may be flushed and made visible in any order.

Stores (including IA-32), which are coalesced, are performed out of order; coalescing may occur in both the space and time domains. For example, a write to bytes 4 and 5 and a write to bytes 6 and 7 may be coalesced into a single write of bytes 4, 5, 6, and 7. In addition, a write of bytes 5 and 6 may be combined with a write of bytes 6 and 7 into a single write of bytes 5, 6, and 7.

Any release operation (regardless of whether it references a page with a coalescing memory attribute), or any fence type instruction, forces write-coalesced data to be flushed and made visible prior to the instruction itself becoming visible. (See Table 4-15 on page 2:83 for a list of release and fence instructions.) Any IA-32 serializing instruction, or access to an uncached memory type, forces write-coalesced data to become flushed and made visible prior to itself becoming visible. Even though IA-32 stores and loads are ordered, the write-coalesced data is not flushed unless the IA-32 stores or loads are to uncached memory types.

The Flush Cache (`fc`, `fc.i`) instruction flushes all write-coalesced data whose address is within at least 32 bytes of the 32-byte aligned address specified by the Flush Cache (`fc`, `fc.i`) instruction, forcing the data to become visible. The Flush Cache (`fc`, `fc.i`) instruction may also flush additional write-coalesced data. The Flush Write buffers (`fwb`) instruction is a "hint" to the processor to expedite flushing (visibility) of any pending stores held in the coalescing buffer(s), without regard to address.

No indication is given when the flushing of the stores is completed. An `fwb` instruction does not ensure ordering of coalesced stores, since later stores may be flushed before prior stores. To ensure prior coalesced stores are made visible before later stores, software must issue a release operation between stores.

The processor may at any time flush coalesced stores in any order before explicitly requested to do so by software.

Coalesced pages are not ensured to be coherent with other processors' coalescing buffers or caches, or with the local processor's caches. Loads to coalesced memory pages by a processor see the results of all prior stores by the same processor to the same coalesced memory page. Memory references made by the coalescing buffer (e.g., buffer flushes) have an unordered non-sequential memory ordering attribute. See "Sequentiality Attribute and Ordering" on page 2:82.

Data that has been read or prefetched into a coalescing buffer prior to execution of an Itanium acquire or fence type instruction is invalidated by the acquire or fence instruction. (See Table 4-15 for a list of acquire and fence instructions.)

4.4.6 Speculation Attributes

For present pages (TLB.p=1) which are marked with a **speculative** or a NaTPage memory attribute, the processor may prefetch instructions (including IA-32), perform address generation and perform load accesses (including IA-32) without resolving prior control dependencies, including predicates, branches and interruptions. A page should only be marked speculative if accesses to that page have no side-effects. For example, many memory-mapped I/O devices have side-effects associated with reads and should be marked non-speculative. If a page is marked speculative, a processor can read any location in the page at any time independent of a programmer's intentions or control flow changes. As a result, software is required, at all times, to maintain valid page table attributes for the ppn, ps and ma fields of all present translations whose memory attribute is speculative or NaTPage. (For example, software should not insert into the TLB, nor create in the VHPT, mappings whose memory attribute is WB, WC or NaTPage unless the entire corresponding physical address range is populated. Placing such mappings in the VHPT or inserting such mappings in the TLB could result in machine check aborts.) High-performance operation is only attainable on speculative pages. The speculative attribute is a hint; a processor may behave non-speculatively.

Prefetches are enabled if a speculative translation exists. Prefetches are asynchronous data and instruction memory accesses that appear logically to initiate and finish between some pair of instructions. This access may not be visible to subsequent flush cache (`fc`, `fc.i`) and/or TLB purge instructions. This behavior is implementation-dependent.

The processor will not initiate memory references (16-byte instruction bundle fetches, IA-32 instruction fetches, RSE fills and spills, VHPT references, and data memory accesses) to non-speculative pages until all previous control dependencies (predicates, branches, and exceptions) are resolved; i.e., the memory reference is required by an in-order execution of the program. Additionally, for references to non-speculative pages, the processor:

- May not generate any memory access for a control or data speculative data reference.
- Will generate exactly one memory access for each aligned, non-speculative data reference. (Misaligned data references may cause multiple memory accesses, although these accesses are guaranteed to be non-overlapping – each byte will be accessed exactly once.)
- May generate multiple 16-byte memory accesses (to the same address) for each 16-byte instruction bundle fetch reference.

To ensure virtual and physical accesses to non-speculative pages are performed in program order and only once per program order occurrence, the rules in Table 4-13 and Table 4-14 are defined. Software should also ensure that RSE spill/fill transactions are not performed to non-speculative memory that may contain I/O devices; otherwise, system behavior is undefined.

Table 4-13. Permitted Speculation

Memory Attribute	Load (ld)[a]	Speculative Load (ld.s)[b]	Advanced Load (ld.a)	Speculative Advanced Load (ld.sa)	Hardware-generated Speculative References[c]
Speculative	Yes	Yes	Yes	Yes	Yes
Non-speculative	Yes	Always Fail	Always Fail	Always Fail	Prohibited
Limited Speculation	Yes	Always Fail	Yes	Always Fail	Limited[d]

a. Includes the faulting form of line prefetch (`lfetch.fault`).
b. Includes the non-faulting form of line prefetch (`lfetch`), which does not cause a cache fill if the memory attribute is non-speculative or limited speculation.
c. Hardware-generated speculative references include non-demand instruction prefetches (including IA-32), hardware-generated data prefetch references, and eager RSE memory references.
d. The processor may only issue hardware-generated speculative references to a 4K-byte physical page if it is a verified page.

Table 4-14. Register Return Values on Non-faulting Advanced/Speculative Loads

Memory Attribute	Speculative Load (ld.s)		Advanced Load (ld.a)		Speculative Advanced Load (ld.sa)	
	Success	Failure	Success	Failure	Success	Failure
Speculative	Value	Nat[a]	Value	N/a	Value	NaT[a]
Non-speculative	N/A	Nat[b]	N/A	Zero[c]	N/A	NaT[b]
Limited Speculation	N/A	Nat[b]	Value	N/a	N/a	NaT[b]

a. Speculative or speculative advanced loads that cause deferred exceptions result in failed speculation. The processor aborts the reference. If the target of the load is a GR, the processor sets the register's NaT bit to one. If the target of the load is an FR, the processor sets the target FR to NaTVal. The processor performs all other side-effects (such as post-increment).

b. Speculative or speculative advanced loads to limited or non-speculative memory pages result in failed speculation. The processor aborts the reference. If the target of the load is a GR, the processor sets the register's NaT bit to 1. If the target of the load is an FR, the processor sets the target FR to NaTVal. The processor performs all other side-effects (such as post-increment).

c. Advanced loads to non-speculative memory pages always fail. The processor aborts the reference, sets the target register to zero, and performs all other side-effects (such as post-increment).

4.4.6.1 Limited Speculation and the WBL Physical Addressing Attribute

Processors are allowed to reference limited speculation pages (WBL pages) speculatively, in order to increase performance, but this speculation is limited to prevent speculative references to 4Kbyte physical pages for which there is no actual memory (which would cause spurious machine checks).

Processors must not make hardware-generated speculative references to a given WBL 4Kbyte page until a **verified reference** has been made. Processors may optionally implement storage to hold the addresses of WBL 4Kbyte pages for which verified references have been made, and may make subsequent hardware-generated speculative references to these pages. Such pages are termed **verified pages**.

A verified reference is an instruction or data reference made to the page by an in-order execution of the program; that is, a reference which would have been made had the instructions from the program been fetched and executed one at a time. A hardware-generated speculative reference does not constitute a verified reference. Hardware-generated speculative references include:

- Instruction fetches when the processor has not yet determined whether prior branches were predicted correctly
- Instruction fetches when the processor has not yet determined whether prior instructions will raise faults or traps
- Data references by instructions when the processor has not yet determined whether prior branches were predicted correctly
- Data references by instructions when the processor has not yet determined whether prior instructions will raise faults or traps
- Hardware-generated instruction prefetch references
- Hardware-generated data prefetch references
- Eager RSE data references

For an instruction fetch to constitute a verified reference, it must only be determined that an in-order execution of the program requires that the IP point to this address, independent of whether the instruction at this address will subsequently take a fault or interrupt.

For a data reference to constitute a verified reference, the instruction must meet one of the following requirements:

- It executes without any fault or interrupt
- It takes an Unaligned Data Reference fault
- It takes a Data Debug fault

- It takes an External interrupt, but if it had not taken an External interrupt, it would have met one of the above qualifications (execute without fault, take an Unaligned Data Reference fault, or take a Data Debug fault)

Data-speculative loads are treated the same as normal loads, and if an in-order execution of the program requires the execution of a data speculative load, it constitutes a verified reference. Control-speculative loads to limited-speculation pages always defer and thus never constitute verified references.

It is not necessary for a processor to determine whether a reference will complete without generating a machine check for it to be a verified reference. If software actually references a physical address which will cause a machine check, hardware may generate multiple speculative references to the same page, potentially causing multiple machine checks.

Processors may access verified pages normally, as they would WB pages, including the use of caching, pipelining and hardware-generate speculative references to improve performance.

Calling the PAL_PREFETCH_VISIBILITY procedure forces the processor to clear the storage holding the addresses of verified pages.

4.4.7 Sequentiality Attribute and Ordering

Memory ordering is defined in Section 4.4.7, "Memory Access Ordering" on page 1:73. This section defines additional ordering rules for non-cacheable memory, cache synchronization (`sync.i`) and global TLB purge operations (`ptc.g`, `ptc.ga`).

As described in Section 4.4.7, "Memory Access Ordering" on page 1:73, read-after-write, write-after-write, and write-after-read dependencies to the same memory location (memory dependency) are performed in program order by the processor. Otherwise, all other memory references may be performed in any order unless the reference is specifically marked as ordered. No ordering exists between instruction accesses and data accesses or between any two instruction accesses. IA-32 memory references follow a stronger processor consistency memory model. See "IA-32 Memory Ordering" on page 2:265. for IA-32 memory ordering details. Explicit ordering takes the form of a set of Itanium instructions: ordered load and check load (`ld.acq`, `ld.c.clr.acq`), ordered store (`st.rel`), semaphores (`cmpxchg`, `xchg`, `fetchadd`), memory fence (`mf`), synchronization (`sync.i`) and global TLB purge (`ptc.g`, `ptc.ga`). The `sync.i` instruction is used to maintain an ordering relationship between instruction and data caches on local and remote processors. The global TLB purge instructions maintain multiprocessor TLB coherence.

For VHPT walks, visibility is defined by the memory read(s) which retrieves translation information, and the associated insertion of the translation into the TLB. VHPT walks are performed asynchronously with respect to program execution, and each walker VHPT read (which appears as though it were performed atomically) is made visible at some single point in the program order. Ordering constraints from Table 4-15 do not prevent VHPT walks from becoming visible.

Table 4-15 defines a set of "Orderable Instructions" that follow one of four ordering semantics: **unordered**, **release**, **acquire** or **fence**. The table defines the ordering semantics and the instructions of each category. Only these Itanium instructions can be used to establish multiprocessor ordering relations.

In the following discussion, the terms **previous** and **subsequent** are used to refer to the program specified order. The term **visible** is used to refer to all architecturally visible effects of performing an instruction. For memory accesses and semaphores this involves at least reading or writing memory. For `mf.a`, visibility is defined by platform acceptance of previous memory accesses. Visibility of `sync.i` is defined by visibility of previous flush cache (`fc`, `fc.i`) operations. For ALAT lookups (`ld.c`, `chk.a`), visibility is determination of ALAT hit or miss. For global TLB purge operations, visibility is defined by removal of an address translation from the TLBs on all processors in the TLB coherence domain. Global TLB purge instructions (`ptc.g` and `ptc.ga`) follow release semantics on the local processor. They are also broadcast to all other processors in the TLB coherence domain. On each such remote processor, a point is chosen in its program-order execution and a local TLB purge operation is inserted at that point; this local TLB purge operation follows release semantics, except with respect to global purge instructions being executed by that remote processor. For local TLB purge operations, visibility is defined by removal of an address translation on the local processor. Local TLB purge instructions (`ptc.l`, `ptc.e`) ensure that all prior stores are made locally visible before the actual purge operation is performed.

Table 4-15. Ordering Semantics and Instructions

Ordering Semantics	Description	Orderable Intel® Itanium® Instructions

Unordered	Unordered instructions may become visible in any order.	`ld, ld.s, ld.a, ld.sa, ld.fill,`
		`ldf, ldf.s, ldf.sa, ldf.fill,`
		`ldfp, ldfp.s, ldfp.sa,`
		`st, st.spill,`
		`stf, stf.spill,`
		`mf.a, sync.i,`
		`ld.c, chk.a`

Release	Release instructions guarantee that all previous orderable instructions are made visible prior to being made visible themselves.	`cmp8xchg16.rel, cmpxchg.rel, fetchadd.rel, st.rel, ptc.g, ptc.ga`
Acquire	Acquire instructions guarantee that they are made visible prior to all subsequent orderable instructions.	`cmp8xchg16.acq, cmpxchg.acq, fetchadd.acq, xchg, ld.acq, ld.c.clr.acq`
Fence	Fence instructions combine the release and acquire semantics into a bi-directional fence; i.e., they guarantee that all previous orderable instructions are made visible prior to any subsequent orderable instruction being made visible.	`mf`

Itanium memory accesses to **sequential** pages occur in program order with respect to all other sequential pages in the same peripheral domain, but are not necessarily ordered with respect to non-sequential page accesses. A peripheral domain is a platform-specific collection of uncacheable addresses. An I/O device is normally contained in a peripheral domain and all sequential accesses from one processor to that device will be ordered with respect to each other. Sequentiality ensures that uncacheable, non-coalescing memory references from one processor to a peripheral domain reach that domain in program order. Sequentiality does not imply visibility.

Inter-Processor Interrupt Messages (8-byte stores to a Processor Interrupt Block address, through a UC memory attribute) are exceptions to the sequential semantics. IPI's are not ordered with respect to other IPI's directed at the same processor. Further, fence operations do not enforce ordering between two IPI's. See Section 5.8.4.2, “Interrupt and IPI Ordering” on page 2:130.

Table 4-16 defines the ordering between unordered, release, acquire and fence type operations to sequential and non-sequential pages. Table 4-16 defines the minimal ordering requirements; an implementation may enforce more restrictive ordering than required by the architecture. The actual mechanism for enforcing memory access ordering is implementation dependent.

Table 4-16. Ordering Semantics

First Operation		Second Operation						
		Fence	Non-sequential			Sequential[a]		
			Acquire	Release	Unordered	Acquire	Release	Unordered
	Fence	O	O	O	O	O	O	O
Non-sequential	Acquire	O	O	O	O	O	O	O
	Release	O	–	O	–	–	O	–
	Unordered	O	–	O	–	–	O	–
Sequential[a]	Acquire	O	O	O	O	OS	OS	OS
	Release	O	–	O	–	S	OS	S
	Unordered	O	–	O[b]	–[c]	S[d]	OS[e]	S

a. Except for IPI.
b. “O” indicates that the first and second operation become visible in program order.
c. A dash indicates no ordering is implied.
d. “S” indicates that the first and the second operation reach a peripheral domain in program order.
e. “OS” implies that both “O” and “S” ordering relations apply.

Table 4-16 establishes an order between operations on a particular processor. For operations to cacheable write-back memory the order established by these rules is observed by all observers in the coherence domain.

For example, when this sequence is executed on a processor:

```
st [a]
st.rel [b]
```

and a second processor executes this sequence:

```
ld.acq [b]
ld [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

Unless an ordering constraint from Table 4-16 prevents a memory read[1] from becoming visible, the read may be satisfied with values found in a store buffer (or any logically equivalent structure). These values need not be globally visible even when the operation that created the value was a `st.rel`. This local bypassing behavior may make

[1] This includes all types of loads (`ld` and `ld.acq`), and RSE memory reads. Note, however, that the read operation of semaphores cannot be satisfied with values found in a store buffer.

accesses of different sizes but with overlapping memory references appear to complete non-atomically. To ensure that a memory write is globally observed prior to a memory read, software must place an explicit fence operation between the two operations.

Aligned `st.rel` and semaphore operations[1] from multiple processors to cacheable write-back memory become visible to all observers in a single total order (i.e., in a particular interleaving; if it becomes visible to any observer, then it is visible to all observers), except that for `st.rel` each processor may observe (via `ld` or `ld.acq`) its own update prior to it being observed globally.

The Itanium architecture ensures this single total order only for aligned `st.rel` and semaphore operations to cacheable write-back memory. Other memory operations[2] from multiple processors are not required to become visible in any particular order, unless they are constrained w.r.t. each other by the ordering rules defined in Table 4-16.

Ordering of loads is further constrained by data dependency. That is, if one load reads a value written by an earlier load by the same processor (either directly or transitively, through either registers or memory), then the two loads become visible in program order.

For example, when this sequence is executed on a processor:

```
        st [a] = data
        st.rel [b] = a
```

and a second processor executes this sequence:

```
        ld x = [b]
        ld y = [x]
```

if the second processor observes the store to [b], it will also observe the store to [a].

Also for example, when this sequence is executed on a processor:

```
        st [a]
        st.rel [b] = 'new'
```

and a second processor executes this sequence:

```
        ld x = [b]
        cmp.eq p1 = x, 'new'
(p1)    ld y = [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

And for example, when this sequence is executed on a processor:

```
        st [a]
        st.rel [b] = 'new'
```

and a second processor executes this sequence:

[1] Both acquire and release semaphore forms

[2] e.g. unordered stores, loads, `ld.acq`, or memory operations to pages with attributes other than write-back cacheable.

```
        ld x = [b]
        cmp.eq p1 = x, 'new'
(p1)    br target
        ...
target:
        ld y = [a]
```

if the second processor observes the store to [b], it will also observe the store to [a].

The flush cache (fc, fc.i) instruction follows data dependency ordering. fc and fc.i are ordered only with respect to previous and subsequent load, store, or semaphore instructions to the same line, regardless of the specified memory attribute. Subsequent memory operations to the same line need not wait for prior fc or fc.i completion before being globally visible. fc and fc.i are not ordered with respect to memory operations to different lines. mf does not ensure visibility of fc and fc.i operations. Instead, the sync.i instruction synchronizes fc and fc.i instructions, and the sync.i is made visible using an mf instruction.

4.4.8 Not a Thing Attribute (NaTPage)

A NaTPage attribute prevents non-speculative references to a page, and ensures that speculative references to the page always defer the Data NaT Page Consumption fault. However, as described in "Speculation Attributes" on page 2:79, the processor may issue memory references to a NaTPage. As a result, all NaTPages must be backed by a valid physical page.

Speculative or speculative advanced loads to pages marked as a NaTPage cause the deferred exception indicator (NaT or NaTVal) to be written to the load target register, and the memory reference is aborted. However, all other effects of the load instruction such as post-increment are performed. Instruction fetches, loads, stores and semaphores (including IA-32), but except for Itanium speculative loads, pages marked as NaTPage raise a NaT Page Consumption fault.

A speculative reference to a page marked as NaTPage may still take lower priority faults, if not explicitly deferred in the DCR. See "Deferral of Speculative Load Faults" on page 2:105.

4.4.9 Effects of Memory Attributes on Memory Reference Instructions

Memory attributes affect the following Itanium instructions.

- ldfe, stfe: Hardware support for 10-byte memory accesses to a page that is neither a cacheable page with write-back write policy nor a NaTPage is optional. On processor implementations that do not support such accesses, an Unsupported Data Reference Fault is raised when an unsupported reference is attempted.

  For extended floating-point loads the fault is delivered only on the normal, advanced, and check load flavors (ldfe, ldfe.a, ldfe.c.nc, ldfe.c.clr). Control speculative flavors of the ldfe instruction that target pages that are not cacheable with write-back policy always defer the fault. Refer to "Deferral of Speculative Load Faults" on page 2:105 for details.

- cmpxchg and xchg: These instructions are only supported to cacheable pages with write-back write policy. cmpxchg and xchg accesses to NaTPages causes a Data NaT

Page Consumption fault. cmpxchg and xchg accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

- fetchadd: The fetchadd instruction can be executed successfully only if the access is to a cacheable page with write-back write policy or to a UCE page. fetchadd accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault. When accessing a cacheable page with write-back write policy, atomic fetch and add operation is ensured by the processor cache-coherence protocol. For highly contended semaphores, the cache line transactions required to guarantee atomicity can limit performance. In such cases, a centralized "fetch and add" semaphore mechanism may improve performance. If supported by the processor and the platform, the UCE attribute allows the processor to "export" the fetchadd operation to the platform as an atomic "fetch and add." Effects of the exported fetchadd are platform dependent. If exporting of fetchadd instructions is not supported by the processor, a fetchadd instruction to a UCE page takes an Unsupported Data Reference fault.
- Flush Cache Instructions – fc instructions must always be "broadcast" to other processors, independent of the memory attribute in the local processor. It is legal to use an uncacheable memory attribute for any valid address when used as a flush cache (fc) instruction target. This behavior is required to enable transitions from one memory attribute to another and in case different memory attributes are associated with the address in another processor.
- Prefetch instructions – lfetch and any implicit prefetches to pages that are not cacheable are suppressed. No transaction is initiated. This allows programs to issue prefetch instructions even if the program is not sure the memory is cacheable.

4.4.10 Effects of Memory Attributes on Advanced/Check Loads

The ALAT behavior of advanced and check loads is dependent on the memory attribute of the page referenced by the load. These behaviors are required; advanced and check load completers are not hints.

All speculative pages have identical behavior with respect to the ALAT. Advanced loads to speculative pages always allocate an ALAT entry for the register, size, and address tuple specified by the advanced load. Speculative advanced loads allocate an ALAT entry if the speculative load is successful (i.e., no deferred exception); if the speculative advanced load results in a deferred exception, any matching ALAT entry is removed and no new ALAT entry is allocated. Check loads with clear completers (ld.c.clr, ld.c.clr.acq, ldf.c.clr) remove a matching ALAT entry on ALAT hit and do not change the state of the ALAT on ALAT miss. Check loads with no-clear completers (ld.c.nc, ldf.c.nc) allocate an ALAT entry on ALAT miss. On ALAT hit, the ALAT is unchanged if an exact ALAT match is found (register, address, and size); a new ALAT entry with the register, address, and size specified by the no-clear check load may be allocated if a partial ALAT match is found (match on register).

Advanced loads (speculative or non-speculative variants) to non-speculative pages always remove any matching ALAT entry. Check loads to non-speculative pages that miss the ALAT never allocate an ALAT entry, even in the case of a no-clear check load. ALAT hits on check loads to non-speculative pages (which can occur if a previous advanced load referenced that page via a speculative memory attribute) result in undefined behavior; when changing an existing page from speculative to non-speculative (or vice-versa), software should ensure that any ALAT entries corresponding to that page are invalidated.

Limited speculation pages behave like non-speculative pages with respect to speculative advanced loads, and behave like speculative pages with respect to all other advanced and/or check loads.

Table 4-17 describes the ALAT behavior of advanced and check loads for the different speculation memory attributes.

Table 4-17. ALAT Behavior on Non-faulting Advanced/Check Loads

Memory Attribute	ld.sa Response		ld.a Response	ld.c.clr, ld.c.clr.acq, ldf.c.clr Response		ld.c.nc, ldf.c.nc Response	
	No NaT	NaT		ALAT Hit	ALAT Miss	ALAT Hit	ALAT Miss
speculative	alloc	remove	alloc	remove	nop	unchanged[a]	alloc
non-speculative	N/A	remove	remove	undefined	nop	undefined	must not alloc
limited speculation	N/A	remove	alloc	remove	nop	unchanged[a]	alloc

[a] May allocate a new ALAT entry if size and/or address are different than the corresponding ld.a or ld.sa whose ALAT entry was matched.

4.4.11 Memory Attribute Transition

If software modifies the memory attributes for a page, it must perform explicit actions to ensure that subsequent reads and writes using the new attribute will be coherent with prior reads and writes that were performed with the old attribute. Processors may have separate buffers for coalescing, uncacheable and cacheable references, and these buffers need not be coherent with each other.

4.4.11.1 Virtual Addressing Memory Attribute Transition

To change a virtually-addressed page from one attribute to another, software must perform the following sequence. (The address of the page whose attribute is being modified is referred to as "X").

Note: This sequence is ONLY required if the new mapping and the old mapping do not have the same memory attribute.

On the processor initiating the transition, perform the following steps 1-3:

1.
```
PTE[X].p = 0 // Mark page as not present
```

   This prevents any processors from reading the old mapping (with the old attribute) from the VHPT after this point.

2.
```
ptc.ga [X] ;; // Global shootdown and ALAT invalidate
               // for the entire page
```

   This removes the mapping from all processor TC's in the coherence domain, and it forces all processors to flush any pending WC or UC stores from write buffers.

3.
```
mf ;;     // Ensure visibility of ptc.ga to local data stream
srlz.i ;; // Ensure visibility of ptc.ga to local instruction stream
```

   After step 3, no processor in the coherence domain will initiate new memory references or prefetches to the old translation. Note, however, that memory references or prefetches initiated to the old translation prior to step 2 may still be in progress after step 3. These outstanding memory references and prefetches may return instructions or data which may be placed in the processor cache hierarchy; this behavior is implementation-specific.

   If the new memory attribute is an uncacheable attribute, and if the old attribute was cacheable (or if it is not known at this point in the code sequence what the old attribute was), then software must drain any current prefetches and ensure that any cached data from the page is removed from caches. To do this, software must perform steps 4-10. If the new memory attribute is cacheable, then software may skip steps 4-10, and go straight to step 11.

4. Call PAL_PREFETCH_VISIBILITY

   Call PAL_PREFETCH_VISIBILITY with the input argument *trans_type* equal to zero to indicate that the transition is for virtual memory attributes. The return argument from this procedure informs the caller if this procedure call is needed on remote processors or not. If this procedure call is not needed on remote processors, then software may skip the IPI in step 5 and go straight to step 6 below.

5. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherence domain, perform step 4 above on all processors in the coherence domain, and wait for all PAL_PREFETCH_VISIBILITY calls to complete on all processors in the coherence domain before continuing.

   After steps 4 and 5, no more new instruction or data prefetches will be made to page "X" by any processor in the coherence domain. However, processor caches in the coherence domain may still contain "stale" data or instructions from prior prefetch or memory references to page "X."

6. Insert a temporary UC translation for page "X."

7.
```
fc [X] // flush all processor caches in the coherence domain
fc [X+32]
fc [X+64]
... // ... for all of page "X" (page size = ps)
fc [X+ps-32] ;;

// Ensure cache flushes are also seen by processors' instruction
fetch
sync.i ;;
```

   After step 7, all flush cache instructions initiated in step 7 are visible to all processors in the coherence domain, i.e., no processor in the coherence domain will respond with a cache hit on a memory reference to an address belonging to page "X."

8. Purge the temporary UC translation from the TLB

9. Call PAL_MC_DRAIN
10. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherence domain, perform step 9 above on all processors in the coherence domain, and wait for all PAL_MC_DRAIN calls to complete on all processors in the coherence domain before continuing.

    This further guarantees that any cache lines containing addresses belonging to page [X] have been evicted from all caches in the coherence domain and forced onto the bus. Note that this operation does not ensure that the cache lines have been written back to memory.
11. Insert the new mapping with the new memory attribute

4.4.11.2 Physical Addressing Attribute Transition – Disabling Prefetch/Speculation and Removing Cacheability

When a verified reference is made to a physical address with the WBL attribute, the 4K page containing that address becomes speculatively accessible. This allows the processor that made the verified reference to subsequently make speculative references to this page. (See the description of limited speculation in Section 4.4.6.1, "Limited Speculation and the WBL Physical Addressing Attribute" on page 2:81.)

If the same physical memory is then to be accessed with the UC attribute, software must first cause all such 4K pages to no longer be verified pages and flush any cached copies from the cache. Otherwise, an uncacheable reference may hit in cache, causing a Machine Check abort.

On the processor initiating the transition, perform the following steps:

1. Call PAL_PREFETCH_VISIBILITY

   Call PAL_PREFETCH_VISIBILITY with the input argument *trans_type* equal to one to indicate that the transition is for physical memory attributes. This PAL call terminates the processor's rights to make speculative references to any limited speculation pages (i.e., it causes all WBL pages to no longer be verified pages – see the discussion on limited speculation in Section 4.4.6.1.)

   The return argument from this procedure informs the caller if this procedure call is needed on remote processors or not. If this procedure call is not needed on remote processors, then software may skip the IPI in step 2 and go straight to step 3 below.
2. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherence domain, perform step 1 above on all processors in the coherence domain, and wait for all PAL_PREFETCH_VISIBILITY calls to complete on all processors in the coherence domain before continuing.

   On the processor initiating the disabling process, continue the sequence:

```
3.  fc [X] // flush all processor caches in the coherence domain
    fc [X+32]
    fc [X+64]
    ... // ... for all of page "X" (page size = ps)
    fc [X+ps-32] ;;
```

```
// Ensure cache flushes are also seen by processors' instruction
fetch
sync.i ;;
```

After step 3, all flush cache instructions initiated in step 3 are visible to all processors in the coherence domain, i.e., no processor in the coherence domain will respond with a cache line hit on a memory reference to an address belonging to page "X."

4. Call PAL_MC_DRAIN.
5. Using the IPI mechanism defined in "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherence domain, perform step 4 above on all processors in the coherence domain, and wait for all PAL_MC_DRAIN calls to complete on all processors in the coherence domain before continuing.

   This further guarantees that any cache lines containing addresses belonging to page [X] have been evicted from all caches in the coherence domain and forced onto the bus. Note that this operation does not ensure that the cache lines have been written back to memory.

This sequence ensures that speculation and prefetch are disabled for all WBL pages, that all outstanding prefetches have completed, and that the caches have been flushed. It may also be necessary to take additional platform-dependent steps to ensure that all cache write-back transactions have completed to memory before re-configuring physical memory.

4.4.11.3 Memory OLD Attribute Transition Sequence

In order to safely delete a memory range online (memory OLD), all speculative reference and prefetches to that range must be halted and all cache lines returned to the memory being deleted. If this is not done, an MCA could occur if data were to be delivered back to the memory controller after the memory had been removed. Software must perform the sequence shown below to ensure that no MCAs occur.

Before performing the memory OLD sequence shown below, all memory in the range being deleted belonging to firmware (PAL and SAL) must be evacuated, and control of the range given to the OS. If firmware cannot be evacuated from the range, then OLD cannot be done.

On the processor performing the memory OLD operation, perform the following:

1. Remove all mappings to all memory pages in this memory range from the page table. (PTE[X].p=0)
2. For each page which has a mapping in TLB, perform one of the following steps:
   a. If there are any translations in TRs, perform `ptr.d` or `ptr.i`, depending on whether the translation is for code or data. If it is not known, do both. (This invalidates all TRs, and as a side effect, the mapping from all TCs on the processor.)
   b. If there are no translations in TRs, perform a `ptc.ga`. (This removes mapping from all TC's and forces processors to flush any pending WC or UC stores from write buffers.)

3. Execute:

   ```
   mf ;;
   srlz.i ;;
   ```

   (The ensures visibility of `ptr.d`, `ptr.i`, or `ptc.ga` to both data and instruction stream, so that no new prefetches will be done to the old translations.)

4. Call PAL_PREFETCH_VISIBILITY with the input argument *trans_type* equal to one to indicate that the transition is for all memory attributes. This PAL call terminates the processor's rights to make speculative references to any limited speculation pages (i.e., it causes all WBL pages to no longer be verified pages – see the discussion on limited speculation in Section 4.4.6.1, "Limited Speculation and the WBL Physical Addressing Attribute" on page 2:81.). It also ensure all prefetches in flight have been completed. The return argument from this procedure informs the caller if this procedure call is needed on remote processors or not. If this procedure call is not needed on remote processors, and step 2.b was used above, then software may skip the IPI in step 5 and go straight to step 6 below.

5. If step 2.a was performed, or if the PAL_PREFETCH_VISIBILITY return argument indicated the call must be made on other processors in the coherency domain, then use the IPI mechanism defined in Section 5.8.4.1, "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherency domain. If step 2a was performed, then steps 2 through 4 must be performed on all processors in the coherency domain. Otherwise, only step 4 must be performed. Wait for all PAL_PREFETCH_VISIBILITY calls to complete on all processors in the coherency domain before continuing. After step 5, no more new instruction or data prefetches will be made to page "X" by any processor in the coherency domain. However, processor caches in the coherency domain may still contain "stale" data or instructions from prior prefetch or memory references to page "X."

6. Perform one of the following steps:

   a. Call PAL_CACHE_FLUSH with input parameters *cache_type*=3 and *operation.inv*=1, or

   b. On the processor where the OLD was initiated, perform the sequence:

      i. If the sequence is to be executed with PSR.dt=1, then insert a temporary translation for the memory range with the "UC" memory attribute.

      ii. Execute the following instruction sequence:

      ```
      fc [X] // flush all processor caches in the coherence domain
      fc [X+32]
      fc [X+64]
      ... // ... for the memory range being OLDed
      fc [X+ps-32] ;;
      // Ensure cache flushes are also seen
      // by processors' instruction fetch
      sync.i ;;
      ```

      iii. If the sequence had been run with PSR.dt=1, then remove the temporary translation inserted in step 6.b.i.

   Note: If the memory range being OLDed is much larger than the caches being flushed, option 6.a. may be significantly faster.

7. Call PAL_MC_DRAIN.

8. If PAL_CACHE_FLUSH is used to flush caches, it must also be called on all processors in the coherency domain. In any case, PAL_MC_DRAIN must be called on all processors. Using the IPI mechanism defined in Section 5.8.4.1, "Inter-processor Interrupt Messages" on page 2:128 to reach all processors in the coherence domain, perform step 6.a, if necessary, and step 7 above in that order on all processors in the coherence domain, and wait for all PAL_MC_DRAIN calls to complete on all processors in the coherence domain before continuing. This further guarantees that any cache lines containing addresses belonging to page [X] have been evicted from all caches in the coherence domain and forced onto the platform fabric. Note that this operation does not ensure that the cache lines have been written back to memory.
9. Perform whatever platform dependent actions are necessary to flush any platform caches of any copies of the memory being OLDed and to force all cache lines back to the memory being OLDed. (Note: Refer to platform specific documentation.)

This sequence ensures that speculation and prefetching is disabled for the memory range, regardless of WB or WBL attribute, that all in-flight prefetches are completed, and that all caches lines are returned to memory.

4.5 Memory Datum Alignment and Atomicity

All Itanium instruction fetches, aligned load, store and semaphore operations (including IA-32) are atomic, except for floating-point extended memory references (`ldfe`, `stfe`, and IA-32 10-byte memory references) to non-write-back cacheable memory. In some processor models, aligned 10-byte Itanium floating-point extended memory references to non-write-back cacheable memory may raise an Unsupported Data Reference fault. See "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86 for details. Loads are allowed to be satisfied with values obtained from a store buffer (or any logically equivalent structure) where architectural ordering permits, and values loaded may appear to be non-atomic. For details, refer to "Sequentiality Attribute and Ordering" on page 2:82.

Load pair instructions are performed atomically under the following conditions: a 16-byte aligned load integer/double pair is performed as an atomic 16-byte memory reference. An 8-byte aligned load single pair is performed as an atomic 8-byte memory reference.

An aligned `ld16` or `st16` instruction is performed as an atomic 16-byte memory reference. For these instructions, the address specified must be 16-byte aligned. Unaligned `ld16` and `st16` instructions result in an Unaligned Data Reference fault regardless of the state of PSR.ac.

Aligned Itanium data memory references never raise an Unaligned Data Reference fault. Minimally, each Itanium instruction and its corresponding template are fetched together atomically. Itanium unordered loads can use the store buffer for data values. See "Sequentiality Attribute and Ordering" on page 2:82 for details.

When PSR.ac is 1, any Itanium data memory reference that is not aligned on a boundary the size of the operand results in an Unaligned Data Reference fault; e.g., 1, 2, 4, 8, 10, and 16-byte datums should be aligned on 1, 2, 4, 8, 16, and 16-byte boundaries respectively to avoid generation of an Unaligned Data Reference fault. When PSR.ac is 1, any IA-32 data memory reference that is not aligned on a boundary the size of the operand results in an IA_32_Exception(AlignmentCheck) fault.

Note: 10-byte and floating-point load double pair datum alignment is 16-bytes. The alignment of long format 32-byte VHPT references is always 32-bytes.

Unaligned Itanium semaphore references (`cmpxchg`, `xchg`, `fetchadd`) result in an Unaligned Data Reference fault regardless of the state of PSR.ac. For the `cmp8xchg16` instruction, the address specified must be 8-byte aligned.

When PSR.ac is 0, Itanium data memory references that are not aligned may or may not result in an Unaligned Data Reference fault based on the implementation. The level of unaligned memory support is implementation specific. However, all implementations will raise an Unaligned Data Reference fault if the datum referenced by an Itanium instruction spans a 4K aligned boundary, and many implementations will raise an Unaligned Data Reference fault if the datum spans a cache line. Implementations may also raise an Unaligned Data Reference fault for any other unaligned Itanium memory reference. Software is strongly encouraged to align data values to avoid possible performance degradation for both IA-32 and Itanium architecture-based code. When PSR.ac is 0 and IA-32 alignment checks are also disabled, no fault is raised regardless of alignment for IA-32 data memory references.

Unaligned advanced loads are supported, though a particular implementation may choose not to allocate an ALAT entry for an unaligned advanced load. Additionally, the ALAT may “pessimistically” allocate an entry for an unaligned load by allocating a larger entry than the natural size of the datum being loaded, as long as the larger entry completely covers the unaligned address range (e.g. a `ld4.a` to address 0x3 may allocate an 8-byte entry starting at address 0x0). Stores (unaligned or otherwise) may also pessimistically invalidate unaligned ALAT entries.

§

Interruptions 5

Interruptions are events that occur during instruction processing, causing the flow control to be passed to an interruption handling routine. In the process, certain processor state is saved automatically by the processor. Upon completion of interruption processing, a return from interruption (`rfi`) is executed which restores the saved processor state. Execution then proceeds with the interrupted instruction.

From the viewpoint of response to interruptions, the processor behaves as if it were not pipelined. That is, it behaves as if a single Itanium instruction (along with its template) is fetched and then executed; or as if a single IA-32 instruction is fetched and then executed. Any interruption conditions raised by the execution of an instruction are handled at execution time, in sequential instruction order. If there are no interruptions, the next Itanium instruction and its template, or the next IA-32 instruction, are fetched.

This chapter describes both the IA-32 and Itanium interruption mechanisms as well as the interactions between them. The descriptions of the Itanium interruption vectors and IA-32 exceptions, interruptions, and intercepts are in Chapter 8.

5.1 Interruption Definitions

Depending on how an interruption is serviced, interruptions are divided into: IVA-based interruptions and PAL-based interruptions.

- **IVA-based interruptions** are serviced by the operating system. IVA-based interruptions are vectored to the Interruption Vector Table (IVT) pointed to by CR2, the IVA control register (see "IVA-based Interruption Vectors" on page 2:113).
- **PAL-based interruptions** are serviced by PAL firmware, system firmware, and possibly the operating system. PAL-based interruptions are vectored through a set of hardware entry points directly into PAL firmware (see Chapter 11, "Processor Abstraction Layer").

Interruptions are divided into four types: Aborts, Interrupts, Faults, and Traps.

- **Aborts**
  A processor has detected a Machine Check (internal malfunction), or a processor reset. Aborts can be either synchronous or asynchronous with respect to the instruction stream. The abort may cause the processor to suspend the instruction stream at an unpredictable location with partially updated register or memory state. Aborts are PAL-based interruptions.
  - **Machine Checks (MCA)**
    A processor has detected a hardware error which requires immediate action. Based on the type and severity of the error the processor may be able to recover from the error and continue execution. The PALE_CHECK entry point is entered to attempt to correct the error.
  - **Processor Reset (RESET)**
    A processor has been powered-on or a reset request has been sent to it. The

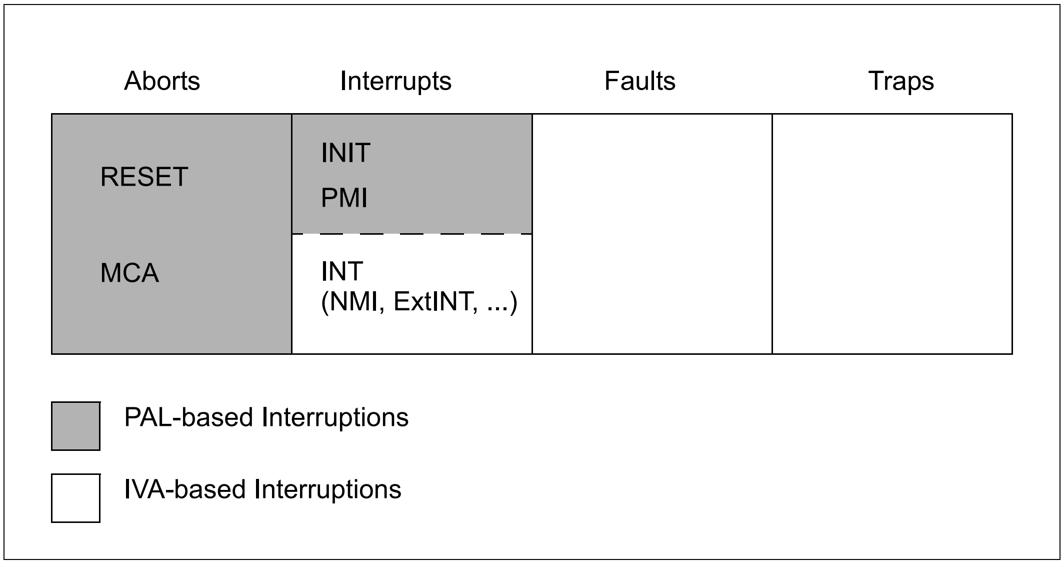
PALE_RESET entry point is entered to perform processor and system self-test and initialization.

- **Interrupts**
  An external or independent entity (e.g., an I/O device, a timer event, or another processor) requires attention. Interrupts are asynchronous with respect to the instruction stream. All previous instructions (including IA-32) appear to have completed. The current and subsequent instructions have no effect on machine state. Interrupts are divided into Initialization interrupts, Platform Management interrupts, and External interrupts. Initialization and Platform Management interrupts are PAL-based interruptions; external interrupts are IVA-based interruptions.
  - **Initialization Interrupts (INIT)**
    A processor has received an initialization request. The PALE_INIT entry point is entered and the processor is placed in a known state.
  - **Platform Management Interrupts (PMI)**
    A platform management request to perform functions such as platform error handling, memory scrubbing, or power management has been received by a processor. The PALE_PMI entry point is entered to service the request. Program execution may be resumed at the point of interruption. PMIs are distinguished by unique vector numbers. Vectors 0 through 3 are available for platform firmware use and are present on every processor model. Vectors 4 through 15 are reserved for processor firmware use. See Section 11.5, "Platform Management Interrupt (PMI)" on page 2:310 for details.
  - **External Interrupts (INT)**
    A processor has received a request to perform a service on behalf of the operating system. Typically these requests come from I/O devices, although the requests could come from any processor in the system including itself. The External Interrupt vector is entered to handle the request. External Interrupts are distinguished by unique vector numbers in the range 0, 2, and 16 through 255. These vector numbers are used to prioritize external interrupts. Two special cases of External Interrupts are Non-Maskable Interrupts and External Controller Interrupts.
    - **Non-Maskable Interrupts (NMI)**
      Non-Maskable Interrupts are used to request critical operating system services. NMIs are assigned external interrupt vector number 2.
    - **External Controller Interrupts (ExtINT)**
      External Controller Interrupts are used to service Intel 8259A-compatible external interrupt controllers. ExtINTs are assigned locally within the processor to external interrupt vector number 0.
- **Faults**
  The current Itanium or IA-32 instruction which requests an action which cannot or should not be carried out, or system intervention is required before the instruction is executed. Faults are synchronous with respect to the instruction stream. The processor completes state changes that have occurred in instructions prior to the faulting instruction. The faulting and subsequent instructions have no effect on machine state. Faults are IVA-based interruptions.
- **Traps**
  The IA-32 or Itanium instruction just executed requires system intervention. Traps are synchronous with respect to the instruction stream. The trapping instruction

and all previous instructions are completed. Subsequent instructions have no effect on machine state. Traps are IVA-based interruptions.

Figure 5-1 summarizes the above classification.

Figure 5-1. Interruption Classification



Unless otherwise indicated, the term “interruptions” in the rest of this chapter refers to IVA-based interruptions. PAL-based interruptions are described in detail in Chapter 11.

5.2 Interruption Programming Model

When an interruption event occurs, hardware saves the minimum processor state required to enable software to resolve the event and continue. The state saved by hardware is held in a set of interruption resources, and together with the interruption vector gives software enough information to either resolve the cause of the interruption, or surface the event to a higher level of the operating system. Software has complete control over the structure of the information communicated, and the conventions between the low-level handlers and the high-level code. Such a scheme allows software rather than hardware to dictate how to best optimize performance for each of the interruptions in its environment. The same basic mechanisms are used in all interruptions to support efficient low-level fault handlers for events such as a TLB fault, speculation fault, or a key miss fault.

On an interruption, the state of the processor is saved to allow a software handler to resolve the interruption with minimal bookkeeping or overhead. The banked general registers (see “Efficient Interruption Handling” on page 2:102) provide an immediate set of scratch registers to begin work. For low-level handlers (e.g., TLB miss) software need not open up register space by spilling registers to either memory or control registers.

Upon an interruption, asynchronous events such as external interrupt delivery are disabled automatically by hardware to allow software to either handle the interruption immediately or to safely unload the interruption resources and save them to memory. Software will either deal with the cause of the interruption and `rfi` back to the point of the interruption, or it will establish a new environment and spill processor state to memory to prepare for a call to higher-level code. Once enough state has been saved (such as the IIP, IPSR, and the interruption resources needed to resolve the fault) the low-level code can re-enable interruptions by restoring the PSR.ic bit and then the PSR.i bit. (See "Re-enabling External Interrupt Delivery" on page 2:120.) Since there is only one set of interruption resources, software must save any interruption resource state the operating system may require prior to unmasking interrupts or performing an operation that may raise a synchronous interruption (such as a memory reference that may cause a TLB miss).

The PSR.ic (interruption state collection) bit supports an efficient nested interruption model. Under normal circumstances the PSR.ic bit is enabled. When an interruption event occurs, the various interruption resources are overwritten with information pertaining to the current event. Prior to saving the current set of interruption resources, it is often advantageous in a miss handler to perform a virtual reference to an area which may not have a translation. To prevent the current set of resources from being overwritten on a nested fault, the PSR.ic bit is cleared on any interruption. This will suppress the writing of critical interruption resources if another interruption occurs while the PSR.ic bit is cleared. If a data TLB miss occurs while the PSR.ic bit is zero, then hardware will vector to the Data Nested TLB fault handler.

For a complete description of interruption resources (IFA, IIP, IPSR, ISR, IIM, IIPA, ITIR, IHA, IFS, IIB0-1) see "Control Registers" on page 2:29.

5.3 Interruption Handling during Instruction Execution

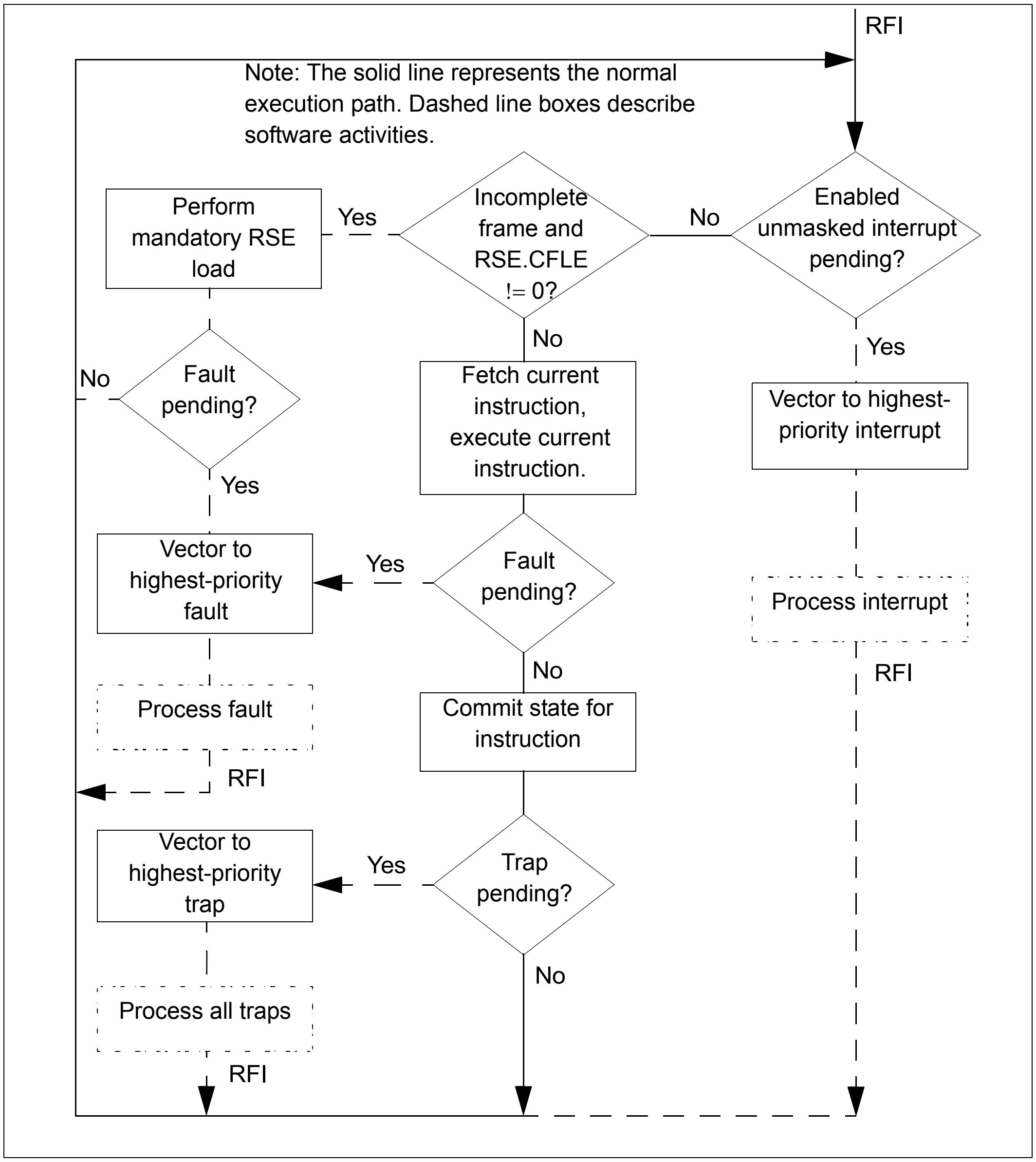
Execution of Itanium instructions involves calculating the address of the current bundle from the region registers and the IP and then fetching, decoding, and executing instructions in that bundle. Execution of IA-32 instructions involves calculating the 64-bit linear address of the current instruction from the EIP, code segment descriptors, and region registers and then fetching, decoding, and executing the IA-32 instruction. (See Section 3.4).

The execution process involves performing the events listed below. The values of the PSR bits are the values that exist before the instruction is executed (except for the case of instructions that are immediately preceded by a mandatory RSE load which clears the PSR.da and PSR.dd bits). Changes to the PSR bits only affect subsequent instructions, and are only guaranteed to be visible by the insertion of the appropriate serializing operation. See "Serialization" on page 2:17. Execution flow is shown in Figure 5-2.

1. Resets are always enabled, and may occur anytime during instruction execution.
2. If the PSR.mc bit is 0 then machine check aborts may occur.
3. The processor checks for enabled pending INITs and PMIs, and for enabled unmasked pending external interrupts.

4. For Itanium architecture-based code, the processor checks for a valid register stack frame.
   - If incomplete and RSE Current Frame Load Enable (RSE.CFLE) is set, then perform a mandatory RSE load and start again at step one. The mandatory load operation may fault. A non-faulting mandatory RSE load will clear PSR.da and PSR.dd.
   - If valid, then clear RSE.CFLE.
5. If the processor implements the Unimplemented Instruction Address (UIA) fault, instead of a UIA trap, it will check the instruction address and take the UIA fault if the instruction pointer (IP) falls outside of the implemented range.

Figure 5-2. Interruption Processing


RFI
Note: The solid line represents the normal execution path. Dashed line boxes describe software activities.
Perform mandatory RSE load
Yes
Incomplete frame and RSE.CFLE != 0?
No
Enabled unmasked interrupt pending?
No
Fault pending?
No
Yes
Fetch current instruction, execute current instruction.
Yes
Vector to highest-priority interrupt
Vector to highest-priority fault
Yes
Fault pending?
Process interrupt
No
RFI
Process fault
Commit state for instruction
RFI
Vector to highest-priority trap
Yes
Trap pending?
No
Process all traps
RFI

6. For IA-32 code, IA-32 instruction addresses are checked for possible instruction

breakpoint faults. The IA-32 effective instruction address (EIP) is converted into a 64-bit virtual linear address IP and IA-32 defined code segmentation and code fetch faults are checked and may result in a fault.

7. When PSR.is is 0, the bundle is fetched using the IP. When PSR.is is 1, an IA-32 instruction is fetched using IP.
   - If the PSR.it bit is 1, virtual address translation of the instruction address is performed. Address translation may result in a fault.
   - If the PSR.pk bit is 1, access key checking is enabled and may result in a fault.
   - For Itanium instructions the IBR registers are checked for possible instruction breakpoint faults.
   - The fetched instruction is decoded and executed.
   - For IA-32 code, the fetched IA-32 instruction is checked to see if the opcode is an illegal opcode, results in an instruction intercept or the opcode bytes are longer than 15 bytes resulting in an fault.
   - If a fault occurs during execution, the processor completes all effects of the instructions prior to the faulting instruction, and does not commit the effect of the faulting instruction and all subsequent instructions. It then takes the interruption for the fault. IIP is loaded with the IP of the bundle or IA-32 instruction which contains the instruction that caused the fault.
   - The PSR.dd, PSR.id, PSR.ia, PSR.da, and PSR.ed bits are set to 0 after an Itanium instruction is successfully executed without raising a fault. The PSR.da and PSR.dd bits are also set to 0 after the execution of each mandatory RSE memory reference that does not raise a fault. PSR.da, PSR.ia, PSR.dd, and PSR.ed bits are cleared before the first IA-32 instruction starts execution after a `br.ia` or `rfi` instruction. EFLAG.rf and PSR.id bits are set to 0 after an IA-32 instruction is successfully executed.
   - If an `rfi` instruction is in the current bundle, then on the execution of `rfi`, the value from the IIP is copied into the IP, the value from IPSR is copied into the PSR, and the RSE.CFLE is set. On an `rfi` if IFS.v is set, then IFS.pfm is copied into CFM and the register stack BOF is decremented by CFM.sof. The following Itanium or IA-32 instruction is executed based on the new IP and PSR values.

8. Traps are handled after execution is complete.
   - If the processor reports unimplemented instruction addresses with an Unimplemented Instruction Address trap (rather than with an Unimplemented Instruction Address fault) and the instruction just completed set the instruction pointer (IP) to an unimplemented address, an Unimplemented Instruction Address trap is taken.
   - If the instruction just completed was an Itanium floating-point instruction which raised a trap, a Floating-point trap is taken.
   - For IA-32 instructions, if Data Breakpoint traps are enabled and one or more data breakpoint registers matched during execution of the instruction, a Data Breakpoint trap is taken.
   - If the PSR.lp bit is 1, and an Itanium branch lowers the privilege level, then a Lower-Privilege Transfer trap is taken.
   - If the PSR.tb bit is 1 and a branch (including IA-32) occurred during execution, then a Taken Branch trap occurs.
   - If no other trap was taken and the PSR.ss bit is 1, then a Single Step trap occurs.

- If more than one trap is triggered (such as Unimplemented Instruction Address trap, Lower-Privilege Transfer trap, and Single Step trap) the highest priority trap is taken. The ISR.code contains a bit vector with one bit set for each trap triggered.

A sequential execution model is presented in the preceding description. Implementations are free to use a variety of performance techniques such as pipelined, speculative, or out-of-order execution provided that, to the programmer, the illusion that instructions are executed sequentially is preserved.

5.4 PAL-based Interruption Handling

PAL-based interruption handling requires the processor to transfer control to the PAL firmware. The PAL firmware will execute handling code and set up the architected exit state before transferring control to the SAL firmware. See Chapter 11, "Processor Abstraction Layer" for more details on the architected exit state between the PAL and SAL firmware layers for PAL-based interruption handling.

It is strongly recommended that software ensure that, if machine check aborts are masked (PSR.mc), external interrupts are also masked (PSR.i). This will avoid cases where a corrected machine check interrupt (a lower priority interrupt) is handled before a machine check abort, which would cause an escalation in machine check abort severity when machine check aborts are unmasked.

5.5 IVA-based Interruption Handling

IVA-based interruption handling is implemented as a fast context switch. On IVA-based interruptions, instruction and data translation is left unchanged, the endian mode is set to the system default, and delivery of most PSR-controlled interruptions is disabled (including delivery of asynchronous events such as external interrupts). The processor is responsible for saving only a minimal amount of state in the interruption resource registers prior to vectoring to the Itanium architecture-based software handler.

When an interruption occurs, the processor takes the following actions:

1. If PSR.ic is 0:
   - IPSR, IIP, IIPA, IIB0-1, and IFS.v are unchanged.
   - Interruption-specific resources IFA, IIM, and IHA are unchanged.

   If PSR.ic is 1:

   - PSR is saved in IPSR. If PSR is in-flight, IPSR will get the most recent in-flight value of PSR (i.e., PSR is serialized by the processor before it is written into IPSR). For Itanium traps, the value written to IPSR.ri is the next instruction slot that would have been executed if there had been no trap. For all other interruptions, the value written to IPSR.ri is the instruction slot on which the interruption occurred (1 for interruptions on the L+X instruction of an MLX). For interruptions in the IA-32 instruction set, IPSR.ri is set to 0.
   - IP is written into IIP. For faults and external interrupts, the saved IP is the IP at which the interruption occurred. For traps, the saved IP is the value after the execution of the IA-32 or Itanium instruction which caused the trap. For

branch-related traps, IIP is written with the target of the branch; for all other traps, IIP is written with the address of the bundle or IA-32 instruction containing the next sequential instruction.

- IIPA receives the IP of the last successfully executed Itanium instruction. For IA-32 instructions, IIPA receives the IP of the faulting or trapping IA-32 instruction.
- The interruption resources IFA, IIB0-1, IIM, IHA, and ITIR are written with information specific to the particular fault, trap, or interruption taken. These registers serve as parameters to each of the interruption vectors. The IFS valid bit (IFS.v) is cleared. All other bits in the IFS are undefined.

If PSR.ic is in-flight:

- Interruption state may or may not be collected in IIP, IPSR, IIPA, ITIR, IFA, IIM, IIB0-1 and IHA.
- The value of IFS (including IFS.v) is undefined.

2. ISR bits are overwritten on all interruptions except for a Data Nested TLB fault. The instruction slot which caused the interruption is saved in ISR.ei (2 for traps, 1 for other interruptions, on the L+X instruction of an MLX). For IA-32 code, ISR.ei is set to 0. If PSR.ic is 0 or in-flight when the interruption occurs, ISR.ni is set to 1. Otherwise, ISR.ni is set to 0. ISR.ni is always 0 for interruptions taken in IA-32 code.
3. The defined bits in the PSR are set to zero except as follows:
   - PSR.up, PSR.mfl, PSR.mfh, PSR.pk, PSR.dt, PSR.rt, PSR.mc, and PSR.it are unchanged for all interruptions.
   - PSR.be is set to the value of the default endian bit (DCR.be). If DCR.be is in-flight at the time of interruption, PSR.be may receive either the old value of DCR.be or the in-flight value.
   - PSR.pp is set to the value of the default privileged performance monitor bit (DCR.pp). If DCR.pp is in-flight at the time of interruption, PSR.pp may receive either the old value of DCR.pp or the in-flight value.

   Since PSR.cpl is set to zero, the processor will execute at the most privileged level.
4. RSE.CFLE is set to zero.
5. IP gets the appropriate IVA vector for the interruption. If IVA is in-flight at the time of interruption, IP receives either the vector specified by the old IVA value or the vector specified by the in-flight value.
6. The processor performs an instruction serialization and execution of Itanium instructions begins at the IP obtained in step 5 above. The instruction serialization event ensures that all previous control register changes and side effects due to such changes are visible to the first instruction of the interruption handler.

5.5.1 Efficient Interruption Handling

A set of 16 banked registers are provided by the processor to assist in the efficient processing of low-level Itanium interruptions and instruction emulation. These registers allow a low-level routine to have immediate access to a small set of static registers without having to save and restore their contents to memory at the start and end of each handler. The extra bank of registers exists in the same name space as the normal registers, overlapping GR16 to GR31. Which set of physical registers are accessed through GR16 to GR31 is determined by the PSR.bn bit. On an interruption this bit is forced to zero allowing access to the alternate set of 16 registers which can be used as scratch space or to hold predetermined values. Software can return to the original set of 16 GRs by setting the PSR.bn bit to one with bsw instruction. The rfi instruction may also restore the PSR.bn bit to the value at the time of the interruption which is held in the IPSR. Eight additional registers (KR0-KR7) can be used to hold latency critical information for a handler. These application registers (KR0-KR7) can be read but not written by non-privileged code.

When the processor handles an interruption event the current stack frame remains unchanged and the IFS valid bit is cleared. The remaining contents of IFS are undefined. While the interruption handler is running, the register stack engine (RSE) may spill/fill registers to/from the backing store if eager RSE stores/loads are enabled. The RSE will not load or store registers in the current frame (except as required on a br.ret or rfi in order to load the contents of the frame before continuing execution). For most low-level interruptions the current frame will not be modified. High-performance interruption handlers will not need to perform any register stack manipulation. For example, a TLB miss handler does not need access to any registers in the interrupted frame. An rfi instruction after an interruption and before a cover operation will also leave the frame marker unchanged (desired behavior for a low-level interruption handler). When an interruption handler falls off the fast path it is required to issue a cover instruction so that the interrupted frame can become part of backing store. See "Switch from Interrupted Context" on page 2:148..

It may be desirable to emulate a faulting instruction in the interruption handler and rfi back to the next sequential instruction rather than resuming at the faulting instruction. Some Itanium instructions can be emulated without having to read the bundle from memory, through knowledge of the vector, software convention, and information from the ISR (e.g., emulation of tpa). However, most Itanium instructions will require reading the bundle from memory and decoding the operation (e.g., an unaligned load). To correctly emulate an unaligned load, the bundle is read from memory using the value in the IIP which contains the bundle address. The instruction within the bundle that caused the interruption is determined by the ISR.ei field. Once the operation is decoded and emulation completes, the effect of the faulting instruction must be nullified when control is returned to the point of the fault.

An Itanium instruction is skipped by adjusting PSR.ri and possibly IIP prior to performing the rfi to the interrupted bundle. This is done by incrementing IPSR.ri by the number of slots this instruction occupies (usually 1). If the resulting IPSR.ri is 3, then reset IPSR.ri to 0 and advance IIP by 1 bundle (16 bytes). Emulating X-unit instructions requires setting IPSR.ri to 0 and setting IIP to the next bundle (X-unit instructions take up two instruction slots). IPSR, IIP, and IFS.pfm (if valid) will be restored on an rfi to the PSR, IP, and CFM registers.

5.5.2 Non-access Instructions and Interruptions

The non-access Itanium instructions are: fc, fc.i, lfetch, probe, probe.fault, tpa, and tak. These instructions reference the TLB but do not directly read or write memory. They are distinguished from normal load/store instructions since an operating system may wish to handle an interruption raised by a non-access instruction differently.

These non-access Itanium instructions can cause interruptions: fc, fc.i, lfetch.fault, probe, probe.fault, tpa, and tak. (tak can cause interruptions only for non-TLB reasons.) ISR.code will be set to indicate which non-access instruction caused the interruption. See Table 5-1 for ISR field settings for non-access instructions.

Table 5-1. ISR Settings for Non-access Instructions

Instruction	ISR Fields			
	code{3:0}	na	r	w
tpa	0	1	0	0
fc, fc.i	1	1	1	0
probe	2	1	0 or 1[a]	0 or 1[a]
tak	3	1	0	0
lfetch, lfetch.fault	4	1	1	0
probe.fault	5	1	0 or 1[a]	0 or 1[a]

[a] Sets r or w or both to 1 depending on the probe form.

5.5.3 Single Stepping

The processor can single step through a series of instructions by enabling the single step PSR.ss bit. This is accomplished by setting the IPSR.ss bit and performing an rfi back to the instruction to be single stepped over. When single stepping, the processor will execute one IA-32 instruction or one Itanium instruction pointed to by the IPSR.ri field.

After single stepping Itanium instruction slot 2 (IPSR.ri = 2) or when the template is MLX and single stepping instruction slot 1 (IPSR.ri = 1), the IIP will point to the next bundle, and IPSR.ri will point to slot 0.

5.5.4 Single Instruction Fault Suppression

Four bits, PSR.id, PSR.da, PSR.ia, and PSR.dd are defined to suppress faults for one Itanium instruction or one mandatory RSE memory operation. The PSR.id bit is used to suppress the instruction debug fault for one IA-32 or Itanium instruction. This bit will be cleared in the PSR after the first successfully executed instruction. The PSR.ia bit is used to suppress the Instruction Access Bit fault for one Itanium instruction. This bit will be cleared in the PSR after the first successfully executed instruction. The PSR.da and PSR.dd bits are used to suppress Dirty-Bit, Data Access-Bit and Data Debug faults for one Itanium instruction, or for one mandatory RSE memory reference. The PSR.da and PSR.dd bits will be cleared in the PSR after the first instruction is executed without raising a fault, or after the first mandatory RSE memory reference that does not raise a fault completes. PSR.da, PSR.ia and PSR.dd are cleared before the first IA-32 instruction starts execution after a br.ia or rfi instruction. Software may set the PSR.id, PSR.da, PSR.ia and PSR.dd bits in the IPSR prior to an rfi. The rfi will restore the PSR from the IPSR. By using these disable bits, software may step over a debug or dirty/access event and continue execution.

5.5.5 Deferral of Speculative Load Faults

Speculative and speculative advanced loads can defer fault handling by suppressing the speculative memory reference, and by setting the deferred exception indicator (NaT bit or NaTVal) of the load target register. Other effects of the instruction (such as post increment) are performed. Additionally, software can suppress the memory reference of speculative and speculative advanced loads independent of any exception.

Deferral is the process of generating a deferred exception indicator and not performing the exception processing at the time of its detection (and potentially never at all). Once a deferred exception indicator is generated, it will propagate through all uses until the speculation is checked by using either a `chk.s` instruction, a `chk.a` instruction (for speculative advanced loads), or a non-speculative use. This causes the appropriate action to be invoked to deal with the exception.

Three different programming models are supported: **no-recovery**, **recovery** and **always-defer**. In the no-recovery model, only fatal exceptional conditions are deferred – these are conditions which cannot be resolved without either involving the program's exception-handling code or terminating the program. In the recovery model, performance may be increased by deferring additional exceptional conditions. The recovery model is used only if the program provides additional "recovery" code to re-execute failed speculative computations. When a speculative load is executed with PSR.ic equal to 1, and ITLB.ed equal to 0, the no-recovery model is in effect. When PSR.ic is 1 and ITLB.ed is 1, the recovery model is in effect. The **always-defer** model is supported for use in system code which has PSR.ic equal to 0. In this model, all exceptional conditions which can be deferred are deferred. This permits speculation in environments where faulting would be unrecoverable.

In addition to the deferral of exceptional conditions, speculative loads may be deferred automatically by hardware based on implementation-dependent criteria, such as the detection of a cache miss. Such deferral is referred to as **spontaneous deferral**, and is done in order to increase performance. Spontaneous deferral is allowed only in the recovery model.

Table 5-2. Programming Models

PSR.ic	PSR.it	ITLB.ed	Model	DCR-based Deferral	Spontaneous Deferral
0	x	x	Always defer	No	No
1	0	x	No recovery	No	No
1	1	0	No recovery	No	No
1	1	1	Recovery	Yes	Yes

Speculative load exceptions are categorized into three groups:

- Ones which always raise a fault
- Ones which always defer
- Ones which always raise a fault in the no-recovery model, but can defer based on the speculative deferral control bits in the DCR control register, in the recovery model.

Aborts, external interrupts, RSE or instruction-fetch-related faults that happen to occur on a speculative load are always raised (since they are not related to the speculative load instruction). Illegal Operation faults and Disabled Floating-point Register faults that occur on a speculative load are always raised.

Processing of exception conditions for speculative and speculative advanced loads is done in three stages: qualification, deferral and prioritization.

During the execution of a load instruction, multiple exception conditions may be detected simultaneously. For non-speculative loads these exception conditions are prioritized and only the highest priority one raises a fault. For speculative loads, however, some exception conditions may be deferred. As a result, it is possible for lower priority exceptions, which are not also deferred, to raise a fault. For some exception conditions, though, other lower priority conditions are meaningless, and are said to be qualified, or precluded. Exception qualification is described in Table 5-3.

Table 5-3. Exception Qualification

Exception Condition	Precluded by Concurrent Exception Condition	
Register NaT Consumption (NaT'ed address)	none	
Unimplemented Data Address	Register NaT Consumption	
Alternate Data TLB	Register NaT Consumption	Unimplemented Data Address
VHPT data	Register NaT Consumption	Unimplemented Data Address
Data TLB	Register NaT Consumption	Unimplemented Data Address
Data Page Not Present	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB
Data NaT Page Consumption	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Key Miss	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Key Permission	Register NaT Consumption Unimplemented Data Address VHPT data Data TLB	Alternate Data TLB Data Page Not Present Data Key Miss
Data Access Rights	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Access Bit	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present
Data Debug	Register NaT Consumption	Unimplemented Data Address
Unaligned Data Reference	Register NaT Consumption	Unimplemented Data Address
Unsupported Data Reference	Register NaT Consumption Unimplemented Data Address VHPT data	Data TLB Alternate Data TLB Data Page Not Present

After exception conditions are detected and qualified, the remaining exception conditions are checked for deferral. Deferral occurs after fault qualification and determines which memory access exceptions raised by speculative loads are automatically deferred by hardware.

Deferral is controlled by PSR.ed, PSR.it, PSR.ic, the speculative deferral control bits in the DCR, the exception deferral bit of the code page's instruction TLB entry (ITLB.ed), and the memory attribute of the referenced data page. The speculative load and speculative advanced load exception deferral conditions are as follows:

- When PSR.ic is 0 and regardless of the state of DCR, and ITLB.ed bits (see Table 5-2), all exception conditions related to the data reference are deferred.
- Regardless of the state of DCR, PSR.it, PSR.ic, and ITLB.ed bits, Unimplemented Data Address exception conditions and Data NaT Page Consumption exception conditions (caused by references to NaTPages) are always deferred.
- When PSR.it and ITLB.ed are both 1, and the appropriate DCR bit is 1 for the exception, the speculative load exception is deferred.
- When PSR.it and ITLB.ed are both 1, Unaligned Data Reference exception conditions are deferred.

The conditions for deferral are given in Table 5-4. See also "Default Control Register (DCR – CR0)" on page 2:31.

Table 5-4. Qualified Exception Deferral

Qualified Exception	Deferred If
Register NaT Consumption (NaT'ed address)	always
Unimplemented Data Address	always
Alternate Data TLB	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
VHPT data	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
Data TLB	!PSR.ic || (PSR.it && ITLB.ed && DCR.dm)
Data Page Not Present	!PSR.ic || (PSR.it && ITLB.ed && DCR.dp)
Data NaT Page Consumption	always
Data Key Miss	!PSR.ic || (PSR.it && ITLB.ed && DCR.dk)
Data Key Permission	!PSR.ic || (PSR.it && ITLB.ed && DCR.dx)
Data Access Rights	!PSR.ic || (PSR.it && ITLB.ed && DCR.dr)
Data Access Bit	!PSR.ic || (PSR.it && ITLB.ed && DCR.da)
Data Debug	!PSR.ic || (PSR.it && ITLB.ed && DCR.dd)
Unaligned Data Reference	!PSR.ic || (PSR.it && ITLB.ed)
Unsupported Data Reference	always

The conditions for spontaneous deferral are given in Table 5-5. See the PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17) procedure for details on enabling/disabling spontaneous deferral.

Table 5-5. Spontaneous Deferral

Implementation-dependent condition may optionally be deferred if
(PSR.ic && PSR.it && ITLB.ed && spontaneous_deferral_enabled())

After checking for deferral, execution of a speculative load instruction proceeds as follows:

- When PSR.ed is 1, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register, regardless of whether it has an exception or not and regardless of the state of DCR, PSR.it, PSR.ic and the ITLB.ed bits.
- If PSR.ed is 0 and there is at least one exception condition which is neither precluded nor deferred, then a fault is taken corresponding to the highest-priority

exception condition which is neither precluded nor deferred. Prioritization of non-deferred speculative load faults follows the same interruption priorities as non-speculative instruction faults (Table 5-6 on page 2:109). However, deferred speculative load faults do not take part in the prioritization. As a result, depending on DCR settings, a lower priority fault may be taken, even if a higher priority exception condition exists, but is deferred.

- If PSR.ed is 0 and there are exception conditions, but all are either precluded or deferred, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register.
- If PSR.ed is 0, and there are no exception conditions, and if the memory attribute of the referenced page is uncacheable or limited speculation, then a deferred exception indicator (NaT bit or NaTVal) is written to the load target register. See "Speculation Attributes" on page 2:79..
- If PSR.ed is 0, and there are no exception conditions, and if spontaneous deferral is enabled and permitted by the programming model, then a deferred exception indicator (NaT bit or NaTVal) may optionally be written to the load target register.
- Otherwise, the load executes normally.

If automatic hardware deferral is not enabled, software may still choose to defer exception processing (for speculative loads) at the time of the fault. If the code page has its ITLB.ed bit equal to 1, then the operating system may choose to defer a non-fatal exception. It is expected that the operating system will always defer fatal exceptions. To assist software in the deferral of non-fatal or fatal exceptions, the system architecture provides three additional resources: ISR.sp, ISR.ed, and PSR.ed.

ISR.sp indicates whether the exception was the result of a speculative or speculative advanced load. The ISR.ed bit captures the code page ITLB.ed bit, and allows deferral of a non-fatal exception due to a speculative load. If both the ISR.sp and ISR.ed bit are 1 on an interruption, then the operating system may defer a non-fatal exception by using the PSR.ed bit to perform the action of hardware deferral for one executed instruction. Software may use the same PSR.ed mechanism to defer fatal speculative load exceptions.

5.6 Interruption Priorities

Table 5-6 contains a complete list of the architecture defined interruptions (including IA-32), grouped according to type (aborts, interrupts, faults and traps), instruction set, and listed in priority order. Interruptions are delivered in priority order. If more than one instruction detects an interruption within a bundle, the interruption occurring in the lowest numbered instruction slot is raised. Lower priority faults and traps are discarded. Lower priority interrupts are held pending.

The shaded interruptions are disabled if the instruction generating the interruption is predicated off. All other interruptions are either "bundle related" (so the predicate bits do not affect them) or are caused by instructions that cannot be predicated off. Incomplete Register frame (IR) faults 7 through 18 are identical in behavior to faults 45, 51 through 62 (exclusive of 60) except they are of a higher priority. IR faults 7 through 18 can only be caused by mandatory RSE load operations that result from `br.ret`, or `rfi` instructions, but not from `loadrs` instructions (for details see Section 6.6, "RSE Interruptions" on page 2:144).

Table 5-6. Interruption Priorities

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[a]
Aborts	IA-32, Intel Itanium	1 Machine Reset (RESET)	PALE_RESET vector	N/A
		2 Machine Check (MCA)	PALE_CHECK vector	
Interrupts		3 Initialization Interrupt (INIT)	PALE_INIT vector	N/A
		4 Platform Management Interrupt (PMI)	PALE_PMI vector	
		5 External Interrupt (INT)	External Interrupt vector	
		6 Virtual External Interrupt (VINT)	Virtual External Interrupt vector	N/A
Faults	Intel Itanium	7 IR Unimplemented Data Address fault	General Exception vector	N/A
		8 IR Data Nested TLB fault	Data Nested TLB vector	
		9 IR Alternate Data TLB fault	Alternate Data TLB vector	
		10 IR VHPT Data fault	VHPT Translation vector	
		11 IR Data TLB fault	Data TLB vector	
		12 IR Data Page Not Present fault	Page Not Present vector	
		13 IR Data NaT Page Consumption fault	NaT Consumption vector	
		14 IR Data Key Miss fault	Data Key Miss vector	
		15 IR Data Key Permission fault	Key Permission vector	
		16 IR Data Access Rights fault	Data Access Rights vector	
		17 IR Data Access Bit fault	Data Access-Bit vector	
		18 IR Data Debug fault	Debug vector	
		19 Unimplemented Instruction Address fault[b]	Lower-Privilege Transfer Trap vector	
Faults	IA-32	20 IA-32 Instruction Breakpoint fault	IA-32 Exception vector (Debug)	A
		21 IA-32 Code Fetch fault[c]	IA-32 Exception vector (GPFault)	
	IA-32, Intel Itanium	22 Alternate Instruction TLB fault	Alternate Instruction TLB vector	
		23 VHPT Instruction fault	VHPT Translation vector	
		24 Instruction TLB fault	Instruction TLB vector	
		25 Instruction Page Not Present fault	Page Not Present vector	
		26 Instruction NaT Page Consumption fault	NaT Consumption vector	
		27 Instruction Key Miss fault	Instruction Key Miss vector	
		28 Instruction Key Permission fault	Key Permission vector	
		29 Instruction Access Rights fault	Instruction Access Rights vector	
		30 Instruction Access Bit fault	Instruction Access-Bit vector	
	Intel Itanium	31 Instruction Debug fault	Debug vector	
	IA-32	32 IA-32 Instruction Length > 15 bytes	IA-32 Exception vector (GPFault)	B
		33 IA-32 Invalid Opcode fault	IA-32 Intercept vector (Instruction)	
		34 IA-32 Instruction Intercept fault	IA-32 Intercept vector (Instruction)	
	Intel Itanium	35 Illegal Operation fault[d]	General Exception vector	
		36 Illegal Dependency fault	General Exception vector	
		37 Break Instruction fault	Break Instruction vector	
		38 Privileged Operation fault	General Exception vector	

Table 5-6. Interruption Priorities (Continued)

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[a]
	IA-32, Intel Itanium	39 Disabled Floating-point Register fault	Disabled FP-Register vector	B
		40 Disabled Instruction Set Transition fault	General Exception vector	
	IA-32	41 IA-32 Device Not Available fault	IA-32 Exception vector (DNA)	
		42 IA-32 FP Error fault[e]	IA-32 Exception vector (FPError)	C
	IA-32, Intel Itanium	43 Register NaT Consumption fault	NaT Consumption vector	
	Intel Itanium	44 Reserved Register/Field fault	General Exception vector	
		45 Unimplemented Data Address fault	General Exception vector	
		46 Privileged Register fault	General Exception vector	
		47 Speculative Operation fault	Speculation vector	
		48 Virtualization fault	Virtualization vector	
	IA-32	49 IA-32 Stack Exception	IA-32 Exception vector (StackFault)	C
		50 IA-32 General Protection Fault	IA-32 Exception vector (GPFault)	
Faults	IA-32, Intel Itanium	51 Data Nested TLB fault	Data Nested TLB vector	
		52 Alternate Data TLB fault[f]	Alternate Data TLB vector	
		53 VHPT Data fault[f]	VHPT Translation vector	
		54 Data TLB fault[f]	Data TLB vector	
		55 Data Page Not Present fault[f]	Page Not Present vector	
		56 Data NaT Page Consumption fault[f]	NaT Consumption vector	
		57 Data Key Miss fault[f]	Data Key Miss vector	
		58 Data Key Permission fault[f]	Key Permission vector	
		59 Data Access Rights fault[f]	Data Access Rights vector	
		60 Data Dirty Bit fault	Dirty-Bit vector	
		61 Data Access Bit fault[f]	Data Access-Bit vector	
	Intel Itanium	62 Data Debug fault[f]	Debug vector	
		63 Unaligned Data Reference fault[f]	Unaligned Reference vector	
	IA-32	64 IA-32 Alignment Check fault	IA-32 Exception vector (AlignmentCheck)	C
		65 IA-32 Locked Data Reference fault	IA-32 Intercept vector (Lock)	
		66 IA-32 Segment Not Present fault	IA-32 Exception vector (NotPresent)	
		67 IA-32 Divide by Zero fault	IA-32 Exception vector (Divide)	
		68 IA-32 Bound fault	IA-32 Exception vector (Bound)	
		69 IA-32 SSE Numeric Error fault	IA-32 Exception vector (StreamSIMD)	
	Intel Itanium	70 Unsupported Data Reference fault	Unsupported Data Reference vector	
		71 Floating-point fault	Floating-point Fault vector	
Traps	Intel Itanium	72 Unimplemented Instruction Address trap[b,g]	Lower-Privilege Transfer Trap vector	
		73 Floating-point trap	Floating-point Trap vector	
		74 Lower-Privilege Transfer trap	Lower-Privilege Transfer Trap vector	
		75 Taken Branch trap	Taken Branch Trap vector	
		76 Single Step trap	Single Step Trap vector	

Table 5-6. Interruption Priorities (Continued)

Type	Instr. Set	Interruption Name	Vector Name	IA-32 Class[a]
	IA-32	77 IA-32 System Flag Intercept trap	IA-32 Intercept vector (SystemFlag)	D
		78 IA-32 Gate Intercept trap	IA-32 Intercept vector (Gate)	
		79 IA-32 INTO trap	IA-32 Exception vector (Overflow)	
		80 IA-32 Breakpoint (INT 3) trap	IA-32 Exception vector (Break)	
		81 IA-32 Software Interrupt (INT) trap	IA-32 Interrupt vector (Vector#)	
		82 IA-32 Data Breakpoint trap	IA-32 Exception vector (Debug)	
		83 IA-32 Taken Branch trap	IA-32 Exception vector (Debug)	
		84 IA-32 Single Step trap	IA-32 Exception vector (Debug)	

a. IA-32 Interruption Class, see Section 5.6.1, “IA-32 Interruption Priorities and Classes” on page 2:111 for details

b. Processor implementations may report unimplemented instruction addresses either with an Unimplemented Instruction Address trap on the taken branch, taken `chk`, or an `rfi` to an unimplemented address, or on a non-branching slot 2 instruction in a bundle at the upper edge of the implemented address space (where the next sequential bundle address would be an unimplemented address), or with an Unimplemented Instruction Address fault on the fetch of the unimplemented address.

c. IA-32 Code Fetch faults include Code Segment Limit Violation and other Code Fetch checks defined in Section 6.2.2.3.3, “IA-32 Environment Runtime Integrity Checks” on page 1:122.

d. Illegal Operation faults can be taken for certain predicated off reserved opcodes. For details, refer to Section 4.1, “Format Summary” on page 3:294.

e. IA-32 FP Error fault conditions detected on an IA-32 FP instruction are reported as a fault on the next IA-32 FP instruction that performs an FWAIT operation.

f. If not deferred.

g. Unimplemented Instruction Address traps on emulated check instructions have a lower priority than Taken Branch trap and Single Step trap. See “Speculation vector (0x5700)” on page 2:198.

5.6.1 IA-32 Interruption Priorities and Classes

Table 5-6 establishes a well defined priority between faults, traps and interrupts (including IA-32). However, IA-32 instruction set generated interruptions are divided into interruption classes. While priority among these IA-32 interruption classes is well defined by the table (except as noted below), interruption priority within each IA-32 interruption class is implementation dependent and may vary from processor to processor as defined below:

Class A – Faults from fetching an instruction. Priority of IA-32 Instruction Breakpoint, IA-32 Code Fetch (GPFault(0)), and Instruction TLB faults (Alternate Instruction TLB fault to Instruction Access Bit fault) may vary based on instruction alignment and page boundaries in a model-specific way. Faults are prioritized as defined in the table if the instruction does not span a virtual page. If an IA-32 instruction spans a virtual page, IA-32 Code Fetch faults (IA_32_Exception(GPFault)) due to code segment (CS) Limit violations can be raised above or below Instruction TLB faults as defined below:

- If the starting effective address of the IA-32 instruction exceeds the code segment limit, then the IA-32 Code Fetch fault has higher priority than any Instruction TLB faults. If the starting effective address of the IA-32 instruction is within the code segment limit, then Instruction TLB faults have higher priority for the starting effective address.
- If the IA-32 instruction spans a virtual page and the code segment limit is equal to the page boundary, the IA-32 Code Fetch fault has higher priority than any Instruction TLB faults on the second page. Otherwise if the code segment limit is

greater than the page boundary, any Instruction TLB faults on the second page have higher priority than the IA-32 Code Fetch fault.

Class B – Faults from decoding an instruction. Priority of IA-32 Instruction Length, IA-32 Invalid Opcode, and IA-32 Instruction Intercept, Disabled Floating Point Register, Disabled Instruction Set Transition, and Device Not Available faults are model specific. If the IA-32 instruction spans a virtual page, IA-32 Instruction Length >15 byte Faults (IA_32_Exception(GPFault)) can have higher priority than Instruction TLB faults as defined below:

- If the IA-32 prefix bytes on the first page are >= 15 bytes, an IA-32 Instruction >15 byte fault (GPFault) is taken first regardless of any Instruction TLB faults on the second page.
- If the IA-32 prefix bytes on the first page are < 15 bytes, Instruction TLB faults on the second page may or may not have priority over any possible IA-32 Instruction Length fault.

Class C – Faults resulting from executing an instruction. Priority of faults is model specific and can vary across processor implementations. Most faults are related to data memory references, other fault priorities can vary due to model-specific differences across processor implementations. The memory fault priorities (IA-32 Stack Exception through Data Access Bit fault) defined in the table only apply to a single IA-32 data memory reference that does not cross a virtual page. If an IA-32 instruction requires multiple data memory references or a single data memory reference crosses a virtual page:

- If any given IA-32 instruction requires multiple data memory references, all possible faults are raised on the first data memory reference before any faults are checked on subsequent data memory references. This implies lower priority faults on an earlier memory reference will be raised before higher priority faults on a later data memory reference within a single IA-32 instruction. The order of data memory references initiated by an IA-32 instruction is implementation dependent and may vary from processor to processor. Software can not assume all higher priority data memory faults are raised before all lower priority data memory faults within a single IA-32 instruction.
- If a single IA-32 data memory reference crosses a virtual page, the processor checks for faults in a model-specific order: Any faults present on one page are checked and reported before any faults are checked and reported on the other page. This implies that a single data reference that crosses a virtual page can raise lower priority data memory faults on one page before higher priority data memory faults are raised on the other page. For example, Data Key Miss faults (lower priority) on the first page could be raised before a Data TLB Miss Fault (higher priority) on the second page. Software can not assume all higher priority data memory faults are raised before all lower priority data memory faults within a single IA-32 instruction.

Class D – Traps on the current IA-32 instruction. Trap conditions are reported concurrently on the same exception vector or via a trap code specifying all concurrent traps.

5.7 IVA-based Interruption Vectors

Table 5-7 contains the processor's interruption vector table (IVT). The base of the IVT is held in the IVA control register. The size of the IVT is 32KB. The first 20 vectors are designed to provide more code space by allowing 64 bundles per vector (16 bytes per bundle) for performance-critical interruption handlers. The second 48 vectors provide 16 bundles per vector. Several vectors have more than one interruption associated with them. Information provided in the ISR allows the handler to distinguish which fault or trap caused the event.

Some vectors require additional software decoding to determine the cause of the interruption. Additional information for this decoding is provided in the ISR.code field. See Chapter 8, "Interruption Vector Descriptions" for a complete specification of the information supplied in the ISR for each of the vectors.

Note: PAL-based interruptions (RESET, MCA, INIT, and PMI) do not reference the IVT.

Table 5-7. Interruption Vector Table (IVT)

Offset	Vector Name	Interruption(s)	Page
0x0000	VHPT Translation vector	10, 23, 53	2:173
0x0400	Instruction TLB vector	24	2:175
0x0800	Data TLB vector	11, 54	2:176
0x0c00	Alternate Instruction TLB vector	22	2:177
0x1000	Alternate Data TLB vector	9, 52	2:178
0x1400	Data Nested TLB vector	8, 51	2:179
0x1800	Instruction Key Miss vector	27	2:180
0x1c00	Data Key Miss vector	14, 57	2:181
0x2000	Dirty-Bit vector	60	2:182
0x2400	Instruction Access-Bit vector	30	2:183
0x2800	Data Access-Bit vector	17, 61	2:184
0x2c00	Break Instruction vector	37	2:185
0x3000	External Interrupt vector	5	2:186
0x3400	Virtual External Interrupt vector	6	2:187
0x3800	Reserved		
0x3c00	Reserved		
0x4000	Reserved		
0x4400	Reserved		
0x4800	Reserved		
0x4c00	Reserved		
0x5000	Page Not Present vector	12, 25, 55	2:188
0x5100	Key Permission vector	15, 28, 58	2:189
0x5200	Instruction Access Rights vector	29	2:190
0x5300	Data Access Rights vector	16, 59	2:191
0x5400	General Exception vector	7, 35, 36, 38, 40, 44, 45, 46	2:192
0x5500	Disabled FP-Register vector	39	2:195
0x5600	NaT Consumption vector	13, 26, 43, 56	2:196
0x5700	Speculation vector	47	2:198
0x5800	Reserved for software use[a]		

Table 5-7. Interruption Vector Table (IVT) (Continued)

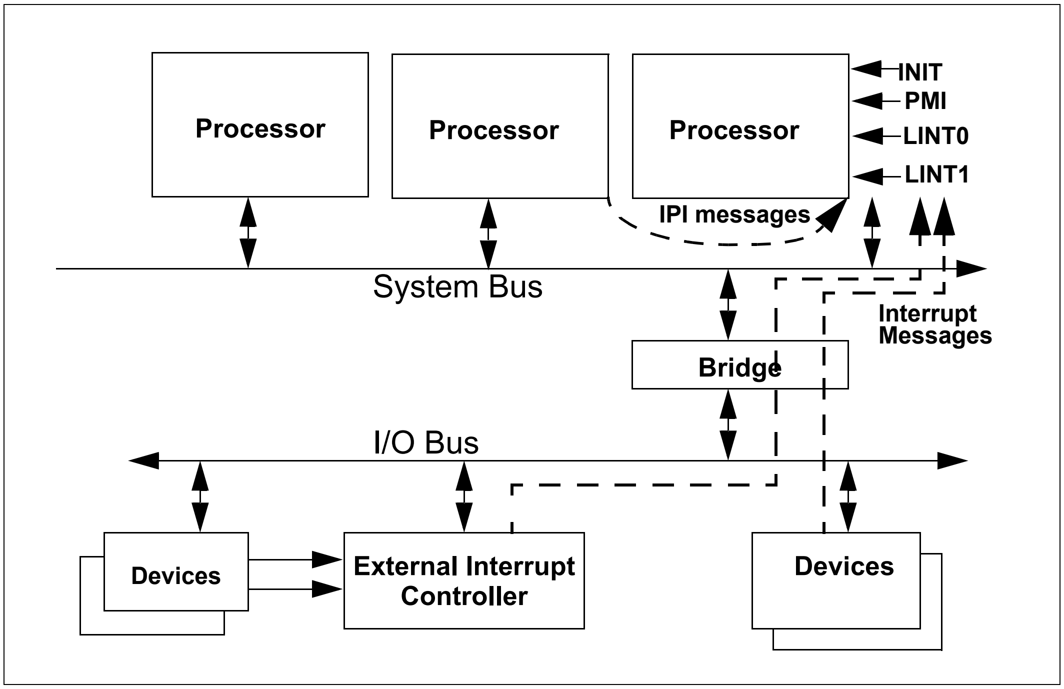
Offset	Vector Name	Interruption(s)	Page
0x5900	Debug vector	18, 31, 62	2:200
0x5a00	Unaligned Reference vector	63	2:201
0x5b00	Unsupported Data Reference vector	70	2:202
0x5c00	Floating-point Fault vector	71	2:203
0x5d00	Floating-point Trap vector	73	2:204
0x5e00	Lower-Privilege Transfer Trap vector	72, 74	2:205
0x5f00	Taken Branch Trap vector	75	2:207
0x6000	Single Step Trap vector	76	2:208
0x6100	Virtualization vector	48	2:209
0x6200	Reserved		
0x6300	Reserved		
0x6400	Reserved		
0x6500	Reserved		
0x6600	Reserved		
0x6700	Reserved		
0x6800	Reserved		
0x6900	IA-32 Exception vector	20, 21, 32, 41, 42, 49, 50, 64, 66, 67, 68, 79, 80, 82, 83, 84	2:210
0x6a00	IA-32 Intercept vector	33, 34, 65, 77, 78	2:211
0x6b00	IA-32 Interrupt vector	81	2:212
0x6c00	Reserved		
…	Reserved		
0x7f00	Reserved		

a. Unlike the other Reserved IVT vectors, which may defined in future revisions of the architecture, vector 0x5800 is permanently reserved. Software may use this vector for any purpose, such as placing in this area portions of other handlers that don't fit into their assigned vector.

5.8 Interrupts

This section describes the programming model of the high performance interrupt architecture. Interrupts are managed by the processor and by one or more intelligent external interrupt controllers or devices in the I/O subsystem. Figure 5-3 shows just one example of a high performance interrupt architecture subsystem; other topologies are possible. The processor is responsible for queuing and masking interrupts, sending and receiving inter-processor interrupt (IPI) messages, receiving interrupt messages from external interrupt controller(s), and managing local interrupt sources. This document describes the processor’s interrupt control mechanism only; for details on external interrupt controllers or I/O devices refer to platform documentation.

Figure 5-3. Interrupt Architecture Overview



As defined in "Interruption Definitions" on page 2:95 there are three kinds of interrupts: initialization interrupts (INITs), platform management interrupts (PMIs), and external interrupts (INTs).

The processors and external interrupt controllers communicate over the processor's system bus with an implementation-specific interrupt messaging protocol. Interrupts are generated by a number of different interrupt sources in the system:

- **External (I/O) devices** – Interrupt messages from any external source can be directed to any one processor by an external interrupt controller or by I/O devices capable of directly sending interrupt messages. An interrupt message informs the processor that an interrupt request is being made, and, in the case of PMIs and external interrupts, specifies a unique vector number for the interrupt. Interrupt messages are only issued on the "assertion edge" of an interrupt; "deassertion" of an interrupt does not result in an interrupt message.
- **Locally connected devices** – These interrupts originate on the processor's interrupt pins (LINT, INIT, PMI)[1], and are always directed to the local processor. The LINT pins can be connected directly to an Intel 8259A-compatible external interrupt controller. The LINT pins are programmable to be either **edge-sensitive** or **level-sensitive**, and for the kind of interrupt that gets generated. If programmed to generate external interrupts, the vector number is a programmed constant per LINT pin. Only the LINT pins connected to the processor can directly generate level-sensitive interrupts (See "Edge- and Level-sensitive Interrupts" on page 2:131). LINT pins cannot be programmed to generate level-sensitive PMIs or INITs. The INIT and PMI pins generate their corresponding interrupts. For PMI pins a PMI vector 0 interrupt is generated.

[1] Processors are not required to support externally connected interrupt pins. Software can query the presence of the INIT, PMI, and LINT pins via the PAL_PROC_GET_FEATURES procedure call.

- **Internal processor interrupts** – such as interval timer, performance monitoring, and corrected machine checks. These are always directed to the local processor. A unique vector number can be programmed for each source.
- **Other processors** – A processor can interrupt any individual processor, including itself, by sending an Inter-Processor Interrupt (IPI) message to a specific target processor. See "Inter-processor Interrupt Messages" on page 2:128.

The destination of an interrupt message is any one processor in the system, and is specified by a unique processor identifier. A different destination can be specified for each interrupt. There is no mechanism to "broadcast" a single interrupt to all processors in the system.

The following terms are used in the interrupt definition:

- The processor is said to **receive** an interrupt, if one of the processor's interrupt pins is asserted, the processor detected an interrupt message bus transaction containing the processor's unique identifier, or the processor detected an internal interrupt event.
- After receiving an interrupt, the processor internally holds the interrupt **pending**. The interrupt is said to be **pended** when it is received and held by the processor.
- For edge-sensitive interrupts, an external interrupt is held pending until the interrupt is acquired by software at which point it is said to be in-service. INITs and PMIs are held pending until the corresponding PAL vector is entered and PAL firmware clears the pending indication at which point they are said to be completed. For level-sensitive interrupts programmed through the LINT pins, the interrupt is held pending as long as the pin is asserted. Deassertion of a level-sensitive interrupt removes the pending indication (see "Edge- and Level-sensitive Interrupts" on page 2:131).
- The processor maintains an individual interrupt pending indication for INITs. Since external interrupts and PMIs are also signified by a unique interrupt **vector** number, the processor maintains individual pending indications per vector. An occurrence of an interrupt on a vector that is already marked as pending cannot be distinguished from previous interrupts on the same vector because the interrupts are pended in the same internal pending bit, and are therefore treated as "the same" interrupt occurrence.
- When interrupt delivery is enabled and the highest priority pending interrupt is unmasked (as defined below), the processor **accepts** the pending interrupt, interrupts the control flow of the processor and transfers control to the software interrupt handler.
- An external interrupt is said to be **in-service** when software **acquires** the interrupt vector from the processor by reading the IVR register (see "External Interrupt Vector Register (IVR – CR65)" on page 2:123). The processor then removes the pending indication for the interrupt vector. The processor maintains one in-service indicator for each unique vector number. Note that there are no in-service indicators for INITs and PMIs.
- Once an external interrupt is in-service it remains so until software indicates service for that external interrupt is **complete.** By writing to the EOI register (see "End of External Interrupt Register (EOI – CR67)" on page 2:124) software indicates that service for the highest-priority in-service external interrupt is complete. The processor then removes the in-service indication for the highest-priority external interrupt vector. INITs and PMIs are completed when PAL firmware clears the corresponding pending indication.

- The **priority** of interrupts is defined in Table 5-8. Entry *A* is higher priority than interrupt *B*, if entry *A* appears at a higher location in the table than entry *B*. Interrupt priority is used to select interrupts that require urgent service over less urgent interrupt requests.
- Interrupt **delivery** is **enabled** when software programs the processor to accept any unmasked interrupt. INITs delivery is enabled when PSR.mc is 0. PMIs delivery is enabled when PSR.ic is 1. For Itanium architecture-based code execution, external interrupts delivery is enabled when PSR.i is 1.
- **Masking** applies only to external interrupts. Unmasked interrupts are those external interrupts of higher priority than the highest priority external interrupt vector currently in-service (if any) and whose priority level is higher than the current priority masking level specified by the TPR register (see "Task Priority Register (TPR – CR66)" on page 2:123). Masking conditions are defined in Table 5-8. PSR.i does not affect masking of external interrupts.

Figure 5-4 shows how this terminology is applied to the handling of a PAL-based interrupt. Similarly, Figure 5-5 shows the handing of a vectored external interrupt *n*. Both figures show the different states and transitions interrupts go through.

Figure 5-4. PAL-based Interrupt States

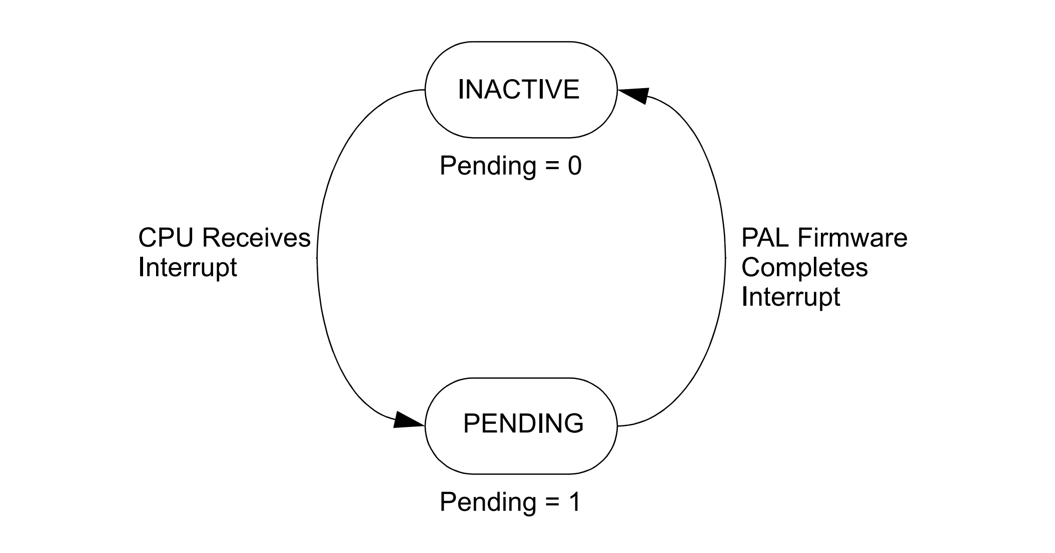
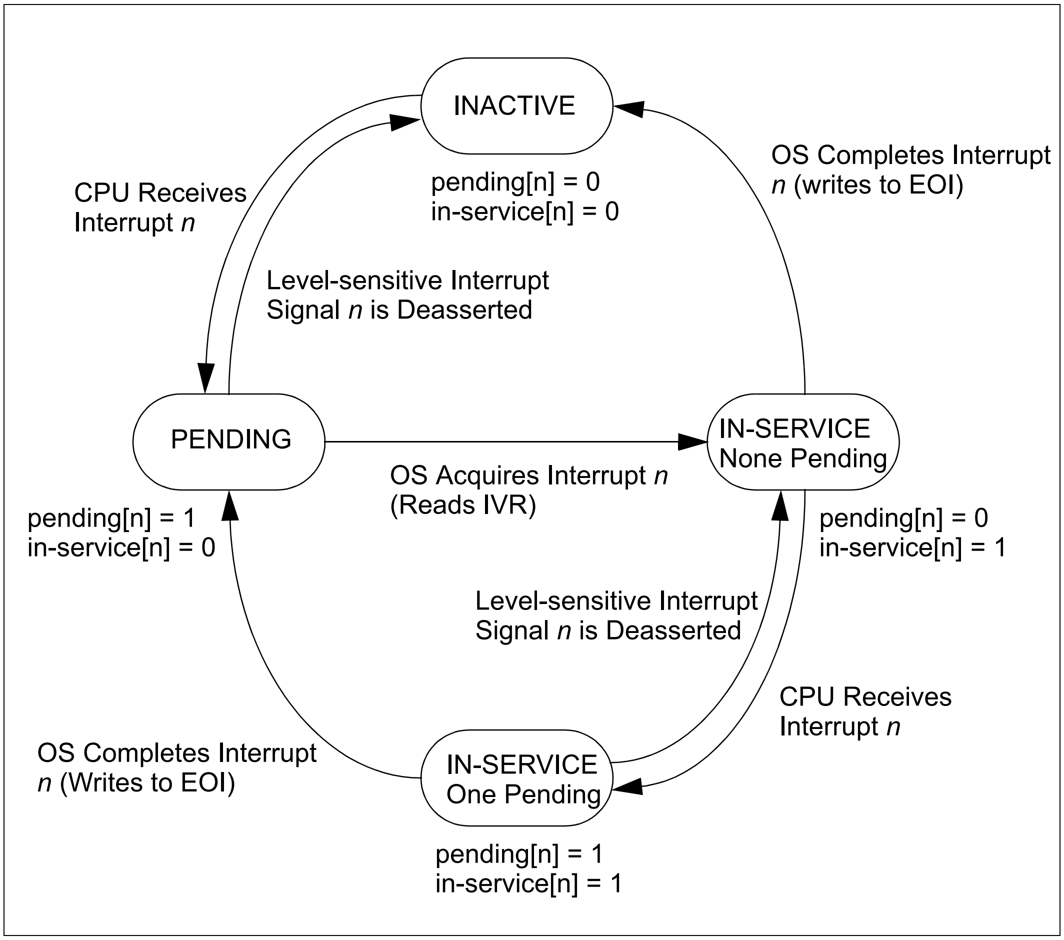


Figure 5-5. External Interrupt States



5.8.1 Interrupt Vectors and Priorities

As indicated in Table 5-6 on page 2:109, INITs have higher priority than PMIs, which in turn have higher priority than external interrupts. PMIs and external interrupts are further prioritized by vector number.

PMIs have a separate vector space from external interrupts. PMI vectors 0-3 can be used by platform firmware. PMI vectors 4 through 15 are reserved for use by processor firmware. Assertion of the processor's PMI pin, when present, results in PMI vector number 0. PMI vector priorities are described in Section 11.5, "Platform Management Interrupt (PMI)" on page 2:310.

Each external interrupt (INT) in the system is distinguished from other external interrupts by a unique vector number. There are 256 distinct vector numbers in the range 0 - 255. Vector numbers 1 and 3 through 14 are reserved for future use. Vector number 0 (ExtINT) is used to service Intel 8259A-compatible external interrupt controllers. Vector number 2 is used for the Non-Maskable Interrupt (NMI). The remaining 240 external interrupt vector numbers (16 through 255) are available for general operating system use. Table 5-8 summarizes the interrupt priority model.

Table 5-8. Interrupt Priorities, Enabling, and Masking

Priority	Priority Class	Interrupt	Vector Number	Interrupt Delivery Enabled	Interrupt Unmasked Condition
Highest	N/A	INIT	N/A	if PSR.mc is 0	Always
		PMI	0..3	if PSR.ic is 1	Always
		INT	2 (NMI)	if PSR.i is 1[a]	Interrupt is higher priority than all in-service external interrupts
			0 (ExtINT)		TPR.mmi is 0, and interrupt is higher priority than all in-service external interrupts
	15		240..255		TPR.mmi is 0, and interrupt is higher priority than all in-service external interrupts, and Vector Number{7:4} > TPR.mic
	14		224..239		
	13		208..223		
	12		192..207		
	11		176..191		
	10		160..175		
	9		144..159		
	8		128..143		
	7		112..127		
	6		96..111		
	5		80..95		
	4		64..79		
	3		48..63		
	2		32..47		
Lowest	1		16..31		

[a] For Itanium architecture-based code execution external interrupt delivery is enabled if PSR.i is 1. For IA-32 code execution external interrupt delivery is enabled if (PSR.i AND (!CFLAG.if OR EFLAG.if)) is true.

NMI (vector 2) has higher interrupt priority than ExtINT (vector 0), which has higher priority than external interrupt vectors 16 through 255.

External interrupts vectors 16 through 255 are divided into 15 interrupt priority classes. Sixteen different interrupt vectors share a single interrupt priority class, with class 1 being the lowest priority and class 15 being the highest. For these external interrupts, higher number external interrupts have priority over lower number external interrupts, including those within the same priority class.

Vector number 15 is used to indicate that the highest priority pending interrupt in the processor is at a priority level that is currently masked or there are no pending external interrupts. This encoding is referred to as a “spurious” interrupt.

5.8.2 Interrupt Enabling and Masking

Upon receiving an interrupt, the processor holds the interrupt pending internally until interrupt delivery is enabled and, in the case of external interrupts, the interrupt is unmasked. When all of the interrupt enabling and unmasking conditions are satisfied (see Table 5-8), the processor accepts the pending interrupt, interrupts the control flow of the processor, and transfers control to the External Interrupt handler for external interrupts, or to PAL firmware for INITs and PMIs.

Note: The TPR controls the masking of external interrupts. TPR is described in “Task Priority Register (TPR – CR66)” on page 2:123.

The processor provides nested interrupt priority support for external interrupt vectors 0, 2, and 16 through 255 by:

- Automatically masking external interrupts of equal or lower priority than the highest priority external interrupt currently in-service. This raises the in-service external interrupt masking level when each external interrupt begins service by an IVR read.
- Associating EOI writes with the highest priority in-service external interrupt, and removing the in-service indication for this external interrupt. This lowers the in-service masking level to that of the next highest priority currently in-service external interrupt (if any).

This mechanism allows software external interrupt handlers to be interrupted by higher priority external interrupts.

For example, assume software acquires an external interrupt vector 45 by reading IVR. During the service of this interrupt other external interrupts can still be received and are pended. If software sets PSR.i to a 1, pending external interrupts of equal or lower priority than 45 are masked. However, a higher priority pending external interrupt can be accepted by the processor (provided it is not masked by TPR.mmi or TPR.mic). Assuming external interrupt vector 80 is received by the processor, the processor will accept the interrupt by interrupting the control flow of the processor. During the service of this interrupt, external interrupts of equal or lower priority than vector 80 are masked. When EOI is issued by software, the processor will remove the in-service indication for external interrupt vector 80. External interrupt masking will then revert back to the next highest priority in-service external interrupt, vector 45. External interrupt vectors of equal or lower priority than vector 45 would remain masked until EOI is issued by software. The in-service indication for vector 45 is then removed by the write to EOI.

5.8.2.1 Re-enabling External Interrupt Delivery

When emerging from code in which external interrupt delivery is disabled and interruption state collection is turned off, the following minimal code sequence describes the architectural method with which to re-enable interruption collection and enable external interrupts:

```
ssm PSR.ic        // enable interruption collection
;;
srlz.d            // guarantee that interruption collection is enabled
ssm PSR.i         // enable external interrupts
```

The processor does not ensure that enabling external interrupts is immediately observed after the `ssm` PSR.i instruction. Software must perform a data serialization operation after `ssm` PSR.i to ensure that external interrupt delivery is enabled prior to a given point in program execution.

5.8.2.2 External Interrupt Sampling

Assuming that external interrupt delivery is currently disabled (PSR.i is 0), the following minimal code sequence describes the architectural method with which to briefly open the external interrupt window for external interrupt sampling (typically PSR.ic is 1 to enable interruption collection):

```
ssm PSR.i
;;
srlz.d          // external interrupts may be sampled anywhere here
;;
rsm PSR.i
```

The stop following the `srlz.d` instruction in the above code sequence is required to force the Reset System Mask (`rsm`) instruction into a subsequent instruction group. The stop guarantees that the `srlz.d` will open the external interrupt window for at least one cycle before the `rsm` instruction closes it again.

Note: In the above code sequence, the effect of disabling interrupts due to the `rsm` instruction is observed on the next instruction following the `rsm`.

5.8.2.3 Disabling of External Interrupt Delivery and rsm

When the current privilege level is zero, an `rsm` instruction whose mask includes PSR.i may cause external interrupt delivery to be disabled for an implementation-dependent number of instructions, even if the qualifying predicate for the `rsm` instruction is false. Architecturally, the extents of this delivery disable "window" are defined as follows:

1. External interrupt delivery may be disabled for any instructions in the same instruction group as the `rsm`, including those that precede the `rsm` in sequential program order, regardless of the value of the qualifying predicate of the `rsm` instruction.
2. If the qualifying predicate of the `rsm` is true, then external interrupt delivery is disabled immediately following the `rsm` instruction.
3. If the qualifying predicate of the `rsm` is false, then external interrupt delivery may be disabled until the next data serialization operation that follows the `rsm` instruction.

The delivery disable window is guaranteed to be no larger than defined by the above criteria, but it may be smaller, depending on the implementation.

When the current privilege level is non-zero, an `rsm` instruction whose mask includes PSR.i may briefly disable external interrupt delivery, regardless of the value of the qualifying predicate of the `rsm` instruction. However, the implementation guarantees that non-privileged code cannot lock out external interrupts indefinitely (e.g., via an arbitrarily long sequence of `rsm` PSR.i instructions with zero-valued qualifying predicates).

5.8.3 External Interrupt Control Registers

Software interacts with external interrupts by reading and writing the external interrupt control registers (CR64-81). These registers are summarized in Table 5-9, and are used to prioritize and deliver external interrupts, and to assign external interrupt vectors for processor-internal interrupt sources such as interval timer, performance monitoring, and corrected machine check.

The external interrupt control registers can only be accessed at privilege level 0, otherwise a Privileged Operation fault is raised.

Table 5-9. External Interrupt Control Registers

Register	Name	Description
CR64	LID	Local ID
CR65	IVR	External Interrupt Vector Register (read only)
CR66	TPR	Task Priority Register
CR67	EOI	End Of External Interrupt
CR68	IRR0	External Interrupt Request Register 0 (read only)
CR69	IRR1	External Interrupt Request Register 1 (read only)
CR70	IRR2	External Interrupt Request Register 2 (read only)
CR71	IRR3	External Interrupt Request Register 3 (read only)
CR72	ITV	Interval Timer Vector
CR73	PMV	Performance Monitoring Vector
CR74	CMCV	Corrected Machine Check Vector
CR80	LRR0	Local Redirection Register 0
CR81	LRR1	Local Redirection Register 1

5.8.3.1 Local ID (LID – CR64)

The LID register contains the processor's local interrupt identifier. Two fields (*id* and *eid*) serve as the processor's physical name for all interrupt messages (external interrupts, INITs, and PMIs). LID is loaded by firmware during platform initialization based on the processor's physical location within the system. Processors receiving an interrupt message on the system interconnect may or may not compare their *id*/*eid* fields with the target address for the interrupt message, depending on the type of system interconnect. If this comparison is performed, then a match would indicate that the interrupt received was intended for this processor. In case of no comparison, processors use other system topology mechanisms to determine the correct target of the interrupt message.

The LID register fields are either read-only or read-write. Details of the programmability of these fields is communicated by PAL at PALE_RESET handoff (see Section 11.2.2, "PALE_RESET Exit State" on page 2:289 for details). Read-only LID bits always return a value of 0. Writes to read-only bits are ignored. To ensure that future arriving interrupts see the updated LID value by a given point in program execution, software must perform a data serialization operation after a LID write and prior to that point. The Local ID fields are defined in Figure 5-6 and Table 5-10.

Figure 5-6. Local ID (LID – CR64)

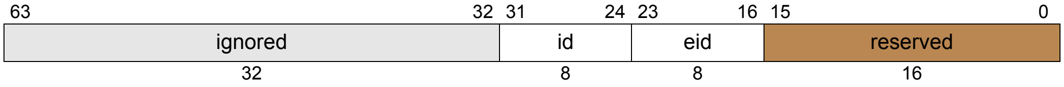


Table 5-10. Local ID Fields

Field	Bits	Description
id/eid	31:16	The low order bits of id correspond to a unique, geographically significant address of the processor on the local system bus. The eid field and the higher order bits of the id field correspond to a unique address of the local system bus within the entire system. These fields are initialized by platform firmware to an implementation-dependent value and should not be modified by software. The two fields corresponds to physical address bits{19:4} of the inter-processor interrupt message.

5.8.3.2 External Interrupt Vector Register (IVR – CR65)

A read of IVR returns the highest priority, pending, unmasked external interrupt vector, independent of the value of PSR.i. The external interrupt vector is an 8-bit encoded number. If there are no pending external interrupts or all external interrupts are currently masked, IVR returns the "spurious" interrupt indication (vector 15). IVR fields are shown in Figure 5-7. See "Interrupt Unmasked Condition" column in Table 5-8 on page 2:119 for masking conditions.

IVR reads also have two atomic side effects:

- The interrupt pending bit in IRR is cleared for the reported external interrupt vector. Subsequent IVR reads will not report the interrupt as pending unless a new interrupt was pended for the specified interrupt vector.
- The processor marks the interrupt vector as being in-service and masks all pending external interrupts with equal or lower priority until software writes the end-of-interrupt (EOI) register for the in-service interrupt.

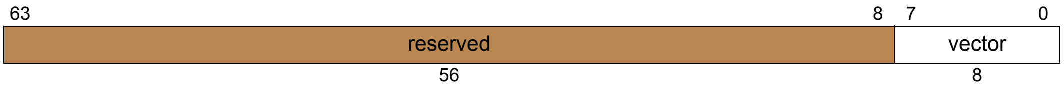
To ensure IVR side effects are observed by a given point in program execution (e.g., before the next IVR read, EOI write, or PSR.i write to enable external interrupt delivery), software must perform a data serialization operation after an IVR read and prior to that point. To ensure that the reported external interrupt vector is correctly masked before the next IVR read, software must perform a data serialization operation after a TPR or EOI write and prior to that IVR read.

Software must be prepared to service any possible external interrupt if it reads IVR, since IVR reads are destructive and removes the highest priority pending external interrupt (if any).

IVR is a read-only register; writes to IVR result in a Illegal Operation fault.

IVR reads do not issue an external INTA cycle. If the interrupt vector must be acquired from an Intel 8259A-compatible external interrupt controller, software should perform a load from the INTA byte. See "Interrupt Acknowledge (INTA) Cycle" on page 2:130 for details.

Figure 5-7. External Interrupt Vector Register (IVR – CR65)



5.8.3.3 Task Priority Register (TPR – CR66)

The processor's Task Priority Register (TPR) provides the ability to create additional masking of external interrupts based on a "priority class." The 240 external interrupt vectors (16 - 255) are divided into 15 priority classes of 16 numerically contiguous interrupt vectors each. The value written in TPR.mic masks all external interrupts of equal or lower priority classes.

To ensure that new priority levels are established by a given point in program execution, software must perform a data serialization operation after a TPR write and prior to that point. For example, if PSR.i is subsequently set to 1, thus enabling interrupts, and the new priority levels need to be in place before this enabling, a data serialization must be performed prior to the setting of PSR.i. Similarly, if PSR.pp or

PSR.up is set to 1, potentially enabling performance monitor interrupts, and the new priority levels need to be in place before this enabling, a data serialization must be performed. (Note that there's no dependence between writing TPR and then changing the PSR for any other bits in the PSR than these.) A data serialization operation must be performed after TPR is written and before IVR is read to ensure that the reported IVR vector is correctly masked. The TPR fields are described in Figure 5-8 and Table 5-11.

Figure 5-8. Task Priority Register (TPR – CR66)

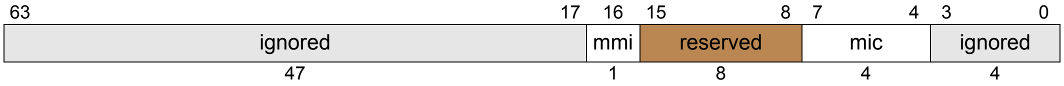


Table 5-11. Task Priority Register Fields

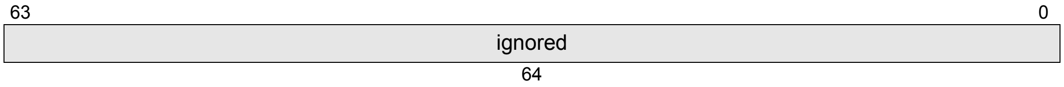
Field	Bits	Description
mic	7:4	Mask Interrupt Class: all external interrupt vectors of equal or lower priority classes then the TPR.mic field are masked. For example, if mic field is 4, interrupt priority classes 1, 2, 3, and 4 are masked. A TPR.mic value of 0 has no masking effect; a value of 15 will mask all external interrupt vectors in the range 16 - 255. TPR.mic has no effect on external interrupt vectors 0 and 2, INITs and PMIs. See “Processor Interrupt Block” on page 2:127..
mmi	16	Mask Maskable Interrupts: When 1, masks all external interrupts other than NMI (vector 2). When 0, external interrupt vectors 16 - 255, are masked by the TPR.mic field.

5.8.3.4 End of External Interrupt Register (EOI – CR67)

A write to the EOI (end-of-external interrupt) register, shown in Figure 5-9, indicates that software has finished servicing the highest priority in-service external interrupt. The processor removes its internal in-service indication for the highest priority currently in-service external interrupt vector. Pending external interrupts are then masked by the next highest priority in-service external interrupt (if any).

Writes to EOI affect the local processor only, and do not propagate to other processors or external interrupt controllers.

Figure 5-9. End of External Interrupt Register (EOI – CR67)



EOI is a read-write register. Reads return 0. Data associated with the EOI writes is ignored.

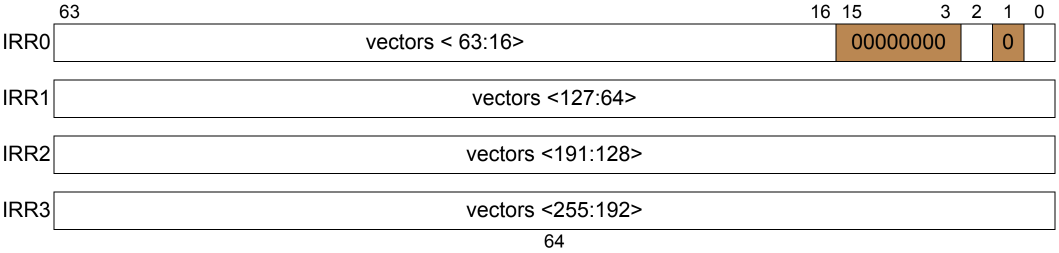
To ensure that the previous in-service interrupt indication has been cleared by a given point in program execution, software must perform a data serialization operation after an EOI write and prior to that point. To ensure that the reported IVR vector is correctly masked before the next IVR read, software must perform a data serialization operation after an EOI write and prior to that IVR read.

5.8.3.5 External Interrupt Request Registers (IRR0-3 – CR68,69,70,71)

Four 64-bit read-only External Interrupt Request Registers (IRR0-3, see Figure 5-10) provide the capability for software to determine the set of pending asynchronous external interrupts. IRR0 contains vectors <63:0> where vector 0 is in bit position 0, IRR1 contains vectors <127:64>, IRR2 contains vectors <191:128>, and IRR3 contains vectors <255:192>. A bit in the IRR, corresponding to the pending interrupt vector number, is set when the processor receives an external interrupt. The IRR bit is cleared when software reads the IVR and the vector number corresponding to the IRR bit value is returned in the IVR. The IRR bit is also cleared when a level-sensitive external interrupt signal is deasserted, effectively removing the pending interrupt.

Since IRR0-3 are read-only registers, writes to these registers result in Illegal Operation faults.

Figure 5-10. External Interrupt Request Register (IRR0-3 – CR68, 69, 70, 71)



5.8.3.6 Interval Timer Vector (ITV – CR72)

ITV specifies the external interrupt vector number for Interval Timer Interrupts. To ensure that subsequent interval timer interrupts reflect the new state of the ITV by a given point in program execution, software must perform a data serialization operation after an ITV write and prior to that point. See Figure 5-11 and Table 5-12 for the definitions of the ITV fields.

Figure 5-11. Interval Timer Vector (ITV – CR72)

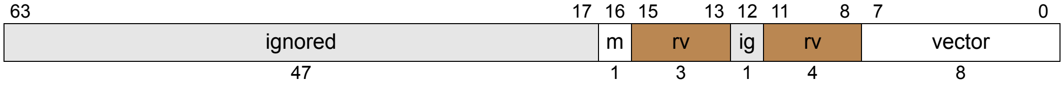


Table 5-12. Interval Timer Vector Fields

Field	Bits	Description
vector	7:0	External interrupt vector number to use when generating an Interval Timer interrupt. Vector values can be 0, 2 or 16-255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Interval Timer interrupts are discarded and not pended. When 0, occurrences of Interval Timer interrupts are pended.

5.8.3.7 Performance Monitoring Vector (PMV – CR73)

PMV specifies the external interrupt vector number for Performance Monitoring overflow interrupts. To ensure that subsequent performance monitor interrupts reflect the new state of PMV by a given point in program execution, software must perform a data serialization operation after a PMV write and prior to that point. See Figure 5-12 and Table 5-13 for the definitions of the PMV fields.

Figure 5-12. Performance Monitor Vector (PMV – CR73)

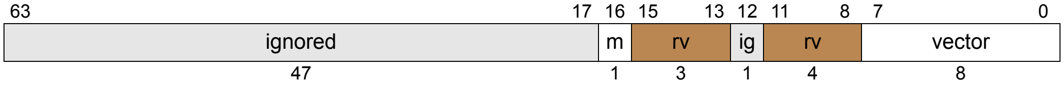


Table 5-13. Performance Monitor Vector Fields

Field	Bits	Description
vector	7:0	Vector number to use when generating a Performance Monitor interrupt. Vector values can be 0, 2, or 16-255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Performance Monitor interrupts are discarded and not pended. When 0, occurrences of Performance Monitor interrupts are pended.

5.8.3.8 Corrected Machine Check Vector (CMCV – CR74)

CMCV specifies the external interrupt vector number for Corrected Machine Checks. To ensure that subsequent corrected machine check interrupts reflect the new state of CMCV by a given point in program execution, software must perform a data serialization operation after a CMCV write and prior to that point. See Figure 5-13 and Table 5-14 for the CMCV field definitions.

Figure 5-13. Corrected Machine Check Vector (CMCV – CR74)

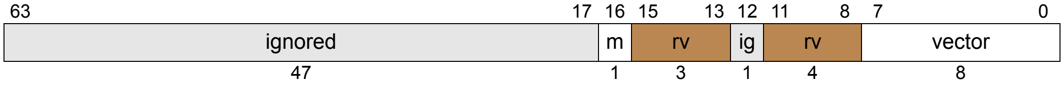


Table 5-14. Corrected Machine Check Vector Fields

Field	Bits	Description
vector	7:0	Vector number to use when generating a Corrected Machine Check. Vector values can be 0, 2, or 16 - 255. All other vectors are ignored and reserved for future use.
m	16	Mask: When 1, occurrences of Corrected Machine Check interrupts are discarded and not pended. When 0, occurrences of Corrected Machine Check interrupts are pended.

5.8.3.9 Local Redirection Registers (LRR0-1 – CR80,81)

Local Redirection Registers (LRR0-1) steer external signal-based interrupts that are directly connected to the local processor to a specific external interrupt vector. Processors may optionally support two direct external interrupt pins. When supported these external interrupt signals (pins) are referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1). Software can query the presence of these pins via the PAL_PROC_GET_FEATURES procedure call.

To ensure that subsequent interrupts from LINT0 and LINT1 reflect the new state of LRR prior to a given point in program execution, software must perform a data serialization operation after an LRR write and prior to that point. In the case when

LINT0 and LINT1 pins are absent, writes to LRR would have no effect, and reads from LRR would return 0. Software can query the presence of the LINT pins via the PAL_PROC_GET_FEATURES procedure call. The LRR fields are defined in Figure 5-14 and Table 5-15.

Figure 5-14. Local Redirection Register (LRR – CR80,81)

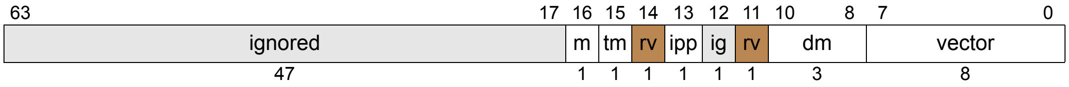


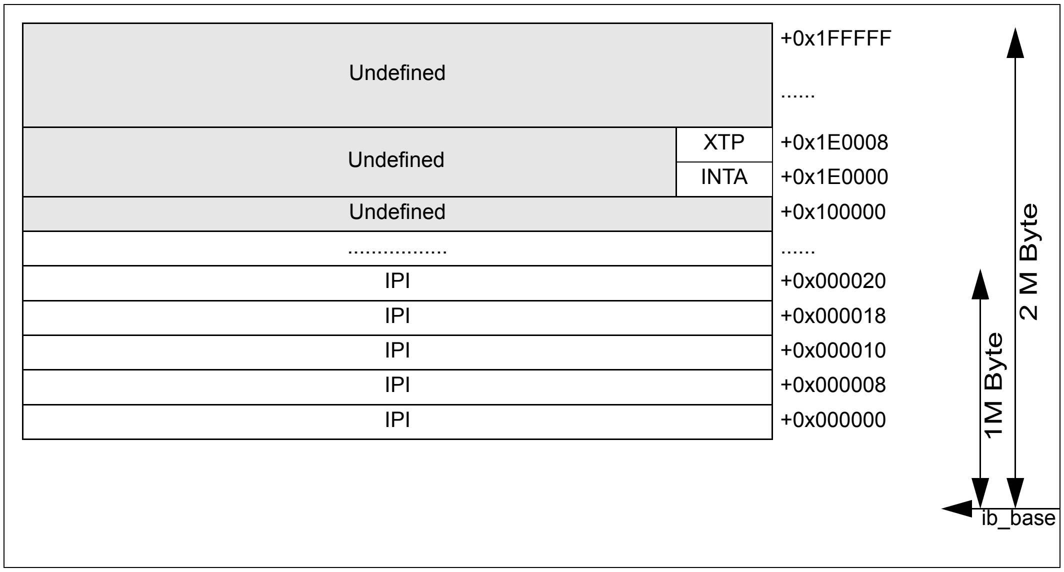
Table 5-15. Local Redirection Register Fields

Field	Bits	Description	
vector	7:0	External interrupt vector number to use when generating an interrupt for this entry. For INT delivery mode, allowed vector values are 0, 2, or 16-255. All other vectors are ignored and reserved for future use. For all other delivery modes this field is ignored.	
dm	10:8	000	INT – pend an external interrupt for the vector number specified by the vector field in LRR. Allowed vector values are 0, 2, or 16-255. All other vector numbers are ignored and reserved for future use.
		001	reserved
		010	PMI – pend a Platform Management Interrupt Vector number 0 for system firmware. The vector field is ignored.
		011	reserved
		100	NMI – pend a Non-Maskable Interrupt. This interrupt is pended at external interrupt vector number 2. The vector field is ignored.
		101	INIT – pend an Initialization Interrupt for system firmware. The vector field is ignored.
		110	reserved
		111	ExtINT – pend an Intel 8259A-compatible interrupt. This interrupt is delivered at external interrupt vector number 0. For details on servicing ExtINT external interrupts see "Interrupt Acknowledge (INTA) Cycle" on page 2:130. The vector field is ignored.
ipp	13	Interrupt Pin Polarity – specifies the polarity of the interrupt signal. When 0, the signal is active high. When 1, the signal is active low.	
tm	15	Trigger Mode – When 0, specifies edge sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an interrupt for the specified vector corresponding to the dm field. The pending interrupt indication is cleared by software servicing the interrupt. When 1, specifies level sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an external interrupt for the specified vector. Deassertion of the corresponding LINT pin clears the pending interrupt indication. The processor has undefined behavior if the dm and tm fields specify level sensitive PMIs or INITs.	
m	16	Mask – When 1, edge or level occurrences of the local interrupt pins are discarded and not pended. When 0, edge or level occurrences of local interrupt pins are pended.	

5.8.4 Processor Interrupt Block

Inter-Processor Interrupt (IPI) messages, Interrupt Acknowledge (INTA) cycles, and External Task Priority (XTP) cycles on the processor system bus are initiated by software by accessing a special physical memory range known as the "Processor Interrupt Block." Figure 5-15 defines its memory layout. The entire 2 MByte Processor Interrupt Block is relocatable by a PAL firmware call and must be aligned on a 2 MByte boundary; by default, the block is located at physical address 0x0000 0000 FEE0 0000.

Figure 5-15. Processor Interrupt Block Memory Layout



The Inter-Processor Interrupt region occupies the lower half of the Processor Interrupt Block; by default its physical address range is 0x0000 0000 FEE0 0000 through 0x0000 0000 FEEF FFFF. A processor generates Inter-Processor Interrupts by performing an aligned 8-byte store to this memory region.

The Processor Interrupt Block does not support all forms of memory operations. Unsupported memory accesses result in undefined processor operation.

- When targeted at the inter-processor interrupt delivery region (lower half of the Processor Interrupt Block), the following memory operations are undefined: instruction fetch, RSE accesses, or memory read references (only writes are permitted), references other than aligned 8-byte accesses, and references through any memory attribute other than UC.
- When targeted at the upper half of the Processor Interrupt Block, the following memory operations are undefined: instruction fetches, references other than 1-byte accesses to the XTP byte and 1-byte read access to the INTA byte, and references through any memory attribute other than UC.

Any memory operation targeted at the lower half of the Processor Interrupt Block which does not correspond to any actual processor is undefined.

5.8.4.1 Inter-processor Interrupt Messages

A processor can interrupt any individual processor, including itself*,* by issuing an inter-processor interrupt message (IPI). A processor generates an IPI by storing an 8-byte interrupt command to an 8-byte aligned address in the interrupt delivery region of the Processor Interrupt Block defined in "Processor Interrupt Block" on page 2:127. (If the address is not 8-byte aligned, the processor must either generate an Unaligned Data Reference Fault, see Section "Memory Datum Alignment and Atomicity" on page 2:93, or have undefined behavior). The address being stored to designates the target processor to receive the interrupt. The store address and data format of the inter-processor interrupt message are defined in Figure 5-16 and Figure 5-17. The data fields are defined in Table 5-17. The address processor identifier fields specify the target processor and are defined in Table 5-16.

Figure 5-16. Address Format for Inter-processor Interrupt Messages

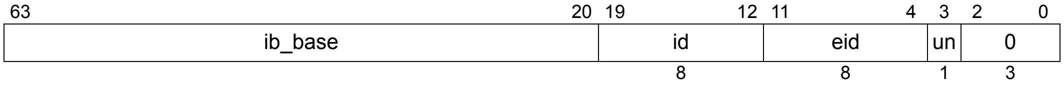


Figure 5-17. Data Format for Inter-processor Interrupt Messages

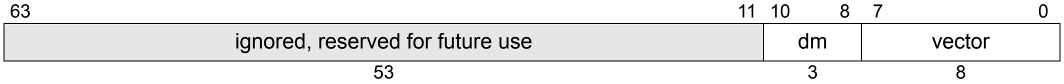


Table 5-16. Address Fields for Inter-processor Interrupt Messages

Field	Bits	Description
un	3	Unused. This field must be set to 0. Behavior of the inter-processor interrupt (IPI) message is undefined if this field is set to 1.
id/eid	19:4	Specify the target processor. See Table 5-10 on page 2:122 for a definition of these fields.
ib_base	63:20	Physical Base address of Processor Interrupt Block. This is a PAL relocatable physical address. The default is 0x0000 0000 FEE. See "Processor Interrupt Block" on page 2:127. Based on the processor model some of the high order physical address bits may be reserved.

Table 5-17. Data Fields for Inter-processor Interrupt Messages

Field	Bits	Description	
vector	7:0	Vector number for the interrupt. For INT delivery, allowed vector values are 0, 2, or 16-255. All other vectors are ignored and reserved for future use. For PMI delivery, allowed PMI vector values are 0-3. All other PMI vector values are reserved for use by processor firmware.	
dm	10:8	000	INT – pend an external interrupt for the specified vector to the processor listed in the destination. Allowed vector values are 0, 2, or 16-255. All other vector numbers are ignored and reserved for future use.
		001	Reserved
		010	PMI – pend a PMI interrupt for the specified vector to the processor listed in the destination. Allowed PMI vector values are 0-3. All other PMI vector values are reserved for use by processor firmware. See Section 11.5, "Platform Management Interrupt (PMI)" on page 2:310 for details.
		011	Reserved
		100	NMI – pend an external interrupt as an NMI (vector 2) to the processor listed in the destination. The vector field is ignored.
		101	INIT – pend an Initialization Interrupt for platform firmware on the processor listed in the destination. The vector field is ignored.
		110	Reserved
		111	ExtINT – pend an Intel 8259A-compatible interrupt. This interrupt is delivered at external interrupt vector number 0. For details on servicing ExtINT external interrupts see "Interrupt Acknowledge (INTA) Cycle" on page 2:130. The vector number field is ignored.
ignored	63:11	Ignored, reserved for future use	

5.8.4.2 Interrupt and IPI Ordering

Interrupt messages from external device(s), or external interrupts routed to the processor's LINT pins, when present, may arrive at one or more processors and become pending in any order. No ordering is enforced by the processor or the platform.

As observed by a receiving processor, IPIs emitted from the same issuing processor may be pended in any order, even when the receiving processor and the issuing processor are the same.

As observed by a receiving processor, IPIs are pended after all prior loads and stores emitted by the same issuing processor are visible if and only if the IPI is issued with a `st.rel` (or proceeded by an `mf`), even when the receiving processor and the issuing processor are the same. For all other cases, no ordering is implied between IPI transactions and prior cacheable or uncached memory references.

As observed by a receiving processor, no ordering is implied between IPIs and subsequent loads/stores from the same issuing processor, even when the receiving processor and the issuing processor are the same. Subsequent loads or stores may become visible before an IPI is seen as pending. Data or instruction serialization operations, memory fences (`mf` or `mf.a`), or `st.rel` do not ensure an IPI is pending at the target processor (including self) by a given point in program execution on the local processor.

5.8.4.3 Interrupt Acknowledge (INTA) Cycle

Intel 8259A-compatible external interrupt controllers can not issue interrupt messages and therefore do not specify an external interrupt vector number when the interrupt request is generated. When accepting an external interrupt, software must inspect the vector number supplied by the IVR register. If the vector matches the vector number assigned to the external controller (can be ExtINT, or any other vector number based on software convention), software must acquire the actual external interrupt vector number from the external interrupt controller by issuing a 1-byte load from the INTA Byte.

The INTA Byte is located within the upper half of the Processor Interrupt Block, at offset 0x1E0000 from the base. A single byte load from the INTA address causes the processor to emit the INTA cycle on the processor system bus. An Intel 8259A-compatible external interrupt controller must respond with the actual interrupt vector number as the data to be loaded. If two INTA cycles are required by the external interrupt controller, the platform must provide this functionality. Any memory operation to the INTA address other than a single byte load is undefined.

Software must issue an EOI to the local processor, to clear the interrupt in-service indication for the vector associated with the external interrupt controller.

5.8.4.4 External Task Priority (XTP) Cycle

Some model-specific system configurations support an External Task Priority Register (XTPR) per processor in external bus logic. A processor's XTPR can be modified by storing one byte of data to the processor's XTP Byte address. This generates a special bus transaction required to change the processor's XTPR within the system. Please refer to system-specific documentation for XTPR bit format and field definitions. The processor does not interpret any data stored to the XTP Byte address and all data bits are passed to the external system unmodified. Any memory operation to the XTP address other than a single byte store is undefined.

XTPR is written by operating system code to notify the system that the processor's current task priority has been changed. Based on this task priority information, system implementations can steer interrupt messages from the I/O subsystems to the processors that have registered the lowest task priority levels. The XTPR register is a system performance hint and need not be updated by operating system code nor be implemented in all system configurations. If the system does not implement the XTPR, it must still accept a processor's XTP cycle and discard it. Operating system code can issue XTPR updates regardless of external system support.

5.8.5 Edge- and Level-sensitive Interrupts

The processor's LINT pins, when present, directly support edge and level sensitive interrupts, however all other interrupt sources are edge sensitive. A single external interrupt messages is issued only on the assertion of an interrupt by external interrupt controllers or devices, deassertion of an external interrupt sends no interrupt message to the processor. Since the processor removes the pending interrupt when the interrupt is serviced, the processor guarantees exactly-one interrupt acceptance for each external interrupt message. By definition external interrupt messages are edge sensitive.

Level sensitive external interrupts can be supported using edge sensitive interrupt messages as follows:

- Software services the external interrupt generated by an edge interrupt message.
- Software removes the external interrupt request from the requesting device, the device should then deassert its interrupt request line.
- To avoid spurious external interrupts, it is highly recommended that software issue a dummy read from the device to ensure that the interrupt request has been actually been removed before the interrupt is resampled in the next step.
- Software issues a command to the external interrupt controller to resample the interrupt (typically an external interrupt controller end-of-interrupt command). The external interrupt controller must issue another interrupt message back to the processor if service is still required by the processor for a given vector number. For example, if there are other devices still requiring service that are attached to the same level sensitive interrupt request line.

§

Register Stack Engine 6

The register stack engine (RSE) moves registers between the register stack and the backing store in memory without explicit program intervention. The RSE operates concurrently with the processor and can take advantage of unused memory bandwidth to dynamically issue register spill and fill operations. In this manner, the latency of register spill/fill operations can be overlapped with useful program work. The basic principles of the register stack are discussed in Section 4.1, “Register Stack” on page 1:47. This chapter presents the internal state, the programming model and the interruption behavior of the register stack engine.

6.1 RSE and Backing Store Overview

The register stack frames are mapped onto a set of physical registers which operate as a circular buffer containing the most recently created frames. The RSE spills and fills these physical registers to/from a backing store in memory. The RSE moves registers between the physical register stack and the backing store without explicit program intervention. As indicated in Figure 6-1, the RSE operates on the physical stacked registers outside of the currently active frame (as defined by CFM). These registers contain the frames of the parent procedures of the current procedure.

As shown in Figure 6-1, the backing store is organized as a stack in memory that grows from lower to higher addresses. The Backing Store Pointer (BSP) application register contains the address of the first (lowest) memory location reserved for the current frame (i.e., the location at which GR32 of the current frame will be spilled). RSE spill/fill activity occurs at addresses below what is contained in the BSP since the RSE spills/fills the frames of the current procedure’s parents. The BSPSTORE application register contains the address at which the next RSE spill will occur. The address register which corresponds to the next RSE fill operation, the BSP load pointer, is not architecturally visible. The addresses contained in BSP and BSPSTORE are always aligned to an 8-byte boundary. The backing store contains the local area of each frame. The output area is not spilled to the backing store (unless it later becomes part of a callee’s local area). Within each stack frame, lower-addressed registers are stored at lower memory addresses. RSE spills of NaTed stacked general registers are subject to the same memory update constraints as software spills (st8.spill) of NaTed static general registers (see “Register Spill and Fill” on page 1:62).

The RSE also spills/fills the NaT bits corresponding to the stacked registers. The NaT bits corresponding to the static subset must be spilled/filled as necessary by software. The NaT bits are the 65th bit of each general register. The NaT bits for the stacked subset are spilled/filled in groups of 63 corresponding to 63 consecutive physical stacked registers. When the RSE spills a register to the backing store the corresponding NaT bit is copied to the RSE NaT collection (RNAT) application register. Whenever bits 8:3 of BSPSTORE are all ones, the RSE stores RNAT to the backing store. As shown in Figure 6-2, this results in a backing store memory image in which every 63 register values are followed by a collection of NaT bits. Bit 0 of the NaT collection corresponds to the first (lowest addressed) of the 63 register values; bit 62 corresponds to the 63rd register value. Bit 63 of the NaT collection is always written as zero. When the RSE fills a stacked register from the backing store it also fills the register's NaT bit. Whenever bits 8:3 of the RSE backing store load pointer are all ones, the RSE reloads a NaT collection from the backing store. Bit 63 of the NaT collection is ignored when read from the backing store.

Figure 6-1. Relationship Between Physical Registers and Backing Store

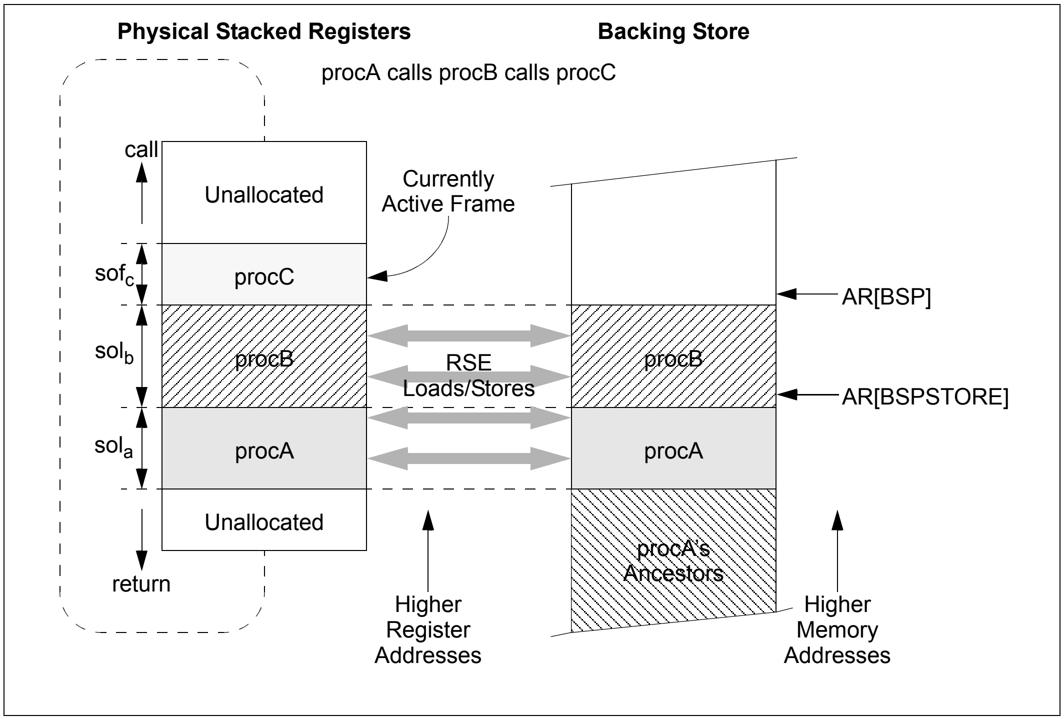
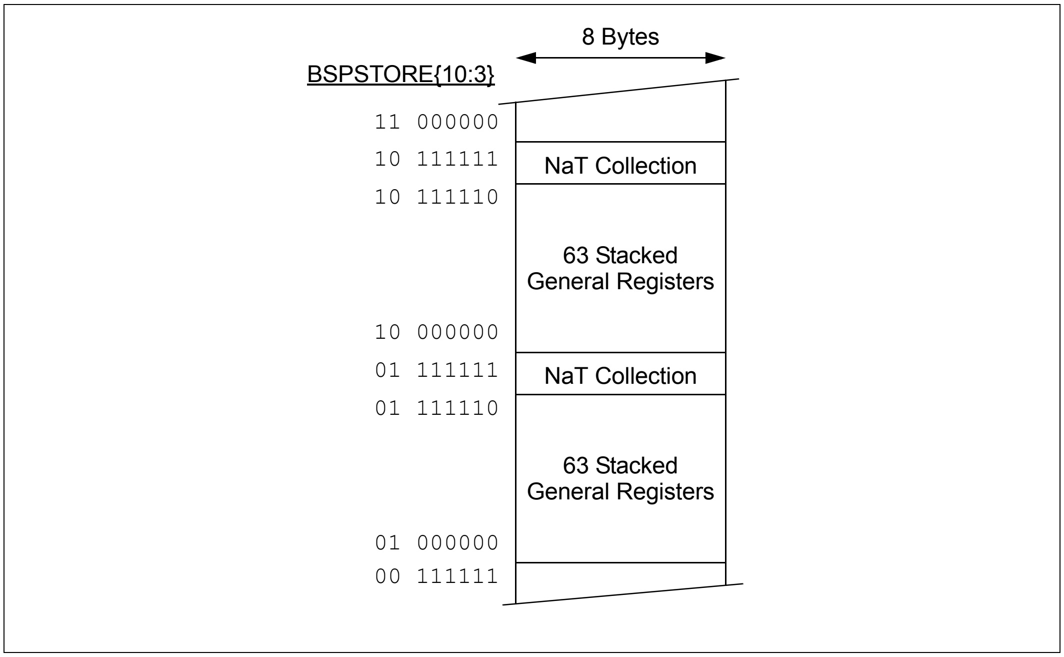


Figure 6-2. Backing Store Memory Format



The RSE operates concurrently and asynchronously with respect to instruction execution by taking advantage of unused memory bandwidth to dynamically perform register spill and fill operations. The algorithm employed by the RSE to determine whether and when to spill/fill is implementation dependent. Software can not depend on the spill/fill algorithm. To ensure that the processor and RSE activities do not interfere with each other, software should not access stacked registers outside of the current stack frame. The architecture guarantees register stack integrity by faulting on writes to out-of-frame registers. Reads from out-of-frame registers may interact with RSE operations and return undefined data values. However, out-of-frame reads are required to propagate NaT bits.

The operation of the RSE is controlled by the Register Stack Configuration (RSC) application register. Activity between the processor and the RSE is synchronized only when `alloc`, `flushrs`, `loadrs`, `br.ret`, or `rfi` instructions actually require registers to be spilled or filled, or when software explicitly requests RSE synchronization by executing a mov to/from RSC, BSPSTORE or RNAT application register instruction.

6.2 RSE Internal State

Table 6-1 describes architectural state that is maintained by the register stack engine. The RSE internal state elements described here are not directly exposed to the programmer as architecturally visible registers. As a consequence, RSE internal state does not need to be preserved across context switches or interruptions. Instead, it is modified as the side-effect of register stack-related instructions. To describe the effects of these instructions a complete definition of the RSE internal state is essential. To distinguish them from architecturally visible resources, all RSE internal state elements are prefixed with "RSE." Other RSE related resources are architecturally visible and are exposed to software as application registers: RSC, BSP, BSPSTORE, and RNAT.

Table 6-1. RSE Internal State

Name	Description	Corresponds To
RSE.N_STACKED_PHYS	Number of Stacked Physical registers: Implementation dependent size of the stacked physical register file.	
RSE.BOF	Bottom-of-frame register number: Physical register number of GR32.	AR[BSP]
RSE.StoreReg	RSE Store Register number: Physical register number of next register to be stored by RSE.	AR[BSPSTORE]
RSE.LoadReg	RSE Load Register number: Physical register number one greater than the next register to load (modulo the number of stacked physical registers).	RSE.BspLoad
RSE.BspLoad	Backing Store Pointer for memory loads: 64-bit Backing Store Address 8 bytes greater than the next address to be loaded by the RSE.	RSE.BspLoad
RSE.RNATBitIndex	RSE NaT Collection Bit Index: 6-bit wide RNAT Collection Bit Index (defines which RNAT collection bit gets updated)	AR[BSPSTORE]{8:3}
RSE.CFLE	RSE Current FrameLoad Enable: Control bit that permits the RSE to load registers in the current frame after a `br.ret` or `rfi`.	

Table 6-1. RSE Internal State (Continued)

Name	Description	Corresponds To
RSE.ndirty	Number of dirty registers on the register stack	
RSE.ndirty_words	Number of dirty words on the register stack plus corresponding number of NaT collection registers	AR[BSP] - AR[BSPSTORE]

6.3 Register Stack Partitions

The processor’s physical register file provides at least 96 stacked registers. The actual number of stacked registers (RSE.N_STACKED_PHYS) is implementation dependent and must be an even multiple of 16. Figure 6-3 illustrates the circular nature of the physical register file, and shows the correspondence of the registers to the backing store. Figure 6-3 also shows the four partitions of the stacked register file:

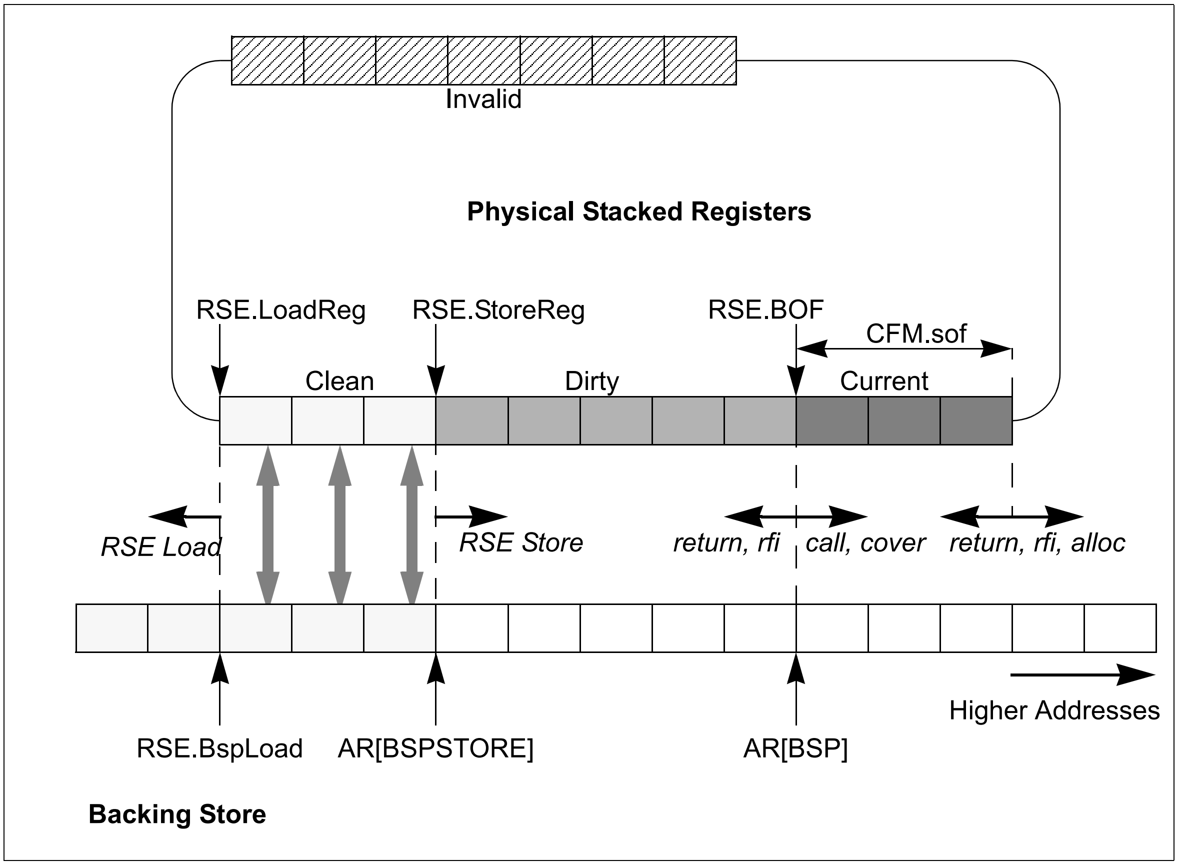
Clean partition (lightly-shaded): registers that contain values from parent procedure frames. The registers in this partition have been successfully spilled to the backing store by the RSE and their contents have not been modified since they were written to the backing store.

Dirty partition (medium-shaded): registers that contain values from parent procedure frames. The registers in this partition have not yet been spilled to the backing store by the RSE. The number of registers contained in the dirty partition (distance between RSE.StoreReg and RSE.BOF) is referred to as RSE.ndirty.

Current frame (shaded dark): stacked registers allocated for computation. The position of the current frame in the physical stacked register file is defined by the Bottom-of-frame register (RSE.BOF). The number of registers in the current frame is defined by the size of frame field in the current frame marker (CFM.sof).

Invalid partition (diagonally striped): registers outside the current frame that do not contain values from parent procedure frames. They are immediately available for allocation into the current frame or for RSE load operations.

Figure 6-3. Four Partitions of the Register Stack



The boundaries between the four register stack partitions are defined by the current frame marker (CFM) and three physical register numbers: a load, store and bottom-of-frame register number. As described in Table 6-1 each of these physical register numbers has a corresponding 64-bit backing store memory address pointer. (For example, AR[BSP] always contains the address where GR[32] of the current frame will be stored.)

Figure 6-3 also shows the effects of various instructions on the partition boundaries. RSE loads use invalid registers. RSE stores use dirty registers. Eager RSE loads and stores grow the clean partition. A `br.call`, `brl.call`, or `cover` instruction can increase the bottom-of-frame pointer (RSE.BOF) which moves registers from the current frame to the dirty partition. An `alloc` may shrink or grow the current frame by updating CFM.sof. A `br.ret` or `rfi` instruction may shrink or grow the current frame by updating both the bottom-of-frame pointer (RSE.BOF) and CFM.sof.

6.4 RSE Operation

The register stack backing store is organized as a stack in memory that grows from lower addresses towards higher addresses. The top of the backing store stack is defined by the Backing Store Pointer (BSP) application register, which points to the first memory location reserved for the current frame. The RSE load and store activities take place at lower addresses, defined relative to BSP by the sizes of the clean and dirty partitions. Although the stack is conceptually infinite in both directions, the effective base of the stack is expected to be the first memory location of the first page allocated to the backing store.

To allow the highest possible degree of concurrent execution, the processor and the RSE operate independently of each other during normal program execution. The RSE distinguishes between **mandatory** and **eager** load/store operations. Mandatory load/store operations occur as the result of `alloc`, `flushrs`, `loadrs`, `br.ret` or `rfi` instructions. Eager operations occur when the RSE is speculatively working ahead of program execution, and it is not known whether this register spill/fill is actually required by the program.

When the RSE works in the background, it issues eager RSE spill and fill operations to extend the size of the clean partition in both directions—by decreasing the RSE load pointer and loading values from the backing store into invalid registers (eager RSE load), and by saving dirty registers to the backing store and increasing the RSE store pointer (eager RSE store). Allocation of a sufficiently large frame (using `alloc`) or execution of a `flushrs` instruction may cause the RSE to suspend program execution and issue mandatory RSE stores until the required number of registers have been spilled to the backing store. Similarly a `br.ret` or `rfi` back to a sufficiently large frame or execution of a `loadrs` instruction may cause the RSE to suspend program execution and issue mandatory RSE loads until the required number of registers have been restored from the backing store. The RSE only operates in the foreground and suspends program execution whenever forward progress of the program actually requires registers to be spilled or filled.

Table 6-2 describes the RSE operation instructions and state modifications.

Table 6-2. RSE Operation Instructions and State Modification

Affected State	Instruction			
	`alloc` r_1=`ar.pfs`,i,l,o,r[a]	`br.call`[a], `brl.call`[a]	`br.ret`[a]	`rfi` when CR[IFS].v = 1
AR[BSP]{63:3}	unchanged	AR[BSP]{63:3} + CFM.sol + (AR[BSP]{8:3} + CFM.sol)/63	AR[BSP]{63:3} - AR[PFS].pfm.sol - (62-AR[BSP]{8:3}+ AR[PFS].pfm.sol)/63	AR[BSP]{63:3} - CR[IFS].ifm.sof - (62-AR[BSP]{8:3}+ CR[IFS].ifm.sof)/63

AR[PFS]	unchanged	AR[PFS].pfm = CFM
		AR[PFS].pec = AR[EC]
		AR[PFS].ppl = PSR.cpl
			unchanged	unchanged

GR[r_1]	AR[PFS]	N/A	N/A	N/A

CFM	CFM.sof = $i+l+o$
	CFM.sol = $i+l$
	CFM.sor = r >> 3
		CFM.sof -= CFM.sol
		CFM.sol = 0
		CFM.sor = 0
		CFM.rrb.gr = 0
		CFM.rrb.fr = 0
		CFM.rrb.pr = 0
			AR[PFS].pfm
			or [b]
			CFM.sof = 0
			CFM.sol = 0
			CFM.sor = 0
			CFM.rrb.gr = 0
			CFM.rrb.fr = 0
			CFM.rrb.pr = 0
				CR[IFS].ifm

a. These instructions have undefined behavior with an incomplete frame. See "RSE Behavior with an Incomplete Register Frame" on page 2:146.

b. Normal `br.ret` instructions restore CFM with AR[PFS].pfm. However, if a bad PFS value is read by the `br.ret` instruction, all CFM fields are set to zero. See "Bad PFS used by Branch Return" on page 2:143.

6.5 RSE Control

The RSE can be controlled at all privilege levels by means of three instructions (`cover`, `flushrs`, and `loadrs`) and by accessing four application registers (mov to/from RSC, BSP, BSPSTORE and RNAT). This section first presents each of the RSE application registers, and then discusses the three RSE control instructions.

6.5.1 Register Stack Configuration Register

The layout of the Register Stack Configuration application register (RSC) is defined in Section 3.1.8.2, "Register Stack Configuration Register (RSC – AR 16)" on page 1:29. This section describes the semantics of the mode, the privilege level and the byte order fields of the RSC. The loadrs field is described as part of the `loadrs` instruction in Section 6.5.4, "RSE Control Instructions" on page 2:142.

RSE Mode: Two mode bits in the RSC register determine when the RSE generates register spill or fill operations. When both mode bits are zero (enforced lazy mode) the RSE issues only mandatory loads and stores (when an `alloc`, `br.ret`, `flushrs` or `rfi` instruction requires registers to be spilled or filled). Bit 0 of the RSC.mode field enables eager RSE stores and bit 1 enables eager RSE loads. Table 6-3 defines all four possible RSE modes. Please see the processor-specific documentation for further information on the RSE modes implemented by the Itanium processor.

Table 6-3. RSE Modes (RSC.mode)

Mode	RSE Loads	RSE Stores	RSC.mode
Enforced Lazy	Mandatory only	Mandatory only	00
Store Intensive	Mandatory only	Eager and Mandatory	01
Load Intensive	Eager and Mandatory	Mandatory only	10
Eager	Eager and Mandatory	Eager and Mandatory	11

The algorithm that decides whether and when to speculatively perform eager register spill or fill operations is implementation dependent. Software may not make any assumptions about the RSE load/store behavior when the RSC.mode is non-zero. Furthermore, access to the BSPSTORE and RNAT application registers and the execution of the `loadrs` instructions require RSC.mode to be zero (enforced lazy mode). If `loadrs`, move to/from BSPSTORE or move to/from RNAT are executed when RSC.mode is non-zero an Illegal operation fault is raised. Eager spill/fill of the RNAT register to/from the backing store is only permitted if the RSE is in store/load intensive or eager mode. In enforced lazy mode, the RSE may spill/fill the RNAT register only if a subsequent mandatory register spill/fill is required.

RSE Privilege Level: When address translation is enabled (PSR.rt is one), the RSE operates at a privilege level defined by two privilege level bits in the Register Stack Configuration register (RSC.pl). All privilege level checks for RSE virtual accesses are performed using the privilege level in RSC.pl. When the RSC is written, the privilege level bits are clipped to the current privilege level of the process, i.e., the numerical maximum of the current privilege level and the privilege level in the source register is written to RSC.pl.

Protection is also checked based on the current entries in the data TLB. The RSE always remains coherent with respect to the data TLB. If a translation that is being used by the RSE is changed or purged, the RSE will immediately begin using the new translation or suffer a TLB miss. Only mandatory loads and stores can cause RSE memory related faults. Details on RSE fault delivery are described in "RSE Interruptions" Although eager RSE loads and stores do not cause interruptions they can, under certain conditions, cause a VHPT walk and TLB insert. Details on when RSE loads and stores can cause a VHPT walk are described in "VHPT Environment" on page 2:67.

The RSE expects its backing store to be mapped to cacheable speculative memory. If RSE spill/fill transactions are performed to non-speculative memory that may contain I/O devices, system behavior is unpredictable.

RSE Byte Order: Because the RSE runs asynchronously with the processor, it may be running on behalf of a context with a different byte order from the current one. Consequently, the RSE defines its own byte ordering bit: RSC.be. When RSC.be is zero, registers are stored in little-endian byte order (least significant bytes to lower addresses). When RSC.be is one, registers are stored in big-endian byte order (most significant bytes to lower addresses). RSC.be also determines the byte order of NaT collections spilled/filled by the RSE. RSC.be may be written by code at any privilege level. Changes to RSC.be should only be made by software when RSC.mode is zero. Failure to do so results in undefined backing store contents.

6.5.2 Register Stack NaT Collection Register

As described in Section 6.1, "RSE and Backing Store Overview" on page 2:133, the RSE is responsible for saving and restoring NaT bits associated with the stacked registers to and from the backing store. The RSE writes its NaT collection register (the RNAT application register) to the backing store whenever BSPSTORE{8:3} = 0x3F (1 NaT collection for every 63 registers). The RNAT acts as a temporary holding area for up to 63 unsaved NaT bits. The RSE NaT collection bit index (RSE.RNATBitIndex) determines which bit of the RNAT register receives the NaT bit of a spilled register as the result of an RSE store. The six-bit wide RSE.RNATBitIndex is always equal to BSPSTORE{8:3}. As a result, RNAT{*x*} corresponds to the register saved at

```
concatenate(BSPSTORE{63:9},x{5:0},0{2:0}).
```

The RSE never saves partial NaT collections to the backing store, so software must save and restore the RNAT application register when switching the backing store pointer. RSE.RNATBitIndex determines which RNAT bits are valid. Bits RNAT{RSE.RNATBitIndex:0} contain defined values, and bits RNAT{62:RSE.RNATBitIndex+1} contain undefined values. Bit 63 of the RNAT application register always reads as zero. Writes to bit 63 of the RNAT application register are ignored. The execution of RSE control instructions `mov` to BSPSTORE and `loadrs` as well as an RSE spill of the RNAT register cause the contents of the RNAT register to become undefined. The RNAT application register can only be accessed when RSC.mode is zero. If RSC.mode is non-zero, accessing the RNAT application register results in an Illegal Operation fault.

6.5.3 Backing Store Pointer Application Registers

The RSE defines two Backing Store Pointer application registers: BSPSTORE and BSP. Since the RSE backing store pointers are always 8-byte aligned, bits {2:0} of the backing store pointers always read as zero. When writing the BSPSTORE application register, bits {2:0} in the presented address are ignored.

The RSE Backing Store Pointer for memory stores (BSPSTORE) is a 64-bit application register that provides the main interface to the three RSE backing store memory pointers: BSP, BSPSTORE and RSE.BspLoad. The BSPSTORE application register can only be accessed when RSC.mode is zero. If RSC.mode is non-zero, accessing BSPSTORE results in an Illegal Operation fault.

Reading BSPSTORE (`mov` from BSPSTORE application register) returns the address of the next RSE store.

Writing BSPSTORE (`mov` to BSPSTORE application register) has side-effects on all three RSE pointers and the NaT collection process. The operation is defined as follows: the BSPSTORE and RSE.BspLoad pointers are both set to the address presented, which forces the size of the clean partition to zero. Writes to the BSPSTORE application register do not change the size of the dirty partition: the BSP pointer is set to the address presented plus the size of the dirty partition plus the size of any intervening NaT collections. The dirty partition is preserved to allow software to change the backing store pointer without having to flush the register stack. Writing BSPSTORE causes the contents of the RNAT register to become undefined. Therefore software must preserve the contents of RNAT prior to writing BSPSTORE. After writing to BSPSTORE, the NaT collection bit index (RSE.RNATBitIndex) is set to bits {8:3} of the presented address. If an unimplemented address in BSPSTORE is used by a mandatory RSE spill or fill, an Unimplemented Data Address fault is raised.

The RSE Backing Store Pointer (BSP) is a 64-bit read-only application register. Writing BSP (`mov` to BSP application register) results in an Illegal Operation fault. Reads from BSP (`mov` from BSP application register) return the address of the top of the register stack in memory. This location is the backing store address to which the current GR32 would be written. Reading BSP does not have any side-effect on any of the internal RSE pointers or the NaT collection process. Therefore, BSP can be read regardless of the RSE mode, i.e., even when RSC.mode is non-zero. Since BSP is determined by BSPSTORE and the size of the dirty partition, it is possible for BSPSTORE to contain an implemented address and for BSP to contain an unimplemented address. BSP reads always return a full 64-bit (possibly unimplemented) address; only a subsequent data memory reference with an unimplemented address will cause an Unimplemented Data Address fault.

Table 6-4 summarizes the effects of the three instructions that access the backing store pointer application registers.

Table 6-4. Backing Store Pointer Application Registers

Affected State	Instruction		
	Read BSP `mov` r_1`=AR[BSP]`	Read BSPSTORE `mov` r_1`=AR[BSPSTORE]`	Write BSPSTORE[a] `mov AR[BSPSTORE]=`r_2
GR[r_1]	AR[BSP]	AR[BSPSTORE]	N/A
AR[BSP]{63:3}	Unchanged	Unchanged	(GR[r_2]{63:3} + RSE.ndirty) + ((GR[r_2]{8:3} + RSE.ndirty)/63)
AR[BSPSTORE]{63:3}	Unchanged	Unchanged	GR[r_2]{63:3}
RSE.BspLoad {63:3}	Unchanged	Unchanged	GR[r_2]{63:3}
AR[RNAT]	Unchanged	Unchanged	UNDEFINED
RSE.RNATBitIndex	Unchanged	Unchanged	GR[r_2]{8:3}

a. Writing to AR[BSPSTORE] has undefined behavior with an incomplete frame. See “RSE Behavior with an Incomplete Register Frame” on page 2:146.

6.5.4 RSE Control Instructions

This section describes the RSE control instructions: `cover`, `flushrs` and `loadrs`. The effects of the three RSE control instructions on the RSE state are summarized in Table 6-5.

The `cover` instruction adds all registers in the current frame to the dirty partition, and allocates a zero-size current frame. As a result AR[BSP] is updated. `cover` clears the register rename base fields in the current frame marker CFM. If PSR.ic is zero, the original value of CFM is copied into CR[IFS].ifm and CR[IFS].v is set to one. The `cover` instruction must the last instruction in an instruction group; otherwise, operation is undefined.

The `flushrs` instruction spills all dirty registers to the backing store. When it completes, RSE.ndirty is defined to be zero, and BSPSTORE equals BSP. Since `flushrs` may cause RSE stores, the RNAT application register is updated. A `flushrs` instruction must be the first instruction in an instruction group otherwise the results are undefined.

The `loadrs` instruction ensures that a specified portion of the backing store below the current BSP is present in the physical stacked registers. The size of the backing store section is specified in the `loadrs` field of the RSC application register (AR[RSC].loadrs). After loadrs completes, all registers and NaT collections between the current BSP and the tear-point (BSP-(RSC.loadrs{13:3} << 3)), and no more than that, are guaranteed to be present and marked as dirty in the stacked physical registers. When `loadrs` completes BSPSTORE and RSE.BspLoad are defined to be equal to the backing store tear-point address. All other physical stacked registers are marked invalid.

- If the tear-point specifies an address below RSE.BspLoad, the RSE issues mandatory loads to restore registers and NaT collections. All registers between the current BSP and the tear-point are marked dirty.
- If the RSE has already loaded registers beyond the tear-point when the `loadrs` instruction executes, the RSE marks clean registers below the tear-point as invalid and marks clean registers above the tear-point as dirty.
- If the tear-point specifies an address greater than BSPSTORE, the RSE marks clean and dirty registers below the tear-point as invalid (in this case dirty registers are lost).

Table 6-5. RSE Control Instructions

Affected State	Instruction		
	cover	flushrs[a]	loadrs[a]
AR[BSP]{63:3}	AR[BSP]{63:3}+ CFM.sof + (AR[BSP]{8:3} + CFM.sof)/63	Unchanged	Unchanged
AR[BSPSTORE]{63:3}	Unchanged	AR[BSP]{63:3}	AR[BSP]{63:3} - AR[RSC].loadrs{13:3}
RSE.BspLoad{63:3}	Unchanged	Model specific[b]	AR[BSP]{63:3} - AR[RSC].loadrs{13:3}
AR[RNAT]	Unchanged	Updated	UNDEFINED
RSE.RNATBitIndex	Unchanged	AR[BSPSTORE]{8:3}	AR[BSPSTORE]{8:3}

CR[IFS]	if (PSR.ic == 0) {
	CR[IFS].ifm = CFM
	CR[IFS].v = 1}
		Unchanged	Unchanged

CFM	CFM.sof = 0
	CFM.sol = 0
	CFM.sor = 0
	CFM.rrb.gr = 0
	CFM.rrb.fr = 0
	CFM.rrb.pr = 0
		Unchanged	Unchanged

a. These instructions have undefined behavior with an incomplete frame. See “RSE Behavior with an Incomplete Register Frame” on page 2:146.

b. In general, eager RSE implementations will preserve RSE.BspLoad during a `flushrs`. Lazy RSE implementations may set RSE.BspLoad to AR[BSPSTORE] after flushrs completes or faults.

By specifying a zero RSC.loadrs value `loadrs` can be used to invalidate all stacked registers outside the current frame. `loadrs` causes the contents of the RNAT register to become undefined. The NaT collection index is set to bits {8:3} of the new BSPSTORE. A `loadrs` instruction must be the first instruction in an instruction group otherwise the results are undefined. The following conditions cause `loadrs` to raise an Illegal Operation fault:

- If RSC.mode is non-zero.
- If both CFM.sof and RSC.loadrs are non-zero.
- If RSC.loadrs specifies more words to be loaded than will fit in the stacked physical register file (RSE.N_STACKED_PHYS).

6.5.5 Bad PFS used by Branch Return

On a `br.ret`, if the PFS application register defines an output area which is larger than the number of implemented stacked registers minus the size of dirty partition ((AR[PFS].sof - AR[PFS].sol) > (RSE.N_STACKED_PHYS - RSE.ndirty)), the return will not restore CFM with AR[PFS].pfm (normal behavior); instead, the return sets all fields in the CFM (of the procedure being returned to) to zero.

Typical procedure call and return sequences that preserve PFS values and that do not use `cover` or `loadrs` instructions will not encounter this situation.

The RSE will detect the above condition on a `br.ret`, and update its state as follows:

- The register rename base (RSE.BOF), AR[BSP], and AR[BSPSTORE] are updated as required by the return.

- The CFM (after the return) is forced to zero; i.e., all CFM fields (including CFM.sof and CFM.sol) are set to zero.
- The registers from the returned-from frame and the preserved registers from the returned-to frame are added to the invalid partition of the register stack.
- The dirty partition of the register stack is shrunk by AR[PFS].pfm.sol.
- The clean partition of the register stack remains unchanged. RSE.BspLoad and RSE.LoadReg remain unchanged.
- No other indication is given to software.

Since the size of the current frame is set to zero, the contents of some (possibly all) stacked GRs may be overwritten by subsequent eager RSE operations or by subsequent instructions allocating a new stack frame and then targeting a stacked GR. Therefore, explicit register stack management sequences that manipulate PFS, use the `cover` instruction, or use the `loadrs` instruction must avoid this situation by executing one of the two following code sequences prior to a `br.ret`:

- Use a `flushrs` instruction prior to the `br.ret`. This preserves all dirty registers to memory, and sets RSE.ndirty to zero, which avoids the condition.
- Use a `loadrs` instruction with an AR[RSC].loadrs value in the following range:
  AR[RSC].loadrs <= 8*(ndirty_max + ((62 - AR[BSP]{8:3} + ndirty_max) / 63)),
  where ndirty_max = (RSE.N_STACKED_PHYS - (AR[PFS].sof - AR[PFS].sol))

This adjusts the size of the dirty partition appropriately to avoid the condition. A `loadrs` with RSC.loadrs=0 works on all processor models, regardless of the number of implemented stacked physical registers. Note that `loadrs` may cause registers in the dirty partition to be lost.

6.6 RSE Interruptions

Although the RSE runs asynchronously to processor execution, RSE related interruptions are delivered synchronously with the instruction stream. These RSE interruptions are a direct consequence of register stack-related instructions such as: `alloc`, `br.ret`, `rfi`, `flushrs`, `loadrs`, or `mov to/from` BSP, BSPSTORE, RSC, PFS, IFS, or RNAT. Register spills and fills that are executed by the RSE in the background (eager RSE loads or stores) do not raise interruptions. If a faulting/trapping register spill or fill operation is required for software to make forward progress (mandatory RSE load or store) then the RSE will raise an interruption.

Mandatory RSE stores occur in the context of `alloc` and `flushrs` instructions only. Any faults raised by these instructions are delivered on the issuing instruction. Faults raised by mandatory RSE loads caused by a `loadrs` are delivered on the issuing instruction. Mandatory RSE loads which fault while restoring the frame for a `br.ret` or `rfi` deliver the fault on the target instruction, and the ISR.ir (incomplete register frame) bit is set. When a mandatory RSE load faults, AR[BSPSTORE] points to a backing store location above the faulting address reported in CR[IFA]. This allows handlers that service RSE load faults to use the backing store switch routine described in "Switch from Interrupted Context" on page 2:148.

The `br.ret` and the `rfi` instructions set the RSE Current Frame Load Enable bit (RSE.CFLE) to one if the register stack frame being returned to is not entirely contained in the stacked register file. This enables the RSE to restore registers for the current frame of the target instruction. When RSE.CFLE is set, instruction execution is stalled until the RSE has completely restored the current frame or an interruption occurs. This is the only time that the RSE issues any memory traffic for the current frame. Interruption delivery clears RSE.CFLE which allows an interruption handler to execute in the presence of an incomplete frame (e.g., to handle the fault raised by the mandatory RSE load). The RSE.CFLE bit is RSE internal state and is not architecturally visible.

Table 6-6 summarizes RSE raised interruptions.

Table 6-6. RSE Interruption Summary

Instruction	Interruption	Description
`alloc`	Illegal Operation fault	Malformed `alloc` immediate.
`alloc`	Reserved Register/Field fault	`alloc` instruction which attempted to change the size of the rotating region when one or more of the RRB values in CFM were non-zero.
`alloc`, `flushrs`, `loadrs`	Unimplemented Data Address fault	AR[BSPSTORE] contains an unimplemented address.
	Data Nested TLB fault	
	Alternate Data TLB fault	
	VHPT Data fault	
	Data TLB fault	
	Data Page Not Present fault	
	Data NaT Page Consumption fault	AR[BSPSTORE] pointed to a NaTVal data page.
	Data Key Miss fault	
	Data Key Permission fault	
	Data Access Rights fault	
	Data Dirty Bit fault	
	Data Access Bit fault	
	Data Debug fault	
`br.call`, `brl.call`	No RSE related interruptions	
`br.ret`	No RSE load related faults	RSE load related faults are delivered on target instruction.
`rfi`	No RSE related interruptions	RSE load related faults are delivered on target instruction.
Target of `br.ret` or `rfi`	IR Unimplemented Data Address fault	Mandatory RSE load targeted an unimplemented address.
	IR Data Nested TLB fault	`br.ret` with PSR.ic = 0 or `rfi` executed when IPSR.ic = 0.
	IR Alternate Data TLB fault	
	IR VHPT Data TLB fault	
	IR Data TLB fault	
	IR Data Page Not Present fault	
	IR Data NaT Page Consumption fault	RSE.BspLoad pointed at a NaTPage.
	IR Data Key Miss fault	
	IR Data Key Permission fault	
	IR Data Access Rights fault	
	IR Data Access Bit fault	
	IR Data Debug fault	

6.7 RSE Behavior on Interruptions

When the processor raises an interruption, the current register stack frame remains unchanged. If PSR.ic is one, the valid bit in the Interruption Function State register (IFS.v) is cleared. When the IFS.v bit is clear, the contents of the interruption frame marker field (IFS.ifm) are undefined.

While an interruption handler is running and the RSE is in store/load intensive or eager mode, the RSE continues spilling/filling registers to/from the backing store on behalf of the interrupted context as long as the registers are not part of the current frame as defined by CFM.

A sequence of mandatory RSE loads or stores (from `alloc`, `br.ret`, `flushrs`, `loadrs` and `rfi`) can be interrupted by an external interrupt.

When PSR.ic is 0, faults taken on mandatory RSE operations may not be recoverable.

6.8 RSE Behavior with an Incomplete Register Frame

The current register frame is considered **incomplete** when one of the mandatory RSE loads after a br.ret or a rfi faults, leaving BSPSTORE pointing to a location above BSP (i.e., RSE.ndirty_words is negative). The frame becomes complete when RSE.ndirty_words becomes non-negative, either by executing a cover instruction, or by handling the fault and completing the original sequence of mandatory RSE loads.

When the current frame is incomplete the following instructions have undefined behavior: `alloc`, `br.call`, `brl.call`, `br.ret`, `flushrs`, `loadrs`, and move to BSPSTORE. Software must guarantee that the current frame is complete before executing these instructions.

6.9 RSE and ALAT Interaction

The ALAT (see "Data Speculation" on page 1:63) uses physical register addresses to track advanced loads. RSE.BOF may only change as the result of a `br.call` (by CFM.sol), `cover` (by CFM.sof), `br.ret` (by AR[PFM].sol) or `rfi` (by CR[IFS].ifm.sof when CR[IFS].v =1). This ensures, for ALAT invalidation purposes, that hardware does not update virtual to physical register address mapping, unless explicitly instructed to do so by software.

When software performs backing store switches that could cause program values to be placed in different physical registers, then the ALAT must be explicitly invalidated with the `invala` instruction. Typically this happens as part of a process or thread context switch, longjmp or call stack unwind, when software re-writes AR[BSPSTORE], but cannot guarantee that RSE.BOF was preserved.

A stacked register is said to be **deallocated** when an `alloc`, `br.ret`, or `rfi` instruction changes the top of the current frame such that the register is no longer part of the current frame. Once a stacked register is deallocated, its value, its corresponding NaT bit, and its ALAT state are undefined. If that register is subsequently made part of the current frame again (either via another alloc instruction, or via a br.ret or rfi to a previous frame that contained that register), the value stored in the register, the NaT bit for the register, and the corresponding ALAT entry for the register remain undefined.

RSE stores do not invalidate ALAT entries. Therefore, software cannot use the ALAT to trace RSE stores to the backing store.

Note: While an implementation is allowed to remove entries from the ALAT at any time, performance considerations strongly encourage not invalidating ALAT entries due to RSE stores.

6.10 Backing Store Coherence and Memory Ordering

RSE loads and stores are coherent with respect to the processor's data cache at all times. The backing store below BSPSTORE is defined to be consistent with the register stack (the memory image contains consecutive register values and NaT collections). Addresses below BSPSTORE are not modified by the RSE until br.ret, rfi or a move to BSPSTORE causes BSP to drop below the original BSPSTORE value. The RSE never writes to a memory address greater than or equal to BSP.

In order for software to modify a value in the backing store and guarantee that it be loaded by the RSE, software must first place the RSE into enforced lazy mode (RSC.mode=0), and read BSP and BSPSTORE to determine the location of the RSE store pointer. If the location to be modified lies between BSPSTORE and BSP, software must issue a flushrs, update the backing store location in memory, and issue a loadrs instruction with the RSC.loadrs set to zero (this invalidates the current contents of the physical stacked registers, except the current frame, which forces the RSE to reload registers from the backing store). If the location to be modified lies below BSPSTORE, unnecessary memory traffic can be avoided as follows: software must read the RNAT application register, update the backing store location in memory, rewrite BSPSTORE with the original value, and then rewrite RNAT.

RSE loads and stores are weakly ordered. The flushrs and loadrs instructions do not include an implicit memory fence. Turning on and off the RSE does not affect memory ordering. To ensure ordering of RSE loads and stores on a multiprocessor system, software is required to issue explicit memory fence (mf) instructions.

6.11 RSE Backing Store Switches

The implementation of system calls, operating system context switches, user-level thread packages, debugging software, and certain types of exception handling (e.g., setjmp/longjmp, structured exception handling and call stack unwinding) require explicit user-level control of the RSE and/or knowledge of the backing store format in memory. Therefore, the RSE and the backing store can be controlled at all privilege levels.

Three RSE backing store switches are described here:

1. Switching from an interrupted context (as part of exception handler or interrupt bubble-up code)
2. Returning to a previously interrupted context

3. Non-preemptive, synchronous backing store switch (covers system calls, user-level thread and operating system context switches)

Failure to follow these sequences may result in undefined RSE and processor behavior.

6.11.1 Switch from Interrupted Context

To switch from the backing store of an interrupted context to a new backing store:

1. Read and save the RSC and PFS application registers.
2. Issue a `cover` instruction for the interrupted frame.
3. Read and save the IFS control register.
4. Place RSE in enforced lazy mode by clearing both RSC.mode bits.
5. Read and save the BSPSTORE and RNAT application registers.
6. Write BSPSTORE with the new backing store address.
7. Read and save the new BSP to calculate the number of dirty registers.
8. Select the desired RSE setting (mode, privilege level and byte order).

6.11.2 Return to Interrupted Context

To return to the backing store of an interrupted context:

1. Allocate a zero-sized frame.
2. Subtract the BSPSTORE value written in step 6 of Section 6.11.1, "Switch from Interrupted Context" from the BSP value read in step 7 of Section 6.11.1, "Switch from Interrupted Context" on page 2:148, and deposit the difference into RSC.loadrs along with a zero into RSC.mode (to place the RSE into enforced lazy mode).
3. Issue a `loadrs` instruction to insure that any registers from the interrupted context which were saved on the new stack have been loaded into the stacked registers.
4. Restore BSPSTORE from the interrupted context (saved in step 5 of Section 6.11.1, "Switch from Interrupted Context").
5. Restore RNAT from the interrupted context (saved in step 5 of Section 6.11.1, "Switch from Interrupted Context").
6. Restore PFS and IFS from the interrupted context (saved in steps 1 and 3 of Section 6.11.1, "Switch from Interrupted Context").
7. Restore RSC from the interrupted context (saved in step 1 of Section 6.11.1, "Switch from Interrupted Context"). This restores the setting of the RSE mode bits as well as privilege level and byte order.
8. Issue an `rfi` instruction (IFS.ifm will become CFM).

6.11.3 Synchronous Backing Store Switch

A non-preemptive, synchronous backing store switch at any privilege level can be accomplished as follows:

1. Read and save the RSC, BSP and PFS application registers.
2. Issue a `flushrs` instruction to flush the dirty registers to the backing store.
3. Place RSE in enforced lazy mode by clearing both RSC.mode bits.
4. Read and save the RNAT application register.
5. Invalidate the ALAT using the `invala` instruction when switching from code that does not restore RSE.BOF to its original setting. A different RSE.BOF will cause program values in the new context to be placed in different physical registers. See "RSE and ALAT Interaction" on page 2:146 for details.
6. Write the new context's BSPSTORE (was BSP after `flushrs` when switching out).
7. Write the new context's PFS and RNAT.
8. Write the new context's RSC which will set the RSE mode, privilege level and byte order.

6.12 RSE Initialization

At processor reset the RSE is defined to be in enforced lazy mode, i.e., the RSC.mode bits are both zero. The RSE privilege level (RSC.pl) is defined to be zero. RSE.BOF points to physical register 32. The values of AR[PFS].pfm and CR[IFS].ifm are undefined. The current frame marker (CFM) is set as follows: sof=96, sol=0, sor=0, rrb.gr=0, rrb.fr=0, and rrb.pr=0. This gives the processor access to 96 stacked registers.

The RSE performs no spill/fill operations until either an `alloc`, `br.ret`, `rfi`, `flushrs` or `loadrs` require a mandatory RSE operation, or software explicitly enables eager RSE operations. Software must provide the RSE with a valid backing store address in the BSPSTORE application register prior to causing any RSE spill/fill operations. Failure to initialize BSPSTORE results in undefined behavior.

§

Debugging and Performance Monitoring 7

Processors based on the Itanium architecture provide comprehensive debugging and performance monitoring facilities for both IA-32 and Itanium instructions. This chapter describes the debug registers, performance monitoring registers and their programming models. The debugging facilities include several data and instruction break point registers, single step trap, breakpoint instruction fault, taken branch trap, lower privilege transfer trap, instruction and data debug faults. The performance monitoring facilities include two sets of registers to configure and collect various performance-related statistics.

7.1 Debugging

Several Data Breakpoint Registers (DBR) and Instruction Breakpoint Registers (IBR) are defined to hold address breakpoint values for data and instruction references. In addition the following debugging facilities are supported:

- **Single Step trap** – When PSR.ss is 1, successful execution of each Itanium instruction results in a Single Step trap. When PSR.ss is 1 or EFLAG.tf is 1, successful execution of each IA-32 instruction results in an IA_32_Exception(Debug) single step trap. After the trap, IIP and IPSR.ri point to the next instruction to be executed. IIPA and ISR.ei point to the trapped instruction. See "Single Stepping" for complete single stepping behavior.
- **Break Instruction fault** – execution of a `break` instruction results in a Break Instruction fault. IIM is loaded with the immediate operand from the instruction. IIM values are defined by software convention. `break` can be used for profiling, debugging and entry into the operating system (although Enter Privileged Code (`epc`) is recommended since it has lower overhead). Execution of the IA-32 INT 3 (break) instruction results in a IA_32_Exception(Break) trap.
- **Taken Branch trap** – When PSR.tb is 1, a Taken Branch trap occurs on every taken Itanium branch instruction. When PSR.tb is 1, a IA_32_Exception(Debug) taken branch trap occurs on every taken IA-32 branch instruction (CALL, Jcc, JMP, RET, LOOP). This trap is useful for debugging and profiling. After the trap, IIP and IPSR.ri point to the branch target instruction and IIPA and ISR.ei point to the trapping branch instruction.
- **Lower Privilege Transfer trap** – When PSR.lp bit is 1, and an Itanium branch demotes the privilege level (numerically higher), a Lower Privilege Transfer trap occurs. This trap allows for auditing of privilege demotions, for example to remove permissions which were granted to higher privilege code. After the trap, IIP and IPSR.ri point to the branch target and IIPA and ISR.ei point to the trapping branch instruction. IA-32 instructions can not raise this trap.
- **Instruction Debug faults** – When PSR.db is 1, any Itanium instruction memory reference that matches the parameters specified by the IBR registers results in an Instruction Debug fault. Instruction Debug faults are reported even if Itanium instructions are nullified due to a false predicate. If PSR.id is 1, Itanium Instruction Debug faults are disabled for one instruction. The successful execution of an Itanium instruction clears PSR.id. When PSR.db is 1, any IA-32 instruction memory

reference that matches the parameters specified by the IBR registers results in an IA_32_Exception(Debug) fault. If PSR.id is 1 or EFLAG.rf is 1, IA-32 Instruction Debug faults are disabled for one instruction. The successful execution of an IA-32 instruction clears the PSR.id and EFLAG.rf bits.

- **Data Debug faults** – When PSR.db is 1, any Itanium data memory reference that matches the parameters specified by the DBR registers results in a Data Debug fault. Data Debug faults are only reported if the qualifying predicate is true. Data Debug faults can be deferred on speculative loads by setting DCR.dd to 1. If PSR.dd is 1, Data Debug faults are disabled for one instruction or one mandatory RSE memory reference. When PSR.db is 1, any IA-32 data memory reference that matches the parameters specified by the DBR registers results in a IA_32_Exception(Debug) trap. IA-32 data debug events are traps, not faults as defined for the Itanium instruction set. The reported trap code returns the match status of the first 4 DBR registers that matched during the execution of the IA-32 instruction. See "IA-32 Trap Code" on page 2:213 for trap code details. Zero, one or more DBR registers may be reported as matching.

7.1.1 Data and Instruction Breakpoint Registers

Instruction or data memory addresses that match the Instruction or Data Breakpoint Registers (IBR/DBR) shown in Figure 7-1 and Figure 7-2 and Table 7-1 result in an Instruction or Data Debug fault. IA-32 Instruction or data memory addresses that match the Instruction or Data Breakpoint Registers (IBR/DBR) result in an IA_32_Exception(Debug) fault or trap. Even numbered registers contain breakpoint addresses, odd registers contain breakpoint mask conditions. At least 4 data and 4 instruction register pairs are implemented on all processor models. Implemented registers are contiguous starting with register 0.

Figure 7-1. Data Breakpoint Registers (DBR)

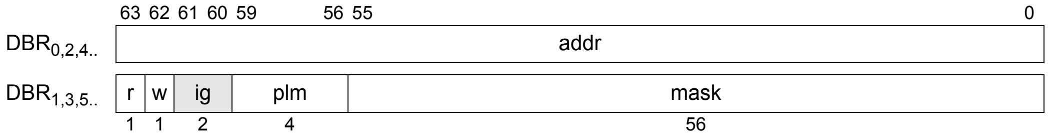
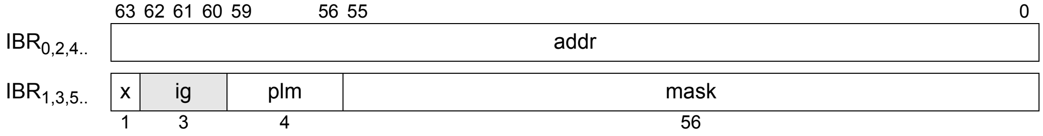


Figure 7-2. Instruction Breakpoint Registers (IBR)



When executing Itanium instructions, instruction and data memory addresses presented for matching are always in the implemented address space. Programming an unimplemented physical address into an IBR/DBR guarantees that physical addresses presented to the IBR/DBR will never match. Similarly, programming an unimplemented virtual address into an IBR/DBR guarantees that virtual addresses presented to the IBR/DBR will never match.

Table 7-1. Debug Breakpoint Register Fields (DBR/IBR)

Field	Bits	Description
addr	63:0	Match Address – 64-bit virtual or physical breakpoint address. Addresses are interpreted as either virtual or physical based on PSR.dt, PSR.it or PSR.rt. Data breakpoint addresses trap on load, store, semaphore, and mandatory RSE memory references. For Intel Itanium instruction set references, IBR.addr{3:0} is ignored in the address match. For IA-32 instruction references, IBR.addr{31:0} are used in the match and IBR.addr{63:32} must be zero to match. All 64 bits are implemented on all processors regardless of the number of implemented address bits.
mask	55:0	Address Mask – determines which address bits in the corresponding address register are compared in determining a breakpoint match. Address bits whose corresponding mask bits are 1, must match for the breakpoint to be signaled, otherwise the address bit is ignored. Address bits{63:56} for which there are no corresponding mask bits are always compared. For IA-32 instruction references, IBR.mask{55:32} are ignored. All 56 bits are implemented on all processors regardless of the number of implemented address bits.
plm	59:56	Privilege Level Mask – enables data breakpoint matching at the specified privilege level. Each bit corresponds to one of the four privilege levels, with bit 56 corresponding to privilege level 0, bit 57 with privilege level 1, etc. A value of 1 indicates that the debug match is enabled at that privilege level.
w	62	Write match enable – When DBR.w is 1, any non-nullified mandatory RSE store, IA-32 or Intel Itanium store, semaphore, probe.w.fault or probe.rw.fault to an address matching the corresponding address register causes a breakpoint.
r	63	Read match enable – When DBR.r is 1, any non-nullified IA-32 or Intel Itanium load, mandatory RSE load, semaphore, lfetch.fault, probe.r.fault or probe.rw.fault to an address matching the corresponding address register causes a breakpoint. When DBR.r is 1, a VHPT access that matches the DBR (except those for a `tak` instruction) will cause an Instruction/Data TLB Miss fault. If DBR.r and DBR.w are both 0, that data breakpoint register is disabled.
x	63	Execute match enable – When IBR.x is 1, execution of an IA-32 instruction or Intel Itanium instruction in a bundle at an address matching the corresponding address register causes a breakpoint. If IBR.x is 0, that instruction breakpoint register is disabled. Instruction breakpoints are reported even if the qualifying predicate is false.
ig	62:60	Ignored

Four privileged instructions, defined in Table 7-2, allow access to the debug registers. Register access is indirect, where the debug register number is determined by the contents of a general register. DBR/IBR registers can only be accessed at privilege level 0, otherwise a Privileged Operation fault is raised.

Table 7-2. Debug Instructions

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov dbr[r_3] = r_2	Move to data breakpoint register	DBR[GR[r_3]] ← GR[r_2]	M	data
mov r_1 = dbr[r_3]	Move from data breakpoint register	GR[r_1] ← DBR[GR[r_3]]	M	none
mov ibr[r_3] = r_2	Move to instruction breakpoint register	IBR[GR[r_3]] ← GR[r_2]	M	inst
mov r_1 = ibr[r_3]	Move from instruction breakpoint register	GR[r_1] ← IBR[GR[r_3]]	M	none

break *imm*	Breakpoint Instruction fault	if (PSR.ic) IIM ← *imm*
		fault(Breakpoint_Instruction)
			B/I/M	none

Changes to debug registers and PSR are not necessarily observed by following instructions. Software should issue a data serialization operation to ensure modifications to DBR, PSR.db, PSR.tb and PSR.lp are observed before a dependent instruction is executed. For register changes to IBR and PSR.db that affect fetching of subsequent instructions, software must issue an instruction serialization operation.

On some implementations, a hardware debugger may use two or more of these registers pairs for its own use. When a hardware debugger is attached, as few as 2 DBR pairs and as few as 2 IBR pairs may be available for software use. Software should be prepared to run with fewer than the implemented number of IBRs and/or DBRs if the software is expected to be debuggable with a hardware debugger. When a hardware debugger is not attached, at least 4 IBR pairs and 4 DBR pairs are available for software use.

Any debug registers used by an attached hardware debugger are allocated from the highest register numbers first (e.g. if only 2 DBR pairs are available to software, the available registers are DBR[0-3]).

Note: When a hardware debugger is attached and is using two or more debug registers pairs, the processor does not forcibly partition the registers between software and hardware debugger use; that is, the processor does not prevent software from reading or modifying any of the debug registers being used by the hardware debugger. However, if software modifies any of the registers being used by the hardware debugger, processor and/or hardware debugger operation may become undefined, or the processor and/or hardware debugger may crash.

7.1.2 Debug Address Breakpoint Match Conditions

For virtual memory accesses, breakpoint address registers contain the virtual addresses of the debug breakpoint. For physical accesses, the addresses in these registers are treated as a physical address. Software should be aware that debug registers configured to fault on virtual references, may also fault on a physical reference if translations are disabled. Likewise a debug register configured for physical references can fault on virtual references that match the debug breakpoint registers.

The range of addresses detected by the DBR and IBR registers for memory references by Itanium instructions is defined as:

- Instruction and single or multi-byte aligned data memory references that access any memory byte specified by the IBR/DBR address and mask fields results in an Instruction/Data Debug fault regardless of datum size. Implementations must only report a Debug fault if the specified aligned byte(s) are referenced.
- Floating-point load double/integer pair, floating-point spill/fill and 10-byte operands are treated as 16-byte datums for breakpoint matching, if the accesses are aligned. Floating-point load single pair operands are treated as 8-byte datums for breakpoint matching, if the accesses are aligned.
- If data memory references are unaligned, multi-byte memory references that access any memory byte specified by DBR address and mask fields result in a breakpoint Data Debug fault regardless of datum size. Processor implementations may also report additional breakpoint Data Debug faults for addresses not specifically specified by the DBR registers. Debugging software should perform a byte by byte breakpoint analysis of each address accessed by multi-byte unaligned datums to detect true breakpoint conditions.

- The `cmp8xchg16` operands are treated as 16-byte datums for both read and write breakpoint matching, even though this instruction only reads 8 bytes.

Address breakpoint Data Debug faults are not reported for the Flush Cache (`fc`, `fc.i`), regular_form `probe`, non-faulting `lfetch`, insert TLB (`itc`, `itr`), purge TLB (`ptc`, `ptr`), or translation access (`thash`, `ttag`, `tak`, `tpa`) instructions. Accesses by the RSE to a debug region are checked, but the Data Debug fault is not reported until a subsequent `alloc`, `br.ret`, `rfi`, `loadrs`, or `flushrs` which requires that the faulting load or store actually occur.

The range of addresses detected by the DBR and IBR registers for IA-32 memory references is defined as:

- Instruction memory references where the first byte of the IA-32 instruction match the IBR address and mask fields results in an IA_32_Exception(Debug) fault. Subsequent bytes of a multiple byte IA-32 instruction are not compared against the IBR registers for breakpoints. The upper 32-bits of the IBR addr field must be zero to detect IA-32 instruction memory references.
- IA-32 single or multi-byte data memory references that access any memory byte specified by the DBR address and mask fields results in an IA_32_Exception(Debug) trap regardless of datum size and alignment. The processor ensures that all data breakpoint traps are precisely reported. Data breakpoint traps are reported if and only if any byte in the IA-32 data memory reference matches the DBR address and mask fields. No spurious data breakpoint events are generated for IA-32 data memory operands that are unaligned, nor are breakpoints reported if no bytes of the operand lie within the address range specified by the DBR address and mask fields.

7.2 Performance Monitoring

Performance monitors allow processor events to be monitored by programmable counters or give an external notification (such as a pin or transaction) on the occurrence of an event. Monitors are useful for tuning application, operating system and system performance. Two sets of performance monitor registers are defined. Performance Monitor Configuration (PMC) registers are used to control the monitors. Performance Monitor Data (PMD) Registers either provide data values from the monitors, or hold data values used by the PMU. The performance monitors can record performance values from either the IA-32 or Itanium instruction set.

As shown in Figure 7-3, all processor implementations provide at least four performance counters (PMC/PMD[4]..PMC/PMD[7] pairs), and four performance counter overflow status registers (PMC[0]..PMC[3]). Performance monitors are also controlled by bits in the processor status register (PSR), the default control register (DCR) and the performance monitor vector register (PMV). Processor implementations may provide additional implementation-dependent PMC and PMD registers to increase the number of “generic” performance counters (PMC/PMD pairs). The remainder of the PMC and PMD register set is implementation dependent.

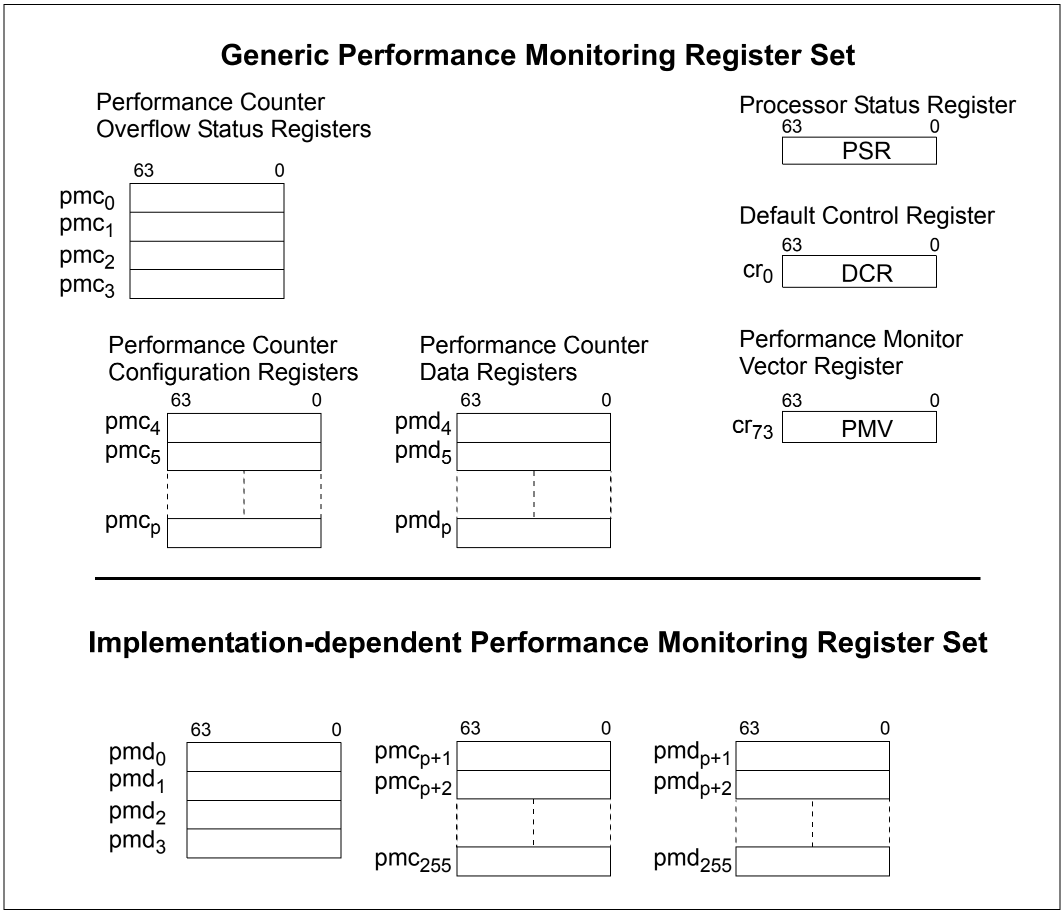
Event collection for implementation-dependent performance monitors is not specified by the architecture. Enabling and disabling functions are implementation dependent. For details, consult processor-specific documentation.

Processor implementations may not populate the entire PMC/PMD register space. Reading of an unimplemented PMC or PMD register returns zero. Writes to unimplemented PMC or PMD registers are ignored; i.e., the written value is discarded.

Writes to PMD and PMC and reads from PMC are privileged operations. At non-zero privilege levels, these operations result in a Privileged Operation fault, regardless of the register address.

Reading of PMD registers by non-zero privilege level code is controlled by PSR.sp. When PSR.sp is one, PMD register reads by non-zero privilege level code return zero.

Figure 7-3. Performance Monitor Register Set



7.2.1 Generic Performance Counter Registers

Generic performance counter registers are PMC/PMD pairs that contiguously populate the PMC/PMD name space starting at index 4. At least 4 performance counter register pairs (PMC/PMD[4]..PMC/PMD[7]) are implemented in all processor models. Each counter can be configured to monitor events for any combination of privilege levels and one of several event metrics. The number of performance counters is implementation specific. The figures and tables use the symbol "p" to represent the index of the last implemented generic PMC/PMD pair. The bit-width W of the counters is also implementation specific.

A counter overflow interrupt occurs when the counter wraps; i.e., a carry out from bit W-1 is detected. Counter overflow interrupts are edge-triggered; that is, the event of a counter incrementing and causing carry out from bit W-1 thus setting the overflow bit and the freeze bit, generates one PMU interrupt. Provided that software does not clear the freeze bit, while either or both of PSR.up and pp are 1, without also clearing the overflow bit (before or concurrent with the write to the freeze bit), no further interrupts are generated based on the fact that the carry out had been earlier detected.

Figure 7-4 and Figure 7-5 show the fields in PMD and PMC respectively, while Table 7-3 and Table 7-4 describe the fields in PMD and PMC respectively.

Figure 7-4. Generic Performance Counter Data Registers (PMD[4]..PMD[p])

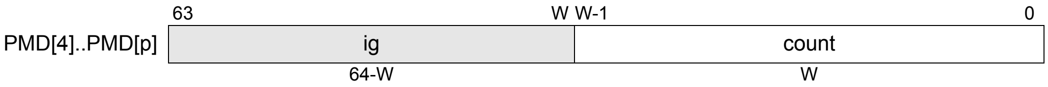


Table 7-3. Generic Performance Counter Data Register Fields

Field	Bits	Description
ig	63:W	Writes are ignored. Reads return 0.
count	W-1:0	Event Count. The counter is defined to overflow when the count field wraps (carry out from bit W-1).

Some implementations do not treat the upper, unimplemented bits of PMDs as ignored bits on reads, but rather return a copy of bit W-1 in these bit positions so that count values appear as if they were sign extended. Subsequent implementations will return 0 for these bits on reads.

Figure 7-5. Generic Performance Counter Configuration Register (PMC[4]..PMC[p])

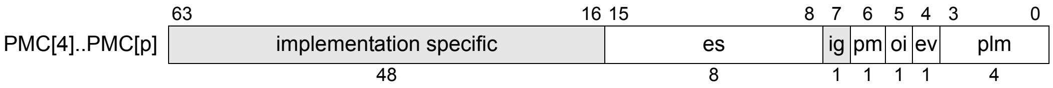


Table 7-4. Generic Performance Counter Configuration Register Fields (PMC[4]..PMC[p])

Field	Bits	Description
plm	3:0	Privilege Level Mask – controls performance monitor operation for a specific privilege level. Each bit corresponds to one of the 4 privilege levels, with bit 0 corresponding to privilege level 0, bit 1 with privilege level 1, etc. A bit value of 1 indicates that the monitor is enabled at that privilege level. Writing zeros to all plm bits effectively disables the monitor. In this state, the corresponding PMD register(s) do not preserve values, and the processor may choose to power down the monitor.
ev	4	External visibility – When 1, an external notification (such as a pin or transaction) may be provided, dependent upon implementation, whenever the monitor overflows. Overflow occurs when a carry out from bit W-1 is detected.
oi	5	Overflow interrupt – If 1, when the monitor overflows, a Performance Monitor Interrupt is raised and the performance monitor freeze bit (PMC[0].fr) is set. If 0, no interrupt is raised and the performance monitor freeze bit (PMC[0].fr) remains unchanged. Overflow occurs when a carry out from bit W-1 is detected. See “Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])” for details on configuring interrupt vectors.

Table 7-4. Generic Performance Counter Configuration Register Fields (PMC[4]..PMC[p]) (Continued)

Field	Bits	Description
pm	6	Privileged monitor – When 0, the performance monitor is configured as a user monitor, and enabled by PSR.up. When PMC.pm is 1, the performance monitor is configured as a privileged monitor, enabled by PSR.pp, and the corresponding PMD can only be read by privileged software.
ig	7	ignored
es	15:8	Event select – selects the performance event to be monitored. Actual event encodings are implementation dependent. Some processor models may not implement all event select (es) bits. At least one bit of es must be implemented on all processors. Unimplemented es bits are ignored.
implem. specific	63:16	Implementation-specific bits – Reads from implemented bits return implementation-dependent values. For portability, software should write what was read; i.e., software may not use these bits as storage. Hardware will ignore writes to unimplemented bits.

Event collection is controlled by the Performance Monitor Configuration (PMC) registers and the processor status register (PSR). Four PSR fields (PSR.up, PSR.pp, PSR.cpl and PSR.sp) and the performance monitor freeze bit (PMC[0].fr) affect the behavior of all generic performance monitor registers. Finer, per monitor, control of generic performance monitors is provided by two PMC register fields (PMC[i].plm, PMC[i].pm). Event collection for a generic monitor is enabled under the following constraints:

- Generic Monitor Enable[i] =(not PMC[0].fr) and PMC[i].plm[PSR.cpl] and ((not (PMC[i].pm) and PSR.up) or (PMC[i].pm and PSR.pp))

Generic performance monitor data registers (PMD[i]) can be configured to be user readable (useful for user level sampling and tracing user level processes) by setting the PMC[i].pm bit to 0. All user-configured monitors can be started and stopped synchronously by the user mask instructions (`rum` and `sum`) by altering PSR.up. User-configured monitors can be secured by setting PSR.sp to 1. A user-configured secured monitor continues to collect performance values; however, reads of PMD, by non-privileged code, return zeros until the monitor is unsecured.

Monitors configured as privileged (PMC[i].pm is 1) are accessible only at privilege level 0; otherwise, reads return zeros. All privileged monitors can be started and stopped synchronously by the system mask instructions (`rsm` and `ssm`) by altering PSR.pp. Table 7-5 summarizes the effects of PSR.sp, PMC[i].pm, and PSR.cpl on reading PMD registers.

Updates to generic PMC registers and PSR bits (up, pp, is, sp, cpl) require implicit or explicit data serialization prior to accessing an affected PMD register. The data serialization ensures that all prior PMD reads and writes as well as all prior PMC writes have completed.

Table 7-5. Reading Performance Monitor Data Registers

PSR.sp	PMC[i].pm	PSR.cpl	PMD Reads Return
0	0	0	PMD value
0	1	0	PMD value
1	0	0	PMD value
1	1	0	PMD value
0	0	>0	PMD value

Table 7-5. Reading Performance Monitor Data Registers (Continued)

PSR.sp	PMC[i].pm	PSR.cpl	PMD Reads Return
0	1	>0	0
1	0	>0	0
1	1	>0	0

Generic PMD counter registers may be read by software without stopping the counters. Under normal counting conditions (PMC[0].fr is zero and has been serialized), the processor guarantees that a sequence of reads of a given PMD will return non-decreasing values corresponding to the program order of the reads. Under frozen count conditions (PMC[0].fr is one and has been serialized), the counters are unchanging and ordering is irrelevant. When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed.

Software must accept a level of sampling error when reading the counters due to various machine stall conditions, interruptions, and bus contention effects, etc. The level of sampling error is implementation specific. More accurate measurements can be obtained by disabling the counters and performing an instruction serialize operation for instruction events or data serialize operation for data events before reading the monitors. Other (non-counter) implementation-dependent PMD registers can only be read reliably when event monitoring is frozen (PMC[0].fr is one).

For accurate PMD reads of disabled counters, data serialization (implicit or explicit) is required between any PMD read and a subsequent `ssm` or `sum` (that could toggle PSR.up or PSR.pp from 0 to 1), or a subsequent `epc`, demoting `br.ret` or branch to IA-32 (`br.ia`) (that could affect PSR.cpl or PSR.is). Note that implicit post-serialization semantics of `sum` do not meet this requirement.

Table 7-6 defines the instructions used to access the PMC and PMD registers.

Table 7-6. Performance Monitor Instructions

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov pmd[r_3] = r_2	Move to performance monitor data register	PMD[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmd[r_3]	Move from performance monitor data register	GR[r_1] ← PMD[GR[r_3]]	M	none[a]
mov pmc[r_3] = r_2	Move to performance monitor configure register	PMC[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmc[r_3]	Move from performance monitor configure register	GR[r_1] ← PMC[GR[r_3]]	M	none

[a] When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed.

7.2.2 Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])

Performance monitor interrupts may be caused by an overflow from a generic performance monitor or an implementation-dependent event from a model-specific monitor. The four performance monitor overflow registers (PMC[0]...PMC[3]) shown in Figure 7-6 indicate which monitor caused the interruption.

Each of the 252 overflow bits in the performance monitoring overflow status registers(PMC[0]...PMC[3]) corresponds to a generic performance counter pair or to an implementation-dependent monitor. For generic performance counter pairs, overflow status bit PMC[i/64]{i%64} corresponds to generic counter pair PMC[i]/PMD[i], where 4<=i<=p, and p is the index of the last implemented generic PMC/PMD pair.

There are currently two criteria for generating a performance monitor interrupt:

1. A generic performance counter pair (PMC[n]/PMD[n]) overflows and its overflow interrupt bit (PMC[n].oi) is 1.
2. An implementation-dependent monitor wants to report an event with an interruption.

If any of these criteria are met, the processor will:

- Set the corresponding overflow status bit in PMC[0]..PMC[3] to 1, and
- Raise a Performance Monitor interrupt, and
- Set the freeze bit (PMC[0].fr) which suspends event monitoring.

PMU interrupts are generated by events, such as the overflowing of a generic counter pair which is configured to interrupt on overflow. Each such event generates one interrupt. Provided that software does not clear the freeze bit, while either or both of PSR.up and pp are 1, before clearing the overflow bits, writes to PMCs and PMDs by software do not generate interrupts, nor cause a monitor which had generated an interrupt to generate a second interrupt. (For overflow bits in PMC 0, even if either or both of PSR.up and .pp are 1, software may clear the overflow bits and the freeze bit with a single write to PMC 0 without causing any additional interrupts to be generated.)

Software may restore PMU state which has the freeze bit equal to 1 and one or more overflow bits equal to 1 without generating any interrupts provided that it ensures either that:

- both PSR.up and pp are zero during the restore, or
- the freeze bit is a 1 (and serialized) before any overflow bits are set to 1

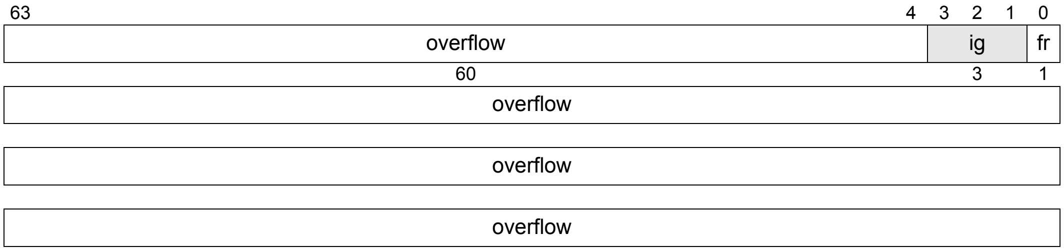
When the PMU is disabled by writing a 0 into PSR.up and .pp and serializing this write, the PMU cannot generate any interrupts and no SW writes to any PMU state can cause any interrupts.

When a generic performance counter pair (PMC[n]/PMD[n]) overflows and its overflow interrupt bit (PMC[n].oi) is 0, the corresponding overflow status register bit is set to 1. However, in this case of counter overflow without interrupt, the freeze bit in the PMC[0] is left unchanged, and event monitoring continues.

If control register bit PMV.m is one, a performance monitoring interrupt is disabled from being pended. When PMV.m is zero, the interruption is received and held pending. (Further masking by the PSR.i, TPR and in-service masking can keep the interrupt from being raised.) Figure 7-6 shows the Performance Monitor Overflow Status registers.

Implementation dependent PMD registers (0-3) cannot report events in the overflow registers; those 4 bit positions are used for other purposes.

Figure 7-6. Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])



Under frozen count conditions when PMC[0].fr is one (either by a performance counter overflow, or an explicit software write and serialization), the processor suspends all event monitoring, i.e. counters do not increment and overflow bits as well as model-specific monitoring are frozen. Normal counting conditions are restored by software writing a zero to the freeze bit and serializing to resume event monitoring. When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed.

Table 7-7. Performance Monitor Overflow Register Fields (PMC[0]...PMC[3])

Register	Field	Bits	Description
PMC[0]	fr	0	Performance Monitor “freeze” bit. This bit is volatile state, i.e., it is set by the processor whenever: • a generic performance monitor overflow occurs and its overflow interrupt bit (PMC[n].oi) is set to one. • a model-specific performance monitor signals an interrupt. The freeze bit can also be set by software to enable or disable all event monitoring. If the freeze bit is one, event monitoring is disabled. If the freeze bit is zero, event monitoring is enabled. If the freeze bit is in-flight, event monitoring behavior is implementation dependent.
PMC[0]	ig	3:1	Ignored
PMC[0]..PMC[3]	overflow	implemented monitors	Bit vector indicating which performance monitor overflowed. Overflow status bits are sticky, they are set to 1 by the processor if the corresponding PMD overflows; otherwise they are left unchanged. Multiple overflow status bits may be set, independent of whether counter overflow causes an interrupt or not.
		unimplemented monitors	Ignored

Multiple overflow bits may be set to 1, if counters overflow concurrently. The overflow bits and the freeze bit are sticky; i.e., the processor sets them to 1 but never resets them to 0. It is software's responsibility to reset the overflow and freeze bits.

The overflow status bits are populated only for implemented counters. Overflow bits of unimplemented counters read as zero and writes are ignored.

7.2.3 Performance Monitor Events

The set of monitored events is implementation-specific. All processor models are required to provide at least two events:

1. The number of retired instructions. These are defined as all instructions which execute without a fault, including nops and those which were predicated off. Generic counters configured for this event count only when the processor is in the NORMAL or LOW-POWER state (see Figure 11-8 on page 2:314).
2. The number of processor clock cycles. Generic counters configured for this event count only when the processor is in the NORMAL or LOW-POWER state (see Figure 11-8 on page 2:314).

Events may be monitorable only by a subset of the available counters. PAL calls provide an implementation-independent interface that provides information on the number of implemented counters, their bit-width, the number and location of other (non-counter) monitors, etc.

7.2.4 Implementation-independent Performance Monitor Code Sequences

This section describes implementation-independent code sequences for servicing overflow interrupts and context switches of the performance monitors. For forward compatibility, the code sequences outlined in Section 7.2.4.1 and Section 7.2.4.2 use PAL-provided implementation-specific information to collect/preserve data values for all implemented counters.

7.2.4.1 Performance Monitor Interrupt Service Routine

When a generic performance counter pair (PMC[n]/PMD[n]) overflows and its overflow interrupt bit (PMC[n].oi) is 1, or an implementation-dependent monitor wants to report an event with an interruption, then the processor:

- Sets the corresponding overflow status bit in PMC[0]..PMC[3] to one,
- Raises a Performance Monitor Interrupt, and
- Sets the freeze bit in PMC[0] which suspends event monitoring.

Event monitoring remains frozen until software clears the freeze bit. When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed. Performance monitor interrupts may be caused by an overflow of any of the counters. The processor indicates which performance monitor overflowed in the performance monitor overflow status registers (PMC[0]...PMC[3]). If multiple counters overflow concurrently, multiple overflow bits will be set to one. For forward compatibility, event collection interrupt handlers must follow the implementation-independent overflow interrupt service routine outlined in Figure 7-7. Use of alternate context-switch sequences may be incompatible with future implementations.

If the outgoing context has an interrupt pending but has not yet invoked the performance monitor interrupt service routine, the interrupt may be delivered to the incoming context even if it is a non-monitored process. The interrupt service routine can recognize this kind of bogus interrupt by noticing that either: the freeze bit is zero or the context is not being monitored.

Figure 7-7. Performance Monitor Interrupt Service Routine (Implementation Independent)

```
//Assumes PSR.up and PSR.pp are switched to zero together
if ((PMC[0].fr==1) && (PSR.up == 1) || (PSR.pp == 1)){
    // freeze bit is set. Search for interrupt.
    for (i=0; i< 4; i++) {
        if (PMC[i] != 0) {
            startbit = (i==0) ? 4 : 0;
            for (j=startbit; j < 64 ; j++) {
                if (PMC[i]{j}) {
                    counter_id = 64*i + j;
                    if (counter_id > PAL_GENERIC_PMCPMD_PAIRS) {
                        Implementation_Specific_Update(counter_id);
                    }
                    else { // Generic PMC/PMD counter
                        if (PMC[counter_id].oi)
                            ovflcount[counter_id] += 1;
                    }
                }
            } // scan overflow bits
        }
    }
}
// Either ignore bogus interrupt or clear PMC[3]..PMC[1]
for (i=3; i>=1; i--) { PMC[i] = 0; }
rfi
```

7.2.4.2 Performance Monitor Context Switch

The context switch routine described in Figure 7-8 defines the implementation-independent context switching of Itanium performance monitors. Using bit masks provided by PAL ($PAL_{PMCmask}$, $PAL_{PMDmask}$) the routine can generically save/restore the contents of all implementation-specific performance monitoring registers. If the outgoing context is monitored, then all PMC and PMD registers whose mask bit is set are preserved by software. But if the outgoing context is monitored and the context switch routine determines that the outgoing context has a pending performance monitor interrupt (by reading the freeze bit with the knowledge that it was not generated by software) then software also preserves the outgoing context's overflow status registers (PMC[0]..PMC[3]) before all PMC and PMD registers whose mask bit is set. Here, it is explicitly assumed that software tracks monitored processes and can determine whether a process is monitored prior to reading the freeze bit. The context switch handler then restores the performance monitor freeze bit which resets event collection for the new context. Sometime into the incoming (possibly unmonitored) context, the performance overflow interrupt service routine will run, but by looking at the status of the freeze bit software can determine whether this interrupt can be ignored (for details refer to Section 7.2.4.1).

When switching back to the original context (that originally caused the counter overflow), the previously saved freeze bit can be inspected. If it was set (meaning there was a pending performance monitor interrupt), then the context switch routine posts an interrupt message to the incoming context's processor at the performance monitor vector specified by the PMV register (see Section 10.5.8, "Inter-processor Interrupts Layout and Example" on page 2:612). This will result in a new performance monitor overflow interrupt in the correct context. Essentially, the interrupt message is "replaying" the overflow interrupt that was missed because of the context switch.

Figure 7-8. Performance Monitor Overflow Context Switch Routine

```
// in context or thread switch

if (outgoing process is monitored) {
    1. Turn-off counting and ignore interrupts for context switch
        of counters.
        1a)     if not already done, raise interrupt priority above
                perf. mon overflow vector
        1b)     read and preserve PSR.up, PSR.pp, PSR.sp
        1c)     clear PSR.up, clear PSR.pp
        1d)     srlz.d
    2. Preserve PMC/PMD contents
        2a)  For each PMC whose PALPMCmask bit is set, preserve PMC.
        2b) For each PMD whose PALPMDmask bit is set, preserve PMD.
}

.... continue context switch ......

// Now in incoming process/thread
if (incoming process is monitored) {
    // Event counting is disabled because PSR.up and pp are both
    // zero (step 1c above).

    3. Restore PMC/PMD contents (inverse of step 4 above)
        3a) For each PMC whose PALPMCmask bit is set, reload PMC.
        3b) For each PMD whose PALPMDmask bit is set, reload PMD.

    4. Restore Interrupt State (inverse of step 2 and 1a above)
        4a)     if (PMC[0].fr) {
                    send myself a performance monitor interrupt
                    (store to interrupt address)
                }
        4b)     Restore PSR.up and PSR.pp
        4c)     srlz.d
        4d)     lower interrupt priority below perf. mon overflow
                vector
}
```

§

Interruption Vector Descriptions 8

Chapter 5 describes the interruption mechanism and programming model for the Itanium architecture. This chapter describes the IVA-based interruption handlers. "Interruption Vector Descriptions" describes all the Itanium IVA-based interruption vectors and "IA-32 Interruption Vector Definitions" describes all of the IA-32 interrupt vectors. PAL-based interruptions are described in Chapter 11, "Processor Abstraction Layer." Note that unless otherwise noted, references to "interruption" in this chapter refer to IVA-based interruptions. See "Interruption Definitions" on page 2:95.

8.1 Interruption Vector Descriptions

The section lists all the Itanium interruption vectors. It describes the interruption vectors and the parameters that are defined when the vector is entered.

If an interruption is independent of the executing instruction set (including IA-32), such as an external interrupt or TLB fault, common Itanium interruption vectors are used. For exceptions and intercept conditions that are specific to the IA-32 instruction set three IA-32 specific vectors are used; IA_32_Exception, IA_32_Interrupt, and IA_32_Intercept.

Table 8-1 defines which interruption resources are written, are left unmodified, or are undefined for each interruption vector. The individual vector descriptions below list interruption-specific resources for each vector.

See "IVA-based Interruption Handling" on page 2:101 for details on how the processor handles an interruption. See "Interruption Control Registers" on page 2:36 for the definition of bit fields within the interruption resources.

8.2 ISR Settings

For each of the interruption vectors, a figure depicts the ISR setting. These figures show the value that hardware writes into the ISR for the corresponding interruption.

Table 8-2 provides an overview of ISR settings for all of the interruption vectors.

For some of the vectors, certain bits will always be 0 (or 1) simply because no instruction that would set that bit differently can ever end up on that vector. For example, ISR.sp is always 0 in the Break Instruction vector because ISR.sp is only set by speculative loads, and speculative loads can never take a Break Instruction fault.

After interruption from the IA-32 instruction set, the following ISR bits will always be zero: ISR.ni, ISR.na, ISR.sp, ISR.rs, ISR.ir, ISR.ei, and ISR.ed.

ISR.code settings for non-access instructions are described in "Non-access Instructions and Interruptions" on page 2:103.

Table 8-3 on page 2:170 provides an overview of ISR.code field on all Itanium traps.

8.3 Interruption Vector Definition

Table 8-1.Writing of Interruption Resources by Vector

Interruption Resource	IIP, IPSR, IIPA, IFS.v		IFA		ITIR		IHA		IIM		ISR		IIB0, IIB1	
PSR.ic at time of interruption	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
Alternate Data TLB vector														
Alternate Data TLB fault	N/A[a]	W[b]	N/A	W	N/A	W	N/A	x[c]	N/A	x	N/A	W	N/A	W
IR Alternate Data TLB fault	N/A	W	N/A	W	N/A	W	N/A	x	N/A	x	N/A	W	N/A	x
Alternate Instruction TLB vector														
Alternate Instruction TLB fault	-[d]	W	-	W	-	W	x	x	x	x	W	W	-	x
Break Instruction vector														
Break Instruction fault	-	W	x	x	x	x	x	x	-	W	W	W	-	W
Data Access Rights vector														
Data Access Rights fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
IR Data Access Rights fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Data Access-Bit vector														
Data Access Bit fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
IR Data Key Miss fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Data Key Miss vector														
Data Key Miss fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
IR Data Key Miss fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Data Nested TLB vector														
Data Nested TLB fault	-	N/A	-	N/A	-	N/A	-	N/A	x	N/A	-	N/A	-	N/A
IR Data Nested TLB fault	-	N/A	-	N/A	-	N/A	-	N/A	x	N/A	-	N/A	-	N/A
Data TLB vector														
Data TLB fault	N/A	W	N/A	W	N/A	W	N/A	W	N/A	x	N/A	W	N/A	W
IR Data TLB fault	N/A	W	N/A	W	N/A	W	N/A	W	N/A	x	N/A	W	N/A	x
Debug vector														
Data Debug fault	-	W	-	W	x	x	x	x	x	x	W	W	-	W
Instruction Debug fault	-	W	-	W	x	x	x	x	x	x	W	W	-	x
IR Data Debug fault	-	W	-	W	x	x	x	x	x	x	W	W	-	x
Dirty-Bit vector														
Data Dirty Bit fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
Disabled FP-Register vector														
Disabled Floating-Point Register fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
External Interrupt vector														
External Interrupt	-	W	x	x	x	x	x	x	x	x	W	W	-	x
Floating-point Fault vector														
Floating-Point Exception fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Floating-point Trap vector														
Floating-Point Exception trap	-	W	x	x	x	x	x	x	x	x	W	W	-	W
General Exception vector														
Disabled ISA Transition fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Illegal Dependency fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Illegal Operation fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
IR Unimplemented Data Address fault	-	W	x	x	x	x	x	x	x	x	W	W	-	x
Privileged Operation fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Privileged Register fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W

Table 8-1.Writing of Interruption Resources by Vector (Continued)

Interruption Resource	IIP, IPSR, IIPA, IFS.v		IFA		ITIR		IHA		IIM		ISR		IIB0, IIB1	
PSR.ic at time of interruption	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Reserved Register/Field fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Unimplemented Data Address fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W
IA-32 Exception vector	N/A	W	N/A	x	N/A	x	N/A	x	N/A	x	N/A	W	N/A	x
IA-32 Intercept vector	N/A	W	N/A	x	N/A	x	N/A	x	N/A	W	N/A	W	N/A	x
IA-32 Interrupt vector	N/A	W	N/A	x	N/A	x	N/A	x	N/A	x	N/A	W	N/A	x
Instruction Access Rights vector														
Instruction Access Rights fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Instruction Access-Bit vector														
Instruction Access Bit fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Instruction Key Miss vector														
Instruction Key Miss fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Instruction TLB vector														
Instruction TLB fault	-	W	-	W	-	W	-	W	x	x	W	W	-	x
Key Permission vector														
Data Key Permission fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
Instruction Key Permission fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
IR Data Key Permission fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Lower-Privilege Transfer Trap vector														
Unimplemented Instruction Address fault	-	W	x	W	x	x	x	x	x	x	W	W	-	x
Lower-Privilege Transfer trap	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Unimplemented Instruction Address trap	-	W	x	x	x	x	x	x	x	x	W	W	-	W
NaT Consumption vector														
Data NaT Page Consumption fault	-	W	-	W	x	x	x	x	x	x	W	W	-	W
Instruction NaT Page Consumption fault	-	W	-	W	x	x	x	x	x	x	W	W	-	x
IR Data NaT Page Consumption fault	-	W	-	W	x	x	x	x	x	x	W	W	-	x
Register NaT Consumption fault	-	W	-	x	x	x	x	x	x	x	W	W	-	W
Page Not Present vector														
Data Page Not Present fault	-	W	-	W	-	W	x	x	x	x	W	W	-	W
Instruction Page Not Present fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
IR Data Page Not Present fault	-	W	-	W	-	W	x	x	x	x	W	W	-	x
Single Step Trap vector														
Single Step trap	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Speculation vector														
Speculative Operation fault	-	W	x	x	x	x	x	x	-	W	W	W	-	W
Taken Branch Trap vector														
Taken Branch trap	-	W	x	x	x	x	x	x	x	x	W	W	-	W
Unaligned Reference vector														

Table 8-1.Writing of Interruption Resources by Vector (Continued)

Interruption Resource	IIP, IPSR, IIPA, IFS.v		IFA		ITIR		IHA		IIM		ISR		IIB0, IIB1	
PSR.ic at time of interruption	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Unaligned Data Reference fault	-	W	-	W	x	x	x	x	x	x	W	W	-	W
Unsupported Data Reference vector														
Unsupported Data Reference fault	-	W	-	W	x	x	x	x	x	x	W	W	-	W
VHPT Translation vector														
IR VHPT Data fault	N/A	W	N/A	W	N/A	W	N/A	W	N/A	x	N/A	W	N/A	x
VHPT Data fault	N/A	W	N/A	W	N/A	W	N/A	W	N/A	x	N/A	W	N/A	W
VHPT Instruction fault	N/A	W	N/A	W	N/A	W	N/A	W	N/A	x	N/A	W	N/A	x
Virtual External Interrupt vector														
Virtual External Interrupt	-	W	x	x	x	x	x	x	x	x	W	W	-	x
Virtualization vector														
Virtualization fault	-	W	x	x	x	x	x	x	x	x	W	W	-	W

a. "N/A" indicates that this cannot happen.
b. "W" indicates that the resource is written with a new value.
c. "x" indicates that the resource may or may not be written; whether it is written and with what value is implementation specific.
d. "-" indicates that the resource is not written.

Table 8-2. ISR Values on Interruption

Vector / Interruption	ed	ei[a]	so	ni[b]	ir[c]	rs[d]	sp[e]	na[f]	r	w	x
Alternate Data TLB vector											
Alternate Data TLB fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
IR Alternate Data TLB fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
Alternate Instruction TLB vector											
Alternate Instruction TLB fault	0	ri	0	ni	0	0	0	0	0	0	1
Break Instruction vector											
Break Instruction fault	0	ri	0	ni	0	0	0	0	0	0	0
Data Access Rights vector											
Data Access Rights fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Access Rights fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Access-Bit vector											
Data Access Bit fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Access Bit fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Key Miss vector											
Data Key Miss fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
IR Data Key Miss fault	0	ri	0	ni	1	1	0	0	1	0	0
Data Nested TLB vector[g]											
Data Nested TLB fault	-	-	-	-	-	-	-	-	-	-	-
IR Data Nested TLB fault	-	-	-	-	-	-	-	-	-	-	-
Data TLB vector											
Data TLB fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
IR Data TLB fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
Debug vector											
Data Debug fault	ed[k]	ri	0	ni	0	rs	sp	na	r	w	0

Table 8-2. ISR Values on Interruption (Continued)

Vector / Interruption	ed	ei[a]	so	ni[b]	ir[c]	rs[d]	sp[e]	na[f]	r	w	x
Instruction Debug fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Debug fault	0	ri	0	ni	1	1	0	0	1	0	0
Dirty-Bit vector											
Data Dirty Bit fault	ed[k]	ri	so	ni	0	rs	0	na[h]	r	1	0
Disabled FP-Register vector											
Disabled Floating-Point Register fault	0	ri	0	ni	0	0	sp	0	r	w	0
External Interrupt vector											
External Interrupt	0	ri	0	ni	ir[i]	0	0	0	0	0	0
Floating-point Fault vector											
Floating-Point Exception fault	0	ri	0	ni	0	0	0	0	0	0	0
Floating-point Trap vector											
Floating-Point Exception trap	0	ei	0	ni	0	0	0	0	0	0	0
General Exception vector											
Disabled ISA Transition fault	0	ri	0	ni	0	0	0	0	0	0	0
Illegal Dependency fault	0	ri	0	ni	0	0	0	0	0	0	0
Illegal Operation fault	0	ri	0	ni	0	0	0	0	0	0	0
IR Unimplemented Data Address fault	0	ri	0	ni	1	1	0	0	1	0	0
Privileged Operation fault	0	ri	0	ni	0	0	0	na	0	0	0
Privileged Register fault	0	ri	0	ni	0	0	0	0	0	0	0
Reserved Register/Field fault	0	ri	0	ni	0	0	0	0	0	0	0
Unimplemented Data Address fault	0	ri	0	ni	0	rs	0	na[j]	r	w	0
IA-32 Exception vector	0	0	0	0	0	0	0	0	0	0	x
IA-32 Intercept vector	0	0	0	0	0	0	0	0	r	w	0
IA-32 Interrupt vector	0	0	0	0	0	0	0	0	0	0	0
Instruction Access Rights vector											
Instruction Access Rights fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction Access-Bit vector											
Instruction Access Bit fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction Key Miss vector											
Instruction Key Miss fault	0	ri	0	ni	0	0	0	0	0	0	1
Instruction TLB vector											
Instruction TLB fault	0	ri	0	ni	0	0	0	0	0	0	1
Key Permission vector											
Data Key Permission fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
Instruction Key Permission fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Key Permission fault	0	ri	0	ni	1	1	0	0	1	0	0
Lower-Privilege Transfer Trap vector											
Unimplemented Instruction Address fault	0	ri	0	ni	ir	0	0	0	0	0	1
Lower-Privilege Transfer trap	0	ei	0	ni	ir	0	0	0	0	0	0
Unimplemented Instruction Address trap	0	ei	0	ni	ir	0	0	0	0	0	0
NaT Consumption vector											
Data NaT Page Consumption fault	0	ri	so	ni	0	rs	0	na	r	w	0
Instruction NaT Page Consumption fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data NaT Page Consumption fault	0	ri	0	ni	1	1	0	0	1	0	0
Register NaT Consumption fault	0	ri	0	ni	0	0	0	na	r	w	0

Table 8-2. ISR Values on Interruption (Continued)

Vector / Interruption	ed	ei[a]	so	ni[b]	ir[c]	rs[d]	sp[e]	na[f]	r	w	x
Page Not Present vector											
Data Page Not Present fault	ed[k]	ri	so	ni	0	rs	sp	na	r	w	0
Instruction Page Not Present fault	0	ri	0	ni	0	0	0	0	0	0	1
IR Data Page Not Present fault	0	ri	0	ni	1	1	0	0	1	0	0
Single Step Trap vector											
Single Step trap	0	ei	0	ni	ir	0	0	0	0	0	0
Speculation vector											
Speculative Operation fault	0	ri	0	ni	0	0	0	0	0	0	0
Taken Branch Trap vector											
Taken Branch trap	0	ei	0	ni	ir	0	0	0	0	0	0
Unaligned Reference vector											
Unaligned Data Reference fault	ed	ri	0	ni	0	0	sp	0	r	w	0
Unsupported Data Reference vector											
Unsupported Data Reference fault	ed	ri	0	ni	0	0	0	0	r	w	0
VHPT Translation vector											
IR VHPT Data fault	0	ri	0	ni[l]	1	1	0	0	1	0	0
VHPT Data fault	ed[k]	ri	so	ni[l]	0	rs	sp	na	r	w	0
VHPT Instruction fault	0	ri	0	ni	0	0	0	0	0	0	1
Virtual External Interrupt vector											
Virtual External Interrupt	0	ri	0	ni	ir[m]	0	0	0	0	0	0
Virtualization vector											
Virtualization fault	0	ri	0	ni	0	0	0	0	0	0	0

a. ISR.ei is equal to IPSR.ri for all faults and external interrupts (1 for faults and interrupts on the L+X instruction of an MLX). For traps, ISR.ei points at the excepting instruction (2 for traps on the L+X instruction of an MLX).
b. If ISR.ni is 1, the interruption occurred either when PSR.ic was 0 or was in-flight.
c. ISR.ir captures the value of RSE.CFLE at the time of an interruption.
d. ISR.rs is 1 for interruptions caused by mandatory RSE fills/spills and 0 for all others.
e. ISR.sp is 1 for interruptions caused by speculative loads and zero for all others.
f. ISR.na is 1 for interruptions caused by non-access instructions and zero for all others.
g. ISR is not written.
h. A faulting `probe.w.fault` or `probe.rw.fault` can cause a Dirty Bit fault on a non-access instruction.
i. ISR.ir is 1 if an external interrupt was taken when mandatory RSE fills caused by a `br.ret` or `rfi` were re-loading the current register stack frame.
j. A faulting `lfetch.fault` or `probe.fault` to an unimplemented address will set ISR.na to 1.
k. ISR.ed is 0 if the interruption was caused by a mandatory RSE fill or spill.
l. If PSR.ic was 0 when the interruption was taken, these faults do not occur, but a Data Nested TLB fault is taken.
m. ISR.ir is 1 if an external interrupt was taken when mandatory RSE fills caused by a `br.ret` or `rfi` were re-loading the current register stack frame.

Table 8-3 provides the definition for the ISR.code field on all Itanium traps. Hardware will always deliver the highest priority enabled trap. Software must look at the ISR.code bit vector to determine if any lower priority trap occurred at the same time as the trap being processed.

Table 8-3. ISR.code Fields on Intel® Itanium® Traps

Field	Bit	Description
fp	0	Floating-Point Exception trap
lp	1	Lower-Privilege Transfer trap

Table 8-3. ISR.code Fields on Intel® Itanium® Traps (Continued)

Field	Bit	Description
tb	2	Taken Branch trap
ss	3	Single Step trap
ui	4	Unimplemented Instruction Address trap
fp trap code	7	IEEE O (overflow) exception (Parallel FP-LO)
fp trap code	8	IEEE U (underflow) exception (Parallel FP-LO)
fp trap code	9	IEEE I (inexact) exception (Parallel FP-LO)
fp trap code	10	FPA, Added one to significand when rounding (Parallel FP-LO)
fp trap code	11	IEEE O (overflow) exception (Normal or Parallel FP-HI)
fp trap code	12	IEEE U (underflow) exception (Normal or Parallel FP-HI)
fp trap code	13	IEEE I (inexact) exception (Normal or Parallel FP-HI)
fp trap code	14	FPA, Added one to significand when rounding (Normal or Parallel FP-HI).

Table 8-4. Interruption Vectors Sorted Alphabetically

Vector Name	Offset	Page
Alternate Data TLB vector	0x1000	2:178
Alternate Instruction TLB vector	0x0c00	2:177
Break Instruction vector	0x2c00	2:185
Data Access Rights vector	0x5300	2:191
Data Access-Bit vector	0x2800	2:184
Data Key Miss vector	0x1c00	2:181
Data Nested TLB vector	0x1400	2:179
Data TLB vector	0x0800	2:176
Debug vector	0x5900	2:200
Dirty-Bit vector	0x2000	2:182
Disabled FP-Register vector	0x5500	2:195
External Interrupt vector	0x3000	2:186
Floating-Point Fault vector	0x5c00	2:203
Floating-Point Trap vector	0x5d00	2:204
General Exception vector	0x5400	2:192
IA-32 Exception vector	0x6900	2:210
IA-32 Intercept vector	0x6a00	2:211
IA-32 Interrupt vector	0x6b00	2:212
Instruction Access Rights vector	0x5200	2:190
Instruction Access-Bit vector	0x2400	2:183
Instruction Key Miss vector	0x1800	2:180
Instruction TLB vector	0x0400	2:175
Key Permission vector	0x5100	2:189
Lower-Privilege Transfer Trap vector	0x5e00	2:205
NaT Consumption vector	0x5600	2:196
Page Not Present vector	0x5000	2:188
Single Step Trap vector	0x6000	2:208
Speculation vector	0x5700	2:198
Taken Branch Trap vector	0x5f00	2:207
Unaligned Reference vector	0x5a00	2:201

Table 8-4. Interruption Vectors Sorted Alphabetically (Continued)

Vector Name	Offset	Page
Unsupported Data Reference vector	0x5b00	2:202
VHPT Translation vector	0x0000	2:173
Virtual External Interrupt vector	0x3400	2:187
Virtualization vector	0x6100	2:209

Name **VHPT Translation vector (0x0000)**

Cause The hardware VHPT walker encountered a TLB miss while attempting to reference the virtually addressed hashed page table for a memory reference (including IA-32).

Interruptions on this vector:

IR VHPT Data fault
VHPT Instruction fault
VHPT Data fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IHA – The virtual address in the hashed page table which the hardware VHPT walker was attempting to reference.

ITIR – The ITIR contains default translation information for the virtual address contained in the IHA. The access key field within this register is set to the region id value from the region register selected by the virtual address in the IHA**.** The ITIR.ps field is set to the RR.ps field from the selected region register. All other fields are set to 0.

IIB0, IIB1 – If implemented, for VHPT Data faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR VHPT Data and VHPT Instruction faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

If the fault is due to a VHPT data fault for both original instruction and data references:

- IFA – The faulting address that the hardware VHPT walker was attempting to resolve.
- ISR – The ISR bits are set to reflect the original access on whose behalf the VHPT walker was operating. If the original operation was a non-access instruction then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ISR.code, ni, ed, ei, ir, rs, sp, and na bits are always 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

If the fault is due to a VHPT instruction fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits or, if the hardware VHPT walker was attempting to resolve a TLB miss, the virtual address of the translation.
- ISR – The ISR bits are set based on the original instruction fetch that the VHPT walker was attempting to resolve. The defined ISR bits are specified below. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ei and ni bits are always 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes

This fault can only occur when PSR.ic is 1 or in-flight, and the VHPT walker is enabled for the referenced region. Refer to "VHPT Environment" on page 2:67 for details on VHPT enabling.

The original IFA address will be needed by the operating system page fault handler in the case where the page containing the VHPT entry has not yet been allocated. When the translation for the VHPT is available the handler must first move the address contained in the IHA to the IFA prior to the TLB insert.

Name **Instruction TLB vector (0x0400)**

Cause The instruction TLB entry needed by an instruction fetch (including IA-32) is absent, and the hardware VHPT walker could not find the translation in the VHPT, or the hardware VHPT walker is enabled but not implemented on this processor.

Interruptions on this vector:

Instruction TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IHA – The virtual address of the hashed page table entry which corresponds to the reference that raised this fault.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. The ISR.ei and ni bits are always 0 for IA-32 memory references.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when PSR.ic is 1 or in-flight, the VHPT hardware walker is enabled for the referenced region, the PSR.it bit is 1, and the fetched instruction bundle is to be executed. Refer to "VHPT Environment" on page 2:67 for details on VHPT enabling.

The hardware VHPT walker may have failed due to an unimplemented page size, tag mismatch, illegal entry, or it may have terminated before reading the data. Software must be able to handle the case where the VHPT walker fails.

Name **Data TLB vector (0x0800)**

Cause For memory references (including IA-32), the data TLB entry needed by the data access is absent, and the hardware VHPT walker could not find the translation in the VHPT, or the hardware VHPT walker is not implemented on this processor.

Interruptions on this vector:

IR Data TLB fault
Data TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IHA – The virtual address of the hashed page table entry which corresponds to the reference that raised this fault.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The address of the data being referenced.

IIB0, IIB1 – If implemented, for Data TLB faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data TLB faults. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. The ISR.code, ed, ei, ir, rs, sp and na bits are always 0 for IA-32 memory references. The defined ISR bits are specified below.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="12">0</td><td colspan="4">code{3:0}</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>ed</td><td colspan="2">ei</td><td>so</td><td>ni</td><td>ir</td><td>rs</td><td>sp</td><td>na</td><td>r</td><td>w</td><td>0</td></tr>
</table>

Notes The fault can only occur on an IA-32 or Itanium load, store, semaphore, or non-access operation when PSR.dt is 1, and the VHPT hardware walker is enabled for the referenced region. This fault can only occur on a mandatory RSE load/store operation if PSR.rt is 1, and the VHPT hardware walker is enabled for the referenced region. Refer to "VHPT Environment" on page 2:67 for details on VHPT enabling.

The hardware VHPT walker may have failed due to an unimplemented page size, tag mismatch, illegal entry, or it may have terminated before reading the data. Software must be able to handle the case where the VHPT walker fails. The Data TLB fault is only taken if PSR.ic is 1 or in-flight, otherwise a Data Nested TLB fault is taken.

Name **Alternate Instruction TLB vector (0x0c00)**

Cause The instruction TLB entry needed by an instruction fetch (including IA-32) is absent, and the hardware VHPT walker was not enabled for this address.

Interruptions on this vector:

Alternate Instruction TLB fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register**.** The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – For Itanium memory references, the ISR.ei bits are set to indicate which instruction caused the exception and ISR.ni is set to 0 if PSR.ic was 1 when the interruption was taken, and set to 1 if PSR.ic was 0 or in-flight. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault can only occur when the VHPT walker is disabled for the referenced region, and the fetched instruction bundle is to be executed. Refer to "VHPT Environment" on page 2:67 for details on VHPT enabling.

Name

Alternate Data TLB vector (0x1000)

Cause

For memory references (including IA-32), the data TLB entry needed by data access is absent, and the hardware VHPT walker was not enabled for this address.

Interruptions on this vector:

IR Alternate Data TLB fault
Alternate Data TLB fault

Parameters

IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The address of the data being referenced.

IIB0, IIB1 – If implemented, for Alternate Data TLB faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Alternate Data TLB faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. The ISR.ni bit is 0 if PSR.ic was 1 when the interruption was taken, and is 1 if PSR.ic was in-flight. For IA-32 memory references the ISR.code, ed, ei, ir, rs, sp and na bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes

The fault can only occur on an IA-32 or Itanium load, store, semaphore, or non-access operation when PSR.dt is 1, and the VHPT hardware walker is disabled for the referenced region. This fault can only occur on a mandatory RSE load/store operation if PSR.rt is 1, and the VHPT hardware walker is disabled for the referenced region. The Alternate Data TLB fault is only taken if PSR.ic is 1 or in-flight, otherwise a Data Nested TLB fault is taken. Refer to “VHPT Environment” on page 2:67 for details on VHPT enabling.

Name **Data Nested TLB vector (0x1400)**

Cause For memory references, the data TLB entry needed for a data reference is absent and PSR.ic is 0. Note: Data Nested TLB faults cannot occur during IA-32 instruction set execution, since PSR.ic must be 1.

Interruptions on this vector:

IR Data Nested TLB fault
Data Nested TLB fault

Parameters IIP, IPSR, IIPA, IFS, ISR are **unchanged** from their previous values; they contain information relating to the original interruption.

ITIR – is **unchanged** from the previous value.

IFA – is **unchanged** from the previous value and contains the original address of the data being referenced.

IIB0, IIB1 – If implemented, the IIB registers are unchanged from their previous values. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

Notes This fault occurs when PSR.dt 1 and PSR.ic is 0 on a load, store, semaphore, and faulting non-access instructions. It also occurs when PSR.dt is 0 and PSR.ic is 0 for a regular_form probe instruction. Finally it can occur when PSR.rt is 1 and PSR.ic is 0 on a RSE mandatory load/store operation. Since the operating system is in control of the code executing at the time of the nested fault, it can by convention know which register contains the address that raised the nested event. As the PSR.ic bit is 0 on a nested fault, the IFA contains the original data address if the original interruption was caused by a data TLB fault. If the translation table entry required by the nested miss handler has not yet been allocated, then the address in the IFA will be passed to the operating system page fault handler. If the translation for the entry is available then the general register containing the nested fault address must be moved to the IFA prior to the insert. The ISR contains the ISR for the original faulting instruction, and not the ISR for the instruction that caused the nested fault.

Name **Instruction Key Miss vector (0x1800)**

Cause For instruction fetches (including IA-32), the PSR.it bit is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address of the executing instruction bundle does not match any of the valid protection keys.

Interruptions on this vector:

Instruction Key Miss fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the original instruction address. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Name **Data Key Miss vector (0x1c00)**

Cause For memory references (including IA-32), the PSR.dt bit is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address referenced by a load, store, probe (regular_form `probe` or `probe.fault`) or semaphore operation does not match any of the valid protection keys. The RSE may cause this fault if PSR.rt is 1, the PSR.pk bit is 1, and the access key from the TLB entry for the address referenced by an RSE mandatory load or store operation does not match any of the valid protection keys.

Interruptions on this vector:

IR Data Key Miss fault
Data Key Miss fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

IIB0, IIB1 – If implemented, for Data Key Miss faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Key Miss faults. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, the ISR.code, ed, ei, ni, ir, rs, sp, and na bits are 0. The value for the ISR bits depend on the type of access performed and are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes `Probe` (regular_form `probe` or `probe.fault`) and the faulting variant of `lfetch` are the only non-access instructions that will cause a data key miss fault.

Name **Dirty-Bit vector (0x2000)**

Cause IA-32 or Itanium store or semaphore operations to a page with the dirty-bit (TLB.d) equal to 0 in the data TLB.

Interruptions on this vector:

Data Dirty Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE spill references, ISR.ed is always 0. For IA-32 memory references, ISR.ed, ei, ni, and rs are 0. If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	0	rs	0	na	r	1	0

Notes Dirty Bit fault can only occur in these situations:

- When PSR.dt is 1 on an IA-32 or Itanium store or semaphore operation
- When PSR.dt is 1 on a `probe.w.fault` or `probe.rw.fault`
- When PSR.rt is 1 on an RSE mandatory store operation

For `probe.w.fault` or `probe.rw.fault` the ISR.na bit is set, and the ISR.code field is written with a value of 5.

Only an IA-32 or Itanium semaphore, or `probe.rw.fault` operation would set ISR.r on a dirty bit fault.

Software is invoked to update the dirty bit in the data TLB entry and the Page table. The PSR.da bit can be used to suppress this fault for one executed instruction or one mandatory RSE store operation.

Name **Instruction Access-Bit vector (0x2400)**

Cause For instruction fetches (including IA-32), the access bit (TLB.a) in the TLB entry for this page is 0, and an instruction on the page is referenced.

Interruptions on this vector:

Instruction Access Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 memory references the ISR.ei and ni bits are 0. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes The fault can only occur when PSR.it is 1 on an instruction reference (including IA-32). Software uses this fault for memory management page replacement algorithms. The PSR.ia bit can be used to suppress this fault for one executed instruction.

Name **Data Access-Bit vector (0x2800)**

Cause For data memory references (including IA-32), the access bit (TLB.a) in the TLB entry for this page is 0, and the page is referenced.

Interruptions on this vector:

IR Data Access Bit fault
Data Access Bit fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

IIB0, IIB1 – If implemented, for Data Access Bit faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Access Bit faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, na and sp are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes These faults can only occur in these situations:

- When PSR.dt is 1 on an IA-32 or Itanium load, store, or semaphore operation
- When PSR.dt is 1 on a `probe.fault`
- When PSR.dt is 1 on an `lfetch.fault`
- When PSR.rt is 1 on an RSE mandatory load/store operation

For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Software uses this fault for memory management page replacement algorithms. The PSR.da bit can be used to suppress this fault for one executed instruction or one mandatory RSE memory reference.

Name **Break Instruction vector (0x2c00)**

Cause An attempt is made to execute an Itanium `break` instruction.

Interruptions on this vector:

Break Instruction fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIM – Is updated with the break instruction immediate value.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Notes This fault cannot be raised by IA-32 instructions.

Name **External Interrupt vector (0x3000)**

Cause There are unmasked external interrupts pending from external devices, other processors, or internal processor events and:

- PSR.i is 1, while executing Itanium instructions
- PSR.i is 1 and (CFLAG.if is 0 or EFLAG.if is 1), while executing IA-32 instructions

IPSR.is indicates which instruction set was executing at the time of the interruption.

Interruptions on this vector:

External Interrupt

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IVR – Highest priority unmasked pending external interrupt vector number. If there are no unmasked pending interrupts the “spurious” interrupt vector (15) is reported.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction was to be executed when the external interrupt event was taken. The defined ISR bits are specified below. For external interrupts taken in the IA-32 instruction set, ISR.ei, ni and ir bits are 0.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	0	0

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44	43	42 41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	ir	0	0	0	0	0	0

Notes: Software is expected to avoid situations which could cause ISR.ni to be 1.

Name **Virtual External Interrupt vector (0x3400)**

Cause The guest highest pending interrupt (GHPI) specified by the VMM is unmasked on the virtual processor.

IPSR.is indicates which instruction set was executing at the time of the interruption.

Interruptions on this vector:

Virtual External Interrupt

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction was to be executed when the external interrupt event was taken. The defined ISR bits are specified below. For external interrupts taken in the IA-32 instruction set, ISR.ei, ni and ir bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Notes: Software is expected to avoid situations which could cause ISR.ni to be 1.

Name **Page Not Present vector (0x5000)**

Cause The bundle or IA-32 instruction being executed resides on a page for which the P-bit (TLB.p) in the instruction TLB entry is 0, or the data being referenced resides on a page for which the P-bit in the data TLB entry is 0.

Interruptions on this vector:

IR Data Page Not Present fault
Instruction Page Not Present fault
Data Page Not Present fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IIB0, IIB1 – If implemented, for Data Page Not Present faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Page Not Present and Instruction Page Not Present faults. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

If the fault is due to a data page not present fault for both instruction and data original references:

- IFA – The virtual address of the data being referenced.
- ISR – If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0. The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, sp and na bits are 0.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="12">0</td><td colspan="4">code{3:0}</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>ed</td><td colspan="2">ei</td><td>so</td><td>ni</td><td>ir</td><td>rs</td><td>sp</td><td>na</td><td>r</td><td>w</td><td>0</td></tr>
</table>

If the fault is due to an instruction page not present fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references the ISR.ei and ni bits are 0.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="16">0</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
</table>

Notes This fault can only occur when PSR.it is 1 on an instruction reference, when PSR.dt is 1 on a load, store, semaphore, or non-access operation, or when PSR.rt is 1 on a RSE mandatory load/store operation.

Name **Key Permission vector (0x5100)**

Cause Data access (including IA-32): The PSR.dt bit is 1, the PSR.pk bit is 1 and read or write permission is disabled by the matching protection register on a load, store, or semaphore operation. The RSE may cause this fault if PSR.rt is 1, the PSR.pk bit is 1 and read or write permission is disabled by the matching protection register on an RSE mandatory load/store operation. Instruction access (including IA-32): The PSR.it bit is 1, the PSR.pk bit is 1 and execute permission is disabled by the matching protection register.

Interruptions on this vector:

IR Data Key Permission fault
Instruction Key Permission fault
Data Key Permission fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register.The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IIB0, IIB1 – If implemented, for Data Key Permission faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Key Permission and Instruction Key Permission faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

If the fault is due to a data key permission fault:

- IFA – Faulting data address.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, the ISR.code, ed, ei, ni, ir, rs, sp bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

If the fault is due to an instruction key permission fault:

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references, ISR.ei and ni are set to 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Name **Instruction Access Rights vector (0x5200)**

Cause For instruction fetches (including IA-32), the PSR.it bit is 1, and the access rights for this page do not allow execution or do not allow execution at the current privilege level.

Interruptions on this vector:

Instruction Access Rights fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below. For IA-32 memory references, ISR.ei and ni bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes This fault does not occur if PSR.it is 0.

Name **Data Access Rights vector (0x5300)**

Cause For memory references (including IA-32), the PSR.dt bit is 1, and the access rights for this page do not allow read access or do not allow read access at the current privilege level for load and semaphore operations. The PSR.dt bit is 1, and the access rights for this page do not allow write access or do not allow write access at the current privilege level for store and semaphore operations.

The PSR.rt bit is 1, and the access rights for this page do not allow read access or do not allow read access at the current privilege level for the RSE mandatory load operation. The PSR.rt bit is 1, and the access rights for this page do not allow write access or do not allow write access at the current privilege level for the RSE mandatory store operation.

Interruptions on this vector:

IR Data Access Rights fault
Data Access Rights fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

ITIR – The ITIR contains default translation information for the address contained in the IFA. The access key field within this register is set to the region id value from the referenced region register. The ITIR.ps field is set to the RR.ps field from the referenced region register. All other fields are set to 0.

IFA – Faulting data address.

IIB0, IIB1 – If implemented, for Data Access Rights faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Access Rights faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For IA-32 memory references, ISR.code, ed, ei, ni, ir, rs, and sp bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		so	ni	ir	rs	sp	na	r	w	0

Notes For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

Name **General Exception vector (0x5400)**

Cause An attempt is being made to execute an illegal operation, privileged instruction, access a privileged register, unimplemented field, unimplemented register, unimplemented address, or take an inter-instruction set branch when disabled.

Interruptions on this vector:

IR Unimplemented Data Address fault
Illegal Operation fault
Illegal Dependency fault
Privileged Operation fault
Disabled Instruction Set Transition fault
Reserved Register/Field fault
Unimplemented Data Address fault
Privileged Register fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP for the following faults:

Illegal Operation fault
Illegal Dependency fault
Privileged Operation fault
Disabled Instruction Set Transition fault
Reserved Register/Field fault
Unimplemented Data Address fault
Privileged Register fault

The IIB registers are undefined for IR Unimplemented Data Address faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. For IA-32 instruction set faults, ISR.ei, ni, na, sp, rs, ir, ed bits are always 0.

- If the fault was caused by a non-access instruction, ISR.code{3:0} specifies which non-access instruction. See “Non-access Instructions and Interruptions” on page 2:103.
- ISR.code{7:4} = 0: Illegal Operation fault. Cannot be raised by IA-32 instructions.
    - An attempt is being made to execute an illegal operation. Illegal operations include:
        - Attempts to execute instructions containing reserved major opcodes, reserved sub-opcodes, or reserved instruction fields, writing GR 0, FR 0 or FR 1, writing a read-only register, or accessing a reserved register.
        - Attempts to execute a reserved template encoding. An `rfi` to a reserved template encoding preserves IPSR.ri and will set ISR.ei to IPSR.ri.
        - Attempts to execute a bundle of template MLX when PSR.ri == 2. This can only be caused by doing an `rfi` with an improper setting of IPSR.ri. In this case, IPSR.ri and ISR.ei will both be 2.
        - Attempts to write outside the current register stack frame.
        - Attempts to specify the same GR, when the instruction has two GR targets (e.g., post-increment).

    - If the instruction has two PR targets, and specifies the same PR for both, predicated-off unconditional compare, `fclass`, `tbit`, `tnat`, and `tf` instructions take this fault, even when their qualifying predicate is zero.
    - Register bank conflict on a floating-point load pair instruction.
    - An access to BSPSTORE or RNAT is performed with a non-zero RSC.mode, or a `loadrs` is performed with a non-zero RSC.mode.
    - A `loadrs` is performed with a non-zero CFM.sof and a non-zero RSC.loadrs, or a `loadrs` causes more registers to be loaded from memory than can fit in the physical stacked register file.
    - Attempts to predicate a `br.ia` instruction or to execute `br.ia` when AR[BSPSTORE] != AR[BSP].
    - Attempts to execute `epc` if PFS.ppl is less than PSR.cpl.
    - Attempts to access interruption registers if PSR.ic is 1.
    - Attempts to execute an `itc` or `itr` instruction if PSR.ic is 1.
    - Attempts to allocate a stack frame larger than 96 registers, or with the rotating region larger than the stack frame, or with the size of locals larger than the stack frame, or specifying a qualifying predicate other than PR 0 on an `alloc` instruction.
    - Attempts to execute instructions that are not supported by the processor.
    - Attempts to execute a `ldfp` instruction with two odd-numbered physical FR targets or two even-numbered physical FR targets.
    - Attempts to access an application register from the wrong unit type.
    - Attempts to execute a `br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, or `br.wexit` other than in slot 2 of a bundle.
    - Attempts to execute an `alloc`, `flushrs` or `loadrs` as other than the first instruction in an instruction group. (The result of such an attempt is undefined, and could result in an Illegal Operation fault, depending on the processor implementation. See Section 3.5, "Undefined Behavior" on page 1:44 for details).
    - Attempts to execute a `clrrrb`, `clrrrb.pr`, `cover`, `itc.d`, `itc.i`, `ptc.g` or `ptc.ga` instruction as other than the last instruction in an instruction group. (The result of such an attempt is undefined, and may possibly result in an Illegal Operation fault, depending on the processor See Section 3.5, "Undefined Behavior" on page 1:44 for details).
- ISR.code{7:4} = 1: Privileged Operation fault. Cannot be raised by IA-32 instructions.
- ISR.code{7:4} = 2: Privileged Register fault. Cannot be raised by IA-32 instructions.
- ISR.code{7:4} = 3: Reserved Register/Field fault, Unimplemented Data Address fault or IR Unimplemented Data Address fault. Cannot be raised by IA-32 instructions. For Unimplemented Data Address fault:
    - If ISR.rs = 0: A data memory reference to an unimplemented address has occurred.
    - If ISR.rs = 1: A mandatory RSE reference to an unimplemented address has occurred.

  For details, refer to "Reserved and Ignored Registers and Fields" on page 1:23 and "Unimplemented Address Bits" on page 2:73.

- ISR.code{7:4} = 4: Disabled Instruction Set Transition fault. An instruction set transition was attempted while PSR.di was 1. This fault can be raised by either the Itanium `br.ia` instruction or the IA-32 `jmpe` instruction. IPSR.is indicates the faulting instruction set.
- ISR.code{7:4} = 8: Illegal Dependency fault. Cannot be raised by IA-32 instructions. The processor has detected a resource dependency violation.

If the fault is due to a Disabled ISA Transition fault, Illegal Dependency fault, Illegal Operation fault, Privileged Register fault or Reserved Register/Field fault:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="8">0</td><td colspan="4">code{7:4}</td><td colspan="4">0</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Otherwise:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="8">0</td><td colspan="4">code{7:4}</td><td colspan="4">code{3:0}</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>ir</td><td>rs</td><td>0</td><td>na</td><td>r</td><td>w</td><td>0</td></tr>
</table>

Name **Disabled FP-Register vector (0x5500)**

Cause An attempt is made to reference a floating-point register set that is disabled.

When PSR.dfl is 1, execution of any IA-32 FP, SSE or MMX technology instructions raises a Disabled FP Register Low Fault (regardless of whether FR2 - FR31 are actually referenced).

When PSR.dfh is 1, execution of the first IA-32 instruction following a `br.ia` or `rfi` raises a Disabled FP Register High fault.

If concurrent IA-32 Disabled FP Register High and Low faults are generated, the Disabled FP Register High fault takes precedence and is reported in the ISR code, the Disabled FP Register Low fault is discarded and not reported in the ISR code.

Interruptions on this vector:

Disabled Floating-Point Register fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The defined ISR bits are specified below.

- ISR.code{0} = 1: FR2 - FR31 disabled and access attempted.
- ISR.code{1} = 1: FR32 - FR127 disabled and access attempted.

For IA-32 references, ISR.ei, ni, sp, r, and w bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0														code	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	sp	0	r	w	0

Name **NaT Consumption vector (0x5600)**

Cause A non-speculative operation (including IA-32) (e.g., load, store, control register access, instruction fetch etc.) read a NaT source register, NaTVal source register, or referenced a NaTPage.

Interruptions on this vector:

IR Data NaT Page Consumption fault
Instruction NaT Page Consumption fault
Register NaT Consumption fault
Data NaT Page Consumption fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, for Register NaT Consumption and Data NaT Page Consumption faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data NaT Page Consumption and Instruction NaT Page Consumption faults. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

If the fault is due to a Data NaT Page Consumption fault or an IR Data NaT Page Consumption fault:

A non-speculative Itanium integer/FP instruction or instruction fetch or IA-32 data memory reference accessed a page with the NaTPage memory attribute.

- IFA – faulting data address.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0. For the IA-32 instruction set, ISR.ed, ei, ni, ir, rs and na bits are 0. For `probe.fault` or `lfetch.fault` the ISR.na bit is set.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								2				code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		so	ni	ir	rs	0	na	r	w	0

If the fault is due to an Instruction NaT Page Consumption fault:

A non-speculative Itanium integer/FP instruction or instruction fetch accessed a page with the NaTPage memory attribute.

- IFA – The virtual address of the bundle or the 16 byte aligned IA-32 instruction address zero extended to 64-bits.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For the IA-32 instruction set, ISR.ni and ei bits are 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0								2				0			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

If the fault is due to an Register NaT Consumption fault:

A non-speculative Itanium instruction reads a NaT’ed GR or an FR containing NaTVal. An IA-32 integer instruction reads a NaT’ed GR. For IA-32 instructions behavior of NaT and NaTVal values is model specific, see Section 6.2.4.3, “NaT/NaTVal Response for IA-32 Instructions” on page 1:134 for details.

- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For the IA-32 instruction set, ISR.ed, ei, ni, ir, rs, r, w, and na bits are 0.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">0</td><td colspan="8">0</td><td colspan="4">1</td><td colspan="4">code{3:0}</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">ei</td><td>0</td><td>ni</td><td>0</td><td>0</td><td>0</td><td>na</td><td>r</td><td>w</td><td>0</td></tr>
</table>

Name **Speculation vector (0x5700)**

Cause A `chk.a`, `chk.s`, or `fchkf` instruction needs to branch to recovery code, and the branching behavior is unimplemented by the processor. This fault cannot be raised by IA-32 instructions.

Interruptions on this vector:

Speculative Operation fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIM – contains the immediate value from the `chk.s`, `chk.a`, or `fchkf` instruction.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The type of instruction which caused the fault is encoded in the lower four bits of the ISR.code field.

- If ISR.code{3:0} = 0: `chk.a` general register speculation fault.
- If ISR.code{3:0} = 1: `chk.s` general register speculation fault.
- If ISR.code{3:0} = 2: `chk.a` floating-point speculation fault.
- If ISR.code{3:0} = 3: `chk.s` floating-point speculation fault.
- If ISR.code{3:0} = 4: `fchkf` fault.

The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Notes The Speculative Operation fault handler is required to perform the following steps:

1. Read the predicates and the IIM, IIP, IPSR, and ISR control registers, into scratch bank 0 general registers.
2. Copy the IIP value to IIPA.
3. Sign-extend the IIM value (from 21 bits to 64), shift it left by 4 bits, add it to the IIP value, and write this value back into IIP.
4. Set the IPSR.ri field to 0.
5. Check whether either IPSR.tb (Taken Branch trap) or IPSR.ss (Single Step enable) is 1. If not, emulation is complete, so restore the predicates and `rfi`. If so, then the check instruction would have taken one of these traps instead of branching to its target, so this handler needs to branch directly to the appropriate trap handler instead of performing the `rfi` (see steps 6 and 7).
6. If IPSR.tb was 1, then update ISR.code with its tb bit set to 1 and its ss bit also set to 1 if IPSR.ss was 1, and all other bits 0. Restore the predicates, execute a `srlz.d`, and branch to the taken branch vector (IVT offset 0x5f00).
7. If IPSR.ss was 1 (but not IPSR.tb), then update ISR.code with its ss bit set to 1, and all other bits 0. Restore the predicates, execute a `srlz.d`, and branch to the single step vector (IVT offset 0x6000).

The Speculative Operation fault handler does not need to check for unimplemented instruction addresses. They will be checked automatically by processor hardware when the handler executes its `rfi`. On processors which report unimplemented instruction addresses with an Unimplemented Instruction Address (UIA) trap, if an emulated check instruction targets an unimplemented address and also needs to take a Single Step trap or Taken Branch trap (or both), the UIA trap will not be raised until after the Single Step and/or Taken Branch trap has been handled, making it appear that the Unimplemented Instruction Address trap has the wrong priority. A Speculative Operation fault handler with this behavior is architecturally compliant. On processors which report unimplemented instruction addresses with an Unimplemented Instruction Address fault, the UIA fault will be taken at the target of the check rather than on the check instruction itself, so any Single Step trap and/or Taken Branch trap on the check will naturally become visible first.

Name **Debug vector (0x5900)**

Cause A debug fault has occurred. Either the instruction address matches the parameters set up in the instruction debug registers, or the data address of a load, store, semaphore, or mandatory RSE fill or spill matches the parameters set up in the data debug registers. All IA-32 instruction set debug events are delivered on the IA_32_Exception(Debug) vector; see Chapter 9, "IA-32 Interruption Vector Descriptions." IA-32 instructions can not raise this fault, IA-32 debug events are delivered on the IA_32_Exception(Debug) vector.

Interruptions on this vector:

IR Data Debug fault
Instruction Debug fault
Data Debug fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, for Data Debug faults, the IIB registers contain the instruction bundle pointed to by IIP. The IIB registers are undefined for IR Data Debug and Instruction Debug faults. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

If the fault is due to a data debug fault or an IR Data Debug fault:

- IFA – The address of the data being referenced.
- ISR – The value for the ISR bits depend on the type of access performed and are specified below. For mandatory RSE fill or spill references, ISR.ed is always 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												code{3:0}			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	ir	rs	sp	na	r	w	0

If the fault is due to an instruction debug fault:

- IFA – Faulting instruction fetch address.
- ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	1

Notes On an instruction reference this fault is suppressed if the PSR.db bit is 0 or if the PSR.id bit is 1. On a data reference this fault is suppressed if the PSR.db bit is 0 or if the PSR.dd bit is 1. The only non-access data operations which can cause a debug fault are the `probe.fault` and `lfetch.fault` instructions.

If unaligned accesses are being performed with debug faults enabled, this fault may be taken even though there is not a match for the address programmed in the breakpoint register. See Section 7.1.2, "Debug Address Breakpoint Match Conditions" on page 2:154.

Name **Unaligned Reference vector (0x5a00)**

Cause If PSR.ac is 1, and the data address being referenced by an Itanium instruction is not aligned to the natural size of the load, store, or semaphore operation, or a data reference is made to a misaligned datum not supported by the implementation. See “Memory Access Instructions” on page 1:57. For IA-32 data memory references, an IA_32_Exception(Alignment Check) fault is raised; see Chapter 9, “IA-32 Interruption Vector Descriptions.” IA-32 instructions can not raise this fault, IA-32 unaligned events are delivered on the IA_32_Exception(Alignment_Check) vector.

If the data reference specified is both unaligned to the natural datum size and unsupported, then an Unaligned Data Reference fault is taken.

Interruptions on this vector:

Unaligned Data Reference fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IFA – The address of the data being referenced.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The value for the ISR bits depend on the type of access performed and are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	0	0	sp	0	r	w	0

Name

Unsupported Data Reference vector (0x5b00)

Cause

An attempt was made to:

- Execute a `fetchadd`, `cmpxchg`, `xchg`, or unsupported `ld16`, `st16` or 10-byte memory reference (`ldfe` or `stfe`) instruction to a page that is neither cacheable with write-back write policy nor a NaTPage.
- Execute a `fetchadd` instruction to a page that is an uncacheable exported (UCE) page and the processor model does not support exporting of `fetchadd` instructions.

See "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86 for details. IA-32 instructions can not raise this fault, IA-32 locked faults are delivered on the IA_32_Intercept(Lock) vector.

If the data reference specified is both unaligned to the natural datum size and unsupported, then an Unaligned Data Reference fault is taken.

IA-32 data memory references that require an external atomic lock when DCR.lc is 1, raise an IA_32_Intercept(Lock) fault; see Chapter 9, "IA-32 Interruption Vector Descriptions."

Interruptions on this vector:

Unsupported Data Reference fault

Parameters

IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IFA – The address of the data being referenced.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The value for the ISR bits depend on the type of access performed and are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				ed	ei		0	ni	0	0	0	0	r	w	0

For `ldfe` and `stfe` instructions, the processor may optionally set both ISR.r and ISR.w to 1, although this is not recommended.

Name **Floating-point Fault vector (0x5c00)**

Cause A floating-point exception fault has occurred. IA-32 numeric instructions can not raise this fault, IA-32 floating point faults are delivered on the IA_32_Exception(Floating-Point) vector.

Interruptions on this vector:

Floating-Point Exception fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

ISR.code contains information about the FP exception fault. The ISR.code field has eight bits defined. See Chapter 5 for details.

- ISR.code{0} = 1: IEEE V (invalid) exception (Normal or Parallel FP-HI)
- ISR.code{1} = 1: Denormal/Unnormal operand exception (Normal or Parallel FP-HI)
- ISR.code{2} = 1: IEEE Z (divide by zero) exception (Normal or Parallel FP-HI)
- ISR.code{3} = 1: Software assist (Normal or Parallel FP-HI)
- ISR.code{4} = 1: IEEE V (invalid) exception (Parallel FP-LO)
- ISR.code{5} = 1: Denormal/Unnormal operand exception (Parallel FP-LO)
- ISR.code{6} = 1: IEEE Z (divide by zero) exception (Parallel FP-LO)
- ISR.code{7} = 1: Software assist (Parallel FP-LO)

The defined ISR bits are specified below:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0	0	0	code{7:0}

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44	43	42 41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	0	0	0	0	0	0	0

Name **Floating-point Trap vector (0x5d00)**

Cause A floating-point exception trap has occurred. IA-32 numeric instructions can not raise this trap.

Interruptions on this vector:

Floating-Point Exception trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIPA. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

ISR.code contains information about the type of FP exception and IEEE information. The ISR code field contains a bit vector (see Table 8-3 on page 2:170) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0	fp trap code								0	0	0	ss	0	0	1

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Name **Lower-Privilege Transfer Trap vector (0x5e00)**

Cause Two trapping conditions transfer control to this vector:

- An attempt is made to transfer control to an unimplemented address, resulting in either an Unimplemented Instruction Address trap or an Unimplemented Instruction Address fault. See “Unimplemented Address Bits” on page 2:73.
- The PSR.lp bit is 1, and a branch lowers the privilege level.

IA-32 instructions can not raise this trap.

Interruptions on this vector:

Unimplemented Instruction Address fault
Unimplemented Instruction Address trap
Lower-Privilege Transfer trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

Note: Please see “Interruption Instruction Bundle Pointer (IIP – CR19)” on page 2:37 for a further clarification of the IIP value for an unimplemented instruction address trap.

IIB0, IIB1 – If implemented, for Lower-Privilege Transfer traps, the IIB registers contain the instruction bundle pointed to by IIPA. The IIB registers are undefined for Unimplemented Instruction Address faults and traps. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – For Unimplemented Instruction Address trap and Lower-Privilege Transfer trap, the ISR.ei bits are set to indicate which instruction caused the exception, and the ISR.code contains a bit vector (see Table 8-3 on page 2:170) for all traps which occurred in the just-executed instruction.

For Unimplemented Instruction Address fault ISR.fp_trap_code is set to 0.

The defined ISR bits are specified below.

If this vector was entered for an Unimplemented Instruction Address fault:

IFA – Faulting unimplemented instruction address

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0	0								0	0	1	0	0	0	0

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ri		0	ni	ir	0	0	0	0	0	1

If this vector was entered for an Unimplemented Instruction Address trap:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0	fp trap code								0	0	1	ss	tb	lp	fp

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

If this vector was entered for a Lower-Privilege Transfer trap:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0									0	0	0	ss	tb	1	0

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Notes

The Unimplemented Instruction Address trap can be the result of a taken branch, a taken `chk`, an `rfi`, or the execution of a slot 2 instruction in a bundle at the last implemented address. The lower privilege transfer trap is only taken on a branch demotion, and not an `rfi` return.

Processors may optionally report unimplemented instruction addresses with an Unimplemented Instruction Address fault on the fetch of the unimplemented address. To system software, this appears the same as if an Unimplemented Instruction Address trap had been taken, except that:

- any concurrent traps (Single Step, Taken Branch, Lower-Privilege Transfer, FP) will be taken first
- asynchronous interrupts (such as External interrupt) may be taken with IIP pointing to the unimplemented address before the Unimplemented Instruction Address fault is taken
- incomplete register stack frame interrupts may be taken with IIP pointing to the unimplemented address before the Unimplemented Instruction Address fault is taken
- ISR.ei will be equal to the value of PSR.ri at the time of the fault (and therefore will not indicate which instruction in the bundle pointed to by IIPA was responsible for the transition to an unimplemented address).

Name **Taken Branch Trap vector (0x5f00)**

Cause A taken branch was executed, and the PSR.tb bit is 1. IA-32 instructions can not raise this trap, IA-32 taken branch traps are delivered on the IA_32_Exception(Debug) vector.

The Taken Branch trap is not taken on an `rfi` instruction.

Interruptions on this vector:

Taken Branch trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

Note: Please see "Interruption Instruction Bundle Pointer (IIP – CR19)" on page 2:37 for a further clarification of the IIP value for an unimplemented instruction address trap or fault.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIPA. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The ISR.code contains a bit vector (see Table 8-3 on page 2:170) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7	6	5	4	3	2	1	0
0	0	0	0	0	0	ss	1	0	0

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44	43	42 41	40	39	38	37	36	35	34	33	32
0	0	ei	0	ni	ir	0	0	0	0	0	0

Name **Single Step Trap vector (0x6000)**

Cause An instruction was successfully executed, and the PSR.ss bit is 1. For IA-32 instruction set, this condition is delivered on the IA_32_Exception(Debug) vector; see Chapter 9, "IA-32 Interruption Vector Descriptions." IA-32 instructions can not raise this trap, IA-32 single step events are delivered on the IA_32_Exception(Debug) vector.

The Single Step trap is not taken on an `rfi` instruction.

Interruptions on this vector:

Single Step trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIPA. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception. The ISR.code contains a bit vector (see Table 8-3 on page 2:170) for all traps which occurred in the just-executed instruction. The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0									0	0	0	1	0	0	0

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	ir	0	0	0	0	0	0

Name **Virtualization vector (0x6100)**

Cause An attempt is made to execute an instruction which requires virtualization. This fault cannot be raised by IA-32 instructions.

Interruptions on this vector:

Virtualization fault

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers contain the instruction bundle pointed to by IIP. Please refer to Section 3.3.5.10, “Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)” on page 2:42 for details on the IIB registers.

ISR – The ISR.ei bits are set to indicate which instruction caused the exception.

The defined ISR bits are specified below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								0								0												0			

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	ei		0	ni	0	0	0	0	0	0	0

Name **IA-32 Exception vector (0x6900)**

Cause A fault or trap was raised while executing from the IA-32 instruction set.

Interruptions on this vector:

IA-32 Instruction Debug fault
IA-32 Code Fetch fault
IA-32 Instruction Length > 15 bytes fault
IA-32 Device Not Available fault
IA-32 FP Error fault
IA-32 Segment Not Present fault
IA-32 Stack Exception fault
IA-32 General Protection fault
IA-32 Divide by Zero fault
IA-32 Alignment Check fault
IA-32 Bound fault
IA-32 INTO trap
IA-32 Breakpoint (INT 3) trap
IA-32 Data Breakpoint trap
IA-32 Taken Branch trap
IA-32 Single Step trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IFA – is undefined. The faulting IA-32 address is contained in IIPA.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – ISR.vector contains the IA-32 exception vector number. ISR.code contains the IA-32 error code for faults or a trap code listing concurrent trap events for traps**.**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								vector								error_code/trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	0		0	0	0	0	0	0	0	0	x

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on each IA-32 Exception and for error code and trap code definition.

Name **IA-32 Intercept vector (0x6a00)**

Cause An intercept fault or trap was raised while executing from the IA-32 instruction set. This vector handles all the IA-32 intercepts described in Chapter 9, "IA-32 Interruption Vector Descriptions."

Interruptions on this vector:

IA-32 Invalid Opcode fault
IA-32 Instruction Intercept fault
IA-32 Locked Data Reference fault
IA-32 System Flag Intercept trap
IA-32 Gate Intercept trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIM – 64-bit information describing the cause of the intercept.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – ISR.vector contains a number specifying the type of intercept. ISR.code contains the IA-32 specific intercept information or a trap code listing concurrent trap events for traps**.**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								intercept_number								intercept_code/trap_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0																				0	0		0	0	0	0	0	0	r	w	0

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on each IA-32 Intercept and for the intercept code and trap code definition.

Name **IA-32 Interrupt vector (0x6b00)**

Cause An IA-32 software interrupt trap was executed. This vector handles all the IA-32 software interrupts described in Chapter 9, "IA-32 Interruption Vector Descriptions."

Interruptions on this vector:

IA-32 Software Interrupt (INT) trap

Parameters IIP, IPSR, IIPA, IFS – are defined; refer to page 2:165 for a detailed description.

IIB0, IIB1 – If implemented, the IIB registers are undefined. Please refer to Section 3.3.5.10, "Interruption Instruction Bundle Registers (IIB0-1 – CR26, 27)" on page 2:42 for details on the IIB registers.

ISR – ISR.vector contains the IA-32 defined interruption vector number. ISR.code contains a trap code listing concurrent trap events.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">0</td><td colspan="8">vector</td><td colspan="16">trap_code</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">0</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Notes See Chapter 9, "IA-32 Interruption Vector Descriptions" for complete details on this vector and the trap code definition.

§

IA-32 Interruption Vector Descriptions 9

This section gives detailed description of all possible IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment. Interruption resources not noted below are undefined after the interruption. For all cases where an interruption is taken out of the IA-32 instruction set, IPSR.is is set to 1.

9.1 IA-32 Trap Code

The following trap code is defined for concurrent traps reported during IA-32 instruction set execution. There is a bit for every possible concurrent trap condition.

Figure 9-1. IA-32 Trap Code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								b3	b2	b1	b0	ss	tb	0	

Figure 9-2. IA-32 Trap Code

Bit	Name	Description
2	tb	taken branch trap, set if an IA-32 branch is taken and branch traps are enabled (PSR.tb is 1).
3	ss	single step trap, set after the successful execution of every IA-32 instruction if PSR.ss or EFLAG.tf is 1.
4-7	b0 to b3	Data breakpoint trap due to a match with the corresponding Intel Itanium data breakpoint registers. Each bit indicates a match with the corresponding DBR registers; b0=DBR0/1, b1=DBR2/3, b2=DBR4/5, b3=DBR6/7. Zero, one or more bits may be set. These bits accumulate data breakpoint register matches that occurred during the duration of executing one IA-32 instruction. In order to be reported, the DBR register address and mask registers must precisely match the IA-32 data memory reference address, and the DBR read, write bits match the type of memory transaction, and the DBR privilege level mask match the value in PSR.cpl.

9.2 IA-32 Interruption Vector Definitions

Following are the definitions of IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium system environment.

Name **IA_32_Exception (Divide) – Divide Fault**

Cause IA-32 IDIV or DIV instruction attempted a divide by zero operation. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								0								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (Debug) – Code Breakpoint Fault**

Cause The Itanium architecture debug facilities triggered an IA-32 code breakpoint fault on a IA-32 instruction fetch and PSR.id and EFLAG.rf are 0. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 1.

ISR.x – 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								1								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	1

Name **IA_32_Exception (Debug) – Data Breakpoint, Single Step, Taken Branch Trap**

Cause The Itanium architecture debug facilities triggered an IA-32 data breakpoint, single-step or branch trap. In the Itanium System Environment, IA-32 Mov SS or Pop SS single step and data breakpoint traps are NOT deferred to the next instruction. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this trap.

Parameters IIPA – virtual address of the trapping IA-32 instruction (zero extended to 64-bits) if there was a taken branch trap. On `jmpe` taken branch traps IIPA contains the address of the `jmpe` instruction. For all other trap events, IIPA is undefined.

IIP – next Itanium instruction address or the virtual IA-32 instruction address zero extended to 64-bits.

ISR.vector – 1.

ISR.code – Trap Code, indicates Concurrent Single Step, Taken Branch, Data Breakpoint Trap events.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">rv</td><td colspan="8">1</td><td colspan="16">trap_code</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">rv</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Name **IA_32_Exception (Break) – INT 3 Trap**

Cause IA-32 breakpoint instruction (INT 3) triggered a trap. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual*** for a complete definition of this trap.

Parameters IIPA – trapping virtual IA-32 instruction address zero extended to 64-bits.

IIP – next virtual IA-32 instruction address zero extended to 64-bits.

ISR.vector – 3.

ISR.code –Trap Code, indicates Concurrent Single Step condition.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">rv</td><td colspan="8">3</td><td colspan="16">trap_code</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">rv</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Name **IA_32_Exception (Overflow) – Overflow Trap**

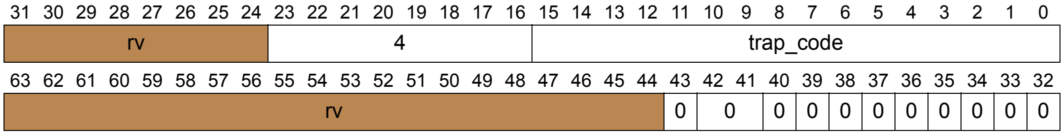
Cause IA-32 INTO instruction execution when EFLAG.of is set to one. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this trap.

Parameters IIPA – trapping virtual IA-32 instruction address zero extended to 64-bits.

IIP – next virtual IA-32 instruction address zero extended to 64-bits.

ISR.vector – 4.

ISR.code – Trap Code, indicates Concurrent Single Step.



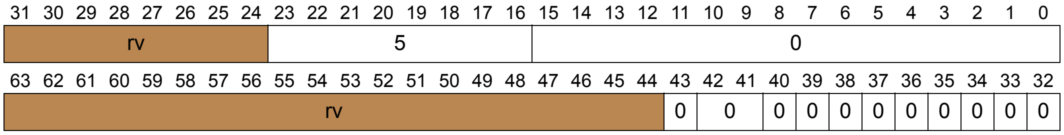
Name **IA_32_Exception (Bound) – Bounds Fault**

Cause Failed IA-32 Bound check instruction. Refer to the ***Intel*® *64 and IA-32 Architectures Software Developer’s Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 5.



Name **IA_32_Exception (InvalidOpcode) – Invalid Opcode Fault**

Cause All IA-32 invalid opcode faults are delivered to the IA_32_Intercept(Instruction) handler, including IA-32 illegal, unimplemented opcodes, MMX technology and SSE instructions if CR0.EM is 1, and SSE instructions if CR4.fxsr is 0. All illegal IA-32 floating-point opcodes result in an IA_32_Intercept(Instruction) regardless of the state of CR0.em.

Name **IA_32_Exception (DNA) – Device Not Available Fault**

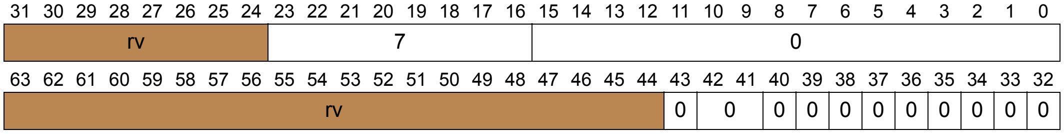
Cause The processor executed an IA-32 ESC or floating-point instruction with CR0.em is 1. Or an IA-32 WAIT, ESC, floating-point instruction, MMX technology or SSE instruction is executed and CR0.ts bit is 1.

Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 7.



Name **Double Fault**

Cause IA-32 Double Faults (IA-32 vector 8) are not generated by the processor in the Itanium System Environment.

Name **Invalid TSS Fault**

Cause IA-32 Invalid TSS Faults (IA-32 vector 10) are not generated in the Itanium System Environment.

Name **IA_32_Exception (NotPresent) – Segment Not Present Fault**

Cause Generated when the processor detects the Present-bit of the memory segment descriptor is zero during an IA-32 segment load or far control transfer instructions. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault and error codes.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of **t**he faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 11.

ISR.code – IA-32 defined error code. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								11								error_code															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (StackFault) – Stack Fault**

Cause IA-32 defined set of stack segment fault conditions detected during stack segment load operations or memory references relative to the stack segment, refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete list of all IA-32 faulting conditions. Stack faults can also be generated when the processor detects an inconsistent stack segment register descriptor value during an IA-32 stack reference instruction (e.g. PUSH, POP, CALL, RET,). See section "Segment Descriptor and Environment Integrity" for a list of possible inconsistent register descriptor conditions.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 12.

ISR.code – IA-32 defined ErrorCode. Zero if an inconsistent register descriptor is detected during a memory reference relative to the stack segment.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								12								error_code or zero															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (GPFault) – General Protection Fault**

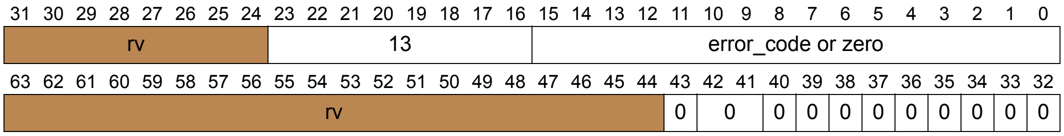
Cause IA-32 defined set of data and code segment fault conditions detected during data or code segment load operations or memory references relative to code or data segments, refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete list of all IA-32 General Protection Fault conditions. General Protection faults can also be generated when the processor detects an inconsistent code or data segment register descriptor value during an IA-32 code fetch or data memory reference. See section "Segment Descriptor and Environment Integrity" for a list of possible inconsistent register descriptor conditions.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 13.

ISR.code – IA-32 defined ErrorCode. Zero if an inconsistent register descriptor is detected during a memory reference relative to a code or data segment.



Name **Page Fault**

Cause IA-32 defined page faults (IA-32 vector 14) can not be generated in the Itanium System Environment.

Name **IA_32_Exception (FPError) – Pending Floating-point Error**

Cause An unmasked IA-32 floating-point exception is delivered on the next non-control IA-32 floating-point, MMX technology, WAIT, or `jmpe` instruction trigger delivery of this exception. Floating-point errors are delivered regardless of the state of CR0.ne in the Itanium System Environment. IA-32 numeric exception delivery is not triggered by Itanium numeric exceptions or the execution of Itanium numeric instructions. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

FSR, FIR, FDR and FCR contain the IA-32 floating-point environment and exception information.

ISR.vector – 16.

.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								16								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **IA_32_Exception (AlignmentCheck) – Alignment Check Fault**

Cause An IA-32 instruction performed an unaligned data memory reference while PSR.ac is 1, or EFLAG.ac is 1 and CR0.am is 1 and the effective privilege level is 3. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

IFA – referenced virtual data address (byte granular) zero extended to 64-bits.

ISR.vector – 17.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								17								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Name **Machine Check**

Cause IA-32 Machine Check (IA-32 vector 18) is not generated in the Itanium System Environment.

Name **IA_32_Exception (StreamingSIMD) – SSE Numeric Error Fault**

Cause An unmasked IA-32 SSE numeric error occurred. Numeric faults generated on SSE instructions are reported precisely on the faulting SSE instruction. SSE instructions do NOT trigger the report of any pending IA-32 floating-point exceptions. SSE instructions always ignore CR0.ne and the IGNNE pin. Refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for a complete definition of this fault.

Parameters IIP – virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

ISR.vector – 19.

.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">rv</td><td colspan="8">19</td><td colspan="16">0</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">rv</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Name **IA_32_Interrupt (Vector #N) – Software Trap**

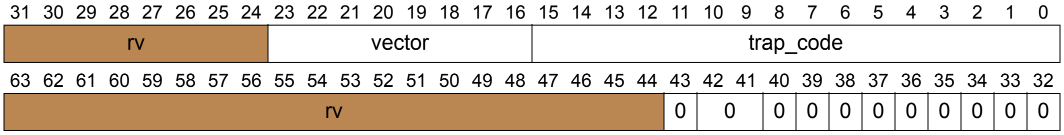
Cause The IA-32 INT n instruction forces an IA-32 interrupt trap. The IA-32 IDT is not consulted nor are any values pushed onto a memory stack.

Parameters IIPA – trapping virtual IA-32 instruction address (points to the INT instruction) zero extended to 64-bits.

IIP – next virtual IA-32 instruction address zero extended to 64-bits.

ISR.vector – vector number.

ISR.code – TrapCode, Indicates Concurrent Single Step Trap condition.



Name **IA_32_Intercept (Instruction) – Instruction Intercept Fault**

Cause Execution of unimplemented IA-32 opcodes, illegal opcodes or sensitive privileged IA-32 operating system instructions results in an instruction intercept. Intercepted opcodes include (but are not limited to); CLTS, HLT, INVD, INVLPG, IRET, LIDT, LGDT, LLDT, LMSW, LTR, MOV to CRs, MOV to/from DRs, RDMSR, RSM, SYSENTER, SYSEXIT, INT1, SIDT, SGDT, SLDT, SMSW, WBINVD, WRMSR, and all other unimplemented and illegal opcode patterns. If CR0.em is 1, execution of all IA-32 Intel MMX technology and IA-32 SSE instructions results in this intercept. If CR4.FXSR is 0, execution of all IA-32 SSE instructions results in this intercept. All illegal IA-32 floating-point opcodes result in an IA_32_Intercept(Instruction) regardless of the state of CR0.em. Intercepted opcodes are nullified and alter no architectural state.

Parameters IIP – Virtual IA-32 instruction address zero extended to 64-bits, points to the first byte of the intercepted IA-32 opcode (including prefixes).

IIPA – Virtual address of the faulting IA-32 instruction zero extended to 64-bits.

IIM – Opcode bytes, contains the first 8-bytes of the IA-32 instruction following all prefix bytes. All prefix bytes are decoded and presented as a bitmask in the Intercept Code along with the prefix length in bytes. Opcode bytes are loaded into IIM in the same format as encountered in memory and as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***. The lowest memory address byte is placed in byte 0 of IIM, higher memory address bytes are placed in increasingly higher numbered bytes within IIM.

The 8-byte opcode loaded into IIM is stripped of the following prefixes; lock, repeat, address size, operand size, and segment override prefixes (opcode bytes 0xF3, 0xF2, 0xF0, 0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65, 0x66, and 0x67). The 0x0F opcode series prefix is not stripped from the opcode bytes loaded into IIM. The opcode loaded into IIM includes all IA-32 opcode components, including 1 to 3 bytes of opcode, mod r/m bytes, sib bytes and any possible immediates and/or displacements.

If the opcode loaded in IIM is less than 8-bytes, the remainder higher order numbered bytes are set to 0. If the opcode is larger than 8-bytes, bytes after the 8th byte (following all stripped prefixes) are not reported. If required, emulation code must retrieve the extra opcode bytes by reading from the memory locations specified by IIP.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
byte3								byte2								byte1								byte0							
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
byte7								byte6								byte5								byte4							

ISR.vector – 0, indicates instruction intercept.

ISR.code – Intercept Code indicates prefixes and prefix lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								0								intercept_code															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	0	0	0

Figure 9-3 defines intercept codes for IA-32 instruction set intercepts. Intercept code fields are defined by Table 9-1 and Table 9-2 on page 2:234.

Figure 9-3. IA-32 Intercept Code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
len				0		seg			sp	np	rp	lp	as	os	0

Table 9-1. Intercept Code Definition

Field	Bits	Description
os	1	Operand Size – (OperandSize Prefix XOR CSD.d bit). When 1, indicates the effective operand size is 32-bits, when 0, 16-bits.
as	2	Address Size – (AddressSize Prefix XOR CSD.d bit). When 1, indicates the effective address size is 32-bits, when 0, 16-bits.
lp	3	Lock Prefix – If 1, indicates a lock prefix is present.
rp	4	REP or REPE/REPZ Prefix – If 1, indicates a REP/REPE/REPZ prefix is in effect.
np	5	REPNE/REPNZ Prefix – If 1, indicates a REPNE/REPNZ prefix is in effect.
sp	6	Segment Prefix – If 1, indicates a Segment Override prefix is present.
seg	7:9	Segment Value – Segment Prefix Override value, see Figure 9-2 for encodings. If there is no segment prefixes this field is undefined.
len	12:15	Length of Prefixes – Length of all prefix (in bytes) stripped from IIM. If there are no prefixes this field has a value of zero.

Table 9-2. Segment Prefix Override Encodings

Seg Value	Segment Prefix
0	ES Segment Override
1	CS Segment Override
2	SS Segment Override
3	DS Segment Override
4	FS Segment Override
5	GS Segment Override
6	reserved
7	reserved

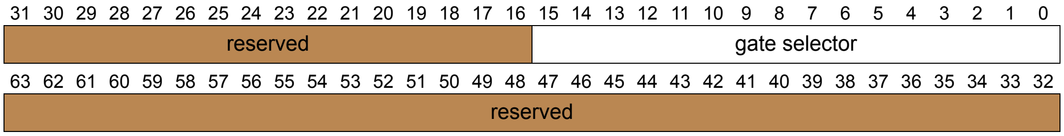
Name **IA_32_Intercept (Gate) – Gate Intercept Trap**

Cause If an IA-32 control transfer is initiated through a GDT/LDT descriptor that transfers control through a Call Gate, Task Gate or Task Segment this interception trap is generated.

Parameters IIPA – trapping virtual IA-32 instruction address zero extended to 64-bits.

IIP – next sequential virtual IA-32 instruction address zero extended to 64-bits.

IFA – Gate Selector. The gate selector is loaded in IFA{15:0}.



IIM – Gate, Task Gate or Task Segment Descriptor. The descriptor loaded in IIM adheres to the IA-32 GDT/LDT memory format, where byte 0 of the descriptor is in IIM{7:0}.

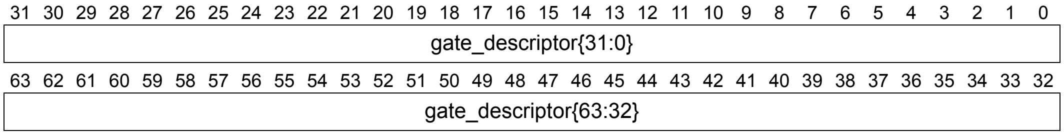


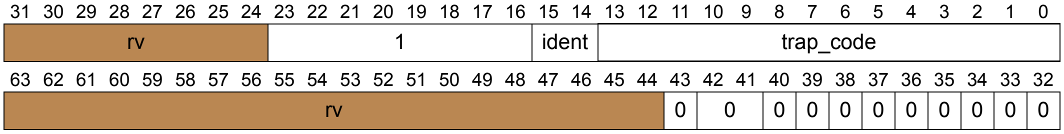
Table 9-3. Gate Intercept Trap Code Identifier

Instruction	ISR.code{15:14}
CALL	00
JMP	01

ISR.vector – 1, indicates gate interception.

ISR.code – TrapCode, Indicates Concurrent Data Debug, taken Branch, and Single Step Events.

ISR.code{15:14} – indicates whether CALL or JMP generated the trap. See Table 9-3 for details.



Name IA_32_Intercept (SystemFlag) – System Flag Trap

Parameters System Flag Intercept Traps are generated for the following conditions:

CLI, STI, POPF, POPFD instructions. If the EFLAG.if bit changes state and CFLG.ii is 1, or EFLAG.tf or EFLAG.ac change state, a System Flag intercept notification trap is delivered after the instruction completes. IIM contains the previous value of EFLAG before the trapping instruction executed. If IA-32 code does not have IOPL or CPL permission to modify the EFLAG bits, no intercept is generated. This intercept trap condition can be used to provide virtual interrupt services, and delay enabling of interrupts after the STI instruction.

MOV SS, POP SS instructions. After these instructions complete execution, a System Flag intercept notification trap is delivered. This intercept trap condition can be used to inhibit interrupts, and code breakpoints between Mov/Pop SS and the next instruction and to inhibit Single Step and Data Breakpoint traps on the Mov, or Pop SS instruction.

IIP – next virtual IA-32 instruction address zero extended to 64-bits.

IIPA – trapping virtual IA-32 instruction address zero extended to 64-bits.

IIM – contains the previous EFLAG value before the trapping instruction.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">old EFLAG</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="32">reserved</td></tr>
</table>

ISR.vector – 2.

ISR.code – Trap Code, indicates Concurrent Single Step Trap, Debug trap condition.

ISR.code{15:14} indicates which instruction generated the trap.

Table 9-4. System Flag Intercept Instruction Trap Code Instruction Identifier

Instruction	ISR.code{15:14}
CLI	00
STI	01
POPF, POPFD	10
MOV/POP SS	11

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">rv</td><td colspan="8">2</td><td colspan="2">ident</td><td colspan="14">trap_code</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="20">rv</td><td>0</td><td colspan="2">0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Name **IA_32_Intercept (Lock) – Locked Data Reference Fault**

Cause For IA-32 locked operations, if the DCR.lc bit is 1, and an atomic operation to made to non-write-back memory or to unaligned write-back memory that would result in a read-modify-write sequence being performed externally under an external bus lock, the processor raises a Locked Data Reference fault.

Parameters IIP – faulting virtual IA-32 instruction address zero extended to 64-bits.

IIPA – virtual address of the faulting IA-32 instruction zero extended to 64-bits.

IFA – faulting virtual data address (byte granular) zero extended to 64-bits.

ISR.vector – 4.

ISR.code – 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rv								4								0															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
rv																				0	0		0	0	0	0	0	0	1	1	0

§

Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications 10

This section describes the IA-32 system execution model from the perspective of an Itanium architecture-based operating system interfacing with IA-32 code, while operating in the Itanium System Environment. The main features covered are:

- IA-32 system and control register behavior
- IA-32 virtual memory support
- IA-32 fault and trap handling
- IA-32 instruction behavior

10.1 Instruction Set Transitions

Instruction set transitions are defined in Section 6.2.1, “Instruction Set Modes” on page 1:110. Operating systems can disable instruction set transitions (`jmpe` and `br.ia`) by setting PSR.di to one. If PSR.di is one, execution of `jmpe` or `br.ia` to IA-32 target results in a Disabled Instruction Set Transition Fault, and the operation is nullified.

The processor also transitions into an Itanium architecture-based operating system when IA-32 privileged system resources are accessed, on an interruption, or when the following conditions are detected:

- Instruction Interception – IA-32 system level privileged instructions are executed
- System Flag Interception – Various EFLAG system flags are modified, (e.g. AC, TF and IF-bits)
- Gate Interception – Control transfers are made through call gate, or transfers through a task switch (TSS segment or Task Gate).

All software interrupts, external interrupts, faults, traps and machine checks transition the processor to the Itanium instruction set, regardless of the state of PSR.di. IA-32 defined exceptions and software interrupts are delivered to Itanium architecture-based interruption handlers.

10.2 System Register Model

Registers are assigned the following conventions during transitions between IA-32 and Itanium instruction sets.

- **IA-32 State**: The register contains an IA-32 register during IA-32 instruction set execution. Expected IA-32 values should be loaded before switching to the IA-32 instruction set. After completion of IA-32 instructions, these registers contain the results of the execution of IA-32 instructions. These registers may contain any value during Itanium instruction execution according to Itanium software conventions. Software should follow IA-32 and Itanium software calling conventions for these registers.

- **Shared**: Shared registers contain values that have similar functionality in either instruction set. For example, all Itanium control registers, debug registers are used for memory references (including IA-32). The stack pointer (ESP) and instruction pointer (IP) are also shared.
- **Unmodified**: These registers are not altered by IA-32 execution. Itanium architecture-based code can rely on these values not being modified during IA-32 instruction set execution. The register will have the have the same contents when entering the IA-32 instruction set and when exiting the IA-32 instruction set.
- **Undefined**: Registers marked as undefined may be used as scratch areas for execution of IA-32 instructions. Software can not rely on the value of these registers across an instruction set transition.

Table 10-1. IA-32 System Register Mapping

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
Application Registers				
EFLAG	EFLAG	IA-32 state	32	IA-32 System/Arithmetic flags, writes of some bits are conditioned by PSR.cpl and EFLAG.iopl.
CSD	CSD		64	IA-32 code segment (register format)
SSD	SSD			IA-32 stack segment (register format)
CFLG	CR0/CR4		64	IA-32 control flags, CR0=CFLG{31:0}, CR4=CFLG{63:32}[a], writable at PSR.cpl=0 only.
Kernel Registers				
KR0	IOBASE[b]	IA-32 state	64	IA-32 virtual I/O port Base register
KR1	TSSD[c]			IA-32 TSS descriptor (register format)
KR2	CR3/CR2[d]			IA-32 CR2=KR2{63:32}, CR3=KR2{31:0}
KR3-7		unmodified		Intel Itanium preserved registers
Banked General Registers				
GR16-31		unmodified		Preserved for operating system use
Control Registers				
DCR		unmodified, shared		Controls instruction set execution (including IA-32)
IFA, IIP, IPSR, ISR, IIM, IIPA, ITIR, IHA, IIB0-1, IFS, IVA		shared	64	Intel Itanium interruption registers may be overwritten on any TLB fault, interruption or exception encountered during IA-32 or Intel Itanium instruction set execution.
PTA		shared	64	Shared page table base for memory references (including IA-32)
ITM		shared		shared Intel Itanium interruption/timer resources

Table 10-1. IA-32 System Register Mapping (Continued)

Intel® Itanium® Reg	IA-32 Reg	Convention	Size	Description
LID, IVR, TPR, EOI, IRR0, IRR1, IRR2, IRR3, ITV, PMV, LRR0, LRR1, CMCV		shared	64	Intel Itanium external interrupt control registers are used to generate, prioritize and delivery external interrupts during IA-32 or Intel Itanium instruction set execution.
Translation Resources				
TRs		shared		All Intel Itanium virtual memory registers can be used for memory references (including IA-32).
TCs				
RRs				
PKRs				
Debug Registers				
IBRs	dr0-3, dr7	shared	64	Intel Itanium debug registers are used memory references (including IA-32).
DBRs	dr0-3, dr7			
Performance Monitors				
PMCs		shared	64	Intel Itanium performance monitors measure performance events (including IA-32).
PMDs		shared	64	reflect performance monitor results of execution (including IA-32)

a. IA-32 MOV from CR0 and CR4 return the value in the CFLG register.

b. The IOBase register is used by IN/OUT instructions. If IN/OUT operations are disabled via CFLG.io, this register can be used for other values.

c. The TSSD registers are used by IN/OUT instructions for I/O permission via CFLG.io. If access to the TSS is disabled, these registers can be used for other values.

d. The Mov from CR2,CR3 instructions return the value contained in KR2.

10.3 IA-32 System Segment Registers

System Descriptors are maintained in an unscrambled format shown in Figure 10-1 that differs from the IA-32 scrambled memory descriptor format. The unscrambled register format is designed to support fast conversion of IA-32 segmented 16/32-bit pointers into virtual addresses by Itanium architecture-based code. IA-32 segment register load instructions unscramble the GDT/LDT memory format into the descriptor register format on a segment register load. Itanium architecture-based software can also directly load descriptor registers provided they are properly unscrambled by software. When Itanium architecture-based software loads these registers, no data integrity checks are performed at that time if illegal values are loaded in any fields. For a complete definition of all bit fields and field semantics refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

Figure 10-1. IA-32 System Segment Register Descriptor Format (LDT, GDT, TSS)

63	62 60	59	58 57	56	55 52	51 32	31 0
g	ig	p	dpl	s	stype	lim{19:0}	base{31:0}

Table 10-2. IA-32 System Segment Register Fields (LDT, GDT, TSS)

Field	Bits	Description
base	31:0	Segment Base value. This value when zero extended to 64-bits, points to the start of the segment in the 64-bit virtual address space for IA-32 instruction set memory references. This value is ignored for Intel Itanium instruction set memory references.
lim	51:32	Segment Limit. Contains the maximum effective address value within the segment. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details and segment limit fault conditions.
stype	55:52	Segment Type identifier. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for encodings and definition.
s	56	Non System Segment. If 1, a data segment, if 0 a system segment.
dpl	58:57	Descriptor Privilege Level. The DPL is checked for memory access permission for IA-32 instruction set memory references.
p	59	Segment Present bit. If 0, and an IA-32 memory reference uses this segment an IA_Exception(GPFault) is generated.
ig	62:60	Ignored – For the LDT/GDT/TSS descriptors reads of this field return the last value written by Itanium architecture-based code. Reads of this field return zero if written by IA-32 descriptor loads.This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by software.
g	63	Segment Limit Granularity. If 1, scales the segment limit by lim=(lim<<12) | 0xFFF for IA-32 instruction set memory references.

System segment selectors and descriptors for GDT and LDT are maintained in Itanium general registers to support segment register loads used extensively by segmented 16-bit code. On the transition into the IA-32 instruction set, GDT/LDT descriptor table must be initialized if IA-32 code will perform protected mode segment register loads or far control transfers.

Within the IA-32 System Environment, GDT and LDT are considered privileged operating system segmentation resources. However, in the Itanium System Environment, applications can transition between the IA-32 and Itanium instruction set and bypass IA-32 segmentation. Itanium user level instructions can also directly modify all selectors and descriptors including GDT and LDT. An operating system should either protect memory with virtual memory management mechanisms defined by the Itanium architecture or disabled application level instruction set transitions. Within the Itanium System Environment, GDT/LDT memory spaces must be mapped into user space, since supervisor overrides for accesses to GDT/LDT are disabled.

The TSSD descriptor points to the I/O Permission Bitmap. If CFLG.io is 1, IN, INS, OUT, and OUTS consult the TSSD I/O permission bitmap as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***. If CFLG.io is 0, the TSSD I/O permission bitmap is not checked. See Section 10.7, "I/O Port Space Model" for details on I/O port permission and for TLB-based access control. The TSSD register is not used within the Itanium System Environment to support task switches, or interlevel control transfers. If the TSSD is used for I/O Permissions, Itanium architecture-based operating system software must ensure that a valid 286 or 386 Task State Descriptor is loaded, otherwise IN/OUT operations to the TSSD I/O permission bitmap will result in undefined behavior.

The IDT descriptor is not supported or defined within the Itanium System Environment.

10.3.1 IA-32 Current Privilege Level

PSR.cpl is the current privilege level of the processor for instruction execution (including IA-32). PSR.cpl is used by the processor for all IA-32 descriptor segmentation and paging permission checks. PSR.cpl is a secured register. Typical IA-32 processors used SSD.dpl as the official privilege level of the processor. Since, SSD.dpl is not secured from user modification, processor implementations must base all privilege checks and state backups based on PSR.cpl.

10.3.2 IA-32 System EFLAG Register

The EFLAG (AR24) register is made of two major components, user arithmetic flags (CF, PF, AF, ZF, SF, OF, and ID) and system control flags (TF, IF, IOPL, NT, RF, VM, AC, VIF, VIP). None of the arithmetic or system flags affect Itanium instruction execution. The arithmetic flags are used by the IA-32 instruction set to reflect the status of IA-32 operations, control IA-32 string operations, and control branch conditions for IA-32 instructions. System flags are typically managed by an operating system and are used to control the overall operations of the processor. System flags are broken into two categories, system flags that control IA-32 instruction set execution behavior and virtualizable system flags. The NT system flag shown in bold font in Figure 10-2 is virtualized.

Figure 10-2. IA-32 EFLAG Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved (set to 0)										id	vip	vif	ac	vm	rf	0	nt	iopl		of	df	if	tf	sf	zf	0	af	0	pf	1	cf
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)																															

System flags AC, TF, RF, VIF, VIP, IOPL and VM directly control the execution of IA-32 instructions. These bits do not control any Itanium instructions. See Table 10-3 for a complete definition these bits.

The NT bit does not directly control the execution of any IA-32 or Itanium instructions. All IA-32 instructions that modify this bit is intercepted (e.g. IRET, Task Switches)

See Table 10-3, "IA-32 EFLAG Field Definition" for the behavior on IA-32 and Itanium instruction reads/writes to this application register.

10.3.2.1 Virtualized Interrupt Flag

To provide for virtualization of IA-32 code, the IF bit is virtualizable in the context of an operating system. Interrupts are enabled for IA-32 instructions, if `(PSR.i and (~CFLG.if or EFLAG.if))` is true. For Itanium architecture-based code, interrupts are enabled if PSR.i is 1.

An optional System Flag intercept trap can be generated if CFLG.ii is 1, and the IF-flag changes state due to IA-32 code executing CLI, STI, or POPF. See Section 10.3.3.1, "IA-32 Control Registers" on page 2:246 for CFLG details. Using this model, virtualization code can set CFLG.if to 0 and CFLG.ii to 0, IA-32 instruction set modifications of EFLAG.if does not affect actual interrupt masking, therefore no notification events need be sent to virtualizing software. When virtualization code, detects and queues an external interrupt for delivery into a virtualized IA-32 operating system/application, it can set CFLG.ii to1 to force notification the next time the IF-bit changes state, indicating IA-32 code is either opening or closing the interrupt window. Setting CFLG.if to 1, allows for direct IA-32 control of interrupt masking.

Virtualization of the IF flag is independent of VME extensions. Both mechanisms can be used independently, see the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the complete VME definition.

Table 10-3. IA-32 EFLAG Field Definition

EFLAG[a]	Bits	Description
EFLAG.cf	0	IA-32 Carry Flag. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
	1	Ignored – For IA-32 instructions, writes are ignored, reads return one. For Itanium instructions, the implementation can either ignore writes and return one on reads; or write the value, and return the last value written on reads.
	3,5, 15	Ignored – For IA-32 instructions, writes are ignored, reads return zero. For Itanium instructions, the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads.
EFLAG.pf	2	IA-32 Parity Flag. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
EFLAG.af	4	IA-32 Aux Flag. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
EFLAG.zf	6	IA-32 Zero Flag. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
EFLAG.sf	7	IA-32 Sign Flag. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
EFLAG.tf	8	IA-32 Trap Flag- In the Intel Itanium System Environment, IA-32 instruction single stepping is enabled when EFLAG.tf is 1 or PSR.ss is 1. EFLAG.tf does not control single stepping for Intel Itanium instruction set execution. When single stepping is enabled, the processor generates a IA_32_Exception(Debug) trap event after the successful execution of each IA-32 instruction. If EFLAG.tf is modified by the POPF or POPFD instruction an IA_32_Intercept(SystemFlag) trap is raised. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
EFLAG.if	9	IA-32 Interruption Flag. In the Intel Itanium System Environment, when PSR.i and (~CFLG.if or EFLAG.if) is 1, external interrupts are enabled during IA-32 instruction set execution, otherwise external interrupts are held pending. If CFLG.if is 1, modification of the EFLAG.if directly affects external interrupt enabling. If CFLG.if is 0, EFLAG.if does not affect interrupt enabling. The IF-bit does not affect external interrupt enabling for Intel Itanium instructions nor NMI interrupts. The IF bit can be modified by IA-32 and Itanium architecture-based code only when PSR.cpl is less than or equal to EFLAG.iopl. If PSR.cpl is greater than EFLAG.iopl, writes to the IF-bit are silently ignored. If CFLG.ii is 1, successful modification of the IF-bit by CLI, STI, or POPF results in an IA_32_Intercept(SystemFlag) trap, otherwise the IF-bit is modified without interception. Modification of this bit by Intel Itanium instructions does not result in an intercept. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
EFLAG.df	10	IA-32 Direction Flag. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.
EFLAG.of	11	IA-32 Overflow Flag. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details.

Table 10-3. IA-32 EFLAG Field Definition (Continued)

EFLAG[a]	Bits	Description
EFLAG.iopl	13:12	IA-32 In/Out Privilege Level, controls accessibility by IA-32 IN/OUT instructions to the I/O port space and permission to modify the IF-bit for Intel Itanium and IA-32 instructions. If PSR.cpl > IOPL, permission is denied for IA-32 IN/OUT instructions, and modifications of EFLAG.if by either IA-32 or Intel Itanium instructions are ignored. IOPL can only be modified by IA-32 or Intel Itanium instructions executing at privilege level 0, otherwise modifications of this bit are silently ignored. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
EFLAG.nt	14	IA-32 Nested Task switch. In the IA-32 System Environment, indicates a nested task flag when chaining interrupted and called IA-32 tasks. IA-32 task switches are not directly supported in the Intel Itanium System Environment, since IRET, interruptions, calls, and jumps through task gates are always intercepted. EFLAG.nt can be modified by the POPF or POPFD instruction in both system environments. Modification of EFLAG.nt by POPF and POPFD does not result in a System Flag Intercept. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
EFLAG.rf	16	IA-32 Resume Flag. In the Intel Itanium System Environment, when EFLAG.rf or PSR.id is 1, code breakpoint faults are temporarily disabled for one IA-32 instruction, so that IA-32 instructions can be restarted after a code breakpoint fault without causing another code breakpoint fault. EFLAG.rf does not affect Intel Itanium Instruction Debug faults. After the successful execution of each IA-32 instruction, PSR.id and EFLAG.rf are cleared to zero. On entry into the IA-32 instruction set via `rfi` or `br.ia`, EFLAG.rf and PSR.id is not cleared until the successful completion of the first (target) IA-32 instruction. `jmpe` clears the PSR.id and the EFLAG.rf bit. EFLAG.rf is set to 1 if a repeat string sequence (REP MOVS, SCANS, CMPS, LODS, STOS, INS, OUTS) takes an external interrupt, trap or fault before the final iteration. EFLAG.rf and PSR.id are set to 0 after the last iteration. For all other cases, external interrupts, faults, traps, and intercept conditions EFLAG.rf is unmodified. The RF-bit can be modified by Intel Itanium instructions running at any privilege level. IA-32 instructions cannot directly modify the RF-bit or PSR.id. Specifically, POPF cannot modify the RF-bit and execution of IRET is always intercepted in the Intel Itanium System Environment. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
EFLAG.vm	17	IA-32 Virtual Mode 86. When 1, IA-32 instructions execute in the VM86 environment. This bit can only be modified by IA-32 or Intel Itanium instructions executing at privilege ring 0, otherwise modifications of this bit by Intel Itanium or IA-32 instructions is silently ignored. Itanium architecture-based software is responsible for initializing the processor with the required VM86 register state before transferring to IA-32 VM86 environment. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for complete details of the VM86 environment. Software must ensure the processor is in IA-32 Protected Mode when setting the VM bit.
EFLAG.ac	18	IA-32 Alignment Check. In the Intel Itanium System Environment, IA-32 instructions raise an IA_32_Exception(AlignmentCheck) fault if an unaligned reference is performed and PSR.ac is 1 or (CFLG.am is 1 and EFLAG.ac is 1 and memory is accessed at an effective privilege level of 3). Neither EFLAG.ac, CR0.am nor privilege level affect alignment check faults for Intel Itanium instructions. See Section 10.6.7, "Memory Alignment" on page 2:263 for details on alignment conditions. This bit can be modified by IA-32 and Intel Itanium instructions at any privilege level. Modification of this bit by the POPF instructions results in an IA_32_Intercept(SystemFlag) trap. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.

Table 10-3. IA-32 EFLAG Field Definition (Continued)

EFLAG[a]	Bits	Description
EFLAG.vif	19	IA-32 Virtual Interrupt Flag. See VME extensions in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details. Affects execution of POPF, PUSHF, CLI and STI. This bit is supported in both the IA-32 and Intel Itanium System Environments. A IA-32 Code Fetch fault (GPFault(0)) is generated on every IA-32 instruction (including the target of `rfi` and `br.ia`), if the following condition is true: EFLAG.vip & EFLAG.vif & CFLG.pe & PSR.cpl==3 & (CFLG.pvi | (EFLAG.vm & CFLG.vme))
EFLAG.vip	20	IA-32 Virtual Interrupt Pending. See VME extensions in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for programming details. Affects execution of POPF, PUSHF, CLI and STI. This bit is supported in both the IA-32 and Intel Itanium System Environments.
EFLAG.id	21	IA-32 Identifier bit, can be written and read by IA-32 instructions, indicates IA-32 CPUID instruction is supported. This bit is supported in both the IA-32 and Intel Itanium System Environments.
	63:22	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

[a] On entry into the IA-32 instruction set all bits may be read by subsequent IA-32 instructions, after exit from the IA-32 instruction set these bits represent the results of all prior IA-32 instructions. None of the EFLAG bits alter the behavior of Itanium instruction set execution.

10.3.3 IA-32 System Registers

IA-32 system registers such as CR3, CR2, debug registers, performance counters. IA-32 control registers do not affect execution of Itanium instructions. All IA-32 privileged instructions that access prior IA-32 system registers are intercepted.

10.3.3.1 IA-32 Control Registers

IA-32 control registers CR0 and CR4 are mapped into the Itanium application register CFLG (AR27). IA-32 control bits, shown in Figure 10-3, only control execution of the IA-32 instruction set. Additional CR0 bits are defined in CFLG to control virtualization of IA-32 code (namely the IO and IF bits) as shown in Figure 10-3. CFLG is readable by Itanium architecture-based code at all privilege levels but can only be written at privilege level 0, otherwise a Privileged Register fault is generated. When Itanium architecture-based software loads this application register (AR24), a Reserved Register/Field fault will be raised if a non-zero value is written into bits listed as reserved.

Figure 10-3. Control Flag Register (CFLG, AR27)

31	30	29	28272625242322212019	18	17	16	1514131211	10	9	8	7	6	5	4	3	2	1	0
PG	CD	NW	ignored (set to 0)	AM	ig	WP	ignored (set to 0)			II	IF	IO	NE	ET	TS	EM	MP	PE
63	62	61	60595857565554535251	50	49	48	4746454443	42	41	40	39	38	37	36	35	34	33	32
reserved (set to 0)								MMXEX	FXSR	PCE	PGE	MCE	PAE	PSE	DE	TSD	PVI	VME

- State in italics is virtualized. This state has no impact on a IA-32 or Itanium instruction set execution**.**
- State in bold only affects IA-32 instruction set execution, Itanium instruction execution is not affected.

Table 10-4 defines all IA-32 control register state and the behavior of each bit in the Itanium System Environment.

Table 10-4. IA-32 Control Register Field Definition

Field	Intel® Itanium® State	Bits	Description
CR0	CFLG{31:0}		CR0: IA-32 Mov to CR0 result in a instruction interception fault. Mov from CR0 returns the value contained in CFLG{31:0}. Modification of CFLG{31:0} by Intel Itanium instructions only alters the CR0 state, no side effects (such as TLB flushes) occur.
CR0.PE	CFLG.pe	0	Protected Mode Enable: This bit determines whether the processor operates in IA-32 Protected Mode or Real Mode. This bit affects only IA-32 instruction set execution, Intel Itanium operations are not affected by this bit. Modification of this bit by Itanium architecture-based code does have NOT any side effects such as flushing the TLBs. This bit is supported in both the IA-32 and Intel Itanium System Environments. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit and the Protected Mode environment.
CR0.MP	CFLG.mp	1	Monitor co-Processor: When CFLG.ts is 1 and CFLG.mp is 1, execution of IA-32 FWAIT/WAIT instructions results in an Device Not Available fault. This bit is ignored by Intel Itanium floating-point instructions. This bit is supported in both IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
CR0.EM	CFLG.em	2	Emulation: When CFLG.em is set, execution of IA-32 ESC and floating-point instructions generates an IA_32_Exception(DNA) fault. When CFLG.em is 1, execution of IA-32 MMX technology or SSE instructions results in an IA_32_Intercept (Instruction) fault. This bit does not affect Intel Itanium floating-point instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
CR0.TS	CFLG.ts	3	Task Switched: When CFLG.ts is 1, execution of an IA-32 ESC, floating-point instruction, MMX technology or SSE instruction results in a IA_32_Exception(DNA) fault. When CFLG.ts is 1 and CFLG.mp is 1, execution of IA-32 FWAIT/WAIT instructions results in an IA_32_Exception(DNA) fault. This bit is ignored by Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
CR0.ET	CFLG.et	4	Extension Type: ET is ignored since i387 co-processor instructions are supported. This bit is reserved on all Pentium processors. Reads always return 1. This bit is supported in both the IA-32 and Intel Itanium System Environments.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Field	Intel® Itanium® State	Bits	Description
CR0.NE	CFLG.ne	5	Numeric Error: Numeric errors are always enabled in the Intel Itanium System Environment. The NE bit and the IGNNE# pin are ignored by the processor and the FERR# pin is not asserted for any numeric errors on IA-32 or Intel Itanium floating-point instructions. In the IA-32 System Environment, this bit is supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.
--	CFLG.io	6	I/O Enable: If CFLG.io is 1 and CPL>IOPL, IA-32 IN, INS, OUT, OUTS instructions consulted the TSS for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced). This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
--	CFLG.if	7	IF Enable: When CFLG.if is 1, EFLAG.if can be used to enabled or disable external interrupts for IA-32 instructions. If CFLG.if is 0, EFLAG.if does not control external interrupt enabling. External interrupts are enabled for the IA-32 instruction set by if PSR.i and (~CLFG.if or EFLAG.if). This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
--	CFLG.ii	8	IF Intercept: When CFLG.ii is 1, successful modification of the EFLAG.if bit by IA-32 CLI, STI or POPF instructions result in a IA_32_Intercept(SystemFlag) trap. This bit always returns zero when read by the IA-32 Mov from CR0 instruction. This bit is not defined in the IA-32 System Environment.
ignored		9:15, 17, 19:28	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by IA-32 software. For Itanium instructions, the implementation can either ignore the writes and return zero on reads, or write the value and return the last value written on reads.
CR0.WP	CFLG.wp	16	Write Protect: This bit is ignored in the Itanium System Environment. In the IA-32 System Environment, WP controls supervisor write-protection for IA-32 paging. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Field	Intel® Itanium® State	Bits	Description
CR0.AM	CFLG.am	18	Alignment Mask: For IA-32 instructions an IA_32_Exception(AlignmentCheck) fault is generated on a reference to an unaligned data memory operand if PSR.ac is 1 or (CFLG.am is 1 and EFLAG.ac is 1 and memory is accessed at an effective privilege level of 3). Neither EFLAG.ac, CR0.am nor privilege level affect alignment check faults for Itanium instructions. This bit is supported in both the IA-32 and Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on this bit.
CR0.NW	CFLG.nw	29	Not Write-through and Cache Disable: These bits are ignored in the Itanium System Environment. Cacheability is controlled virtual memory attributes. These bits are provided as storage for compatibility purposes.
CR0.CD	CFLG.cd	30	
CR0.PG	CFLG.pg	31	Paging Enable: In the Intel Itanium System Environment, this bit is ignored for IA-32 and Intel Itanium memory references. Virtual translations are enabled via PSR.it and PSR.dt. This bit is provided as storage for compatibility purposes. Modification of this bit by Itanium architecture-based code does NOT have any side effects such as flushing the TLBs. This bit is supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the IA-32 System Environment.
CR2	KR2{63:32}		IA-32 Page Fault Virtual Address: IA-32 Mov to CR2 result in an interception fault. Mov from CR2 returns the value contained in KR2{63:32}. CR2 is replaced by IFA in the Intel Itanium System Environment.
CR3	KR2{31:0}		IA-32 Page Table Address: IA-32 Mov to CR3 result in an interception fault. Mov from CR3 return the value contained in KR2{31:0}. CR3 is replaced by PTA in the Intel Itanium System Environment. Modification of KR2{31:0} by Itanium architecture-based code does NOT have the side effect of flushing the TLBs.
CR3.PWT	KR4.pwt		Page Write-Through and Cache Disabled: In the Intel Itanium System Environment, these bits are ignored. This bit are provided as storage for compatibility purposes. These bits are supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the IA-32 System Environment.
CR3.PCD	KR4.pcd		
CR4	CFLG{63:32}		CR4: A-32 Mov to CR4 result in an instruction interception fault. Mov from CR4 returns the value contained in CFLG{63:32}. Modification of CFLG{63:32} by Intel Itanium instructions only alters the register state, no side effects (such as TLB flushes) occur.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Field	Intel® Itanium® State	Bits	Description
CR4.VME	CFLG.vme	32	IA-32 Virtual Machine Extension and Protected Mode Virtual Interrupt Enable: These bits control the VM86 VME extensions and Protected Mode Virtual Interrupt extensions defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for STI, CLI and PUSHF. These bits have no effect on Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments.
CR4.PVI	CFLG.pvi	33	
CR4.TSD	CFLG.tsd	34	Time Stamp Disable: IA-32 RDTSC user level reads of the Time Stamp Counter are enabled when CR4.tsd when zero. Otherwise execution of the RDTSC instruction results in a GPFault. CFLG.tsd is ignored by Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on these bits.
CR4.DE	CFLG.de	25	Debug Extensions: In the Intel Itanium System Environment, this bit is ignored by IA-32 or Intel Itanium references to the I/O port space. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the IA-32 System Environment.
CR4.PSE	CFLG.pse	36	Page Size Extensions: In the Intel Itanium System Environment, this bit is ignored by IA-32 or Intel Itanium references. In the IA-32 System Environment, this bit enables 4M-byte page extensions for IA-32 paging. Modification of this bit by Itanium architecture-based code does have any side effects such as flushing the TLBs.
CR4.PAE	CFLG.pae	37	Physical Address Extensions: In the IA-32 System Environment, this bit enables IA-32 Physical Address Extensions for IA-32 paging This bit is ignored in the Intel Itanium System Environment. Modification of this bit by Itanium architecture-based code does have any side effects such as flushing the TLBs.
CR4.MCE	CFLG.mce	38	Machine Check Enable: This bit is ignored in the Intel Itanium System Environment. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the IA-32 System Environment.
CR4.PGE	CFLG.pge	39	Paging Global Enable: This bit is ignored in the Intel Itanium System Environment. This bit is provided as storage for compatibility purposes. This bit is supported as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the IA-32 System Environment, where this bit enables global pages for the IA-32 paging. Modification of this bit by Itanium architecture-based code does have any side effects such as flushing the TLBs.

Table 10-4. IA-32 Control Register Field Definition (Continued)

Field	Intel® Itanium® State	Bits	Description
CR4.PCE	CFLG.pce	40	Performance Counter Enable: IA-32 RDPMC user level reads of the performance counters are enabled when CR4.pce is 1. Otherwise execution of the RDPMC instruction results in a GPFault. CFLG.pce is ignored by Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on these bits.
CR4.FXSR	CFLG.FXSR	41	SSE FXSR Enable. When 1, enables the SSE register context. When 0, execution of all SSE instructions results in an IA_32_Intercept(Instruction) fault. This bit does not control the behavior of Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on these bits.
CR4.MMXEX	CFLG.MMXEX	42	SSE Exception Enable: When 1, enables SSE unmasked exceptions. When 0, all SSE Exceptions are masked. This bit does not control the behavior of Intel Itanium instructions. This bit is supported in both the IA-32 and Intel Itanium System Environments. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for details on these bits.
reserved		43:63	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault) and return the last value written on reads.

10.3.3.2 IA-32 Debug Registers

Within the Itanium System Environment, the IA-32 debug registers (DR0 - DR7) are superseded by the Itanium debug registers DBR0-7 and IBR0-7, see Section 10.8.1, "Data Breakpoint Register Matching" on page 2:274 for details. Accesses to the IA-32 debug registers result in an interception fault.

The Itanium debug registers are designed to facilitate debugging of both IA-32 and Itanium architecture-based code. Specifically, instruction and data breakpoints can be programmed by loading 64-bit virtual addresses into IBR and DBR along with an address mask. Itanium defined single stepping mechanisms, and taken branch traps are also defined to trap on IA-32 instructions. See Section 10.8.1, "Data Breakpoint Register Matching" on page 2:274 for details on IA-32 instruction set behavior with respect to the debug facilities defined by the Itanium architecture.

10.3.3.3 IA-32 Memory Type Range Registers (MTRRs)

Within the Itanium System Environment, IA-32 MTRR registers are superseded by physical memory attributes supplied by the TLB, as defined in Section 4.4.3, "Cacheability and Coherency Attribute" on page 2:77. IA-32 instruction references to the MTRRs in the MSR register space results in an instruction intercept fault.

10.3.3.4 IA-32 Model Specific and Test Registers

Within the Itanium System Environment, the IA-32 Model Specific Register space (MSRs) are superseded by the PAL firmware interface. Cache testing, initialization, processor configuration should be performed through the PAL interface. See Section 11.10, "PAL Procedures" on page 2:353 for a complete definition of the PAL functions and interfaces. Accesses to the IA-32 Model Specific Register space result in an instruction interception fault.

10.3.3.5 IA-32 Performance Monitor Registers

Within the Itanium System Environment, the Itanium performance monitors are designed to measure IA-32 and Itanium instructions, and system performance through a unified performance monitoring facility. Itanium architecture-based code can program the performance monitors for IA-32 and/or Itanium events by configuring the PMC registers. Count values are accumulated in the PMD registers for both IA-32 and Itanium events. See implementation-specific documentation for the list of supported events and encodings.

IA-32 code can sample the performance counters by issuing the RDPMC instruction. RDPMC returns count values from the specified Itanium performance monitor. Operating systems can secure the monitors from being read by IA-32 code by setting PSR.sp to 1, or setting CR4.pce to 0, or setting the performance monitor's pm-bit. Reads of a secured counter by RDPMC return a IA_32_Exception(GPFault(0)). IA-32 code cannot write or configure the performance monitors, all writes to the MSR register space are intercepted.

10.3.3.6 IA-32 Machine Check Registers

Within the Itanium System Environment, IA-32 machine check registers are superseded by the Itanium machine check architecture. See Section 11.3, "Machine Checks" on page 2:296 for details. IA-32 accesses to the Pentium III Processor machine check registers results in an instruction intercept.

10.4 Register Context Switch Guidelines for IA-32 Code

The following section gives operating system performance guidelines to minimize the amount of register context that must be saved and restored for IA-32 processes during a context switch.

10.4.1 Entering IA-32 Processes

High FP registers (FR32-127) – The processor requires access to all high FP registers during the execution of IA-32 instructions. It is recommended on entering an IA-32 process, that the OS save the high FP registers belonging to a prior context and then **enable** the high FP registers (PSR.dfh is 0). Otherwise, the processor will immediately raise a Disabled FP Register fault on the first IA-32 instruction executed in the IA-32 process. Performing the state save of the prior high FP register context during the context switch avoids the unnecessary generation of the Disabled FP Register fault.

Low FP registers (FR2-31) – The processor does not require access to the low FP registers unless executing IA-32 FP, MMX technology or SSE instructions. It is recommended on entry to an IA-32 process, that the OS **disable** the low FP registers by setting PSR.dfl to 1. PSR.dfl set to 0 indicates that there was a possibility that IA-32 FP, MMX technology or SSE instruction could execute and write FR8-31. If the low FP registers are enabled on entry to an IA-32 process (PSR.dfl is 0), all low FP registers will appear to be dirty on IA-32 process exit.

High Integer Registers (GR32-127) – Since the processor leaves all high registers in the register stack in an undefined state, these registers must be saved by the RSE before entering an IA-32 process.

Low Integer registers (GR1-31) – These registers must be explicitly saved before entering an IA-32 process.

10.4.2 Exiting IA-32 Processes

High FP registers (FR32-127) – PSR.mfh is unmodified when leaving the IA-32 instruction set. IA-32 instruction set execution leaves FR32-127 in an undefined state. Software can not rely on register values being preserved across an instruction set transition. These registers do NOT need to be preserved across a context switch.

Low FP registers (FR2-31) – PSR.mfl indicates there is a possibility that FR8-31 were modified by IA-32 FP, MMX technology, or SSE instruction. The modify bit is set by the processor when leaving the IA-32 instruction set, if PSR.dfl is 0, otherwise PSR.mfl is unmodified. During the state save of the outbound IA-32 process, it is recommended that the OS save FR2-31 if and only if the lower FP registers are marked as modified.

High Integer Registers (GR32-127) – Since the processor leaves all high registers undefined across an instruction set transition, these registers do NOT need to be preserved across an IA-32 context switch.

Low Integer registers (GR1-31) – These registers must be explicitly preserved across a context switch.

10.5 IA-32 Instruction Set Behavior Summary

Table 10-5 summarizes IA-32 instruction behavior within the Itanium System Environment. All IA-32 instructions are unchanged from the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** except where noted. IA-32 instructions can also generate additional Itanium register and memory faults as defined in

Table 5-6. Please refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the behavior of all IA-32 instructions in the IA-32 System Environment.

For all listed and unlisted IA-32 instructions in Table 10-5 the following relationships hold:

- Writes of any IA-32 general purpose, floating-point or MMX technology or SSE registers by IA-32 instructions are reflected in the Itanium registers defined to hold that IA-32 state when the IA-32 instruction set completes execution.
- Reads of any IA-32 general purpose, floating-point or MMX technology or SSE registers by IA-32 instructions see the state of the Itanium registers defined to hold the IA-32 state after entering the IA-32 instruction set.
- IA-32 numeric instructions are controlled by and reflect their status in FCW, FSW, FTW, FCS, FIP, FOP, FDS and FEA. On exit from the IA-32 instruction set, Itanium registers defined to hold IA-32 state reflect the results of all IA-32 prior numeric instructions (FSR, FCR, FIR, FDR). Itanium numeric status and control resources defined to hold IA-32 state are honored by IA-32 numeric instructions when entering the IA-32 instruction set.

In Table 10-5 *unchanged* indicates there is no change in behavior with respect to the IA-32 System Environment.

Table 10-5. IA-32 Instruction Summary

IA-32 Instruction	Intel® Itanium® System Environment	Comments
AAA, AAD. AAM, AAS	unchanged	
ADC, ADD, AND,		
ADDPS, ADDSS, ANDNPS, ANDPS		
ARPL		
BOUND		
BSF, BSR		
BSWAP		
BT, BTC, BTS, BTR		
CALL	near: no change far: no change gate more privilege: Gate Intercept gate same privilege: Gate Intercept task gate: Gate Intercept + additional taken branch trap	Intercept if through a call or task gate If PSR.tb is 1, raise a taken branch trap.
CBW, CWDE, CDQ	unchanged	
CLC, CLD		
CLI	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1
CLTS	Instruction Intercept	IA-32 privileged instruction
CMC	unchanged	
CMOV		
CMP		
CMPPS, CMPSS, COMISS		
CMPS		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium® System Environment	Comments
CMPXCHG, 8B	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required
CPUID	unchanged	
CWD, CDQ		
CVTPI2PS, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2PI, CVTTSS2SI		
DAA, DAS		
DEC		
DIV		
DIVPS, DIVSS		
ENTER		
EMMS		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium® System Environment	Comments
F2XM1	unchanged	IA-32 numeric instructions manipulate the IA-32 numeric register stack contained in f8-f15, status is reflected in FSR. Modification of the IA-32 numeric environment changes FIR, FDR FCR and FSR.
FABS		
FADD, FADDP, FIADD		
FBLD		
FBSTP		
FCHS		
FCLEX, FNCLEX		
FCMOV		
FCOM, FCOMPP		
FCOMI, FCOMIP		
FUCOMI, FUCOMIP		
FCOS		
FDECSTP		
FDIV, FDIVP, FIDIV		
FDIVR, FDIVRP, FDIVR		
FFREE		
FICOM, FICOMP		
FILD		
FINCSTP		
FINIT, FNINIT		
FIST, FISTP		
FLD		
FLD constant		
FLDCW		
FLDENV		
FMUL, FMULP, FIMUL		
FNOP		
FPATAN, FPTAN		
FPREM, FPREM1		
FRNDINT		
FRSTOR		
FSAVE, FNSAVE		
FSCALE		
FSIN, FSINCOS		
FSQRT		
FST, FSTP		
FSTCW, FNSTCW		
FSTENV, FNSTENV		
FSTSW, FNSTSW		
FSUB, FSUBP, FISUB		
FSUBR, FSUBRP, FISUBR		
FTST		
FUCOM, FUCOMP		
FWAIT		
FXAM		
FXCH		
FXTRACT		
FXRSTOR, FXSAVE		
FYL2X, FYL2XP1		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium® System Environment	Comments
HLT	Instruction Intercept	IA-32 privileged instruction
IDIV	unchanged	
IMUL		
IN, INS	unchanged + I/O ports are mapped virtually	If CFLG.io is 0, the TSS I/O permission bitmap is not consulted. Intel Itanium TLB faults control accessibility to I/O ports.
INC	unchanged	
INT 3, INTO	Mandatory Exception vector #	Delivered as an IA_32_Interrupt
INT n	Mandatory Interruption vector #	Delivered as an IA_32_Exception
INVD	Instruction Intercept	IA-32 privilege instruction
INVLPG		
IRET, IRETD	Real Mode: Instruction Intercept to VM86: Instruction Intercept from VM86: Instruction Intercept same privilege: Instruction Intercept less privilege: Instruction Intercept different task: Instruction Intercept	All forms of IRET result in an instruction intercept
Jcc	additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
JMP	near: no change far: no change gate task: Gate Intercept call gate: Gate Intercept additional taken branch trap	Intercept fault if through a call or task gate If PSR.tb is 1, raise a taken branch trap.
JMPE		Jumps to the Intel Itanium instruction set
LAHF	unchanged	
LAR		
LDMXCSR		
LDS, LES, LFS, LGS, LSS		
LEA		
LEAVE		
LGDT, LLDT	Instruction Intercept	IA-32 privileged register resource
LIDT		
LMSW		
Lock prefix	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required
LODS	unchanged	
LOOP, LOOPcc	additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
LSL	unchanged	User level instruction
LTR	Instruction Intercept	IA-32 privileged register
MASKMOVQ	unchanged	
MAXPS, MAXSS, MINPS, MINSS		
MOV		
MOVNTPS, MOVNTQ		

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium® System Environment	Comments
MOV from CR	unchanged	
MOV to CR	Instruction Intercept	IA-32 privileged system registers
MOV to/from DR		
Mov SS	System Flag Intercept Trap	System Flag Intercept Trap after instruction completes
MOVAPS, MOVHPS, MOVLPS. MOVMSKPS, MOVSS, MOVUPS	unchanged	
MOVD, MOVQ		
MOVS		
MOVSX, MOVZX		
MUL		
MULPS, MULSS		
NEG		
NOP		
NOT		
OR		
ORPS		
OUT, OUTS	unchanged + I/O ports are mapped virtually	If CFLG.io is 0, the TSS I/O permission bitmap is not consulted. Intel Itanium TLB faults control accessibility to I/O ports.
PACKSS, PACKUS	unchanged	
PADD, PADDS, PADDUS		
PAND, PANDN		
PCMPEQ, PCMPGT		
PEXTRW, PINSRW		
PMADD		
PMULHW, PMULLW, PMULHUW		
PMOVMSKB		
POP, POPA		
POP SS	System Flag Intercept	System Flag Intercept Trap after instruction completes
POPF, POPFD	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1 Intercept if EFLAG.ac, or tf change state.
POR	unchanged	
PREFETCH		
PSHUFW		
PSLL, PSRA, PSRL		
PSUB, PSUBS, PSUBUS		
PUNPCKH, PUNPCKL		
PXOR		
PUSH, PUSA	unchanged	
PUSHF, PUSHFD		Pushes value in EFLAG, no intercept
RCL, RCR, ROL, ROR		
RCPPS, RSQRTPS		
RDMSR	Instruction Intercept	IA-32 privileged system register space
RDTSC	Optional GPFault	No longer faults in VM86, GPFault if secured by PSR.si or CFLG.tsd.
RDPMC		
REP, REPcc prefix	unchanged	

Table 10-5. IA-32 Instruction Summary (Continued)

IA-32 Instruction	Intel® Itanium® System Environment	Comments
RET	near: no change far: no change less privilege: no change same privilege: no change + additional taken branch trap	If PSR.tb is 1, raise a taken branch trap.
RSM	Instruction Intercept	IA-32 privileged instruction
SAHF	unchanged	
SAL, SAR, SHL, SHR		
SBB		
SCAS		
SFENCE		
SETcc		
SGDT, SLDT	Instruction Intercept	IA-32 privileged instruction
SHLD, SHRD	unchanged	
SHUFPS, SQRTPS, SQRTSS		
SIDT	Instruction Intercept	IA-32 privileged instructions
SMSW		
STC, STD	unchanged	
STI	Optional System Flag Intercept	Intercept if EFLAG.if changes state and CFLG.ii is 1
STMXCSR	unchanged	
STOS		
STR	Instruction Intercept	IA-32 privileged instruction
SUB	unchanged	
SUBPS, SUBSS		
SYSENTER, SYSEXIT	Instruction Intercept	
TEST	unchanged	
UCOMISS		
UNPCKHPS, UNPCKLPS		
UD2	Instruction Intercept	Reserved undefined opcodes
VERR, VERW	unchanged	User level instruction
WAIT		
WBINVD	Instruction Intercept	IA-32 privileged instructions
WRMSR		
XADD	Optional Lock Intercept	If Locks are disabled (DCR.lc is 1) and a processor external lock transaction is required than a Lock Intercept.
XCHG		
XLAT, XLATB	unchanged	
XOR		
XORPS		

10.6 System Memory Model

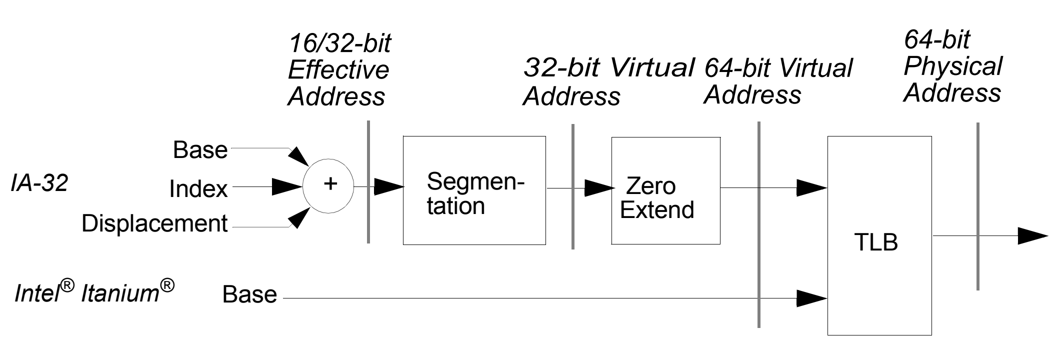
Within the Itanium System Environment, a unified memory model is presented to the programmer. Applications and the operating system see the same 64-bit virtual memory space and virtual addressing mechanisms regardless of the referencing instruction set. A virtual address points to the same physical storage location from both IA-32 and Itanium instruction sets.

Itanium architecture-based operating systems must not use IA-32 segmentation as a protected system resource. An Itanium architecture-based operating system must use virtual memory management defined by the Itanium architecture to secure IA-32 and Itanium architecture-based applications, memory and I/O devices. The Itanium architecture is defined to be *unsegmented architecture and all Itanium memory references bypass IA-32 segmentation and protection checks*. In addition, Itanium architecture-based user level code can directly modify IA-32 segment selector and descriptor values for all segments (including GDT and LDT). If operating systems do not rely on segmentation for protection, there are no security concerns for exposing IA-32 segment registers and descriptors to Itanium architecture-based user level applications

IA-32 instruction and data reference addresses are generated as 16/32-bit effective addresses as shown in Figure 10-2. IA-32 segmentation is then applied to map 32-bit effective addresses into 32-bit virtual addresses, the processor then converts the address into a 64-bit virtual address by zero extension from 32 to 64-bits. Itanium instructions bypass all of these steps and directly generate addresses within the 64-bit virtual address space.

For both IA-32 and Itanium instruction set memory references, virtual memory management defined by the Itanium architecture is used to map a given virtual address into a physical address. Itanium architecture-based virtual memory management hardware does not distinguish between Itanium and IA-32 instruction set generated memory references during the conversion from a virtual to physical address.

Figure 10-4. Virtual Memory Addressing



10.6.1 Virtual Memory References

In the Itanium System Environment the following virtual memory options are available for supporting IA-32 and Itanium memory references.

- Software TLB fills (TLBs are enabled, but the VHPT is disabled).
- 8-byte short format VHPT, see Section 4.1.5, "Virtual Hash Page Table (VHPT)" on page 2:61 for details.
- 32-byte long format VHPT.

Itanium virtual memory resources can be used by the operating system for all IA-32 memory references. These resources include virtual Region Registers (RR), Protection Key Registers (PKR), the Virtual Hash Page Table (VHPT), all supported range of page sizes, Translation Registers (ITR, DTR), the Translation Cache (ITC, DTC) and the complete set of Itanium virtual memory management faults defined in Chapter 5.

10.6.2 IA-32 Virtual Memory References

By definition, IA-32 instruction and data memory references are confined to 32-bits of virtual addressing, the first 4 G-bytes of virtual region 0. However, IA-32 memory references can be mapped anywhere within the implemented physical address space by operating system code.

Virtual addresses are converted into physical addresses through the process defined in Section 4.1, "Virtual Addressing" on page 2:45. IA-32 references use the Itanium TLB resources as follows.

- **Region Identifiers** – Operating systems can place IA-32 processes within virtual region 0, and use the entire 2^{24} region identifier name space. By using region identifiers there is no requirement to flush IA-32 mappings on a context switch.
- **Protection Keys** – Operating systems can place mappings used by IA-32 processes within any number of protection domains. If PSR.pk is 1, all IA-32 references search the Protection Key Registers (PKR) for matching keys. If a key is not found, a Key Miss fault is generated. Otherwise, key read, write, execute permissions are verified.
- **TLB Access Bit** – If this bit is zero, an Access Bit fault is generated during Itanium or IA-32 instruction set memory references. Note: the processor does not automatically set the Access bit in the VHPT on every reference to the page. Access bit updates are controlled by the operating system.
- **TLB Dirty Bit** – If this bit is zero, a Dirty bit fault is generated during any Itanium or IA-32 instruction that stores to a dirty page. Note: the processor does not automatically set the Dirty bit in the VHPT on every write. Dirty bit updates are managed by the operating system.

10.6.3 IA-32 TLB Forward Progress Requirements

To ensure forward progress while executing IA-32 instructions, additional TLB resources and replacement policies must be defined over and above the definition given in Section 4.1.1.2, "Translation Cache (TC)" on page 2:49. IA-32 instructions and data accesses may not be aligned resulting in a worst case scenario for two possible pages being referenced for every memory datum referenced during the execution of an IA-32 instruction. Furthermore, the worst case non-intercepted IA-32 opcode can reference up to 4 independent data pages.

The Translation Cache's (TC) are required to have the following minimum set of resources to ensure forward progress. Given that software TLB fills can be used to insert entries into the TLB and a hardware page table walker is not necessarily used, the following requirements must be satisfied by the processor:

- Instruction Translation Cache – at least 1 way set associative with 2 sets, or 2 entries in a fully associative design. Replacement algorithms must not consistently displace the last 2 entries installed by software.
- Data Translation Cache – at least 4 way set associative with 2 sets, or 8 entries in a fully associative design. Replacement algorithms must not consistently displace the last 8 entries installed by software or the last 8 translations referenced by an IA-32 instruction.

- Unified Translation Cache – at least 5 way set associative with 2 sets, or 10 entries in a fully associative design. The processor must not consistently displace the last 10 entries installed or the last 10 translations referenced by an IA-32 instruction.

The processor must ensure that the minimum number of entries can co-exist in the TLB, and TC replacement algorithms allow software insertion of the required entries such that the required number of translations can be co-resident in the TLB.

The processor cannot ensure forward progress unless translations mapping the Itanium architecture-based TLB Miss handlers are statically mapped by the Instruction Translation Registers.

10.6.4 Multiprocessor TLB Coherency

Global TLB purges can not occur on another processor unless that processor is at an interruptible point. For IA-32 instruction set execution, interruptible points are defined as; 1) when the processor is between instructions (regardless of the state of PSR.i and EFLAG.if), and 2) each iteration of an IA-32 string instruction, regardless of the state of PSR.i and EFLAG.if

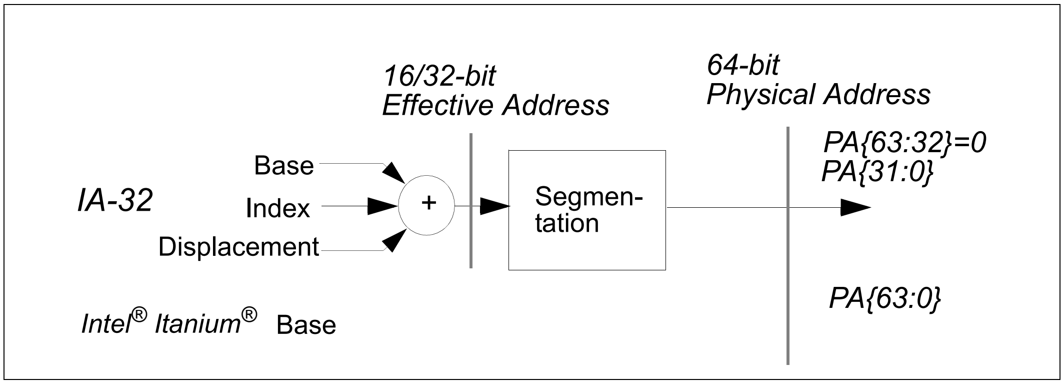
The processor may delay in its response and acknowledgment to a broadcast purge TC transaction until the processor executing an IA-32 instruction has reached a point (e.g. an IA-32 instruction boundary) where it is safe to process the purge TC request. The amount of the delay is implementation specific and can vary depending on the receiving processor and what instructions or operations are executing when it receives the purge request.

10.6.5 IA-32 Physical Memory References

When running IA-32 code, virtual addressing must be utilized by setting PSR.dt to 1 and PSR.it to 1, otherwise processor operation is undefined. Undefined behavior can include, but is not limited to: machine check abort on entry to IA-32 code, and unpredictable behavior for IA-32 self modifying code.

Operating systems must ensure PSR.dt and PSR.it are 1 before invoking IA-32 code. From a practical standpoint, the TLBs must be enabled so IA-32 code can access the virtual address space, and access memory areas other than WB (e.g. UC or the I/O port space).

Figure 10-5. Physical Memory Addressing



10.6.6 Supervisor Accesses

If the processor is operating in the Itanium System Environment, supervisor override is disabled, and LDT, GDT, TSS references are performed at the privilege level specified by PSR.cpl. Unaligned processor references to LDT, GDT, and TSS segments will never generate an EFLAG.ac enabled IA-32 Exception (AlignmentCheck) fault, even if PSR.cpl equals 3 and supervisor override is disabled.

Operating systems must ensure that the GDT/LDT are mapped to pages with user level read/write access.

Write permission is required if GDT, or LDT memory descriptor Access-bits are zero regardless of supervisor override conditions. If all GDT/LDT descriptor Access-bits are one, write permission can be removed. Otherwise, Access Rights, Key Miss or Key Miss faults can be generated during all segment descriptor referencing instructions.

If a fault is generated during a supervisory access, the ISR.so bit indicates that CPL is zero or a supervisor override condition was in effect (reference as made to GDT, LDT or TSS).

10.6.7 Memory Alignment

Depending on software conventions, memory structures may have different alignment or padding restrictions for the IA-32 and Itanium instruction sets. IA-32 and Itanium architecture-based software should use aligned memory operands as much as possible to avoid possible severe performance degradation associated with un-aligned values and extra over-head for unaligned data memory fault handlers.

The processor provides full functional support for all cases of un-aligned IA-32 data memory references. If PSR.ac is 1 or EFLAG.ac is 1 and CR0.am is 1and the effective privilege level is 3, unaligned IA-32 memory references result in an IA-32 Exception (AlignmentCheck) fault. Unaligned processor references to LDT, GDT, and TSS segments will never generate an EFLAG.ac enabled IA-32 Exception (AlignmentCheck) fault, even if the effective privilege level is 3 and supervisor override is disabled.

Alignment conditions for Itanium memory references are not affected by the EFLAG.ac, CFLG.am bits.

If EFLAG.ac and CFLG.am are 1 and the reference is done at privilege level 3, IA-32 instruction set unaligned conditions are; 2-byte references not a 2-byte boundary, 4-byte references not on a 4-byte boundary, 8-byte references not on a 8-byte boundary, and 10-byte references not on a 8-byte boundary.

If PSR.ac is 1, IA-32 instruction set unaligned conditions are; 2-byte references not a 2-byte boundary, 4-byte references not on a 4-byte boundary, 8-byte references not on a 8-byte boundary, and 10-byte references not on a **16**-byte boundary.

The processor exhibits the following behavior when accesses are made to un-aligned data operands that span virtual page boundaries:

- IA-32 instruction set – If either page contains a fault, no memory location is modified. For reads, the destination register is not modified.
- Itanium instruction set – All page crossers result in an unaligned reference fault. Memory contents and register contents are not modified.

10.6.8 Atomic Operations

All Itanium load/stores and IA-32 non-locked memory references up to 64-bits that are aligned to their natural data boundaries are atomic.

Both IA-32 and Itanium atomic semaphore operations can be performed on the same shared memory location. The processor ensures IA-32 locked read-modify-write opcodes and Itanium semaphore operations are performed atomically even if the operations are initiated from the other instruction set by the same processors, or between multiple processors in an multiprocessing system.

There are some restrictions placed on Itanium atomic operations that may prevent Itanium architecture-based code from manipulating IA-32 semaphores in some rare cases:

- Unaligned Itanium semaphore operations result in an Unaligned Data Reference fault. Itanium architecture-based code manipulation of an IA-32 semaphore can only be performed if the IA-32 semaphore is aligned.
- Itanium semaphore operations to memory which is neither write-back cacheable nor a NaTPage result in an Unsupported Data Reference fault (regardless of the state of the DCR.lc). Itanium architecture-based code manipulation of an IA-32 semaphore can only be performed if the IA-32 semaphore is allocated in aligned write-back cacheable memory.

If an IA-32 locked atomic operation is defined as requiring a read-modify-write operation external to the processor under external bus lock and if DCR.lc is set to 1, an IA_32_Intercept(Lock) fault is generated. (IA-32 atomic memory references are defined to require an external bus lock for atomicity when the memory transaction is made to non-write-back memory or are unaligned across an implementation-specific non-supported alignment boundary.) If DCR.lc is set to 0, the processor may either execute the transaction as a series of non-atomic transactions or perform the transaction with an external bus lock, depending on the processor implementation. For processor implementations that do support external bus locks, software must ensure that the Bus Lock Mask bit is set to one, in order to ensure atomicity of these IA-32 operations when DCR.lc=0. The Bus Lock Mask bit is a feature controllable by the PAL_BUS_SET_FEATURES procedure. (See Table 11-63 on page 2:368 for more information).

If the processor supports external bus locks, unaligned IA-32 atomic references are supported, but their usage is strongly discouraged since they are typically performed outside the processor's cache which can severely degrade performance of the system. IA-32 external bus locks are not supported on all processor implementations.

For IA-32 semaphores, atomicity to uncached memory areas (UC) is platform specific, atomicity can only be ensured by the platform design and can not be enforced by the processor.

10.6.9 Multiprocessor Instruction Cache Coherency

The processor and platform ensure the processor’s instruction cache is coherent for the following conditions:

- For all processors in the coherence domain, local and remote instruction cache coherency on all processors is enforced for any store generated by any processor running the IA-32 instruction set.
- For all processors in the coherence domain, instruction cache coherency on all processors is enforced for all coherent I/O traffic. (For non-coherent I/O, a processor may or may not see the results of an I/O operation.)
- For all processors in the coherence domain, instruction cache coherency is not enforced for stores generated by any processor running the Itanium instruction set. To ensure instruction cache coherency, Itanium architecture-based code must use the code sequence defined in Section 4.4.6.2, “Memory Consistency” on page 1:72.

Table 10-6. Instruction Cache Coherency Rules

Originating Instruction Set	Local Processor	External Processor	Coherent, I/O	Non-Coherent I/O
IA-32	Coherent	Coherent	Coherent	Maybe Non-Coherent
Intel Itanium	May be Non-coherent	May be Non-coherent		

10.6.10 IA-32 Memory Ordering

IA-32 memory ordering follows the Pentium III defined ***processor ordered*** model for cacheable and uncacheable memory. IA-32 *processor ordered* memory references are mapped onto the Itanium memory ordering model as follows:

- All IA-32 stores have ***release*** semantics. Except for IA-32 stores to write-coalescing memory that are unordered. Subsequent loads are allowed to bypass buffered local store data before it is globally visible. The amount of store buffering is model specific and can vary across processor generations.
- All IA-32 loads have ***acquire*** semantics. Some high performance processor implementations may speculatively issue *acquire* loads into the memory system for speculative memory types, if and only if the loads do not *appear* to pass other loads as observed by the program. If there is a coherency action that would result in the appearance to the program of a load bypassing other load, the processor will reissue the load operation(s) in program order.
- All IA-32 read-modify-write or locked instructions have memory ***fence*** semantics. All buffered stores are flushed.
- IA-32 IN, OUT and serializing operations (as defined in the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual***) have memory ***fence*** semantics. In addition, the processor will wait for completion (acceptance by the platform) of the IN or OUT before executing the next instruction. All buffered stores are flushed before the IN or OUT operation.
- IA-32 SFENCE has ***release*** semantics and will flush all buffered stores.

Table 10-7. IA-32 Load/Store Sequentiality and Ordering

IA-32 Memory Reference	Uncacheable	Write Coalescing	Cacheable
store	sequential release[a]	non-sequential unordered	non-sequential release[b]
load	sequential acquire[a]	non-sequential unordered	non-sequential acquire[b]

Table 10-7. IA-32 Load/Store Sequentiality and Ordering (Continued)

IA-32 Memory Reference	Uncacheable	Write Coalescing	Cacheable
locked or read-modify-write operation	sequential fence flush prior stores	non-sequential fence flush prior stores	non-sequential fence flush prior stores
IN, INS, OUT, OUTS	sequential fence flush prior stores	undefined	undefined
IA-32 Serialization	fence, flush prior stores		
SFENCE	release, flush prior stores		

a. However, IA-32 loads/stores to uncacheable memory flush the write coalescing buffers.

b. However, IA-32 load/stores to cacheable memory do not flush the write coalescing buffers.

Per Table 10-7, IA-32 memory references can be expressed in terms of acquire, release, fence and sequential ordering rules defined by the Itanium architecture. IA-32 data memory references follow the same ordering relationships as defined for Itanium architecture-based code as defined in Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82. The following additional clarifications need to be made for IA-32 instruction set execution:

- IA-32 loads and instruction fetches to speculative memory can occur randomly. Read accesses to speculative memory can occur at arbitrary times even if the in-order execution of the program does not require a read or instruction fetch from a given memory location.
- IA-32 instruction fetches and loads to non-speculative memory occur in program order. IA-32 instruction cache line fetch accesses to uncached memory occur in the order specified by an in-order execution of the program. Note however that the same cache line may be fetched multiple times by the processor as multiple instructions within the cache line are executed. The size of a cache line and number of instruction fetches is model specific.
- IA-32 instruction fetches are not perceived as passing prior IA-32 stores. IA-32 stores into the IA-32 instruction stream are observed by the processor's self modifying code logic. Speculative instruction fetches may be emitted by the processor before a store is seen to the instruction stream and then discarded. Self modifying code due to Itanium stores is not detected by the processor.
- IA-32 instruction fetches can pass prior loads or memory fence operations from the same processor. Data memory accesses, and memory fences are not ordered with respect to IA-32 instruction fetches.
- IA-32 instruction fetches can not pass any serializing instructions, including Itanium `srlz.i` and IA-32 CPUID. For speculative memory types the processor may prefetch ahead of a serialization operation and then discard the prefetched instructions.
- IA-32 serializing operations wait for all previous stores and loads to complete, and for all prior stores buffered by the processor to become visible. IA-32 serializing instructions include CPUID.
- IA-32 OUT instructions may be buffered, however the processor will not start execution of the next IA-32 instruction until the OUT has completed (been accepted by the platform).
- The processor does not begin execution of the next IA-32 instruction until the IN or OUT has been completed (accepted) by the platform. This statement does not apply

for Itanium memory references to the I/O port space. The processor may issue instruction fetches and VHPT walks ahead of an IN or OUT.

- VHPT Walks are speculative and can occur at any time. VHPT walks can pass all prior IA-32 loads, stores, instruction fetches, I/O operations and serializing instructions.

10.6.10.1 Instruction Set Transitions

Instruction set transitions do not automatically fence memory data references. To ensure proper ordering software needs to take into account the following ordering rules.

10.6.10.1.1 Transitions from Intel® Itanium® Instruction Set to IA-32 Instruction Set

- All data dependencies are honored, IA-32 loads see the results of all prior Itanium and IA-32 stores.
- IA-32 stores (*release*) can not pass any prior Itanium load or store.
- IA-32 loads (*acquire*) can pass prior Itanium unordered loads or any prior Itanium store to a different address. Itanium architecture-based software can prevent IA-32 loads from passing prior Itanium loads and stores by issuing an *acquire* operation (or `mf`) before the instruction set transition.

10.6.10.1.2 Transitions from IA-32 Instruction Set to Intel® Itanium® Instruction Set

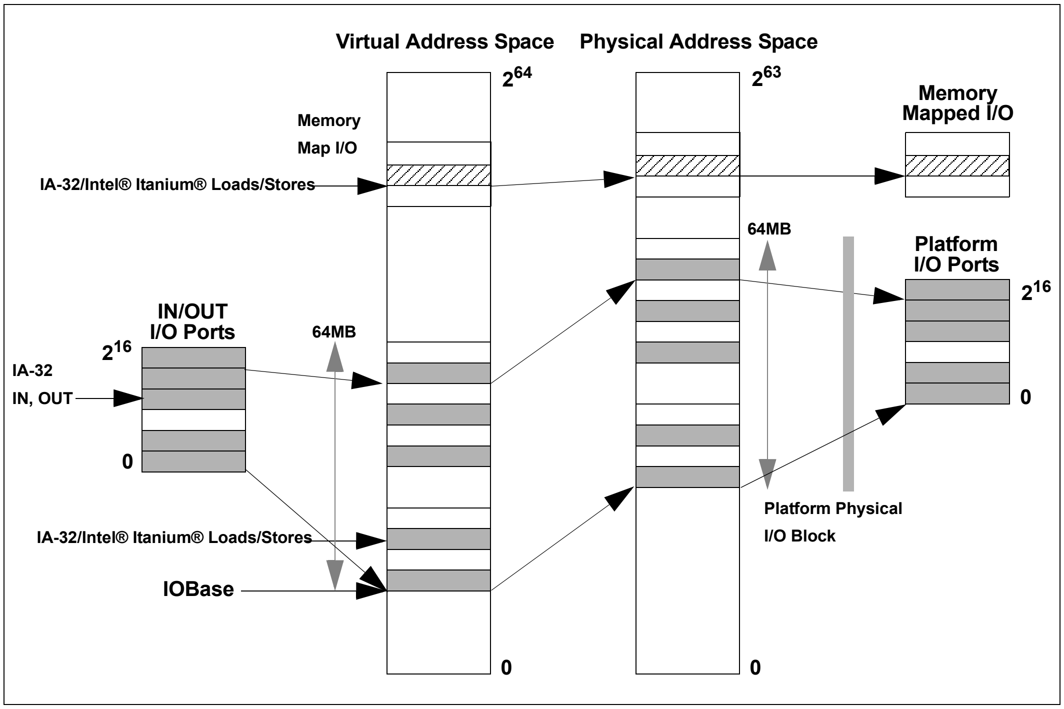
- All data dependencies are honored, Itanium loads see the results of all prior Itanium and IA-32 stores.
- Itanium stores or loads can not pass prior IA-32 loads (*acquire*).
- Itanium unordered stores or any Itanium load can pass prior IA-32 stores (*release*) to a different address. Itanium architecture-based software can prevent Itanium loads and stores from passing prior IA-32 stores by issuing a *release* operation (or `mf`) after the instruction set transition.

10.7 I/O Port Space Model

A consistent unified addressing model is used for both IA-32 and Itanium references to the I/O port space. On prior IA-32 processors two I/O models exist; memory mapped I/O and the 64KB I/O port space. On processors based on the Itanium instruction set, the 64KB I/O port space defined by IA-32 processors is effectively mapped into the 64-bit virtual address space of the processor, producing a single memory mapped I/O model as shown in Figure 10-1. This model allows Itanium normal load and store instructions to also access the I/O port space.

Itanium architecture-based operating system code can directly control IA-32 IN, OUT instruction and accessibility by IA-32 or Itanium load/store instructions to blocks of 4 virtual I/O ports using the TLBs. The entire range of virtual memory mechanisms defined by the Itanium architecture: access rights, dirty, access bits, protection keys, region identifiers can be used to control permission and addressability.

Figure 10-1. I/O Port Space Model



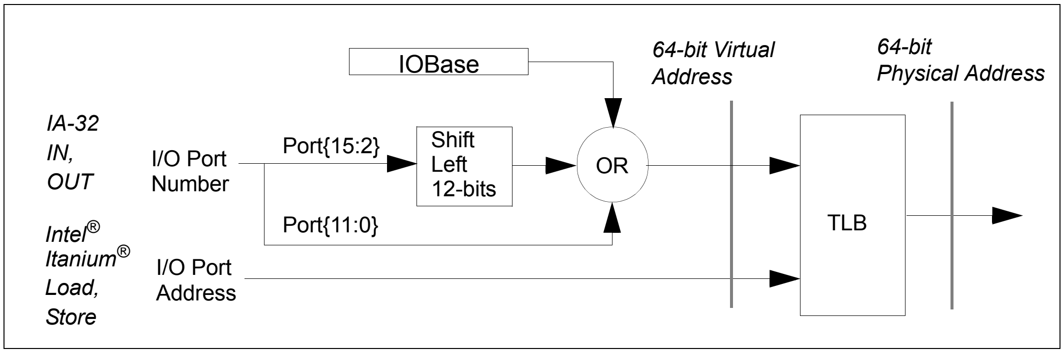
In the Itanium System Environment, the virtual location of the 64 MB I/O port space is determined by operating system. For IA-32 IN and OUT instructions, the operating system can specify the virtual base location via the I/O base register.

Any IA-32 or Itanium load or store within the virtual region mapped by the operating system to the platform's physical 64 MB I/O port block, directly accesses physical I/O devices within the I/O port space. The location of the 64 MB I/O port block within the 2^{63} byte physical address space is determined by platform conventions, see Section 10.7.2, "Physical I/O Port Addressing" on page 2:270 for details.

10.7.1 Virtual I/O Port Addressing

The IA-32 defined 64-KB I/O port space is expanded into 64 MB. This effectively places 4 I/O ports per each 4KB virtual and physical page. Since there are 4 ports per virtual page, the TLBs can be used port address translation, and permission checks as shown in Figure 10-2.

Figure 10-2. I/O Port Space Addressing



For IA-32 IN and OUT instructions a port's virtual address is computed as:

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

This address computation places 4 ports on each 4K page and expands the space to 64MB, with the ports being at a relative offset specified by port{11:0} within each 4K-byte virtual page. IOBase is a kernel register *(KR)* maintained by the operating system that points to the base of the 64MB Virtual I/O port space. *The value in IOBase must be aligned on a 64MB boundary otherwise port address aliasing will occur and processor operation is undefined.*

For Itanium load and stores accesses to the I/O port space, a port's virtual address can be computed in the same manner, specifically.

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

In practice this address is a constant for any given physical I/O device.

Note: In the generation of the I/O port virtual address, software MUST ensure that port_virtual_address{11:2} are equal to port{11:2} bits. Otherwise, some processors implementations may place the port data on the wrong bytes of the processor's bus and the port will not be correctly accessed.

IA-32 IN and OUT instructions and Itanium or IA-32 load/store instructions can reference I/O ports in 1, 2, or 4-byte transactions. References to the legacy I/O port space cannot be performed with greater than 4 byte transactions due to bus limitations in most systems. Since an IA-32 IN/OUT instruction can access up to 4 bytes at port address 0xFFFF, the I/O port space effectively extends 3 bytes beyond the 64KB boundary. Operating systems can; 1) not map the excess 3 bytes, resulting in denial of permission for the excess 3 bytes, or 2) map via the TLB the excess 3 bytes back to port address 0 effectively wrapping the I/O port space at 64KB.

Operating system code can map each virtual I/O port space page anywhere within the physical address space using the Data Translation Registers or the Data Translation Cache. Large page translations can be used to reduce the number of mappings required in the TLB to map the I/O port space. For example, one 64MB translation is sufficient to map the entire expanded 64MB I/O port space. The **UC memory attribute** must be used for all I/O port space mappings to avoid speculative processor references to I/O devices, otherwise processor and platform operation is undefined.

Operating System Warning: Operating system code can not remap a given port to another port address within the I/O port space, such that port_physical_address{21:12} != port_physical_address{11:2}. Otherwise, based on the processor model, I/O port data may be placed on the wrong bytes of the processor's bus and the port will not be correctly accessed.

I/O port space breakpoints can be configured by loading the address and mask fields with the virtual address defined by the operating system to correspond to the I/O port space.

The processor (as defined in the next section) ensures that load, store references will not result in references to I/O devices for which permission was not granted.

All memory related faults defined in Chapter 5, "Interruptions" can be generated by IA-32 IN and OUT references to the I/O port space, including IA_32_Exception(Debug) traps for data address breakpoints and IA_32_Exception(AlignmentCheck) for unaligned references. (EFLAG.ac enabled unaligned port references are not detected by the processor). Itanium Data Breakpoint registers (DBRs) can be configured to generate debug traps for references into the I/O port space by either IA-32 IN/OUT instructions or by IA-32 or Itanium load/store instructions.

10.7.2 Physical I/O Port Addressing

Some processors implementations will provide an M/IO pin or bus indication by decoding physical addresses if references are within the 64MB physical I/O block. If so the 64MB I/O port space is compressed back to 64KB. Subsequent processor implementations may drop the M/IO pin (or bus indication) and rely on platform or chip-set decoding of a range of the 64MB physical address space.

Through the PAL firmware interface, the 64MB physical I/O block can be programmed to any arbitrary physical location. It is suggested that to be compatible with IA-32 based platforms, the platform physical location of the physical I/O block be programmed above 4G-bytes and above all useful DRAM, ROM and existing memory mapped I/O areas. See PAL_PLATFORM_ADDR on page 2:442 for details.

Based on the platform design, some platforms can accept addresses for the expanded 64MB I/O block, while other platforms will require that the I/O port space be compressed back to 64KB by the processor. If the I/O port space needs to be compressed either the processor or platform (based on the implementation) will perform the following operation for all memory references within the physical I/O block.

```
IO_address{1:0} = PA{1:0}
IO_address{15:2} = PA{25:12}// byte strobes are generated
                             // from the lower I/O_address bits
```

The processor ensures that the bus byte strobes and bus port address are derived from PA{25:12,1:0}. Thus allowing an operating system to control security of each 4 ports via TLB management of PA{25:12}.

10.7.2.1 I/O Port Addressing Restrictions

For the 64MB physical I/O port block the following operations are undefined and may result in unpredictable processor operation; references larger than 4-bytes, instruction fetch references, references to any memory attribute other than UC, or semaphore references which require an atomic lock. To ensure I/O ports accesses are not granted for which the TLB has not been consulted, the processor ensures:

- All byte addresses within the same 4KB page alias to the 4 ports defined by the mapped physical I/O port page.
- All IA-32 and Itanium unaligned loads and stores that cross a 4-byte boundary to the processor's physical I/O port block are truncated. That is the bus transaction to the area before the 4-byte boundary is performed (the number of bytes emitted is model specific). No bus transaction is performed for the bytes that are beyond the 4-byte boundary. 4-byte crosser loads while return some undefined data. 4-byte crosser stores will not write all intended bytes.
- For IA-32 IN/OUT accesses that cross a 4-port boundary the processor will break the operation into smaller 1, 2, or 3 byte I/O port transactions within each 4KB virtual page.

10.7.3 IA-32 IN/OUT instructions

IA-32 I/O instructions (IN, OUT, INS, OUTS) defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** are augmented as follows:

- I/O instructions first check for IOPL permission. If PSR.cpl<=EFLAG.iopl, access permission is granted. Otherwise the TSS I/O permission bitmap may be consulted as defined below. If the Bitmap denies permission or is not consulted an IA_32_Exception(GPFault) is generated.
- If IOPL permission is denied and CFLG.io is 1, the TSS I/O permission bitmap is consulted for access permission. If the corresponding bit(s) for the I/O port(s) is 1, indicating permission is denied, a GPFault is generated. Otherwise access permission is granted. The TSS I/O permission bitmap provides 1 port permission control at the expense of additional processor data memory references. This mechanism can be used in the Itanium System Environment, but is not recommended since TLB access controls defined by the Itanium architecture are faster and provide a consistent control mechanism for both IA-32 and Itanium architecture-based code. Whereas, the TLB mechanism provides a control mechanism for both IA-32 and Itanium memory references.
- If CFLG.io is 0, the TSS I/O permission bitmap is not consulted and if the IOPL check failed an IA_32_Exception(GPFault) is generated. By setting CFLG.io to 0, operating system code can disable all processor references to the TSS. By setting IOPL<PSR.cpl and setting CFLG.io to 0, operating system code can block all user level execution of IA-32 I/O instructions, no TSS needs to be allocated or defined by the operating system.
- I/O port references generate a virtual port address relative to the IOBase register as defined in Section 10.7.1, "Virtual I/O Port Addressing" on page 2:268.
- If data translations are enabled, the TLB is consulted for the required virtual to physical mapping. If the required mapping is not present a VHPT Translation, Data TLB Miss or Alternative Data TLB Miss fault is generated.
- If data translations are enabled, Access Rights, Permission Keys, Access, Dirty and Present bits are checked and faults generated.

- If data translations are disabled (PSR.dt is 0) or the referenced I/O port is mapped to an unimplemented virtual address (via the IOBase register), a GPFault is raised on the referencing IA-32 IN, OUT, INS, or OUTS instruction.
- Alignment and Data Address breakpoints are also checked and may result in an IA_32_Exception(AlignmentCheck) fault (if PSR.ac is 1) or IA_32_Exception(Debug) trap.
- If an IA-32 IN/OUT I/O port Accesses cross a 4-port boundary the processor will break the operation into smaller 1, 2, or 3 byte transactions.
- Assuming no faults, a physical transaction is emitted to the mapped or specified physical address.

The processor ensures that IA-32 IN, INS, OUT, OUTS references are fully ordered and will not allow prior or future data memory references to pass the I/O operation as defined in Section 10.6.10, “IA-32 Memory Ordering” on page 2:265. The processor will wait for acceptance for IN and OUT operations before proceeding with subsequent externally visible bus transactions.

10.7.4 I/O Port Accesses by Loads and Stores

If an access is made to the I/O port block using IA-32 or Itanium loads and stores the following differences in behavior should be noted; EFLAG.iopl permission is not checked, TSS permission bitmap is not checked, and stores and loads do not honor IN and OUT memory ordering and acceptance semantics (the processor will not automatically wait for a store to be accepted by the platform).

Virtual addresses for the I/O port space should be computed as defined in Section 10.7.1, “Virtual I/O Port Addressing” on page 2:268 If data translations are enabled, the TLB is consulted for mappings and permission, and the resulting mapped physical address used to address the physical I/O device.

If IA-32 ordering semantics are required to a particular I/O port device (or memory mapped I/O device), IA-32 or Itanium architecture-based software must enforce ordering to the I/O device. Software can either perform a memory ordering fence before and after the transaction, or use an load acquire or store release

To ensure the processor does not speculatively access an I/O device, all I/O devices must be mapped by the UC memory attribute.

If IA-32 acceptance semantics are required (i.e. additional data memory transactions are not initiated until the I/O transaction is completed), Itanium architecture-based code can issue a memory acceptance fence. Conversely, if certain I/O devices do not require IA-32 IN/OUT ordering or acceptance semantics, Itanium architecture-based code can relax ordering and acceptance requirements as shown below.

OUT

```
[mf]//Fence prior memory references, if required

add port_addr = IO_Port_Base, Expanded_Port_Number
st.rel (port_addr), data
[mf.a] //Wait for platform acceptance, if required
[mf]   //Fence future memory operations, if required
```

IN

```
[mf]   //Fence prior memory references, if required
add port_addr = IO_Port_Base, Expanded_Port_Number
ld.acq data, (port_addr)
[mf.a] //Wait for platform acceptance, if required
[mf]   //Fence future memory references, if required
```

10.8 Debug Model

The debug facilitates defined by the Itanium architecture are designed to support debugging of both the Itanium and IA-32 instruction set. The following debug events can be triggered during IA-32 instruction set execution by Itanium debug resources.

- **Single Step trap** – When PSR.ss is 1 (or EFLAG.tf is 1), successful execution of each IA-32 instruction, results in an IA_32_Exception(Debug) trap. After the single step trap, IIP points to the next IA-32 instruction to be executed.
- **Breakpoint Instruction trap** – execution of INT 3 (breakpoint) instruction results in a IA_32_Exception(Debug) trap.
- **Instruction Debug fault** – When PSR.db is 1 and PSR.id is 0 and EFLAG.rf is 0, any IA-32 instruction fetch that matches the parameters specified by the IBR registers results in an IA_32_Exception(Debug) fault. After servicing a Debug fault, debuggers can set PSR.id (or EFLAG.rf for IA-32 instructions) before restarting the faulting instruction. If PSR.id is 1, Instruction Debug faults are temporarily disabled for one Itanium instruction. If PSR.id is 1 or EFLAG.rf is 1, Instruction Debug faults are temporarily disabled for one IA-32 instruction. The successful execution of an IA-32 instruction clears both PSR.id and EFLAG.rf bits. The successful execution of an Itanium instruction only clears PSR.id.
- **Data Debug traps** – When PSR.db is 1, any IA-32 data memory reference that matches the parameters specified by the DBR registers results in a IA_32_Exception(Debug) trap. IA-32 data debug events are traps, not faults as defined for Itanium instruction set data debug events. Trap behavior is required since any given IA-32 instruction can access several memory locations during its execution. The reported trap code returns the match status of the first four DBR registers that matched during the execution of the IA-32 instruction. Zero, one or DBR registers may be reported as matching.
- **Taken Branch trap** – When PSR.tb is 1, a IA_32_Exception(Debug) trap occurs on every IA-32 taken branch instruction (CALL, Jcc, JMP, RET, LOOP). After the trap, IIP points to the branch target.
- **Lower Privilege Transfer trap** – Does not occur during IA-32 instruction set execution.

For virtual memory accesses, breakpoint address registers contain the virtual addresses of the debug breakpoint. For physical accesses, the addresses in these registers are treated as a physical address. Software should be aware that debug registers configured to fault on virtual references, may also fault on a physical reference if translations are disabled. Likewise a debug register configured for physical references can fault on virtual references that match the debug breakpoint registers.

10.8.1 Data Breakpoint Register Matching

Each Itanium data breakpoint register has the following matching behavior for IA-32 instruction set data memory references:

- **DBR.addr** – IA-32 single or multi-byte data memory references that access ANY memory byte specified by the DBR address and mask fields results in a debug breakpoint trap regardless of datum size and alignment. The upper 32-bits of DBR.addr must be zero to detect IA-32 data memory references. Since IA-32 data breakpoints are traps, all processor implementations ensure data breakpoint traps are precise. Traps are only reported if any byte in the data memory reference ANDed with the DBR mask bitwise matches the DBR address field ANDed with the DBR mask. No spurious data breakpoint faults are generated for IA-32 data memory operands that are unaligned, nor are matches reported if no bytes of the operand lie within the address range specified by the DBR address and mask fields. Note, Itanium instruction set generated unaligned data memory references may result in spurious imprecise breakpoint faults.
- **DBR.mask** – by programming the mask a breakpoint range of 1, 2, 4, 8, or any power of 2 combination can be supported. Mask bits above bit 31 are checked by the processor during IA-32 data memory references
- **Trap code B bits** – are set indicating a match with the corresponding data breakpoint register DBR0-3. For IA-32 data debug traps, any number of B-bits can be set indicating a match.

The B-bits are only set and a data breakpoint trap generated if 1) the breakpoint register precisely matches the specified DBR address and mask, 2) it is enabled by the DBR read or write bits for the type of the memory transaction, 3) the DBR privilege field matches PSR.cpl, 4) PSR.db is 1, and 5) no other higher priority faults are taken.

I/O port space breakpoints can be configured by loading the address and mask fields with the virtual address defined by the operating system to correspond to the I/O port space.

10.8.2 Instruction Breakpoint Register Matching

Each Itanium instruction breakpoint register has the following matching behavior for IA-32 instruction set memory fetches:

- **IBR.addr** – an IBR register matches an IA-32 instruction fetch address, if the first byte of an IA-32 instruction address ANDed with the IBR mask bitwise matches the IBR address field ANDed with the IBR mask. Note that only the first byte is analyzed. The upper 32-bits of IBR.addr must be zero to detect IA-32 instruction fetches.
- **IBR.mask** – by programming the mask a breakpoint range of 1, 2, 4, 8, or any power of 2 combination can be supported. Mask bits above bit 31 are ignored during IA-32 instruction fetches.

The instruction breakpoint fault is generated if 1) the breakpoint register precisely matches the specified IBR address and mask, 2) it is enabled by the IBR execute bit, 3) the IBR privilege field matches PSR.cpl, 4) PSR.db is 1, 5) PSR.id is 0, and 6) no other higher priority faults are taken.

10.9 Interruption Model

Within the Itanium System Environment, all interruptions originating out of the IA-32 or Itanium instruction sets are delivered to Itanium architecture-based Interruption Handlers within the Itanium architecture-based operating system. Virtual memory management faults, machine checks, and external interrupts are always delivered to Itanium architecture-based interruption handlers regardless of the instruction set running at the time of the interruption. IA-32 exceptions, control transfers through gates, task switches, and accesses to sensitive IA-32 system resources are intercepted into Itanium architecture-based interruption handlers. Using these intercepts, Itanium architecture-based software can implement specific policies with regard to that resource. Policies may include virtualization, emulation of an IA-32 opcode or memory access, or various permission policies.

In general, if an interruption is independent of the executing instruction set (such as an external interruption or TLB fault) common Itanium architecture-based handlers are invoked. For classes of exceptions and intercept conditions that are specific to the IA-32 instruction set, three special Itanium architecture-based software handlers are invoked to deal with IA-32 instruction set interruptions. Table 10-8 shows the three interruption handlers defined to support IA-32 events. See Section 9.2, "IA-32 Interruption Vector Definitions" on page 2:213 for details on these interruption handlers.

Table 10-8. IA-32 Interruption Vector Summary

Handler	Description
IA_32_Intercept	Intercepted IA-32 instructions, I/O, system flag manipulation and gate transfers.
IA_32_Exception	IA-32 instruction set generated exceptions.
IA_32_Interrupt	IA-32 instruction set generated software interrupts

This grouping of interruption handlers simplifies software handlers such that they do not need to be concerned with behavior of both IA-32 and Itanium instruction sets.

Interruption registers (defined in Chapter 3) record the state of IA-32 execution at the point of interruption. For IA-32 exceptions, ISR contains IA-32 defined error codes and vector numbers as defined by the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***. IA-32 instruction set related exceptions and software interruptions vector directly through the interruption mechanism defined by the Itanium architecture without consulting the IA-32 IDT or performing any memory stack pushes.

10.9.1 Interruption Summary

Table 10-9 summarizes the set of all IA-32 interruptions and how they are mapped to Itanium architecture-based interruption handlers within the Itanium System Environment. See Chapter 9 and Chapter 8 for a detailed definition of each interruption.

Table 10-9. IA-32 Interruption Summary

IA-32 Vector	Itanium® Architecture-based Interruption Handler	ISR Vector	ISR Code	Description
IA-32 Defined Interruptions				
0	IA_32_Exception (Divide)	0	0	IA-32 divide by zero fault.

Table 10-9. IA-32 Interruption Summary (Continued)

IA-32 Vector	Itanium® Architecture-based Interruption Handler	ISR Vector	ISR Code	Description
1	IA_32_Exception (Debug)	1	0	IA-32 instruction breakpoint fault.
1	IA_32_Exception (Debug)	1	TrapCode	IA-32 debug events. The Trap Code indicates concurrent taken branch, data breakpoint and single step trap conditions.
2	External Interrupt	0	0	NMI is delivered through the Intel Itanium External Interrupt vector.
3	IA_32_Exception(Break)	3	TrapCode	IA-32 INT 3 instruction.
4	IA_32_Exception(INTO)	4	TrapCode	IA-32 INTO detected overflow trap.
5	IA_32_Exception (Bound)	5	0	IA-32 BOUND range exceeded fault.
6	IA_32_Intercept(Inst)	0	InterceptCode	All IA-32 unimplemented and illegal opcodes.
7	IA_32_Exception(DNA)	7	0	IA-32 Device not available fault.
8	--	N/A		IA-32 Double fault can not be generated in the Intel Itanium System Environment, Intel reserved.
9	--	N/A		Intel reserved
10	--	N/A		IA-32 Invalid TSS fault can not generated in the Intel Itanium System Environment, Intel reserved,
11	IA_32_Exception(NotPresent)	11	ErrorCode[a]	IA-32 Segment Not present fault.
12	IA_32_Exception (Stack)	12	ErrorCode	IA-32 Stack Exception fault.
13	IA_32_Exception (GPFault)	13	ErrorCode	IA-32 General Protection fault.
14	Intel Itanium TLB faults	see Data TLB faults below		IA-32 Page fault can not be generated in the Intel Itanium System Environment, replaced by Intel Itanium TLB faults, Intel reserved,
15	--	N/A		Intel reserved.
16	IA_32_Exception (FPError)	16	0	IA-32 floating-point fault.
17	IA_32_Exception(AlignCheck)	17	0	IA-32 un-aligned data references.
18	Corrected MCHK	N/A		IA-32 Machine Check can not be generated in the Intel Itanium System Environment, replaced by the PAL Machine Check Architecture, Intel reserved.
19	IA_32_Exception (StreamSIMD)	19	0	IA-32 SSE Numeric Error fault.
20-31	--	N/A		Intel reserved.
0-255	External Interrupt	0	0	External interrupts are delivered through the Intel Itanium External Interrupt vector. Software must read the IVR register to determine the vector number.
0-255	IA_32_Interrupt (vector #)	Vect#	TrapCode	IA-32 Software Interrupt trap. ISR contains the vector number.
IA-32 Interceptions				

Table 10-9. IA-32 Interruption Summary (Continued)

IA-32 Vector	Itanium® Architecture-based Interruption Handler	ISR Vector	ISR Code	Description
	IA_32_Intercept(Inst)	0	InterceptCode	Intercept for unimplemented, illegal or privileged IA-32 opcodes.
	IA_32_Intercept(Gate)	1	TrapCode	Intercept for control transfers through a Call Gate, Task gate or Task Segment.
	IA_32_Intercept(SystemFlag)	2	TrapCode	Intercept for modification of system flag values.
	IA_32_Intercept(Lock)	4	0	IA-32 semaphore operation requires an external bus lock when DCR.lc is 1.
		3,5-255	--	Intel reserved

a. The IA-32 Error Code is defined as a Selector Index, and TI, IDT and EXT bits are set based on the exception. See ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the complete definition.

10.9.2 IA-32 Numeric Exception Model

IA-32 numeric instructions follow the IA-32 delayed floating-point exception model. Specifically IA-32 numeric exceptions are held pending until the next IA-32 numeric or MMX technology instruction as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***. Numeric faults generated on SSE instructions are reported precisely on the faulting SSE instruction. SSE instructions do NOT trigger the report of pending IA-32 numeric exceptions.

For voluntary transitions out of the IA-32 instruction, an implicit FWAIT operation is performed by the `jmpe` instruction to ensure all pending numeric exceptions are reported. For involuntary transitions out of the IA-32 instruction set (external interruptions, TLB faults, exceptions, etc.) the processor does not perform a FWAIT operation. However, every IA-32 numeric instruction that generates a pending numeric exception loads the application registers FSR, FIR, and FDR with the IA-32 floating-point state on the instruction that generating the exception. This state contains information defined by the IA-32 FSTENV and FLDENV instructions. During a process context switch, the operating system must save and restore FSR, FIR, and FDR (effectively performing an FSTENV and FLDENV) to ensure numeric exceptions are correctly reported across a process switch.

10.10 Processor Bus Considerations for IA-32 Application Support

The section briefly discusses bus and platform considerations when supporting IA-32 applications in the Itanium System Environment.

Itanium architecture-based code does not assert the SPLCK and LOCK pins. The LOCK pin is used by IA-32 code to signal an external atomic bus transaction for which atomicity cannot be enforced within the processor's caches, whereas, SPLCK indicates if an unaligned external bus lock requires a split lock operation and hence several bus transactions. For IA-32 code, if the platform does not support LOCK or SPLCK, the operating system must disable external bus lock transactions by setting DCR.lc to 1. When DCR.lc is 1, any IA-32 atomic reference not serviced internally in the processor's caches results in an IA_32_Intercept(Lock) fault. See Section 3.3.4.1, "Default Control Register (DCR – CR0)" on page 2:31 for details. When DCR.lc is 0, operating system code is responsible for emulation of the IA-32 instruction and ensuring atomicity (if required).

The A20M and IGNE pins are ignored in the Itanium System Environment. FERR is not asserted in the Itanium System Environment.

In both IA-32 and Itanium System Environments, the M/IO pin (or an external bus indication) is asserted by any memory reference to the 64MB I/O port block range of the physical address space. See Section 10.7, "I/O Port Space Model" on page 2:267 for details.

SMI and the SMM environment are not supported on processors based on the Itanium architecture. The PMI interrupt and PAL firmware environment replace them. See Section 11.5, "Platform Management Interrupt (PMI)" on page 2:310 for details.

10.10.1 IA-32 Compatible Bus Transactions

Within the Itanium System Environment, the following bus transactions are initiated:

- INTA – Interrupt Acknowledge - emitted by the operating system (via a read to the INTA byte in the processor's Interrupt Block) to acquire the interrupt vector number from an external interrupt controller.
- HALT – Emitted when the processor has entered the halt state due to the operating system/platform firmware calling PAL_HALT or PAL_HALT_LIGHT.
- SHUTDOWN – Emitted when the processor has entered the shutdown state. This can only be generated when the processor has entered into the IA-32 System Environment by calling PAL_ENTER_IA_32_ENV procedure call.
- STPACK – Stop Acknowledge. Emitted by calling an implementation-specific PAL firmware procedure. See the processor-specific firmware guide for more information.
- FLUSH – Emitted when the WBINVD or INVD instruction is executed when running in the IA-32 System Environment entered by calling PAL_ENTER_IA_32_ENV procedure call. Indicates that external caches (if any) should be invalidated.
- SYNC – Emitted when the WBINVD instruction is executed when running in the IA-32 System Environment entered by calling PAL_ENTER_IA_32_ENV procedure call. Indicates that external caches (if any) should copy all modified cache lines back to main memory.

§

Processor Abstraction Layer 11

This chapter defines the architectural requirements for the **Processor Abstraction Layer (PAL)** for all processors based on the Itanium architecture. It is intended for processor designers, firmware/BIOS designers, system designers, and writers of diagnostic and low level operating system software.

PAL is part of the Itanium processor architecture and its goal is to provide a consistent firmware interface to abstract processor implementation-specific features.

The objectives of this chapter are to define:

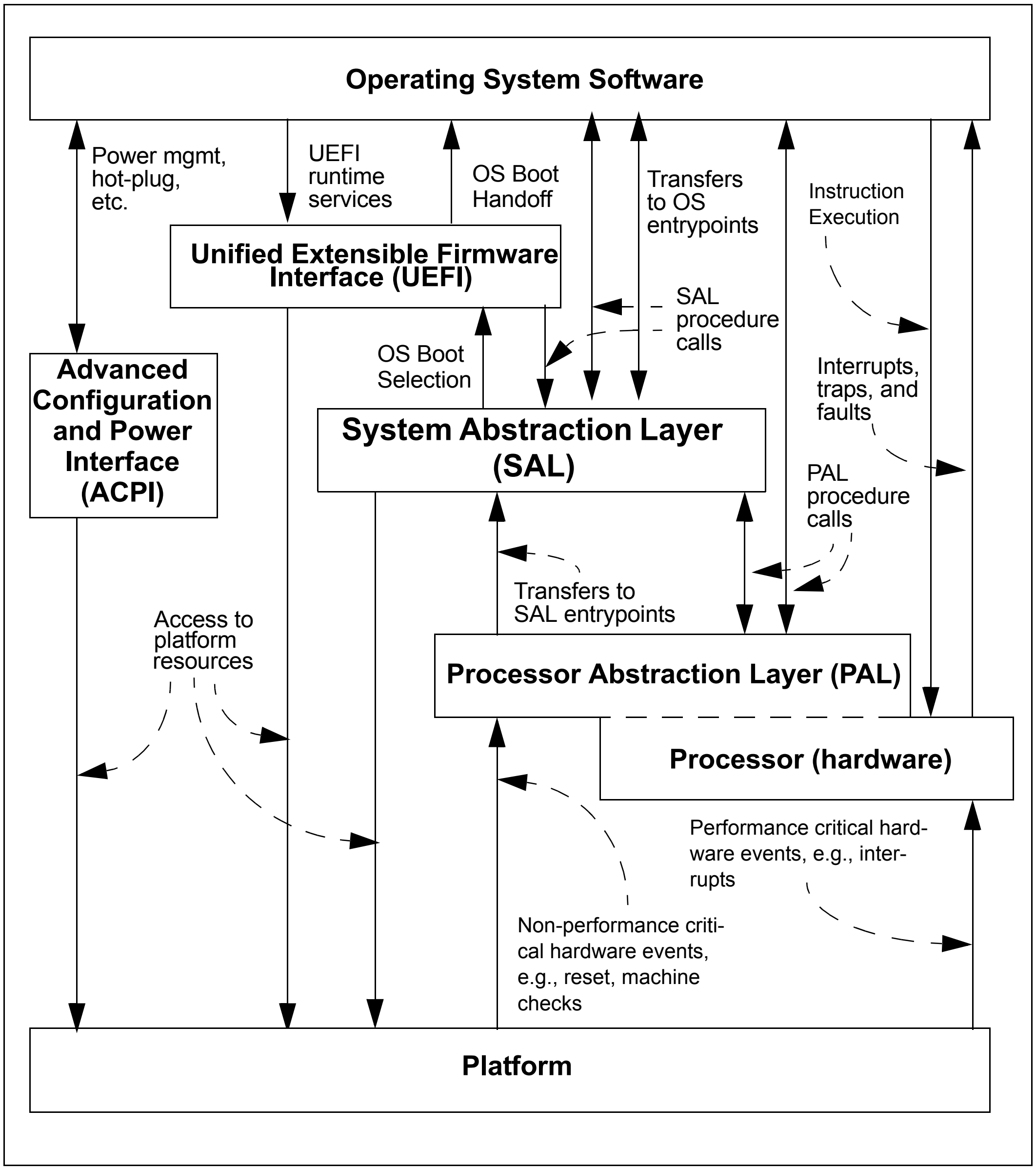
- The architectural behavior and interface requirements for processor testing, configuration and error recovery. This includes the hardware entrypoints into PAL and the PAL interfaces to platform firmware and system software.
- A set of boot and runtime PAL procedures to access processor implementation-specific hardware and to return information about processor implementation-dependent configuration.
- A computing environment for both PAL entrypoints and procedures such that:
  - Memory used by PAL procedures is allocated by the caller of PAL procedures.
  - PAL code runs little endian.
  - PAL interface is as endian neutral as possible.
  - PAL is Itanium architecture-based code.
  - PAL code runs at privilege level 0.
  - PAL procedures can be called without backing store, except where memory-based parameters are returned.
- The processor and platform hardware requirements for PAL. This includes minimizing PAL dependencies on platform hardware and clearly stating where those dependencies exist.
- A PAL interface and requirements to support firmware update and recovery.

11.1 Firmware Model

As shown in Figure 11-1, Itanium architecture-based firmware consists of several major components: Processor Abstraction Layer (PAL), System Abstraction Layer (SAL), Unified Extensible Firmware Interface (UEFI) and Advanced Configuration and Power Interface (ACPI). PAL, SAL, UEFI and ACPI together provide processor and system initialization for an operating system boot. PAL and SAL provide machine check abort handling. PAL, SAL, UEFI and ACPI provide various run-time services for system functions which may vary across implementations. The interactions of the various services that PAL, SAL, UEFI and ACPI provide are illustrated in Figure 11-1.

In the context of this model and throughout the rest of this chapter, the System Abstraction Layer (SAL) is a firmware layer which isolates operating system and other higher level software from implementation differences in the platform, while PAL is the firmware layer that abstracts the processor implementation.

Figure 11-1. Firmware Model


Operating System Software
Power mgmt, hot-plug, etc.
UEFI runtime services
OS Boot Handoff
Transfers to OS entrypoints
Instruction Execution
Unified Extensible Firmware Interface (UEFI)
SAL procedure calls
Advanced Configuration and Power Interface (ACPI)
OS Boot Selection
Interrupts, traps, and faults
System Abstraction Layer (SAL)
PAL procedure calls
Transfers to SAL entrypoints
Access to platform resources
Processor Abstraction Layer (PAL)
Processor (hardware)
Performance critical hardware events, e.g., interrupts
Non-performance critical hardware events, e.g., reset, machine checks
Platform

11.1.1 Processor Abstraction Layer (PAL) Overview

The purpose of the Processor Abstraction Layer, is to provide a firmware abstraction between the processor hardware implementation and system software and platform firmware, so as to maintain a single software interface for multiple implementations of the processor hardware. PAL is defined to be independent of the number of processors on a platform.

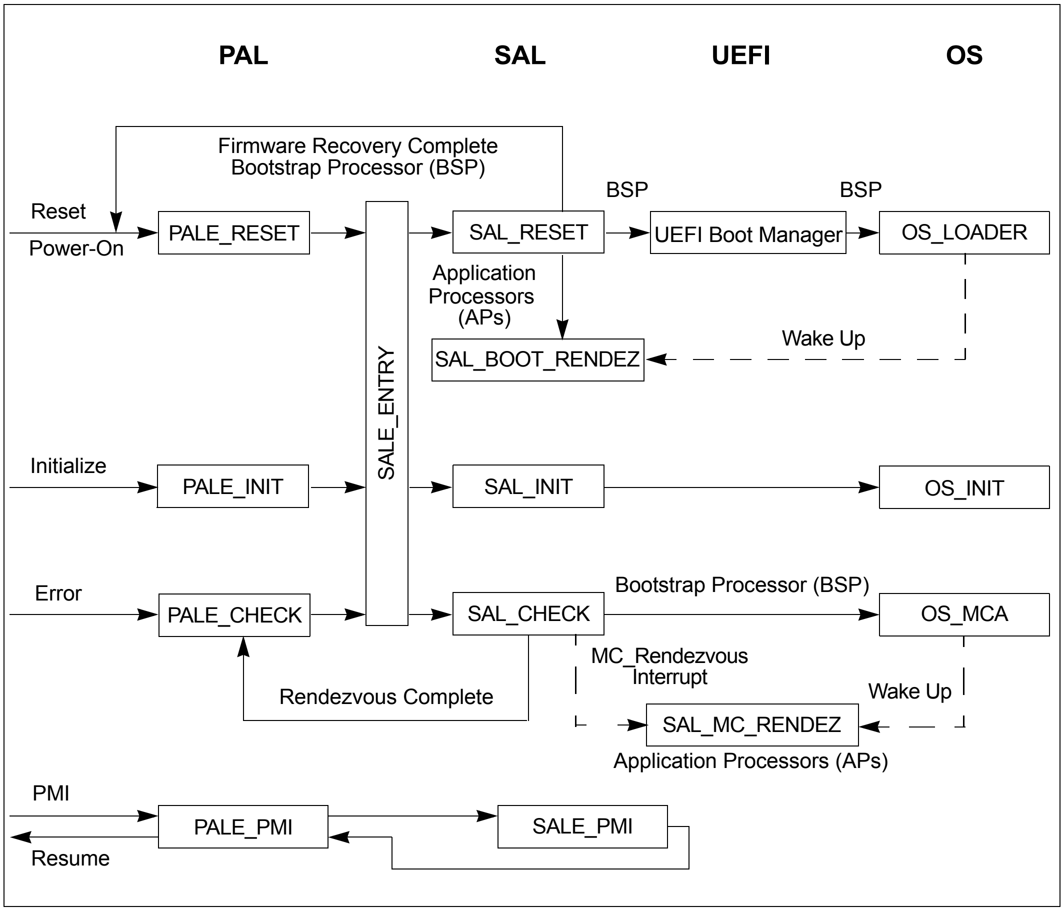
PAL encapsulates those processor functions that are likely to change on an implementation to implementation basis so that SAL firmware and operating system software can maintain a consistent view of the processor. These include non-performance critical functions dealing such as processor initialization, configuration and error handling.

PAL consists of two main components:

- Entrypoints, which are invoked directly by hardware events such as reset, init and machine checks. These interruption entrypoints perform functions such as processor initialization and error recovery.
- Procedures, which may be called by higher level firmware and software to obtain information about the identification, configuration, and capabilities of the processor implementation; to perform implementation-dependent functions such as cache initialization; or to allow software to interact with the hardware through such functions as power management or enabling/disabling processor features.

11.1.2 Firmware Entrypoints

Figure 11-2. Firmware Entrypoints Logical Model



11.1.3 PAL Entrypoints

The following hardware events can trigger the execution of a PAL entrypoint:

- Power-on/reset
- Hardware errors (both correctable and uncorrectable)
- Initialization event (via external interrupt bus message or processor pin)
- Platform management interrupt (via external interrupt bus message or processor pin)

These hardware events trigger the execution of one of the following PAL entrypoints (as shown in Figure 11-2):

- PALE_RESET – Initializes and tests the processor following power-on or reset and then branches to SALE_ENTRY to determine whether to perform firmware recovery update, or to boot the machine for OS use. See Section 11.1.4, “SAL Entrypoints” on page 2:282.
- PALE_CHECK – Determines if errors are processor related, saves processor related error information and corrects errors where possible (for example, by flushing a corrupted instruction cache line and marking the cache line as unusable). In all cases, PALE_CHECK branches to SALE_ENTRY to complete the error logging, correction, and reporting.
- PALE_INIT – Saves the processor state, places the processor in a known state, and branches to SALE_ENTRY. PALE_INIT is entered as a response to an initialization event.
- PALE_PMI – Saves the processor state and branches to SALE_PMI. PALE_PMI is entered as a response to a platform management interrupt.

11.1.4 SAL Entrypoints

There are two entrypoints from PAL into SAL:

- SALE_ENTRY – PAL branches to this SAL entrypoint after a power-on, reset, machine check, or initialization event. If SALE_ENTRY was invoked by a machine check or initialization event, SALE_ENTRY branches to the appropriate routine:
  - SAL_CHECK is invoked after a machine check.
  - SAL_INIT is invoked after an initialization event.

  If SALE_ENTRY was invoked by a reset or power on, it checks to determine if a firmware recovery condition exists. If it does, SALE_ENTRY performs the firmware update, then performs a RESET operation to invoke PAL_RESET. If a recovery condition does not exist, SAL_ENTRY returns to PAL_RESET to complete processor self-test. PAL_RESET then branches back to SALE_ENTRY, which, in turn, branches to SAL_RESET.

- SALE_PMI – platform management interrupt. PALE_PMI branches to this SAL entrypoint after saving processor state in response to the platform management interrupt.

11.1.5 OS Entrypoints

There are several entrypoints from SAL into an operating system (or equivalent software). Entrypoints from SAL into the operating system are expected to meet the following model:

- OS_BOOT – Operating System Boot interface.
- OS_MCA – Operating System Machine Check Abort Handler.
- OS_INIT – Operating System Initialization Handler.
- OS_RENDEZ – Operating System Multiprocessor Rendezvous interface.

11.1.6 Firmware Address Space

The firmware address space occupies the 16 MB region between 4 GB - 16 MB and 4 GB (addresses 0xFF00_0000 through 0xFFFF_FFFF). There are two primary layouts of this address space. The first version is shown in Figure 11-3 and the second version is shown in Figure 11-4. The first version has one PAL_A component. This layout allows for robust recovery of PAL_B and SAL_B components. This layout is useful for cases where PAL_A will not need to be upgraded. The second version splits the PAL_A block into two components. The first component is referred to as the generic PAL_A and the second component is the processor-specific PAL_A. Splitting the PAL_A up in this manner allows for a robust upgrade of the processor-specific PAL_A firmware as well as the PAL_B and SAL_B components. This is very useful if a platform is designed to support multiple processor generations which would require a PAL_A upgrade when the new processor generation is released. The generic PAL_A which resides in the Protected Boot Block will work across processor generations for a given platform. The processor-specific PAL_A resides outside the Protected Boot Block and works for a specific processor generation.

Figure 11-3. Firmware Address Space

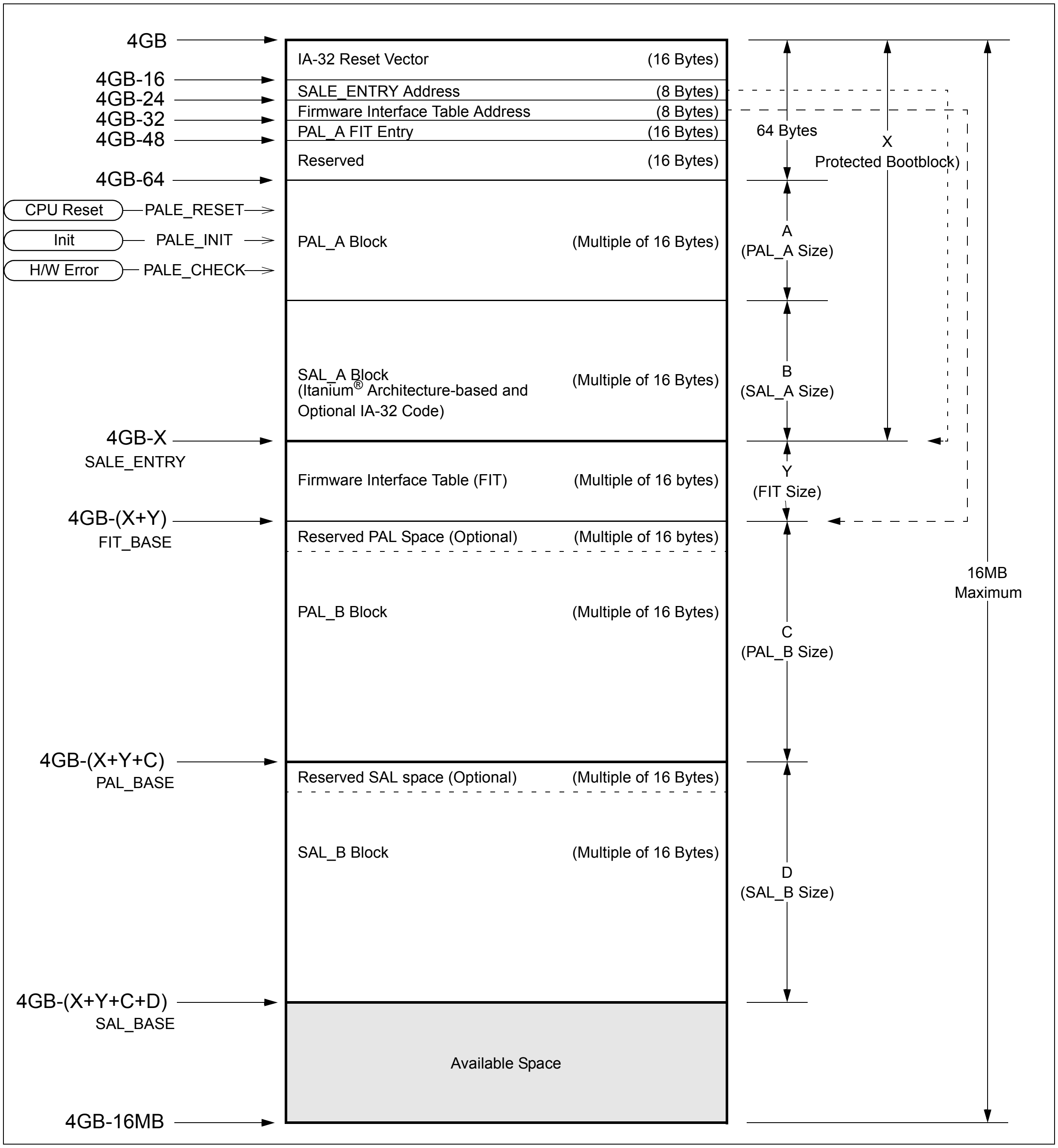
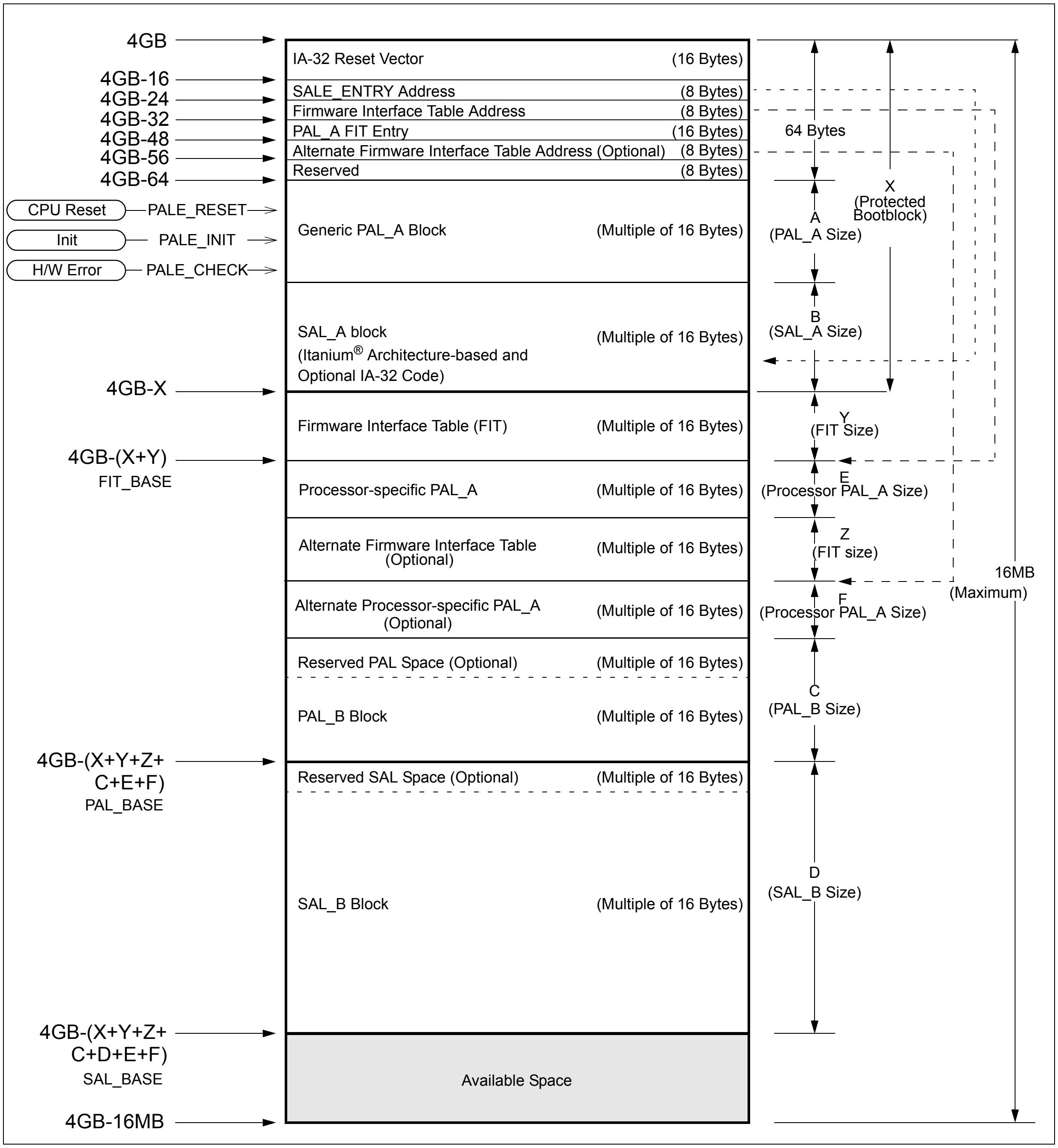

4GB
4GB-16
4GB-24
4GB-32
4GB-48
4GB-64
CPU Reset
PALE_RESET
Init
PALE_INIT
H/W Error
PALE_CHECK
4GB-X
SALE_ENTRY
4GB-(X+Y)
FIT_BASE
4GB-(X+Y+C)
PAL_BASE
4GB-(X+Y+C+D)
SAL_BASE
4GB-16MB
IA-32 Reset Vector (16 Bytes)
SALE_ENTRY Address (8 Bytes)
Firmware Interface Table Address (8 Bytes)
PAL_A FIT Entry (16 Bytes)
Reserved (16 Bytes)
PAL_A Block (Multiple of 16 Bytes)
SAL_A Block (Itanium® Architecture-based and Optional IA-32 Code) (Multiple of 16 Bytes)
Firmware Interface Table (FIT) (Multiple of 16 bytes)
Reserved PAL Space (Optional) (Multiple of 16 bytes)
PAL_B Block (Multiple of 16 Bytes)
Reserved SAL space (Optional) (Multiple of 16 Bytes)
SAL_B Block (Multiple of 16 Bytes)
Available Space
64 Bytes
X
Protected Bootblock)
A
(PAL_A Size)
B
(SAL_A Size)
Y
(FIT Size)
C
(PAL_B Size)
D
(SAL_B Size)
16MB
Maximum

Figure 11-4. Firmware Address Space with Processor-specific PAL_A Components



4GB
4GB-16
4GB-24
4GB-32
4GB-48
4GB-56
4GB-64
CPU Reset
PALE_RESET
Init
PALE_INIT
H/W Error
PALE_CHECK
4GB-X
4GB-(X+Y)
FIT_BASE
4GB-(X+Y+Z+C+E+F)
PAL_BASE
4GB-(X+Y+Z+C+D+E+F)
SAL_BASE
4GB-16MB
IA-32 Reset Vector (16 Bytes)
SALE_ENTRY Address (8 Bytes)
Firmware Interface Table Address (8 Bytes)
PAL_A FIT Entry (16 Bytes)
Alternate Firmware Interface Table Address (Optional) (8 Bytes)
Reserved (8 Bytes)
Generic PAL_A Block (Multiple of 16 Bytes)
SAL_A block (Itanium® Architecture-based and Optional IA-32 Code) (Multiple of 16 Bytes)
Firmware Interface Table (FIT) (Multiple of 16 Bytes)
Processor-specific PAL_A (Multiple of 16 Bytes)
Alternate Firmware Interface Table (Optional) (Multiple of 16 Bytes)
Alternate Processor-specific PAL_A (Optional) (Multiple of 16 Bytes)
Reserved PAL Space (Optional) (Multiple of 16 Bytes)
PAL_B Block (Multiple of 16 Bytes)
Reserved SAL Space (Optional) (Multiple of 16 Bytes)
SAL_B Block (Multiple of 16 Bytes)
Available Space
64 Bytes
X (Protected Bootblock)
A (PAL_A Size)
B (SAL_A Size)
Y (FIT Size)
E (Processor PAL_A Size)
Z (FIT size)
F (Processor PAL_A Size)
C (PAL_B Size)
D (SAL_B Size)
16MB (Maximum)

The firmware address space is shared by SAL and PAL. Some of the SAL/PAL boundaries are implementation dependent. The address space contains the following regions and locations.

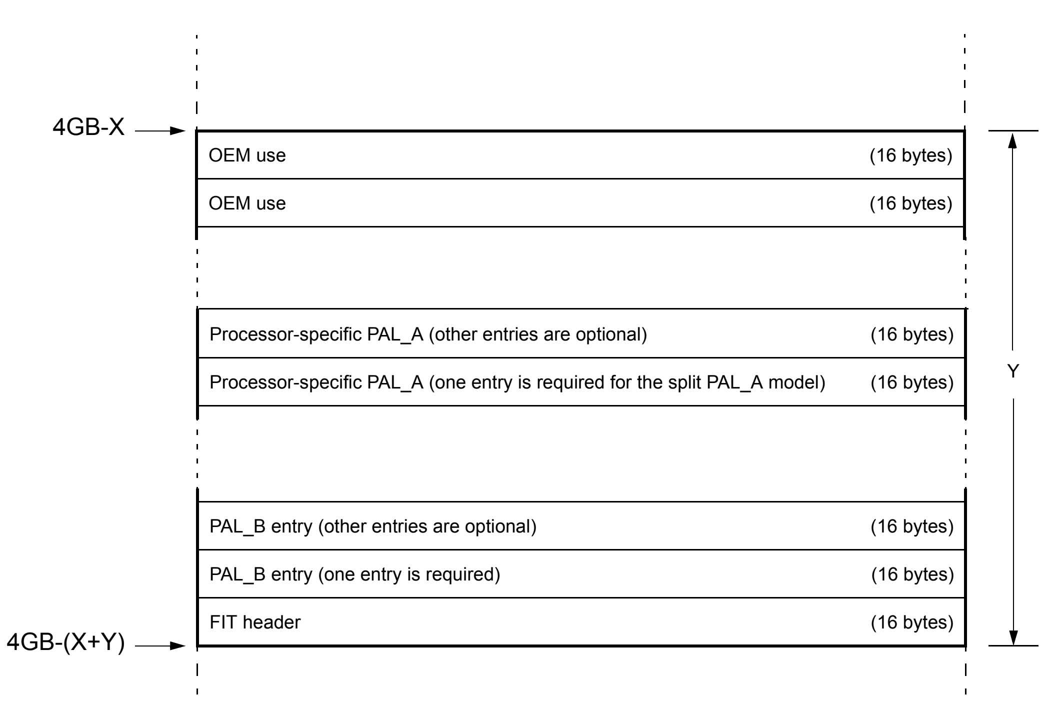
- The 16 bytes at 0xFFFF_FFF0 (4GB-16) contain IA-32 Reset Code.
- The 8 bytes at 0xFFFF_FFE8 (4GB-24) contain the physical address of the SALE_ENTRY entrypoint.

- The 8 bytes at 0xFFFF_FFE0 (4GB-32) contain the physical address of the Firmware Interface Table.
- The 16 bytes at 0xFFFF_FFD0 (4GB-48) contain the FIT entry for the PAL_A (or generic PAL_A in the split PAL_A model) code provided by the processor vendor. The format of this FIT entry is described in Figure 11-6.
- The 8 bytes at 0xFFFF_FFC8 (4GB-56) contains the physical address of the alternate Firmware Interface Table. This pointer is optional and is only needed if the firmware contains an alternate FIT table. If no alternate FIT table it provided a value of 0x0 should be encoded in this entry.
- The 8 bytes at 0xFFFF_FFC0 (4GB-64) are zero-filled and reserved for future use.
- PAL_A code (also known as generic PAL_A code in split PAL_A model) resides below 0xFFFF_FFC0. This area contains the hardware-triggered entrypoints PALE_RESET, PALE_INIT, and PALE_CHECK. In the model where PAL_A is not split, the PAL_A code will perform any processor-specific initialization needed in order for SAL to perform a firmware recovery. In the split PAL_A model, the generic PAL_A will search the FIT table(s) to find the first compatible and error-free processor-specific PAL_A code. It will then branch to this code to perform the processor-specific initialization needed in order for SAL to perform a firmware recovery. The PAL_A code area is a multiple of 16 bytes in length.
- SAL_A code occupies the region immediately below the PAL_A code. This area contains the SALE_ENTRY entrypoint as well as optional implementation-independent firmware update code. The SAL_A code area is a multiple of 16 bytes in length.
- The collection of regions above from the beginning of the SAL_A code to 4GB is called the Protected Bootblock. The size of the Protected Bootblock is SAL_A size + PAL_A size + 64.
- The Firmware Interface Table (FIT) comprises of 16-byte entries containing starting address and size information for the firmware components. The FIT is generated at build time, based on the size and location of the firmware components. Optionally, an alternate FIT may be included in the firmware. The alternate FIT will only be used if the primary FIT failed its checksum. In the split PAL_A model, this allows the generic PAL_A firmware to find the processor-specific PAL_A component(s), even if the primary FIT is corrupt. This feature allows hand-off to the SAL recovery code, even if there is a primary FIT checksum failure.
- The processor-specific PAL_A contains the code that is required to be run before handing off to SAL for a firmware recovery check. This component is only available on processors that support a split PAL_A firmware model. One processor-specific PAL_A is architecturally required in this model. The firmware may optionally contain two or more processor-specific PAL_A components.
- The PAL_B block is comprised of code that is not required to be executed for SAL to perform a firmware recovery update. The PAL_B code area is a multiple of 16 bytes in length. The PAL_B block must be aligned on a 32K byte boundary or a 64K byte boundary depending on the implementation. Processor specific documentation provides the requirement for alignment. An OEM can choose to have more than one PAL_B block in the firmware image.
- The remainder of the firmware address space is occupied by SAL_B code. SAL_B may include IA-32 BIOS code. The location of the SAL_B and IA-32 BIOS code within the firmware address space is implementation dependent.

At a minimum, all of the PAL firmware components, pointers at the top of the firmware address space, FIT tables and the portion of the SAL code that is executed at the RECOVERY CHECK hand-off must be accessible from the processor without any special system fabric initialization sequence. This implies that the system fabric is implicitly initialized at power on for accessing the portions of the firmware address space listed above or that the special hardware which contains the firmware code and data is implemented on the processor and not accessed across the system fabric. The entire firmware code and data area can also be implicitly initialized at power on from the processor as well, but the minimum set is listed above.

The Firmware Interface Table (FIT) contains starting addresses and sizes for the different firmware components. Because these code blocks may be compiled at different times and places, code in one block (such as PAL_A) cannot branch to code in another block (such as PAL_B) directly. The FIT allows code in one block to find entrypoints in another. Figure 11-5 below shows the FIT layout.

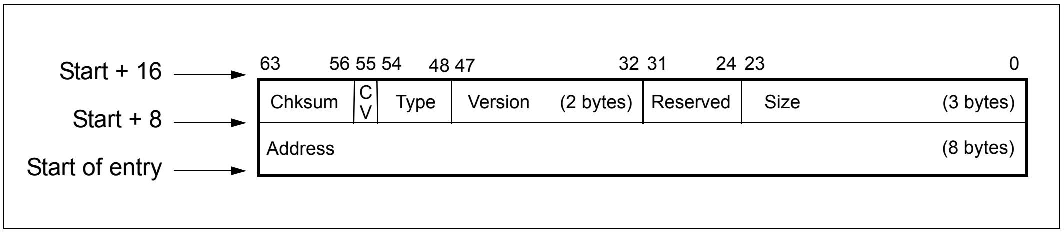
Figure 11-5. Firmware Interface Table


4GB-X
OEM use (16 bytes)
OEM use (16 bytes)
Processor-specific PAL_A (other entries are optional) (16 bytes)
Processor-specific PAL_A (one entry is required for the split PAL_A model) (16 bytes)
Y
PAL_B entry (other entries are optional) (16 bytes)
PAL_B entry (one entry is required) (16 bytes)
FIT header (16 bytes)
4GB-(X+Y)

Each FIT entry contains information for the corresponding firmware component. The first entry contains size and checksum information for the FIT itself. The order of the following FIT entries must be arranged in ascending order by the type field, otherwise execution of firmware code will be unpredictable. Multiple FIT entries of the same type are allowed as shown in Figure 11-5.

When multiple entries of the same type exist for PAL components, PAL searches the FIT table in ascending order looking for the first entry that is compatible and error free for the processor it is currently executing on.

Figure 11-6. Firmware Interface Table Entry



- *Size* – A 3-byte field containing the size of the component in bytes divided by 16.
- Reserved – All fields listed as reserved must be zero filled.
- *Version* – A 2-byte field containing the component's version number.
- *Type* – A 7-bit field containing the type code for the element. Types are defined in Table 11-1.

Table 11-1. FIT Entry Types

Type	Meaning
0x00	FIT Header
0x01	PAL_B (required)
0x02-0x0D	Reserved
0x0E	Processor-specific PAL_A
0x0F	PAL_A (also generic PAL_A)[a]
0x10-0x7E	OEM-defined
0x7F	Unused Entry

[a] The PAL_A FIT entry is located at 0xFFFF_FFDO (4GB-48) and is not part of the actual FIT table.

OEMs may define unique types for one or more blocks of SAL_B, IA-32 BIOS, etc., within the OEM-defined type range of 0x10 to 0x7E.

- *C_V* – A 1-bit flag indicating whether the component has a valid checksum. If this field is zero, the value in the *Chksum* field is not valid.
- *Chksum* – A 1-byte field containing the component's checksum. The modulo sum of all the bytes in the component and the value in this field (Chksum) must add up to zero. This field is only valid if the *C_V* flag is non-zero. If the checksum option is selected for the FIT, in the FIT Header entry (FIT type 0), the modulo sum of all the bytes in the FIT table must add up to zero.

  Note: The PAL_A FIT entry is not part of the FIT table checksum.
- *Address* – An 8-byte field containing the base address of the component. For the FIT header, this field contains the ASCII value of "_FIT_<sp><sp><sp>" (<sp> represents the space character).

The FIT allows simpler firmware updates. Different components may be updated independently. This address layout can also support firmware images spanning multiple storage devices. FIT entries must be arranged in ascending order by the *type* field, otherwise execution of firmware code will be unpredictable.

11.2 PAL Power On/Reset

11.2.1 PALE_RESET

The purpose of PALE_RESET is to initialize and test the processor. Upon receipt of a power-on/reset event the processor begins executing code from the PALE_RESET entrypoint in the firmware address space. PALE_RESET initializes the processor and may perform a minimal processor self test. PAL may optionally perform authentication of the PAL firmware to ensure data integrity. If the authentication code runs cacheable by default, then a processor-specific mechanism will be provided to disable caching for diagnostic purposes.

PALE_RESET then branches to SALE_ENTRY to determine if a recovery condition exists, which would require an update of the firmware. If it does, SALE_ENTRY performs the update and resets the system. If no firmware recovery is needed, SAL returns to PALE_RESET to perform the processor self-tests and initialization. SAL can control the length and coverage of the PAL processor self-test by examining and modifying the self-test control word passed to SAL at the firmware recovery hand-off state. Please see Section 11.2.3, “PAL Self-test Control Word” for more information on the self-test control word.

The PAL processor self-tests are split into two phases. The first phase is written to test processor features that do not require external memory to be present to execute correctly. These tests are automatically run when SAL returns to PAL after the branch to SALE_ENTRY for a firmware recovery check. This section is referred to as phase one of processor self-test and they are generally run early during the processor boot process. The second phase is written requiring that external memory is available to execute correctly. These tests are run when a call to the PAL procedure PAL_TEST_PROC is made with the correct parameters set up. These tests are referred to as phase two of processor self-test since they are usually run later in the processor boot process after external memory has been initialized on the platform.

PAL may execute IA-32 instructions to fully test and initialize the processor. This IA-32 code will not generate any special IA-32 bus transactions nor will it require any special platform features to correctly execute. PAL then branches to SALE_ENTRY to conduct platform initialization and testing before loading the operating system software.

11.2.2 PALE_RESET Exit State

- GRs: The contents of all general registers are undefined except the following:
  - GR20 (bank 1) contains the SALE_ENTRY State Parameter as defined in Figure 11-7. For the function field of the SALE_ENTRY State Parameter, only the values 3, RECOVERY CHECK, for the first call to SALE_ENTRY, and 0, RESET, for the second call to SALE_ENTRY are valid.
  - GR32 contains 0 indicating that SALE_ENTRY was entered from PALE_RESET.
  - GR33 contains information about the geographically significant unique processor ID, and a mask that indicates which bits in the LID register (CR64) are read-only. Firmware should write the processor's local interrupt identifier in the programmable portion of the LID register. Writes to the read-only bits are ignored. See Figure 11-8 for the definition of this parameter.

  - GR34 contains the physical address for making a PAL procedure call. If the call is for RECOVERY CHECK, only the subset of PAL procedures needed for SALE_ENTRY to perform firmware recovery will be available. These procedures are:
    - PAL_FREQ_RATIOS
    - PAL_LOGICAL_TO_PHYSICAL
    - PAL_PLATFORM_ADDR
    - An implementation-specific PAL procedure for PAL authentication.
  - GR35 contains the Self Test State Parameter as defined in Figure 11-9.
  - GR36 contains the PAL_RESET return address for SALE_ENTRY to return to if a recovery condition does not exist. When PAL_RESET calls SALE_ENTRY the second time to initialize the system for operating system use, this register will contain the physical address for making an implementation-specific PAL procedure call for PAL authentication.

    Note: For all other PAL procedure calls, the physical address at GR34 should be used.
  - GR37 contains the self-test control word as defined in Figure 11-10. This control word is processor implementation-specific and informs SAL if self-test control is implemented and the number of controllable bits. If self-test control is implemented, PAL will read this value when SAL returns to PAL after firmware recovery check. If the self-test control is not supported, this register will be ignored when SAL returns to PAL after firmware recovery check.
  - GR38 – Indicates if the PAL_MEMORY_BUFFER procedure is required to be called on this processor implementation for correct behavior. Also indicates the minimum buffer size required for the PAL_MEMORY_BUFFER procedure. Table 11-2 defines the layout of this register.

Table 11-2. GR38 Reset Layout

Bit Field	Description
31:0	Unsigned integer denoting the minimum number of bytes required by the PAL_MEMORY_BUFFER procedure.
32:62	Reserved
63	Indicates if the PAL_MEMORY_BUFFER procedure is required by this processor implementation. A value of 1 indicates that it is required, a value of 0 indicates that it is not required.

  - Banked GRs: All bank 0 general registers are undefined.
- FRs: The contents of all floating-point registers are undefined. The floating-point registers are enabled unless the *state* field of the Self Test State Parameter is FUNCTIONALLY RESTRICTED and the floating-point unit failed self test. Then, the floating-point registers are disabled. Refer to Section 11.2.2.3, "Definition of Self Test State Parameter" for the definition of FUNCTIONALLY RESTRICTED.
- Predicates: The contents of all predicate registers are undefined.
- BRs: The contents of all branch registers are undefined.
- ARs: The contents of all application registers are undefined except the following:
  - RSC: All fields in the register stack configuration register are 0, which places the RSE in enforced lazy mode.
- CFM: The CFM is set up so that all stacked registers are accessible, CFM.sof = 96 and all other CFM fields are 0.

- PSR: PSR.bn is 1; PSR.df1 and PSR.dfh are 1 if the floating-point unit failed self test. All other PSR bits are 0. PSR.ic and PSR.i are zero to ensure external interrupts, NMI and PMI interrupts are disabled.
- CRs: The contents of all control registers are undefined except the following:
    - DCR: contains the value 0.
    - IVA: contains the physical address of an interruption vector table previously set up by PAL. SAL may choose to change this value. The IVA will be 0 when the SALE_ENTRY State Parameter function is RECOVERY CHECK.
- RRs: The contents of all region registers are undefined.
- PKRs: The contents of all protection key registers are undefined.
- DBRs: The contents of all data breakpoint registers are undefined
- IBRs: The contents of all instruction breakpoint registers are undefined.
- PMCs: The contents of all performance monitor control registers are undefined.
- PMDs: The contents of all performance monitor data registers are undefined.
- Cache: The processor internal caches are enabled and invalidated. Unless directed otherwise by the self-test control word, phase one of the processor self-test verifies the caches themselves and the paths from the caches to the processor core. The path from external memory to the caches cannot be tested until phase two of the processor self-test.

    Note: All cache contents will be invalidated when SAL returns to PAL after the RECOVERY_CHECK hand-off. If the SAL uses the caches in their RECOVERY_CHECK code, it is SAL's responsibility to write back any modified data in the caches before returning to PAL

- TLB: The TRs and TCs are initialized with all entries having been invalidated. The TLB is disabled because PSR.it=PSR.dt=PSR.rt=0. The TLBs cannot be fully tested until phase two of the processor self-test.

Prior to passing control to SALE_ENTRY, PALE_RESET must ensure that the processor Interrupt block pointer is set to point to address 0x0000_0000_FEE0_0000.

11.2.2.1 Definition of SALE_ENTRY State Parameter

Figure 11-7. SALE_ENTRY State Parameter

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">reserved</td><td colspan="8">status</td><td colspan="8">function</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="8">min-state_size</td><td colspan="24">reserved</td></tr>
</table>

- *function* – An 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-3. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition

All other values of *function* are reserved.

- *status* – A function-dependent 8-bit field indicating the firmware status on entry to SALE_ENTRY. If the function value is RESET or RECOVERY_CHECK, the *status* values are:

Table 11-4. *status* Field Values

Status	Value	Description
Normal	0	Normal reset.
FIT Header Failure	1	FIT header for FIT and alternate FIT (if supported) is incorrect
FIT Checksum Failure	2	FIT checksum for FIT and alternate FIT (if supported) is incorrect
PAL_B Checksum Failure	3	PAL_B checksum (for all compatible PAL_B's found) is incorrect
PAL_A Authentication Failure	4	PAL_A (generic in split model) failed authentication
PAL_B Authentication Failure	5	PAL_B (for all compatible PAL_B's found) failed authentication
PAL_B Not Found	6	FIT Entry for PAL_B missing from the FIT and alternate FIT (if supported)
Incompatible	7	No PAL_B was found in the FIT and alternate FIT (if supported) that is compatible with the processor stepping
32K Unaligned	8	No PAL_B was found in the FIT and alternate FIT (if supported) that was correctly aligned to a 32KB boundary
PAL_A_Spec Not Found / FIT Checksum Failure	9	No compatible processor-specific PAL_A was found in the FIT because of a FIT checksum failure and no compatible processor-specific PAL_A was found in the alternate FIT (if supported)
PAL_A_Spec Found / FIT Checksum Failure	10	A compatible processor-specific PAL_A was found in the alternate FIT. No compatible processor-specific PAL_A was found in the FIT due to a FIT checksum failure.
PAL_A_Spec Failure / Good PAL_A_Spec found in FIT	11	One or more compatible processor-specific PAL_A's found in the FIT failed its checksum or authentication. Another compatible processor-specific PAL_A was found in the FIT that passed its checksum and authentication.
PAL_A_Spec Auth Failure	12	No compatible processor-specific PAL_A's were found in the FIT or alternate FIT (if supported) that passed its checksum and authentication
PAL_A_Spec Auth Failure / Good PAL_A_Spec found in AF	13	One or more compatible processor-specific PAL_A's found in the FIT or alternate FIT (if supported) failed its checksum and authentication. Another compatible processor-specific PAL_A was found in the alternate FIT that passed its checksum and authentication.
PAL_A_Spec Not Found	14	No compatible processor-specific PAL_A was found in the FIT or alternate FIT (if supported)
PAL_A_Spec Not Found in FIT / Good PAL_A_Spec found in AF	15	No compatible processor-specific PAL_A was found in the FIT. A compatible processor-specific PAL_A was found in the alternate FIT.

Table 11-4. *status* Field Values (Continued)

Status	Value	Description
PAL_B Auth Failure / Good PAL_B found	16	One or more compatible PAL_B's failed authentication and checksum. Another compatible PAL_B was found that passed authentication and checksum.
64K Unaligned	17	No PAL_B was found in the FIT and alternate FIT (if supported) that was correctly aligned to a 64KB boundary.

All other values of *status* are reserved.

Definitions of *status* values for other values of *function* are listed in the machine check and init sections.

For the case of RECOVERY CHECK, authentication of PAL_A and PAL_B should be completed before call to SALE_ENTRY.

- *min-state_size* – An 8-bit field indicating the size in kilobytes (KB) of the min-state save area required for this implementation. A value of zero indicates a size of 4KB. A value greater than zero indicates the actual size in KB of the min-state save area required for this implementation. Values of 1-4 are reserved. For more information about the min-state save area, please refer to Section 11.3.2.4, "Processor Min-state Save Area Layout" on page 2:302.

11.2.2.2 Definition of Geographically Significant Processor Identifier Parameter

Figure 11-8. Geographically Significant Processor Identifier

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved																proc_id															

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved																eid_mask								id_mask							

Table 11-5. Geographically Significant Processor Identifier Fields

Field	Bits	Description
proc_id	15:0	Geographically significant processor ID. The value returned in this field is the same as that returned by PAL_FIXED_ADDR.
Reserved	31:16	Reserved
id_mask	39:32	Mask indicating which bits in *id* are programmable: 0 = Programmable 1 = Read-only
eid_mask	47:40	Mask indicating which bits in *eid* are programmable: 0 = Programmable 1 = Read-only
Reserved	63:48	Reserved

11.2.2.3 Definition of Self Test State Parameter

Figure 11-9. Self Test State Parameter

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved												mf	fp	ia	vm	reserved													te	state	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
test_status																															

- *state* – A 2-bit field indicating the state of the processor after self-test. If SAL directed PAL to skip some self-tests by modifying the self-test control word, failures related to these self-tests will not be reflected in this state.

Table 11-6. *state* Field Values

State	Value	Description
Catastrophic Failure	N/A	The processor is not capable of continuing. In this case it does not branch to SALE_ENTRY.
Healthy	00	No hardware failures have occurred in testing that would affect either the performance or functionality of the processor.
Performance Restricted	01	A hardware failure has occurred in testing that does not affect the functionality of the processor, but performance may be degraded.
Functionally Restricted	10	A hardware failure has occurred in testing that affects the functionality of the processor, but firmware code can still be run. The processor may also be performance restricted.

To further qualify FUNCTIONALLY RESTRICTED, the following requirements will be met:

  - The processor has detected and isolated the failing component so that it will not be used.
  - The processor must have at least one functioning memory unit, ALU, shifter, and branch unit.
  - The floating-point unit may be disabled.
  - The RSE is not required to work, but register renaming logic must work properly.
  - The paths between the processor controlled caches and the register files have been shown to work. The path between the processor caches and memory cannot be validated until phase two of the processor self-test invoked by the PAL_TEST_PROC procedure.
  - Loads and stores to firmware address space must work correctly.

Additional information about the failure can be obtained by examining the *test_status* field of the *Self Test State Parameter*.

For the case of FUNCTIONALLY RESTRICTED, it is required that higher level firmware or OS not use failing functional units during their execution. PAL will not prevent failing functional units from being used.

- *te* – A 1-bit field indicating whether testing has occurred. If this field is zero, the processor has not been tested, and no other fields in the *Self Test State Parameter* are valid. The processor can be tested prior to entering SALE_ENTRY for both RECOVERY CHECK and RESET functions.

  If the *state* field indicates that the processor is functionally restricted, then the fields *vm, ia* & *fp* specify additional information about the functional failure.

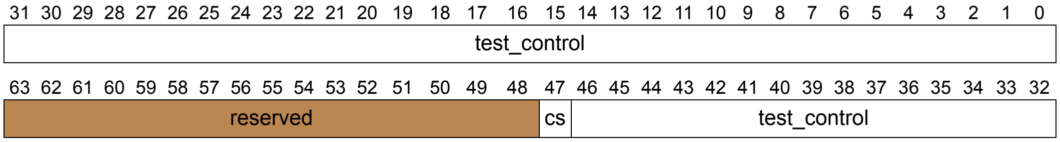
  - *vm* – a 1-bit field, if set to 1, indicating that virtual memory features are not available
  - *ia* – a 1-bit field, if set to 1, indicating that IA-32 execution is not available
  - *fp* – a 1-bit field, if set to 1, indicating that floating-point unit is not available
  - *mf* – a 1-bit field, if set to 1, indicating miscellaneous functional failure other than *vm, ia,* or *fp*. The *test_status* field provides additional information about this failure on an implementation-specific basis.

- *test_status* – An unsigned 32-bit-field providing additional information on test failures when the *state* field returns a value of PERFORMANCE RESTRICTED or FUNCTIONALLY RESTRICTED. The value returned is implementation dependent.

11.2.3 PAL Self-test Control Word

The PAL self-test control word is a 48-bit value. This bit field is defined in Figure 11-10.

Figure 11-10. Self-test Control Word



- *test_control* – This is an ordered implementation-specific control word that allows the user control over the length and runtime of the processor self-tests. This control word is ordered from the longest running tests up to the shortest running tests with bit 0 controlling the longest running test.

  PAL may not implement all 47-bits of the *test_control* word. PAL communicates if a bit provides control by placing a zero in that bit. If a bit provides no control, PAL will place a one in it.

  PAL will have two sets of *test_control* bits for the two phases of the processor self-test.

  PAL provides information about implemented *test_control* bits at the hand-off from PAL to SAL for the firmware recovery check. These *test_control* bits provide control for phase one of processor self-test. It also provides this information via the PAL procedure call PAL_TEST_INFO for both the phase one and phase two processor tests depending on which information the caller is requesting.

  PAL interprets these bits as input parameters on two occasions. The first time is when SAL passes control back to PAL after the firmware recovery check. The second time is when a call to PAL_TEST_PROC is made. When PAL interprets these bits it will only interpret implemented *test_control* bits and will ignore the values located in the unimplemented *test_control* bits.

  PAL interprets the implemented bits such that if a bit contains a zero, this indicates to run the test. If a bit contains a one, this indicates to PAL to skip the test.

  If the *cs* bit indicates that control is not available, the *test_control* bits will be ignored or generate an illegal argument in procedure calls if the caller sets these bits.

- *cs* – Control Support: This bit defines if an implementation supports control of the PAL self-tests via the self-test control word. If this bit is 0, the implementation does not support control of the processor self-tests via the self-test control word. If this bit is 1, the implementation does support control of the processor self-tests via the self-test control word.

  If control is not supported, GR37 will be ignored at the hand-off between SAL and PAL after the firmware recovery check and the PAL procedures related to the processor self-tests may return illegal arguments if a user tries to use the self-test control features.

11.3 Machine Checks

11.3.1 PALE_CHECK

When a machine check abort (MCA) occurs, PALE_CHECK is responsible for saving minimal processor state to a uncacheable platform-specific memory location previously registered with PAL via the PAL_MC_REGISTER_MEM procedure. This platform location is called the Minimal State Save Area (min-state save area) and is described in Section 11.3.2.4, "Processor Min-state Save Area Layout" on page 2:302. PALE_CHECK is also responsible for correcting processor related errors whenever possible. PALE_CHECK terminates either by returning to the interrupted context or by branching to SALE_ENTRY, passing the state of the processor at the time of the error. The level of recovery provided by PALE_CHECK is implementation dependent and is beyond the scope of this specification.

At the hand-off from PALE_CHECK to SALE_ENTRY, error information is passed in the Processor State Parameter described in Section 11.3.2.1, "Processor State Parameter (GR 18)" on page 2:299. After exit from PALE_CHECK, more detailed error information is available by calling the PAL_MC_ERROR_INFO procedure. Information about implementation-dependent state is available by calling the PAL_MC_DYNAMIC_STATE procedure. The interrupted process may be resumed by calling the PAL_MC_RESUME procedure. See Section 11.3.3, "Returning to the Interrupted Process" for more information on returning to the interrupted context and Section 11.10, "PAL Procedures" on page 2:353 for detailed descriptions of all these procedure calls.

Code for handling machine checks must take into consideration the possibility that nested machine checks may occur. A nested machine check is a machine check that occurs while a previous machine check is being handled.

PALE_CHECK is entered in the following conditions:

- When PSR.mc = 0 and an error occurs which results in a machine check, or
- When PSR.mc changes from 1 to 0 and there is a pending machine check from an earlier error.

PSR.mc is set to 1 by the hardware when PALE_CHECK is entered. When PALE_CHECK branches to SALE_ENTRY, PSR.mc remains set (PSR.mc is restored to its original value if PALE_CHECK terminates by returning to the interrupted context). SAL must not clear PSR.mc to 0 before all the information from the current machine check is logged. If SAL enables machine checks (by setting PSR.mc=0) during the SAL MCA handling, there is a potential for the error logs in the processor and the min-state save area to be overwritten by a subsequent MCA event.

The error information logged will reflect the state at the time the error occurred. State information from a different point in time will NOT be logged. If complete information is not available a code is logged which indicates that the information is not available.

- The processor state information used to resume a process for which an error has been corrected will reflect the state at the time the machine check interruption occurred and will be sufficient to resume the interrupted process.
- When a single error is signalled multiple times (for example, multiple operations to a single bad cache line), hardware and firmware will be able to perform the same logging and recovery as if the error had been signalled once.

For testing and configuration purposes, it may be necessary for software to intentionally generate a machine check. In this case PALE_CHECK will log the error information, but not attempt recovery before branching to SALE_ENTRY. To allow for this, the PAL_MC_EXPECTED procedure call is defined to indicate that PALE_CHECK should not to attempt recovery.

11.3.1.1 Resources Required for Machine Check and Initialization Event Recovery

While the level of recovery from machine checks is implementation dependent, for each particular level of recovery there is a set of architecturally required resources. The following paragraphs define the required and optional resources needed to support firmware and software recovery of machine checks and initialization events.

- Minimal resources required to allow software recovery of machines checks when PSR.ic=1:
  - XR0 register: memory pointer to min-state save area previously registered with PAL via the PAL_MC_REGISTER_MEM procedure. The layout of this memory area is described in Section 11.3.2.4, “Processor Min-state Save Area Layout” on page 2:302.
  - Bank zero registers GR 24 through GR 31. These registers are not preserved across interruptions and may be used as scratch registers by machine check recovery code. See Section 3.3.7, “Banked General Registers” on page 2:42 for the definition of the bank 0 registers.
- Additional resources required to allow software recovery of machine checks when PSR.ic=0. The presence of these resources is processor implementation specific. The PAL_PROC_GET_FEATURES procedure described on page 2:440 returns information on the existence of these optional resources.
  - XIP, XPSR, XFS: interruption resources implemented to store information about the IIP, IPSR and IFS when the machine check occurred. A model-specific version of the `rfi` instruction must also be implemented to restore the machine context from these resources.
  - XR1-XR3: scratch registers implemented to preserve bank 0 GR 24 through GR 31.

Each of the registers described above should be accessed only by PAL in order to support firmware and software recovery of machine checks.

11.3.2 PALE_CHECK Exit State

The state of the processor on exiting PALE_CHECK is listed below. For registers described as being saved to the min-state save area and available for use, the actual values in these registers are undefined unless specifically stated otherwise.

- GRs: The contents of all non-banked static registers (GR1-GR15), bank zero static registers and bank one static registers (GR16-31) at the time of the MCA have been saved in the min-state save area and are available for use.
  - If recovery is not supported when PSR.ic=0 then GR24 - GR31 (bank 0) are undefined and their contents have been lost. In this case, recovery is not possible. See Section 11.3.1.1, “Resources Required for Machine Check and Initialization Event Recovery” for details.

  - GR16 through GR20 (bank 0) contain parameters which PALE_CHECK passes to SALE_ENTRY for diagnostic and recovery purposes:
    - GR16 contains the address to the first available location in the min-state save area for use by SAL. The address is 8-byte aligned.
    - GR17 contains the value of the min-state save area address stored in XR0.
    - GR18 contains the Processor State Parameter, as defined in Figure 11-11.
    - GR19 contains the PALE_CHECK return address for rendezvous, or 0 if no return is expected. (See Section 11.3.2.2, “Multiprocessor Rendezvous Requirements for Handling Machine Checks”)
    - GR20 contains the SALE_ENTRY State Parameter as defined in Figure 11-4.
- FRs: The contents of all floating-point registers are unchanged from the time of the MCA.
- Predicates: All predicate registers have been saved in the min-state save area and are available for use.
- BRs: The contents of all branch registers are unchanged from the time of the MCA, except the following.
  - BR0 and BR1 have been saved to the min-state save area and are available for use. Either register may have been changed from the time of entry into PALE_CHECK.
- ARs: The contents of all application registers are unchanged from the time of the MCA, except the RSE control register (RSC), the RSE backing store pointer (BSP), and the ITC and RUC counters. The RSC register is unchanged, except that the RSC.mode field will be set to 0 (enforced lazy mode) and the RSC register at the time of the MCA has been saved in the min-state save area. A cover instruction is executed in the PALE_CHECK handler which allocates a new stack frame of zero size. BSP will be modified to point to a new location, since all the registers from the current frame at the time of interruption were added to the RSE dirty partition by the allocation of a new stack frame. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the MCA handler. The RUC register will not be directly modified by PAL, but will continue to count during the execution of the MCA handler while the processor is active.
- CFM: The CFM register points to a zero-size current frame and all the rotating register bases are set to zero. The CFM register at the time of the MCA has been saved to the min-state save area in either the IFS or XFS slot depending on the implementation.
- RSE: Is in enforced lazy mode, and stacked registers are unchanged from the time of the MCA.
- PSR: PSR.mc is 1; PSR.mfl, PSR.mfh, and PSR.pk are unchanged; all other bits are 0. The PSR at the time of the MCA is saved in the min-state save area.
- CRs: The contents of all control registers are unchanged from the time of the MCA with the exception of interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the MCA.
- PKRs: The contents of all protection key registers are unchanged from the time of the MCA.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the MCA.
- PMCs/PMDs: The contents of the PMC registers are unchanged from the time of the MCA. The contents of the PMD registers are not modified by PAL code, but may be modified if events it is monitoring are encountered.

- Cache: The processor internal cache is enabled and is unchanged from the time of the MCA except for any lines that were invalidated to correct the error.
- TLB: The TCs may be initialized and the TRs are unchanged from the time of the MCA.
- Interruption Resources:
  - IRR: PALE_CHECK may not change the IRR, but interrupts may have arrived asynchronously, changing the contents of the IRRs.
  - The contents of IIP, IPSR and IFS at the time of the MCA are saved to the min-state save area and are available for use.

11.3.2.1 Processor State Parameter (GR 18)

Figure 11-11. Processor State Parameter

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
gr	b0	b1	fp	pr	br	ar	rr	tr	dr	pc	cr	ex	cm	rs	in	dy	pm	pi	mi	tl	hd	us	ci	co	sy	mn	me	ra	rz	rsvd	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
uc	rc	bc	tc	cc	reserved										se	dsize															

The term "valid" in Table 11-7 indicates that the registers are either unchanged from the time of interruption or that the values have been preserved in the min-state save area.

Table 11-7. Processor State Parameter Fields

Field	Bits	Description
rsvd	1:0	Reserved
rz	2	The attempted processor rendezvous was successful if set to 1.
ra	3	A processor rendezvous was attempted if set to 1.
me	4	Distinct multiple errors have occurred, not multiple occurrences of a single error. Software recovery may be possible if error information has not been lost.
mn	5	Min-state save area has been registered with PAL if set to 1.
sy	6	Storage integrity synchronized. A value of 1 indicates that all loads and stores prior to the instruction on which the machine check occurred completed successfully, and that no loads or stores beyond that point occurred. See Table 11-8.
co	7	Continuable. A value of 1 indicates that all in-flight operations from the processor where the machine check occurred were either completed successfully (such as a load), were tagged with an error indication (such as a poisoned store), or were suppressed and will be re-issued if the current instruction stream is restarted. This bit can only be set if the architectural state saved on a machine check is all valid. If this bit is set, then *us* must be cleared to 0, and *ci* must be set to 1. See Table 11-8.
ci	8	Machine check is isolated. A value of 1 indicates that the error has been isolated by the system, it may or may not be recoverable. If 0, the hardware was unable to isolate the error within the CPU and memory hierarchy. The error may have propagated off the system (to persistent storage or the network). If *ci* = 0 then *us* will be set to 1, and *co* and *sy* are cleared to 0. See Table 11-8.
us	9	Uncontained storage damage. A value of 1 indicates the error is contained within the CPU and memory hierarchy, but that some memory locations may be corrupt. If *us* is set to 1, then *co* and *sy* will always be cleared to 0. See Table 11-8.
hd	10	Hardware damage. A value of 1 indicates that as a result of the machine check some non essential hardware is no longer available causing this processor to execute with degraded performance (no functionality has been lost).

Table 11-7. Processor State Parameter Fields (Continued)

Field	Bits	Description
tl	11	Trap lost. A value of 1 indicates the machine check occurred after an instruction was executed but before a trap that resulted from the instruction execution could be generated.
mi	12	More information. A value of 1 indicates that more error information about the machine check event is available by making the PAL_MC_ERROR_INFO procedure call.
pi	13	Precise instruction pointer. A value of 1 indicates that the machine logged the instruction pointer to the bundle responsible for generating the machine check.
pm	14	Precise min-state save area. A value of 1 indicates that the min-state save area contains the state of the machine for the instruction responsible for generating the machine check. When this bit is set, the *pi* bit will always be set as well.
dy	15	Processor Dynamic State is valid. (1=valid, 0=not valid) See the PAL_MC_DYNAMIC_STATE procedure call for more information.
in	16	Interruption caused by INIT. (0=machine check, 1=INIT)
rs	17	The RSE is valid. (1=valid, 0=not valid)
cm	18	The machine check has been corrected. (1=corrected, 0=not corrected)
ex	19	A machine check was expected. (1=expected, 0=not expected)
cr	20	Control registers are valid. (1=valid, 0=not valid)
pc	21	Performance counters are valid. (1=valid, 0=not valid)
dr	22	Debug registers are valid. (1=valid, 0=not valid)
tr	23	Translation registers are valid. (1=valid, 0=not valid)
rr	24	Region registers are valid. (1=valid, 0=not valid)
ar	25	Application registers are valid. (1=valid, 0=not valid)
br	26	Branch registers are valid. (1=valid, 0=not valid)
pr	27	Predicate registers are valid. (1=valid, 0=not valid)
fp	28	Floating-point registers are valid. (1=valid, 0=not valid)
b1	29	Preserved bank one general registers are valid. (1=valid, 0=not valid)
b0	30	Preserved bank zero general registers are valid. (1=valid, 0=not valid)
gr	31	General registers are valid. (1=valid, 0=not valid) (does not include banked registers)
dsize	47:32	Size in bytes of Processor Dynamic State returned by PAL_MC_DYNAMIC_STATE.
se	48	Shared Error. Machine check corresponds to structure shared by multiple logical processors.
rsvd	58:49	Reserved
cc	59	Cache check. A value of 1 indicates that a cache related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information. This bit must not be set for non-cacheable transaction errors.
tc	60	TLB check. A value of 1 indicates that a TLB related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
bc	61	Bus check. A value of 1 indicates that a bus related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
rc	62	Register file check. A value of 1 indicates that a register file related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
uc	63	Uarch check. A value of 1 indicates that a micro-architectural related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

11.3.2.1.1 Using Processor State Parameter to Determine if Software Recovery of a Machine Check is Possible

The *us*, *ci, co, and sy* bits in the Processor State Parameter are valid only if the error has not been previously corrected in hardware or firmware (*cm* bit is 0). Even then, only the bit combinations shown in Table 11-8 are valid. If the multiple error bit is set (*me*=1) both the *co* and *sy* bits must be 0. The *us* and *ci* bits will be set according to the worst case of the errors that occurred.

Table 11-8. Software Recovery Bits in Processor State Parameter

cm	us	ci	co	sy	Description
1	x	x	x	x	The machine check is corrected. The *us, ci, co,* and *sy* bits are not valid.
0	1	0	0	0	The error was not isolated. Software must reset system. Data on disk may be corrupt.
0	1	1	0	0	The error was isolated but not contained. Corrupt data was not written to I/O, but may remain in the CPU or memory untagged. Software must reset system.
0	0	1	0	0	The error was isolated and contained, but is not continuable. The current instruction stream cannot be restarted without loss of information. Partial recovery may be possible.
0	0	1	1	0	The error was isolated, contained, and is continuable. If software can correct the error the current instruction stream can be continued with no loss of information.
0	0	1	1	1	The error was isolated, contained, and is continuable. The instruction pointer points to the instruction where the error occurred. If software can correct the error the current instruction stream can be continued with no loss of information.

11.3.2.2 Multiprocessor Rendezvous Requirements for Handling Machine Checks

When PALE_CHECK has determined that an error has occurred which could cause a multiprocessor system to lose error containment, it must rendezvous the other processors in the system before proceeding with further processing of the machine check. This is accomplished by branching to SALE_ENTRY with a non-zero return vector address in GR19. It is then the responsibility of SAL to rendezvous the other processors and return to PALE_CHECK through the address in GR19. If the rendezvous was successful GR19 must be set to 0 before return.

At the time PALE_CHECK makes the rendezvous call to SALE_ENTRY, the processor state is exactly the same as defined in See “PALE_CHECK Exit State” on page 2:297. with the following requirement on the use of registers by SAL:

Any processor state not listed below must be either unchanged or restored by SAL before returning to PALE_CHECK.

- SAL will preserve the values in GR4-GR7 and GR17-GR18.
- SAL will return to PALE_CHECK via the address in GR19.
- SAL will set up GR19 to indicate the success of the rendezvous before returning to PAL.
  - GR19 is zero to indicate the rendezvous was successful.
  - GR19 is non zero to indicate that the rendezvous was unsuccessful.
- All other non-banked (GR1-3, GR8-15), bank 0 GRs (GR20-GR31) and BR0 are undefined and available for use by SAL.

After return from the SAL rendezvous call, PALE_CHECK will complete processing the machine check if the rendezvous was successful and then branch to SALE_ENTRY with GR19 set to zero. The processor state when transferring to SAL is as defined in Section 11.3.2, “PALE_CHECK Exit State” on page 2:297. If the rendezvous failed PALE_CHECK will simply construct the Processor State Parameter and branch to SALE_ENTRY.

Any further discussion of multiprocessor rendezvous, including platform requirements and implications, is beyond the scope of this specification. See the relevant SAL/Error handling documents for further information.

11.3.2.3 Unconsumed Data-Poisoning Event Handling

If, during the transfer/access of information between levels of the cache/memory hierarchy, there is data found to have an uncorrectable error and is marked poison, error reporting events may be raised. If such an error event is sent to a processor that doesn't consume the corrupted data, then the error is termed an **unconsumed data-poisoning event**.

Unconsumed data-poisoning events are by default reported as a CMC and can optionally be promoted to an MCA via bit 53 of *feature_set* 0 of PAL_PROC_SET_FEATURES. When they are signaled as a CMC the PSP.cm is set to 1 to indicate that the error has been corrected (in the sense that the line has been marked poison, preventing any silent data corruption).

If bit 53 is 1, unconsumed data-poisoning events are reported as MCAs. To immediately report unconsumed data-poisoning events as **uncorrected errors** (in the sense that the data in question has been lost), the caller can set bit 53 to 1. PSP settings for a data-poisoning event with bit 53 equal to 1 are given in the table below. See also Table 11-8.

Table 11-9. PSP Bit Settings for Unconsumed Data-poisoning Events on MCA

cm	us	ci	co	sy
0	0	1	1	0

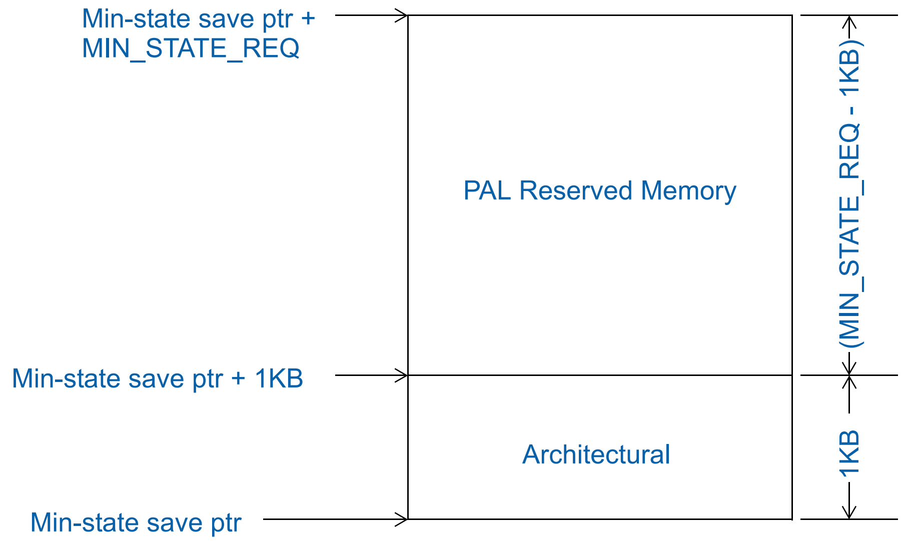
When promotion is enabled (bit 53 is 1), and a continuable data-poisoning event is indicated (i.e., the PSP bits are set as in the above table, and either cache_check.dp, bus_check.dp or both are 1), and if no other MCAs occur at the same time (i.e., no other errors are indicated in the error information from PAL_MC_ERROR_INFO), the interrupted process is always continuable. Promotion to MCA with bit 53 allows the OS to take proactive measures to recover from the poisoned data, but this is not required for the interrupted process to be continuable.

11.3.2.4 Processor Min-state Save Area Layout

The processor min-state save area is minimally 4KB in size, but an implementation may require larger sizes. The reset hand-off state indicates if a size greater than 4KB is required and also provides the required size. Please refer to Section 11.2.2.1, “Definition of SALE_ENTRY State Parameter” on page 2:291 for more information on the reset hand-off state. The required size is referred to as MIN_STATE_REQ. The min-state save area is required to be in an uncacheable region. The first 1KB of this area is architectural state needed by the PAL code to resume during MCA and INIT events (architected min-state save area + reserved). The remaining space in the buffer is a scratch space reserved exclusively for PAL use, therefore SAL and OS must not use this area. The layout of the processor min-state save area is shown in Figure 11-1.

The processor min-state save area is 4KB in size and must be in an uncacheable region. The first 1KB of this area is architectural state needed by the PAL code to resume during MCA and INIT events (architected min-state save area + reserved). The remaining 3KB is a scratch buffer reserved exclusively for PAL use, therefore SAL and OS must not use this area. The layout of the processor min-state save area is shown in Figure 11-1.

Figure 11-1. Processor Min-state Save Area Layout



The layout for the processors portion of the architectural 1KB processor min-state save area is shown in Figure 11-2. When SAL registers the area with PAL, it passes in a pointer to offset zero of the area. When PALE_CHECK is entered as a result of a machine check, it fills in processor state, processes the machine check, and branches to SALE_ENTRY with a pointer to the first available memory location that SAL can use in GR16. SAL may allocate a variable sized area above the address passed in GR16 up to the 1KB architectural limit, but this is internal to SAL and not known to PAL.

The base address of the min-state save area must minimally be aligned to a 512-byte boundary, but larger alignments are allowed. All saves and restores to and from the min-state save area are made using 8-byte wide load and store instructions. If the processor min-state save area is not registered via the PAL_MC_REGISTER_MEM procedure prior to the machine check, software recovery is not possible.

Figure 11-2. Processor State Saved in Min-state Save Area

0xf8	Bank 0 GR31			
0xf0	Bank 0 GR30			
0xe8	Bank 0 GR29			
0xe0	Bank 0 GR28			
0xd8	Bank 0 GR27			← GR16
0xd0	Bank 0 GR26	≈	≈	
0xc8	Bank 0 GR25	0x1c8	BR1	
0xc0	Bank 0 GR24	0x1c0	XFS or undefined	
0xb8	Bank 0 GR23	0x1b8	XPSR or undefined	
0xb0	Bank 0 GR22	0x1b0	XIP or undefined	
0xa8	Bank 0 GR21	0x1a8	IFS	
0xa0	Bank 0 GR20	0x1a0	IPSR	
0x98	Bank 0 GR19	0x198	IIP	
0x90	Bank 0 GR18	0x190	RSC	
0x88	Bank 0 GR17	0x188	BR0	
0x80	Bank 0 GR16	0x180	Predicate Registers	
0x78	GR15	0x178	Bank 1 GR31	
0x70	GR14	0x170	Bank 1 GR30	
0x68	GR13	0x168	Bank 1 GR29	
0x60	GR12	0x160	Bank 1 GR28	
0x58	GR11	0x158	Bank 1 GR27	
0x50	GR10	0x150	Bank 1 GR26	
0x48	GR9	0x148	Bank 1 GR25	
0x40	GR8	0x140	Bank 1 GR24	
0x38	GR7	0x138	Bank 1 GR23	
0x30	GR6	0x130	Bank 1 GR22	
0x28	GR5	0x128	Bank 1 GR21	
0x20	GR4	0x120	Bank 1 GR20	
0x18	GR3	0x118	Bank 1 GR19	
0x10	GR2	0x110	Bank 1 GR18	
0x8	GR1	0x108	Bank 1 GR17	
0x0	NaT bits for saved GRs	0x100	Bank 1 GR16	

The NaT bits stored in the first entry of the min-state save area have the following layout.

Figure 11-3. NaT Bits for Saved GRs

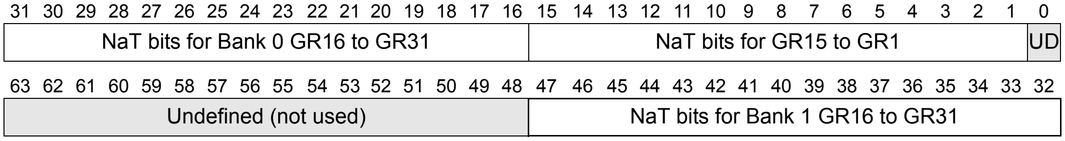


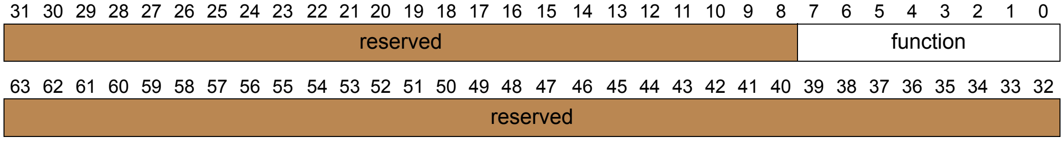
Table 11-10. NaT Bits for Saved GRs

Bits	Description
0	Undefined (not used)
15:1	NaT bits for GR15 to GR1. Bit 1 represents GR1 and subsequent bits follow the ascending pattern.
31:16	NaT bits for Bank 0 GR16 to GR31. Bit 16 represents Bank 0 GR16 and subsequent bits follow the ascending pattern.
47:32	NaT bits for Bank 1 GR16 to GR31. Bit 32 represents Bank 1 GR16 and subsequent bits follow the ascending pattern.
63:48	Undefined (not used)

The value passed in GR16 to SAL may point beyond the defined processor state shown in Figure 11-2. PAL may use this area for implementation-dependent processor state that needs to be saved and restored.

11.3.2.5 Definition of SALE_ENTRY State Parameter

Figure 11-4. SALE_ENTRY State Parameter



- *function* – An 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-11. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition in SAL

All other values of *function* are reserved.

11.3.3 Returning to the Interrupted Process

The PAL_MC_RESUME procedure is defined to return to the interrupted context after handling a machine check or initialization event. See page 2:436 for a description of the PAL_MC_RESUME procedure. If software attempts to return to the interrupted context without using this procedure, processor behavior is undefined.

There are certain error cases that may require returning to a new context in order to recover from the machine check. If this occurs a new context can be returned to via the PAL_MC_RESUME procedure with the *new_context* flag set. The caller needs to set up the new processor min-state save area as shown in Figure 11-2 for all the listed register states. If the caller wants to return to a context where PSR.ic is zero (i.e., an interruption handler) the IIP, IPSR and IFS values in the min-state save area must be set up with the first level return values. These are the values for the IP, PSR and CFM of the interruption handler it wishes to return to. The XIP, XPSR, XFS values in the min-state save area must be set up with the second level return values. These are the IP, PSR and CFM values for where the interruption handler will return to. If the caller wants to return to a context where PSR.ic is one, it must set up the IIP, IPSR, IFS and the XIP, XPSR, and XFS both to contain the new instruction pointer, PSR value, and CFM values.

When returning to a new context, the memory area from BR1 up to the 1KB architectural limit is ignored by the PAL_MC_RESUME procedure. The software constructing the new context min-state save area does not have to worry filling in this memory area with any values. When a new context is returned to, the state originally saved in the min-state save area (old context) shall be discarded and never used again.

In order to return to the interrupted context without loss of any architectural state, the caller must restore all register state that is not stored in the processors min-state save area before making the PAL_MC_RESUME procedure call. Since BR0 and BR1 are the only two branch registers saved in the min-state save area, the caller must only use these two branch registers when making the PAL_MC_RESUME procedure call.

11.4 PAL Initialization Events

11.4.1 PALE_INIT

PALE_INIT is entered when an initialization event (INIT) occurs, as a result of the assertion on an INIT signal to the processor or an INIT interruption occurring. If PSR.mc = 1, the initialization event is held pending until PSR.mc becomes 0. The purpose of PALE_INIT is to save the architecturally defined processor state to the Minimal State Save Area (min-state save area) and to branch to SALE_ENTRY. The code sequence interrupted by the initialization event can be restarted via PAL_MC_RESUME if PSR.ic = 1. The code sequence interrupted by the initialization event can be restarted if PSR.ic = 0 and the processor has implemented the optional recovery resources described in Section 11.3.1.1, "Resources Required for Machine Check and Initialization Event Recovery" on page 2:297. If PSR.ic = 0 and the optional recovery resources have not been implemented, then the initialization event is not recoverable.

11.4.2 PALE_INIT Exit State

The state of the processor on exiting PALE_INIT is listed below. For registers described as being saved to the min-state save area and available for use, the actual values in these registers are undefined unless specifically stated otherwise.

- GRs: The contents of all non-banked static registers (GR1-GR15), bank zero static registers and bank one static registers (GR16-31) at the time of the INIT have been saved in the min-state save area and are available for use.

    - If recovery is not supported when PSR.ic=0 then GR24 - GR31 (bank 0) are undefined and their contents have been lost. In this case, recovery is not possible. See Section 11.3.1.1, "Resources Required for Machine Check and Initialization Event Recovery" for details.
    - GR16 through GR20 (bank 0) contain parameters which PALE_INIT passes to SALE_ENTRY for diagnostic and recovery purposes:
        - GR16 contains the address to the first available location in the min-state save area for use by SAL. The address is 8-byte aligned.
        - GR17 contains the value of the min-state save area address stored in XR0.
        - GR18 contains the Processor State Parameter, as defined in Figure 11-5 on page 2:308.
        - GR19 contains the PALE_INIT return address for rendezvous, or 0 if no return is expected. (See Section 11.3.2.2, "Multiprocessor Rendezvous Requirements for Handling Machine Checks")
        - GR20 contains the SALE_ENTRY state as defined in Figure 11-4.
- FRs: The contents of all floating-point registers are unchanged from the time of the INIT.
- Predicates: All predicate registers have been saved in the min-state save area and are available for use.
- BRs: The contents of all branch registers are unchanged from the time of the INIT except the following:
    - BR0 and BR1 have been saved to the min-state save area and are available for use. Either register may have been changed from the time of entry into PALE_CHECK.
- ARs: The contents of all application registers are unchanged from the time of the INIT, except the RSE control register (RSC), the RSE backing store pointer (BSP), and the ITC and RUC counters. The RSC register is unchanged, except that the RSC.mode field will be set to 0 (enforced lazy mode) and the RSC register at the time of the INIT has been saved in the min-state save area. A cover instruction is executed in the PALE_INIT handler which allocates a new stack frame of zero size. BSP will be modified to point to a new location, since all the registers from the current frame at the time of interruption were added to the RSE dirty partition by the allocation of a new stack frame. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the INIT handler. The RUC register will not be directly modified by PAL, but will continue to count during the execution of the INIT handler while the processor is active.
- CFM: The CFM register points to a zero-size current frame and all the rotating register bases are set to zero. The CFM register at the time of the INIT has been saved to the min-state save area in either the IFS or XFS slot depending on the implementation.
- RSE: The RSE is in enforced lazy mode, and all stacked registers are unchanged from the time of the INIT.
- PSR: PSR.mc is 1; PSR.mfl, PSR.mfh, and PSR.pk are unchanged; all other bits are 0. The PSR at the time of the INIT is saved in the min-state save area.
- CRs: The contents of all control registers are unchanged from the time of the INIT with the exception of the interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the INIT.
- PKRs: The contents of all protection key registers are unchanged from the time of the INIT.

- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the INIT.
- PMCs/PMDs: The contents of the PMC registers are unchanged from the time of the INIT. The contents of the PMD registers are not modified by PAL code, but may be modified if events it is monitoring are encountered.
- Cache: The contents of the caches are unchanged from the time of the INIT.
- TLB: The TCs may be initialized and the TRs are unchanged from the time of the INIT.
- Interruption Resources:
  - IRR: PALE_INIT may not change the IRR, but interrupts may have arrived asynchronously, changing the contents of the IRRs.
  - The contents of IIP, IPSR and IFS at the time of INIT are saved to the min-state save area and are available for use.

11.4.2.1 Processor State Parameter (GR18)

Figure 11-5. Processor State Parameter

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
gr	b0	b1	fp	pr	br	ar	rr	tr	dr	pc	cr	ex	cm	rs	in	dy	pm	pi	mi	tl	hd	us	ci	co	sy	mn	me	ra	rz	rsvd	

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
uc	rc	bc	tc	cc	reserved										se	dsize															

The term "valid" in Table 11-7 indicates that the registers are either unchanged from the time of interruption or that the values have been preserved in the min-state save area.

Table 11-12. Processor State Parameter Fields

Field	Bits	INIT value	Description
rsvd	1:0		Reserved
rz	2	x[a]	The attempted processor rendezvous was successful if set to 1.
ra	3	x[a]	A processor rendezvous was attempted if set to 1.
me	4	0	Distinct multiple errors have occurred, not multiple occurrences of a single error. Software recovery may be possible if error information has not been lost.
mn	5	x[a]	Min-state save area has been registered with PAL if set to 1.
sy	6	0	Storage integrity synchronized. A value of 1 indicates that all loads and stores prior to the instruction on which the machine check occurred completed successfully, and that no loads or stores beyond that point occurred. See Table 11-8.
co	7	1	Continuable. A value of 1 indicates that all in-flight operations from the processor where the machine check occurred were either completed successfully (such as a load), were tagged with an error indication (such as a poisoned store), or were suppressed and will be re-issued if the current instruction stream is restarted. This bit can only be set if the architectural state saved on a machine check is all valid. If this bit is set, then *us* must be cleared to 0, and *ci* must be set to 1. See Table 11-8.
ci	8	1	Machine check is isolated. A value of 1 indicates that the error has been isolated by the system, it may or may not be recoverable. If 0, the hardware was unable to isolate the error within the CPU and memory hierarchy. The error may have propagated off the system (to persistent storage or the network). If *ci* = 0 then *us* will be set to 1, and *co* and *sy* are cleared to 0. See Table 11-8.

Table 11-12. Processor State Parameter Fields (Continued)

Field	Bits	INIT value	Description
us	9	0	Uncontained storage damage. A value of 1 indicates the error is contained within the CPU and memory hierarchy, but that some memory locations may be corrupt. If *us* is set to 1, then *co* and *sy* will always be cleared to 0. See Table 11-8.
hd	10	0	Hardware damage. A value of 1 indicates that as a result of the machine check some non essential hardware is no longer available causing this processor to execute with degraded performance (no functionality has been lost).
tl	11	0	Trap lost. A value of 1 indicates the machine check occurred after an instruction was executed but before a trap that resulted from the instruction execution could be generated.
mi	12	0	More information. A value of 1 indicates that more error information about the machine check event is available by making the PAL_MC_ERROR_INFO procedure call.
pi	13	0	Precise instruction pointer. A value of 1 indicates that the machine logged the instruction pointer to the bundle responsible for generating the machine check.
pm	14	0	Precise min-state save area. A value of 1 indicates that the min-state save area contains the state of the machine for the instruction responsible for generating the machine check. When this bit is set, the *pi* bit will always be set as well.
dy	15	x[a]	Processor Dynamic State is valid. (1=valid, 0=not valid) See the PAL_MC_DYNAMIC_STATE procedure call for more information.
in	16	1	Interruption caused by INIT. (0=machine check, 1=INIT)
rs	17	x[a]	The RSE is valid. (1=valid, 0=not valid)
cm	18	0	The machine check has been corrected. (1=corrected, 0=not corrected)
ex	19	0	A machine check was expected. (1=expected, 0=not expected)
cr	20	x[a]	Control registers are valid. (1=valid, 0=not valid)
pc	21	x[a]	Performance counters are valid. (1=valid, 0=not valid)
dr	22	x[a]	Debug registers are valid. (1=valid, 0=not valid)
tr	23	x[a]	Translation registers are valid. (1=valid, 0=not valid)
rr	24	x[a]	Region registers are valid. (1=valid, 0=not valid)
ar	25	x[a]	Application registers are valid. (1=valid, 0=not valid)
br	26	x[a]	Branch registers are valid. (1=valid, 0=not valid)
pr	27	x[a]	Predicate registers are valid. (1=valid, 0=not valid)
fp	28	x[a]	Floating-point registers are valid. (1=valid, 0=not valid)
b1	29	x[a]	Preserved bank one general registers are valid. (1=valid, 0=not valid)
b0	30	x[a]	Preserved bank zero general registers are valid. (1=valid, 0=not valid)
gr	31	x[a]	General registers are valid. (1=valid, 0=not valid) (does not include banked registers)
dsize	47:32	x[a]	Size in bytes of Processor Dynamic State returned by PAL_MC_DYNAMIC_STATE.
se	48	0	Shared Error. Machine check corresponds to structure shared by multiple logical processors.
rsvd	58:49		Reserved
cc	59	0	Cache check. A value of 1 indicates that a cache related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
tc	60	0	TLB check. A value of 1 indicates that a TLB related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
bc	61	0	Bus check. A value of 1 indicates that a bus related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

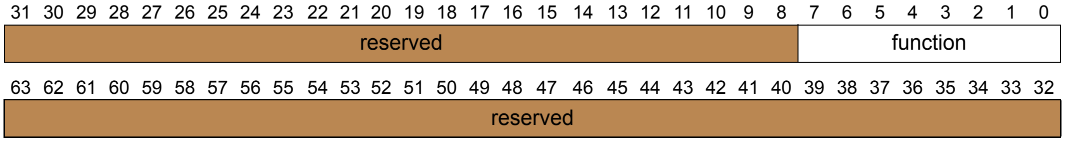
Table 11-12. Processor State Parameter Fields (Continued)

Field	Bits	INIT value	Description
rc	62	0	Register file check. A value of 1 indicates that a register file related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
uc	63	0	Uarch check. A value of 1 indicates that a micro-architectural related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

a. The values of the fields marked with x are set by the PAL INIT handler based on the INIT handling.

11.4.2.2 Definition of SALE_ENTRY State Parameter

Figure 11-6. SALE_ENTRY State Parameter



- *function* – An 8-bit field indicating the reason for branching to SALE_ENTRY.

Table 11-13. *function* Field Values

Function	Value	Description
RESET	0	System reset or power-on
MACHINE CHECK	1	Machine check event
INIT	2	Initialization event
RECOVERY CHECK	3	Check for recovery condition in SAL

All other values of *function* are reserved.

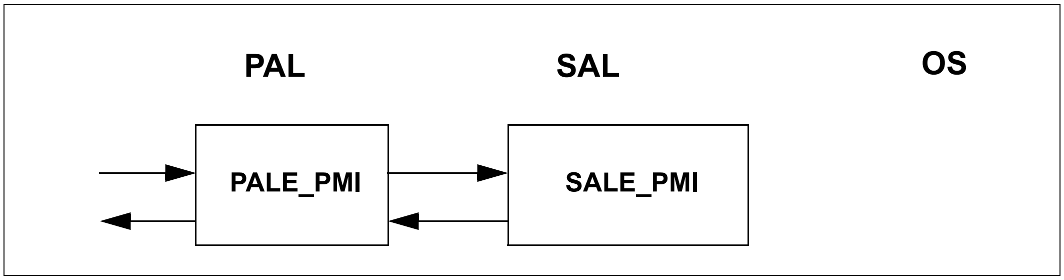
11.5 Platform Management Interrupt (PMI)

11.5.1 PMI Overview

PMI is an asynchronous interrupt that encapsulates a collection of platform-specific interrupts. Platform Management Interrupts occur during instruction processing, causing the flow of control to be passed to the PAL PMI handler. In the process, state is saved in the interruption registers (IIP, IPSR) by the processor hardware and the processor starts executing instructions at the PALE_PMI entrypoint. The PAL code will save some additional state in the bank 0 registers. The PAL will either handle the PMI if it is PAL related PMI or transition to the SAL PMI code if it is a SAL related PMI. Upon completion of processing, the SAL PMI code returns to PAL PMI code to restore the interrupted processor state and to resume execution at the interrupted instruction.

As shown in Figure 11-7, PMI code consists of two major components, namely the PAL PMI handler which handles all processor-specific processing, and the SAL PMI handler which handles all platform-related processing. The location of the PALE_PMI and SALE_PMI handlers are programmable. The location of the PALE_PMI handler can be programmed by the PAL_COPY_PAL procedure described on page 2:389. The SALE_PMI handler can be programmed by the PAL_PMI_ENTRYPOINT procedure described on page 2:443. If a PMI is taken very early in the boot sequence before PAL has a chance to register its PALE_PMI entrypoint, processor operation is undefined. If a SAL related PMI is seen before the SAL PMI handler is registered, the PAL PMI code will just return to the interrupted context

Figure 11-7. PMI Entrypoints



The hardware events that can cause the PMI request are referred to as PMI events. PMI events are asynchronous interrupts higher priority than all external interrupts and are only maskable when the system software is processing very critical tasks with PSR.ic=0. When PSR.ic is 1, PMI events are unmasked. PSR.i has no effect on PMI events. All PMI events are internally latched into an array of implementation-specific latches in the processor. The PAL PMI handler reads the latches to determine what PMI vector requests are pending and dispatches them in priority order. Table 11-14 lists the PMI events and their priority.

Table 11-14. PMI Events and Priorities

PMI Events	Priority
PMI message for PAL (vectors 4-15)	High
PMI message for SAL (vectors 1-3)	
PMI pin[a] (vector 0)	Low

[a] PMI pin is not required to be present on all systems.

PMI messages can be delivered by an external interrupt controller, or as an inter-processor interrupt using delivery mode 010. Table 11-15 shows the PMI message vector assignments. Vectors 4-15 are reserved for PAL, and within these PAL vectors, a higher vector number has higher priority. Vectors 1-3 are available for SAL to use, and within these SAL vectors, a higher vector number has higher priority. A PMI pin event, when the PMI pin[1] is present, is indicated by vector 0. The PMI vector number is passed to the SAL PMI handler in GR 24.

Table 11-15. PMI Message Vector Assignments

Priority		Vector	Description
Low	SAL Vectors	0	PMI pin
↓		1	Available for SAL firmware
		2	
High		3	

[1] PMI pin is not required to be present. Software can query the presence of PMI pin via the PAL_PROC_GET_FEATURES procedure call.

Table 11-15. PMI Message Vector Assignments

Priority		Vector	Description
Low	PAL Reserved	4	PAL Reserved
		5	
		6	
		7	
		8	
		9	
		10	
		11	
		12	
High		13	IA-32 Machine Check Rendezvous
		14	PAL Reserved
		15	

11.5.2 PALE_PMI Exit State

The state of the processor on exiting PALE_PMI is:

- GRs: The contents of non-banked general registers are unchanged from the time of the interruption.
  - Bank 1 GRs: The contents of all bank one general registers are unchanged from the time of the interruption.
  - Bank 0:GR16-23: The contents of these bank zero general registers are unchanged from the time of the interruption.
  - Bank 0:GR24-31: contain parameters which PALE_PMI passes to SALE_PMI:
    - GR24 contains the value decoded as follows:
      - Bits 7-0: PMI Vector Number
      - Bit 63-8: Reserved
    - GR25 contains the value of the min-state save area address stored in XR0.
    - GR26 contains the value of saved RSC. The contents of this register shall be preserved by SAL PMI handler.
    - GR27 contains the value of saved B0. The contents of this register shall be preserved by SAL PMI handler.
    - GR28 contains the value of saved B1. The contents of this register shall be preserved by SAL PMI handler.
    - GR29 contains the value of the saved predicate registers. The contents of this register shall be preserved by SAL PMI handler
    - GR30-31 are scratch registers available for use.
- FRs: The contents of all floating-point registers are unchanged from the time of the interruption.
- Predicates: The contents of all predicate registers are undefined and available for use.
- BRs: The contents of all branch registers are unchanged, except the following which contain the defined state.
  - BR1 is undefined and available for use.

    - BR0 PAL PMI return address.
- ARs: The contents of all application registers are unchanged from the time of the interruption, except the RSE control register (RSC) and the ITC and RUC counters. The RSC.mode field will be set to 0 (enforced lazy mode) while the other fields in the RSC are unchanged. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the PMI handler. The RUC register will not be directly modified by PAL, but will continue to count during the execution of the PMI handler while the processor is active.
- CFM: The contents of the CFM register is unchanged from the time of the interruption.
- RSE: Is in enforced lazy mode, and stacked registers are unchanged from the time of the interruption.
- PSR: PSR.mc, PSR.mfl, PSR.mfh, and PSR.pk are unchanged; all other bits are 0.
- CRs: The contents of all control registers are unchanged from the time of the interruption with the exception of interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the interruption.
- PKRs: The contents of all protection key registers are unchanged from the time of the interruption.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the interruption.
- PMCs/PMDs: The contents of the PMC registers are unchanged from the time of the PMI. The contents of the PMD registers are not modified by PAL code, but may be modified if events it is monitoring are encountered
- Cache: The processor internal cache is not specifically modified by the PMI handler but may be modified due to normal cache activity of running the handler code.
- TLB: The TCs are not modified by the PALE_PMI handler and the TRs are unchanged from the time of the interruption.
- Interruption Resources:
    - IRRs: The contents of IRRs are unchanged from the time of the interruption.
    - IIP and IPSR contain the value of IP and PSR. The IFS.v bit is reset to 0.

11.5.3 Resume from the PMI Handler

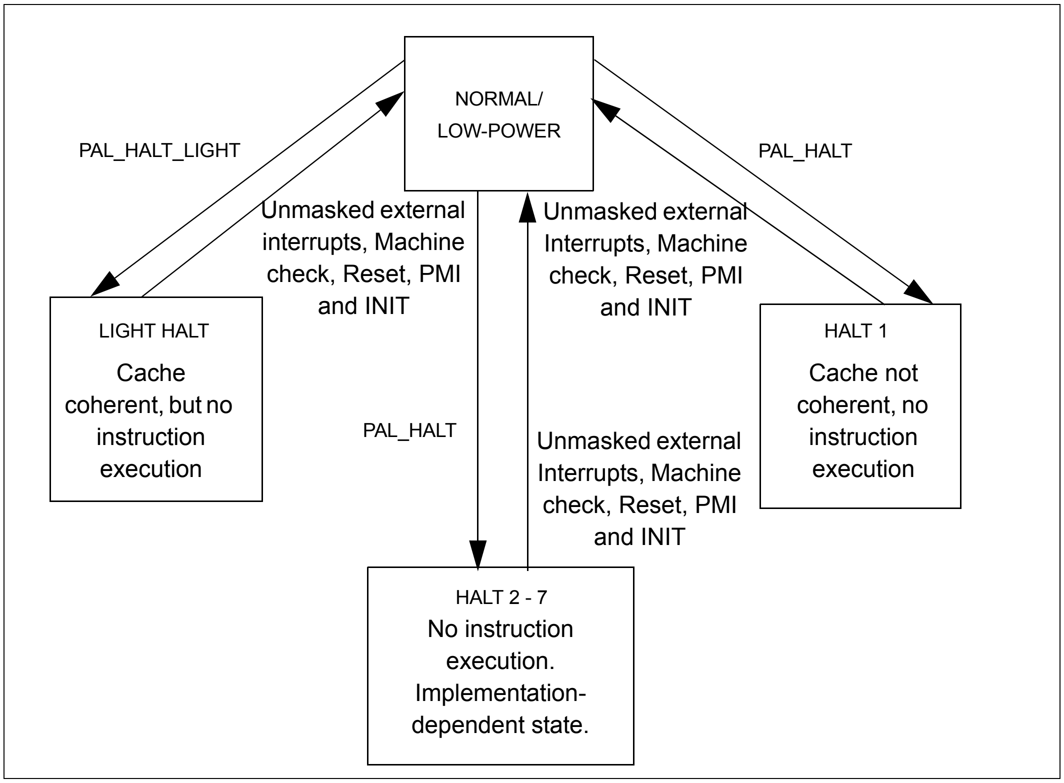
To return to the instruction that was interrupted by the PMI event, SAL PMI must branch to the PAL PMI target address in BR0. All register contents must be preserved as specified in Section 11.5.2, “PALE_PMI Exit State” on page 2:312.

11.6 Power Management

This section describes the architecturally supported set of required and optional power states that may be implemented to reduce power consumption in implementations where this is a design goal. In addition, the PAL interfaces required to manage these states are described.

Figure 11-8 shows state transitions for the various power states and the software interfaces required for the transitions.

Figure 11-8. Power States


NORMAL/
LOW-POWER
PAL_HALT_LIGHT
PAL_HALT
Unmasked external interrupts, Machine check, Reset, PMI and INIT
Unmasked external Interrupts, Machine check, Reset, PMI and INIT
LIGHT HALT
Cache coherent, but no instruction execution
PAL_HALT
Unmasked external Interrupts, Machine check, Reset, PMI and INIT
HALT 1
Cache not coherent, no instruction execution
HALT 2 - 7
No instruction execution. Implementation-dependent state.

- NORMAL – The normal, fully functional, highest power state.
- LOW-POWER – An implementation may choose to dynamically reduce power via microarchitectural low power techniques. The operation of interrupts, snoops, etc., in low-power mode will be identical to those in normal-power mode. This dynamic power reduction is optional for an implementation to support. The PAL procedures PAL_PROC_GET_FEATURES and PAL_PROC_SET_FEATURES returns whether an implementation supports dynamic power reduction. If an implementation supports dynamic power reduction then this procedure will allow the caller to enable or disable this feature.

The following software controllable low power states may be provided. They are described below.

- LIGHT_HALT – Entered by calling PAL_HALT_LIGHT. This state reduces power by stopping instruction execution, but maintains cache and TLB coherence in response to external requests. The processor transitions from this state to the NORMAL state in response to any unmasked external interrupt (including NMI), machine check, reset, PMI or INIT. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields. This state is a required state.
- HALT 1 – Entered by calling PAL_HALT with a power state argument equal to one. This implementation-dependent low-power state will maintain the processor caches but will ignore any coherency bus traffic. This state is optional for a processor to

implement. It is the responsibility of the caller to ensure cache coherency in this state.

- HALT 2 - 7 – These are optional implementation-dependent states entered by calling PAL_HALT with a power state argument in the range of 2-7. Before making this procedure call, the operating system software should first ascertain that the states are implemented by calling PAL_HALT_INFO. The information returned from the PAL_HALT_INFO procedure will also specify the coherency of caches and TLBs for each of these low-power states.

The interval timer within the processor will function at a constant frequency in all the power states as long as the input clock to the processor is maintained. If all logical processors on the physical processor are in a halt state, the resource utilization counter for the last logical processor to enter a halt state will function at a constant frequency as long as the input clock to the processor is maintained. However, the performance monitor event that counts the number of processor clock cycles will only increment in either the NORMAL or LOW-POWER state.

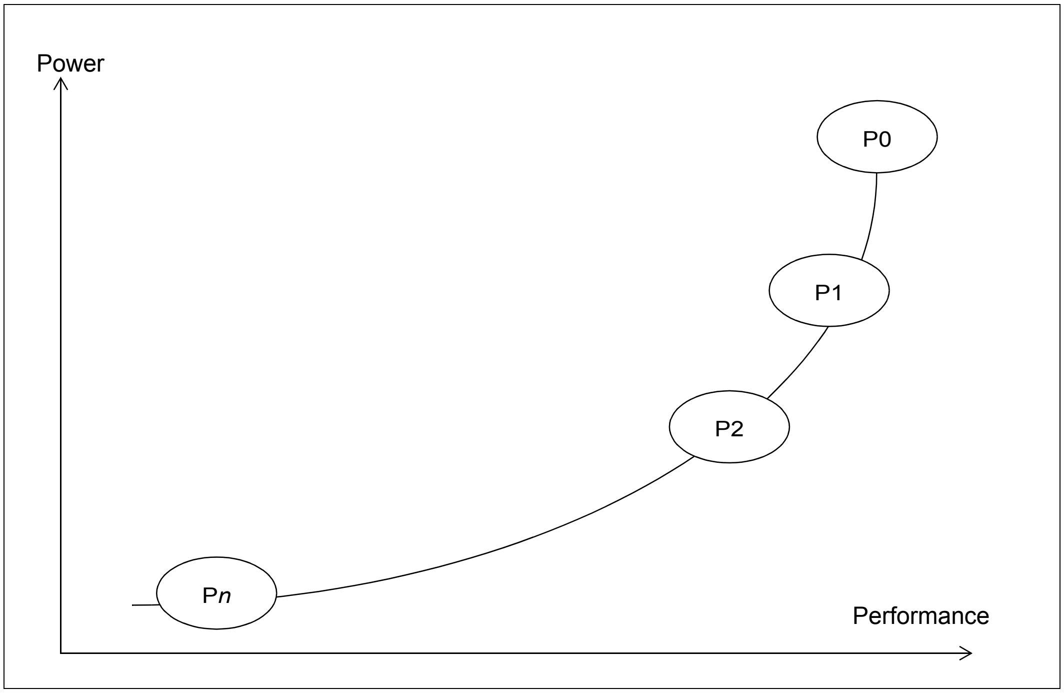
The PAL procedure PAL_HALT_INFO returns information about the power states implemented in a particular processor. This information allows the caller to decide which low power states are implemented and which ones to call based on the callers requirements.

11.6.1 Power/Performance States (P-states)

This section describes the power/performance states (hence to be referred to as P-states) supported by the Itanium architecture. P-states enable the caller to adjust the power/performance characteristics of the processor in response to changing workload requirements. This allows for implementation of a processor-level power management policy which is driven by system demand and response time requirements.

The P-states are defined within the context of the active/executing processor state. At the highest performing P-state (referred to as the P0 state), the processor uses its maximum performance capability and may consume maximum power. In the next P-state (P1), the processor performance capability is limited below the maximum performance, and it consumes less than the maximum power. Successive P-states continue to have reduced performance capabilities and reduced power consumption. The Itanium architecture supports a maximum of 16 P-states, with the highest numbered P-state that is available on an implementation providing the least possible performance capability and minimal power consumption while remaining in a non-HALT state.

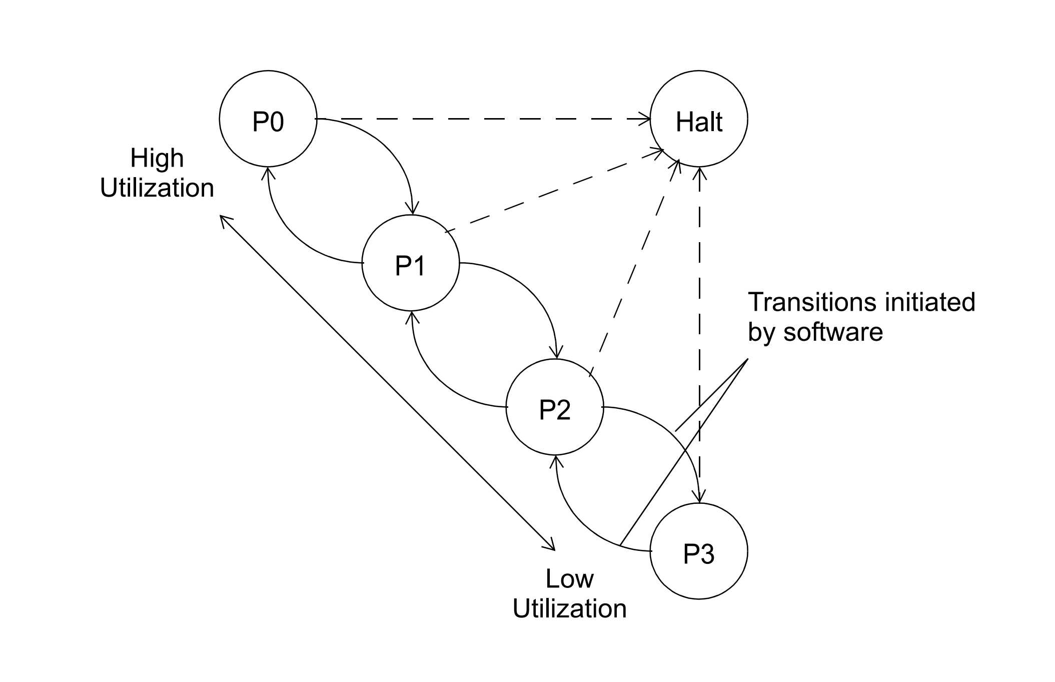
Figure 11-9. Power and Performance Characteristics for P-states



P-states can be utilized by software to implement a demand-based dynamic power management policy where it would continuously try to adapt the processor performance to the current workload characteristics. This allows software to achieve power savings at the system level, while allowing it to quickly respond to changing workload requirements.

The example in Figure 11-10 assumes four P-states (P0, P1, P2 and P3), and a software policy that transitions between the states depending on the current system utilization. During times of high utilization, the software migrates the processor towards lower-numbered P-states, which increases processor performance and increases the dissipated power. When system utilization is low, the software policy migrates the processor towards higher-numbered P-states, thereby reducing the processor performance and reducing dissipated power. The figure also shows the HALT state, which the software can transition to at any time from a given P-state.

Figure 11-10. Example of a P-state Transition Policy



11.6.1.1 Power Dependency Domains

The concept of P-states applies to each logical processor, and this gives software the required granularity to individually control the power/performance characteristics for each available thread of execution in the system. In the most simplistic case, the processor package has only one thread of execution, and this allows software to apply the same P-state policy at the package-level as well as at the logical processor level. However, with implementations that support multithreading and multiple cores, a single package can have multiple logical processors (threads of execution). These may have P-state dependencies among them, which may not allow for individual P-state control flexibility at the software level. For example, these logical processors may be sharing the same clock and power delivery network. In such circumstances, software would need to know which logical processors have dependencies and what the nature of the dependencies is, so that appropriate coordination techniques can be applied. To allow the architecture definition to comprehend multi-threaded/multi-core designs, we define the concept of dependency domain and coordination mechanisms.

A **dependency domain** is comprised of logical processors that share a common set of implementation-dependant domain parameters that affect power consumption and performance for all logical processors in that domain. As an example, a processor package comprised of two cores controlled by the same clock and power distribution network are part of the same dependency domain, since changing either the operating frequency or voltage will affect power consumption and performance for both cores. Alternatively, if these two cores on the processor package had independent distribution networks for clocks and power, then a change in the parameters for one core would not have any effect on the other core, and in that case, the cores would not belong to the same dependency domain. Software can utilize P-states to effect changes in the domain parameters. Each P-state maps to a set of values for the domain parameters, and hence a P-state transition results in a change in the underlying power/performance characteristics for the logical processor.

The Itanium architecture supports different types of dependency domains, which enables software to have different degrees of control for P-state changes affecting logical processors in the domain.

A **software-coordinated dependency domain (SCDD)** relies on the software to coordinate P-state changes among the processors in that dependency domain. Software will have knowledge about logical processors belonging to that domain, and will decide when it is appropriate to request the P-state transition. The software policy has to be aware that a P-state change on any logical processor will change the P-state for all logical processors in that domain. As an example, let us assume that the SCDD consisted of two cores with the same clock and power distribution networks and the intent of the software policy was to lower power/performance only when the workload utilization was low on both cores. Software could then monitor utilization on both cores, and when both cores were under-utilized (i.e., were running at a higher performance P-state than required by the current system demand), it could migrate one of the cores to a lower performance P-state. This transition would simultaneously reduce performance and power dissipation for both cores, and would result in both cores operating at the same lower P-state.

A **hardware-coordinated dependency domain (HCDD)** relies on hardware-based mechanisms to synchronize P-state changes. Software can make independent P-state change requests on individual processors, recognizing that hardware is responsible for the required coordination with other processors in the same HCDD. Hardware-based coordination mechanisms would be implemented to allow for changes to the logical processor's power and performance local parameters (which are implementation-dependant), in addition to the existing domain parameters. Hardware would use a combination of changes to both of these parameters to satisfy the software-initiated P-state change request. This type of coordination mechanism is effective when it is desired to have individual control over all logical processors, and when the hardware has local parameters for power/performance at the logical processor level. The local parameters allow for fine-grained control (affecting only the logical processor power/performance), whereas the domain parameters allow for coarse-grained control (affecting all logical processors). Domain parameters are set by hardware according to the highest requested power/performance level (i.e., the lowest numbered P-state) of the logical processors in the power domain. As an example, let us assume that the HCDD consisted of two cores with the same clock and power distribution networks, and that there were also some other techniques to affect power and performance which were local to each logical processor. Let us also assume that software has initially set both cores to the P0 state. When software initiates a P-state transition to P1 (which is a lower power/performance level) on the first core, hardware would use only the local parameters to carry out the request, and the domain parameters would remain at P0. Suppose software on the second core then initiates a P-state transition to P3. Hardware would then set the local parameters for the second core to reflect this request, undo the changes to the local parameters for the first core plus initiate changes to the domain parameters to transition the domain to the P1 state (the highest requested power/performance level of the two cores).

A **hardware-independent dependency domain (HIDD)** is a self-contained domain that typically means that every logical processor is the only logical processor in that domain, and its domain parameters are individually controllable. Since there are no dependencies with any other logical processors, there is no P-state coordination needed for such domains. Software can make P-state change requests independently on that logical processor.

11.6.1.2 Platform Power-Cap and P-states

Some processor implementations include mechanisms which allow the platform hardware and firmware to temporarily decrease the operating frequency of logical processors, to implement fast-response power capping. This is referred to as a **Platform Power-Cap**. In such implementations, the P-state requested by software is not changed by the platform power-cap. Software is able to change its P-state request during platform power-caps; when the platform power-cap is removed, the processor operating frequency returns to the frequency determined by software's most recent P-state settings.

Platform power caps are meant to have a very short duration and very low duty cycle so they do not significantly affect software methods for managing power through P-states. Platform power-caps do not affect the instantaneous operating P-state observed by software, but do affect the weighted-average performance index reported to software by PAL, so that software may take into account any small effects. (See the PAL_GET_PSTATE procedure for details.)

11.6.1.3 PAL Interfaces for P-states

The PAL procedure PAL_PROC_GET_FEATURES returns whether an implementation supports P-states. If an implementation supports P-states then the PAL_PROC_SET_FEATURE procedure will allow the caller to enable or disable this feature.

The Itanium architecture provides three PAL procedures to enable P-state functionality.

PAL_PSTATE_INFO: This procedure returns information about the P-states implemented on a particular processor. For details on the information returned by this procedure, please refer to the procedure description on page 2:396. The Itanium architecture supports a maximum of 16 P-states.

PAL_SET_PSTATE: This procedure allows the caller to request the transition of the processor to a new P-state. The procedure can either return with transition success (request was accepted) or transition failure (request was not accepted) depending on hardware capabilities, implementation-specific event conditions, and the spacing between successive PAL_SET_PSTATE procedure calls.

If hardware has the ability to either preempt a previous in-progress P-state transition, or to queue successive P-state requests while the first request is in transition, then the implementation has a pre-emptive policy for P-state request handling. The architecture also allows for a non-preemptive policy for P-state request handling, whereby a new PAL_SET_PSTATE request is not accepted if a previous P-state transition is already in progress. The PAL_SET_PSTATE procedure returns different status values corresponding to the accepted and not accepted cases for P-state requests. If the transition is not accepted, no P-state transition is initiated by the PAL_SET_PSTATE procedure, and the caller is expected to make another PAL_SET_PSTATE request to transition to the desired P-state. The *transition_latency_2* field in the *pstate_buffer* returned by PAL_PSTATE_INFO indicates the time interval the caller needs to wait to have a reasonable chance of success when initiating another PAL_SET_PSTATE call.

Implementation-specific event conditions may prevent a PAL_SET_PSTATE request from being accepted (e.g., due to a thermal protection mechanism), in which case the PAL procedure returns a status of *transition failure*. Such events are expected to be rare and to happen only in abnormal situations.

It should be noted that platform power-caps do not cause a PAL_SET_PSTATE request to fail. The requested P-state is registered with PAL, and the procedure returns a status of *transition success*.

SCDD: If the logical processor belongs to a software-coordinated dependency domain, the PAL_SET_PSTATE procedure will change the domain parameters resulting in a transition to the requested P-state for all logical processors in that domain.

HCDD: If the logical processor belongs to a hardware-coordinated dependency domain, the PAL_SET_PSTATE procedure will attempt to change the power/performance characteristics for that logical processor. Since the power/performance characteristics for the domain depend on the P-state settings of the other logical processors in the domain, a PAL_SET_PSTATE call on one logical processor may result in either partial or complete transition to the requested P-state. In case of partial transition (see Figure 11-11, “Computation of performance_index” on page 2:321 for an example, where the logical processor transitions from state P0 to state P3 in partial increments), the logical processor may attempt to perform changes at a later time to the local parameters and/or domain parameters to transition to the originally requested P-state based on P-state transition requests on other logical processors. Software can also approximate the behavior of a SCDD by forcing P-state transitions. See the description of the PAL_SET_PSTATE procedure for more details.

HIDD: If the logical processor belongs to a hardware-independent dependency domain, the PAL_SET_PSTATE procedure will attempt to change the domain parameters, which will transition the logical processor in that domain to the requested P-state.

PAL_GET_PSTATE: This procedure returns the performance index of the logical processor, relative to the highest available P-state (P0). A value of 100 in P0 represents the minimum processor performance in the P0 state. For example, if the value returned by the procedure is 80, this indicates that the performance of the logical processor over the last time period was 20% lower than the minimum P0 performance. For processors that support variable P-states, it is possible for a processor to report a number greater than 100, representing that the processor is running at a performance level greater than the minimum P0 performance. For example, if the value returned by the processor is 120, it indicates that the performance of the logical processor over the last time period was 20% higher than the minimum P0 performance. The performance index is measured over the time interval since the last PAL_GET_PSTATE call with a type operand of 1. If the processor supports variable P-state performance then the PAL_PROC_SET_FEATURE procedure can be used to enable or disable this feature. Software may choose, on each invocation of the PAL_GET_PSTATE procedure, whether to reset the internal performance measurement logic; resetting the measurement logic initiates a new *performance_index* count, which is reported when the next PAL_GET_PSTATE procedure call is made. A call to PAL_GET_PSTATE with a *type* operand of 1 resets the performance measurement logic.

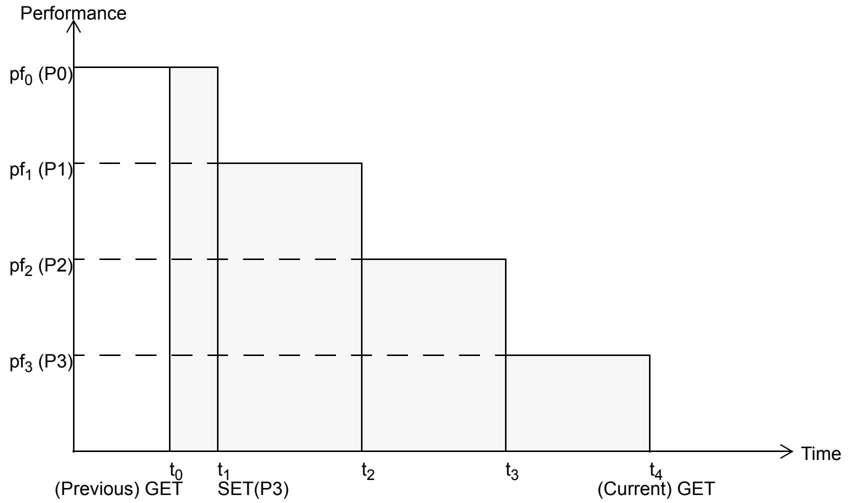
SCDD: If the logical processor belongs to a software-coordinated dependency domain, the performance index returned (for either *type*=0 or 3) corresponds to the target P-state requested by the most recent successful PAL_SET_PSTATE procedure call. No weighted average (*type*=1 or 2) is computed by PAL; calling PAL_GET_PSTATE with *type*=1 or 2 on a SCDD logical processor is undefined.

HCDD: If the logical processor belongs to a hardware-coordinated dependency domain, the performance index returned (*type*=1 or 2) will be a weighted-average sum of the *performance_index* values corresponding to the different P-states that the logical processor was operating in since performance measurement was last reset. Note that this return value may not necessarily correspond to the performance index of the target P-state requested by the most recent PAL_SET_PSTATE procedure call. For example, let's assume that the previous PAL_GET_PSTATE procedure was called at time t_0, when the processor was operating in state P0. The previous PAL_SET_PSTATE procedure requested a transition from P0 to P3. The transition happened over a period of time, such that the logical processor went through states P1 at time t_1, P2 at time t_2 and P3 at time t_3, and was in state P3 at time t_4 when the current PAL_GET_PSTATE procedure was called. The *performance_index* returned is calculated as:

performance_index =
((time spent in P0 after the previous PAL_GET_PSTATE) * (*performance_index* for P0)+
(time spent in P1) * (*performance_index* for P1) +
(time spent in P2) * (*performance_index* for P2) +
(time spent in P3 up to the current PAL_GET_PSTATE) * (*performance_index* for P3)) /
(time interval between previous and current PAL_GET_PSTATE) =

$$\frac{(t_1 - t_0) \times pf_0 + (t_2 - t_1) \times pf_1 + (t_3 - t_2) \times pf_2 + (t_4 - t_3) \times pf_3}{t_4 - t_0}$$

Figure 11-11. Computation of *performance_index*



As seen above, for a HCDD, the PAL_GET_PSTATE procedure allows the caller to get feedback on the dynamic performance of the processor over a software-controlled time period. The caller can use this information to get better system utilization over a subsequent time period by changing the P-state in correlation with the current workload demand. The caller can also use PAL_GET_PSTATE to see the most recent P-state set for this logical processor (*type*=0) and the instantaneous current P-state that the domain parameters are set to (*type*=3). Platform power-caps do not affect either of these return values.

HIDD: If the logical processor belongs to a hardware-independent dependency domain, a weighted-average performance index can be returned by PAL_GET_PSTATE (*type*=1 or 2). Since software could calculate the performance index based on P-states it set, the weighted-average performance index is only of value when factoring in the effect of platform power-caps.

Note that P-state transitions typically do not happen instantaneously. An implementation-specific amount of time is required for a given transition to complete. The computation of the weighted-average *performance_index* may not take into account the fact that transitions of power/performance are gradual, but may be done as though they were instantaneous at the point when the transition starts. The expectation is that any errors in computing the *performance_index* due to non-instantaneous transitions to higher and lower P-states will tend to cancel out, and to the extent that they do not, will be insignificant.

11.6.1.4 Variable P-state Performance

Some processors support variable P-state performance which allows the frequency to vary within a given P-state in order to achieve the maximum performance for that P-state's power budget. The PAL_PROC_GET_FEATURES procedure indicates whether the processor supports variable P-state performance (see "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details).

Since the frequency within a P-state can vary, the performance index calculation is slightly different when a processor supports variable P-state performance. Frequencies for a given P-state are represented by an index value $F_{x,y}$. The value *x* is the P-state number and *y* represents a frequency point in the range from 0 to N. A value of 0 represents the minimum frequency index value for the given P-state. For example:

$F_{0,0}$ to $F_{0,N}$ – Frequency index values for the P0 state
$F_{1,0}$ to $F_{1,N}$ – Frequency index values for the P1 state
…etc.

$F_{0,0}$ is the minimum frequency index for the P0 state and its value is 100. $F_{0,1}$ represents a higher frequency point for P0 and will have a value greater than 100. For example, if $F_{0,1}$ frequency is 5% greater than $F_{0,0}$ it would have a value of 105.

The *performance_index* equation for P0 is calculated as follows:

$((F_{0,0}$ * time spent in $F_{0,0})$ + $(F_{0,1}$ * time spent in $F_{0,1})$+ .. $(F_{0,N}$ * time spent in $F_{0,N}))$ / (Total Time spent in P_0)

For example, let's say the minimum frequency of P0 is 1 GHz and the maximum frequency of P0 is 1.5 GHz. If we are at 1 GHz for a time period of 4, 1.25 GHz for a time period of 16 and 1.5 GHz for a time period of 20, the average performance index is:

((100 * 4) + (125 * 16) + (150 * 20)) / (5 + 15 + 20) = 135

The *performance_index* equation for other P-states can be calculated in a similar manner using their respective frequency index values.

The total *performance_index* equation for a processor with four P-states (P0, P1, P2, P3) would be:

(($F_{0,0}$ * time spent in $F_{0,0}$) + ($F_{0,1}$ * time spent in $F_{0,1}$)+ .. ($F_{0,N}$ * time spent in $F_{0,N}$)+
($F_{1,0}$ * time spent in $F_{1,0}$) + ($F_{1,1}$ * time spent in $F_{1,1}$)+ .. ($F_{1,N}$ * time spent in $F_{1,N}$)+
($F_{2,0}$ * time spent in $F_{2,0}$) + ($F_{2,1}$ * time spent in $F_{2,1}$)+ .. ($F_{2,N}$ * time spent in $F_{2,N}$)+
($F_{3,0}$ * time spent in $F_{3,0}$) + ($F_{3,1}$ * time spent in $F_{3,1}$)+ .. ($F_{3,N}$ * time spent in $F_{3,N}$)) /
(Total Time)

11.6.1.5 Interaction of P-states with HALT State

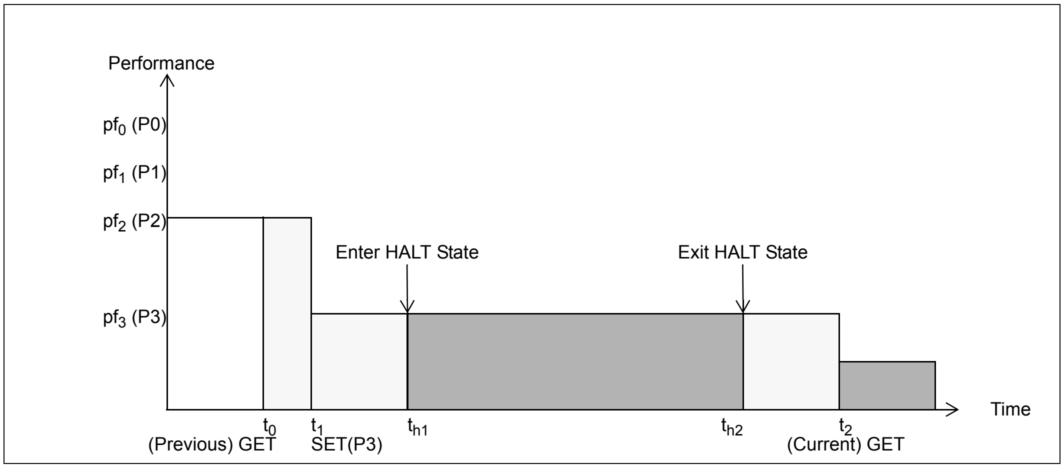
It is possible for a logical processor to enter and exit a HALT state between two consecutive calls to PAL_GET_PSTATE. Since the logical processor is not executing any instructions while in the HALT state, the performance index contribution during this period is essentially 0, and will not be accounted for in the *performance_index* value returned when the next PAL_GET_PSTATE procedure call is made.

For example, let us assume that the previous PAL_GET_PSTATE procedure was called at time t_0, when the processor was operating in state P2. The previous PAL_SET_PSTATE procedure initiated a transition from P2 to P3 at time t_1. The processor entered HALT state at time t_{h1}, and exited the HALT state at time t_{h2}, and was in state P3 at time t_2 when the current PAL_GET_PSTATE procedure was called. The *performance_index* returned is calculated as:

performance_index =
((time in P2 after the previous PAL_GET_PSTATE) * (*performance_index* for P2) +
(time in P3 before entering HALT state) * (*performance_index* for P3) +
(time in P3 after exiting HALT up to current PAL_GET_PSTATE))) * (*performance_index* for P3)) /
(time interval between previous and current GET, excluding time spent in HALT) =

$$\frac{(t_1 - t_0) \times pf_2 + (t_{h1} - t_1) \times pf_3 + (t_2 - t_{h2}) \times pf_3}{(t_2 - t_0) - (t_{h2} - t_{h1})}$$

Figure 11-12. Interaction of P-states with HALT State



As shown above, the value returned for *performance_index* does not account for the performance during the time spent by the logical processor in the HALT state. This provides for better accuracy in the value reported for *performance_index*, allowing the caller to make optimal adjustments to the system utilization even in scenarios where we have interactions between P-states and HALT state.

11.7 PAL Virtualization Support

This section describes the PAL architectural support for Itanium processor virtualization.

On processors in the Itanium Processor Family that support processor virtualization, the PAL virtualization support described in this document will be available. Itanium processor virtualization support can be determined by calling PAL_PROC_GET_FEATURES.

The virtualization support in PAL presents an implementation-independent interface to enable the VMM to implement software policies to manage/support virtualization of Itanium processors.

The PAL extensions for virtualization consist of three main components:

1. A set of procedures to support virtualization operations. These procedures allow the VMM to configure logical processors for virtualization operations and suspend/resume virtual processors on logical processors. Details for this component are described in Section 11.10, "PAL Procedures" on page 2:353.
2. A set of services to provide low-latency, low-overhead support for performance-critical VMM operations. Details for this component are described in Section 11.11, "PAL Virtualization Services" on page 2:486.
3. A PAL intercept interface to allow PAL to deliver virtualization events to the VMM in a low-latency, low-overhead manner. This PAL-to-VMM interface also allows PAL to provide optimizations for VMM operations. Details for this component are described in Section 11.7.3, "PAL Intercepts in Virtual Environment" on page 2:332.

The VMM is responsible for managing the set of available system resources (CPU, memory, peripherals) and implement policies to virtualize these resources. In order to support virtual processor operations, the VMM will create a **virtual environment** and associate logical processors with the virtual environment. A virtual environment consists of one or more logical processors plus the memory resource allocated by the VMM during PAL_VP_INIT_ENV.

The VMM creates a virtual environment by calling PAL_VP_ENV_INFO to obtain the memory requirement for creating a virtual environment, and then by calling PAL_VP_INIT_ENV on each logical processor that is to be part of the virtual environment. After a virtual environment is created, the VMM can create and initialize virtual processors to run in the environment by calling PAL_VP_CREATE.

The state of a virtual processor belonging to a virtual environment can be restored/saved on a logical processor in the environment by calling PAL_VP_RESTORE or PAL_VP_SAVE respectively. The VMM starts virtual processor operations on a logical processor by invoking either PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

The VMM can add/remove a logical processor from a virtual environment at any time by calling PAL_VP_INIT_ENV or PAL_VP_EXIT_ENV respectively.

11.7.1 Virtual Processor Descriptor (VPD)

The Virtual Processor Descriptor (VPD) represents the abstraction of processor resources of a single virtual processor. The VPD consists of per-virtual-processor control information together with performance-critical architectural state. The VPD is 64K in size and the base must be 32K aligned. Table 11-16 shows the fields and layout of the VPD. The values in the VPD can be stored in little or big endian format, depending on the setting of *be* field setting in "config_options – Global Configuration Options" during PAL_VP_INIT_ENV call. See "PAL_VP_INIT_ENV – PAL Initialize Virtual Environment (268)" on page 2:478 for details. The VPD is divided into two classes – the first class stores control information and the second class stores the performance-critical architectural state of the virtual processor.

The VMM must keep the virtual processor state in the VPD for a particular state entry either: always, or only when one or more particular accelerations is enabled, as described in the Class columns of Table 11-16, Table 11-17 and Table 11-18. See Section 11.7.4.2, "Virtualization Accelerations" on page 2:337 for details.

Note: Not all architectural state of the virtual processor is included in the VPD. The VMM is responsible for setting up all the required virtual processor state in the architectural registers as well as in the VPD prior to resuming virtual processor execution. See Table 11-122, "Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER" on page 2:489 and Table 11-123, "Processor Status Register Settings for Virtual Processor Execution" on page 2:490 for details.

Table 11-16. Virtual Processor Descriptor (VPD)

Name	Entries	Offset	Description	Class
vac	1	0	Virtualization Acceleration Control – these control bits enable virtualization acceleration of a particular resource or instruction. See Section 11.7.1.1, "Virtualization Controls" on page 2:329 for details.	Control [always]
vdc	1	8	Virtualization Disable Control – these control bits disable the virtualization of a particular resource or instruction. See Section 11.7.1.1, "Virtualization Controls" on page 2:329 for details.	Control [always]
virt_env_vaddr	1	16	PAL Virtual Environment Buffer Address – this field stores the host virtual address of the virtual environment which the virtual processor belongs to. The value in this field must be the same as the *vbase_addr* field during PAL_VP_INIT_ENV call.	Control [always]
Reserved	29	24	Reserved Area – Reserved for future expansion.	Reserved
vhpi	1	256	Virtual Highest Priority Pending Interrupt – Specifies the current highest priority pending interrupt for the virtual processor. See Table 11-124, "vhpi – Virtual Highest Priority Pending Interrupt" on page 2:495 for details.	Control [a_int]
Reserved	95	264	Reserved Area – Reserved for future expansion.	Reserved
vgr[16-31]	16	1024	Virtual General Registers – Represent the bank 1 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 1, the values in these entries are undefined.	Architectural State [a_bsw]
vbgr[16-31]	16	1152	Virtual Banked General Registers – Represent the bank 0 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 0, the values in these entries are undefined.	Architectural State [a_bsw]
vnat	1	1280	Virtual General Register NaTs – Bits 0-15 represent the NaT values corresponding to vgr16-31, where the NaT bit for vgr16 is in bit 0. Bits 16-63 are don't cares.	Architectural State [a_bsw]
vbnat	1	1288	Virtual Banked Register NaTs – Bits 16-31 represent the NaT values corresponding to vbgr16-31, where the NaT bit for vbgr16 is in bit 16. Bits 0-15 and 32-63 are don't cares.	Architectural State [a_bsw]
vcpuid[0-4]	5	1296	Virtual CPUID Registers – Represent cpuid registers 0-4 of the virtual processor. NOTE: If a_tf is disabled or not supported, vcpuid[0-1] and vcpuid[4]{63:32} must contain the same values as the corresponding values of the logical processor on which this virtual processor is running. If a_tf is enabled, The VMM may maintain a different VCPUID[4]{63:32} value from the CPUID[4]{63:32} value of the logical processor on which the virtual processor is running.	Architectural State [a_from_cpuid, a_tf[a]]

Table 11-16. Virtual Processor Descriptor (VPD) (Continued)

Name	Entries	Offset	Description	Class
Reserved	11	1336	Reserved Area – Reserved for future expansion.	Reserved
vpsr	1	1424	Virtual Processor Status Register – Represents the Processor Status Register of the virtual processor.	Architectural State See Table 11-17 for details.
vpr	1	1432	Virtual Predicate Registers – Represents the Predicate Registers of the virtual processor. The bit positions in vpr correspond to predicate registers in the same manner as with the mov predicates instruction. The contents in this field are undefined except at virtualization intercept handoff. The VMM can not rely on the contents in this field to be preserved when the virtual processor is running.	Architectural State [always]
Reserved	76	1440	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state.	Reserved
vcr[0-127]	128	2048	Virtual Control Registers – Represent the control registers of the virtual processor. For the reserved control registers, the corresponding VPD entries are reserved.	Architectural State See Table 11-18 for details.
Reserved	128	3072	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
Reserved	3456	4096	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
vmm_avail	128	31744	Available for VMM use. This area is ignored by the processor and PAL.	Ignored
Reserved	4096	32768	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved

a. The a_tf acceleration only requires vcpuid[4] be kept in the VPD.

Table 11-17 provides details on which vpsr bits are required to be store in the VPD for different accelerations. Two bits, vpsr.ic and vpsr.si are always required to be in the VPD. The remaining vpsr bits are only required to be stored in the VPD if certain virtualization accelerations are enabled. Even though some fields are not required to be stored in the VPD, the VMM is free to store the entire vpsr in the VPD.

Table 11-17. Virtual Processor Descriptor (VPD) – VPSR

Field	Bits	Class
User Mask = PSR{5:0}		
rv	0	Reserved
be	1	No accelerations require these fields.[a]
up	2	
ac	3	
mfl	4	
mfh	5	
System Mask = PSR{23:0}		
ic	13	Always
i	14	a_int, a_from_psr
pk	15	a_from_psr
rv	12:6, 16	Reserved
dt	17	a_from_psr
dfl	18	
dfh	19	
sp	20	
pp	21	
di	22	
si	23	Always
PSR.l = PSR{31:0}		
db	24	a_from_psr
lp	25	
tb	26	
rt	27	
rv	31:28	Reserved
PSR{63:0}		
cpl	33:32	No accelerations require these fields.
is	34	
mc	35	a_from_psr
it	36	
id	37	No accelerations require these fields.
da	38	
dd	39	
ss	40	
ri	42:41	
ed	43	
bn	44	a_bsw
ia	45	No accelerations require these fields.
vm	46	
rv	63:47	Reserved

a. The user mask is not virtualized. See Section 11.7.4.2.4, “MOV-from-PSR Optimization” on page 2:341 and Section 11.7.4.2.10, “Interruption Collection and User Mask Optimization” on page 2:345 for further details.

Table 11-18. Virtual Processor Descriptor (VPD) – VCR[0-127]

Register	Name	Class
VCR0-15		No accelerations require these virtual control registers.
VCR16	VIPSR	a_from_int_cr, a_to_int_cr
VCR17	VISR	
VCR18		No accelerations require this virtual control register.
VCR19	VIIP	a_from_int_cr, a_to_int_cr
VCR20	VIFA	Always
VCR21	VITIR	Always
VCR22	VIIPA	a_from_int_cr, a_to_int_cr
VCR23	VIFS	a_cover, a_from_int_cr, a_to_int_cr
VCR24	VIIM	a_from_int_cr, a_to_int_cr
VCR25	VIHA	
VCR26	VIIB0	
VCR27	VIIB1	
VCR28-65		No accelerations require these virtual control registers.
VCR66	VTPR	a_int
VCR67-127		No accelerations require these virtual control registers.

11.7.1.1 Virtualization Controls

The Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*) fields in the VPD contain configuration control bits which define the set of events that will cause an intercept from PAL to the VMM. The virtualization controls are divided into two categories:

1. Virtualization Acceleration Control – these control bits enable virtualization optimization support of a particular resource or instruction. Figure 11-13 and Table 11-19 describe these control bits.
2. Virtualization Disable Control – these control bits disable the virtualization of a particular resource or instruction. Figure 11-14 and Table 11-20 describe these control bits.

The *vac* and *vdc* settings are specified by the VMM during virtual processor initialization when the PAL_VP_CREATE procedure is called, and cannot be changed until the virtual processor is terminated by PAL_VP_TERMINATE.

Figure 11-13. Virtualization Acceleration Control (*vac*)

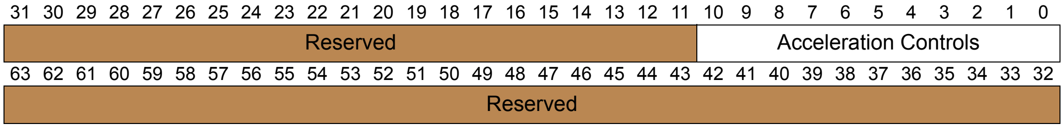


Table 11-19. Virtualization Acceleration Control (*vac*) Fields

Field	Bit	Description
a_int	0	Enable the virtual external interrupt optimization. See Section 11.7.4.2.1, "Virtual External Interrupt Optimization" on page 2:338 for details.
a_from_int_cr	1	Enable the interruption control register (CR16-27) read optimization. See Section 11.7.4.2.2, "Interruption Control Register Read Optimization" on page 2:340 for details.

Table 11-19. Virtualization Acceleration Control (*vac*) Fields (Continued)

Field	Bit	Description
a_to_int_cr	2	Enable the interruption control register (CR16-27) write optimization. See Section 11.7.4.2.3, “Interruption Control Register Write Optimization” on page 2:341 for details.
a_from_psr	3	Enable the processor status register read optimization. See Section 11.7.4.2.4, “MOV-from-PSR Optimization” on page 2:341 for details.
a_from_cpuid	4	Enable the CPUID read optimization. See Section 11.7.4.2.5, “MOV-from-CPUID Optimization” on page 2:342 for details.
a_cover	5	Enable the `cover` instruction optimization. See Section 11.7.4.2.6, “Cover Optimization” on page 2:343 for details.
a_bsw	6	Enable the `bsw` instruction optimization. See Section 11.7.4.2.7, “Bank Switch Optimization” on page 2:343 for details.
a_all_probes	7	Enable virtualization of probe instructions. See Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.
a_select_probes	8	
a_tf	9	Enable the test feature optimization. See Section 11.7.4.2.9, “Test Feature Optimization” on page 2:345 for details.
a_ic_um	10	Enable the interruption collection and user mask optimization. See Section 11.7.4.2.10, “Interruption Collection and User Mask Optimization” on page 2:345 for details.
Reserved	63:11	Reserved

Figure 11-14. Virtualization Disable Control (*vdc*)

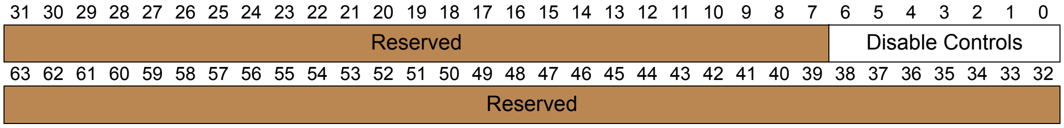


Table 11-20. Virtualization Disable Control (*vdc*) Fields

Field	Bits	Description
d_vmsw	0	Disable `vmsw` instruction – If 1, disables `vmsw` instruction on the logical processor. Execution of the `vmsw` instruction, independent of the state of PSR.vm, will cause a virtualization intercept.
d_extint	1	Disable external interrupt control register virtualization – If 1, accesses (reads/writes) of the external interrupt control registers (CR65-71) are not virtualized. Code running with PSR.vm==1 can read and write the external interrupt control registers of the logical processor directly and without handling off to the VMM. See Section 11.7.4.3.2, “Disable External Interrupt Control Register Virtualization” on page 2:347 for details.
d_ibr_dbr	2	Disable breakpoint register virtualization – If 1, accesses (reads/writes) of the data and instruction breakpoint registers (IBR/DBR) are not virtualized. Code running with PSR.vm==1 can read and write the data/instruction breakpoint registers of the logical processor directly and without handling off to the VMM. If 0, accesses of the breakpoint registers with PSR.vm==1 result in virtualization intercepts.
d_pmc	3	Disable PMC virtualization – If 1, accesses (reads/writes) of the performance monitor configuration registers (PMCs) are not virtualized. Code running with PSR.vm==1 can read and write the performance monitor configuration registers of the logical processor directly and without handling off to the VMM. If 0, accesses of the performance counter configuration registers with PSR.vm==1 result in virtualization intercepts.

Table 11-20. Virtualization Disable Control (*vdc*) Fields (Continued)

Field	Bits	Description
d_to_pmd	4	Disable PMD write virtualization – If 1, writes to the performance monitor data registers (PMDs) are not virtualized. Code running with PSR.vm==1 can write the performance monitor data registers of the logical processor directly and without handling off to the VMM. If 0, writes of the performance counter data registers with PSR.vm==1 result in virtualization intercepts.
d_itm	5	Disable ITM virtualization – If 1, writes to the Interval Timer Match (ITM) register are not virtualized. Code running with PSR.vm==1 can write the ITM register of the logical processor directly and without handling off to the VMM. If 0, writes of the ITM register with PSR.vm==1 result in virtualization intercepts.
d_psr_i	6	Disable PSR.i virtualization – If 1, accesses (reads/writes) to the interrupt bit in processor state register (PSR.i) are not virtualized. Code running with PSR.vm==1 can read and write only the interrupt bit via the `ssm` and `rsm` instructions directly without handling off to the VMM. Attempts to modify other PSR bits in addition to the interrupt bit via the `ssm` and `rsm` instructions will result in virtualization intercepts. Attempts to modify the interrupt bit with the `mov psr.l` instruction will continue to result in virtualization intercepts. If 0, accesses to the PSR.i bit with PSR.vm==1 result in virtualization intercepts.
Reserved	63:7	Reserved

11.7.2 Interruption Handling in a Virtual Environment

For logical processors which have been added to a virtual environment through PAL_VP_INIT_ENV, all IVA-based interruptions continue to be delivered to the **host IVT** independent of the state of PSR.vm at the time of interruption. All IVA-based interruptions are serviced by the host IVT pointed to by the IVA (CR2) control register on the logical processor.

IVA-based interruptions that do not represent virtualization events will be delivered to the **guest IVT** by the VMM. The guest IVT is specified by the VIVA control register in the VPD of the virtual processor.

For IVA-based interruption handling during virtual processor operations, PAL provides maximum flexibility to the VMM by supporting **per-virtual-processor host IVTs**. This allows the VMM to provide a different host IVT with optimizations specific to a particular guest operating system on the virtual processor. The VMM can also choose to provide the same IVT for some or all of the virtual processors in a virtual environment.

Hence, at any time in a virtual environment, the IVA (CR2) control register of the logical processor will be pointing to either:

- The per-virtual-processor host IVT
- The generic host IVT not specific to any virtual processor

The per-virtual-processor host IVT for each virtual processor is setup by PAL when the virtual processor is first created (PAL_VP_CREATE) or registered (PAL_VP_REGISTER) in the virtual environment. The VMM passes a pointer to the host IVT specific to the virtual processor as an incoming parameter to the PAL_VP_CREATE or PAL_VP_REGISTER procedures. The per-virtual-processor host IVT is setup to perform long branches to the corresponding vector of the IVT specified in the incoming parameter for all IVA-based interruptions except the Virtualization vector. Virtualization vector will be delivered as virtualization intercept in the per-virtual-processor host IVT. See Section 11.7.3, “PAL Intercepts in Virtual Environment” on page 2:332 for details on PAL intercepts.

In the virtual environment, the IVA (CR2) control register will be set by PAL virtualization-related procedures and services as summarized in Table 11-21.

Table 11-21. IVA Settings after PAL Virtualization-related Procedures and Services

PAL Virtualization-related Procedure / Service	Description
PAL_VP_CREATE	These procedures do not change the IVA control register.
PAL_VP_ENV_INFO	
PAL_VP_EXIT_ENV	This procedure sets the IVA control register to point to the IVT specified by the caller.
PAL_VM_INIT_ENV	These procedures do not change the IVA control register.
PAL_VP_REGISTER	
PAL_VP_RESTORE / PAL_VPS_RESTORE	This procedure / service sets the IVA control register to point to the per-virtual-processor host IVT.
PAL_VP_SAVE / PAL_VPS_SAVE	This procedure / service does not change the IVA control register.
PAL_VP_TERMINATE	This procedure sets the IVA control register to point to the IVT specified by the caller.

After successful execution of PAL_VP_RESTORE procedure or PAL_VPS_RESTORE service, the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. After successful completion of PAL_VP_RESTORE procedure, the VMM must not change the IVA control register on the logical processor until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE.

On IVA-based interruptions when a virtual processor is running (after PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER), the IVA control register on the logical processor is unchanged and will continue to point to the per-virtual-processor host IVT. On resume execution to the same virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER PAL services, the VMM must ensure the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT at the time of interruption.[1]

11.7.3 PAL Intercepts in Virtual Environment

When the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT, virtualization intercepts will be raised at the Virtualization vector or at an optional virtualization intercept handler specified by the VMM. By default, virtualization intercepts are delivered to the Virtualization vector of the IVT specified by the VMM during PAL_VP_CREATE / PAL_VP_REGISTER. If the VMM specified the optional virtualization intercept handler, all virtualization intercepts are delivered to that handler (instead of the Virtualization vector.)

[1] In other words, the VMM is allowed to change to another IVT after IVA-based interruptions happening during virtual processor execution. The VMM must ensure the per-virtual processor IVT is restored before resuming to the same virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

Section 11.7.3.1, “PAL Virtualization Intercept Handoff State” on page 2:333 describes the handoff state of the PAL intercepts. For all interruption vectors other than Virtualization vector, the architectural state at the corresponding IVA-based interruption vector is the same as defined in Chapter 8, “Interruption Vector Descriptions” in Volume 2.

11.7.3.1 PAL Virtualization Intercept Handoff State

The state of the logical processor at virtualization intercept handoff is:

- GRs:
    - Non-banked GRs: The contents of non-banked general registers are preserved from the time of the interruption.
    - Bank 1 GRs: The contents of all bank one general registers are preserved from the time of the interruption.
    - Bank 0: GR16-23: The contents of these bank zero general registers are preserved from the time of the interruption.
    - Bank 0: GR24-31: Scratch, contains parameters/state for VMM:
        - GR24 indicates the cause of virtualization intercept. See Table 11-22, “PAL Virtualization Intercept Handoff Cause (GR24)” for details. This field is not provided to the VMM if the value of the *cause* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0. If the value of the *cause* field in the *config_options* parameter passed to PAL_VP_INIT_ENV is 0, the value of GR24 on virtualization intercept handoff is undefined.
        - GR25 contains the 41-bit opcode in little endian format and the type of the instruction which caused the fault, excluding the qualifying predicate (qp) field. See Figure 11-15, “PAL Virtualization Intercept Handoff Opcode (GR25),” on page 2:335 for details.
        - GR26-31 are available for the VMM to use.
- FRs: The contents of all floating-point registers are preserved from the time of the interruption.
- Predicates: The contents of all predicate registers are undefined and available for use. The original contents are saved in the VPD.
- BRs: The contents of all branch registers are preserved from the time of the interruption.
- ARs: The contents of all application registers are preserved from the time of the interruption, except the ITC and RUC counters. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the virtualization intercept handler. The RUC register will not be directly modified by PAL, but will continue to count during the execution of the virtualization intercept handler while the processor is active.
- CFM: The contents of the CFM register is preserved from the time of the interruption.
- RSE: All RSE state is preserved from the time of the interruption.
- PSR: PSR fields are set according to the “Interruption State” column in Table 3-2, “Processor Status Register Fields” on page 2:24. PSR.up and pp are set to 0 when *fr_pmc* field in *config_options* parameter during PAL_VP_INIT_ENV is 1.
- CRs: The contents of all control registers are preserved from the time of the interruption with the exception of resources described below.

  - IRRs: The contents of IRRs are not changed by PAL. Incoming interruptions may change the contents.
  - IFS: IFS is unchanged from the time of the interruption.
  - IIP: Contains the value of IP at the time of the interruption.
  - IPSR: Contains the value of PSR at the time of the interruption.
- RRs: The contents of all region registers are preserved from the time of the interruption.
- PKRs: The contents of all protection key registers are preserved from the time of the interruption.
- DBRs/IBRs: The contents of all breakpoint registers are preserved from the time of the interruption.
- PMCs/PMDs: The contents of the PMC registers are preserved from the time of the virtualization intercept. The contents of the PMD registers are not modified by PAL code, but may be modified if events being monitored are encountered. The performance counters will be frozen if specified by the VMM through a parameter of PAL_VP_INIT_ENV procedure.
- Cache: The processor internal cache is not specifically modified by PAL handler but may be modified due to normal cache activity of running the handler code.
- TLB: The TRs are unchanged from the time of the interruption.

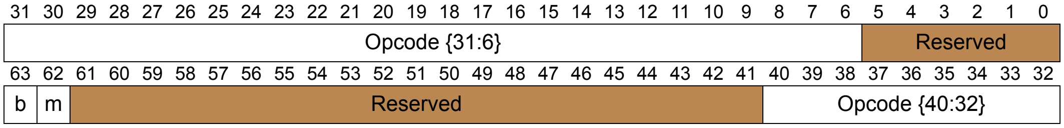
Table 11-22. PAL Virtualization Intercept Handoff Cause (GR24)

Value	Cause	Description
1	toAR	Due to MOV-to-AR instruction.
2	toARimm	Due to MOV-to-AR-imm instruction.
3	fromAR	Due to MOV-from-AR instruction.
4	toCR	Due to MOV-to-CR instruction.
5	fromCR	Due to MOV-from-CR instruction.
6	toPSR	Due to MOV-to-PSR instruction.
7	fromPSR	Due to MOV-from-PSR instruction.
8	itc_d	Due to `itc.d` instruction.
9	itc_i	Due to `itc.i` instruction.
10	toRR	Due to MOV-to-RR instruction.
11	toDBR	Due to MOV-to-DBR instruction.
12	toIBR	Due to MOV-to-IBR instruction.
13	toPKR	Due to MOV-to-PKR instruction.
14	toPMC	Due to MOV-to-PMC instruction.
15	toPMD	Due to MOV-to-PMD instruction.
16	itr_d	Due to `itr.d` instruction.
17	itr_i	Due to `itr.i` instruction.
18	fromRR	Due to MOV-from-RR instruction.
19	fromDBR	Due to MOV-from-DBR instruction.
20	fromIBR	Due to MOV-from-IBR instruction.
21	fromPKR	Due to MOV-from-PKR instruction.
22	fromPMC	Due to MOV-from-PMC instruction.
23	fromCPUID	Due to MOV-from-CPUID instruction.
24	ssm	Due to `ssm` instruction.
25	rsm	Due to `rsm` instruction.
26	ptc_l	Due to `ptc.l` instruction.

Table 11-22. PAL Virtualization Intercept Handoff Cause (GR24) (Continued)

Value	Cause	Description
27	ptc_g	Due to `ptc.g` instruction.
28	ptc_ga	Due to `ptc.ga` instruction.
29	ptr_d	Due to `ptr.d` instruction.
30	ptr_i	Due to `ptr.i` instruction.
31	thash	Due to `thash` instruction.
32	ttag	Due to `ttag` instruction.
33	tpa	Due to `tpa` instruction.
34	tak	Due to `tak` instruction.
35	ptc_e	Due to `ptc.e` instruction.
36	cover	Due to `cover` instruction.
37	rfi	Due to `rfi` instruction.
38	bsw_0	Due to `bsw.0` instruction.
39	bsw_1	Due to `bsw.1` instruction.
40	vmsw	Due to `vmsw` instruction.
41	probe	Due to `probe` instruction.
All other values	Reserved	Reserved for future expansion.

Figure 11-15. PAL Virtualization Intercept Handoff Opcode (GR25)



11.7.4 Virtualization Optimizations

After the PAL_VP_INIT_ENV procedure is called, execution of the virtualized instructions listed in Table 3-10, "Virtualized Instructions" on page 2:44 with PSR.vm==1 results in virtualization intercepts to the VMM. Virtualization optimizations reduce overall virtualization overhead by allowing these instructions to execute, with PSR.vm==1, without causing intercepts to the VMM. There are two types of virtualization optimizations – global and local. Local virtualization optimizations are further divided into virtualization accelerations and virtualization disables.

Global virtualization optimizations are specified during initialization of the virtual environment (i.e., during PAL_VP_INIT_ENV). The specified optimizations are applicable to all the virtual processors running in the virtual environment. See Section 11.7.4.1, "Global Virtualization Optimizations" for details on the global virtualization optimizations supported in the architecture.

Local virtualization optimizations are specified during the creation of the virtual processor (i.e., during PAL_VP_CREATE). The optimization settings were specified in the VPD and hence local to each virtual processor. The VMM can specify different local optimization settings for different virtual processors. The two classes of local virtualization optimizations are:

- Virtualization accelerations – Virtualization accelerations optimize the execution of virtualized instructions by supporting fast access to the virtual instance of the resource and perform the virtualized operations based on the virtual instance of the resource without handling off to the VMM. Section 11.7.4.2, "Virtualization Accelerations" on page 2:337 describes the supported Virtualization accelerations in the architecture.

- Virtualization disables – Virtualization disables optimize the execution of virtualized instructions by disabling virtualization of a particular resource or instruction. Accesses to the virtualization-disabled resources or executions of virtualization-disabled instructions, even with PSR.vm==1, will not cause intercepts to the VMM. Section 11.7.4.3, "Virtualization Disables" on page 2:346 describes the supported Virtualization disables in the architecture.

11.7.4.1 Global Virtualization Optimizations

Table 11-23 summarizes the global virtualization optimizations supported in Itanium architecture.

Table 11-23.Global Virtualization Optimizations Summary

Optimization	*config_options*[a]	Description
Virtualization Opcode Optimization	opcode	Section 11.7.4.1.1
Virtualization Cause Optimization	cause	Section 11.7.4.1.2
Guest MOV-from-AR.ITC Optimization	gitc	Section 11.7.4.1.3

[a] *config_options* is a parameter for the PAL_VP_INIT_ENV procedure. See "PAL_VP_INIT_ENV – PAL Initialize Virtual Environment (268)" on page 2:478 for details.

Certain global virtualization optimizations have VPD synchronization requirements. Please refer to the corresponding section of each global virtualization optimizations for more details on these requirements.

11.7.4.1.1 Virtualization Opcode Optimization

Virtualization opcode optimization is always enabled. Opcode information is provided to the VMM during PAL intercepts in the virtual environment. In some processor implementations, the opcode provided may not be guaranteed to be the opcode that triggered the intercept; virtual machine monitors can determine whether this is guaranteed from the *vp_env_info* return value of PAL_VP_ENV_INFO.

Table 11-24 and Table 11-16, "Virtual Processor Descriptor (VPD)" on page 2:326 shows the synchronization requirements and the VPD states that will be accessed for this optimization.

Table 11-24.Synchronization Requirements for Virtualization Opcode Optimization

VPD Resource	Synchronization Required
vpsr.ic	Write
vpsr.si	Write
vifa	Write
vitir	Write

11.7.4.1.2 Virtualization Cause Optimization

Virtualization cause optimization is enabled by the *cause* bit in the *config_options* parameter of PAL_VP_INIT_ENV. When enabled, the causes of virtualization intercepts will be provided to the VMM during PAL intercept handoffs within the virtual environment. When disabled, no cause information will be provided during PAL intercept handoffs.

This optimization requires no special synchronization.

11.7.4.1.3 Guest MOV-from-AR.ITC Optimization

Guest MOV-from-AR.ITC optimization allows software running with PSR.vm==1 to execute MOV-from-AR.ITC instructions without any intercepts to the VMM. The value returned will be the sum of the value in the interval timer counter register (ITC) and interval timer offset register (ITO), unless a fault condition is detected (see Table 11-25, "Behavior of Guest MOV-from-AR.ITC Instruction in Virtual Environment" for details). The VMM is responsible for programming the ITO register to provide the desired return value for guest execution with PSR.vm = 1 of the MOV-from-ITC instruction when this optimization is enabled.

This optimization is enabled by the *gitc* bit in the *config_options* parameter of PAL_VP_INIT_ENV. The behavior of the guest MOV-from-AR.ITC instruction is affected by the settings of psr.ic and vpsr.ic as well, as shown in Table 11-25.

This optimization requires no special synchronization.

This optimization is not supported on all processor implementations. Software can call PAL_VP_ENV_INFO to determine the availability of this feature.

Table 11-25.Behavior of Guest MOV-from-AR.ITC Instruction in Virtual Environment

gitc[a]	psr.si	vpsr.si	MOV-from-AR.ITC when PSR.vm==1
0	0	0	No virtualization intercept – guest reads AR.ITC
	0	1	Invalid setting – behavior is undefined.
	1	0	Virtualization intercept
	1	1	If vpsr.cpl is not zero: Privileged Register fault If vpsr.cpl is zero: Virtualization intercept
1	0	0	No virtualization intercept – guest reads the sum of ITC and ITO
	0	1	If vpsr.cpl is not zero: Privileged Register fault If vpsr.cpl is zero: No Virtualization intercept – guest reads the sum of ITC and ITO
	1	0	Virtualization intercept.
	1	1	If vpsr.cpl is not zero: Privileged Register fault If vpsr.cpl is zero: Virtualization intercept

a. gitc=0: Optimization disabled; gitc=1: Optimization enabled.

11.7.4.2 Virtualization Accelerations

Table 11-26 summarizes the virtualization accelerations supported in Itanium architecture.

Table 11-26. Virtualization Accelerations Summary

Optimization	Virtualization Acceleration Control (*vac*)[a]	Description
Virtual External Interrupt Optimization	a_int	Section 11.7.4.2.1
Interruption Control Register Read Optimization	a_from_int_cr	Section 11.7.4.2.2
Interruption Control Register Write Optimization	a_to_int_cr	Section 11.7.4.2.3
MOV-from-PSR Optimization	a_from_psr	Section 11.7.4.2.4
MOV-from-CPUID Optimization	a_from_cpuid	Section 11.7.4.2.5
Cover Optimization	a_cover	Section 11.7.4.2.6
Bank Switch Optimization	a_bsw	Section 11.7.4.2.7
Virtualize all `probe` instructions	a_all_probes	Section 11.7.4.2.8
Virtualize selected `probe` instructions	a_select_probes	
Test Feature Optimization	a_tf	Section 11.7.4.2.9
Interruption Collection and User Mask Optimization	a_ic_um	Section 11.7.4.2.10

a. The Virtualization Acceleration Control (*vac*) field resides in the Virtual Processor Descriptor (VPD), see Section 11.7.1, “Virtual Processor Descriptor (VPD)” on page 2:325 for details.

For each of the accelerations, certain virtual processor control and architectural state is managed directly by hardware/firmware, and hence must be maintained in the VPD, and synchronization is required when the VMM reads or writes this state in the VPD. Some entries must be maintained in the VPD independent of any accelerations. (These are marked as [always].) See Table 11-16 for details on which VPD state is used with each of the accelerations. See Section 11.11, “PAL Virtualization Services” on page 2:486 for a description of the synchronization services.

11.7.4.2.1 Virtual External Interrupt Optimization

The virtual external interrupt optimization allows the VMM to specify the virtual highest priority pending interrupt so that a virtual external interrupt is raised on changes of vtpr or vpsr.i only when that the virtual highest priority pending interrupt is unmasked. For details on virtual external interrupts, see “Virtual External Interrupt vector (0x3400)” on page 2:187.

The virtual external interrupt optimization is enabled by the a_int bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, the VMM specifies the virtual highest priority pending interrupt (vhpi) through the PAL_VPS_SET_PENDING_INTERRUPT service described in Section 11.11.2, “PAL Virtualization Service Specifications” on page 2:488. If this optimization is disabled, processor behavior is undefined if PAL_VPS_SET_PENDING_INTERRUPT is invoked.

When this optimization is enabled, execution of `rsm` and `ssm` instructions[1], with PSR.vm==1, which modify only vpsr.i will not intercept to the VMM and vpsr.i is updated with the new value, unless a fault condition is detected (see Table 11-29 for details).

[1] The execution of `rsm` and `ssm` instructions with PSR.vm==1 is affected by both the virtual external interrupt optimization (a_int) and the interruption collection and user mask optimization (a_ic_um). Software can enable or disable both optimizations together, or enable each optimization independently. Section 11.7.4.4.1, “Virtual External Interrupt Optimization and Interruption Collection and User Mask Optimization” on page 2:349 describes the behavior when both optimizations are enabled.

When this optimization is enabled, execution of `rsm` and `ssm` instructions[1], with PSR.vm==1 and system mask equal to zero (0x0), will not intercept to the VMM unless a fault condition is detected (see Table 11-29 for details).

A virtual external interrupt is raised if the virtual highest priority pending interrupt (vhpi) is unmasked by the new vpsr.i and vtpr. If the virtual highest priority pending interrupt (vhpi) is still masked by the new vpsr.i or vtpr, no virtual external interrupt will be raised. Note that execution of MOV-to-PSR instructions with PSR.vm==1 always results in a virtualization intercept no matter which PSR bits are modified.

When this optimization is enabled, execution of `rsm` and `ssm` instructions[1], with PSR.vm==1, which modify any bits in addition to vpsr.i result in a virtualization intercepts. No virtual external interrupts are raised and the VMM is responsible for delivering a virtual external interrupt if the virtual highest priority pending interrupt (vhpi) is unmasked.

When this optimization is enabled, execution of a MOV-from-CR instruction, with PSR.vm==1, targeting vtpr reads the most recent value, unless a fault condition is detected (see Table 11-29 for details).

When this optimization is enabled, on execution of MOV-to-TPR instructions with PSR.vm==1, vtpr will be updated with the new value without handling off to the VMM, unless a fault condition is detected (see Table 11-29 for details). A virtual external interrupt is raised if the virtual highest priority pending interrupt (vhpi) is unmasked by the new vpsr.i and vtpr. No virtual external interrupt is raised if the virtual highest priority pending interrupt is still masked by vpsr.i or vtpr.

When this optimization is enabled, after completion of an instruction with PSR.vm==1 which modifies vtpr or vpsr.i (if the instruction completes without an intercept), a determination is made as to whether the new state unmasks the virtual highest priority pending interrupt. If it does, then a virtual external interrupt will be raised and the VMM will be entered on the Virtual External Interrupt vector. See Table 11-27 for details on the detection of virtual external interrupts.

Table 11-27. Detection of Virtual External Interrupts

Condition	Virtual External Interrupt
vhpi <= (!vpsr.i << 5 | vtpr.mmi <<4 | vtpr.mic)	No – virtual highest priority pending interrupt is still masked.
vhpi > (!vpsr.i << 5 | vtpr.mmi <<4 | vtpr.mic)	Yes – virtual highest priority pending interrupt is unmasked.

Synchronization is required when this optimization is enabled, see Table 11-28 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, “Virtual Processor Descriptor (VPD)” on page 2:326.

Table 11-28. Synchronization Requirements for Virtual External Interrupt Optimization

VPD Resource	Synchronization Required
vtpr	Read, Write
vpsr.i	Read, Write
vhpi	Write

Table 11-29. Interruptions when Virtual External Interrupt Optimization is Enabled

Instructions	Interruptions
`rsm`, `ssm`	When the virtual external interrupt optimization is enabled, execution of `rsm` and `ssm` instructions with PSR.vm==1 which modify only vpsr.i, may raise the following faults: • Privileged Operation fault – if vpsr.cpl is not zero
MOV-from-TPR	When the virtual external interrupt optimization is enabled, execution of MOV-from-CR instruction targeting vtpr with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register • Privileged Operation fault – if vpsr.cpl is not zero
MOV-to-TPR	When the virtual external interrupt optimization is enabled, execution of MOV-to-CR instruction targeting vtpr with PSR.vm==1, may raise the following faults: • Privileged Operation fault – if vpsr.cpl is not zero • Register NaT Consumption fault – if the NaT bit in the source register is one • Reserved Register/Field fault – if the reserved field in the vtpr is being written with a non-zero value

Note: This field cannot be enabled together with d_extint or d_psr_i virtualization disables. If this control is enabled together with any one of described disables, an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER. See Section 11.7.4.4, “Virtualization Optimization Combinations” on page 2:349 for details.

11.7.4.2.2 Interruption Control Register Read Optimization

The interruption control register read optimization is enabled by the a_from_int_cr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, and vpsr.ic is 0, software running with PSR.vm==1 will be able to read the virtual interruption control registers (vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha, viib0-1) without any intercepts to the VMM, unless a fault condition is detected (see Table 11-31 for details).

If this optimization is disabled, a read of the interruption CRs with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-30 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, “Virtual Processor Descriptor (VPD)” on page 2:326.

Table 11-30. Synchronization Requirements for Interruption Control Register Read Optimization

VPD Resource	Synchronization Required
vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha, viib0-1	Write

Table 11-31. Interruptions when Interruption Control Register Read Optimization is Enabled

Instructions	Interruptions
Move from interruption control registers	When the interruption control register read optimization is enabled, reads of interruption control registers with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if vpsr.ic is not zero or the target operand specifies GR 0 or an out-of-frame stacked register • Privileged Operation fault – if vpsr.cpl is not zero

11.7.4.2.3 Interruption Control Register Write Optimization

The interruption control register write optimization is enabled by the a_to_int_cr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, and vpsr.ic is 0, software running with PSR.vm==1 will be able to write the virtual interruption control registers (vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha, viib0-1) without any intercepts to the VMM, unless a fault condition is detected (see Table 11-33 for details).

If this optimization is disabled, a write of the interruption control registers with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-32 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, “Virtual Processor Descriptor (VPD)” on page 2:326.

Table 11-32. Synchronization Requirements for Interruption Control Register Write Optimization

VPD Resource	Synchronization Required
vipsr, visr, viip, vifa, vitir, viipa, vifs, viim, viha, viib0-1	Read

Table 11-33. Interruptions when Interruption Control Register Write Optimization is Enabled

Instructions	Interruptions
Move to interruption control registers	When the interruption control register write optimization is enabled, writes to interruption control registers with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if vpsr.ic is not zero • Privileged Operation fault – if vpsr.cpl is not zero • Register NaT Consumption fault – if the NaT bit of the source operand is one • Reserved Register/Field fault – if any reserved field in the specified control register is written with a non-zero value • Unimplemented Data Address fault – if writing to vifa and an unimplemented virtual address is specified

11.7.4.2.4 MOV-from-PSR Optimization

The MOV-from-PSR optimization is enabled by the a_from_psr bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute MOV-from-PSR instructions to read the virtual processor status register without any intercepts to the VMM; and the last value written to the vpsr will be returned, unless a fault condition is detected (see Table 11-35 for details). The value returned for the fml, mfh, ac, up and be bits are simply the values of those bits in the PSR of the logical processor, since those bits are not virtualized.

If this optimization is disabled, execution of a MOV-from-PSR instruction with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-34 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, "Virtual Processor Descriptor (VPD)" on page 2:326.

Table 11-34. Synchronization Requirements for MOV-from-PSR Optimization

VPD Resource	Synchronization Required
vpsr{36:35, 31:6} See Table 11-17, "Virtual Processor Descriptor (VPD) – VPSR" on page 2:328 for details.	Write

Table 11-35. Interruptions when MOV-from-PSR Optimization is Enabled

Instructions	Interruptions
MOV-from-PSR	When the MOV-from-PSR optimization is enabled, MOV-from-PSR instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register • Privileged Operation fault – if vpsr.cpl is not zero

Note: This field cannot be enabled together with the d_psr_i virtualization disable control (vdc) described in Section 11.7.4.3.7, "Disable PSR Interrupt-bit Virtualization" on page 2:348. If this control is enabled together with the d_psr_i control, an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER. See Section 11.7.4.4, "Virtualization Optimization Combinations" on page 2:349 for details.

11.7.4.2.5 MOV-from-CPUID Optimization

The MOV-from-CPUID optimization is enabled by the a_from_cpuid bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute MOV-from-CPUID instruction to read the virtual CPUID registers without any intercepts to the VMM; and the corresponding VCPUID value from the VPD will be returned, unless a fault condition is detected (see Table 11-37 for details).

If this optimization is disabled, execution of a MOV-from-CPUID instruction with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-36 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, "Virtual Processor Descriptor (VPD)" on page 2:326.

Table 11-36. Synchronization Requirements for MOV-from-CPUID Optimization

VPD Resource	Synchronization Required
vcpuid0-4	Write

Table 11-37. Interruptions when MOV-from-CPUID Optimization is Enabled

Instructions	Interruptions
MOV-from-CPUID	When the MOV-from-CPUID optimization is enabled, MOV-from-CPUID instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the target operand specifies GR 0 or an out-of-frame stacked register • Register NaT Consumption fault – if the NaT bit in the target register is one • Reserved Register/Field fault – if a reserved CPUID register is being read

11.7.4.2.6 Cover Optimization

The cover optimization is enabled by the a_cover bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, software running with PSR.vm==1 will be able to execute `cover` instructions without any intercepts to the VMM, unless a fault condition is detected (see Table 11-39 for details). The `cover` instruction will execute and vcr.ifs will be updated if vpsr.ic is 0.

If this optimization is disabled, execution of the `cover` instruction with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-38 for details.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, “Virtual Processor Descriptor (VPD)” on page 2:326.

Table 11-38. Synchronization Requirements for Cover Optimization

VPD Resource	Synchronization Required
vifs	Read, Write

Table 11-39. Interruptions when Cover Optimization is Enabled

Instructions	Interruptions
`cover`	When the cover optimization is enabled, `cover` instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the instruction is not the last instruction in an instruction group

11.7.4.2.7 Bank Switch Optimization

The bank switch optimization is enabled by the a_bsw bit in the Virtualization Acceleration Control (*vac*) field in the VPD. When this optimization is enabled, execution of the `bsw` instruction with PSR.vm==1 spills the currently active banked registers and the corresponding NaT bits to the VPD, and loads the other banked registers and the corresponding NaT bits from the VPD. vpsr.bn is updated to reflect the new register bank without any intercepts to the VMM, unless a fault condition is detected (see Table 11-46 for details).

If this optimization is disabled, execution of the bsw instruction with PSR.vm==1 results in a virtualization intercept.

Synchronization is required when this optimization is enabled, see Table 11-40 for details.

Table 11-40. Synchronization Requirements for Bank Switch Optimization

VPD Resource	Synchronization Required
vpsr.bn	Read, Write

Table 11-41. Interruptions when Bank Switch Optimization is Enabled

Instructions	Interruptions
bsw	When the bank switch optimization is enabled, bsw instructions with PSR.vm==1, may raise the following faults: • Illegal Operation fault – if the instruction is not the last instruction in an instruction group • Privileged Operation fault – if vpsr.cpl is not zero

Note: This field cannot be enabled together with the d_psr_i virtualization disable control (vdc) described in Section 11.7.4.3.7, "Disable PSR Interrupt-bit Virtualization" on page 2:348. If this control is enabled together with the d_psr_i control, an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER. See Section 11.7.4.4, "Virtualization Optimization Combinations" on page 2:349 for details.

11.7.4.2.8 Probe Instruction Virtualization

The probe instruction virtualization is controlled by the a_all_probes and a_select_probes bits in the Virtualization Acceleration Control (*vac*) field in the VPD.

When the a_all_probes bit is set to 1, all probe instructions running at all privilege levels with PSR.vm==1 will result in virtualization intercepts.

When the a_select_probes bit is set to 1, the following probe instructions will raise virtualization intercepts when executed with PSR.vm==1 at the most privileged level (VPSR.cpl==0):

- probe instructions in immediate-form, with immediate field equal to privilege level 0
- All probe instructions in register-form

Please refer to the instruction description page for the probe instruction for details on the usage of immediate-form and register-form of the instruction.

Note: Software cannot enable both a_all_probes and a_select_probes bits together - an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER.

The virtualization of probe instructions is not supported on all processor implementations. Software can call PAL_VP_ENV_INFO to determine the availability of this feature.

There is no synchronization requirement for the virtualization of `probe` instructions.

11.7.4.2.9 Test Feature Optimization

The test feature optimization is enabled by the a_tf bit in the Virtualization Acceleration Control (*vac*) field in the VPD.

When this optimization is enabled, test feature (`tf`) instructions running with PSR.vm==1 will test the VCPUID[4] register in the VPD. The VMM may maintain a different VCPUID[4]{63:32} value from the CPUID[4]{63:32} value of the logical processor on which the virtual processor is running.

If the VMM indicates to a guest that an instruction is not supported by clearing the corresponding bit in VCPUID[63:32], then guest execution of that instruction, when a_tf is enabled, will behave the same as it would in implementations that do not implement that instruction. See Table 11-42 for more information.

Table 11-42.Impact of clearing VCPUID bits with the a_tf optimization

VCPUID[4] bit	Instructions affected	Behavior when vCPUID[4] is bit is 0
32	`clz`	Illegal Operation fault
33	`mpy4`	Illegal Operation fault
	`mpyshl4`	Illegal Operation fault

If this optimization is disabled or not supported, execution of the test feature (`tf`) instruction with PSR.vm==1 will test the CPUID[4] register. The VMM must maintain the same VCPUID[4]{63:32} value as the CPUID[4]{63:32} value of the logical processor on which the virtual processor is running.

Synchronization is required when this optimization is enabled; see Table 11-43 for details.

This optimization is not supported on all processor implementations. Software can call PAL_VP_ENV_INFO to determine the availability of this feature.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, “Virtual Processor Descriptor (VPD)” on page 2:326.

Table 11-43.Synchronization Requirements for Test Feature Optimization

VPD Resource	Synchronization Required
vcpuid[4]{63:32}	Write

11.7.4.2.10 Interruption Collection and User Mask Optimization

The interruption collection and user mask optimization is enabled by the a_ic_um bit in the Virtualization Acceleration Control (*vac*) field in the VPD.

When this optimization is enabled and PSR.vm==1, execution of `rsm` and `ssm` instructions[1] with a mask targeting no fields other than the ic and user mask fields will not intercept to the VMM, unless a fault condition is detected (see Table 11-45 for details). The ic field in vpsr and user mask bits in PSR targeted by the mask will be updated with the new value.

When this optimization is enabled, execution of rsm and ssm instructions, with PSR.vm==1 and system mask equal to zero (0x0), will not intercept to the VMM unless a fault condition is detected (see Table 11-45 for details).

When PSR.vm==1, execution of rsm and ssm instructions[1], which modify any bits other than vpsr.ic and user mask fields will result in virtualization intercepts independent of whether this optimization is enabled or not.

Synchronization is required when this optimization is enabled; see Table 11-44 for details.

This optimization is not supported on all processor implementations. Software can call PAL_VP_ENV_INFO to determine the availability of this feature.

When this optimization is enabled, certain VPD state is accessed, as described in Table 11-16, "Virtual Processor Descriptor (VPD)" on page 2:326.

Table 11-44.Synchronization Requirements for Interrupt Collection and User Mask Optimization

VPD Resource	Synchronization Required
vpsr.ic	Read, Write

Table 11-45.Interruptions when Interrupt Collection and User Mask Optimization is Enabled

Instructions	Interruptions
rsm, ssm	When the interruption collection and user mask optimization is enabled, execution of rsm and ssm instructions with PSR.vm==1 which modify vpsr.ic and any user mask fields, may raise the following faults: •Privileged Operation fault – if vpsr.cpl is not zero

11.7.4.3 Virtualization Disables

Table 11-26 summarizes the virtualization disables supported in Itanium architecture.

Table 11-46. Virtualization Disables Summary

Disable	Virtualization Disable Control (*vdc*)[a]	Description
Disable VMSW Instruction	d_vmsw	Section 11.7.4.3.1
Disable External Interrupt Control Register Virtualization	d_extint	Section 11.7.4.3.2
Disable Breakpoint Register Virtualization	d_ibr_dbr	Section 11.7.4.3.3
Disable PMC Virtualization	d_pmc	Section 11.7.4.3.4
Disable MOV-to-PMD Virtualization	d_to_pmd	Section 11.7.4.3.5

[1] The execution of rsm and ssm instructions with PSR.vm==1 is affected by both the virtual external interrupt optimization (a_int) and the interruption collection and user mask optimization (a_ic_um). Software can enable or disable both optimizations together, or enable each optimization independently. Section 11.7.4.4.1, "Virtual External Interrupt Optimization and Interruption Collection and User Mask Optimization" on page 2:349 describes the behavior when both optimizations are enabled.

Table 11-46. Virtualization Disables Summary (Continued)

Disable	Virtualization Disable Control (*vdc*)[a]	Description
Disable ITM Virtualization	d_itm	Section 11.7.4.3.6
Disable PSR Interrupt-bit Virtualization	d_psr_i	Section 11.7.4.3.7

[a] The Virtualization Disable Control (*vdc*) field resides in the Virtual Processor Descriptor (VPD), see Section 11.7.1, “Virtual Processor Descriptor (VPD)” on page 2:325 for details.

11.7.4.3.1 Disable VMSW Instruction

The VMSW instruction disable is controlled by the d_vmsw bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, the `vmsw` instruction is disabled on the logical processor. Execution of the `vmsw` instruction, independent of the state of PSR.vm, results in a virtualization intercept.

If this control is set to 0, the `vmsw` instruction can be executed by both the VMM and guest without virtualization intercepts, if PSR.it is 1 and the `vmsw` instruction is executed on a page with access rights of 7.

11.7.4.3.2 Disable External Interrupt Control Register Virtualization

The external interrupt control register virtualization disable is controlled by the d_extint bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, the external interrupt control registers (CR65-71) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the external interruption control registers with PSR.vm==1 result in virtualization intercepts.

Note: This field cannot be enabled together with the a_int virtualization acceleration control (vac) described in Section 11.7.4.2.1, “Virtual External Interrupt Optimization” on page 2:338. If this control is enabled together with the a_int control, an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER. See Section 11.7.4.4, “Virtualization Optimization Combinations” on page 2:349 for details.

11.7.4.3.3 Disable Breakpoint Register Virtualization

The breakpoint register virtualization disable is controlled by the d_ibr_dbr bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the data and instruction breakpoint registers (DBR/IBR) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the breakpoint registers with PSR.vm==1 result in virtualization intercepts.

11.7.4.3.4 Disable PMC Virtualization

The PMC virtualization disable is controlled by the d_pmc bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the performance monitor configuration registers (PMCs) are not virtualized, and code running with PSR.vm==1 can read and write these resources directly without any intercepts to the VMM.

If this control is set to 0, accesses (reads/writes) to the performance counter configuration registers with PSR.vm==1 result in virtualization intercepts.

11.7.4.3.5 Disable MOV-to-PMD Virtualization

The MOV-to-PMD[1] virtualization disable is controlled by the d_to_pmd bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, writes to the performance monitor data registers (PMDs) are not virtualized, and code running with PSR.vm==1 can write these resources directly without any intercepts to the VMM.

If this control is set to 0, writes to the performance monitor data registers with PSR.vm==1 result in virtualization intercepts.

11.7.4.3.6 Disable ITM Virtualization

The ITM virtualization disable is controlled by the d_itm bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, writes to the Interval Timer Match (ITM) register are not virtualized, and code running with PSR.vm==1 can write this resource directly without any intercepts to the VMM.

If this control is set to 0, writes to the ITM register with PSR.vm==1 result in virtualization intercepts.

11.7.4.3.7 Disable PSR Interrupt-bit Virtualization

The PSR interrupt-bit virtualization disable is controlled by the d_psr_i bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, accesses (reads/writes) to the interrupt bit in processor state register (PSR.i) are not virtualized. Code running with PSR.vm==1 can read and write to PSR.i through `ssm` and `rsm` instructions without any intercepts to the VMM. Attempts to modify other PSR bits in addition to the interrupt bit via the `ssm` and `rsm` instructions will result in virtualization intercepts.

This control has no effect on `mov psr.l` instructions; attempts to modify the interrupt bit with the mov psr.l instruction result in virtualization intercepts.

Note: This field cannot be enabled together with a_int, a_from_psr or a_bsw virtualization accelerations. If this control is enabled together with any one of described accelerations, an error will be returned during PAL_VP_CREATE and PAL_VP_REGISTER. See Section 11.7.4.4, "Virtualization Optimization Combinations" on page 2:349 for details.

[1] The MOV-from-PMD instruction is not virtualized. Hence there is no need to provide optimizations for the MOV-from-PMD instruction.

11.7.4.4 Virtualization Optimization Combinations

Table 11-47 describes the supported combinations of virtualization accelerations and disables.

Table 11-47.Supported Virtualization Optimization Combinations

	d_vmsw	d_extint	d_ibr_dbr	d_pmc	d_to_pmd	d_itm	d_psr_i
a_int	o[a]	x[b]	o	o	o	o	x
a_from_int_cr	o	o	o	o	o	o	o
a_to_int_cr	o	o	o	o	o	o	o
a_from_psr	o	o	o	o	o	o	x
a_from_cpuid	o	o	o	o	o	o	o
a_cover	o	o	o	o	o	o	o
a_bsw	o	o	o	o	o	o	x
a_all_probes	o	o	o	o	o	o	o
a_select_probes	o	o	o	o	o	o	o
a_tf	o	o	o	o	o	o	o
a_ic_um	o	o	o	o	o	o	o

a. "o" indicates the corresponding virtualization acceleration and disable can be enabled together.
b. "x" indicates the corresponding virtualization acceleration and disable cannot be enabled together.

11.7.4.4.1 Virtual External Interrupt Optimization and Interruption Collection and User Mask Optimization

The execution of `rsm` and `ssm` instructions with PSR.vm==1 is affected by both of these optimizations:

- Virtual External Interrupt Optimization (a_int), described in Section 11.7.4.2.1, "Virtual External Interrupt Optimization", and
- Interruption Collection and User Mask Optimization (a_ic_um), described in Section 11.7.4.2.10, "Interruption Collection and User Mask Optimization".

Software can enable or disable both optimizations together, or enable each optimization independently.

When both optimizations are enabled and PSR.vm==1, `rsm` and `ssm` instructions with a mask targeting any fields in i, ic and user mask will not be intercepted to the VMM, unless a fault condition is detected, The i and ic fields in vpsr and user mask in PSR will be updated with the new value.

When PSR.vm==1, `rsm` and `ssm` instructions with a mask targeting any fields other than i, ic and user mask fields will result in virtualization intercepts independent of whether these two optimizations are enabled or not.

11.7.4.5 Virtualization Synchronizations

When certain virtualization accelerations described in Section 11.7.4.2, "Virtualization Accelerations" on page 2:337 are enabled, processor implementations can provide implementation-specific control resources to enhance the performance of virtual processors. Two PAL services are provided to synchronize the implementation-specific control resources and the resources in the VPD. There are two types of synchronizations:

1. **Read synchronization** – When a specific acceleration is enabled, after interruptions and intercepts that occur when PSR.vm was 1, the VMM must invoke PAL_VPS_SYNC_READ to synchronize the related resources before reading their values from the VPD.
2. **Write synchronization** – When a specific acceleration is enabled, the VMM must invoke PAL_VPS_SYNC_WRITE to synchronize the related resources after modifying their values in the VPD and before resuming the virtual processor.

For details on PAL_VPS_SYNC_READ and PAL_VPS_SYNC_WRITE, see Section 11.11.2, "PAL Virtualization Service Specifications" on page 2:488.

Read and/or write synchronizations are required only if the specific acceleration is enabled. For the resources that require synchronizations if the acceleration is enabled, failure to perform the proper synchronizations will result in undefined processor behavior[1].

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 11.7.4.2, "Virtualization Accelerations" on page 2:337.

No synchronization is required for any of the virtualization disables.

11.8 PAL Glossary

Corrected Error
All errors of this type are corrected by the platform or processor in either hardware or firmware. This severity is for logging purposes only. There is no architectural damage caused by the detecting and reporting functions. Corrected errors require no operating system intervention to correct the error.

Corrected Machine Check (CMC)
A corrected machine check is a machine check that as been successfully corrected by hardware and/or firmware. Information about the cause of the error is recorded, and an interrupt is set to allow the Operating System software to examine and diagnose the error. Return is controlled to the program executing at the time of the error.

Entrypoint
A firmware entrypoint is a piece of code which is triggered by a hardware event, usually the assertion of a processor pin, or the receipt of an interruption. If return to the caller is done, it is though the RFI instruction. The currently defined PAL entrypoints are PALE_RESET, PALE_INIT, PALE_PMI, and PALE_CHECK.

Fatal Error
An uncorrected error which can corrupt state, and the state information is not known. These type of errors cannot be corrected by the hardware, firmware, or the operating system. The integrity of the system, including the IO devices is not guaranteed and may require I/O device initialization and a system reboot to continue. Fatal errors may or may not be contained within the processor or memory hierarchy.

[1] Virtual machine monitors must perform all the required synchronizations specified. Virtual machine monitors not conforming to this specification are not guaranteed to work on all processor implementations.

Machine Check (MC)
A machine check is a hardware event that indicates that a hardware error or architectural violation has occurred that threatens to damage the architectural state of the machine, possibly causing data corruption. The occurrence of the error triggers the execution of firmware code which records information about the error, and attempts to recover when possible.

OLR
On line replacement. The replacement of a computer component while the system is up and running without requiring a reboot.

PAL Intercepts
Interfaces where PAL transfers control to the VMM on virtualization events (execution of virtualized instructions/operations with PSR.vm==1). For details see Section 11.7.3, "PAL Intercepts in Virtual Environment" on page 2:332.

Power-on
The reset event that occurs when the power input to the processor is applied and the reset input to the processor is asserted.

Preserved
When applied to an entrypoint, preserved means that the value contained in a register at exit from the entrypoint code is the same as the value at the time of the hardware event that caused the entrypoint to be invoked. When applied to a procedure, preserved means that the value contained in a register at exit from the procedure is the same as the value at entry to the procedure. The value may have been changed and restored before exit.

Processor Abstraction Layer (PAL)
PAL is firmware that abstracts processor implementation differences and provides a consistent interface to higher level firmware and software. PAL has no knowledge of platform implementation details.

Procedure
A firmware procedure is a piece of code which is called from other firmware or software, and which uses the return mechanism of the Itanium Runtime Calling Conventions to return to its caller.

Recoverable Error
An uncorrected error which can corrupt state, but the state information is known. Recoverable errors cannot be corrected by either the hardware or firmware. This type of error requires operating system analysis and a corrective action to recover. System operation/state may be impacted.

Reserved
When applied to a data variable, it means that the variable must not be used to convey information. All software passing the variable must place a value of zero in the variable. The occurrence of a non-zero value may cause undefined results.

When applied to a value or range of values, any values not defined in the range and specified as reserved must not be used. The occurrence of a reserved value may cause undefined results.

Reset
The reset event that occurs when the reset input to the processor is asserted.

Scratch
When applied to either an entrypoint or procedure, scratch means that the contents of the register has no meaning and need not be preserved. Further the register is available for the storage of local variables. Unless otherwise noted, the register should not be relied upon to contain any particular value after exit.

Stacked Calling Convention
The firmware calling convention which adheres fully to the Itanium Runtime Calling Conventions. To use this calling convention, the RSE must be working and usable.

Static Calling Convention
The firmware calling convention which adheres to the Itanium Runtime Calling Conventions for the static general registers, branch registers, predicate registers, but for which all other registers are unused except for the RSE control registers. The RSE is placed in enforced lazy mode, and the stacked general registers or memory are not referenced.

System Abstraction Layer (SAL)
SAL is firmware that abstracts platform implementation differences for higher level software. SAL has no knowledge of processor implementation details.

Unchanged
When applied to an entrypoint, unchanged means that the register referenced has not been changed from the time of the hardware event that caused the entrypoint to be invoked until it exited to higher level firmware or software. When applied to a procedure, unchanged means that the register referenced has not been changed from procedure entry until procedure exit. In all cases, the value at exit is the same as the value at entry or the occurrence of the hardware event.

Virtual Machine Monitor (VMM)
The VMM is the system software which implements software policies to manage/support virtualization of processor and platform resources.

Virtual Processor Descriptor (VPD)
Represents the abstraction of the processor resources of a single virtual processor. The VPD consists of per-virtual-processor control information together with performance-critical architectural state. See Section 11.7.1, “Virtual Processor Descriptor (VPD)” on page 2:325 for details.

Virtual Processor State
A memory data structure which represents the architectural state of a virtual processor. Part of the virtual processor state is located in the Virtual Processor Descriptor (VPD), and the rest is located in memory data structures maintained by the virtual machine monitor.

11.9 PAL Code Memory Accesses and Restrictions

PAL issues load and store operations to memory in the following cases with the following memory attributes:

- During machine check/INIT handling to the min-state save area memory region registered with PAL using the UC memory attribute.

- During the execution of PAL procedures to the memory buffer allocated by the caller of the procedure using the memory attribute of the address passed by the caller.
- PAL may also issue loads from the architected firmware address space and loads/stores from the registered min-state save area whenever it is executing a PAL procedure or handling PAL-based interruptions (reset, MCA, INIT and PMI). PAL code may use either the UC or WBL memory attribute when accessing these areas.

PAL code will not send IPIs that require any special support from the platform.

11.10 PAL Procedures

PAL procedures may be called by higher-level firmware and software to obtain information about the identification, configuration, and capabilities of the processor implementation, or to perform implementation-dependent functions such as cache initialization. These procedures access processor implementation-dependent hardware to return information that characterizes and identifies the processor or implements a defined function on that particular processor.

PAL procedures are implemented by a combination of firmware code and hardware. The PAL procedures are defined to be relocatable from the firmware address space. Higher level firmware and software must perform this relocation during the reset flow. The PAL procedures may be called both before and after this relocation occurs, but performance will usually be better after the relocation. In order to ensure no problems occur due to the relocation of the PAL procedures, these procedures are written to be position independent. All references to constant data done by the procedures is done in an IP relative way.

PAL procedures are provided to return information or allow configuration of the following processor features:

- Cache and memory features supported by the processor
- Processor identification, features, and configuration
- Machine Check Abort handling
- Power state information and management
- Processor self test
- Firmware utilities

PAL procedures are implemented as a single high level procedure, named PAL_PROC, whose first argument is an index which specifies which PAL procedure is being called. Indices are assigned depending on the nature of the PAL procedure being referenced, according to Table 11-48.

Table 11-48. PAL Procedure Index Assignment

Index	Description
0	Reserved
1 - 255	Architected procedures; static register calling conventions
256 - 511	Architected procedures; stacked register calling conventions
512 - 767	Implementation-specific procedures; static registers calling conventions
768 - 1023	Implementation-specific procedures; stacked register calling conventions
1024 +	Reserved

The assignment of indices for all architected procedures is controlled by this document. The assignment of indices for implementation-specific procedures is controlled by the specific processor for which the procedures are implemented. No implementation-specific procedure calls are required for the correct operation of a processor. No SAL or operating system code should ever have to call an implementation-specific procedure call for normal activity. They are reserved for diagnostic and bring-up software and the results of such calls may be unpredictable.

Architected procedures may be designated as required or optional. If a procedure is designated as optional, a unique return code will be returned to indicate the procedure is not present in this PAL implementation. It is the caller's responsibility to check for this return code after calling any optional PAL procedure

In addition to the calling conventions described below, PAL procedure calls may be made in physical mode (PSR.it=0, PSR.rt=0, and PSR.dt=0) or virtual mode (PSR.it=1, PSR.rt=1, and PSR.dt=1). All PAL procedures may be called in physical mode. Only those procedures specified later in this chapter may be called in virtual mode. PAL procedures written to support virtual mode, and the caller of PAL procedures written in virtual mode must obey the restrictions documented in this chapter, otherwise the results of such procedure calls may be unpredictable.

11.10.1 PAL Procedure Summary

The following tables summarize the PAL procedures by application area. Included are the name of the procedure, the index of the procedure, the class of the procedure (whether required or optional), the calling convention used for the procedure (static or stacked), and whether the procedure can be called in physical mode only, virtual mode only, or both physical and virtual modes.

On processor implementations with multiple logical processors in a physical processor package, calling a certain PAL procedures may affect resources shared by the logical processors. In the following tables, procedures that may affect resources on multiple processors are marked next to the corresponding procedure names; procedures that are not marked have no effects on other logical processors.

Table 11-49.PAL Cache and Memory Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_CACHE_FLUSH[a]	1	Req.	Static	Both	No	Flush the instruction or data caches.
PAL_CACHE_INFO	2	Req.	Static	Both	No	Return detailed instruction or data cache information.
PAL_CACHE_INIT[a]	3	Req.	Static	Phys.	No	Initialize the instruction or data caches.

Table 11-49.PAL Cache and Memory Procedures (Continued)

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_CACHE_PROT_INFO	38	Req.	Static	Both	No	Return instruction or data cache protection information.
PAL_CACHE_SHARED_INFO	43	Opt.	Static	Both	No	Returns information on which logical processors share caches.
PAL_CACHE_SUMMARY	4	Req.	Static	Both	No	Return a summary of the cache hierarchy.
PAL_MEM_ATTRIB	5	Req.	Static	Both	No	Return a list of supported memory attributes.
PAL_PREFETCH_VISIBILITY	41	Req.	Static	Both	No	Used in architected sequence to transition pages from a cacheable, speculative attribute to an uncacheable attribute. See Section 4.4.11.2, “Physical Addressing Attribute Transition – Disabling Prefetch/Speculation and Removing Cacheability” on page 2:90.
PAL_PTCE_INFO	6	Req.	Static	Both	No	Return information needed for `ptc.e` instruction to purge entire TC.
PAL_VM_INFO	7	Req.	Static	Both	No	Return detailed information about virtual memory features supported in the processor.
PAL_VM_PAGE_SIZE	34	Req.	Static	Both	No	Return virtual memory TC and hardware walker page sizes supported in the processor.
PAL_VM_SUMMARY	8	Req.	Static	Both	No	Return summary information about virtual memory features supported in the processor.
PAL_VM_TR_READ	261	Req.	Stacked	Phys.	No	Read contents of a translation register.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-50.PAL Processor Identification, Features, and Configuration Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_BRAND_INFO	274	Opt.	Stacked	Both	No	Provides processor branding information.
PAL_BUS_GET_FEATURES	9	Req.	Static	Phys.	No	Return configurable processor bus interface features and their current settings.
PAL_BUS_SET_FEATURES[a]	10	Req.	Static	Phys.	No	Enable or disable configurable features in processor bus interface.
PAL_DEBUG_INFO	11	Req.	Static	Both	No	Return the number of instruction and data breakpoint registers.
PAL_FIXED_ADDR	12	Req.	Static	Both	No	Return the fixed component of a processor’s directed address.
PAL_FREQ_BASE	13	Opt.	Static	Both	No	Return the frequency of the output clock for use by the platform, if generated by the processor.
PAL_FREQ_RATIOS	14	Req.	Static	Both	No	Return ratio of processor, bus, and interval time counter to processor input clock or output clock for platform use, if generated by the processor.
PAL_GET_HW_POLICY	48	Opt.	Static	Both	Dep.	Get current hardware resource sharing policy.
PAL_LOGICAL_TO_PHYSICAL	42	Opt.	Static	Both	No	Return information on which logical processors map to a physical processor package.
PAL_PERF_MON_INFO	15	Req.	Static	Both	No	Return the number and type of performance monitors.
PAL_PLATFORM_ADDR[a]	16	Req.	Static	Both	No	Specify processor interrupt block address and I/O port space address.
PAL_PROC_GET_FEATURES	17	Req.	Static	Phys.	No	Return configurable processor features and their current setting.

Table 11-50.PAL Processor Identification, Features, and Configuration Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_PROC_SET_FEATURES[a]	18	Req.	Static	Phys.	No	Enable or disable configurable processor features.
PAL_REGISTER_INFO	39	Req.	Static	Both	No	Return AR and CR register information.
PAL_RSE_INFO	19	Req.	Static	Both	No	Return RSE information.
PAL_SET_HW_POLICY[a]	49	Opt.	Static	Both	Dep.	Set current hardware resource sharing policy.
PAL_VERSION	20	Req.	Static	Both	No	Return version of PAL code.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-51.PAL Machine Check Handling Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_MC_CLEAR_LOG[a]	21	Req.	Static	Both	No	Clear all error information from processor error logging registers.
PAL_MC_DRAIN	22	Req.	Static	Both	No	Ensure that all operations that could cause an MCA have completed.
PAL_MC_DYNAMIC_STATE	24	Opt.	Static	Both	No	Return Processor Dynamic State for logging by SAL.
PAL_MC_ERROR_INFO	25	Req.	Static	Both	No	Return Processor Machine Check Information and Processor Static State for logging by SAL.
PAL_MC_ERROR_INJECT[a]	276	Opt.	Stacked	Both	Dep.	Injects the requested processor error or returns information on the supported injection capabilities for this particular processor implementation.
PAL_MC_EXPECTED	23	Req.	Static	Phys.	No	Set/Reset Expected Machine Check Indicator.
PAL_MC_HW_TRACKING	51	Opt.	Static	Both	Dep.	Query which hardware structures are performing hardware status tracking
PAL_MC_REGISTER_MEM	27	Req.	Static	Phys.	No	Register min-state save area with PAL for machine checks and inits.
PAL_MC_RESUME	26	Req.	Static	Phys.	No	Restore minimal architected state and return to interrupted process.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-52.PAL Power Information and Management Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_GET_PSTATE	262	Opt.	Stacked	Both	Dep.	Returns information on the performance index of the processor.
PAL_HALT	28	Opt.	Static	Phys	No	Enter the low-power HALT state or an implementation-dependent low-power state.
PAL_HALT_INFO	257	Req.	Stacked	Both	No	Return the low power capabilities of the processor.
PAL_HALT_LIGHT	29	Req.	Static	Both	No	Enter the low power LIGHT HALT state.
PAL_PSTATE_INFO	44	Opt.	Static	Both	No	Returns information about the P-states supported by the processor.
PAL_SET_PSTATE[a]	263	Opt.	Stacked	Both	Dep.	Request processor to enter power/performance state.
PAL_SHUTDOWN	45	Opt.	Static	Phys	Dep.	Puts the processor in a low power state which can be exited only by a reset event.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-53.PAL Processor Self Test Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_CACHE_LINE_INIT[a]	31	Req.	Static	Phys.	No	Initialize tags and data of a cache line for processor testing.
PAL_CACHE_READ	259	Opt.	Stacked	Phys.	No	Read tag and data of a cache line for diagnostic testing.
PAL_CACHE_WRITE[a]	260	Opt.	Stacked	Phys.	No	Write tag and data of a cache for diagnostic testing.
PAL_TEST_INFO	37	Req.	Static	Phys.	No	Returns alignment and size requirements needed for the memory buffer passed to the PAL_TEST_PROC procedure as well as information on self-test control words for the processor self tests.
PAL_TEST_PROC[a]	258	Req.	Stacked	Phys.	No	Perform late processor self test.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-54.PAL Support Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_COPY_INFO	30	Req.	Static	Phys.	No	Return information needed to relocate PAL procedures and PAL PMI code to memory.
PAL_COPY_PAL	256	Req.	Stacked	Phys.	No	Relocate PAL procedures and PAL PMI code to memory.
PAL_MEMORY_BUFFER[a]	277	Opt.	Stacked	Phys.	No	Provides cacheable memory to PAL for exclusive use during runtime.
PAL_PMI_ENTRYPOINT[a]	32	Req.	Static	Phys.	No	Register PMI memory entrypoints with processor.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

Table 11-55.PAL Virtualization Support Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_VP_CREATE	265	Opt.	Stacked	Virt.	Dep.	Initializes a new VPD for the operation of a new virtual processor in the virtual environment.
PAL_VP_ENV_INFO	266	Opt.	Stacked	Virt.	Dep.	Returns the parameters needed to enter a virtual environment.
PAL_VP_EXIT_ENV	267	Opt.	Stacked	Virt.	Dep.	Allows a logical processor to exit a virtual environment.
PAL_VP_INFO	50	Opt.	Static	Phys.	No	Returns information about virtual processor features.
PAL_VP_INIT_ENV	268	Opt.	Stacked	Virt.	Dep.	Allows a logical processor to enter a virtual environment.
PAL_VP_REGISTER	269	Opt.	Stacked	Virt.	Dep.	Register a different host IVT for the virtual processor.
PAL_VP_RESTORE	270	Opt.	Stacked	Virt.	Dep.	Restore virtual processor state on the logical processor.

Table 11-55.PAL Virtualization Support Procedures (Continued)

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_VP_SAVE	271	Opt.	Stacked	Virt.	Dep.	Save virtual processor state on the logical processor.
PAL_VP_TERMINATE	272	Opt.	Stacked	Virt.	Dep.	Terminates operation for the specified virtual processor.

11.10.2 PAL Calling Conventions

The following general rules govern the definition of the PAL procedure calling conventions.

11.10.2.1 Overview of Calling Conventions

There are two calling conventions supported for PAL procedures: static registers only and stacked registers. Any single PAL procedure will support only one of the two calling conventions. In addition, PAL procedure may be called in either physical mode (PSR.it=0, PSR.rt=0, and PSR.dt=0) or virtual mode (PSR.it=1, PSR.rt=1, and PSR.dt=1).

11.10.2.1.1 Static Registers Only

This calling convention is intended for boot time usage before main memory may be available or error recovery situations, where memory or the RSE may not be reliable. All parameters are passed in the lower 32 static general registers. The stacked registers will not be used within the procedure. No memory arguments may be passed as parameters to or from PAL procedures written using the static register calling convention. To avoid RSE activity, static register PAL procedures must be called with the br.cond instruction, not the br.call instruction. Please refer to Table 11-59 for a detailed list of the general register usage for static registers only calling convention.

11.10.2.1.2 Stacked Registers

This calling convention is intended for usage after memory has been made available, and for procedures which require memory pointers as arguments. The stacked registers are also used for parameter passing and local variable allocation. This convention conforms to the *Itanium Software Conventions and Runtime Architecture Guide*. Thus, procedures using the stacked register calling convention can be written in the C language. There are two exceptions to the runtime conventions.

1. The first argument to the procedure must also be copied to GR28 prior to making the procedure call. This allows procedures written using both static and stacked register calling conventions to call a single PAL_PROC entrypoint. This should be accomplished by having the stacked register procedures call a stub module which copies GR32 to GR28, then call PAL_PROC. It is the responsibility of the caller to provide this stub. Please refer to Table 11-60 for a detailed list of the general register usage for the stacked register calling convention.
2. Floating point registers 32-127 are preserved (rather than scratch, as in the normal Itanium Software Conventions), except on the PAL_TEST_PROC procedure. This allows OSs to avoid having to save and restore these registers around a stacked-convention PAL procedure call.

11.10.2.1.3 Making PAL Procedure Calls in Physical or Virtual Mode

PAL procedure calls which are made in physical mode must obey the calling conventions described in this chapter, but there are no additional restrictions beyond those noted above. PAL procedure calls made in virtual mode must have the region occupied by PAL_PROC virtually mapped with an ITR. It needs to map this same area with either a DTR or DTC using the same translation as the ITR. In addition, it must also provide a DTR or DTC mapping for any memory buffer pointers passed as arguments to a procedure. It is the responsibility of the caller to provide these mappings.

If the caller chooses to map the PAL_PROC area or any memory pointers with a DTC it must call the procedure with PSR.ic = 1 to handle any TLB faults that could occur. The PAL_PROC code needs to be mapped with an ITR. Unpredictable results may occur if it is mapped with an ITC register.

11.10.2.1.4 Dependence on the PAL Memory Buffer

The PAL_MEMORY_BUFFER procedure must be called to establish a PAL memory buffer before calling certain PAL procedures that are dependent on the buffer.

11.10.2.2 Processor State

The PAL procedures are only available to the code running at privilege level 0. They must run in physical mode (unless specified as callable in virtual mode). PAL procedures are not interruptible by external interrupt or NMI, since PSR.i must be 0 when the PAL procedure is called. PAL procedures are not interruptible by PMI events, if PSR.ic is 0. If PSR.ic is 1, PAL procedures can be interrupted by PMI events. PAL procedures can be interrupted by machine checks and initialization events.

Generally PAL procedures will not enable interruptions not already enabled by the caller. Any PAL call that might cause interruptions (besides data TLB faults, see Section 11.10.2.1.3, "Making PAL Procedure Calls in Physical or Virtual Mode"), must install an IVA handler to handle them. PAL_TEST_PROC may generate any interruptions it needs to test the processor.

Table 11-56 defines the requirements for the PSR at entry to and at exit from a PAL procedure call. The operating system must follow the state requirements for PSR shown below. PAL procedure calls made by SAL may impose additional requirements. PAL_TEST_PROC may change PSR bits shown as unchanged in order to test the processor. These bits will be preserved in this case. PSR bits are described in increasing bit number order. Any PSR bit numbers not specified are reserved and unchanged.

Table 11-56. State Requirements for PSR

PSR Bit	Description	Entry	Exit	Class
be	big-endian memory access enable	0	0	preserved
up	user performance monitor enable	C	C	unchanged
ac	alignment check	C	C	preserved
mfl	floating-point registers f2-f31 written	C	C	preserved
mfh	floating-point registers f32-f127 written	C	C	preserved
ic	interruption state collection enable	0	0	unchanged
		1	1	preserved
i	interrupt enable	0	0	unchanged

Table 11-56. State Requirements for PSR (Continued)

PSR Bit	Description	Entry	Exit	Class
pk	protection key validation enable	C	C	unchanged
dt	data address translation enable[a]	0	0	unchanged
		1	1	preserved
dfl	disabled FP register f2 to f31	0	0	unchanged
dfh	disabled FP register f32 to f127[b]	0	0	unchanged
		1	1	unchanged
sp	secure performance monitors	C	C	unchanged
pp	privileged performance monitor enable	C	C	unchanged
di	disable ISA transition	C	C	preserved
si	secure interval timer	C	C	unchanged
db	debug breakpoint fault enable	0	0	unchanged
lp	lower-privilege transfer trap enable	0	0	unchanged
tb	taken branch trap enable	0	0	unchanged
rt	register stack translation enable[a]	0	0	unchanged
		1	1	preserved
cpl	current privilege level	0	0	unchanged
is	instruction set	0	0	preserved
mc	machine check abort mask[c]	0	0	preserved
		1	1	unchanged
it	instruction address translation enable[a]	0	0	unchanged
		1	1	preserved
id	instruction debug fault disable	0	0	unchanged
da	data access and dirty-bit fault disable	0	0	unchanged
dd	data debug fault disable	0	0	unchanged
ss	single step trap enable	0	0	unchanged
ri	restart instruction	0	0	preserved
ed	exception deferral	0	0	preserved
bn	register bank	1	1	preserved
ia	instruction access-bit fault disable	0	0	unchanged
vm	processor virtualization	0	0	unchanged

a. PAL procedures which are called in physical mode must remain in physical mode for the duration of the call. PAL procedures which are called in virtual mode, may perform specific actions in physical mode, but must return to the same virtual mode state before returning from the call.

b. PAL_TEST_PROC and an implementation-specific authentication procedure call need to be called with PSR.dfh equal to 0. If they are not they will return invalid argument. All other PAL procedure calls may be called with PSR.dfh equal to 0 or 1.

c. Most PAL runtime procedures should be called with PSR.mc = 0 except for code flow involved in handling machine checks.

11.10.2.2.1 Definition of Terms

The terms used in the definition of the requirements have the following meaning:

Table 11-57. Definition of Terms

Term	Description
entry	Start of the first instruction of the PAL procedure.
exit	Start of the first instruction after return to caller’s code.

Table 11-57. Definition of Terms

Term	Description
0	Must be zero at entry to the procedure or on exit from the procedure. If the value at entry is not zero, the procedure may return an illegal argument or execute in an undefined manner.
1	Must be one at entry to the procedure or on exit from the procedure. If the value at entry is not one, the procedure may return an illegal argument or execute in an undefined manner.
reserved	When any input parameter is listed as reserved, this value must be zero. If an input value has a range of values, any values outside the range, listed as reserved, must not be used. For either case, the PAL procedure may return an illegal argument or execute in an undefined manner.
C	The state of bits marked with C are defined by the caller. If the value at exit is also C, it must be the same as the value at entry.
unchanged	The PAL procedure must not change these values from their entry values during execution of the procedure.
scratch	The PAL procedure may modify these values as necessary during execution of the procedure. The caller cannot rely on these values.
preserved	The PAL procedure may modify these values as necessary during execution of the procedure. However, they will be restored to their entry values prior to exit from the procedure.

11.10.2.2.2 System Registers

The PAL_TEST_PROC procedure may change system registers marked as unchanged in order to fully test the processor. When this is done, the values of the system registers will be preserved.

Table 11-58. System Register Conventions

Name	Description	Class
DCR	Default Control Register	preserved
ITM	Interval Timer Match Register	unchanged
IVA	Interruption Vector Address	preserved[a]
PTA	Page Table Address	preserved
GPTA	Guest Page Table Address	preserved
IPSR	Interruption Processor Status Register	scratch
ISR	Interruption Status Register	scratch
IIP	Interruption Instruction Bundle Pointer	scratch
IFA	Interruption Faulting Address	scratch
ITIR	Interruption TLB Insertion Register	scratch
IIPA	Interruption Instruction Previous Address	scratch
IFS	Interruption Function State	scratch
IIM	Interruption Immediate Register	scratch
IHA	Interruption Hash Address	scratch
IIB0-1	Interruption Instruction Bundle Registers	scratch
LID	Local Interrupt ID	unchanged
IVR	Interrupt Vector Register (read only)	unchanged
TPR	Task Priority Register	unchanged
EOI	End Of Interrupt	unchanged
IRR0-IRR3	Interrupt Request Registers 0-3 (read only)	unchanged
ITV	Interval Timer Vector	unchanged
PMV	Performance Monitoring Vector	unchanged

Table 11-58. System Register Conventions (Continued)

Name	Description	Class
CMCV	Corrected Machine Check Vector	unchanged
LRR0-LRR1	Local Redirection Registers 0-1	unchanged
RR	Region Registers	preserved
PKR	Protection Key Registers	preserved
TR	Translation Registers	unchanged[b]
TC	Translation Cache	scratch
IBR/DBR	Break Point Registers	preserved[c]
PMC	Performance Monitor Control Registers	preserved
PMD	Performance Monitor Data Registers	unchanged[d]

[a] On some implementations, PAL virtualization support procedures may program IVA to a different value. Refer to the description of the PAL virtualization procedures for details.

[b] If an implementation provides a means to read TRs for PAL, this should be preserved.

[c] The PAL_MC_ERROR_INJECT may modify these registers if the caller is using the triggering capability. Refer to "PAL_MC_ERROR_INJECT – Inject Processor Error (276)" on page 2:421 for more information.

[d] No PAL procedure writes to the PMD. Depending on the PMC, the PMD may be kept counting performance monitor events during a procedure call. The exception is PAL_TEST_PROC, which tests the performance counters.

11.10.2.2.3 General Registers

PAL will use one of two general register calling conventions described in Section 11.10.2.1, "Overview of Calling Conventions" on page 2:358, depending on the availability of memory and the stacked registers at the time of the call. The following tables describe the contents of the general registers.

Table 11-59. General Registers – Static Calling Convention

Register	Conventions
GR0	always 0
GR1	preserved
GR2 - GR3	scratch, used with 22 bit immediate add
GR4 - GR7	preserved
GR8 - GR11	scratch, procedure return value
GR12	preserved
GR13	unchanged
GR14 - GR27	scratch
GR28 - GR31	input arguments, scratch (PAL index must be passed in GR28)
Bank 0 Registers (GR16 - GR23)	preserved
Bank 0 Registers (GR 24 - GR31)	scratch
GR32 - GR127	unchanged

Table 11-60. General Registers – Stacked Calling Conventions

Register	Conventions
GR0	always 0
GR1	preserved
GR2 - GR3	scratch, used with 22 bit immediate add
GR4 - GR7	preserved

Table 11-60. General Registers – Stacked Calling Conventions (Continued)

Register	Conventions
GR8 - GR11	scratch, procedure return value
GR12	special, stack pointer (sp)
GR13	special, thread pointer (tp)
GR14 - GR27	scratch
GR28	input argument, scratch (PAL Index must be passed in GR28)
GR29-GR31	scratch
Bank 0 Registers (GR16 - GR23)	preserved
Bank 0 Registers (GR 24 - GR31)	scratch
GR32 - GR127	stacked registers; in0 - in95: input arguments (PAL index must be in0) loc0 - loc95: local variables out0 - out95: output arguments

The caller must initialize SP for physical and virtual procedure calls only prior to calling a PAL procedure. A minimum 8 KB of room must be available for the stack space of the PAL procedure. The caller to a PAL procedure should set up the RSE backing store before making any procedure calls using the stacked calling conventions. The backing store memory should have a minimum of 8 KB of room for RSE spills.

PAL shall be called with PSR.bn=1. The GR specifications for GR16 through GR31 are for bank one. The bank zero register requirements are specified as a separate line item.

11.10.2.2.4 Floating-point Registers

Floating point registers 32-127 are preserved. PAL must either not use these, or must save and restore them, except on the PAL_TEST_PROC procedure, which may overwrite these registers without preserving them. The remainder of the floating-point register conventions are the same as those of the *Itanium Software Conventions and Runtime Architecture Guide*.

11.10.2.2.5 Predicate Registers

The conventions for the predicate registers follow the *Itanium Software Conventions and Runtime Architecture Guide*.

11.10.2.2.6 Branch Registers

The conventions for the branch registers follow the *Itanium Software Conventions and Runtime Architecture Guide*.

11.10.2.2.7 Application Registers

Table 11-61. Application Register Conventions

Register	Description	Class
KR0-7	Kernel Registers	unchanged
RSC	Register Stack Configuration Register	unchanged
BSP	Backing Store Pointer (read only)	unchanged[a]

Table 11-61. Application Register Conventions

Register	Description	Class
BSPSTORE	Backing Store Pointer for Memory Stores	unchanged[a]
RNAT	RSE NaT Collection Register	unchanged[a]
FCR	IA-32 Floating-point Control Registers	preserved
EFLAG	IA-32 EFLAG register	preserved
CSD	IA-32 Code Segment Descriptor	preserved
SSD	IA-32 Stack Segment Descriptor	preserved
CFLG	IA-32 Combined CR0 and CR4 Register	preserved
FSR	IA-32 Floating-point Status Register	preserved
FIR	IA-32 Floating-point Instruction Register	preserved
FDR	IA-32 Floating-point Data Register	preserved
CCV	Compare and Exchange Compare Value Register	scratch
UNAT	User NaT Collection Register	according to GR class
FPSR	Floating-point Status Register	preserved
ITC	Interval Time Counter	unchanged[b]
RUC	Resource Utilization Counter	unchanged[c]
PFS	Previous Function State	preserved
LC	Loop Counter Register	preserved
EC	Epilog Counter Register	preserved

[a] BSP, BSPSTORE, and RNAT may not be changed by PAL, but the value at exit may be different due to RSE activity. PAL_TEST_PROC is an exception to this rule, and the RSE contents may not be relied on after making this procedure call.

[b] No PAL procedure writes to the ITC. The value at exit is the value at entry plus the elapsed time of the procedure call.

[c] No PAL procedure writes to the RUC. The value at exit is the value at entry plus the number of cycles provided to the processor during the procedure call.

PAL procedures that use the static calling conventions do not use stacked registers or the RSE. Therefore RSE internal state and the CFM are unchanged by these procedures.

11.10.2.3 Return Buffers

Any addresses passed to PAL procedures as buffers for return parameters must be 8-byte aligned. Unaligned addresses may cause undefined results.

11.10.2.4 Invalid Arguments

The PAL procedure calling conventions specify rules that must be followed. These rules specify certain PSR values, they specify that reserved fields and arguments must be zero filled and specify that values not defined in a range and defined as reserved must not be used.

If the caller of a PAL procedure does not follow these rules, an invalid argument return value may be returned or undefined results may occur during the execution of the procedure. If the caller passes in a PAL procedure index value that is not defined, PAL will return an Unimplemented procedure (-1) status to the caller.

11.10.3 PAL Procedure Specifications

The following pages provide detailed interface specifications for each of the PAL procedures defined in this document. Included in the specification are the input parameters, the output parameters, and any required behavior.

PAL_BRAND_INFO – Provides Processor Branding Information (274)

Purpose: Provides processor branding information.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_BRAND_INFO within the list of PAL procedures.
info_request	Unsigned 64-bit integer specifying the information that is being requested. (See Table 11-62)
address	Unsigned 64-bit integer specifying the address of the 128-byte block to which the processor brand string shall be written.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BRAND_INFO procedure.
brand_info	Brand information returned. The format of this value is dependent on the input values passed.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-6	Input argument is not implemented
-9	Call requires PAL memory buffer

Description: PAL_BRAND_INFO procedure calls are used to ascertain the processor branding information.

The *info_request* input argument for PAL_BRAND_INFO describes which processor branding information is being requested. The *info_request* values are split into two categories: architected and implementation-specific. The architected *info_request* have values from 0-15. The implementation-specific *info_request* have values 16 and above. The architected *info_request* are described in this document. The implementation-specific *info_request* are described in processor-specific documentation.

This call returns the processor brand information as requested with the *info_request* argument. Table 11-62 describes the values.

Table 11-62. Processor Brand Information Requested

Value	Description
0	The ASCII brand identification string will be copied to the address specified in the address input argument. The processor brand identification string is defined to be a maximum of 128 characters long; 127 bytes will contain characters and the 128th byte is defined to be NULL (0). A processor may return less than the 127 ASCII characters as long as the string is null terminated. The string length will be placed in the *brand_info* return argument.
All Other Values	Reserved

This procedure will return an invalid argument if an unsupported *info_request* argument is passed as an input or a -6 if the requested information was not available on the current processor.

PAL_BUS_GET_FEATURES – Get Processor Bus Dependent Configuration Features (9)

Purpose: Provides information about configurable processor bus features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_BUS_GET_FEATURES within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BUS_GET_FEATURES procedure.
features_avail	64-bit vector of features implemented. See Table 11-63. (1=implemented, 0=not implemented)
feature_status	64-bit vector of current feature settings. See Table 11-63.
feature_control	64-bit vector of features controllable by software. (1=controllable, 0= not controllable)

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Table 11-63 defines the set of possible bus interface features and their bit position in the return vector. Different busses will implement similar features in different ways. For example, data error detection may be implemented by ECC or parity. In other cases, certain features may be tied together. In this case, enabling any one feature in a group will enable all features in the group, and similarly, disabling any one feature in a group will disable all features. Caller algorithms should be written to obtain desired results in these instances. Only those configuration features for which a 1 is returned in *feature_control* can be changed via PAL_BUS_SET_FEATURES.

For all values in Table 11-63, the *Class* field indicates whether a feature is required to be available (Req.) or is optional (Opt.). The *Control* field indicates which features are required to be controllable. These features will either be controllable through this PAL call or through other hardware means like forcing bus pins to a certain value during processor reset. The *control* field applies only when the feature is available. PALE_CHECK and PALE_INIT should not modify these features. An operating system should not modify bus features without detailed information about the platform it is running on.

Table 11-63. Processor Bus Features

Bits	Class	Control	Description
63	Opt.	Req.	Disable Bus Data Error Checking. When 0, bus data errors are detected and single bit errors are corrected. When 1, no error detection or correction is done.
62	Opt.	Req.	Disable Bus Address Error Signalling. When 0, bus address errors are signalled on the bus. When 1, no bus errors are signalled on the bus. If Disable Bus Address Error Checking is 1, this bit is ignored.
61	Opt.	Req.	Disable Bus Address Error Checking. When 0, bus errors are detected, single bit errors are corrected., and a CMCI or MCA is generated internally to the processor. When 1, no bus address errors are detected or corrected.
60	Opt.	Req.	Disable Bus Initialization Event Signaling. When 0, bus protocol errors (BINIT#) are signaled by the processor on the bus. When 1, bus protocol errors (BINIT#) are not signaled on the bus. If Disable Bus Initialization Event Checking is 1, this bit is ignored.
59	Opt.	Req.	Disable Bus Initialization Event Checking. When 0, bus protocol errors (BINIT#) are detected and sampled and an MCA is generated internally to the processor. When 1, the processor will ignore bus protocol error conditions (BINIT#).
58	Opt.	Req.	Disable Bus Requester Bus Error Signalling. When 0, BERR# is signalled if a bus error is detected. When 1, bus errors are not signalled on the bus.
57	Opt.	Req.	Disable Bus Requester Internal Error Signalling. When 0, BERR# is signalled when internal processor requestor initiated bus errors are detected. When 1, internal requester bus errors are not signalled on the bus.
56	Opt.	Req.	Disable Bus Error Checking. When 0, the processor takes an MCA if BERR# is asserted. When 1, the processor ignores the BERR# signal.
55	Opt.	Req.	Disable Response Error Checking. When 0, the processor asserts BINIT# if it detects a parity error on the signals which identify the transactions to which this is a response. When 1, the processor ignores parity on these signals.
54	Opt.	Req.	Disable Transaction Queuing. When 0, the in-order transaction queue is limited only by the number of hardware entries. When 1, the processor's in-order transactions queue is limited to one entry.
53	Opt.	Req.	Enable a bus cache line replacement transaction when a cache line in the exclusive state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.
52	Opt.	Req.	Enable a bus cache line replacement transaction when a cache line in the shared or exclusive state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.
51:32	N/A	N/A	Reserved
31	Opt.	Opt.	Enable Half transfer rate. When 0, the data bus is configured at the 2x data transfer rate.When 1, the data bus is configured at the 1x data transfer rate,
30	Opt.	Req.	Disable Bus Lock Mask. When 0, the processor executes locked transactions atomically. When 1, the processor masks the bus lock signal and executes locked transactions as a non-atomic series of transactions.
29	Opt.	Req.	Request Bus Parking. When 0, the processor will deassert bus request when finished with each transaction. When 1, the processor will continue to assert bus request after it has finished, if it was the last agent to own the bus and if there are no other pending requests.
28:0	N/A	N/A	Reserved

PAL_BUS_SET_FEATURES – Set Processor Bus Dependent Configuration Features (10)

Purpose: Enables/disables specific processor bus features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_BUS_SET_FEATURES within the list of PAL procedures.
feature_select	64-bit vector denoting desired state of each feature (1=select, 0=non-select).
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BUS_SET_FEATURES procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: PAL_BUS_GET_FEATURES should be called to ascertain the implemented processor bus configuration features, their current setting, and whether they are software controllable, before calling PAL_BUS_SET_FEATURES. The list of possible processor features is defined in Table 11-63. Attempting to enable or disable any feature that cannot be changed will be ignored.

PAL_CACHE_FLUSH – Flush Data or Instruction Caches (1)

Purpose: Flushes the processor instruction or data caches.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_FLUSH within the list of PAL procedures.
cache_type	Unsigned 64-bit integer indicating which cache to flush. See Table 11-64.
operation	Formatted bit vector indicating the operation of this call. See Figure 11-1.
progress_indicator	Unsigned 64-bit integer specifying the starting position of the flush operation.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_FLUSH procedure.
vector	Unsigned 64-bit integer specifying the vector number of the pending interrupt.
progress_indicator	Unsigned 64-bit integer specifying the starting position of the flush operation.
Reserved	0

Status:

Status Value	Description
2	Call completed without error, but a PMI was taken during the execution of this procedure.
1	Call has not completed flushing due to a pending interrupt
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Flushes the instruction or data caches controlled by the processor as specified by the *cache_type* parameter. Encoding for the *cache_type* parameter follows:

Table 11-64. *cache_type* Encoding

Value	Description
1	Flush all caches containing instructions.
2	Flush all caches containing data.
3	Flush all caches (instruction and data).
4	Make local instruction caches coherent with the data caches.

All other values of *cache_type* are reserved. If the cache is unified, both instruction and data lines are flushed, regardless of the value of *cache_type*.

Flushing all caches containing instructions, causes the instruction and unified caches to be flushed. Flushing all caches containing data, causes all data and unified caches to be flushed. Flushing all caches causes all data, instruction, and unified caches to be flushed.

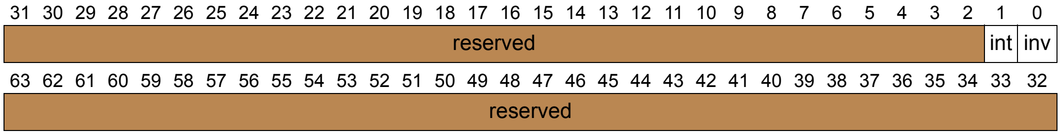
When the caller specifies to make local instruction caches coherent with the data caches, this procedure will ensure that the instruction caches on the processor that this procedure call was made, will see the effects of stores to instruction code performed by this processor. This procedure is not required to ensure coherency of instruction caches on other processors in the system when this input argument is used. Refer to Section 4.4.3, "Cacheability and Coherency Attribute" on page 2:77 for more information on stores and their coherency requirements with local instruction caches.

The effects of flushing data and unified caches is broadcast throughout the coherence domain. The effects of flushing instruction caches may or may not be broadcast throughout the coherence domain. The procedure will perform the necessary serialization and synchronization as required by the architecture.

This call does not ensure that data in the processors coalescing buffers are flushed to memory. See Section 4.4.5, "Coalescing Attribute" on page 2:78 on how to flush the coalescing buffers.

The *operation* parameter controls how this call will operate. The *operation* parameter has the following format:

Figure 11-1. *operation* Parameter Layout



- *inv* – 1 bit field indicating whether to invalidate clean lines in the cache.

  If this bit is 0, all modified cache lines for the specified *cache_type* are copied back to memory. Optimally, modified and non-modified cache lines are left valid in the specified cache in a clean (non-modified) state. However, based on the processor implementation, cache lines in the specified cache may alternatively be invalidated.

  If this bit is 1, all modified cache lines for the specified *cache_type* are flushed by copying the cache line to memory. All cache lines in the specified cache are then invalidated.

  If *cache_type* is equal to 4 (make local instruction caches coherent with the data caches) the *inv* bit will be ignored.

  Table 11-65 will clarify the effects of the *inv* bit. The modified state represents a cache line that contains modified data. The clean state represents a cache line that contains no modified data.

- *int* – 1 bit field indicating if the processor will periodically poll for external interrupts while flushing the specified *cache_type*(s).

  If this bit is a 0, unmasked external interrupts will not be polled. The processor will ignore all pending unmasked external interrupts until all cache lines in the specified *cache_type*(s) are flushed. Depending on the size of the processor's caches, bus bandwidth and implementation characteristics, flushing the caches can take a long period of time, possibly delaying interrupt response times and potentially causing I/O devices to fail.

  If this bit is a 1, external interrupts will be polled periodically and will exit the procedure if one is seen. If an unmasked external interrupt becomes pending, this procedure will return and allow the caller to service the interrupt before all cache lines in the specified *cache_type*(s) are flushed.

Table 11-65. Cache Line State when *inv* = 0

Old State	New State	Comments
Invalid	Invalid	
Clean	Clean[a]	
Modified	Clean[a]	Modified data is copied back to memory

[a] Based on the processor implementation the cache line can be invalidated or left in a model-specific clean state

Table 11-66. Cache Line State when *inv* = 1

Old State	New State	Comments
Invalid	Invalid	
Clean	Invalid	
Modified	Invalid	Modified data is copied back to memory.

The *progress_indicator* is an unsigned 64-bit integer specifying the starting position of the flush operation. Values in this parameter are model specific and will vary across processor implementations.

The first time this procedure is called, the *progress_indicator* must be set to zero. If this procedure exits due to an external interrupt and this procedure is then again called to resume flushing, the *progress_indicator* must be set to the value previously returned by PAL_CACHE_FLUSH. Software must program no value other than zero or the value previously returned by PAL_CACHE_FLUSH otherwise behavior is undefined.

This procedure makes one flush pass through all caches specified by *cache_type* and all sets and associativities within those caches. The specified *cache_type*(s) are ensured to be flushed only of cache lines resident in the caches prior to PAL_CACHE_FLUSH initially being called with the *progress_indicator* set to 0.

This procedure ensures that prefetches initiated prior to making this call with *progress_indicator* set to 0 are flushed based on the *cache_type* argument passed.

- If *cache_type* specifies to flush all instruction caches then the call ensures all prior instruction prefetches are flushed.
- If *cache_type* specifies to flush all data caches then the call ensures all prior data prefetches are flushed.
- If *cache_type* specifies to flush all caches then the call ensures all prior instruction and data prefetches are flushed from the caches.
- If *cache_type* specifies to make local instruction caches coherent with the data caches, then the call will ensure all prior instruction prefetches are flushed.

Due to the following conditions, software cannot assume that when this procedure completes the entire flush pass that the specified *cache_type*(s) are empty of all clean and/or modified cache lines.

- After an interruption, the flush pass resumes at the interruption point (specified by *progress_indicator*). Due to execution of the interrupt handlers during the flush pass, the specified caches may contain new and possibly modified cache lines in sections of the caches already flushed. The caller specifies if this procedure should poll for interrupts via the *int* bit of the *operation* parameter.
- Prior prefetches initiated before this procedure is called are disabled and flushed from the cache as described above. However, if a speculative translation exists in either the ITLB or DTLB, speculative instruction or data prefetch operation could immediately reload a non-modified cache line after it was flushed. To ensure prefetches do not occur, software must remove all speculative translation before

calling this routine. Alternatively, software can disable the TLBs by setting PSR.it, PSR.dt, and PSR.rt to 0.

- The specified caches may also contain PAL firmware code cache entries required to flush the cache.
- The specified caches may contain PAL and SAL PMI code if this call was made with PSR.ic = 1 and a PMI interrupt is seen during the execution of the call.
- The specified caches may contain SAL or OS machine check or INIT code if these handlers run in a cacheable mode and a machine check or INIT event is seen.
- In a processor that contains multiple logical processors, the specified caches may contain new and possibly modified cache lines in sections of the cache already flushed due to execution of instructions on other logical processors that share the specified caches. Information about how caches are shared among logical processors is described in the PAL_CACHE_SHARED_INFO procedure on page 2:382. Information about logical processors on the same physical processor package are described in the PAL_LOGICAL_TO_PHYSICAL procedure on page 2:404.

This procedure does ensure that all cache lines resident in the specified *cache_type*(s) prior to this procedure being initially called are flushed regardless of intervening external interrupts. It also ensures that prefetches initiated prior to the initial call to this procedure that affect the caches specified in *cache_type*, as described above, are flushed regardless of intervening external interrupts.

To ensure forward progress, PAL_CACHE_FLUSH advances through the cache flush sequence at least by one cache line before sampling for pending external interrupts. The amount of flushing that occurs before interrupts are polled will vary across implementations.

PAL_CACHE_FLUSH will return the following values to indicate to the caller the status of the call.

- *status* – When the call returns a 1, it indicates that the call did not have any errors but is returning due to a pending unmasked external interrupt. To continue flushing the caches, the caller must call PAL_CACHE_FLUSH again with the value returned in the *progress_indicator* return value.

  When the call returns a 0, it indicates that the call completed without any errors. All cache lines that were present in the cache (when the most recent call to PAL_CACHE_FLUSH with a *progress_indicator* of zero) are flushed and possibly invalidated. All intermediate calls must have used the proper *progress_indicator*, otherwise behavior is undefined.

  When the call returns a 2, it indicates that the call completed without any errors but that a PMI was taken during the execution of this call. This indicates to the caller that all cache lines that were present in the cache (when the most recent call to PAL_CACHE_FLUSH with a *progress_indicator* of zero) are flushed but that code and data related to handling PMIs may be present in the cache.
- *vector* – If the return status is 1 and this procedure exited due to a pending unmasked external interrupt, this field returns the interrupt vector number. The external interrupt will have been removed. The interrupt is considered to be “in-service” and software must service this interrupt for the specified vector and then issue EOI. If the return status is not 1, the values returned is undefined.
- *progress_indicator* – When the return status is 1, specifies the current position in the flush pass. The value returned is model specific and will vary across processor implementations. If the return status is not 1, the value returned is undefined.

PAL_CACHE_INFO – Get Detailed Cache Information (2)

Purpose: Returns information about a particular processor instruction or data cache at a specified level in the cache hierarchy.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_INFO within the list of PAL procedures.
cache_level	Unsigned 64-bit integer specifying the level in the cache hierarchy for which information is requested. This value must be between 0 and one less than the value returned in the *cache_levels* return value from PAL_CACHE_SUMMARY.
cache_type	Unsigned 64-bit integer with a value of 1 for instruction cache and 2 for data or unified cache. All other values are reserved.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_INFO procedure.
config_info_1	The format of *config_info_1* is shown in Figure 11-2.
config_info_2	The format of *config_info_2* is shown in Figure 11-3.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call describes in detail the characteristics of a given processor controlled cache in the cache hierarchy. It returns information in the *config_info_1* and *config_info_2* returns parameters.

The *config_info_1* return value has the following structure:

Figure 11-2. *config_info_1* Return Value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">stride</td><td colspan="8">line_size</td><td colspan="8">associativity</td><td colspan="5">reserved</td><td colspan="2">at</td><td>u</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="8">load_hints</td><td colspan="8">store_hints</td><td colspan="8">load_latency</td><td colspan="8">store_latency</td></tr>
</table>

- *u* – Bit that is 1 if the cache is unified and 0 if the cache is split.
- *at* - 2-bit field denoting cache memory attributes, as follows:

Table 11-67. Cache Memory Attributes

Value	Description
0	Write through cache
1	Write back cache
2-3	Reserved

- *associativity* – Unsigned 8-bit integer denoting the associativity of the cache. A value of 0 indicates a fully associative cache. A value of 1 indicates a direct mapped cache.
- *line_size* – Unsigned 8-bit integer denoting the binary logarithm (log2) of the minimum write back size of a flush operation to memory or the line size of the

cache if the cache contents never get flushed to memory (for example an instruction cache).

- *stride* – Unsigned 8-bit integer denoting the binary log of the most effective stride in bytes for flushing the cache.
- *store_latency* – Unsigned 8-bit integer denoting the number of cycles after issue until the value is written into the cache. If the cache cannot accept a store (like an instruction cache) the value returned will be 256 (0xff).
- *load_latency* – Unsigned 8-bit integer denoting the number of processor cycles after issue until the value may be used if it is found in the cache.
- *store_hints* – 8-bit vector denoting hints implemented by the processor store instruction. For instruction caches this bit vector will be zero indicating no store hints are supported.

Table 11-68. Cache Store Hints

Bits	Description
0	Temporal, level 1
2:1	Reserved
3	Non-temporal, all levels
7:4	Reserved

- *load_hints* – 8-bit vector denoting hints implemented by the processor load instruction.

Table 11-69. Cache Load Hints

Bits	Hint
0	Temporal, level 1
1	Non-temporal, level 1
2	Reserved
3	Non-temporal, all levels
7:4	Reserved

The *config_info_2* return value has the following structure:

Figure 11-3. *config_info_2* Return Value

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cache_size

63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32
reserved	tag_ms_bit	tag_ls_bit	alias_boundary

- *cache_size* – Unsigned 32-bit integer denoting the size of the cache in bytes.
- *alias_boundary* – Unsigned 8-bit integer indicating the binary log of the minimum number of bytes which must separate aliased addresses in order to obtain the highest performance.
- *tag_ls_bit* – Unsigned 8-bit integer denoting the least-significant address bit of the tag.
- *tag_ms_bit* – Unsigned 8-bit integer denoting the most-significant address bit of the tag.

PAL_CACHE_INIT – Initialize Caches (3)

Purpose: Initializes the processor controlled caches.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_INIT within the list of PAL procedures.
level	Unsigned 64-bit integer containing the level of cache to initialize. If the cache level can be initialized independently, only that level will be initialized. Otherwise implementation-dependent side-effects will occur.
cache_type	Unsigned 64-bit integer with a value of 1 to initialize the instruction cache, 2 to initialize the data cache, or 3 to initialize both. All other values are reserved.
restrict	Unsigned 64-bit integer with a value of 0 or 1. All other values are reserved. If *restrict* is 1 and initializing the specified level and *cache_type* of the cache would cause side-effects, PAL_CACHE_INIT will return -4 instead of initializing the cache.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_INIT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-4	Call could not initialize the specified level and *cache_type* of the cache without side-effects and *restrict* was 1.

Description: Initializes one or all the processor's caches. The effect of this procedure is to initialize the caches without causing writebacks. This procedure cannot be used where coherency is required because any data in the caches will be lost.

The *level* argument must either be -1, indicating all cache levels, or a non-negative number indicating the specific level to initialize. In the latter case, *level* must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY:

Table 11-70. PAL_CACHE_INIT *level* Argument Values

Value	Description
-1	Initializes all cache levels (*cache_type* and *restrict* are ignored)
0 to N	Initialize only the specified cache level.

The *restrict* argument specifies how to handle potential side-effects, where:

Table 11-71. PAL_CACHE_INIT *restrict* Argument Values

Value	Description
0	No restriction: initialize the specified level and *cache_type* of the cache, even if doing so will cause side effects in other caches.
1	Restrict initialization to the specified level and *cache_type* without side effects to other cache levels. If this cannot be done, return -4.

All other values of *restrict* are reserved.

PAL_CACHE_LINE_INIT – Initialize a Data Cache Line (31)

Purpose: Initializes the tags and data of a data or unified cache line of a processor controlled cache to known values without the availability of backing memory.

Calling Conv: Static

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_LINE_INIT within the list of PAL procedures.
address	Unsigned 64-bit integer value denoting the physical address from which the physical page number is to be generated. The address must be an implemented physical address, bit 63 must be zero.
data_value	64-bit data value which is used to initialize the cache line.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_LINE_INIT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: A line in the data or unified cache is initialized to the values passed in the arguments of this procedure. The physical page number of the line is derived from the *address* value passed. The tags of the line are set to Private, Dirty, and Valid. The cache line is initialized using *data_value* repeated until it fills the line. This procedure replicates *data_value* to a size equal to the largest line size in the processor-controlled cache hierarchy.

This procedure call cannot be used where coherency is required.

PAL_CACHE_PROT_INFO – Get Detailed Cache Protection Information (38)

Purpose: Returns protection information about a particular processor instruction or data cache at a specified level in the cache hierarchy.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_PROT_INFO within the list of PAL procedures.
cache_level	Unsigned 64-bit integer specifying the level in the cache hierarchy for which information is requested. This value must be between 0 and one less than the value returned in the *cache_levels* return value from PAL_CACHE_SUMMARY.
cache_type	Unsigned 64-bit integer with a value of 1 for instruction cache and 2 for data or unified cache. All other values are reserved.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_PROT_INFO procedure.
config_info_1	The format of *config_info_1* is shown in Figure 11-4.
config_info_2	The format of *config_info_2* is shown in Figure 11-5.
config_info_3	The format of *config_info_3* is shown in Figure 11-6.

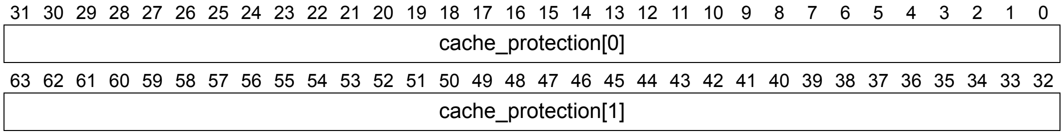
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_CACHE_PROT_INFO returns information about the data and tag protection method for the specified cache. The three returns compose a six-element array of 32-bit protection information structures.

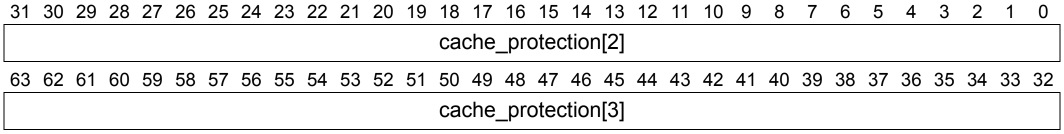
The *config_info_1* return value has the following structure:

Figure 11-4. *config_info_1* Return Value



The *config_info_2* return value has the following structure:

Figure 11-5. *config_info_2* Return Value



The *config_info_3* return value has the following structure:

Figure 11-6. *config_info_3* Return Value

31 … 0
cache_protection[4]

63 … 32
cache_protection[5]

Each *cache_protection* element has the following structure:

Figure 11-7. *cache_protection* Fields

31–30	29–26	25–20	19–14	13–8	7–0
t_d	method	prot_bits	tagprot_msb	tagprot_lsb	data_bits

- *data_bits* – Unsigned 8-bit integer denoting the number of data bits that each unit of protection covers. For example, if the cache hardware generates 8 bits of ECC per 64 bits of data, *data_bits* would be 64. This field is only valid if *t_d* is 0, 2, or 3.
- *tagprot_lsb* – Unsigned 6-bit integer denoting the least-significant tag address bit that this protection method covers. This field is only valid if *t_d* is 1, 2, or 3.
- *tagprot_msb* – Unsigned 6-bit integer denoting the most-significant tag address bit that this protection method covers. This field is only valid if *t_d* is 1, 2, or 3.
- *prot_bits* – Unsigned 6-bit integer denoting the number of protection bits generated for the field specified by the *t_d* element.
- *method* – Unsigned 4-bit integer denoting the protection method, where:

Table 11-72. *method* Values

Value	Description
0	no ECC or parity protection
1	odd parity protection
2	even parity protection
3	ECC protection

All other values of *method* are reserved.

- *t_d* – 2-bit field denoting whether this protection method applies to the tag, the data, or both. If over both, the tag and data unit could be concatenated with the tag either to the left (more significant) or to the right (less significant) than a unit of data. For the values of 2 and 3, the difference of *tagprot_msb* and *tagprot_lsb* indicates the number of tag bits that are protected with the data bits. The data bits are described by the *data_bits* field below. This field is encoded as follows:

Table 11-73. *t_d* Values

Value	Description
0	Data protection
1	Tag protection
2	Tag+data protection (tag is more significant)
3	Data+tag protection (data is more significant)

When obtaining cache information via this call, information for the data cache should be obtained first, then if the *u* bit of the *config_info_1* parameter is not set, obtain the information for the instruction cache.

PAL_CACHE_READ – Read Values from the Processor Cache (259)

Purpose: Reads the data and tag of a processor-controlled cache line for diagnostic testing.

Calling Conv: Stacked Registers

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_READ within the list of PAL procedures.
line_id	8-byte formatted value describing where in the cache to read the data.
address	64-bit 8-byte aligned physical address from which to read the data. The address must be an implemented physical address on the processor model with bit 63 set to zero.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_READ procedure.
data	Right-justified value returned from the cache line.
length	The number of bits returned in *data*.
mesi	The status of the cache line.

Status:

Status Value	Description
1	The word at *address* was found in the cache, but the line was invalid.
0	Call completed without error.
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error.
-5	The word at *address* was not found in the cache.
-7	The operation requested is not supported for this *cache_type* and *level.*

Description: A value is read from the specified cache line, if present. This procedure allows reading cache data, tag, protection, or status bits.

The *line_id* argument is an 8-byte quantity in the following format:

Figure 11-8. Layout of *line_id* Return Value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
part								way								level								cache_type							

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved																															

- *cache_type* – Unsigned 8-bit integer denoting whether to read from instruction (1) or data/unified (2) cache. All other values are reserved.
- *level* – Unsigned 8-bit integer specifying which cache within the cache hierarchy to read. This value must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY.
- *way* – Unsigned 8-bit integer denoting within which cache way to read. If the cache is direct-mapped this argument is ignored.
- *part* – Unsigned 8-bit integer denoting which portion of the specified cache line to read:

Table 11-74. *part* Input Values

Value	Description
0	data
1	tag
2	data protection bits
3	tag protection bits
4	combined protection bits for data and tags[a]

[a] Note that for this *part* no data is returned. Only protection bits are returned.

All other values of *part* are reserved.

The *data* return value contains the value read from the cache. Its contents are interpreted according to the *line_id.part* field as follows:

Table 11-75. *part* Input Values and corresponding *data* Return Values

Part	Data
0	64-bit data.
1	right-justified tag of the specified line.
2	right-justified protection bits corresponding to the 64 bits of data at *address*. If the cache uses less than 64-bits of data to generate protection, *data* will contain more than one value. For example if a cache generates parity for every 8-bits of data, this return value would contain 8 parity values. The PAL_CACHE_PROT_INFO call returns information on how a cache generates protection information in order to decode this return value. If a cache uses greater than 64-bits of data to generate protection, *data* will contain the value to use for the portion of the cache line indicated by *address*.
3	right-justified protection bits for the cache line tag.
4	right-justified protection bits for the cache line tag and 64 bits of data at *address*.

The *length* return value contains the number of valid bits returned in *data*.

The *mesi* return value contains the status bits of the cache line. Values are defined as follows:

Table 11-76. *mesi* Return Values

Value	Description
0	invalid
1	shared
2	exclusive
3	modified

All other values of *mesi* are reserved.

To guarantee correct behavior for this procedure, it is required that there shall be no RSE activity that may cause cache side effects.

PAL_CACHE_SHARED_INFO – Get Information on Caches Shared by Logical Processors (43)

Purpose: Returns information on caches shared between logical processors.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_SHARED_INFO within the list of PAL procedures.
cache_level	Unsigned 64-bit integer specifying the level in the cache hierarchy for which information is requested. This value must be between 0 and one less than the value returned in the cache_levels return value from PAL_CACHE_SUMMARY.
cache_type	Unsigned 64-bit integer with a value of 1 for instruction cache and 2 for data or unified cache. All other values are reserved.
proc_number	Unsigned 64-bit integer that specifies for which logical processor information is being requested. This input argument must be zero for the first call to this procedure and can be a maximum value of one less than the number of logical processors sharing this cache, which is returned by the *num_shared* return value.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_SHARED_INFO procedure.
num_shared	Unsigned integer that returns the number of logical processors that share the processor cache level and type, for which information was requested.
proc_n_cache_info1	The format of *proc_n_cache_info1* is shown in Figure 11-9.
proc_n_cache_info2	The format of *proc_n_cache_info2* is shown in Figure 11-10.

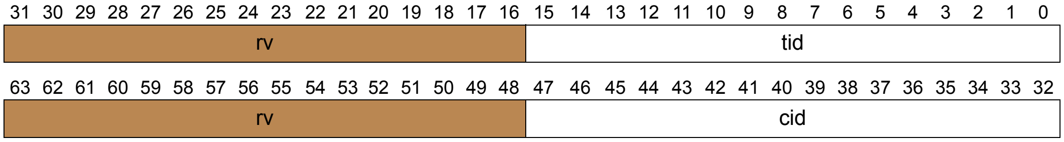
Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure will return information about how the processor caches are shared among logical processors (See "PAL_LOGICAL_TO_PHYSICAL – Get Information on Logical to Physical Processor Mappings (42)" on page 2:404 for a definition of a logical processor). If the caller is only interested in how many logical processors are sharing a particular cache level, this procedure will only need to be called once. If the caller is interested in identifying which logical processors are sharing the processor caches, this procedure will need to be called a number of times equal to the value returned in *num_shared* to gather identification information for all the logical processors sharing the particular cache for which information is being requested.

Identification information about the logical processors sharing the cache is in the return values *proc_n_cache_info1* and *proc_n_cache_info2*. The format of these return values is shown in Figure 11-9 and Figure 11-10.

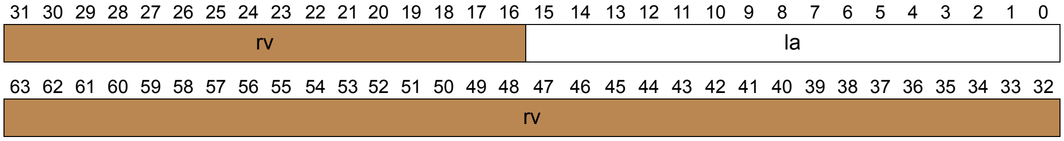
Figure 11-9. Layout of *proc_n_cache_info1* Return Value



- *tid* – Thread id: The thread identifier of the logical processor for which information is being returned. This value will be unique on a per core basis.
- *rv* – Reserved
- *cid* – Core id: The core identifier of the logical processor for which information is being returned. This value will be unique on a per physical processor package basis.
- *rv* – Reserved

There is no guarantee that the core id's and thread id's will be contiguous on a given physical processor package.

Figure 11-10. Layout of *proc_n_cache_info2* Return Value



- *la* – Logical address: geographical address of the logical processor for which information is being returned. This is the same value that is returned by the PAL_FIXED_ADDR procedure when it is called on the logical processor.
- *rv* – Reserved

This procedure must be supported on all implementations that contain more than one logical processor on a physical processor package and returns an unimplemented procedure error code otherwise.

PAL_CACHE_SUMMARY – Get Cache Hierarchy Summary (4)

Purpose: Returns summary information about the hierarchy of caches controlled by the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_SUMMARY within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_SUMMARY procedure.
cache_levels	Unsigned 64-bit integer denoting the number of levels of cache implemented by the processor. Strictly, this is the number of levels for which the cache controller is integrated into the processor (the cache SRAMs may be external to the processor).
unique_caches	Unsigned 64-bit integer denoting the number of unique caches implemented by the processor. This has a maximum of 2**cache_levels,* but may be less if any of the levels in the cache hierarchy are unified caches or do not have both instruction and data caches.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Software is expected to call PAL_CACHE_SUMMARY before calling PAL_CACHE_INFO to determine the number of times PAL_CACHE_INFO should be called and the amount of storage that must be allocated to hold all of the information returned by PAL_CACHE_INFO.

PAL_CACHE_WRITE – Write Values into the Processor Cache (260)

Purpose: Writes the data and tag of a processor-controlled cache line for diagnostic testing.

Calling Conv: Stacked Registers

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_CACHE_WRITE within the list of PAL procedures.
line_id	8-byte formatted value describing where in the cache to write the data.
address	64-bit 8-byte aligned physical address at which the data should be written. The address must be an implemented physical address on the processor model with bit 63 set to 0.
data	unsigned 64-bit integer value to write into the specified *part* of the cache.

Returns:

Return Value	Description
status	Return status of the PAL_CACHE_WRITE procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error.
-7	The operation requested is not supported for this *cache_type* and *level.*

Description: The value of *data* is written into the specified level, way, and part of the cache. This procedure allows writing cache data, tag, protection, or status bits.

This procedure may also be used to seed errors into a cache line. It calculates the protection bits based on the value of *data*, then inverts a specified bit field before writing *data* to the cache. Bit field inversion is only used for writes to the cache data or tag.

If seeding an error into the instruction cache or seeding an unrecoverable error, then return back to the caller may not be possible.

This procedure call cannot be used where coherency is required.

The *line_id* argument is an 8-byte quantity in the following format:

Figure 11-11. Layout of *line_id* Return Value

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
part	way	level	cache_type

63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32
trigger	length	start	mesi

- *cache_type* – Unsigned 8-bit integer denoting whether to write to instruction (1) or data/unified (2) cache. All other values are reserved.
- *level* – Unsigned 8-bit integer specifying which cache within the cache hierarchy to write *data.* This value must be in the range from 0 up to one less than the *cache_levels* return value from PAL_CACHE_SUMMARY.
- *way* – Unsigned 8-bit integer denoting within which cache way to write *data*. If the cache is direct-mapped this argument is ignored.
- *part* – Unsigned 8-bit integer denoting where to write *data* into the cache:

Table 11-77. *part* Input Values

Value	Description
0	data
1	tag
2	data protection
3	tag protection
4	combined data and tag protection

All other values of *part* are reserved.

- *mesi* – Unsigned 8-bit integer denoting whether the line should be written as clean or dirty, shared or exclusive. Though there may be multiple calls to PAL_CACHE_WRITE to the same cache line, the last call's *mesi* will be in effect. Values are defined as follows:

Table 11-78. *mesi* Return Values

Value	Description
0	invalid
1	shared
2	exclusive
3	modified

All other values of *mesi* are reserved.

- *start* – Unsigned 8-bit integer denoting the least-significant bit of the field in *data* to invert. If *length* is 0 or *part* is not 0 or 1, this field is ignored.
- *length* – Unsigned 8-bit integer denoting the number of bits to invert. If *length* is 0, no bits are inverted and *start* is ignored. If *part* is not 0 or 1, this field is ignored.
- *trigger* – Unsigned 8-bit integer denoting whether to trigger the error while in procedure. If *trigger* is 0, the procedure writes *data* and returns. If *trigger* is 1 and *cache_type* is data/unified, the procedure writes *data* and executes a 64-bit load from *address* before returning. If *trigger* is 1 and *cache_type* is set to instruction, the procedure writes *data* and branches to the *address*. All other values are reserved.

The *data* argument contains the value to write into the cache. Its contents are interpreted based on the *part* field as follows:

Table 11-79. Interpretation of *data* Input Field

Part	Data
0	64-bit data to write to the specified line (with optional bit field inversion).
1	right-justified tag to write into the specified line (with optional bit field inversion).
2	right-justified protection bits corresponding to the 64 bits of data at *address*. If the cache uses less than 64-bits of data to generate protection, *data* will contain more than one value. For example if a cache generates parity for every 8-bits of data, this return value would contain 8 parity values. The PAL_CACHE_PROT_INFO call returns information on how a cache generates protection information in order to decode this return value. If a cache uses greater than 64-bits of data to generate protection, *data* will contain the value to use for the portion of the cache line indicated by *address*.
3	right-justified protection bits for the cache line tag.
4	right-justified protection bits for the cache line tag and 64 bits of data at *address*.

To guarantee correct behavior for this procedure, it is required that there shall be no RSE activity that may cause cache side effects.

PAL_COPY_INFO – Return Parameters to Copy PAL Code to Memory (30)

Purpose: Returns the parameters needed to copy relocatable PAL code from the firmware address space to memory.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_COPY_INFO within the list of PAL procedures.
copy_type	Unsigned integer denoting type of procedures for which copy information is requested.
Reserved	0
mca_proc_state_info	Unsigned integer denoting the number of bytes that SAL needs for the min-state save area for each processor.

Returns:

Return Value	Description
status	Return status of the PAL_COPY_INFO procedure.
buffer_size	Unsigned integer denoting the number of bytes of PAL information that must be copied to main memory.
buffer_align	Unsigned integer denoting the starting alignment of the data to be copied.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to obtain the information needed to relocate runtime PAL procedures and PAL PMI code from the firmware address space to memory. The information returned in this call is used by SAL to allocate a memory region on the required alignment, and call PAL_COPY_PAL to copy the relocatable PAL code.

The *copy_type* input argument indicates which type of procedure for which copying information is requested. A value of 0 denotes procedures required for SAL, PMI, and Itanium architecture-based operating systems. All other values are reserved. If the copy_type is 0, then SAL shall call PAL_COPY_PAL call subsequently to copy the PAL procedures and PAL PMI code to the allocated memory region.

The *buffer_align* return value must be a power of two between 4 KB and 1 MB.

PAL_COPY_PAL – Copy PAL Code to Memory (256)

Purpose: Copy relocatable PAL code from the firmware address space to memory.

Calling Conv: Stacked Registers

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_COPY_PAL within the list of PAL procedures.
target_addr	Physical address of a memory buffer to copy relocatable PAL procedures and PAL PMI code.
alloc_size	Unsigned integer denoting the size of the buffer passed by SAL for the copy operation.
copy_option	Unsigned integer indicating whether relocatable PAL code and PAL PMI code should be copied from firmware address space to main memory.

Returns:

Return Value	Description
status	Return status of the PAL_COPY_PAL procedure.
proc_offset	Unsigned integer denoting the offset of PAL_PROC in the relocatable segment copied.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to relocate runtime PAL procedures and PAL PMI code from the firmware address space to main memory. A value of 0 for the *copy_option* indicates that the relocation should be performed; a value of 1 indicates that the relocation should not be performed. This procedure also updates the PALE_PMI entrypoint in hardware. All other values are reserved.

PAL_COPY_INFO should be called first to determine the size and alignment requirements of the memory buffer to which the PAL code will be copied. Bit 63 of *target_addr* must be set consistently with the cacheability attribute of the memory buffer being copied to. It is PAL's responsibility to ensure that the firmware address space contents that are being copied from, are not in any processor caches. It is the caller’s responsibility to ensure that the contents of the memory buffer copied to, are flushed out of the internal processor's data caches if *target_addr* has a cacheable memory attribute.

If a PAL procedure makes calls to internal PAL functions that execute only out of the firmware address space, that portion of code will continue to execute out of the firmware address space, even though the main procedure has been copied to RAM. This is true only for some PAL procedures that can be called only in physical mode.

PAL_COPY_PAL call is mandatory as part of the system boot process. Higher level firmware should guarantee that PAL_COPY_PAL is called on all processors before OS launch. This is to guarantee that full processor functionality is available. This procedure can be called more than once.

PAL_DEBUG_INFO – Get Debug Registers Information (11)

Purpose: Returns the number of instruction and data debug register pairs.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_DEBUG_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_DEBUG_INFO procedure.
i_regs	Unsigned 64-bit integer denoting the number of pairs of instruction debug registers implemented by the processor.
d_regs	Unsigned 64-bit integer denoting the number of pairs of data debug registers implemented by the processor.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call returns the number of pairs of registers. Even numbered registers contain breakpoint addresses and odd numbered registers contain breakpoint mask conditions. For example if *i_regs* is 4, there are 8 instruction debug registers of which 4 are breakpoint address registers ($IBR_{0,2,4,6}$) and 4 are breakpoint mask registers ($IBR_{1,3,5,7}$). The minimum value for both *i_regs* and *d_regs* is 4.

On some implementations, a hardware debugger may use two or more debug register pairs for its own use. When a hardware debugger is attached, PAL_DEBUG_INFO may return a value for *i_regs* and/or *d_regs* less than the implemented number of debug registers. When a hardware debugger is attached, PAL_DEBUG_INFO may return a minimum value of 2 for *d_regs* and a minimum of 2 for *i_regs*.

PAL_FIXED_ADDR – Get Fixed Geographical Address of Processor (12)

Purpose: Returns a unique geographical address of this processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_FIXED_ADDR call within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_FIXED_ADDR procedure.
address	Fixed geographical address of this processor.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The *address* return value will contain a unique unsigned integer denoting the position of this processor on its system interconnect. This is an arbitrary number which is expected to have geographical significance and is unique for the system interconnect to which the processor is connected. If the processor is connected to multiple system interconnects, the *address* return value must be unique among all such interconnects. The maximum size of the *address* returned corresponds to the size of the fields (id and eid) in the LID register (CR64).

PAL_FREQ_BASE – Get Processor Base Frequency (13)

Purpose: Returns the frequency of the output clock for use by the platform is generated by the processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_FREQ_BASE within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_FREQ_BASE procedure.
base_freq	Base frequency of the platform if generated by the processor chip.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Can not complete call without error

Description: If the processor outputs a clock for use by the platform, the *base_freq* return parameter will be the frequency of this output clock in ticks per second. If the processor does not generate an output clock for use by the platform, this procedure will return with a status of -1.

PAL_FREQ_RATIOS – Get Processor Frequency Ratios (14)

Purpose: Returns the ratios of the processor frequency, bus frequency, and interval timer to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_FREQ_RATIOS within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

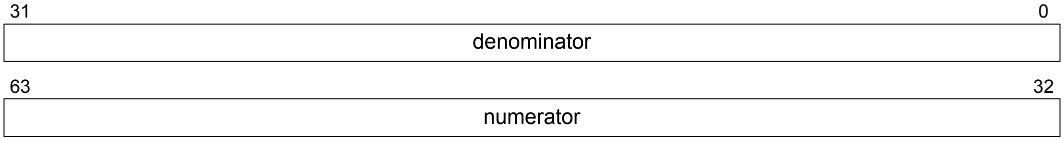
Return Value	Description
status	Return status of the PAL_FREQ_RATIOS procedure.
proc_ratio	Ratio of the processor frequency to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.
bus_ratio	Ratio of the bus frequency to the input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.
itc_ratio	Ratio of the interval timer counter rate to input clock of the processor, if the platform clock is generated externally or to the output clock to the platform, if the platform clock is generated by the processor.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Can not complete call without error

Description: Each of the ratios returned is an unsigned 64-bit value, where the upper unsigned 32 bits contain the numerator and the lower unsigned 32 bits contain the denominator of the ratio, as depicted in Figure 11-12. Each ratio is given by dividing the numerator by the denominator.

Figure 11-12. Return values



- denominator – Unsigned 32-bit integer
- numerator – Unsigned 32-bit integer

PAL_GET_HW_POLICY – Retrieve Current Hardware Resource Sharing Policy (48)

Purpose: Returns the current hardware resource sharing policy of the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_GET_HW_POLICY within the list of PAL procedures.
proc_num	Unsigned 64-bit integer that specifies for which logical processor information is being requested. This input argument must be zero for the first call to this procedure and can be a maximum value of one less than the number of logical processors impacted by the hardware resource sharing policy, which is returned by the *num_impacted* return value.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_GET_HW_POLICY procedure.
cur_policy	Unsigned 64-bit integer representing the current hardware resource sharing policy.
num_impacted	Unsigned 64-bit integer that returns the number of logical processors impacted by the *policy* input argument.
la	Unsigned 64-bit integer containing the logical address of one of the logical processors impacted by policy modification.

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure is used to return information on the current hardware resource sharing policy. This procedure can also be used to identify which logical processors (see "PAL_LOGICAL_TO_PHYSICAL – Get Information on Logical to Physical Processor Mappings (42)" on page 2:404 for a definition of a logical processor) are impacted by the various hardware sharing policies supported on the processor.

The procedure returns information about the current hardware sharing policy, the total number of logical processors impacted by hardware sharing policies and the logical address of one of the processors impacted by the hardware sharing policy.

The definition of the hardware sharing policies that can be returned in the *cur_policy* value are defined in Table 11-80.

Table 11-80. Hardware policies returned in *cur_policy*

Value	Name	Description
0	Performance	The processor has its hardware resources configured to achieve maximum performance across all logical processors that share hardware with the logical processor the procedure was made on.
1	Fairness	The processor has its hardware resources configured to approximately achieve equal sharing of competing hardware resources among all the logical processors that share hardware with the logical processor the procedure was made on.
2	High-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a greater share of the competing hardware resources.
3	Exclusive High-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a greater share of the competing hardware resources. See "PAL_SET_HW_POLICY – Set Current Hardware Resource Sharing Policy (49)" on page 2:456 for differences between high-priority and exclusive high priority.
4	Low-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a smaller share of the competing hardware resources. This occurs when a competing logical processor has itself set as high priority or exclusive high priority.
All Other Values		Reserved

The return value *num_impacted* specifies the number of logical processors impacted by the hardware sharing policy. The return value *la* returns the logical address of one of the logical processors impacted by the hardware sharing policy. The return value *la* is the same value and format of that is returned by the PAL_FIXED_ADDR procedure, see "PAL_FIXED_ADDR – Get Fixed Geographical Address of Processor (12)" on page 2:391 for details.

If the caller is interested in identifying all the logical processors impacted by the hardware sharing policy, this procedure will need to be called a number of times equal to the value returned in *num_impacted* return value. For each subsequent call it needs to increment the 'proc_num' input argument.

The logical processor this procedure is made on can only return information about how the hardware sharing policy impacts logical processors it is sharing hardware resources with. For example a physical processor package may contain two multi-threaded cores. On this example implementation the hardware sharing policy only impacts the two threads on the core and this procedure would only return the two *la*'s of the threads on that core, but would not return the *la*'s of the threads on the other core. When this procedure was made on the other core, then that procedure call would return the *la*'s of the two threads on that core.

This procedure is only supported on processors that have multiple logical processors sharing hardware resources that can be configured. On all other processor implementations, this procedure will return the Unimplemented procedure return status.

PAL_GET_PSTATE – Return Information on the Performance Index of the Processor (262)

Purpose: Returns the performance index of the processor.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_GET_PSTATE within the list of PAL procedures.
type	Type of *performance_index* value to be returned by this procedure.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_GET_PSTATE procedure.
performance_index	Unsigned integer denoting the processor performance for the time duration since the last PAL_GET_PSTATE procedure call was made. The value returned is relative to the performance index of the highest available P-state.
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error, but accuracy of performance index has been impacted by a thermal throttling event, or a hardware-initiated event.
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure returns a performance index of the processor, and is relative to the highest available P-state, P0. A value of 100 represents the minimum processor performance in the P0 state. For processors that support variable P-state performance, it is possible for a processor to report a number greater than 100, representing that the processor is running at a performance level greater than the minimum P0 performance. The PAL procedure "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 indicates whether the processor supports variable P-state performance.

The *type* argument allows the caller to select the *performance_index* value that will be returned. See Table 11-81 below for details.

Table 11-81. PAL_GET_PSTATE *type* Argument

type	Description
0	The *performance_index* returned will correspond to the target P-state requested by software. • For SCDD (software-coordinated dependency domain) logical processors, this is the P-state requested by the most recent PAL_SET_PSTATE procedure call made by any logical processor in the domain. • For HCDD (hardware-coordinated dependency domain) or HIDD (hardware-independent dependency domain) logical processors, this is simply the P-state requested by the most recent PAL_SET_PSTATE procedure call on this logical processor. The value returned is not affected by platform power-caps.
1	The *performance_index* is a weighted-average value of the different P-states that the processor was operating in for the time duration between the current PAL_GET_PSTATE procedure call, and the previous invocation of PAL_GET_PSTATE with type=1. This allows the caller to establish a new starting point for subsequent computation of the weighted-average *performance_index*. See Section 11.6.1, “Power/Performance States (P-states)” on page 2:315 for more details on how the weighted average value is derived.
2	The *performance_index* is a weighted-average value of the different P-states that the processor was operating in for the time duration between the current PAL_GET_PSTATE procedure call, and the previous invocation of PAL_GET_PSTATE with type=1. This allows the caller to sample the current value of the *performance_index*, without affecting the starting point used for computing the weighted-average performance_index.
3	The *performance_index* returned will correspond to the current instantaneous P-state of the dependency domain containing the logical processor, at the time of the procedure call. The value returned is not affected by platform power-caps. When variable P-states performance is supported, the *performance_index* may be higher than the P-state requested. Please see Section 11.6.1.4, “Variable P-state Performance” on page 2:322 for more information about variable P-state performance.
All Other Values	Reserved

For SCDD logical processors, or HIDD logical processors that do not support platform power-caps, note that the *performance_index* returned for *type*=0 and *type*=3 will have identical values. This is because the most recent PAL_SET_PSTATE procedure call that returned a status of 0 will always succeed in transitioning to the requested performance state for these coordination domains (see PAL_SET_PSTATE procedure description for additional details).

For SCDD logical processors, the PAL_GET_PSTATE procedure should always be called with *type* argument value of 0 or 3. On such processors, calling PAL_GET_PSTATE with *type* argument value of 1 or 2 is undefined.

For HIDD logical processors, the *type* argument values of 1 and 2 are supported, since such processors can also support platform power-caps, which affect the weighted-average performance index.

If there was a thermal-throttling or hardware-initiated event (other than a platform power-cap) which affected the processor power/performance for the current time period, and the accuracy of the *performance_index* value has been impacted by the event, then the procedure will return with *status*=1. The *performance_index* returned in this case will still have a value that falls within the range of possible *performance_index* values for this processor implementation (i.e., 0 up to the highest variable p-state *performance_index* value).

The procedure, when called with *type*=1 or *type*=2, returns a fixed *performance_index* value of 100 until the procedure has been called with *type*=1 to reset computation of the weighted-average *performance_index*. For subsequent invocations with *type*=1 or *type*=2, the procedure will return the *performance_index* value corresponding to the processor performance in the time duration between the previous call to PAL_GET_PSTATE with *type*=1 and the current call.

If the processor had transitioned to a HALT state (see Section 11.6.1, “Power/Performance States (P-states)” on page 2:315) in between successive invocations to the PAL_GET_PSTATE procedure, the performance index computation returned will not take into account the performance of the processor during the time spent in HALT state (see Section 11.6.1.5, “Interaction of P-states with HALT State” on page 2:323 for details).

PAL_HALT – Halt Processor (28)

Purpose: Causes the processor to enter the HALT state, or one of the implementation-dependent low-power states.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_HALT within the list of PAL procedures.
halt_state	Unsigned 64-bit integer denoting low power state requested.
io_detail_ptr	8-byte aligned physical address pointer to information on the type of I/O (load/store) requested.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT procedure.
load_return	Value returned if a load instruction is requested in the *io_detail_ptr*
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This call places the processor in a low power state designated by *halt_state.* This procedure can optionally let the platform know it is about to enter the low power state via an I/O transaction.

halt_state is an unsigned 64-bit integer denoting the low power state requested. The value passed must be a valid halt state in the range from 1 to 7, for which information is returned by PAL_HALT_INFO. All other values are reserved.

The processor informs the platform that it has entered the requested low-power state in an implementation-specific manner.

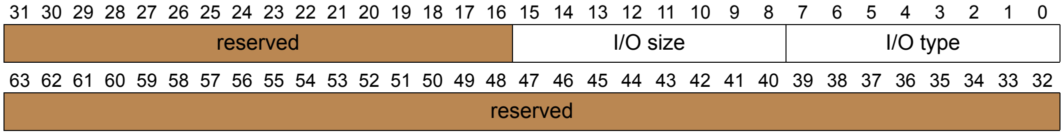
The layout of the information pointed to by the *io_detail_ptr* is shown Table 11-82.

Table 11-82. I/O Detail Pointer Description

Offset	Description
0x0	I/O size and type information
0x8	Address for I/O
0x10	Data value to store

- I/O size and type information has the format shown in Figure 11-13.

Figure 11-13. I/O Size and Type Information Layout



- *I/O type* is an unsigned 8-bit integer denoting the type of I/O transaction to complete.

Table 11-83. I/O Type Definition

Value	Description
0	No transaction
1	Perform a load
2	Perform a store

All other values for *I/O type* are reserved.

- *I/O size* is an unsigned 8-bit integer denoting the size of the I/O transaction to complete.

Table 11-84. I/O Size Definition

Value	Description
0	No transaction
1	1 byte size
2	2 byte size
4	4 byte size
8	8 byte size

All other values for *I/O size* are reserved.

- Address for the I/O transaction is a physical pointer for the load or store. The address passed should be aligned according to the size of the I/O transaction requested. The most significant bit (63) of the physical address should be set according to the cacheability attribute wanted for the I/O transaction.
- The data value to store is the value that will be stored out if the *io_type* is 2. If *io_type* is not equal to a 2, then this value is a don't care.

If an I/O transaction is requested by the caller, the processor will wait until this transaction has been received by the platform before entering the low power state.

On receipt of a PMI, machine check, INIT, reset, or unmasked external interrupt (including NMI), PAL transitions the processor to the normal state. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields in the TPR control register. PAL sets the value in the *load_return* return parameter if the *io_type* is 1, otherwise this value is set to zero.

If the processor transitions to normal state via an unmasked external interrupt, execution resumes to the caller.

If the processor transitions to normal state via a PMI, execution resumes to the caller if PMIs are masked, otherwise execution will resume to the PMI handler.

If the processor transitions to the normal state via a machine check or INIT, execution resumes to the caller if machine checks and INITs are masked, otherwise execution will resume to the corresponding handler.

If the processor transitions to the normal state via a reset event, the processor will reset itself and start execution at the PAL reset address.

For more information on power management, please refer to Section 11.6, "Power Management" on page 2:313.

PAL_HALT_INFO – Get Halt State Information for Power Management (257)

Purpose: Returns information about the processor’s power management capabilities.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_HALT_INFO within the list of PAL procedures.
power_buffer	64-bit pointer to a 64-byte buffer aligned on an 8-byte boundary.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT_INFO procedure.
Reserved	0
Reserved	0
Reserved	0

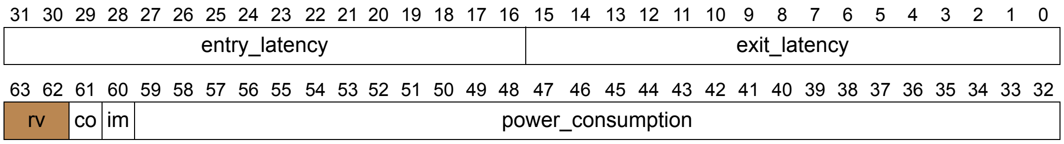
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The power information requested is returned in the data buffer referenced by *power_buffer*. Power information is returned about the 8 power states. The low power states are LIGHT_HALT, HALT, plus 6 other low power states. The LIGHT_HALT state is index 0 in the buffer, and the HALT state is index 1. All 8 low power states need not be implemented

The information returned is in the format of Figure 11-14. The information about the HALT states will be in ascending order of the index values.

Figure 11-14. Layout of *power_buffer* Return Value



- *exit latency* – 16-bit unsigned integer denoting the minimum number of processor cycles to transition to the NORMAL state.
- *entry_latency* – 16-bit unsigned integer denoting the minimum number of processor cycles to transition from the NORMAL state.
- *power_consumption* – 28-bit unsigned integer denoting the typical power consumption of the state, measured in milliwatts.
- *im* – 1-bit field denoting whether this low power state is implemented or not. A value of 1 indicates that the low power state is implemented, a value of 0 indicates that it is not implemented. If this value is 0 then all other fields are invalid.
- *co* – 1-bit field denoting if the low power state maintains cache and TLB coherency. A value of 1 indicates that the low power state keeps the caches and TLBs coherent, a value of 0 indicates that it does not.

The latency numbers given are the minimum number of processor cycles that will be required to transition the states. The maximum or average cannot be determined by PAL due to its dependency on outstanding bus transactions.

For more information on power management, please refer to Section 11.6, “Power Management” on page 2:313.

PAL_HALT_LIGHT – Cause Processor to Enter Coherent Halt State (29)

Purpose: Causes the processor to enter the LIGHT HALT state, where prefetching and execution are suspended, but cache and TLB coherency is maintained.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_HALT_LIGHT within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_HALT_LIGHT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call places the processor in the LIGHT HALT state in an implementation-dependent fashion where cache and TLB coherency is maintained, but power consumption is minimized.

The processor acknowledges to the platform that it has entered the LIGHT HALT low-power state in an implementation-specific manner.

On receipt of a PMI, machine check, INIT, reset, or unmasked external interrupt (including NMI), PAL transitions the processor to the normal state. An unmasked external interrupt is defined to be an interrupt that is permitted to interrupt the processor based on the current setting of the TPR.mic and TPR.mmi fields in the TPR control register.

If the processor transitions to normal state via an unmasked external interrupt, execution resumes to the caller.

If the processor transitions to normal state via a PMI, execution resumes to the caller if PMIs are masked, otherwise execution will resume to the PMI handler.

If the processor transitions to the normal state via a machine check or INIT, execution resumes to the caller if machine checks and INITs are masked, otherwise execution will resume to the corresponding handler.

If the processor transitions to the normal state via a reset event, the processor will reset itself and start execution at the PAL reset address.

For more information on power management, please refer to Section 11.6, “Power Management” on page 2:313.

PAL_LOGICAL_TO_PHYSICAL – Get Information on Logical to Physical Processor Mappings (42)

Purpose: Returns information on the logical to physical processor mapping.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_LOGICAL_TO_PHYSICAL within the list of PAL procedures.
proc_number	Signed 64-bit integer that specifies for which logical processor information is being requested. When this input argument is -1, information is returned about the logical processor on which the procedure call is made. This input argument must be in the range of -1 up to one less than the number of logical processors returned by *num_log* in the *log_overview* return value.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_LOGICAL_TO_PHYSICAL procedure.
log_overview	The format of *log_overview* is shown in Figure 11-15.
proc_n_log_info1	The format of *proc_n_log_info1* is shown in Figure 11-16.
proc_n_log_info2	The format of *proc_n_log_info2* is shown in Figure 11-17.

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure will return information about the logical processors contained on the physical processor package that the procedure call is made on. A physical processor package can contain one or more logical processors, organized into threads and cores. A logical processor is a compute-capability-centric view of the CPU that allows the physical processor package to execute from more than one instruction stream. A physical processor package that can execute from *n* instruction streams has *n* logical processors. Threads are logical processors that share core pipeline execution resources. Cores are defined as a collection of hardware that implements the main execution pipeline of the processor. Multiple cores on a physical processor package do not share core pipeline resources but may share caches and bus interfaces. A core may support multiple threads of execution.

The *log_overview* return value provides an overview of the logical processors on the physical processor package this procedure call was made on. The format of the *log_overview* return argument is shown in Figure 11-15.

Figure 11-15. Layout of *log_overview* Return Value

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
rv	tpc	num_log

63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32
rv	ppid	rv	cpp

- *num_log* – Total number of logical processors on this physical processor package that are enabled.
- *tpc* – Threads per core. Number of threads per core.
- *rv* – Reserved
- *cpp* – Cores per processor. Total number of cores on this physical processor package.
- *rv* – Reserved
- *ppid* – Physical processor package ID. Physical processor package identifier which was assigned at reset by the platform or bus controller. This value may or may not be unique across the entire platform since it depends on the platform vendor's policy.
- *rv* – Reserved

It is not ensured that *num_log* will always be equal to *cpp* multiplied by *tpc*. This is possible if some logical processors are disabled through implementation specific means.

The caller uses the value returned in *num_log* to gather additional information about the other logical processors on the same physical processor package. This procedure will need to be called multiple times (equal to the number of logical processors returned in *num_log*) to gather all additional information about the logical processors on the physical processor package this procedure call was made on. This procedure may be called from any logical processor on the physical processor package to gather information about all the logical processors. It may also be called to get information about the logical processor on which the procedure is running. Information about the logical processors is in the return values *proc_n_log_info1* and *proc_n_log_info2*. The format of these return values is shown in Figure 11-16 and Figure 11-17.

Figure 11-16. Layout of *proc_n_log_info1* Return Value

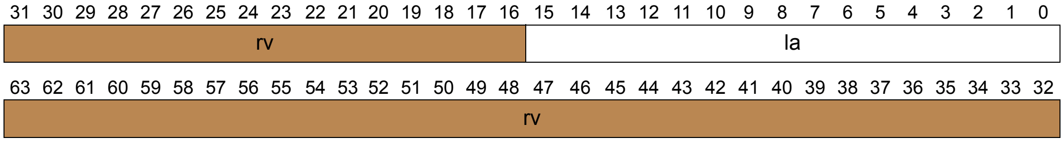
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
rv	tid

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
rv	cid

- *tid* – Thread id: The thread identifier of the logical processor for which information is being returned. This value will be unique on a per core basis.
- *rv* – Reserved
- *cid* – Core id: The core identifier of the logical processor for which information is being returned. This value will be unique on a per physical processor package basis.
- *rv* – Reserved

There is no guarantee that the core id's and thread id's will be contiguous on a given physical processor package.

Figure 11-17. Layout of *proc_n_log_info2* Return Value



- *la* – Logical address: geographical address of the logical processor for which information is being returned. This is the same value that is returned by the PAL_FIXED_ADDR procedure when it is called on the logical processor.
- *rv* – Reserved

This procedure must be supported on all implementations that contain more than one logical processor on a physical processor package and returns an unimplemented procedure error code otherwise.

PAL_MC_CLEAR_LOG – Clear Processor Error Logging Registers (21)

Purpose: Clears all processor error logging registers and resets the indicator that allows the error logging registers to be written. This procedure also checks the pending machine check bit and pending INIT bit and reports their states.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_CLEAR_LOG within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_CLEAR_LOG procedure.
pending	64-bit vector denoting whether an event is pending.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to clear processor error logging registers after all error information has been obtained. This procedures re-enables the logging registers in the case of a subsequent error. It clears any information that would be returned by either the PAL_MC_ERROR_INFO or PAL_MC_DYNAMIC_STATE procedures.

This procedure does not clear any pending machine checks. The *pending* return parameter returns a value of 0 if no subsequent event is pending, a 1 in bit position 0, if a machine check is pending, and/or a 1 in bit position 1 if an INIT is pending. All other values are reserved.

Figure 11-18. *Pending* Return Parameter

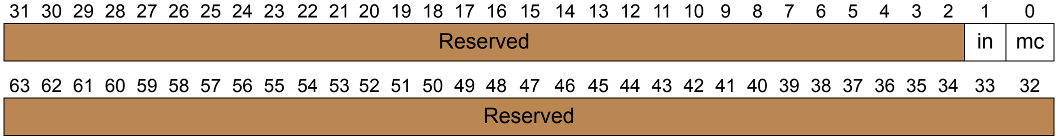


Table 11-85. Pending Return Parameter Fields

Field	Description
mc	Pending machine check
in	Pending initialization event

PAL_MC_DRAIN – Complete Outstanding Transactions (22)

Purpose: Ensures that all outstanding transactions in a processor are completed or that any MCA due to these outstanding transactions is taken.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_DRAIN within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_DRAIN procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This call causes all outstanding transactions in the processor to be completed. For example:

- Flushes (`fc`) invalidate the cache, lines that have been modified are written back (issued to the fabric) to memory before invalidation.
- Instruction cache coherence flushes (`fc.i`) invalidate lines and/or write them back to main memory, if this is required to make the instruction caches coherent with the data caches.
- Loads get their data returned.
- Stores either update the cache or issue transactions to the system fabric.
- Prefetches are either completed or cancelled,

As a result of completing these outstanding transactions Machine Check Aborts (MCAs) may be taken. This call is typically issued by code that needs to guarantee that no MCAs due to outstanding transactions will occur after a given point.

PAL_MC_DYNAMIC_STATE – Returns Dynamic Processor State (24)

Purpose: Returns the Machine Check Dynamic Processor State.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_DYNAMIC_STATE within the list of PAL procedures.
info_type	Unsigned 64-bit value indicating the type of information to return
dy_buffer	64-bit pointer to a buffer aligned on an 8-byte boundary
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_DYNAMIC_STATE procedure.
max_size	Maximum size (in bytes) of the data that can be returned by this procedure for this processor family.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: The *info_type* input argument designates the type of information the procedure will return. When *info_type* is 0, the procedure returns the maximum size (in bytes) of processor dynamic state that can be returned for this processor family in the *max_size* return value.

When *info_type* is 1, the procedure will copy processor dynamic state into memory pointed to by the input argument *dy_buffer*. This copy will occur using the addressing attributes used to make the procedure call (physical or virtual) and the caller needs to ensure the *dy_buffer* input pointer matches this addressing attribute.

The amount of data returned can vary depending on the state of the machine at the time the procedure is called, and may not always return the maximum size for every call. The amount of data returned is provided in the processor state parameter field *dsize*. Please see Table 11-7 for more information on the processor state parameter. The caller of the procedure needs to ensure that the buffer is large enough to handle the *max_size* that is returned by this procedure.

The contents of the processor dynamic state is implementation dependent. Portions of this information may be cleared by the PAL_MC_CLEAR_LOG procedure. This procedure should be invoked before PAL_MC_CLEAR_LOG to ensure all the data is captured.

PAL_MC_ERROR_INFO – Get Processor Error Information (25)

Purpose: Returns the Processor Machine Check Information

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_ERROR_INFO within the list of PAL procedures.
info_index	Unsigned 64-bit integer identifying the error information that is being requested. (See Table 11-86).
level_index	8-byte formatted value identifying the structure to return error information on.(See Figure 11-19).
err_type_index	Unsigned 64-bit integer denoting the type of error information that is being requested for the structure identified in *level_index*.

Returns:

Return Value	Description
status	Return status of the PAL_MC_ERROR_INFO procedure.
error_info	Error information returned. The format of this value is dependant on the input values passed.
inc_err_type	If this value is zero, all the error information specified by *err_type_index* has been returned. If this value is one, more structure-specific error information is available and the caller needs to make this procedure call again with *level_index* unchanged and *err_type_index,* incremented.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-6	Argument was valid, but no error information was available

Description: This procedure returns error information for machine checks as specified by *info_index*, *level_index* and *err_type_index*. Higher level software is informed that additional machine check information is available when the processor state parameter *mi* bit is set to one. See Table 11-7, “Processor State Parameter Fields,” on page 2:299 for more information on the processor state parameter and the *mi* bit description.

The *info_index* argument specifies which error information is being requested. See Table 11-86 for the definition of the *info_index* values.

Table 11-86. *info_index* Values

info_index	Error Information Type	Description
0	Processor Error Map	This *info_index* value will return the processor error map. This return value specifies the processor core identification, the processor thread identification, and a bit-map indicating which structure(s) of the processor generated the machine check. This bit-map has the same layout as the *level_index*. A one in the structure bit-map indicates that there is error information available for the structure. The layout of the *level_index* is described in Figure 11-19, "level_index Layout" on page 2:411.
1	Processor State Parameter	This *info_index* value will return the same processor state parameter that is passed at the PALE_CHECK exit state for a machine check event (provided a valid min-state save area has been registered) or will construct a processor state parameter for a corrected machine check events. This parameter describes the severity of the error and the validity of the processor state when the machine check or CMCI occurred. This procedure will not return a valid PSP for INIT events. The Processor State Parameter is described in Figure 11-11, "Processor State Parameter," on page 2:299.
2	Structure-specific Error Information	This *info_index* value will return error information specific to a processor structure. The structure is specified by the caller using the *level_index* and *err_type_index* input parameters. The value returned in *error_info* is specific to the structure and type of information requested.

All other values of *info_index* are reserved. When *info_index* is equal to 0 or 1, the *level_index* and *err_type_index* input values are ignored. When *info_index* is equal to 2, the *level_index* and *err_type_index* define the format of the *error_info* return value.

The caller is expected to first make this procedure call with *info_index* equal to zero to obtain the processor error map. This error map informs the caller about the processor core identification, the processor thread identification and indicates which structure(s) caused the machine check. If more than one structure generated a machine check, multiple structure bits will be set. The caller then uses this information to make sub-sequent calls to this procedure for each structure identified in the processor error map to obtain detailed error information.

The *level_index* input argument specifies which processor core, processor thread and structure for which information is being requested. See Table 11-87 on page 2:412 for the definition of the *level_index* fields. This procedure call can only return information about one processor structure at a time. The caller is responsible for ensuring that only one structure bit in the l*evel_index* input argument is set at a time when retrieving information, otherwise the call will return that an invalid argument was passed.

Figure 11-19. *level_index* Layout

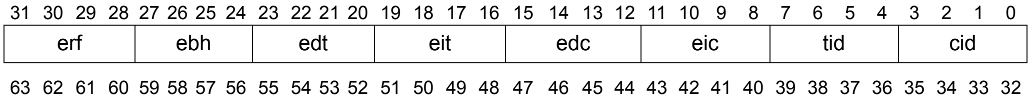


Figure 11-19. *level_index* Layout

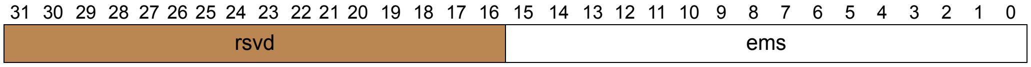


Table 11-87. *level_index* Fields

Field	Bits	Description
cid	3:0	Processor core ID (default is 0 for processors with a single core)
tid	7:4	Logical thread ID (default is 0 for processors that execute a single thread)
eic	11:8	Error information is available for 1st, 2nd, 3rd, and 4th level instruction caches
edc	15:12	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified caches
eit	19:16	Error information is available for 1st, 2nd, 3rd, and 4th level instruction TLB
edt	23:20	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified TLB
ebh	27:24	Error information is available for the 1st, 2nd, 3rd, and 4th level processor bus hierarchy
erf	31:28	Error information is available on register file structures
ems	47:32	Error information is available on micro-architectural structures
rsvd	63:48	Reserved

The convention for levels and hierarchy in the *level_index* field is such that the least significant bit in the error information bit-fields represent the lowest level of the structures hierarchy. For example bit 8 if the *eic* field represents the first level instruction cache.

The *erf* field is 4-bits wide to allow reporting of 4 concurrent register related machine checks at one time. One bit would be set for each error. The *ems* field is 16-bits wide to allow reporting of 16-concurrent micro-architectural structures at one time. There is no significance in the order of these bits. If only one register file related error occurred, it could be reported in any one of the 4-bits.

The *err_type_index* specifies the type of information will be returned in *error_info* for a particular structure. See Table 11-88 for the values of *err_type_index*

Table 11-88. *err_type_index* Values

err_type_index value mod 8	Return Value	Description
0	Structure-specific error information specified by *level_index*	The information returned in *error_info* is dependant on the structure specified in *level_index*. See Table 11-89 for the error_info return formats.
1	Target address	The target address is a 64-bit integer containing the physical address where the data was to be delivered or obtained. The target address also can return the incoming address for external snoops and TLB shoot-downs that generated a machine check. The structure-specific error information informs the caller if there is a valid target address to be returned for the requested structure.
2	Requester identifier	The requester identifier is a 64-bit integer that specifies the bus agent that generated the transaction responsible for generating the machine check. The structure-specific error information informs the caller if there is a valid requester identifier.

Table 11-88. *err_type_index* Values (Continued)

err_type_index value mod 8	Return Value	Description
3	Responder identifier	The responder identifier is a 64-bit integer that specifies the bus agent that responded to a transaction that was responsible for generating the machine check. The structure-specific error information informs the caller if there is a valid responder identifier.
4	Precise instruction pointer	The precise instruction pointer is a 64-bit virtual address that points to the bundle that contained the instruction responsible for the machine check. The structure-specific error information informs the caller if there is a valid precise instruction pointer.
5-7	Reserved	Reserved

See Table 11-89 for the format of *error_info* when structure-specific information is requested.

Table 11-89. *error_info* Return Format when *info_index* = 2 and *err_type_index* = 0

level_index Field Input	*error_info* Return Format
eic	cache_check return format
edc	cache_check return format
eit	tlb_check return format
edt	tlb_check return format
ebh	bus_check return format
erf	reg_file_check return format
ems	uarch_check return format

The structure specified by the *level_index* may have the ability to log distinct multiple errors. This can occur if the structure is accessed at the same time by more than one instruction and the processor can log machine check information for each access. To inform the caller of this occurrence, this procedure will return a value of one in the *inc_err_type* return value.

It is important to note, that when the caller sees that the *inc_err_type* return value is one, it should make a sub-sequent call with the *err_type_index* value incremented by 8. If the structure-specific error information returns that there is a valid target address, requester identifier, responder identifier or precise instruction pointer these can be returned as well by incrementing the *err_type_index* value in the same manner. Refer to the following example for more information.

For example, to gather information on the first error of a structure that can log multiple errors, *err_type_index* would be called with the value of 0 first. The caller examines the information returned in *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise instruction pointer available for logging. If there is, it makes sub-sequent calls with *err_type_index* equal to 1, 2, 3 and/or 4 depending on which valid bits are set. Additionally if the *inc_err_type* return value was set to one, the caller knows that this structure logged multiple errors. To get the second error of the structure it sets the *err_type_index* = 8 and the structure-specific information is returned in *error_info*. The caller examines this *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise instruction pointer available for logging on the second error. If there is, it makes sub-sequent calls with *err_type_index* equal to 9, 10, 11, and/or 12 depending on which valid bits are set. The caller continues incrementing the *err_type_index* value in this fashion until the *inc_err_type* return value is zero.

As shown in Table 11-89, the information returned in *error_info* varies based on which structure information is being requested on. The next sections describe the *error_info* return format for the different structures.

Cache_Check Return Format: The cache check return format is returned in *error_info* when the user requests information on any instruction or data/unified caches in the *level_index* input argument. The cache_check return format must be used to report errors in cacheable transactions. These errors may also be reported using the bus_check return format if the bus structures can detect these errors. The cache_check return format is a bit-field that is described in Figure 11-20 and Table 11-90.

Figure 11-20. cache_check Layout

31 30	29 28 27 26 25 24	23	22	21	20 19 18 17 16	15	14 13 12	11	10	9	8	7 6	5 4	3 2 1 0
hlth	rsvd	dp	rv	wiv	way	mv	mesi	ic	dc	tl	dl	rsvd	level	op

63	62	61	60	59	58	57 56	55	54	53 52	51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
pi	rp	rq	tv	mcc	pv	pl	iv	is	rsvd	index

Table 11-90. cache_check Fields

Field	Bits	Description
op	3:0	Type of cache operation that caused the machine check: 0 – unknown or internal error 1 – load 2 – store 3 – instruction fetch or instruction prefetch 4 – data prefetch (both hardware and software) 5 – snoop (coherency check) 6 – cast out (explicit or implicit write-back of a cache line) 7 – move in (cache line fill) All other values are reserved.
level	5:4	Level of cache where the error occurred. A value of 0 indicates the first level of cache.
rsvd	7:6	Reserved
dl	8	Failure located in the data part of the cache line.
tl	9	Failure located in the tag part of the cache line.
dc	10	Failure located in the data cache
ic	11	Failure located in the instruction cache
mesi	14:12	0 – cache line is invalid. 1 – cache line is held shared. 2 – cache line is held exclusive. 3 – cache line is modified. All other values are reserved.
mv	15	The *mesi* field in the cache_check parameter is valid.
way	20:16	Failure located in the way of the cache indicated by this value.
wiv	21	The *way* and *index* field in the cache_check parameter is valid.
rsvd	22	Reserved
dp	23	An uncorrectable (typically multiple-bit) error was detected and data was poisoned for the corresponding cache line, without any corrupted data being consumed (i.e., no corrupted data has been copied to processor registers).

Table 11-90. cache_check Fields (Continued)

Field	Bits	Description
rsvd	29:24	Reserved
hlth	31:30	Health indicator. This field will report if the cache type and level reporting this error supports hardware status tracking and the current status of this cache. 00 – No hardware status tracking is provided for the cache type and level reporting this event. 01 – Status tracking is provided for this cache type and level and the current status is normal status.[a] 10 – Status tracking is provided for the cache type and level and the current status is cautionary.[a] When a cache reports a cautionary status the "hardware damage" bit of the PSP (see Figure 11-11, "Processor State Parameter," on page 2:299) will be set as well. 11 – Reserved
index	51:32	Index of the cache line where the error occurred.
rsvd	53:52	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the cache_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the cache_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

a. Hardware is tracking the operating status of the structure type and level reporting the error. The hardware reports a "normal" status when the number of entries within a structure reporting repeated corrections is at or below a pre-defined threshold. A "cautionary" status is reported when the number of affected entries exceeds a pre-defined threshold.

TLB_Check Return Format: The tlb_check return format is returned in *error_info* when the user requests information on any instruction or data/unified TLB in the *level_index* input argument. The tlb_check return format is a bit-field that is described in Figure 11-21 and Table 11-91.

Figure 11-21. tlb_check Layout

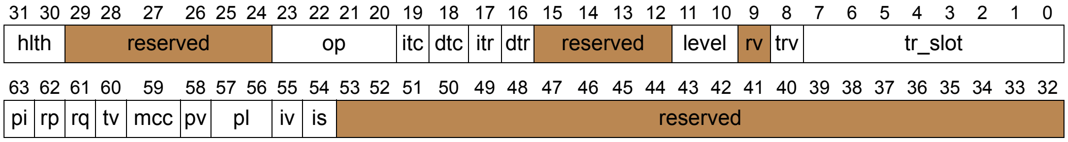


Table 11-91. tlb_check Fields

Field	Bits	Description
tr_slot	7:0	Slot number of the translation register where the failure occurred.
trv	8	The *tr_slot* field in the TLB_check parameter is valid.

Table 11-91. tlb_check Fields (Continued)

Field	Bits	Description
rv	9	Reserved
level	11:10	The level of the TLB where the error occurred. A value of 0 indicates the first level of TLB
reserved	15:12	Reserved
dtr	16	Error occurred in the data translation registers
itr	17	Error occurred in the instruction translation registers
dtc	18	Error occurred in data translation cache
itc	19	Error occurred in the instruction translation cache
op	23:20	Type of cache operation that caused the machine check: 0 – unknown 1 – TLB access due to load instruction 2 – TLB access due to store instruction 3 – TLB access due to instruction fetch or instruction prefetch 4 – TLB access due to data prefetch (both hardware and software) 5 – TLB shoot down access 6 – TLB probe instruction (probe, tpa) 7 – move in (VHPT fill) 8 – purge (insert operation that purges entries or a TLB purge instruction) All other values are reserved.
reserved	29:24	Reserved
hlth	31:30	Health indicator. This field will report if the tlb type and level reporting this error supports hardware status tracking and the current status of this tlb. 00 – No hardware status tracking is provided for the tlb type and level reporting this event. 01 – Status tracking is provided for this tlb type and level and the current status is normal.[a] 10 – Status tracking is provided for the tlb type and level and the current status is cautionary.[a] When a tlb reports a cautionary status the "hardware damage" bit of the PSP (see Figure 11-11, “Processor State Parameter,” on page 2:299) will be set as well. 11 – Reserved
reserved	53:32	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the TLB_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the TLB_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

a. Hardware is tracking the operating status of the structure type and level reporting the error. The hardware reports a "normal" status when the number of entries within a structure reporting repeated corrections is at or below a pre-defined threshold. A "cautionary" status is reported when the number of affected entries exceeds a pre-defined threshold.

Bus_Check Return Format: The bus_check return format is returned in *error_info* when the user requests information on any level of hierarchy of the processor bus structures as specified in the *level_index* input argument. The bus_check return format must be used to report errors in uncacheable transactions. These errors must not be reported using the cache_check return format. The bus_check return format is a bit-field that is described in Figure 11-22 and Table 11-92.

Figure 11-22. bus_check Layout

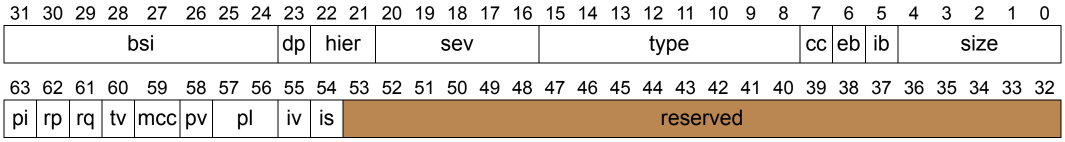


Table 11-92. bus_check Fields

Field	Bits	Description
size	4:0	Size in bytes of the transaction that caused the machine check abort.
ib	5	Internal bus error
eb	6	External bus error
cc	7	Error occurred during a cache to cache transfer.
type	15:8	Type of transaction that caused the machine check abort. 0 – unknown 1 – partial read 2 – partial write 3 – full line read 4 – full line write 5 – implicit or explicit write-back operation 6 – snoop probe 7 – incoming or outgoing ptc.g 8 – write coalescing transactions 9 – I/O space read 10 – I/O space write 11 – inter-processor interrupt message (IPI) 12 – interrupt acknowledge or external task priority cycle All other values are reserved
sev	20:16	Bus error severity. The encodings of error severity are platform specific.
hier	22:21	This value indicates which level or bus hierarchy the error occurred in. A value of 0 indicates the first level of hierarchy.
dp	23	A multiple-bit error was detected, and data was poisoned for the incoming cache line.
bsi	31:24	Bus error status information. It describes the type of bus error. This field is processor bus specific.
reserved	53:32	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the bus_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the bus_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.

Table 11-92. bus_check Fields (Continued)

Field	Bits	Description
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Reg_File_Check Return Format: The reg_file_check return format is returned in *error_info* when the user requests information on any of the registers as specified in the *level_index* input argument. The reg_file_check return format is a bit-field that is described in Figure 11-23 and Table 11-93. When the reg_file_check return format is returned, the target address, the requester identifier and the responder identifier will always be invalid.

Figure 11-23. reg_file_check Layout

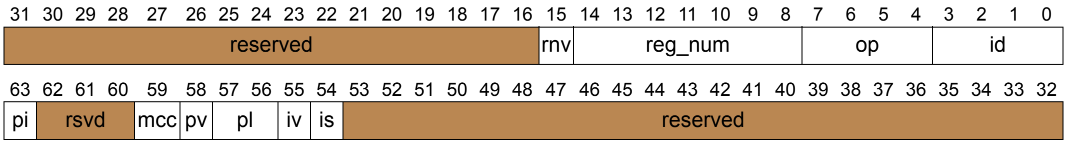


Table 11-93. reg_file_check Fields

Field	Bits	Description
id	3:0	Register file identifier: 0 – unknown/unclassified 1 – General register (bank1) 2 – General register (bank 0) 3 – Floating-point register 4 – Branch register 5 – Predicate register 6 – Application register 7 – Control register 8 – Region register 9 – Protection key register 10 – Data breakpoint register 11 – Instruction breakpoint register 12 – Performance monitor control register 13 – Performance monitor data register All other values are reserved
op	7:4	Identifies the operation that caused the machine check 0 – unknown 1 – read 2 – write All other values are processor specific
reg_num	14:8	Identifies the register number that was responsible for generating the machine check
rnv	15	Specifies if the *reg_num* field is valid
reserved	53:16	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the reg_file_check parameter is valid.

Table 11-93. reg_file_check Fields

Field	Bits	Description
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the reg_file_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
reserved	62:60	Reserved
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Uarch_Check Return Format: The uarch_check return format is returned in *error_info* when the user requests information on any of the micro-architectural structures as specified in the *level_index* input argument. The uarch_check return format is a bit-field that is described in Figure 11-24 and Table 11-94.

Figure 11-24. uarch_check Layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved								xv	wv	way						op				array_id				level			sid				

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
pi	rp	rq	tv	mcc	pv	pl		iv	is	reserved														index							

Table 11-94. uarch_check Fields

Field	Bits	Description
sid	4:0	Structure identification. These bits identify the micro-architectural structure where the error occurred. The definition of these bits are implementation specific.
level	7:5	Level of the micro-architectural structure where the error was generated. A value of 0 indicates the first level.
array_id	11:8	Identification of the array in the micro architectural structure where the error was generated. 0 – unknown/unclassified All other values are implementation specific
op	15:12	Type of operation that caused the error 0 – unknown 1 – read or load 2 – write or store All other values are implementation specific
way	21:16	Way of the micro-architectural structure where the error was located.
wv	22	The *way* field in the uarch_check parameter is valid.
xv	23	The *index* field in the uarch_check parameter is valid.
reserved	31:24	Reserved
index	39:32	Index or set of the micro-architectural structure where the error was located.
reserved	53:40	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the bus_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the bus_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

PAL_MC_ERROR_INJECT – Inject Processor Error (276)

Purpose: Injects the requested processor error or returns information on the supported injection capabilities for this particular processor implementation.

Calling Conv: Stacked

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_MC_ERROR_INJECT within the list of PAL procedures.
err_type_info	Unsigned 64-bit integer specifying the first level error information which identifies the error structure and corresponding structure hierarchy, and the error severity.
err_struct_info	Unsigned 64-bit integer identifying the optional structure specific information that provides the second level details for the requested error.
err_data_buffer	Unsigned 64-bit integer specifying the address of the buffer providing additional parameters for the requested error. The address of this buffer must be 8-byte aligned.

Returns:

Return Value	Description
status	Return status of the PAL_MC_ERROR_INJECT procedure.
capabilities	64-bit vector specifying the supported error injection capabilities for the input argument combination of *struct_hier*, *err_struct* and *err_sev* fields in *err_type_info.*
resources	64-bit vector specifying the architectural resources that are used by the procedure.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-4	Call completed with error; the requested error could not be injected due to failure in locating the target location in the specified structure.
-5	Argument was valid, but requested error injection capability is not supported.
-9	Call requires PAL memory buffer

Description: This procedure enables error injection into processor structures based on information specified by *err_type_info*, *err_struct_info* and *err_data_buffer.* Each invocation of the procedure enables a single error to be injected. The procedure supports error injection for at least one error of each severity type (correctable, recoverable, fatal).

The *err_type_info* argument specifies details of the error injection operation that is being requested (see Figure 11-25). The *err_struct_info* and *err_data_buffer* specify additional optional information. The format of *err_struct_info* is specified for each supported structure type indicated by the *err_struct* field in *err_type_info*. *err_data_buffer* is optional, depending on the structure type and whether *trigger* functionality is used. If *err_data_buffer* is not required for the error injection, PAL will not attempt to access the memory location specified in this parameter.

Figure 11-25. *err_type_info*

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13	12 11 10 9 8	7 6	5 4 3	2 1 0
Reserved	struct_hier	err_struct	err_sev	err_inj	mode

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
Impl_Spec	Reserved

Table 11-95. *err_type_info*

Field	Bits	Description
mode	2:0	Indicates the mode of operation for this procedure: 0 – Query mode 1 – Error inject mode (*err_inj* should also be specified) 2 – Cancel outstanding trigger. All other fields in *err_type_info*, *err_struct_info* and *err_data_buffer* are ignored. All other values are reserved.
err_inj	5:3	Indicates the mode of error injection: 0 – Error inject only (no error consumption) 1 – Error inject and consume All other values are reserved.
err_sev	7:6	Indicates the severity desired for error injection/query. Definitions of the different error severity types is given in Section 11.8, “PAL Glossary” on page 2:350. 0 – Corrected error 1 – Recoverable error 2 – Fatal error 3 – Reserved
err_struct	12:8	Indicates the structure identification for error injection/query: 0 - Any structure (cannot be used during *query mode*). When selected, the structure type used for error injection is determined by PAL. 1 – Cache 2 – TLB 3 – Register file 4 – Bus/System interconnect 5-15 – Reserved 16-31 – Processor specific error injection capabilities. *err_data_buffer* is used to specify error types. Please refer to the processor specific documentation for additional details.
struct_hier	15:13	Indicates the structure hierarchy for error injection/query: 0 - Any level of hierarchy (cannot be used during *query mode*). When selected, the structure hierarchy used for error injection is determined by PAL. 1 – Error structure hierarchy level-1 2 – Error structure hierarchy level-2 3 – Error structure hierarchy level-3 4 – Error structure hierarchy level-4 All other values are reserved.
Reserved	47:16	Reserved
Impl_Spec	63:48	Processor specific error injection capabilities. Please refer to processor specific documentation for additional details.

If *query mode* is selected through the mode bit in the *err_type_info* parameter, the return value in the *capabilities* vector indicates which error injection types are *individually* supported on the underlying implementation for the corresponding values of *err_struct*, *struct_hier* and *err_sev* fields in *err_type_info*. The caller is expected to iterate through all combinations of *err_inj*, *err_sev*, *err_struct*, and *struct_hier* to determine the full extent of *individual* error injection types supported by the underlying implementation.

The *capabilities* vector does not indicate which combinations of error injection inputs from *err_struct_info* are supported by the implementation. For example, if an implementation supports *tag* error injection only for instruction caches and *data* error injection only for data caches, this cannot be determined by the *capabilities* vector. In this instance, the *capabilities* vector will report *i=1, d=1, tag=1, data=1*, indicating that the error injection is supported *individually* for instruction and data caches, and for *tag* and *data* fields, but not indicating which *combinations* of *i*, *d*, *tag*, and *data* are supported for error injection. The caller is required to use the *query mode* with appropriate inputs in *err_struct_info* to determine which combinations of error injection types are supported. If a given combination is not supported, the procedure returns with status -5.

The procedure supports both an *Error inject* and *Error inject and consume* mode (selectable through the *err_inj* field in *err_type_info*). In the former mode, the procedure performs the requested error injection in the specified structure, but does not perform any additional actions that can lead to consumption of the error and generation of the subsequent machine check. In *Error inject and consume* mode, the procedure will inject the error in the specified structure, and will perform additional operations to ensure that the error condition is encountered resulting in a machine check. Note that in this case, the machine check will be generated within the context of this procedure.

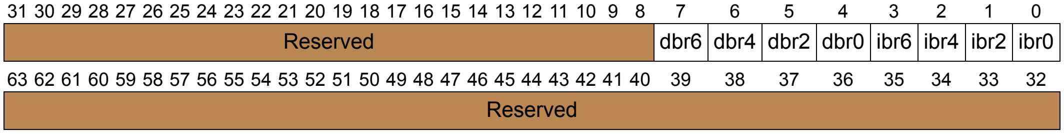
The procedure also provides the ability to set an error injection trigger. In this case, the error injection is delayed until the operation specified by the trigger is encountered and the executing context has the specified privilege level. In the absence of a trigger, the error injection is performed at the time of procedure execution. If an error injection trigger is specified, the mode field in *err_type_info* determines whether the error is injected, or injected and consumed when the trigger operation is encountered. There can be only one outstanding trigger programmed at a time. Subsequent procedure calls that use the trigger functionality will overwrite the previous trigger parameters. Once a trigger is programmed it remains active until either the trigger operation is encountered or software cancels the outstanding trigger via this call. Software can cancel outstanding triggers by specifying *Cancel outstanding trigger* via the *mode* bit in *err_type_info*. The *resources* value returned is all zeroes, indicating that the procedure is no longer using any architectural resources (specified in *resources*) for triggering purposes. When using this mode, it is possible that the procedure execution may itself satisfy the trigger conditions while in the process of cancelling the last programmed trigger.

To support triggers, PAL may use existing architectural resources. The *resources* return value defines the list of resources that are being used by PAL (see Figure 11-26).

In order for triggering to work when PAL is using the IBR or DBR registers, certain PSR bits are required to be set. Software needs to ensure that the PSR.db and the PSR.ic bits are set to one when executing the code that it is targeting with the trigger. If either one of these bits are not set, then triggers will not work as defined.

Procedure operation is undefined if software overwrites or modifies the IBR/DBR resources that PAL indicates it is using for a trigger. The IBR/DBR resources that PAL is not using are available for software to program for their own use.

Figure 11-26. *resources* Return Value



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7	6	5	4	3	2	1	0
Reserved	dbr6	dbr4	dbr2	dbr0	ibr6	ibr4	ibr2	ibr0

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40	39	38	37	36	35	34	33	32
Reserved								

Table 11-96. *resources* Return Value

Field	Bits	Description
ibr0	0	When 1, indicates that IBR0,1 are being used by the procedure for trigger functionality.
ibr2	1	When 1, indicates that IBR2,3 are being used by the procedure for trigger functionality.
ibr4	2	When 1, indicates that IBR4,5 are being used by the procedure for trigger functionality.
ibr6	3	When 1, indicates that IBR6,7 are being used by the procedure for trigger functionality.
dbr0	4	When 1, indicates that DBR0,1 are being used by the procedure for trigger functionality.
dbr2	5	When 1, indicates that DBR2,3 are being used by the procedure for trigger functionality.
dbr4	6	When 1, indicates that DBR4,5 are being used by the procedure for trigger functionality.
dbr6	7	When 1, indicates that DBR6,7 are being used by the procedure for trigger functionality.

Multiprocessor coherency is not guaranteed when error injection is performed using this procedure. Please refer to the processor-specific documentation for further details regarding possible scenarios which can result in loss of coherency.

In cases where an error cannot be injected due to failure in locating the specified target location (cache line, TC, TR, register number) for the given set of input arguments, the procedure will return with status -4. For example, if the caller requests an error injection in the cache and specifies *cl_id*=1 (virtual address provided), then PAL will attempt to locate the cache line as indicated by the input virtual address. If the corresponding cache line cannot be found (the cache line could have been evicted from the cache in the time interval between the procedure call and the search process, or the cache line may be in *invalid* state), then the procedure returns with a status value of -4.

The procedure does not check the settings of the error promotion bits (bit 53 and bit 60 in PAL_PROC_GET_FEATURES) before injecting an error in the specified structure. Based on the configuration of these bits, the severity of the error reported may vary.

The detailed descriptions of *err_struct_info* and *err_data_buffer* are shown below.

Figure 11-27. *err_struct_info* – Cache

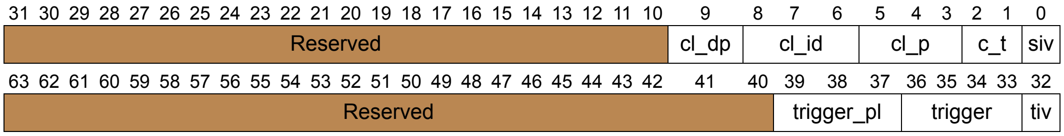


Table 11-97. *err_struct_info* – Cache

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*c_t,cl_p,cl_id*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
c_t	2:1	Indicates which cache should be used for error injection: 0 – Reserved 1 – Instruction cache 2 – Data or unified cache 3 – Reserved
cl_p	5:3	Indicates the portion of the cache line where the error should be injected: 0 – Reserved 1 – Tag 2 – Data 3 – mesi All other values are reserved.

Table 11-97. *err_struct_info* – Cache (Continued)

Field	Bits	Description
cl_id	8:6	Indicates which mechanism is used to identify the cache line to be used for error injection: 0 – Reserved 1 – Virtual address provided in the *inj_addr* field of the buffer pointed to by *err_data_buffer* should be used to identify the cache line for error injection. 2 – Physical address provided in the *inj_addr* field of the buffer pointed to by *err_data_buffer*should be used to identify the cache line for error injection. 3 – *way* and *index* fields provided in *err_data_buffer* should be used to identify the cache line for error injection. All other values are reserved.
cl_dp	9	When 1, indicates that a multiple bit, non-correctable error should be injected in the cache line specified by *cl_id*. If this injected error is not consumed, it may eventually cause a data-poisoning event resulting in a corrected error signal, when the associated cache line is cast out (implicit or explicit write-back of the cache line). The error severity specified by *err_sev* in *err_type_info* must be set to 0 (*corrected error*) when this bit is set.
Reserved	31:10	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*. 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, "Debug Address Breakpoint Match Conditions" on page 2:154. 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, "Debug Address Breakpoint Match Conditions" on page 2:154. All other values are reserved.
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected: 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-28. *capabilities* vector for cache

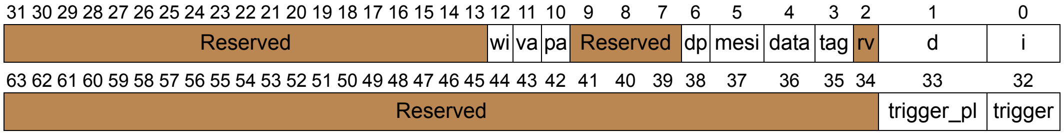


Table 11-98. *capabilities* vector for cache

Field	Bits	Description
i	0	Error injection for instruction caches is supported
d	1	Error injection for data caches is supported
rv	2	Reserved

Table 11-98. *capabilities* vector for cache (Continued)

Field	Bits	Description
tag	3	Error injection in *tag* portion of cache line is supported
data	4	Error injection in *data* portion of cache line is supported
mesi	5	Error injection in *mesi* portion of cache line is supported
dp	6	Error injection that results in data poisoning events is supported
Reserved	9:6	Reserved
pa	10	Error injection with physical address input is supported
va	11	Error injection with virtual address input is supported
wi	12	Error injection with *way* and *index* input is supported
Reserved	31:13	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer needs to be specified for *cache* only if *siv* is 1 or *tiv* is 1, in *err_struct_info*.

Figure 11-29. Buffer pointed to by *err_data_buffer* – Cache

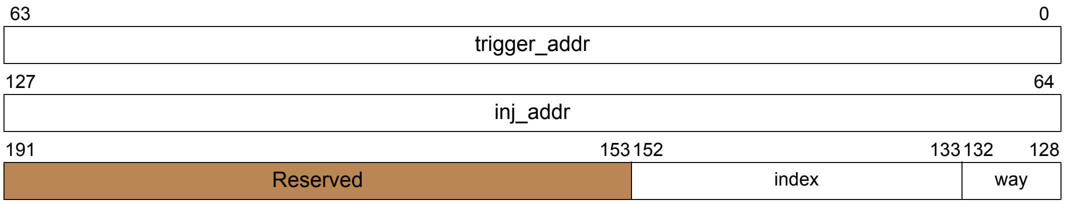


Table 11-99. Buffer pointed to by *err_data_buffer* – Cache

Field	Bits	Description
trigger_addr	63:0	64-bit virtual address to be used by the *trigger* in the *err_struct_info* input argument. This field is ignored if *tiv* in *err_struct_info* is 0. The field is defined similar to the *addr* field in the debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 2:153.
inj_addr	127:64	64-bit virtual or physical address used to identify the cache line to be used for error injection. This field is valid only if *cl_id* in *err_struct_info* corresponds to either *va* or *pa* (value 1 or 2).
way	132:128	Indicates the *way* information for error injection. This is used in combination with the *index* field to identify the cache line for error injection. This field is valid only if *cl_id* in *err_struct_info* is 3, else it is ignored.
index	152:133	Indicates the *index* information for error injection. This is used in combination with the *way* field to identify the cache line for error injection. This field is valid only if *cl_id* in *err_struct_info* is 3, else it is ignored.
Reserved	191:153	Reserved

Figure 11-30. *err_struct_info* – TLB

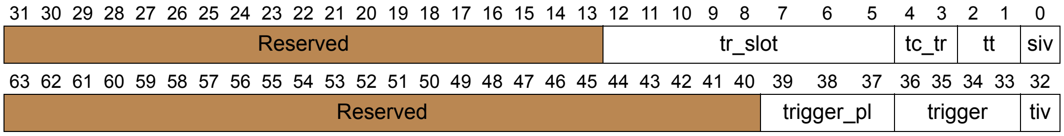


Table 11-100. *err_struct_info* – TLB

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*tt, tc_tr, tr_slot*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
tt	2:1	Indicates which TLB should be used for error injection: 0 – Reserved 1 – Instruction TLB 2 – Data TLB 3 – Reserved
tc_tr	4:3	Indicates which portion of TLB should be used for error injection: 0 – Reserved 1 – tc: error should in injected in a Translation Cache (TC) entry. For TC insertion, the entry is identified by the *vpn* and *rid* fields in *err_data_buffer* 2 – tr: error should in injected in a Translation Register (TR) entry. For TR insertion, the slot number is specified by the *tr_slot* field. 3 – Reserved
tr_slot	12:5	Indicates the Translation Register (TR) slot number where the error should be injected. This field is valid only when *tc_tr* is 2, else it is ignored.
Reserved	31:13	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The virtual address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*. 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 2:154. 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 2:154. All other values are reserved.
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-31. *capabilities* vector for TLB

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3	2	1	0
Reserved	tr	tc	rv	i	d

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34	33	32
Reserved	trigger_pl	trigger

Table 11-101. *capabilities* vector for TLB

Field	Bits	Description
d	0	Error injection for data TLB is supported
i	1	Error injection for instruction TLB is supported
rv	2	Reserved
tc	3	Error injection in TC entries is supported
tr	4	Error injection in TR entries is supported
Reserved	31:5	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer needs to be specified for *TLB* only if *tiv* is 1 or if *tc_tr* value corresponds to *tc*, in *err_struct_info*.

Figure 11-32. Buffer pointed to by *err_data_buffer* – TLB

63 0
trigger_addr

127	115 64
Reserved	vpn

191 152	151 133 132 128
Reserved	rid

Table 11-102. Buffer pointed to by *err_data_buffer* – TLB

Field	Bits	Description
trigger_addr	63:0	64-bit virtual address to be used by the *trigger* in the *err_struct_info* input argument. The field is defined similar to the *addr* field in debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 2:153.
vpn	115:64	Indicates the Virtual page number. This field is valid only when *tc_tr* in *err_struct_info* is 1. *vpn* used in combination with *rid* to identify the TC entry for error injection.
Reserved	127:116	Reserved
rid	151:128	Indicates the region identifier. This field is valid only when *tc_tr* in *err_struct_info* is 1. *rid* is used in combination with *vpn* to identify the TC entry for error injection.
Reserved	191:152	Reserved

Figure 11-33. *err_struct_info* – Register File

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13	12 11 10 9 8 7 6 5	4 3 2 1	0
Reserved	reg_num	regfile_id	siv

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40	39 38 37	36 35 34 33	32
Reserved	trigger_pl	trigger	tiv

Table 11-103. *err_struct_info* – Register File

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*regfile_id, reg_num*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
regfile_id	4:1	Identifies the register file where the error should be injected: 0 – Any register file type. When selected, the register file used for error injection is determined by PAL. 1 – General register (bank0)(GR16-31) 2 – General register (bank1)(GR0-127) 3 – Floating point register 4 – Branch register 5 – Predicate register 6 – Application register 7 – Control register 8 – Region register 9 – Protection key register 10 – Data breakpoint register 11 – Instruction breakpoint register 12 – Performance monitor control register 13 – Performance monitor data register All other values are reserved.
reg_num	12:5	Indicates the register number where the error should be injected. Procedure operation is undefined if there is a conflict between the register number chosen for error injection, and the registers being used by the procedure for code execution (see PAL calling conventions, Section 11.9.2). 0-127: Specific register number corresponding to *regfile_id* 128-254: Reserved for future use 255: Any register number. When selected, the actual register number used for error injection is determined by PAL.
Reserved	31:13	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*. 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 2:154 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 2:154. All other values are reserved.
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected: 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-34. *capabilities* Vector for Register File

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17	16	15 14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	regnum	rsvd	pmd	pmc	ibr	dbr	pkr	rr	cr	ar	pr	br	fr	gr_b1	gr_b0
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49	48	47 46 45	44	43	42	41	40	39	38	37	36	35	34	33	32
Reserved														trigger_pl	trigger

Table 11-104. *capabilities* Vector for Register File

Field	Bits	Description
gr_b0	0	Error injection for General register (bank0) is supported
gr_b1	1	Error injection for General register (bank1) is supported
fr	2	Error injection for Floating point register is supported
br	3	Error injection for Branch register is supported
pr	4	Error injection for Predicate register is supported
ar	5	Error injection for Application register is supported
cr	6	Error injection for Control register is supported
rr	7	Error injection for Region register is supported
pkr	8	Error injection for Protection key register is supported
dbr	9	Error injection for Data breakpoint register is supported
ibr	10	Error injection for Instruction breakpoint register is supported
pmc	11	Error injection for Performance monitor control register is supported
pmd	12	Error injection for Performance monitor data register is supported
Reserved	15:13	Reserved
regnum	16	Error injection with register number input is supported
Reserved	31:17	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer needs to be specified for *register file* only if *tiv* in *err_struct_info* is 1.

Figure 11-35. Buffer pointed to by *err_data_buffer* – Register File

63	0
trigger_addr	

Table 11-105. Buffer pointed to by *err_data_buffer* – Register File

Field	Bits	Description
trigger_addr	63:0	64-bit address to be used by the *trigger* in the *err_struct_info* input argument. The field is defined similar to the *addr* field in the debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 2:153.

Figure 11-36. *err_struct_info* – Bus/Processor Interconnect

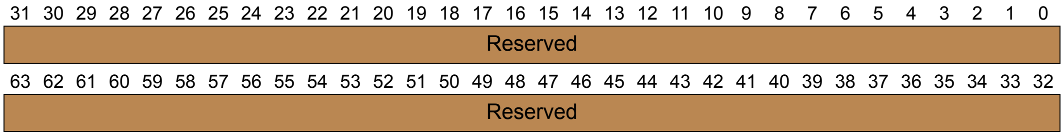


Table 11-106. *err_struct_info* – Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

Figure 11-37. *capabilities* vector for Bus/Processor Interconnect

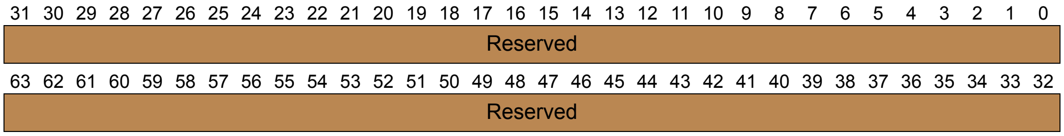


Table 11-107. *capabilities* vector for Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

err_data_buffer does not need to be specified for *bus/system interconnect*.

PAL_MC_HW_TRACKING – Query which hardware structures are performing hardware status tracking (51)

Purpose: Provide a way to query which hardware structures are performing hardware status tracking for corrected machine check events.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_MC_HW_TRACKING within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_HW_TRACKING procedure.
hw_track	64-bit vector denoting which hardware structures are providing hardware status tracking. See Figure 11-38.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure will return information about which hardware structures are providing hardware status tracking for corrected machine check events. This information is also returned in the error logs for corrected machine check events.

The layout of the tracked return value is shown in Figure 11-38.

Figure 11-38. Layout of *hw_track* Return Value

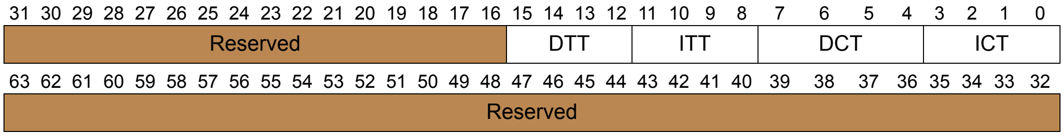


Table 11-108. *hw_check* Fields

Field	Bits	Description
ICT	3:0	Instruction cache tracking. This is a 4-bit vector denoting which levels of instruction cache provide hardware tracking.
DCT	7:4	Data cache tracking. This is a 4-bit vector denoting which levels of data/unified caches provide hardware tracking.
ITT	11:8	Instruction TLB tracking. This is a 4-bit vector denoting which levels of the instruction TLB provide hardware tracking.
DTT	15:12	Data TLB tracking. This is a 4-bit vector denoting which levels of data/unified TLB provide hardware tracking.
Reserved	63:16	Reserved

The convention for the levels in the *hw_track* field is such that the least significant bit in the field represents the lowest level of the structures hierarchy. For example, bit 0 of the ICT field represents the first level instruction cache.

PAL_MC_EXPECTED – Set/Reset Expected Machine Check Indicator (23)

Purpose: Informs PALE_CHECK whether a machine check is expected so that PALE_CHECK will not attempt to correct any expected machine checks.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_EXPECTED within the list of PAL procedures.
expected	Unsigned integer with a value of 0 or 1 to set or reset the hardware resource PALE_CHECK examines for expected machine checks.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_EXPECTED procedure.
previous	Unsigned integer denoting whether a machine check was previously expected.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: If the argument *expected* contains a value of 1, an implementation-dependent hardware resource is set to inform PALE_CHECK to expect a machine check. If the argument *expected* is 0, the resource is reset, so that PALE_CHECK does not expect any following machine checks. All other values of *expected* are reserved.

The implementation-dependent hardware resource should be, by default, in the "not expected" state. Software or firmware should only call PAL_MC_EXPECTED immediately prior to issuing an instruction which might generated an expected machine check. It should then immediately reset the bit to the "not expected" state after checking the results of the operation.

The *previous* return parameter indicates the previous state of the hardware resource to inform PALE_CHECK of an expected machine check. A value of 0 indicates that a machine check was not expected. A value of 1 indicated that a machine check was expected. All other values of *previous* are reserved.

PAL_MC_REGISTER_MEM – Register Memory with PAL for Machine Check and Init (27)

Purpose: Registers a platform dependent location with PAL to which it can save minimal processor state in the event of a machine check or initialization event.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_REGISTER_MEM within the list of PAL procedures.
address	Physical address of the buffer to be registered with PAL.
size	Unsigned integer indicating the size in kilobytes (KB) of the buffer passed. This input argument is only required when passing in a size greater than 4KB. The implementation indicates when a size greater than 4KB is required at the reset hand-off. Refer to Section 11.2.2.1, "Definition of SALE_ENTRY State Parameter" on page 2:291 for more information.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_REGISTER_MEM procedure.
req_size	Returns the required size of the min-state save area in kilobytes (KB) if the *size* input argument did not match the required size for this implementation.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is used to register with PAL an uncacheable min-state save area memory buffer that is used for machine check and initialization event handling. The size of the min-state save area is either 4KB or a larger size that is indicated in the reset hand-off state described in Section 11.2.2.1, "Definition of SALE_ENTRY State Parameter" on page 2:291. The input argument *size* indicates the size of the min-state save buffer in kilobytes (KB) when it is greater than 4KB. If the *size* input argument does not match the required size, the procedure returns an invalid argument return status and a min-state area is not registered. The procedure will also return the required size of the min-state save area in the *req_size* return value.

The layout of the min-state save area is defined in Section 11.3.2.4, "Processor Min-state Save Area Layout" on page 2:302. The address passed has a minimum alignment requirement of 512-bytes.

PAL_MC_RESUME – Restore Minimal Architected State and Return (26)

Purpose: Restores the minimal architectural processor state, sets the CMC interrupt if necessary, and resumes execution.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_RESUME within the list of PAL procedures.
set_cmci	Unsigned 64 bit integer denoting whether to set the CMC interrupt. A value of 0 indicates not to set the interrupt, a value of 1 indicated to set the interrupt, and all other values are reserved.
save_ptr	Physical address of min-state save area used to used to restore processor state.
new_context	Unsigned 64-bit integer denoting whether the caller is returning to a new context. A value of 0 indicates the caller is returning to the interrupted context, a value of 1 indicates that the caller is returning to a new context.

Returns:

Return Value	Description
status	Return status of the PAL_MC_RESUME procedure[a].
Reserved	0
Reserved	0
Reserved	0

[a] This procedure returns to the caller only in an error situation.

Status:

Status Value	Description
-2	Invalid argument
-3	Call completed with error

Description: This procedure will restore the processor minimal architected state and optionally set the CMC interrupt.

If the *set_cmci* argument is set to one, this procedure will set the CMC interrupt and return to the interrupted context. The CMC interrupt handler will be invoked sometime after returning to the interrupted context.

The *save_ptr* argument specifies the processor min-state save area buffer from which the processor state will be restored. This pointer has the same alignment and size restrictions as the address passed to PAL_MC_REGISTER_MEM procedure on page 2:435.

This procedure is used to resume execution of the interrupted context for both machine check and initialization events. This procedure can resume execution to the same context or a new context. If software attempts to resume execution for these events without using this call, processor behavior is undefined.

If the caller is resuming to the same context, the *new_context* argument must be set to 0 and the *save_ptr* argument has to point to a copy of the min-state save area written by PAL when the event occurred.

If the caller is resuming to a new context, the new_context argument must be set to 1 and the save_ptr argument must point to a new min-state save area set up by the caller.

Please see Section 11.3.3, "Returning to the Interrupted Process" on page 2:305 3for more information on resuming to the interrupted context.

PAL_MEM_ATTRIB – Get Memory Attributes (5)

Purpose: Returns the memory attributes implemented by processor.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MEM_ATTRIB within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

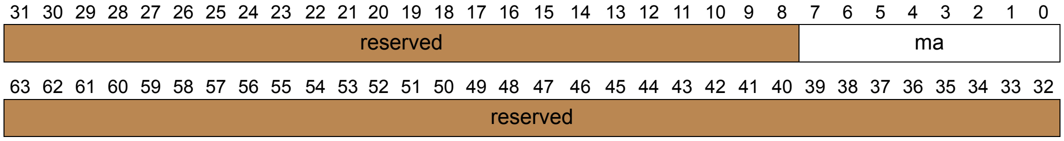
Return Value	Description
status	Return status of the PAL_MEM_ATTRIB procedure.
attrib	8-bit vector of memory attributes implemented by processor.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: Returns a 8-bit vector in the low order 8 bits of the return register that specifies the set of memory attributes implemented by the processor. The return register is formatted as follows:

Figure 11-39. Layout of *attrib* Return Value



Each bit in the bit field *ma* represents one of the eight possible memory attributes implemented by the processor. The bit field position corresponds to the numeric memory attribute encoding defined in Section 4.4, “Memory Attributes” on page 2:75.

PAL_MEMORY_BUFFER – Allocate a cacheable memory buffer for exclusive PAL usage (277)

Purpose: Provides cacheable memory to PAL for exclusive use during runtime.

Calling Conv: Stacked

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MEMORY_BUFFER within the list of PAL procedures.
base_address	Physical address of the memory buffer allocated for PAL use.
alloc_size	Unsigned integer denoting the size of the memory buffer.
control_word	Formatted bit vector that provides control options for this procedure. See Table 11-109.

Returns:

Return Value	Description
status	Return status of the PAL_MEMORY_BUFFER procedure.
min_size	Returns the minimum size of the memory buffer required if the *alloc_size* input argument was not large enough.
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call has not completed a buffer relocation due to a pending interrupt
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure is used to provide PAL firmware a cacheable memory buffer for its exclusive use as well as the ability to relocate this buffer at a later point in time if necessary. PAL provides information at reset hand-off about the minimum buffer size required by this procedure, and also indicates if this procedure is required to be called for correct functionality of the processor. See Section 11.2.2, "PALE_RESET Exit State" on page 2:289 for additional information on the reset hand-off state.

The *base_address* input argument specifies the beginning address for the memory buffer. The *alloc_size* input argument specifies the size of the memory buffer allocated for PAL use. The minimum alignment requirement for this buffer is 4K. If the *base_address* is not at least 4K aligned, the procedure will return an invalid argument. If the *alloc_size* input argument is smaller than the minimum size passed at PAL reset handoff state, the procedure will return an invalid argument and provide the minimum size required in the *min_size* return argument.

The *control_word* input argument specifies if this procedure is being used to register the memory buffer or if it is being used to relocate the memory buffer. The format of the *control_word* is shown in Table 11-109.

Table 11-109. *control_word* Layout

Field	Bits	Description
reg	0	Value of 0 indicates registration for the first time of the buffer. A value of 1 indicates a relocation of the buffer.
int	1	Value of 1 indicates that the procedure should periodically poll for pending external interrupts. If this value is 0, interrupts will be masked during the execution of the entire procedure.
Reserved	63:2	Reserved

A memory buffer must be allocated for each physical package, and is shared by all logical processors on that package. Software is required to call this procedure on all logical processors on a given package with the same input values. If not, processor operation is undefined.

If the PAL reset hand-off state indicates that the memory buffer is required but no call is made to allocate the memory buffer for a given physical package before calling buffer-dependent PAL procedures on a logical processor on that package, those procedures return an error.

If software would like to relocate this memory buffer at a later point in time, it can do so by setting the value of *reg* field in *control_word* to one. PAL will copy the contents of the existing buffer to a new buffer. Software is still required to make this call on all logical processors with the same input arguments when relocating the buffer. Once the call has been made on all logical processors in the physical package, the old memory can be reclaimed.

Software can choose if it wants this procedure to periodically poll for interrupts during the execution of the procedure. If an interrupt is seen, the procedure will return a value of 1 and software must re-call this procedure again on the same logical processor, with the same input arguments, until the copy is completed. If this procedure returns with a value of 1, both the old memory buffer and the new memory buffer will be in use by PAL until PAL returns that the procedure has completed execution successfully by setting the return value to 0.

An error will be returned if software calls this procedure with the *reg* value set to one to re-register a buffer and a call has never been made to register the buffer.

It is required that PAL firmware only perform cacheable memory accesses to this buffer.

PAL_PERF_MON_INFO – Get Processor Performance Monitor Information (15)

Purpose: Returns Performance Monitor information about what can be counted and how to configure the monitors to count the desired events.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PERF_MON_INFO within the list of PAL procedures.
pm_buffer	An address to an 8-byte aligned 128-byte memory buffer.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PERF_MON_INFO procedure.
pm_info	Information about the performance monitors implemented.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_PERF_MON_INFO is called to determine the number of performance monitors and the events which can be counted on the performance monitors. For more information on performance monitoring, see Section 7.2, “Performance Monitoring” on page 2:155. *pm_info* is a formatted 64-bit return register, as shown in Figure 11-40.

.

Figure 11-40. Layout of *pm_info* Return Value

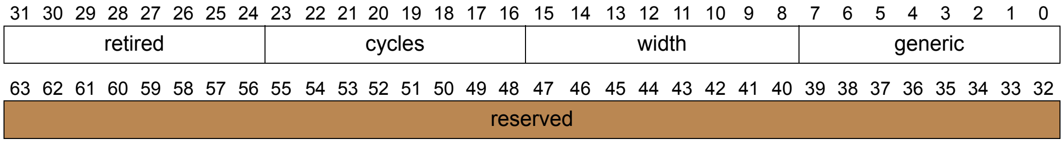


Table 11-110. *pm_info* Fields

Field	Description
generic	Unsigned 8-bit number defining the number of generic PMC/PMD pairs.
width	Unsigned 8-bit number in the range 0:60 defining the number of implemented counter bits.
cycles	Unsigned 8-bit number defining the event type for counting processor cycles.
retired	Unsigned 8-bit number defining the event type for retired instruction bundles.

The *pm_buffer* argument points to a 128-byte memory area where mask information is returned. The layout of *pm_buffer* is shown in Table 11-111.

Table 11-111. *pm_buffer* Layout

Offset	Description
0x0	256-bit mask defining which PMC registers are implemented.
0x20	256-bit mask defining which PMD registers are implemented.

Table 11-111. *pm_buffer* Layout (Continued)

Offset	Description
0x40	256-bit mask defining which registers can count cycles.
0x60	256-bit mask defining which registers can count retired bundles.

PAL_PLATFORM_ADDR – Set Processor Interrupt Block Address and I/O Port Space Address (16)

Purpose: Specifies the physical address of the processor Interrupt Block and I/O Port Space.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PLATFORM_ADDR within the list of PAL procedures.
type	Unsigned 64-bit integer specifying the type of block. 0 indicates that the processor interrupt block pointer should be initialized. 1 indicates that the processor I/O block pointer should be initialized.
address	Unsigned 64-bit integer specifying the address to which the processor I/O block or interrupt block shall be set. The address must specify an implemented physical address on the processor model, bit 63 is ignored.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PLATFORM_ADDR procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: PAL_PLATFORM_ADDR specifies the physical address that the processor shall interpret as accesses to the SAPIC memory or the I/O Port space areas.

The default value for the Interrupt block pointer is 0x00000000 FEE00000. If an alternate address is selected by this call, it must be aligned on a 2 MB boundary, else the procedure will return an error status. The address specified must also not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary.

The default value for the I/O block pointer is to the beginning of the 64 MB block at the highest physical address supported by the processor. Therefore, its physical address is implementation dependent. If an alternate address is selected by this call, it must be aligned on a 64MB boundary, else the procedure will return an error status. The address specified must also not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary.

The Interrupt and I/O Block pointers should be initialized by firmware before any Inter-Processor Interrupt messages or I/O Port accesses. Otherwise the default block pointer values will be used.

Some processor implementations may not support relocation of the interrupt and I/O block pointers and an unimplemented procedure return status will be returned. In these cases the default address spaces will be used.

PAL_PMI_ENTRYPOINT – Setup SAL PMI Entrypoint in Memory (32)

Purpose: Sets the SAL PMI entrypoint in memory.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PMI_ENTRYPOINT within the list of PAL procedures.
SAL_PMI_entry	256-byte aligned physical address of SAL PMI entrypoint in memory.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PMI_ENTRYPOINT procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: This procedure is called to set the SAL PMI entrypoint so that the SAL PMI code shall be executed out of main memory instead of the firmware address space. Some processor implementations will allow initialization of the PMI entrypoint only once. Under those situations, this procedure may be called only once after a boot to initialize the PMI entrypoint register. Subsequent calls will return a status of -3. This call must be made before PMI is enabled by SAL.

PAL_PREFETCH_VISIBILITY – Make Processor Prefetches Visible (41)

Purpose: Used in the architected sequences for memory attribute transitions described in Section 4.4.11, "Memory Attribute Transition" on page 2:88 to transition a page (or set of pages) from a one memory attribute to another.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PREFETCH_VISIBILITY within the list of PAL procedures.
trans_type	Unsigned integer specifying the type of memory attribute transition that is being performed
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PREFETCH_VISIBILITY procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error; this call is not necessary on remote processors
0	Call completed without error; this call must also be performed on all remote processors in the coherence domain
-2	Invalid argument
-3	Call completed with error

Description: This call is intended to be used only in the architected sequences described in Section 4.4.11, "Memory Attribute Transition" on page 2:88.

The *trans_type* input indicates the type of memory attribute transition the user is making. An input value of 0 is used when transition virtual memory attributes only. A value of 1 is used when transitioning physical memory attributes only, or when transitioning memory that may have a combination of virtual and physical memory attributes. All other values are reserved.

This procedure, when used for transitioning virtual memory attributes, will ensure that all prefetches that were initiated by the processor to the cacheable, speculative memory prior to the call, will either not be cached; have been aborted; or are visible to subsequent `fc` instructions. (from both the local processor and from remote processors).

This procedure when used for transitioning physical memory attributes will ensure that all prefetches that were initiated by the processor to the cacheable, limited speculative memory prior to the call, will either not be cached; have been aborted; or are visible to subsequent `fc` instructions (from both the local processor and from remote processors). It will also terminate the ability for the processor to make speculative references to any limited speculation pages. For the processor to make any speculative reference to a limited speculation page after this call, there must be a verified reference made to that page after this call. See the discussion on limited speculation in Section 4.4.6.1, "Limited Speculation and the WBL Physical Addressing Attribute" on page 2:81.

This procedure, when used to delete a memory range on-line, will ensure that all of the conditions described in both of the preceding paragraphs regarding transition of virtual memory attributes and physical memory attributes are met.

If the processor implementation does not require this procedure call to be made on remote processors in the sequences, this procedure will return a 1 upon successful completion.

A return value of 0 upon successful completion of this procedure is an indication to software that the processor implementation requires that this call be performed on all processors in the coherence domain to make prefetches visible in the sequences.

These return code can be used to tune the architected sequence to the particular system on which is running; see Section 4.4.11, “Memory Attribute Transition” for details.

PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)

Purpose: Provides information about configurable processor features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PROC_GET_FEATURES within the list of PAL procedures.
Reserved	0
feature_set	Feature set information is being requested for.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PROC_GET_FEATURES procedure.
features_avail	64-bit vector of features implemented. See Table 11-112.
feature_status	64-bit vector of current feature settings. See Table 11-112.
feature_control	64-bit vector of features controllable by software.

Status:

Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

Description: PAL_PROC_GET_FEATURES and PAL_PROC_SET_FEATURES procedure calls are used together to describe current settings of processor features and to allow modification of some of these processor features.

The *feature_set* input argument for PAL_PROC_GET_FEATURES describes which processor *feature_set* information is being requested. Table 11-112 describes processor *feature_set* zero. The *feature_set* values are split into two categories: architected and implementation-specific. The architected feature sets have values from 0-15. The implementation-specific feature sets are values 16 and above. The architected feature sets are described in this document. The implementation-specific feature sets are described in processor-specific documentation.

This procedure will return an invalid argument if an unsupported architectural *feature_set* is passed as an input. Implementation-specific feature sets will start at 16 and will expand in an ascending order as new implementation-specific feature sets are added. The return *status* is used by the caller to know which implementation-specific feature sets are currently supported on a particular processor.

For each valid *feature_set*, this procedure returns which processor features are implemented in the *features_avail* return argument, the current feature setting is in *feature_status* return argument, and the feature controllability in the *feature_control* return argument. Only the processor features which are implemented and controllable can be changed via PAL_PROC_SET_FEATURES. Features for which features_avail are 0 (unimplemented features) also have features_status and features_control of 0.

In Table 11-112, the *class* field indicates whether a feature is required to be available (*Req.*) or is optional (*Opt.*). The *control* field indicates which features are required to be controllable. *Req.* indicates that the feature must be controllable, *Opt.* indicates that the feature may optionally be controllable, and *No* indicates that the feature cannot be controllable. The *control* field applies only when the feature is available. The sense of the bits is chosen so that for features which are controllable, the default hand-off value at exit from PALE_RESET should be 0. PALE_CHECK and PALE_INIT will not modify these features.

Table 11-112. Processor Features

Bit	Class	Control	Scope	Description
63	Opt.	Req.	May[a]	Enable BERR promotion. When 1, the Bus Error (BERR) signal is promoted to the Bus Initialization (BINIT) signal, and the BINIT pin is asserted on the occurrence of each Bus Error. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
62	Opt.	Req.	May	Enable MCA promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Error signal, and the BERR pin is assert on each occurrence of an MCA. Setting this bit has no effect if BERR signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
61	Opt.	Req.	May	Enable MCA to BINIT promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Initialization signal, and the BINIT pin is assert on each occurrence of an MCA. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
60	Opt.	Req.	No[b]	Enable CMCI promotion When 1, Corrected Machine Check Interrupts (CMCI) are promoted to MCAs. They are also further promoted to BERR if bit 39, Enable MCA promotion, is also set and they are promoted to BINIT if bit 38, Enable MCA to BINIT promotion, is also set. This bit has no effect if MCA signalling is disabled (see PAL_BUS_GET/SET_FEATURES)
59	Opt.	Req.	May	Disable Cache. When 0, the processor performs cast outs on cacheable pages and issues and responds to coherency requests normally. When 1, the processor performs a memory access for each reference regardless of cache contents and issues no coherence requests and responds as if the line were not present. Cache contents cannot be relied upon when the cache is disabled. WARNING: Semaphore instructions may not be atomic or may cause Unsupported Data Reference faults if caches are disabled.
58	Opt.	Req.	May	Disable Coherency. When 0, the processor uses normal coherency requests and responses. When 1, the processor answers all requests as if the line were not present.
57	Opt.	Req.	May	Disable Dynamic Power Management (DPM). When 0, the hardware may reduce power consumption by removing the clock input from idle functional units. When 1, all functional units will receive clock input, even when idle.
56	Opt.	Req.	May	Disable a BINIT on internal processor time-out. When 0, the processor may generate a BINIT on an internal processor time-out. When 1, the processor will not generate a BINIT on an internal processor time-out. The event is silently ignored.
55	Opt.	Req.	May	Enable external notification when the processor detects hardware errors caused by environmental factors that could cause loss of deterministic behavior of the processor. When 1, this bit will enable external notification, when 0 external notification is not provided. The type of external notification of these errors is processor-dependent. A loss of processor deterministic behavior is considered to have occurred if these environmentally induced errors cause the processor to deviate from its normal execution and eventually causes different behavior which can be observed at the processor bus pins. Processor errors that do not have this effects (i.e., software induced machine checks) may or may not be promoted depending on the processor implementation.

Table 11-112. Processor Features (Continued)

Bit	Class	Control	Scope	Description
54	Opt.	Req.	No	Enable the use of the vmsw instruction. When 0, the `vmsw` instruction causes a Virtualization fault when executed at the most privileged level. When 1, this bit will enable normal operation of the `vmsw` instruction. This bit has no effect if virtual machine features are disabled (see bit 40).
53	Opt.	Req.	May	Enable MCA signaling on unconsumed data-poisoning event detection. When 0, a CMCI will be signaled on error detection. When 1, an MCA will be signaled on error detection. Note that the reported error severity depends on which method is chosen for signaling; see Section 11.3.2.3, "Unconsumed Data-Poisoning Event Handling" for details.If this feature is not supported, then the corresponding argument is ignored when calling PAL_PROC_SET_FEATURES. Note that the functionality of this bit is independent of the setting in bit 60 (Enable CMCI promotion), and that the bit 60 setting does not affect CMCI signaling for data-poisoning related events.
52	Opt.	Req.	May	Disable P-states. Provides the ability to disable p-states when they are implemented by the processor. When the feature is available and status is 1 or when the feature is not available, the PAL P-state procedures (PAL_PSTATE_INFO, PAL_SET_PSTATE, PAL_GET_PSTATE) will return with a status of -1 (Unimplemented procedure). When the feature is available and the status is 0, the PAL P-state procedures will operate normally.
51:48	N/A	N/A	N/A	Reserved
47	Opt.	Opt.	May	Disable Dynamic branch prediction. When 0, the processor may predict branch targets and speculatively execute, but may not commit results. When 1, the processor must wait until branch targets are known to execute.
46	Opt	Opt.	May	Disable Dynamic Instruction Cache Prefetch. When 0, the processor may prefetch into the caches any instruction which has not been executed, but whose execution is likely. When 1, instructions may not be fetched until needed or hinted for execution. (Prefetch for a hinted branch is allowed even when dynamic instruction cache prefetch is disabled.)
45	Opt.	Opt.	May	Disable Dynamic Data Cache Prefetch. When 0, the processor may prefetch into the caches any data which has not been accessed by instruction execution, but which is likely to be accessed. When 1, no data may be fetched until it is needed for instruction execution or is fetched by an lfetch instruction.
44	Opt.	Req.	No	Disable Spontaneous Deferral. When 1, the processor may optionally defer speculative loads that do not encounter any exception conditions, but that trigger other implementation-dependent conditions (e.g., cache miss). This behavior is gated by the programming model described in Section 5.5.5, "Deferral of Speculative Load Faults" on page 2:105. When 0, spontaneous deferral is disabled.
43	Opt.	Opt.	No	Disable Dynamic Predicate Prediction. When 0, the processor may predict predicate results and execute speculatively, but may not commit results until the actual predicates are known. When 1, the processor shall not execute predicated instructions until the actual predicates are known.
42	Opt.	No	RO[c]	XR1 through XR3 implemented. Denotes whether XR1 - XR3 are implemented for machine check recovery. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
41	Opt.	No	RO	XIP, XPSR, and XFS implemented. Denotes whether XIP, XPSR, and XFS are implemented for machine check recovery. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.

Table 11-112. Processor Features (Continued)

Bit	Class	Control	Scope	Description
40	Opt.	Opt.	No	Virtual Machine features implemented and enabled. When 1, PSR.vm is implemented and virtual machines features are not disabled. When 0 (features_status) and when the corresponding features_avail bit is 1, virtual machines features are implemented but are disabled. When both the features_avail and features_status bits are 0, virtual machine features are not implemented. If implemented and controllable, virtual machine features may be disabled by writing this bit to 0 with PAL_PROC_SET_FEATURES. However, virtual machine features cannot be re-enabled except via a power-on; hence, if virtual machine features are disabled, this bit reads as 0 for both features_status and features_control (but still 1 for features_avail).
39	Opt.	Req.	May	Variable P-state performance: A value of 1 indicates that the processor is optimizing performance for the given P-state power budget by dynamically varying the frequency, such that maximum performance is achieved for the power budget. A value of 0 indicates that P-states have no frequency variation or very small frequency variations for their given power budget.
38	Opt.	No	RO	Simple implementation of unimplemented instruction addresses. Denotes how an unimplemented instruction address is recorded in IIP on an Unimplemented Instruction Address trap or fault. When 1, the full unimplemented address is recorded in IIP; when 0, the address is sign extended (virtual addresses) or zero extended (physical addresses). See Section 3.3.5.3, “Interruption Instruction Bundle Pointer (IIP – CR19)” for details. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
37	Opt.	No	RO	INIT, PMI, and LINT pins present. Denotes the absence of INIT, PMI, LINT0 and LINT1 pins on the processor. When 1, the pins are absent. When 0, the pins are present. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
36	Opt.	No	RO	Unimplemented instruction address reported as fault. Denotes how the processor reports the detection of unimplemented instruction addresses. When 1, the processor reports an Unimplemented Instruction Address fault on the unimplemented address; when 0, it reports an Unimplemented Instruction Address trap on the previous instruction in program order. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
35	Opt.	Req.	May	Disable data speculation and the ALAT. When 1, data speculation checks (`chk.a`) always fail (i.e., always branch to the target address), thus triggering recovery code; check loads (`ld.c`) always re-load the target register. When 0, data speculation works as normal.
34	Opt.	No	RO	Interruption Instruction Bundle interruption registers (IIB0, IIB1) implemented. Denotes whether IIB registers are implemented. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
33	Opt.	No	RO	Interval Timer Offset register (ITO) implemented. Denotes whether ITO register is implemented. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.
32:0	N/A	N/A	N/A	Reserved

a. May-span-multiple-logical-processors. Readers should refer to implementation-specific document for details.
b. Setting this bit affect logical-processor only.
c. Read-only bit.

PAL_PROC_SET_FEATURES – Set Processor Dependent Features (18)

Purpose: Enables/disables specific processor features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PROC_SET_FEATURES within the list of PAL procedures.
feature_select	64-bit vector denoting desired state of each feature (1=select, 0=non-select).
feature_set	Feature set to apply changes to. See PAL_PROC_GET_FEATURES for more information on feature sets.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PROC_SET_FEATURES procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

Description: PAL_PROC_GET_FEATURES should be called to ascertain the implemented processor features and their current setting before calling PAL_PROC_SET_FEATURES. The list of possible processor features is defined in Table 11-112. Any attempt to set processor features which cannot be set will be ignored.

PAL_PSTATE_INFO – Get Information for Power/Performance States (44)

Purpose: Returns information about the P-states supported by the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PSTATE_INFO within the list of PAL procedures.
pstate_buffer	64-bit pointer to a 256-byte buffer aligned on an 8-byte boundary.
Reserved	0
Reserved	0

Returns:

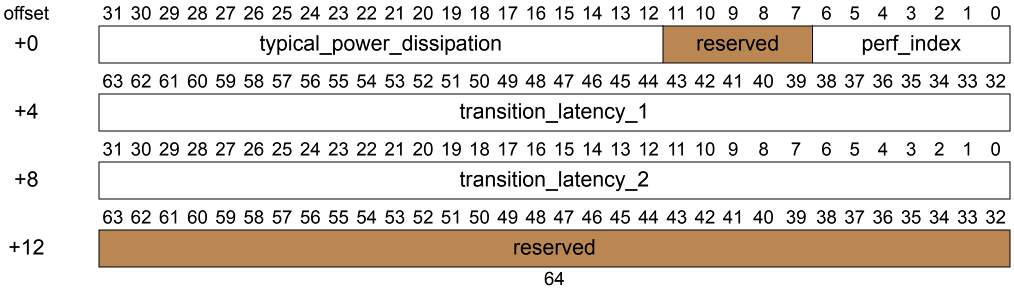
Return Value	Description
status	Return status of the PAL_PSTATE_INFO procedure.
pstate_num	Unsigned integer denoting the number of P-states supported. The maximum value of this field is 16.
dd_info	Dependency domain information
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: Information about available P-states is returned in the data buffer referenced by *pstate_buffer*. Entries in the buffer are organized in an ascending order. For example, P0 (the highest performance P-state) state information is index 0 in the buffer, P1 state is index 1 in the buffer, and so on. The return argument *pstate_num* indicates the number of P-states supported on the given implementation. For example, if *pstate_num* is 4, it indicates that P-states P0-P3 are available for that implementation. Information in *pstate_buffer* is returned only for entries corresponding to the available P-states. Entries corresponding to unimplemented P-states must be ignored. Figure 11-41 illustrates the format of the *pstate_buffer*.

Figure 11-41. Layout of *pstate_buffer* Entry



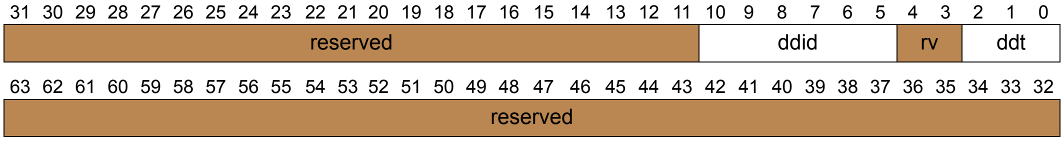
- *typical_power_dissipation* is a 20-bit field denoting the typical processor package power dissipation if all logical processors on the package are placed in this P-state, measured in milliwatts.
- *perf_index* is a 7-bit field denoting the performance index of this P-state, relative to the highest available P-state (P0). This field is enumerated relative to the index of the highest-performing P-state. A value of 100 represents the minimum processor

performance in the P0 state. For example, if the P1-state has a value of 75, and the next P-state (P2) has a value of 50, it implies that P1 performance is 25% lower than P0 performance, and P2 performance is 50% lower than P0 performance.

- *transition_latency_1* is a 32-bit field indicating the minimum number of processor cycles required to initiate a transition to this P-state from any other P-state.
- *transition_latency_2* is a 32-bit field indicating the minimum recommended number of processor cycles that the caller should wait, before initiating a new P-state transition with a reasonable chance of acceptance. This field is intended to give the caller an estimation of the frequency with which PAL_SET_PSTATE procedure calls should be made, without having the transition request be not accepted.

Dependency domain details for the logical processor are returned in *dd_info*. See Figure 11-42 for *dd_info* layout.

Figure 11-42. Layout of *dd_info* Parameter



- *ddt* (Dependency Domain Type) is a 3-bit unsigned integer denoting the type of dependency domains that exist on the processor package. The possible values are shown in Table 11-113. See Section 11.6.1, "Power/Performance States (P-states)" on page 2:315 for details of the values in this field.

Table 11-113. Values for *ddt* Field

Value	Description
0	Hardware independent (HIDD)
1	Hardware coordinated (HCDD)
2	Software coordinated (SCDD)
3-7	Reserved

- *ddid* (Dependency Domain Identifier) is a 6-bit unsigned integer denoting this logical processor's dependency domain. The *ddid* values are unique only for a given processor package. Software can use the *ddid* field to determine which logical processors belong to the same dependency domain within the package.

For more information on performance states and power management, refer to Section 11.6.1, "Power/Performance States (P-states)" on page 2:315.

PAL_PTCE_INFO – Get PTCE Purge Loop Information (6)

Purpose: Returns information required for the architected loop used to purge (initialize) the entire TC.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PTCE_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PTCE_INFO procedure.
tc_base	Unsigned 64-bit integer denoting the beginning address to be used by the first PTCE instruction in the purge loop.
tc_counts	Two unsigned 32-bit integers denoting the loop counts of the outer (loop 1) and inner (loop 2) purge loops. count1 (loop 1) is contained in bits 63:32 of the parameter, and count2 (loop 2) is contained in bits 31:0 of the parameter.
tc_strides	Two unsigned 32-bit integers denoting the loop strides of the outer (loop 1) and inner (loop 2) purge loops. stride1 (loop 1) is contained in bits 63:32 of the parameter, and stride2 (loop 2) is contained in bits 31:0 of the parameter.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: No explicit hardware support is required by this call. See the purge loop example in the description of the `ptc.e` instruction in Chapter 2, “Instruction Reference” in Volume 3.

PAL_REGISTER_INFO – Return Information about Implemented Processor Registers (39)

Purpose: Returns information about implemented Application and Control Registers.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_REGISTER_INFO within the list of PAL procedures.
info_request	Unsigned 64-bit integer denoting what register information is requested.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_REGISTER_INFO procedure.
reg_info_1	64-bit vector denoting information for registers 0-63. Bit 0 is register 0, bit 63 is register 63.
reg_info_2	64-bit vector denoting information for registers 64-127. Bit 0 is register 64, bit 63 is register 127.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

This procedure is called to obtain information about the implementation of Application Registers and Control Registers. Table 11-114 shows the information that is returned for each request.

Table 11-114. *info_request* Return Value

info_request	Meaning of Return Bit Vector
0	A 0-bit in the return vector indicates that the corresponding Application Register is not implemented, a 1-bit in the return vector indicates that the corresponding Application Register is implemented.
1	A 0-bit in the return vector indicated that the corresponding Application Register can be read without side effects, a 1-bit in the return vector indicated that the corresponding Application registers may cause side effects when read.
2	A 0-bit in the return vector indicates that the corresponding Control Register is not implemented, a 1-bit in the return vector indicates that the corresponding Control Register is implemented.
3	A 0-bit in the return vector indicated that the corresponding Control Register can be read without side effects, a 1-bit in the return vector indicated that the corresponding Control Register may cause side effects when read.
All others	Reserved.

PAL_RSE_INFO – Get RSE Information (19)

Purpose: Returns information about the register stack and RSE for this processor implementation.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_RSE_INFO within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_RSE_INFO procedure.
phys_stacked	Number of physical stacked general registers.
hints	RSE hints supported by processor.
Reserved	0

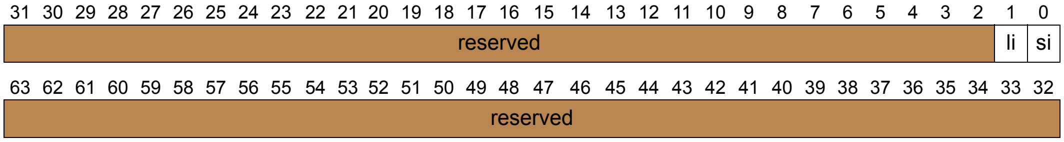
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The return parameter *phys_stacked* contains a 64-bit unsigned integer that specifies the number of physical registers implemented by the processor for the stacked general registers, r32-r127. *phys_stacked* will be an integer multiple of 16 greater than or equal to 96.

The return parameter *hints* contains a 2-bit field that specifies which RSE load/store hints are implemented.

Figure 11-43. Layout of *hints* Return Value



A bit field value of 1 specifies that the corresponding mode is implemented; a value of 0 specifies that the mode is not implemented. The bit field encodings are:

Table 11-115. RSE Hints Implemented

li	si	RSE Hints	Class
0	0	enforced lazy	Required
0	1	eager stores	Optional
1	0	eager loads	Optional
1	1	eager stores and loads	Optional

“Lazy” is the default RSE mode and must be implemented. Hardware is not required to implement any of the other modes.

PAL_SET_HW_POLICY – Set Current Hardware Resource Sharing Policy (49)

Purpose: Sets the current hardware resource sharing policy of the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_SET_HW_POLICY within the list of PAL procedures.
policy	Unsigned 64-bit integer specifying the hardware resource sharing policy the caller is setting.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_SET_HW_POLICY procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed successfully but could not change the hardware policy since a competing logical processor is set in exclusive high priority
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure is used to set the hardware resource sharing policy on the logical processor it is called on. The setting of this policy will impact other logical processors on the physical processor package. The logical processors impacted is returned by the PAL_GET_HW_POLICY procedure, see "PAL_GET_HW_POLICY – Retrieve Current Hardware Resource Sharing Policy (48)" on page 2:394 for details.

The input argument *policy* selects the hardware policy the caller would like to set. The supported hardware policies are listed in Table 11-116 below. By default the hardware always sets the processor in the performance policy at reset.

Table 11-116. Processor Hardware Sharing Policies

Value	Name	Description
0	Performance	The processor has its hardware resources configured to achieve maximum performance across all logical processors.
1	Fairness	The processor configures hardware resources to approximately achieve equal sharing of competing hardware resources among all impacted logical processors.

Table 11-116. Processor Hardware Sharing Policies *(Continued)*

Value	Name	Description
2	High-priority	The processor configures hardware resources to provide the logical processor this procedure was called on a greater share of the competing hardware resources. All competing logical processors will get a smaller share of the competing hardware resources.
3	Exclusive High-priority	The processor configures hardware resources such that the logical processor this procedure was called on has a greater share of the competing hardware resources. All competing logical processors will get a smaller share of the competing hardware resources. This policy also ensures that no other competing logical processor can modify the hardware sharing policy until the logical processor that is in exclusive high priority releases exclusive high-priority by selecting a different policy.
All Other Values		Reserved

The caller must be aware of which logical processors are impacted by hardware policy changes, since making a call on one of the logical processors will impact all logical processors that share the same hardware resources. For example if the caller selects the high-priority policy on one logical processor A and then later in time selects fairness policy on one of the competing logical processors B, the procedure will take away high-priority status from logical processor A and change all impacted logical processors to the fairness policy without an error.

If a caller wants to ensure that high-priority will not be taken away from a logical processor, it can use the exclusive high-priority policy. This policy will return an error if any competing logical processor tries to change the hardware policy. This ensures that the caller can ensure a certain logical processor will retain high-priority status until that status is explicitly released by that logical processor.

This procedure is only supported on processors that have multiple logical processors sharing hardware resources that can be configured. On all other processor implementations, this procedure will return the Unimplemented procedure return status.

PAL_SET_PSTATE – Request Processor to Enter Power/Performance State (263)

Purpose: To request a processor transition to a given P-state.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_SET_PSTATE within the list of PAL procedures.
p_state	Unsigned integer denoting the processor P-state being requested.
force_pstate	Unsigned integer denoting whether the P-state change should be forced for the logical processor.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_SET_PSTATE procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error, but transition request was not accepted
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_SET_PSTATE is used to request the transition of the processor to the P-state specified by the *p_state* input parameter. The PAL_SET_PSTATE procedure does not wait for the transition to complete before returning back to the caller. The request may either be accepted (*status* = 0) or not accepted (*status* = 1), depending on hardware capabilities and implementation-specific event conditions. The presence of a platform power-cap does not prevent the request from being accepted. (See Section 11.6.1, "Power/Performance States (P-states)" on page 2:315 for details.) If the request is not accepted, then no transition is performed, and it is up to the caller to make another PAL_SET_PSTATE procedure call to transition to the desired P-state. When the request is accepted, the processor will attempt to initiate a transition to the requested performance state. For SCDD or HIDD logical processors, the procedure will always succeed in transitioning to the requested performance state. For HCDD logical processors, the procedure will make a best-case attempt at fulfilling the transition request, based on the nature of the dependencies that exist between the logical processors in the domain. In such circumstances, the procedure may initiate no transition, partial transition or full transition to the requested P-state.

The *force_pstate* argument may be used for a HCDD when it is necessary to get a deterministic response for the P-state transition at the expense of compromising the power/performance of other logical processors in the same domain. If the *force_pstate* argument is non-zero, and if the request is accepted, the procedure will initiate the P-state transition on the logical processor regardless of any dependencies that exist in the dependency domain at the time the procedure is called. Forcing the P-state does not change the P-states requested by other logical processors in the dependency domain, nor the value seen on other logical processors when they do a PAL_GET_PSTATE with *type*=0; rather, forcing the P-state effectively suspends hardware coordination. A subsequent call to PAL_SET_PSTATE on any logical processor in the dependency domain (with a *force_pstate* argument of zero) reinstates hardware coordination. The *force_pstate* argument is ignored on SCDD and HIDD logical processors.

Calling this procedure on some processor implementations may affect P-states of other processors in the same dependency domain. Please refer to Section 11.6.1, "Power/Performance States (P-states)" on page 2:315 and implementation-specific reference manuals for details.

PAL_SHUTDOWN – Shutdown the Processor (45)

Purpose: Put the logical processor into a low power state which can be exited only by a reset event.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_SHUTDOWN within the list of PAL procedures.
notify_platform	8-byte aligned physical address pointer providing details on how to optionally notify the platform that the processor is entering a shutdown state.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_SHUTDOWN procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This call places the logical processor in a low power state which can be exited only by asserting a reset. This procedure can optionally let the platform know that it is about to shutdown by performing a store operation as specified in the *notify_platform* input argument.

If the *notify_platform* input argument is zero, no store operation will be performed. If the *notify_platform* input argument is non-zero, the layout for this argument is shown in Table 11-117.

Table 11-117. *notify_platform* Layout

Offset	Description
0x0	Size of the store operation to perform (1, 2, 4 or 8 are the only valid values for this field).
0x8	Aligned physical address of the store operation. The most significant bit (63) of the physical address should be set according to the cacheability attribute wanted for the store transaction.
0x10	Data value for the store operation.
All others	Reserved.

If the address value is not naturally aligned to the size selected, this procedure will return an error.

The logical processor will wait until this transaction has been received by the platform before entering the shutdown state.

On receipt of a reset event, the logical processor will reset itself and start execution at the PAL reset address. All other events will are ignored by the logical processor when in shutdown state.

PAL_TEST_INFO – Information for Processor Self-test (37)

Purpose: Returns the alignment and size requirements needed for the memory buffer passed to the PAL_TEST_PROC procedure as well as information on self-test control words for the processor self-tests.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_TEST_INFO within the list of PAL procedures.
test_phase	Unsigned integer that specifies which phase of the processor self-test information is being requested on. A value of 0 indicates the phase two of the processor self-test and a value of 1 indicates phase one of the processor self-test. All other values are reserved.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_TEST_INFO procedure.
bytes_needed	Unsigned 64-bit integer denoting the number of bytes of main memory needed to perform the second phase of processor self-test.
alignment	Unsigned 64-bit integer denoting the alignment required for the memory buffer.
st_control	48-bit wide bit-field indicating if control of the processor self-tests is supported and which bits of the *test_control* field are defined for use.

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_TEST_INFO returns the size and alignment requirements for the memory buffer that is passed to the PAL_TEST_PROC procedure and returns information on the implementation of the self-test control word based on the *test_phase* input argument. Please see Section 11.2.3, "PAL Self-test Control Word" on page 2:295 for more information on the self-test control word.

When *test_phase* is equal to zero, information is returned about phase two of the processor self-test. These are the tests that require external memory to execute properly. When *test_phase* is equal to one, information is returned about phase one of the processor self-test. These are the tests that are normally run during PALE_RESET and do not require external memory to properly execute. When information is requested about phase one of the processor self-test a memory buffer and alignment argument will be returned as well since these tests may need to save and restore processor state to this memory buffer if executed from the PAL_TEST_PROC procedure.

PAL_TEST_PROC – Perform a Processor Self-test (258)

Purpose: Performs the second phase of processor self test.

Calling Conv: Stacked Registers

PAL_TEST_PROC may modify some registers marked unchanged in the Stacked Register calling convention. See additional description below.

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_TEST_PROC within the list of PAL procedures.
test_address	64-bit physical address of main memory area to be used by processor self-test. The memory region passed must be cacheable, bit 63 must be zero.
test_info	Input argument specifying the size of the memory buffer passed and the phase of the processor self-test that should be run. See Figure 11-44.
test_params	Input argument specifying the self-test control word and the allowable memory attributes that can be used with the memory buffer. See Figure 11-45.

Returns:

Return Value	Description
status	Return status of the PAL_TEST_PROC procedure.
self-test_state	Formatted 8-byte value denoting the state of the processor after self-test. The format is described in Section 11.2.2.3, “Definition of Self Test State Parameter” on page 2:293.
Reserved	0
Reserved	0

Status:

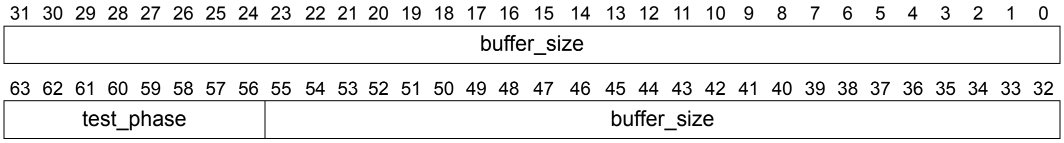
Status Value	Description
1	Call completed without error, but hardware failures occurred during self-test
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The PAL_TEST_PROC procedure will perform a phase of the processor self-tests as directed by the *test_info* and the *test_control* input parameters.

test_address points to a contiguous memory region to be used by PAL_TEST_PROC. This memory region must be aligned as specified by the alignment return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value. The PAL_TEST_PROC routine requires that the memory has been initialized and that there are no known uncorrected errors in the allocated memory.

The *test_info* input parameter specifies the size of the memory buffer passed to the procedure and which phase of the processor self-test is requested to be run (either phase one or phase two).

Figure 11-44. Layout of *test_info* Argument

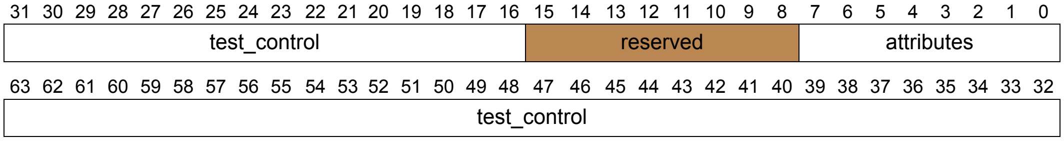


- *buffer_size* indicates the size in bytes of the memory buffer that is passed to this procedure. *buffer_size* must be greater than or equal in size to the *bytes_needed* return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value.

- *test_phase* defines which phase of the processor self-tests are requested to be run. A value of zero indicates to run phase two of the processor self-tests. Phase two of the processor self-tests are ones that require external memory to execute correctly. A value of one indicates to run phase one of the processor self-tests. Phase one of the processor self-tests are tests run during PALE_RESET and do not depend on external memory to run correctly. When the caller requests to have phase one of the processor self-test run via this procedure call, a memory buffer may be needed to save and restore state as required by the PAL calling conventions. The procedure PAL_TEST_INFO informs the caller about the requirements of the memory buffer.

The *test_params* input argument specifies which memory attributes are allowed to be used with the memory buffer passed to this procedure as well as the self-test control word. The self-test control word *test_control* controls the runtime and coverage of the processor self-test phase specified in the *test_phase* parameter.

Figure 11-45. Layout of *test_param* Argument



- *attributes* specifies the memory attributes that are allowed to be used with the memory buffer passed to this procedure. The *attributes* parameter is a vector where each bit represents one of the virtual memory attributes defined by the architecture. The bit field position corresponds to the numeric memory attribute encoding defined in Section 4.4, "Memory Attributes" on page 2:75. The caller is required to support the cacheable attribute for the memory buffer, otherwise an invalid argument will be returned.
- *test_control* is the self-test control word corresponding to the *test_phase* passed. This *test_control* directs the coverage and runtime of the processor self-tests specified by the *test_phase* input argument. Information about the self-test control word can be found in Section 11.2.3, "PAL Self-test Control Word" on page 2:295 and information on if this feature is implemented and the number of bits supported can be obtained by the PAL_TEST_INFO procedure call. If this feature is implemented by the processor, the caller can selectively skip parts of the processor self-test by setting *test_control* bits to a one. If a bit has a zero, this test will be run. The values in the unimplemented bits are ignored. If PAL_TEST_INFO indicated that the self-test control word is not implemented, this procedure will return with an invalid argument status if the caller sets any of the *test_control* bits.

PAL_TEST_PROC will classify the processor after the self-test in one of four states: CATASTROPHIC FAILURE, FUNCTIONALLY RESTRICTED, PERFORMANCE RESTRICTED, or HEALTHY. These processor self-test states are described in Figure 11-9 on page 2:293. If PAL_TEST_PROC returns in the FUNCTIONALLY RESTRICTED or PERFORMANCE RESTRICTED states the *self-test_status* return value can provide additional information regarding the nature of the failure. In the case of a CATASTROPHIC FAILURE, the procedure does not return.

The procedure will only perform memory accesses to the buffer passed to it using the memory attributes indicated in the *attributes* bit-field. The caller must ensure that the memory region passed to the procedure is in a coherent state.

PAL_TEST_PROC may modify PSR bits or system registers as necessary to test the processor. These bits or registers must be restored upon exit from PAL_TEST_PROC with the exception of the translation caches, which are evicted as a result of testing. PAL_TEST_PROC is free to invalidate all cache contents. If the caller depends on the contents of the cache, they should be flushed before making this call. PAL_TEST_PROC requires that the RSE is set up properly to handle spills and fills to a valid memory location if the contents of the register stack are needed. PAL_TEST_PROC requires that the memory buffer passed to it is not shared with other processors running this procedure in the system at the same time. PAL_TEST_PROC will use this memory region in a non-coherent manner. PAL_TEST_PROC may overwrite floating point registers 32-127 without restoring their values upon exit.

PAL_VERSION – Get PAL Version Number Information (20)

Purpose: Returns PAL version information.

Calling Conv: Static registers only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VERSION within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VERSION procedure.
min_pal_ver	8-byte formatted value returning the minimum PAL version needed for proper operation of the processor. See Figure 11-46.
current_pal_ver	8-byte formatted value returning the current PAL version running on the processor. See Figure 11-46.
Reserved	0

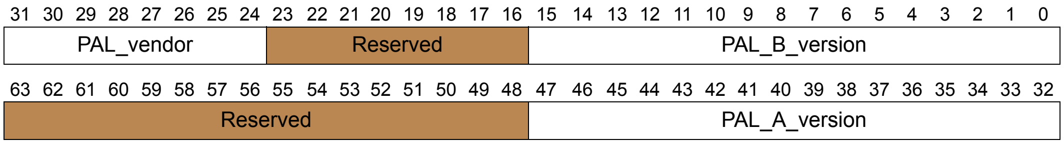
Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: PAL_VERSION provides the caller the minimum PAL version needed for proper operation of the processor as well as the current PAL version running on the processor.

The *min_pal_ver* and *current_pal_ver* return values are 8-byte values in the following format:

Figure 11-46. Layout of *min_pal_ver* and *current_pal_ver* Return Values



- *PAL_B_version* is a 16-bit binary coded decimal (BCD) number that provides identification information about the PAL_B firmware.
- *PAL_vendor* is an unsigned 8-bit integer indicating the vendor of the PAL code.
- *PAL_A_version* is a 16-bit binary coded decimal (BCD) number that provides identification information about the PAL_A firmware. In the split PAL_A model, this return value is the version number of the processor-specific PAL_A. The generic PAL_A version is not returned by this procedure in the split PAL_A model.

The version numbers selected for the PAL_A and PAL_B firmware is specific to the *PAL_vendor*. The version numbers selected will always have the property that later versions of firmware will have a higher number than earlier versions of firmware.

PAL_VM_INFO – Get Virtual Memory Information (7)

Purpose: Return information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VM_INFO within the list of PAL procedures.
tc_level	Unsigned 64-bit integer specifying the level in the TLB hierarchy for which information is required. This value must be between 0 and one less than the value returned in the *vm_info_1.num_tc_levels* return value from PAL_VM_SUMMARY.
tc_type	Unsigned 64-bit integer with a value of 1 for instruction translation cache and 2 for data or unified translation cache. All other values are reserved.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VM_INFO procedure.
tc_info	8-byte formatted value returning information about the specified TC.
tc_pages	64-bit vector containing a bit for each page size supported in the specified TC, where bit position n indicates a page size of 2**n.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument.
-3	Call completed with error.

Description: The *tc_info return* is an 8-byte quantity in the following format:

Figure 11-47. Layout of *tc_info* Return Value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">num_entries</td><td colspan="8">num_ways</td><td colspan="8">num_sets</td></tr>
</table>

<table>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="29">Reserved</td><td>tr</td><td>ut</td><td>pf</td></tr>
</table>

- *num_sets* – Unsigned 8-bit integer denoting the number of hash sets for the specified level (1=fully associative)
- *num_ways* – Unsigned 8-bit integer denoting the associativity of the specified level (1=direct).
- *num_entries* – Unsigned 16-bit integer denoting the number of entries in the specified TC.
- *pf* – Flag denoting whether the specified level is optimized for the region’s preferred page size (1=optimized). *tc_pages* indicates which page sizes are usable by this translation cache.
- *ut* – Flag denoting whether the specified TC is unified (1=unified).
- tr – Flag denoting whether installed translation registers will reduce the number of entries within the specified TC.

The *num_entries* will always equal *num_ways* * *num_sets*. For a direct mapped TC, *num_ways* = 1 and *num_sets* = *num_entries*. For a fully associative TC, *num_sets* = 1 and *num_ways* = *num_entries*.

PAL_VM_PAGE_SIZE – Get Virtual Memory Page Size Information (34)

Purpose: Returns page size information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VM_PAGE_SIZE within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VM_PAGE_SIZE procedure.
insertable_pages	64-bit vector containing a bit for each architected page size that is supported for TLB insertions and region registers.
purge_pages	64-bit vector containing a bit for each architected page size supported for TLB purge operations.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: The values returned from this call are all 64-bit bitmaps. One bit is set for each page size implemented by the processor where bit n represents a page size of 2**n. Please refer to Table 4-5 on page 2:58 for the minimum page sizes that are supported.

The *insertable_pages* returns the page sizes that are supported for TLB insertions and region registers.

The *purge_pages* returns the page sizes that are supported for the TLB purge operations.

PAL_VM_SUMMARY – Get Virtual Memory Summary Information (8)

Purpose: Returns summary information about the virtual memory characteristics of the processor implementation.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VM_SUMMARY within the list of PAL procedures.
Reserved	0
Reserved	0
Reserved	0

Returns:

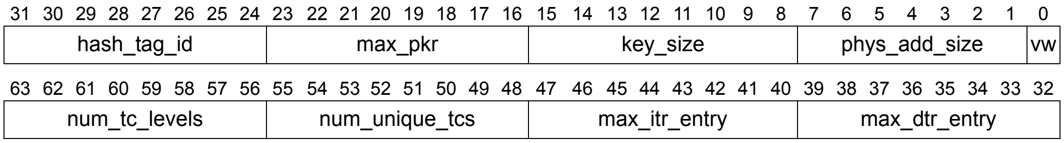
Return Value	Description
status	Return status of the PAL_VM_SUMMARY procedure.
vm_info_1	8-byte formatted value returning global virtual memory information.
vm_info_2	8-byte formatted value returning global virtual memory information.
Reserved	0

Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: The *vm_info_1* return is an 8-byte quantity in the following format:

Figure 11-48. Layout of *vm_info_1* Return Value



- *vw* – 1-bit flag indicating whether a hardware TLB walker is implemented (1 = walker present).
- *phys_add_size* – Unsigned 7-bit integer denoting the number of bits of physical address implemented.
- *key_size* – Unsigned 8-bit integer denoting the number of bits implemented in the PKR.key field.
- *max_pkr* – Unsigned 8-bit integer denoting the maximum PKR index (number of PKRs-1).
- *hash_tag_id* – Unsigned 8-bit integer which uniquely identifies the processor hash and tag algorithm.
- *max_dtr_entry* – Unsigned 8 bit integer denoting the maximum data translation register index (number of dtr entries - 1).
- *max_itr_entry* – Unsigned 8 bit integer denoting the maximum instruction translation register index (number of itr entries - 1).
- *num_unique_tcs* – Unsigned 8-bit integer denoting the number of unique TCs implemented. This is a maximum of 2**num_tc_levels*.
- *num_tc_levels* – Unsigned 8-bit integer denoting the number of TC levels.

The *vm_info_2* return is an 8-byte quantity in the following format:

Figure 11-49. Layout of *vm_info_2* Return Value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">max_purges</td><td colspan="8">rid_size</td><td colspan="8">impl_va_msb</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="32">Reserved</td></tr>
</table>

- *impl_va_msb* – Unsigned 8-bit integer denoting the bit number of the most significant virtual address bit. This is the total number of virtual address bits - 1.
- *rid_size* – Unsigned 8-bit integer denoting the number of bits implemented in the RR.rid field.
- *max_purges* – Unsigned 16 bit integer denoting the maximum number of concurrent outstanding TLB purges allowed by the processor. A value of 0 indicates one outstanding purge allowed. A value of 2^{16}-1 indicates no limit on outstanding purges. All other values indicate the actual number of concurrent outstanding purges allowed.

PAL_VM_TR_READ – Read a Translation Register (261)

Purpose: Reads a translation register.

Calling Conv: Stacked Registers

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VM_TR_READ within the list of PAL procedures.
reg_num	Unsigned 64-bit number denoting which TR to read.
tr_type	Unsigned 64-bit number denoting whether to read an ITR (0) or DTR (1). All other values are reserved.
tr_buffer	64-bit pointer to the 32-byte memory buffer in which translation data is returned.

Returns:

Return Value	Description
status	Return status of the PAL_VM_TR_READ procedure.
TR_valid	Formatted bit vector denoting which fields are valid. See Figure 11-50.
Reserved	0
Reserved	0

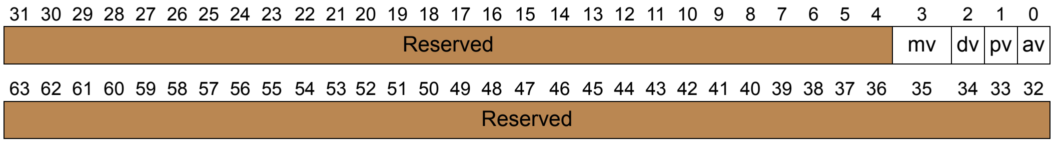
Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

Description: This procedure reads the specified translation register and returns its data in the buffer starting at *tr_buffer*. The format of the data is returned in Translation Insertion Format, as described in Figure 4-5, "Translation Insertion Format," on page 2:54. In addition, bit 0 of the IFA in Figure 4-5 (an ignored field in the figure) will return whether the translation is valid. If bit 0 is 1, the translation is valid.

Some fields of the translation register returned may be invalid. The validity of these fields is indicated by the return argument *TR_valid*. If these fields are not valid, the caller should ignore the indicated fields when reading the translation register returned in *tr_buffer*.

Figure 11-50. Layout of *TR_valid* Return Value



- av – denotes that the access rights field is valid
- pv – denotes that the privilege level field is valid
- dv – denotes that the dirty bit is valid
- mv – denotes that the memory attributes are valid.

A value of 1 denotes a valid field. A value of 0 denotes an invalid field. Any value returned in an invalid field must be ignored.

The *tr_buffer* parameter should be aligned on an 8 byte boundary.

Note: This procedure may have the side effect of flushing all the translation cache entries depending on the implementation.

PAL_VP_CREATE – PAL Create New Virtual Processor (265)

Purpose: Initializes a new *vpd* for the operation of a new virtual processor in the virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_CREATE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
host_iva	64-bit host virtual pointer to the host IVT for the virtual processor
opt_handler	64-bit non-zero host-virtual pointer to an optional handler for virtualization intercepts. See Section 11.7.3, "PAL Intercepts in Virtual Environment" on page 2:332 for details.

Returns:

Return Value	Description
status	Return status of the PAL_VP_CREATE procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: Initializes a new *vpd* for the operation of a new virtual processor within the virtual environment.

The caller must pass a pointer to the new Virtual Processor Descriptor (*vpd*) as argument. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped with either a DTR or DTC. See Section 11.10.2.1.3, "Making PAL Procedure Calls in Physical or Virtual Mode" on page 2:359 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. The *vac*, *vdc* and *virt_env_vaddr* parameters in the VPD must already be initialized before calling this procedure. Invalid argument is returned on unsupported *vac*/*vdc* combinations. See Section 11.7.4.4, "Virtualization Optimization Combinations" on page 2:349 for details.

The *host_iva* parameter specifies the host IVT to handle IVA-based interruptions when this virtual processor is running. The VMM can use the same or different *host_iva* for each virtual processor. The *opt_handler* specifies an optional virtualization intercept handler. If a non-zero value is specified, all virtualization intercepts are delivered to this handler. If a zero value is specified, all virtualization intercepts are delivered to the Virtualization vector in the host IVT. If the VMM relocates the IVT specified by the *host_iva* parameter and/or the virtualization intercept handler specified by the *opt_handler* parameter after this procedure, PAL_VP_REGISTER must be called to register the new host IVT and virtualization intercept handler before resuming virtual processor execution or allowing any IVA-based interruptions to occur; otherwise processor operation is undefined.

Upon return, the VMM is responsible for setting up the rest of the VMD state before the new virtual processor is launched (via PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER).

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 for details.

PAL_VP_ENV_INFO – PAL Virtual Environment Information (266)

Purpose: Returns the parameters needed to enter a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_ENV_INFO within the list of PAL procedures
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_ENV_INFO procedure
buffer_size	Unsigned integer denoting the number of bytes required by the PAL virtual environment buffer during PAL_VP_INIT_ENV
vp_env_info	64-bit vector of virtual environment information. See Table 11-118. for details
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure returns the configuration options and the PAL virtual environment buffer size required by PAL_VP_INIT_ENV. This procedure is used by the VMM to setup a virtual environment and determine the amount of memory / resources required. The VMM can then allocate the required amount of physical memory, set up the virtual to physical instruction and data translations that cover the PAL virtual environment buffer in TRs and call PAL_VP_INIT_ENV. The buffer allocated must be at least 4K aligned.

On a multiprocessor system, this procedure need only be invoked once (on any one logical processor) to obtain virtual environment information.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 for details.

Table 11-118. *vp_env_info* – Virtual Environment Information Parameter

Field	Bit	Description
Reserved	7:0	Reserved
opcode	8	If 1, hardware support to provide opcode information during PAL intercepts is available. The opcode (and the decoding of cause) passed as parameters to the VMM on intercept will represent the instruction that triggered the intercept. If 0, opcode information during PAL intercepts is provided by PAL. The opcode (and the decoding of cause) passed as parameters to the VMM on intercept will not necessarily represent the instruction that triggered the intercept, but may represent some value that was written to memory between the time the instruction that triggered the intercept was fetched, and when the intercept was triggered.
Reserved	9	Reserved
gitc	10	If 1, guest MOV-from-AR.ITC optimization is supported.[a] If 0, guest MOV-from-AR.ITC optimization is not supported.

Table 11-118. *vp_env_info* – Virtual Environment Information Parameter

Field	Bit	Description
Reserved	31:11	Reserved
probe	32	If 1, processor supports interception of probe instructions. See Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details on the usage of this control. If 0, intercept of probe instructions is not supported.
tf	33	If 1, guest test feature optimization is supported. If 0, this optimization is not supported. See Section 11.7.4.2.9, “Test Feature Optimization” on page 2:345 for details.
ic_um	34	If 1, guest interruption collection and user mask optimization is supported. If 0, this optimization is not supported. See Section 11.7.4.2.10, “Interruption Collection and User Mask Optimization” on page 2:345 for details.
Reserved	63:35	Reserved

a. Architecturally, an implementation which supports guest MOV-from-AR.ITC will also support the interval timer offset (ITO) register.

PAL_VP_EXIT_ENV – PAL Exit Virtual Environment (267)

Purpose: Allows a logical processor to exit a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_EXIT_ENV within the list of PAL procedures
iva	Optional 64-bit host virtual pointer to the IVT when this procedure is done
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_EXIT_ENV procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure allows a logical processor to exit a virtual environment.

Upon successful execution of the PAL_VP_EXIT_ENV procedure and if the *iva* parameter is non-zero, the IVA control register will contain the value from the *iva* parameter.

On a multiprocessor system, the VMM must allow the last logical processor in this environment to complete the procedure before freeing the memory resource allocated to the virtual environment.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, "Processor Virtualization" on page 2:44 and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details.

PAL_VP_INFO – PAL Virtual Processor Information (50)

Purpose: Returns information about virtual processor features.

Calling Conv: Static

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VP_INFO within the list of PAL procedures
feature_set	Feature set information is being requested for.
vp_buffer	An address to an 8-byte aligned memory buffer (if used).
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_INFO procedure
vp_info	Information about the virtual processor.
vmm_id	Unique identifier for the VMM.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-8	Specified feature_set is not implemented

Description: The PAL_VP_INFO procedure call is used to describe virtual processor features.

The *feature_set* input argument for PAL_VP_INFO describes which virtual-processor *feature_set* information is being requested, and is composed of two fields as shown:

63 48	47 0
vmm_id	index
16	48

A *vmm_id* of 0 indicates architected feature sets, while others are implementation-specific feature sets. Implementation-specific feature sets are described in VMM-specific documentation.

This procedure will return a -8 if an unsupported *feature_set* argument is passed as an input. The return status is used by the caller to know which feature sets are currently supported on a particular VMM. This procedure always returns unimplemented (-1) when called on physical processors.

For each valid *feature_set*, this procedure returns information about the virtual processor in *vp_info*. Additional information may be returned in the memory buffer pointed to by *vp_buffer*, as needed. Details, for a given implementation-specific *feature_set*, of whether information is returned in the buffer, the size of the buffer, and the representation of this information in the buffer and in *vp_info* are described in VMM-specific documentation.

Architected *feature_set* 0 (*vmm_id* 0, *index* 0) is defined and required to be implemented (if this procedure is implemented), but there are no architected features defined in it yet, and so all bits in *vp_info* are reserved for architected *feature_set* 0. Other architected feature sets (*vmm_id* 0, *index*>0) are undefined, and return -8 (Specified *feature_set* is not implemented). Software can call PAL_VP_INFO with a *feature_set* argument of 0 to get the *vmm_id*, although *vmm_id* is also returned for any other implemented feature sets as well. For *feature_set* 0, the *vp_buffer* argument is ignored.

PAL_VP_INIT_ENV – PAL Initialize Virtual Environment (268)

Purpose: Allows a logical processor to enter a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_INIT_ENV within the list of PAL procedures
config_options	64-bit vector of global configuration settings – See Table 11-119. for details
pbase_addr	Host physical base address of a block of contiguous physical memory for the PAL virtual environment buffer – This memory area must be allocated by the VMM and be 4K aligned. The first logical processor to enter the environment will initialize the physical block for virtualization operations.
vbase_addr	Host virtual base address of the corresponding physical memory block for the PAL virtual environment buffer – The VMM must maintain the host virtual to host physical data and instruction translations in TRs for addresses within the allocated address space. Logical processors in this virtual environment will use this address when transitioning to virtual mode operations.

Returns:

Return Value	Description
status	Return status of the PAL_VP_INIT_ENV procedure
vsa_base	Virtualization Service Address – VSA specifies the virtual base address of the PAL virtualization services in this virtual environment.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure allows a logical processor to enter a virtual environment. This call must be made after calling PAL_VP_ENV_INFO and before calling other PAL virtualization procedures and services. All of the logical processors in a virtual environment share the same **PAL virtual environment buffer**. The buffer must be 4K aligned. The first logical processor entering the virtual environment initializes the buffer provided by the VMM. Subsequent processors can enter the virtual environment at any time and will not perform initialization to the buffer.

PAL_VP_ENV_INFO must be called before this procedure to determine the configuration options and size requirements for the virtual environment. The VMM is required to maintain the ITR and DTR translations of the PAL virtual environment buffer throughout this procedure. See "PAL_VP_ENV_INFO – PAL Virtual Environment Information (266)" on page 2:473 for more information on PAL_VP_ENV_INFO.

After this procedure, it is optional for the VMM to maintain the TR mapping for the PAL virtual environment buffer. If the TR translations for the buffer are not installed, the VMM must not make any PAL virtualization service calls; and the VMM must be prepared to handle DTLB faults during any PAL virtualization procedure calls.

Table 11-119 shows the layout of the *config_options* parameter. The *config_options* parameter configures the global configuration options and global virtualization optimizations for all the logical processors in the virtual environment. All logical processors in the virtual environment must specify the same value in the *config_options* parameter during PAL_VP_INIT_ENV, otherwise processor operation is undefined.

Table 11-119. *config_options* – Global Configuration Options

	Field	Bit	Description
Global Configuration Options	initialize	0	If 1, this procedure will initialize the PAL virtual environment buffer for this virtual environment. If 0, this procedure will not initialize the PAL virtual environment buffer. On a multiprocessor system, the VMM must wait until this procedure completes on the first logical processor before calling this procedure on additional logical processors; otherwise processor operation is undefined.
	fr_pmc	1	If 1, for virtualization intercepts the performance counters are disabled by setting PSR.up and pp to 0, see Section 11.7.3.1, “PAL Virtualization Intercept Handoff State” on page 2:333 for details on PSR settings at virtualization intercepts; for all other IVA-based interruptions PSR.pp and up are set according to Interruption State column described in Processor Status Field table described in Table 3-2, “Processor Status Register Fields” on page 2:24. The VMM must have DCR.pp equal to 0 when the *fr_pmc* option is 1, whenever the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. See Section 11.7.2, “Interruption Handling in a Virtual Environment” on page 2:331 and Table 11-21, “IVA Settings after PAL Virtualization-related Procedures and Services” on page 2:332 for details on per-virtual-processor host IVT. If 0, PSR.pp and up are set according to Interruption State column described in Processor Status Field table described in Table 3-2, “Processor Status Register Fields” on page 2:24
	be	2	Big-endian – Indicates the endian setting of the VMM. If 1, the values in the VPD are stored in big-endian format and the PAL services calls are made with PSR.be bit equal to 1. If 0, the values in the VPD are stored in little-endian format and the PAL services calls are made with PSR.be bit equal to 0. The VMM must match DCR.be with the value set in this field when the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. See Section 11.7.2, “Interruption Handling in a Virtual Environment” on page 2:331 and Table 11-21, “IVA Settings after PAL Virtualization-related Procedures and Services” on page 2:332 for details on per-virtual-processor host IVT.
	Reserved	7:3	Reserved.

Table 11-119. ***config_options*** **– Global Configuration Options (Continued)**

	Field	Bit	Description
Global Virtualization Optimizations	opcode	8	This bit must be set to 1 – opcode information will be provided to the VMM during PAL intercepts within the virtual environment. This opcode may or may not be guaranteed to be the opcode that triggered the intercept. See Table 11-118, "vp_env_info – Virtual Environment Information Parameter" on page 2:473 for details. This procedure returns an error if this bit is not set to 1.
	cause	9	If 1, the causes of virtualization intercepts will be provided to the VMM during PAL intercept handoffs within the virtual environment. No information will be provided if 0. See Section 11.7.3.1, "PAL Virtualization Intercept Handoff State" on page 2:333 for details of virtualization intercept handoffs.
	gitc	10	If 1, enables guest MOV-from-AR.ITC optimization. For details see Section 11.7.4.1.3, "Guest MOV-from-AR.ITC Optimization" on page 2:337 and Section 3.3.4.4, "Interval Timer Offset (ITO – CR4)" on page 2:34. This bit is reserved if guest MOV-from-AR.ITC optimization is not supported.
	Reserved	62:11	Reserved.
	impl	63	Implementation-specific configuration option. This field is ignored if not implemented. Please refer to processor-specific documentation for details.

The *fr_pmc* bit in the global *config_options* parameter specifies whether the performance counters will be frozen when the Virtualization optimizations specified in the Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*) are running. When a virtual processor is running, the *vac* field in the corresponding VPD specifies whether a certain virtualization accelerations are enabled. If the *fr_pmc* in the virtual environment was also enabled, the performance counters will be frozen when the enabled virtualization optimizations are running. See Section 11.7.4, "Virtualization Optimizations" on page 2:335 for details on Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*).

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, "Processor Virtualization" on page 2:44 and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details.

PAL_VP_REGISTER – PAL Register Virtual Processor (269)

Purpose: Register a different host IVT and/or a different optional virtualization intercept handler for the virtual processor specified by *vpd*.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_REGISTER within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
host_iva	64-bit host virtual pointer to the host IVT for the virtual processor
opt_handler	64-bit non-zero host-virtual pointer to an optional handler for virtualization intercepts. See Section 11.7.3, “PAL Intercepts in Virtual Environment” on page 2:332 for details.

Returns:

Return Value	Description
status	Return status of the PAL_VP_REGISTER procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_VP_REGISTER registers a different host IVT and/or a different optional virtualization intercept handler specific to the virtual processor specified by *vpd*. On creation of a virtual processor by PAL_VP_CREATE, the VMM specifies a host IVT specific to the virtual processor. This procedure allows the VMM to specify a host IVT different from the one specified during PAL_VP_CREATE.

The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped with either a DTR or DTC. See Section 11.10.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:359 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. The *virt_env_vaddr* parameter in the VPD must be setup with the host virtual address of the PAL virtual environment buffer before calling this procedure.

The *host_iva* parameter specifies the host IVT to handle IVA-based interruptions when this virtual processor is running. The VMM can use the same or different *host_iva* for each virtual processor. The *opt_handler* specifies an optional virtualization intercept handler. If a non-zero value is specified, all virtualization intercepts are delivered to this handler. If a zero value is specified, all virtualization intercepts are delivered to the Virtualization vector in the host IVT. Upon completion of this procedure, the VMM must not relocate the IVT specified by the *host_iva* parameter and/or the virtualization intercept handler specified by the *opt_handler* parameter. The VMM can call this procedure again in case it wishes to associate a different host IVT and/or virtualization intercept handler with the virtual processor.

PAL_VP_REGISTER returns invalid argument on unsupported virtualization optimization combinations in *vpd*. See Section 11.7.4.4, “Virtualization Optimization Combinations” on page 2:349 for details.

This procedure can be used by the VMM to:

- Relocate the host IVT associated with the virtual processor.
- Specify a different optional virtualization intercept handler for the virtual processor.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 for details.

PAL_VP_RESTORE – PAL Restore Virtual Processor (270)

Purpose: Restores virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_RESTORE within the list of PAL procedures.
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD.)
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_RESTORE procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_VP_RESTORE performs an implementation-specific restore operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* and the PAL virtual environment buffer must be mapped by instruction and data translation registers (TR). The instruction and data translation must be maintained until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE and a different host IVT is set up by the VMM by writing to the IVA control register. PAL_VP_RESTORE configures the logical processor to run the specified virtual processor by loading implementation-specific virtual processor context from the VPD, and returns control back to the VMM.

This procedure performs an implicit PAL_VPS_SYNC_WRITE; there is no need for the VMM to invoke PAL_VPS_SYNC_WRITE unless the VPD values are modified before resuming the virtual processor. After the procedure, the caller is responsible for restoring all of the architectural state before resuming to the new virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

Upon completion of this procedure, the IVA-based interruptions will be delivered to the host IVT associated with this virtual processor.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 for details.

PAL_VP_SAVE – PAL Save Virtual Processor (271)

Purpose: Saves virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_SAVE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_SAVE procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_VP_SAVE performs an implementation-specific save operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR).

This procedure performs an implicit PAL_VPS_SYNC_READ; there is no need for the VMM to invoke PAL_VPS_SYNC_READ to synchronize the implementation-specific control resources before this procedure.

Upon completion of this procedure, the IVA-based interruptions will continue to be delivered to the host IVT associated with this virtual processor. After this procedure, the VMM can setup the IVA control register to use a different host IVT.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, "Processor Virtualization" on page 2:44 and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 for details.

PAL_VP_TERMINATE – PAL Terminate Virtual Processor (272)

Purpose: Terminates operation for the specified virtual processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_TERMINATE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
iva	Optional 64-bit host virtual pointer to the IVT when this procedure is done
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_TERMINATE procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: Terminates operation of the virtual processor specified by *vpd* on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR). See Section 11.10.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:359 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. All resources allocated for the execution of the virtual machine are freed.

Upon successful execution of PAL_VP_TERMINATE procedure and if the *iva* parameter is non-zero, the IVA control register will contain the value from the *iva* parameter.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:446 for details.

11.11 PAL Virtualization Services

In order to support efficient handling of interruptions when PSR.vm was 1, a set of PAL virtualization services is defined to allow certain high-frequency PAL functions to be performed in a low-latency and low-overhead manner.

Upon successful completion of PAL_VP_INIT_ENV, the virtual base address of the PAL virtualization services (VSA) is returned to the VMM. VMM can invoke PAL services by branching to the defined offsets from the virtual base address. See Table 11-120 for the defined services. See Section 11.11, "PAL Virtualization Services" on page 2:486 for details on PAL virtualization services.

These PAL virtualization services will only make references to the PAL virtual environment buffer. The VMM is required to maintain the ITR and DTR translations of the PAL virtual environment buffer during any PAL virtualization service calls.

Table 11-120. PAL Virtualization Services

Offset	PAL Service
0x0000	PAL_VPS_RESUME_NORMAL
0x0400	PAL_VPS_RESUME_HANDLER
0x0800	PAL_VPS_SYNC_READ
0x0c00	PAL_VPS_SYNC_WRITE
0x1000	PAL_VPS_SET_PENDING_INTERRUPT
0x1400	PAL_VPS_THASH
0x1800	PAL_VPS_TTAG
0x1c00	PAL_VPS_RESTORE
0x2000	PAL_VPS_SAVE
All other offsets	Reserved

11.11.1 PAL Virtualization Service Invocation Convention

This section describes the required parameters applicable to all PAL Virtualization Services. Additional parameters are listed in the description section of specific PAL Virtualization Services. Architectural state not listed in this section is managed by the VMM and can contain both VMM and/or virtual processor state. The architectural state not listed is unchanged by PAL virtualization services.

The state of the processor on handing off to any PAL Virtualization Service is:

- GR24-31: Parameters for PAL virtualization services.
- BRs:
  - BR0: Scratch, the VMM will use BR0 to specify the 64-bit host virtual address of the PAL Virtualization Service being invoked.
- Predicates: The predicates are preserved by the PAL virtualization services.
- PSR State (see Table 11-121 for details):
  - PSR.be, i, cpl, is, ss, db, tb, vm must be 0.
  - PSR.dt, rt and it must be 1.
  - All other values are don’t cares.

Table 11-121. State Requirements for PSR for PAL Virtualization Services

PSR Bit	Description	Value
be	big-endian memory access enable	-[a]
up	user performance monitor enable	-
ac	alignment check	-
mfl	floating-point registers f2-f31 written	-
mfh	floating-point registers f32-f127 written	-
ic	interruption state collection enable	0[b] -[c]
i	interrupt enable	0
pk	protection key validation enable	-
dt	data address translation enable	1
dfl	disabled FP register f2 to f31	-
dfh	disabled FP register f32 to f127	-
sp	secure performance monitors	-
pp	privileged performance monitor enable	-
di	disable ISA transition	-
si	secure interval timer	-
db	debug breakpoint fault enable	0
lp	lower-privilege transfer trap enable	-
tb	taken branch trap enable	0
rt	register stack translation enable	1
cpl	current privilege level	0
is	instruction set	0
mc	machine check abort mask	-
it	instruction address translation enable	1
id	instruction debug fault disable	-
da	data access and dirty-bit fault disable	-
dd	data debug fault disable	-
ss	single step trap enable	0
ri	restart instruction	-
ed	exception deferral	-
bn	register bank	-[d] 0[e]
ia	instruction access-bit fault disable	-
vm	processor virtualization	0

a. PAL services can be called with PSR.be bit equal to 0 or 1. The behavior is undefined if PSR.be setting does not match the *be* parameter during PAL_VP_INIT_ENV. See "PAL_VP_INIT_ENV – PAL Initialize Virtual Environment (268)" on page 2:478 for details.

b. Most PAL services are invoked with PSR.ic equal to 0.

c. Specific PAL services can be invoked with PSR.ic equal to 1 or 0. See the description of specific PAL services for details.
d. Most PAL services can be invoked with PSR.bn equal to 1 or 0.
e. Specific PAL services must be invoked with PSR.bn equal to 0. See the description of specific PAL services for details.

11.11.2 PAL Virtualization Service Specifications

The following pages provide detailed interface specifications for each of the PAL Virtualization Services.

PAL_VPS_RESUME_NORMAL – Resume Virtual Processor Normal (0x0000)

Purpose: Resumes the current virtual processor. This service is used when vpsr.ic is 1. This service can also be used independent of the state of vpsr.ic if all virtualization accelerations and disables are disabled.

Arguments:

Argument	Description
GR24	VBR0
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns: PAL_VPS_RESUME_NORMAL does not return to the VMM.

Description: On interruptions or intercepts, PAL_VPS_RESUME_NORMAL allows the VMM to resume the same virtual processor where the vpsr.ic is 1. PAL_VP_RESTORE can be used to restore the state of a different virtual processor.

The VMM specifies the VBR0 of the virtual processor in GR24 and the 64-bit virtual pointer to the VPD in GR25.

The VMM is responsible for setting up all the required virtual processor state in the architectural registers as well as in the VPD prior to invoking this service. See Table 11-122, "Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER" on page 2:489 for details.

PAL_VPS_RESUME_NORMAL must be called with PSR.bn equal to 0.

If all virtualization accelerations and disables are disabled, PAL_VPS_RESUME_NORMAL can also be used to resume to the guest independent on the state of vpsr.ic.

Table 11-122. Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER

Resource	Description
Bank 1 GRs	Contains state of bank 0/1 GRs of the virtual processor (depends on vpsr.bn.)
FRs	Contains floating-point register state of the virtual processor.
Predicate Register	Contains the predicates of the virtual processor.
Branch Registers	BR1-BR7 contains the state of the virtual processor. BR0 of the virtual processor resides in bank 0 GR24.
Application Registers	Contains application register state of the virtual processor.
Interval Timer Offset Register[a]	If guest MOV-from-AR.ITC optimization is enabled, this register contains an offset, programmed by the VMM, to ensure that guest reads of ITC get the proper value.
Interruption Control Registers	IIP, IPSR and IFS contains the IP, PSR and CFM of the virtual processor. See Table 11-123 for the PSR settings for the execution of the virtual processor. The rest of the interruption control registers are don't cares. For PAL_VPS_RESUME_HANDLER, the virtual interruption control registers are specified in the VPD. See Section 11.7.4, "Virtualization Optimizations" on page 2:335 for synchronization of VPD resources before resuming the virtual processor.

Table 11-122. Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER

Resource	Description
External Interrupt Control Registers	The external interrupt control registers contain the state of the virtual processor if d_extint in Virtualization Disable Control (*vdc*) is 1. Otherwise the external interrupt control registers are virtualized by the VMM and contain VMM state.
Data/Instruction Breakpoint Registers	The data/instruction breakpoint registers contain the state of the virtual processor if d_ibr_dbr in Virtualization Disable Control (*vdc*) is 1. Otherwise the data/instruction breakpoint registers are virtualized by the VMM and contain VMM state.
Performance Monitor Configuration Registers	The performance monitor configuration registers contain the state of the virtual processor if d_pmc in Virtualization Disable Control (*vdc*) is 1. Otherwise the performance monitor configuration registers are virtualized by the VMM and contain VMM state.
Performance Monitor Data Registers	Contain the state of the virtual processor.

a. Interval Timer Offset register is not supported on all processor implementations. See Section 3.3.4.4, “Interval Timer Offset (ITO – CR4)” on page 2:34 for details.

Table 11-123. Processor Status Register Settings for Virtual Processor Execution

Field	Bits	Description
User Mask = PSR{5:0}		
rv	0	Reserved
be	1	Contain user mask of the virtual processor.
up	2	
ac	3	
mfl	4	
mfh	5	
System Mask = PSR{23:0}		
ic	13	Must be 1.
i	14	VMM-specific.
pk	15	
rv	12:6, 16	Reserved
dt	17	Must be 1.
dfl	18	VMM-specific.
dfh	19	
sp	20	
pp	21	
di	22	
si	23	
PSR.l = PSR{31:0}		
db	24	VMM-specific.
lp	25	Contains the lp bit of the virtual processor.
tb	26	Contains the tb bit of the virtual processor.
rt	27	Must be 1.
rv	31:28	Reserved
PSR{63:0}		

Table 11-123. Processor Status Register Settings for Virtual Processor Execution (Continued)

Field	Bits	Description
cpl	33:32	Contains the cpl field of the virtual processor.
is	34	VMM-specific.
mc	35	VMM-specific.
it	36	Must be 1.
id	37	VMM-specific.
da	38	VMM-specific.
dd	39	VMM-specific.
ss	40	VMM-specific.
ri	42:41	Contains the ri field of the virtual processor.
ed	43	Contains the ed bit of the virtual processor.
bn	44	Must be 1.
ia	45	VMM-specific.
vm	46	Must be 1.
rv	63:47	Reserved

PAL_VPS_RESUME_NORMAL performs the following actions:

- Perform any implementation-specific setup to run a virtual processor.
- Re-enable performance counters if the value of the *fr_pmc* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.
- Resume the virtual processor.

PAL_VPS_RESUME_HANDLER – Resume Virtual Processor Handler (0x0400)

Purpose: Resumes the current virtual processor. This service is used when vpsr.ic is 0.

Arguments:

Argument	Description
GR24	VBR0
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Virtualization Acceleration Control (*vac*) field from the VPD specified in GR25 and CFLE setting at the target instruction.
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns: PAL_VPS_RESUME_HANDLER does not return to the VMM.

Description: On interruptions or intercepts, PAL_VPS_RESUME_HANDLER allows the VMM to resume to the same virtual processor where the vpsr.ic is 0[1].

GR24 specifies the BR0 of the virtual processor; GR25 specifies the 64-bit virtual pointer to the VPD; GR26 specifies the *vac* field of the VPD argument specified in GR25; bit 63 of GR26 specifies the value of CFLE setting at the target instruction. Behavior is undefined if the *vac* in GR26 does not match the *vac* field in the VPD argument specified in GR25.

The VMM is responsible for setting up all the required virtual processor state in the architectural registers as well as in the VPD prior to invoking this service. See Table 11-122, "Virtual Processor Settings in Architectural Resources for PAL_VPS_RESUME_NORMAL and PAL_VPS_RESUME_HANDLER" on page 2:489 for details.

PAL_VPS_RESUME_HANDLER must be called with PSR.bn equal to 0.

PAL_VPS_RESUME_HANDLER performs the following actions:

- Perform any implementation-specific setup to run a virtual processor.
- Re-enable performance counters if the value of the *fr_pmc* field in the *config_options* parameter passed to PAL_VP_INIT_ENV was 1.
- Resume the virtual processor.

[1] PAL_VP_RESTORE can be used to restore the state of a different virtual processor.

PAL_VPS_SYNC_READ – Synchronize VPD State for Reads (0x0800)

Purpose: Synchronize VPD with the latest implementation-specific virtual architectural state.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: On processor implementations that support virtualization accelerations, implementation-specific control resources can be provided to enhance performance of virtual processors. When a specific acceleration is enabled, after interruptions and intercepts which occur when PSR.vm was 1, the VMM must invoke this service to synchronize the related resources before reading the value from the VPD. For the accelerations that are disabled, the corresponding resources in the VPD are unchanged.

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 11.7.4.2, "Virtualization Accelerations" on page 2:337.

PAL_VPS_SYNC_READ performs the following actions:

- Copy implementation-specific control resources of the enabled accelerations into VPD.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_SYNC_WRITE – Synchronize VPD State for Writes (0x0c00)

Purpose: Synchronize the implementation-specific virtual architectural state with VPD.

Arguments:

Argument	Description
GR24	64-bit host virtual return address.
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD.)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: On processor implementations that support virtualization accelerations, implementation-specific control resources can be provided to enhance performance of virtual processors. When a specific acceleration is enabled, the VMM must invoke this service to synchronize the related resources after modifying the value in the VPD and before resuming the virtual processor. For the accelerations that are disabled, the corresponding resources in the VPD are ignored.

The synchronization requirements of the related resources for each acceleration are described in the corresponding sections for each acceleration in Section 11.7.4.2, "Virtualization Accelerations" on page 2:337.

PAL_VPS_SYNC_WRITE performs the following actions:

- Copy values of the enabled accelerations in the VPD into implementation-specific control resources.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_SET_PENDING_INTERRUPT – Register Highest Priority Pending Interrupt (0x1000)

Purpose: Register highest priority pending interrupt of the running virtual processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Reserved
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_SET_PENDING_INTERRUPT allows the VMM to register the highest priority pending interrupt for the virtual processor. The virtual highest priority pending interrupt is specified in the vhpi field in the VPD. See Table 11-124, “vhpi – Virtual Highest Priority Pending Interrupt” on page 2:495 for details.

PAL_VPS_SET_PENDING_INTERRUPT can be called with PSR.ic equal to 1 or 0.

Table 11-124. *vhpi* – Virtual Highest Priority Pending Interrupt

Value	Description
0	Nothing pending.
1	Class 1 interrupt pending.
2	Class 2 interrupt pending.
3	Class 3 interrupt pending.
4	Class 4 interrupt pending.
5	Class 5 interrupt pending.
6	Class 6 interrupt pending.
7	Class 7 interrupt pending.
8	Class 8 interrupt pending.
9	Class 9 interrupt pending.
10	Class 10 interrupt pending.
11	Class 11 interrupt pending.
12	Class 12 interrupt pending.
13	Class 13 interrupt pending.
14	Class 14 interrupt pending.
15	Class 15 interrupt pending.
16	ExtINT pending.
17-31	Reserved.
32	NMI pending.
33+	Reserved.

PAL_VPS_SET_PENDING_INTERRUPT performs the following actions:

- Copy the virtual highest priority pending interrupt from the VPD into implementation-specific resources.
- Return to VMM by an indirect branch specified in the GR24 parameter.

PAL_VPS_THASH – Compute Long Format VHPT Entry Address (0x1400)

Purpose: Compute a long format VHPT entry address.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit virtual address used to compute the hash entry address
GR26	Region register value used to compute the hash entry address
GR27	Virtual PTA
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	64-bit VHPT entry address

Description: PAL_VPS_THASH computes a long format Virtual Hashed Page Table (VHPT) entry address based on the input arguments and the result is returned in GR31. The format of the region register parameter (GR26) is defined in Section 4.1.2, "Region Registers (RR)" on page 2:58, the ve field is ignored by the service. The format of the Virtual PTA parameter (GR27) is defined in Section 3.3.4.6, "Page Table Address (PTA – CR8)" on page 2:35, the vf field is ignored by the service.

PAL_VPS_THASH returns the same long format VHPT entry address given the same input arguments across different implementations. The long format VHPT entry address returned may not be the same as the long format VHPT entry address generated by the `thash` instruction of the processor.

PAL_VPS_THASH can be called with PSR.ic equal to 1 or 0.

PAL_VPS_TTAG – Compute Translated Hashed Entry Tag (0x1800)

Purpose: Compute the long format translated hashed entry tag.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit virtual address used to compute the hash entry tag
GR26	Region register value used to compute the hash entry tag
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	64-bit VHPT entry tag

Description: PAL_VPS_TTAG computes the tag value of the long format Virtual Hashed Page Table (VHPT) based on the input arguments and the result is returned in GR31. The format of the region register parameter (GR26) is defined in Section 4.1.2, "Region Registers (RR)" on page 2:58, the ve field is ignored by the service.

PAL_VPS_TTAG returns the same tag value given the same input arguments across different implementations. The tag value returned may not be the same as the tag value generated by the `ttag` instruction of the processor.

PAL_VPS_TTAG can be called with PSR.ic equal to 1 or 0.

PAL_VPS_RESTORE – Fast Restore Virtual Processor State (0x1c00)

Purpose: Performs an implementation-specific light-weight restore operation for the specified VPD on the logical processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Skip implicit synchronization
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_RESTORE performs an implementation-specific light-weight restore operation of the virtual processor specified by the VPD parameter (GR25) on the logical processor. The host virtual to host physical translation of the 64K region specified by the VPD parameter (GR25) and the PAL virtual environment buffer must be mapped by instruction and data translation registers (TR). The instruction and data translation must be maintained until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE and a different host IVT is set up by the VMM by writing to the IVA control register. PAL_VPS_RESTORE configures the logical processor to run the specified virtual processor by loading the minimal implementation-specific virtual processor context from the VPD, and returns control back to the VMM.

If GR26 is zero, this service performs an implicit PAL_VPS_SYNC_WRITE; there is no need for the VMM to invoke PAL_VPS_SYNC_WRITE to synchronize the implementation-specific control resources before this service. If GR26 is one (0x1), no implicit synchronization will be performed by this service.

Upon completion of this service, the IVA-based interruptions will be delivered to the host IVT associated with this virtual processor.

This service does not restore any PAL procedure implementation-specific state[1]. The caller of this service is responsible to manage the difference in settings for the PAL procedures between the VMM and virtual processors.

[1] PAL_VP_RESTORE can be used to restore PAL procedure implementation-specific state. See "PAL_VP_RESTORE – PAL Restore Virtual Processor (270)" on page 2:483 for details.

PAL_VPS_SAVE – Fast Save Virtual Processor State (0x2000)

Purpose: Performs an implementation-specific light-weight save operation for the specified VPD on the logical processor.

Arguments:

Argument	Description
GR24	64-bit host virtual return address
GR25	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
GR26	Skip implicit synchronization
GR27	Reserved
GR28	Reserved
GR29	Reserved
GR30	Reserved
GR31	Reserved

Returns:

Return Value	Description
GR24	Scratch
GR25	Scratch
GR26	Scratch
GR27	Scratch
GR28	Scratch
GR29	Scratch
GR30	Scratch
GR31	Scratch

Description: PAL_VPS_SAVE performs an implementation-specific light-weight save operation of the virtual processor specified by the VPD parameter (GR25) on the logical processor. The host virtual to host physical translation of the 64K region specified by the VPD parameter (GR25) must be mapped by instruction and data translation registers (TR).

If GR26 is zero, this service performs an implicit PAL_VPS_SYNC_READ; there is no need for the VMM to invoke PAL_VPS_SYNC_READ to synchronize the implementation-specific control resources before this service. If GR26 is one (0x1), no implicit synchronization will be performed by this service.

Upon completion of this service, the IVA-based interruptions will continue to be delivered to the host IVT associated with this virtual processor. After this service, the VMM can setup the IVA control register to use a different host IVT.

This service does not save any PAL procedure implementation-specific state[1]. The caller of this service is responsible to manage the difference in settings for the PAL procedures between the VMM and virtual processors.

§

[1] PAL_VP_SAVE can be used to save PAL procedure implementation-specific state. See “PAL_VP_SAVE – PAL Save Virtual Processor (271)” on page 2:484 for details.

Part II: System Programmer’s Guide

About the System Programmer’s Guide 1

Part II: System Programmer’s Guide is intended as a companion section to the information presented in Part I:, “System Architecture Guide”. While *Part I* provides a crisp and concise architectural definition of the Itanium instruction set, *Part II* provides insight into programming and usage models of the Itanium system architecture. This section emphasizes how the various architecture features fit together and explains how they contribute to high performance system software.

The intended audience for this section is system programmers who would like to better understand the Itanium system architecture. The goal of this document is to:

- Familiarize system programmers with Itanium system architecture principles and usage models.
- Provide recommendations, code examples, and performance guidelines.

This section does not re-define the Itanium instruction set. Please refer to Part I:, “System Architecture Guide” as the authoritative definition of the system architecture.

The reader is expected to be familiar with the contents of *Part I* and is expected to be familiar with modern virtual memory and multiprocessing concepts. Furthermore, this document is platform architecture neutral (i.e. no assumptions are made about platform architecture capabilities, such as busses, chipsets, or I/O devices).

1.1 Overview of the System Programmer’s Guide

The Itanium architecture provides numerous performance enhancing features of interest to the system programmer. Many of these instruction set features focus on reducing overhead in common situations. The chapters outlined below discuss different aspects of the Itanium system architecture.

Chapter 2, “MP Coherence and Synchronization” describes Itanium architecture-based multiprocessing synchronization primitives and the Itanium memory ordering model. This chapter also discusses programming rules for self- and cross-modifying code. This chapter is useful for application and system programmers who write multi-threaded code.

Chapter 3, “Interruptions and Serialization” discusses how the Itanium architecture, despite its explicitly parallel instruction execution semantics, provides the system programmer with a precise interruption model. This chapter describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken. This chapter introduces the interrupt vector table and describes how low-level kernel code is expected to transfer control to higher level operating system code written in a high-level programming language. This chapter is useful for operating system and firmware programmers.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents. In addition to spilling and filling a register’s data value, the Itanium architecture also requires software to preserve control and data speculative state associated with that register, i.e. its NaT bit and ALAT state. This chapter also discusses system architecture mechanisms that allow an operating system to significantly reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches. These optimizations improve the performance of an Itanium architecture-based operating system by reducing the amount of required memory traffic. This chapter is useful for operating system programmers.

Chapter 5, “Memory Management” introduces various memory management strategies in the Itanium architecture: region register model, protection keys, and the virtual hash page table usage models are described. This chapter is of interest to virtual memory management software developers.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation. This chapter describes various speculation software models and their associated operating system implications. This chapter is of interest to operating system developers and compiler writers.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support. This chapter is useful for operating system developers.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the Itanium architecture-based software stack provides complete IEEE-754 compliance. This includes a discussion of the floating-point software assist firmware, the FP SWA EFI driver. This chapter also describes how Itanium architecture-based operating systems are expected to support IEEE floating-point exception filters. This chapter is useful for operating system developers and floating-point numerics experts.

Chapter 9, “IA-32 Application Support” outlines how software needs to perform instruction set transitions, and what low-level kernel handlers are required in an Itanium architecture-based operating system to support IA-32 applications. This chapter is useful for operating system developers.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software. Basic interrupt prioritization, masking, and harvesting capabilities are discussed in this chapter. This chapter is of interest to operating system developers and to device driver writers.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform considerations and support for the existing IA-32 I/O port space platform infrastructure. This chapter is of interest to operating system developers and to device driver writers.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of operating system support is needed from Itanium architecture-based operating systems. This chapter is of interest to operating system and performance tool developers.

Chapter 13, “Firmware Overview” introduces the firmware model and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization and operating system boot. This chapter also discusses how firmware layers and the operating system work together to provide error detection, error logging, as well as fault containment capabilities. This chapter is of interest to platform firmware and operating system developers.

1.2 Related Documents

The following documents are referred to fairly often in this document. For more details on software conventions and platform firmware, please consult these manuals (available at http://developer.intel.com).

[SWC] ***Intel® Itanium® Software Conventions and Runtime Architecture Guide***

[UEFI] *Unified Extensible Firmware Interface Specification*

[SAL] ***Intel® Itanium® Processor Family System Abstraction Layer Specification***

§

MP Coherence and Synchronization 2

This chapter describes how to enforce an ordering of memory operations, how to update code images, and presents examples of several simple multiprocessor synchronization primitives on a processor based on the Itanium architecture. These topics are relevant to anyone who writes either user- or system-level software for multiprocessor systems based on the Itanium architecture.

The chapter begins with a brief overview of Itanium memory access instructions intended to summarize the behaviors that are relevant to later discussions in the chapter. Next, this chapter presents the Itanium memory ordering model and compares it to a sequentially-consistent ordering model. It then explores versions of several common synchronization primitives. This chapter closes by describing how to correctly update code images to implement self-modifying code, cross-modifying code, and paging of code using programmed I/O.

2.1 An Overview of Intel® Itanium® Memory Access Instructions

The Itanium architecture provides load, store, and semaphore instructions to access memory. In addition, it also provides a memory fence instruction to enforce further ordering relationships between memory accesses. As Section 4.4.7, "Memory Access Ordering" on page 1:73 describes, memory operations in the Itanium architecture come with one of four semantics: unordered, acquire, release, or fence. Section 2.2 on page 2:510 describes how the memory ordering model uses these semantics to indicate how memory operations can be ordered with respect to each other.

Section 2.1.1 defines the four memory operation semantics. Section 2.2, Section 2.3, and Section 2.4 present brief outlines of load and store, semaphore, and memory fence instructions in the Itanium architecture. Refer to Chapter 2, "Instruction Reference" for more information on the behavior and capabilities of these instructions.

2.1.1 Memory Ordering of Cacheable Memory References

The Itanium architecture has a relaxed memory ordering model which provides unordered memory opcodes, explicitly ordered memory opcodes, and a fencing operation that software can use to implement stronger ordering. Each memory operation establishes an ordering relationship with other operations through one of four semantics:

- *Unordered* semantics imply that the instruction is made visible in any order with respect to other orderable instructions.
- *Acquire* semantics imply that the instruction is made visible prior to all subsequent orderable instructions.
- *Release* semantics imply that the instruction is made visible after all prior orderable instructions.

- *Fence* semantics combine acquire and release semantics (i.e. the instruction is made visible after all prior orderable instructions and before all subsequent orderable instructions).

In the above definitions "prior" and "subsequent" refer to the program-specified order. An "orderable instruction" is an instruction that the memory ordering model can use to establish ordering relationships[1]. The term "visible" refers to all architecturally-visible (from the standpoint of multiprocessor coherency) effects of performing an instruction. Specifically,

- Accesses to uncacheable or write-coalescing memory regions are visible when they reach the processor bus.
- Loads from cacheable memory regions are visible when they hit a non-programmer-visible structure such as a cache or store buffer.
- Stores to cacheable memory regions are visible when they enter a snooped (in a multiprocessor coherency sense) structure.

Memory access instructions typically have an ordered and an unordered form (i.e. a form with unordered semantics and a form with either acquire, release, or fence semantics). The Itanium architecture does not provide all possible combinations of instructions and ordering semantics. For example, the Itanium instruction set does not contain a store with fence semantics.

Section 4.4.7, "Memory Access Ordering" on page 1:73 and Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82 discuss ordering, orderable instructions, and visibility in greater depth.

Section 2.2 on page 2:510 describes how the ordering semantics affect the Itanium memory ordering model.

2.1.2 Loads and Stores

In the Itanium architecture, a load instruction has either unordered or acquire semantics while a store instruction has either unordered or release semantics. By using acquire loads (`ld.acq`) and release stores (`st.rel`), the memory reference stream of an Itanium architecture-based program can be made to operate according to the IA-32 ordering model. The Itanium architecture uses this behavior to provide IA-32 compatibility. That is, an Itanium acquire load is equivalent to an IA-32 load and an Itanium release store is equivalent to an IA-32 store, from a memory ordering perspective.

Loads can be either speculative or non-speculative. The speculative forms (`ld.s`, `ld.sa`, and `ld.a`) support control and data speculation.

2.1.3 Semaphores

The Itanium architecture provides a set of three semaphore instructions: exchange (`xchg`), compare and exchange (`cmpxchg`), and fetch and add (`fetchadd`). Both `cmpxchg` and `fetchadd` may have either acquire or release semantics depending on the

[1] The ordering semantics of an instruction *do not* imply the orderability of the instruction. Specifically, unordered ordering semantics alone *do not* make an instruction unorderable; there are orderable instructions with each of the four ordering semantics.

specific opcode chosen. The xchg instruction always has acquire semantics. These instructions read a value from memory, modify this value using an instruction-specific operation, and then write the modified value back to memory. The read-modify-write sequence is atomic by definition.

2.1.3.1 Considerations for using Semaphores

The memory location on which a semaphore instruction operates on must obey two constraints. First, the location must be cacheable (the fetchadd instruction is an exception to this rule; it may also operate on exported uncacheable locations, UCE). Thus, with the exception of fetchadd to UCE locations, the Itanium architecture does not support semaphores in uncacheable memory. Second, the location must be naturally-aligned to the size of the semaphore access. If either of these two constraints are not met, the processor generates a fault.

The exported uncacheable memory attribute, UCE, allows a processor based on the Itanium architecture to export fetch and add operations to the platform. A processor that does not support exported fetchadd will fault when executing a fetchadd to a UCE memory location. If the processor supports exported fetchadd but the platform does not, the behavior is undefined when executing a fetchadd to a UCE memory location.

Sharing locks between IA-32 and Itanium architecture-based code does work with the following restrictions:

- Itanium architecture-based code can only manipulate an IA-32 semaphore if the IA-32 semaphore is aligned.
- Itanium architecture-based code can only manipulate an IA-32 semaphore if the IA-32 semaphore is allocated in write-back cacheable memory.

An Itanium architecture-based operating system can emulate IA-32 uncacheable or misaligned semaphores by using the technique described in the next section.

2.1.3.2 Behavior of Uncacheable and Misaligned Semaphores

A processor based on the Itanium architecture raises an Unsupported Data Reference fault if it executes a semaphore that accesses a location with a memory attribute that the semaphore does not support.

If the alignment requirement for Itanium architecture-based semaphores is not met, a processor based on the Itanium architecture raises an Unaligned Data Reference fault. This fault is taken regardless of the setting of the user mask alignment checking bit, UM.ac.

The DCR.lc bit controls how the processor behaves when executing an atomic IA-32 memory reference under an external bus lock. When the DCR.lc bit (see Section 3.3.4.1, “Default Control Register (DCR – CR0)”) is 1 and an IA-32 atomic memory reference requires a non-cacheable or misaligned read-modify-write operation, an IA_32_Intercept(Lock) fault is raised. Such memory references require an external bus lock to execute correctly. To preserve LOCK pin functionality, an Itanium architecture-based operating system can virtualize the bus lock by implementing a shared cacheable global LOCK variable.

To support existing IA-32 atomic read-modify-write operations that require the `LOCK` pin, an Itanium architecture-based operating system can use the DCR.lc bit to intercept all external IA-32 read-modify-write operations. Then, the IA_32_Intercept(Lock) handler can emulate these operations by first acquiring a cacheable virtualized `LOCK` variable, then performing the required memory operations non-atomically, and then releasing the virtualized `LOCK` variable. This emulation allows the read-modify-write sequence to appear atomic to other processors that use the semaphore.

2.1.4 Memory Fences

The memory fence instruction (`mf`) is the only instruction in the Itanium instruction set with fence semantics. This instruction serializes the set of memory accesses before the memory fence in program order with respect to the set of memory accesses that follow the fence in program order.

2.2 Memory Ordering in the Intel® Itanium® Architecture

Understanding a system's memory ordering model is key to writing either user- or system-level multiprocessor software that uses shared memory to communicate between processes and also that executes correctly on a shared-memory multiprocessor system. For a general introduction to memory ordering models, see Adve and Gharachorloo [AG95].

Four factors determine how a processor or system based on the Itanium architecture orders a group of memory operations with respect to each other:

- *Data dependencies* define the relationship between operations from the same processor that have register or memory dependencies on the same address[1]. This relationship need only be honored by the local processor (i.e. the processor that executes the operations).
- The *memory ordering semantics* define the relationship between memory operations from a particular processor that reference different addresses. For cacheable references, this relationship is honored by *all* observers in the coherence domain.
- Aligned *release stores* and *semaphore operations* (both require and release forms) become visible to all observers in the coherence domain in a single total order except each processor may observe its own release stores (via loads or acquire loads) prior to their being observed globally[2].
- Non-programmer-visible state, such as *store buffers, processor caches,* or any logically-equivalent structure, may satisfy read requests from loads or acquire loads on the local processor before the data in the structure is made globally visible to other observers.

[1] That is, A precedes B in program order and A produces a value that B consumes. This relationship is transitive.

[2] Consequently, each such operation appears to become visible to each observer in the coherence domain at the same time, with the exception that a release store can become visible to the storing processor before others.

In the Itanium architecture, dependencies between operations by a processor have implications for the ordering of those operations at that processor. The discussion in Section 2.2.1.6 on page 2:515 and Section 2.2.1.7 on page 2:516 explores this issue in greater depth.

The following sections examine the Itanium ordering model in detail. Section 2.2.1 presents several memory ordering executions to illustrate important behaviors of the model. Section 2.2.2 discusses how memory attributes and the ordering model interact. Finally, Section 2.2.3 describes how the Itanium memory ordering model compares with other memory ordering models.

2.2.1 Memory Ordering Executions

Multiprocessor software that uses shared memory to communicate between processes often makes assumptions about the order in which other agents in the system will observe memory accesses. As Section 2.1.1 on page 2:507 describes, the Itanium architecture provides a rich set of ordering semantics that allows software to express different ordering constraints on a memory operation, such as a load. Writing correct multiprocessor software requires that the programmer (or compiler) select the ordering semantic appropriate to enforce the expected behavior.

For example, an algorithm that requires two store operations A and B become visible to other processors in the order {A, B} will use stores with different ordering semantics than an algorithm that does not require any particular ordering of A and B. Although it is always safe to enforce stricter ordering constraints than an algorithm requires, doing so may lead to lower performance. If the ordering of memory operations is not important, software should use unordered ordering semantics whenever possible for best possible performance.

This section presents multiprocessor executions to demonstrate the ordering behaviors that the Itanium architecture allows and to contrast the Itanium ordering model with other ordering models. The executions consist of sequences of memory accesses that execute on two or more processors and highlight outcomes that the Itanium memory ordering model either allows or disallows once all accesses on all processors complete. A programmer can use these executions as a guide to determine which Itanium memory ordering semantics are appropriate to ensure a particular visibility order of memory accesses.

Section 2.2.1.1 presents the assumptions and notational conventions that the upcoming discussions use to examine the executions. The remaining eleven sections each explore a different facet of the Itanium ordering model:

- Relaxed ordering of unordered memory operations (Section 2.2.1.2).
- Using acquire and release semantics to order operations (Section 2.2.1.3).
- Loads may pass stores (Section 2.2.1.4) and how to prevent this behavior (Section 2.2.1.5).
- When dependencies do or do not establish memory ordering (Section 2.2.1.6 and Section 2.2.1.7).
- Satisfying loads from store buffers (Section 2.2.1.8) and how to prevent this behavior (Section 2.2.1.9).
- Semaphore operations and local bypass (Section 2.2.1.10).

- Global visibility order of memory operations (Section 2.2.1.11 and Section 2.2.1.12).

This presentation is organized to begin with simple behaviors and move to increasingly complex behaviors.

2.2.1.1 Assumptions and Notation

The discussions of the multiprocessor executions in the upcoming sections adopt two main notational conventions.

First, the memory accesses in the executions in this document are written using a pseudo-Itanium architecture-based assembly language that allows a store to write an immediate operand to memory. All memory locations are cacheable and aligned. Unless stated otherwise, memory locations do not overlap. Initially, all registers and memory locations contain zero.

Second, given two different memory operations X and Y, $X \gg Y$ specifies that X precedes Y in program order and $X \rightarrow Y$ indicates that X is visible if Y is visible (i.e. X becomes visible before Y).

Using this notation, Figure 2-1 expresses the Itanium ordering semantics from Section 2.1.1, "Memory Ordering of Cacheable Memory References" on page 2:507 and also Section 4.4.7, "Memory Access Ordering" on page 1:73. There are no implications regarding the ordering of the visibility for the following pairs of operations: a release followed by an unordered operation; a release followed by an acquire; an unordered operation followed by another; or an unordered operation followed by an acquire.

Figure 2-1. Intel® Itanium® Ordering Semantics

$$\text{Acquire} \gg X \Rightarrow \text{Acquire} \rightarrow X$$
$$X \gg \text{Release} \Rightarrow X \rightarrow \text{Release}$$
$$X \gg \text{Fence} \Rightarrow X \rightarrow \text{Fence}$$
$$\text{Fence} \gg Y \Rightarrow \text{Fence} \rightarrow Y$$

In Figure 2-1, "Acquire", "Release", and "Fence" represent an orderable instruction with the corresponding memory ordering semantics whereas "X" and "Y" indicate any orderable instruction.

2.2.1.2 The Intel® Itanium® Architecture Provides a Relaxed Ordering Model

The Itanium memory ordering model is a relaxed model. As a result, the Itanium architecture permits any outcome when executing the code shown in Table 2-1.

Table 2-1. Intel® Itanium® Architecture Provides a Relaxed Ordering Model

Processor #0	Processor #1

`st [x] = 1 // M1`
`st [y] = 1 // M2`
	`ld r1 = [y] // M3`
	`ld r2 = [x] // M4`

Outcomes: all are allowed

Because all of the operations in Table 2-1 are unordered, the Itanium memory ordering model does not place any constraints on the order in which a processor based on the Itanium architecture makes the operations visible.

Observing a particular value in r2, for example, does not allow any inferences to be made about the value of r1 because the pair of stores on Processor #0 may become visible in any order. Therefore, all outcomes are possible as the system may interleave M1, M2, M3, and M4 in any order without violating the memory ordering constraints.

2.2.1.3 Enforcing Basic Ordering

Using acquire and release ordering semantics enforces an ordering between both the Processor #0 operations M1 and M2 and the Processor #1 operations M3 and M4 from the Table 2-1 execution as shown in Table 2-1.

Table 2-2. Acquire and Release Semantics Order Intel® Itanium® Memory Operations

Processor #0	Processor #1
`st      [x] = 1      // M1`	`ld.acq  r1 = [y]     // M3`
`st.rel  [y] = 1      // M2`	`ld      r2 = [x]     // M4`

Outcome: only r1 = 1 and r2 = 0 is not allowed

The Itanium ordering model only disallows the outcome r1 = 1 and r2 = 0 in this execution. The release semantics on M2 and acquire semantics on M3 affect the following ordering constraints:

$$M1 \rightarrow M2$$
$$M3 \rightarrow M4$$

Given the code in Table 2-2, these two ordering constraints along with the assumption that the outcome is r1 = 1 and r2 = 0 together imply that:

$$r1 = 1 \Rightarrow M2 \rightarrow M3 \Rightarrow M1 \rightarrow M4 \text{ (because } M1 \rightarrow M2 \text{ and } M3 \rightarrow M4) \Rightarrow r2 = 1$$

This contradicts the postulated outcome r1 = 1 and r2 = 0 and thus the Itanium ordering model disallows the r1 = 1 and r2 = 0 outcome.

In operational terms, if Processor #1 observes M2, the release store to y (i.e. r1 is 1), it must have also observed M1, the unordered store to x (i.e. r2 is 1 as well), given the ordering constraints. Therefore, the Itanium ordering model must disallow the outcome r1 = 1 and r2 = 0 in this execution as this outcome violates these constraints.

Stronger ordering models that do not relax load-to-load and store-to-store ordering, such as sequential consistency, impose these same ordering constraints on M1, M2, M3, and M4 and therefore also do not allow the outcome r1 = 1 and r2 = 0.

2.2.1.4 Allow Loads to Pass Stores to Different Locations

The Itanium memory ordering model allows loads to pass stores as shown in the execution sequence in Table 2-3. Permitting this behavior can improve performance by allowing the processor to complete loads that follow a store that misses the cache.

The Itanium ordering semantics always allow a processor to make operations that follow a release visible before the release and to make operations that precede an acquire visible after the acquire.

Table 2-3. Loads May Pass Stores to Different Locations

Processor #0	Processor #1
`st.rel [x] = 1 // M1`	`st.rel [y] = 1 // M3`
`ld.acq r1 = [y] // M2`	`ld.acq r2 = [x] // M4`

Outcomes: all are allowed

Like the execution shown in Table 2-1, the Itanium memory ordering model does not place any constraints on the ordering of the operations on each processor in this execution either.

Therefore, for reasons similar to those given in Section 2.2.1.2 for the execution shown in Table 2-1, the Itanium memory ordering model allows any outcome in this execution as well. Further, the Itanium memory ordering model also allows all outcomes in similar executions that differ only in the ordering semantics of the load and store operations (e.g. those that replace M1 with an unordered store, etc.). There is no combination of legal ordering semantics on these operations (recall that the Itanium instruction set does not provide stores with acquire or fence semantics) that enforce either $M1 \rightarrow M2$ or $M3 \rightarrow M4$.

2.2.1.5 Preventing Loads from Passing Stores to Different Locations

The only way to prevent the loads from moving ahead of the stores in the Table 2-3 execution is to separate them with a memory fence as the execution in Table 2-4 illustrates.

Table 2-4. Loads May Not Pass Stores in the Presence of a Memory Fence

Processor #0	Processor #1
`st [x] = 1 // M1`	`st [y] = 1 // M4`
`mf // M2`	`mf // M5`
`ld r1 = [y] // M3`	`ld r2 = [x] // M6`

Outcome: only r1 = 0 and r2 = 0 is not allowed

The Itanium memory ordering model only disallows the outcome r1 = 0 and r2 = 0 in this execution. The memory fences on Processor #0 and Processor #1 (operations M2 and M5) force the load and store memory accesses to be made visible in program order; no re-ordering is permitted across the fence. Thus, the following ordering constraints must be met:

$$M1 \rightarrow M2 \rightarrow M3$$
$$M4 \rightarrow M5 \rightarrow M6$$

Given the code in Table 2-4, these two constraints along with the assumption that the outcome is r1 = 0 and r2 = 0 together imply that

$$r1 = 0 \Rightarrow M3 \rightarrow M4 \Rightarrow M3 \rightarrow M6 \text{ because } M4 \rightarrow M5 \rightarrow M6$$
$$r1= 0 \Rightarrow M1 \rightarrow M3 \text{ because } M1 \rightarrow M2 \rightarrow M3$$
$$M1 \rightarrow M3 \text{ and } M3 \rightarrow M6 \Rightarrow M1 \rightarrow M6 \Rightarrow r2 = 1$$

This contradicts the postulated outcome r1 = 0 and r2 = 0 and thus the Itanium memory ordering model disallows the r1 = 1 and r2 = 0 outcome. Specifically, if M3 reads 0, then M4, M5, and M6 may not yet be visible but M1 and M2 must be visible. Thus, when M6 becomes visible it must observe x = 1 because M1 is already visible.

2.2.1.6 Data Dependency Does Not Establish MP Ordering

The dependency rules define the relationship between memory operations that access the same address. Specifically, the Itanium architecture resolves read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies through memory in program order on the local processor. As Section 2.2 discusses, dependencies are fundamentally different from the ordering semantics even though both affect ordering relationships between groups of memory accesses.

The execution shown in Table 2-5 illustrates this difference.

Table 2-5. Dependencies Do Not Establish MP Ordering (1)

Processor #0			Processor #1		

`st`
`ld`
`st`
	`[x] = 1 ;;`
	`r1 = [x] ;;`
	`[y] = r1 ;;`
		`// M1`
		`// M2`
		`// M3`
			`ld.acq`
			`ld`
				`r2 = [y]`
				`r3 = [x]`
					`// M4`
					`// M5`

Outcomes: r1 = 1, r2 = 1, and r3 = 0 is allowed

The following discussion focuses on the outcome r1 = 1, r2 = 1, and r3 = 0. This outcome is allowed only because the Itanium architecture treats data dependencies and the ordering semantics differently.

The ordering semantics require $M4 \rightarrow M5$, but do not place any constraints on the relative order of operations M1, M2, or M3. Due to the register and memory dependencies between the instructions on Processor #0, these operations complete *in program order* on Processor #0 and also become *locally* visible in this order. However, the operations need *not* be made visible to remote processors in program order. In this outcome it appears to Processor #0 as if $M1 \rightarrow M3$ while to Processor #1 it appears that $M3 \rightarrow M1$. There are two things to note here. First, the behavior is another example of the local bypass behavior that Section 2.2.1.8 presents on page 2:518. Second, there are no dependencies *directly* between M1 and M3 that requires them to become globally visible in program order.

Note: All processors will observe the order established by a particular processor in case of a WAW memory dependency to the same location. For example, all processors in the coherence domain eventually see a value of 1 in location x in the following code:

```
st     [x] = 0        //  M1: set [x] to 0
st     [x] = 1        //  M2: set [x] to 1,
                      //  cannot move above M1 due to WAW
```

because there is a WAW memory dependency between from M2 to M1 and the Itanium architecture requires that the local processor resolves RAW, WAR, and WAW dependencies between its memory accesses in program order. Thus, $M1 \rightarrow M2$ even though the ordering semantics do not place any constraints on the relative ordering of M1 and M2.

2.2.1.7 Data Dependency Establishes Local Ordering

In the Itanium architecture, a dependency (e.g., a later operation reading the value written by an earlier operation) can imply a local ordering relationship between the two operations. This section focuses on dependencies through registers only. Section 2.2.1.6 discusses dependencies and MP ordering.

The execution shown in Table 2-6 illustrates how data dependency and memory ordering interact in a simple "pointer chase."

Table 2-6. Memory Ordering and Data Dependency

Processor #0	Processor #1
`st        [x] = 1        // M1`	`ld        r1 = [y] ;;    // M3`
`st.rel    [y] = x        // M2`	`ld        r2 = [r1]      // M4`

Outcome: r1 = x and r2 = 0 is not allowed

In this example, Processor #0 could be executing code that updates a shared object with M1 and then publishes a pointer to the object with M2. Processor #1 then loads the pointer and dereferences it to read the contents of the shared object. The outcome r1 = x and r2 = 0 implies that Processor #1 observes the new value of the object pointer, y, but the old value of the data field, x.

The ordering semantics require $\mathrm{M1} \rightarrow \mathrm{M2}$ but place no requirements on the relative ordering of M3 and M4.

Thus, the memory semantics alone would allow the outcome r1 = x and r2 = 0 in the absence of other constraints. Using an acquire load for M3 can avoid this outcome as doing so forces $\mathrm{M3} \rightarrow \mathrm{M4}$ and thus prevents the outcome. However, this use of acquire is non-intuitive given the RAW dependency through register r1 between M3 and M4. That is, M3 produces a value that M4 requires in order to execute so how should it be possible for them to go out of order? Further, using an acquire in this case prevents any memory operation following M3 from moving above M3, even if they are completely independent of M3.

To avoid this potential confusion and performance issue, the Itanium architecture treats data dependency and memory ordering in the same fashion on the local processor. That is, if $\mathrm{A} \gg \mathrm{B}$ and A produces a value that B consumes, then $\mathrm{A} \rightarrow \mathrm{B}$ on the local processor. This relationship is also transitive as the execution in Table 2-7 illustrates.

Table 2-7. Memory Ordering and Data Dependency Through a Predicate Register

Processor #0	Processor #1
`st        [x] = 1        // M1`	`    ld      r1 = [y]            // M3`
`st.rel    [y] = x        // M2`	`    cmp.eq  p1, p2 = r1, x ;;   // C1`
	`(p1)ld      r2 = [x]            // M4`

Outcome: r1 = x and r2 = 0 is not allowed

The Processor #0 code is the same as in Table 2-6. The Processor #1 now performs the following operation: if the pointer value y is equal to x, load a value from x.

The Itanium architecture does not allow the outcome r1 = x and r2 = 0 in this execution either. Unlike the execution in Table 2-6, there is no *direct* dependency between the values that M3 produces and the values that M4 consumes. However, there is a RAW through register r1 from M3 to C1 and a RAW through register p1 from C1 to M4. Thus, by transitivity, $\mathrm{M3} \rightarrow \mathrm{M4}$.

The execution in Table 2-8 illustrates a similar construct but introduces a control dependency.

Table 2-8. Memory Ordering and Data and Control Dependencies

Processor #0	Processor #1

`st      [x] = 1      // M1`
`st.rel  [y] = x      // M2`
	`    ld     r1 = [y];;      // M3`
	`    cmp.eq p1, p2 = r1, x // C1`
	`(p2)br     t               // B1`
	`    ld     r2 = [x]        // M4`
	`t:`

Outcome: r1 = x and r2 = 0 is not allowed

This execution is semantically the same as the execution in Table 2-7; however, this execution uses a control dependency rather than predication to conditionally execute M4. As a result, the outcome r1 = x and r2 = 0 is not allowed in the Table 2-8 execution.

The execution of the load M4 is data-dependent on the value of p2 that the branch B1 uses to resolve. Further, p2 is dependent on the value of r1 that the load M3 produces through the compare C1. Thus, $\mathrm{M3} \rightarrow \mathrm{M4}$.

The execution in Table 2-9 is a variation on the execution from Table 2-8 where the loads are truly independent.

Table 2-9. Memory Ordering and Control Dependency

Processor #0	Processor #1

`st      [x] = 1      // M1`
`st.rel  [y] = x      // M2`
	`     ld    r1 = [y]         // M3`
	`     cmp   p1, p2 = r3, x  // C1`
	`(p2) br    t                // B1`
	`     ld    r2 = [x]         // M4`
	`t:`

Outcome: all are allowed

In this execution, there is no dependency between M3 and M4, and thus, there are no constraints on the relative ordering of M3 and M4. Like the execution in Table 2-8, M4 is data-dependent on the value of p2 that the branch B1 uses to resolve. However, p2 is *independent* of the value that the load M3 produces (specifically, because the compare does not use the value of register r1 that the load produces). Thus, there is no chain of dependencies between M3 and M4 and therefore there are no constraints on the relative ordering of M3 and M4. As a result, all outcomes are allowed in this execution.

2.2.1.8 Store Buffers May Satisfy Local Loads

In the Itanium memory ordering model, store buffers (or other logically-equivalent structures) may satisfy local read requests from loads or acquire loads even if the stored data is not yet visible to other agents in the coherence domain. Such bypassing must honor any ordering semantics in the memory reference stream. Table 2-10 and Table 2-11 that Section 2.2.1.9 presents illustrate this behavior.

Table 2-10. Store Buffers May Satisfy Loads if the Stored Data is Not Yet Globally Visible

Processor #0	Processor #1

`st.rel [x] = 1 // M1`
`ld.acq r1 = [x] // M2`
`ld r2 = [y] // M3`
	`st.rel [y] = 1 // M4`
	`ld.acq r3 = [y] // M5`
	`ld r4 = [x] // M6`

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is allowed

In this sequence, each processor bypasses its locally-written value from a store buffer before the value becomes visible to the other processor. This behavior may make accesses of different sizes that have overlapping memory addresses appear to complete non-atomically.

The following discussion focuses on the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 because this outcome is allowed if and only if store buffers can satisfy local loads (other outcomes are allowed but do not depend on being able to satisfy local loads from a store buffer).

The Itanium memory ordering semantics only require that $M2 \rightarrow M3$ and $M5 \rightarrow M6$. There are no constraints on the relative ordering of M1 and M2 or M3 nor on the relative ordering of M4 and M5 or M6.

Remember that both dependencies and the memory ordering model place requirements on the manner in which a processor based on the Itanium architecture may re-order accesses. Even though the Itanium memory ordering model allows loads to pass stores, a processor based on the Itanium architecture cannot re-order the following sequence:

```
st.rel      [x] = r0        // M1: store 0 to [x]
ld.acq      r1 = [x]        // M2: cannot move above st.rel due to RAW
```

This is because there is a RAW dependency through memory between M1 and M2 and the Itanium memory ordering model requires that the local processor resolve RAW, WAR, and WAW dependencies between its memory accesses in program order. Thus, $M1 \rightarrow M2$ even though the ordering semantics place no constraints on the relative ordering of M1 and M2.

Because there is a RAW dependency through memory between M1 and M2 and between M4 and M5, the ordering constraints *effectively* become:[1]

$$M1 \rightarrow M2 \rightarrow M3$$
$$M4 \rightarrow M5 \rightarrow M6$$

[1] That is, the store operations must become visible to the local processors before their loads that read the stored value.

to account for both the memory ordering semantics and dependencies. It is important to keep in mind that the observance of a dependency between two operations does not imply an ordering relationship (from the standpoint of the memory ordering model) between the operations as Section 2.2.1.6 describes.

Assuming that a processor can bypass locally-written values before they are made globally-visible implies that there is a local and a global visibility points for a memory operation where a value always becomes locally visible before it becomes globally visible. Since M1 and M4 can have local visibility with respect to M2 and M5 as well as global visibility,

$$\mathrm{m1} \rightarrow \mathrm{M2} \rightarrow \mathrm{M3};\ \mathrm{m1} \rightarrow \mathrm{M1}$$
$$\mathrm{m4} \rightarrow \mathrm{M5} \rightarrow \mathrm{M6};\ \mathrm{m4} \rightarrow \mathrm{M4}$$

where m1 and M1 represent local and global visibility of memory operation 1, respectively. There are two things to note. First, the ordering of the local visibilities of operations M1 and M4 (m1 and m4, respectively) allow each processor to honor its data dependencies. That is, Processor #2 honors the RAW dependency through memory between M1 and M2 by requiring m1 to become visible before M2. Second, that these requirements do not place any constraints on the relative ordering perceived by a *remote* observer of operation M1 with M2 and M3 or of operation M4 with M5 and M6 (as the local visibilities meet the *local* ordering constraints that the dependencies impose).

The code in Table 2-10 and these constraints together imply that

$$\mathrm{r1} = 1 \Rightarrow \mathrm{m1} \rightarrow \mathrm{M2}$$
$$\mathrm{r3} = 1 \Rightarrow \mathrm{m4} \rightarrow \mathrm{M5}$$
$$\mathrm{r2} = 0 \Rightarrow \mathrm{M3} \rightarrow \mathrm{M4} \Rightarrow \mathrm{m1} \rightarrow \mathrm{M6} \text{ because } \mathrm{m1} \rightarrow \mathrm{M3} \text{ and } \mathrm{M3} \rightarrow \mathrm{M4} \text{ and } \mathrm{M4} \rightarrow \mathrm{M6}$$
$$\mathrm{r4} = 0 \Rightarrow \mathrm{M6} \rightarrow \mathrm{M1}$$
$$\mathrm{m1} \rightarrow \mathrm{M6} \text{ and } \mathrm{M6} \rightarrow \mathrm{M1} \Rightarrow \mathrm{m1} \rightarrow \mathrm{M1}$$

Thus, the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is allowed because these statements are consistent with our definition of local and global visibility. Specifically, a value becomes locally visible before it becomes globally visible. Similar reasoning can show that the constraints also imply that $\mathrm{m4} \rightarrow \mathrm{M4}$.

2.2.1.9 Preventing Store Buffers from Satisfying Local Loads

In the code shown in Table 2-10 from Section 2.2.1.8, there are no ordering constraints between the store and acquire load from the standpoint of memory ordering semantics (however, there is a RAW dependency through memory that forces the acquire load to follow the store). Bypassing may not occur if doing so violates the memory ordering constraints of memory operations between the store and the bypassing read. Table 2-11 presents a variation on the execution in Table 2-10 from Section 2.2.1.8 that illustrates this behavior.

Table 2-11. Preventing Store Buffers from Satisfying Local Loads

Processor #0	Processor #1

`st      [x] = 1     // M1`
`mf                  // M2`
`ld.acq  r1 = [x]    // M3`
`ld      r2 = [y]    // M4`
	`st      [y] = 1     // M5`
	`mf                  // M6`
	`ld.acq  r3 = [y]    // M7`
	`ld      r4 = [x]    // M8`

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed

Like Section 2.2.1.8, the discussion in this section focuses on the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 because it is allowed if and only if store buffers can satisfy local loads. The line of reasoning to show that the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed in Table 2-11 is similar to the reasoning used to show that this outcome is allowed in the Table 2-10 execution from Section 2.2.1.8 on page 2:518.

By the definition of the Itanium memory ordering semantics,

$$\mathrm{M1} \rightarrow \mathrm{M2} \rightarrow \mathrm{M3} \rightarrow \mathrm{M4}$$
$$\mathrm{M5} \rightarrow \mathrm{M6} \rightarrow \mathrm{M7} \rightarrow \mathrm{M8}$$

By allowing local and global visibility of operations M1 and M5 (similar to the discussion in Section 2.2.1.8), this assumption, along with the above constraints, together imply that,

$$\mathrm{m1} \rightarrow \mathrm{M1} \Rightarrow \mathrm{m1} \rightarrow \mathrm{M2} \rightarrow \mathrm{M3} \rightarrow \mathrm{M4}$$
$$\mathrm{m5} \rightarrow \mathrm{M5} \Rightarrow \mathrm{m5} \rightarrow \mathrm{M6} \rightarrow \mathrm{M7} \rightarrow \mathrm{M8}$$

Consider these constraints on the Processor #0 operations m1, M1, M2, M3, and M4. Making m1 visible before M2, M3, and M4 correctly honors the data dependency through memory on Processor #0. However, unless it constrains the global visibility of M1 to occur before M2, M3, and M4, Processor #0 violates the Itanium ordering semantics. Specifically, the memory fence M2 must always be made visible after the store M1. Allowing global and local visibilities of M1 in this case violates this constraint, and thus, is not allowed. Essentially, by allowing M1 to become locally visible early, M3 would see M1 before the fence semantics for M2 were met (namely, that M1 be visible before M2 and thus M3). Without local and global visibility of M1 and M5, the ordering constraints are as this example originally postulated.

The code in Table 2-11 and these constraints together imply that

$$\mathrm{r2} = 0 \;\Rightarrow \mathrm{M4} \rightarrow \mathrm{M5} \Rightarrow \mathrm{M1} \rightarrow \mathrm{M8} \text{ because } \mathrm{M1} \rightarrow \mathrm{M4} \text{ and } \mathrm{M4} \rightarrow \mathrm{M5} \text{ and } \mathrm{M5} \rightarrow \mathrm{M8} \Rightarrow \mathrm{r4} = 1$$

This contradicts the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome. The visibility of the memory fence, M2, implies that all prior operations including the store to x, M1, are globally visible. Thus, the load from x on Processor #1, M8, must observe the new value of x and $\mathrm{M1} \rightarrow \mathrm{M8}$ but the outcome requires $\mathrm{M8} \rightarrow \mathrm{M1}$.

2.2.1.10 Semaphores Do Not Locally Bypass

As Section 2.2.1.8 and Section 2.2.1.9 discuss, loads and acquire loads may be satisfied with values placed in local store buffers (or other logically-equivalent structures) by stores or release stores before the stored data becomes visible to other agents in the coherence domain. The Itanium architecture explicitly prohibits such local bypass either to or from semaphore operations. That is, semaphore operations cannot be satisfied in this way nor can the data they store be used to satisfy loads or acquire loads in this way.

The execution in Table 2-12 illustrates a variation on the execution in Table 2-10 where the acquire loads have been replaced with exchange semaphore operations (which also have acquire semantics).

Table 2-12. Bypassing to a Semaphore Operation

Processor #0	Processor #1
`mov        r5 = 2`	`mov        r6 = 2`
`st.rel     [x] = 1          // M1`	`st.rel     [y] = 1          // M4`
`xchg       r1 = [x], r5     // M2`	`xchg       r3 = [y], r6     // M5`
`ld         r2 = [y]         // M3`	`ld         r4 = [x]         // M6`

Outcome: r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed

Although each semaphore operation can be decomposed into a read access followed by a write access, the Itanium architecture does *not* allow a read request by a semaphore to be satisfied from a store buffer (or other logically-equivalent structure). As a result, the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed. The reasoning is similar to that presented in Section 2.2.1.9.

Specifically, by the definition of the Itanium memory ordering semantics, $\mathrm{M2} \rightarrow \mathrm{M3}$ and $\mathrm{M5} \rightarrow \mathrm{M6}$. The relative ordering between operation M1 and operations M2 or M3 is not constrained. Likewise, the relative ordering between operation M4 and operations M5 and M6.

Now, assume the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0. Given that r1 = 1, r3 = 1, and r2 = 0, we observe the following:

$$\begin{gathered}
\mathrm{r1} = 1 \Rightarrow \mathrm{M1} \rightarrow \mathrm{M2} \\
\mathrm{r3} = 1 \Rightarrow \mathrm{M4} \rightarrow \mathrm{M5} \\
\mathrm{r2} = 0 \Rightarrow \mathrm{M3} \rightarrow \mathrm{M4} \\
\mathrm{M3} \rightarrow \mathrm{M4} \Rightarrow \mathrm{M1} \rightarrow \mathrm{M6} \text{ because } \mathrm{M1} \rightarrow \mathrm{M3} \rightarrow \mathrm{M4} \rightarrow \mathrm{M6} \\
\mathrm{M1} \rightarrow \mathrm{M6} \ \Rightarrow \mathrm{r4} = 2
\end{gathered}$$

This conclusion contradicts the assumed outcome where r4 = 0 and thus the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 is not allowed. Because M1 and M4 cannot become locally-visible to M2 and M5 before they become globally-visible to M6 and M3 (as read accesses from semaphores may not bypass from store buffers or other logically-equivalent structures), it is not possible to avoid this contradiction.

The Itanium architecture also prohibits local bypass from a semaphore operation to a local read access from a load or acquire load as shown in the execution in Table 2-13.

Table 2-13. Bypassing from a Semaphore Operation

Processor #0	Processor #1
`fetchadd.rel   r5 = [x], 1   // M1`	`fetchadd.rel   r6 = [y], 1   // M4`
`ld.acq         r1 = [x]      // M2`	`ld.acq         r3 = [y]      // M5`
`ld             r2 = [y]      // M3`	`ld             r4 = [x]      // M6`

Outcome: r1 = 1, r3 = 1, r2 = 0, r4 = 0, r5 = 0, and r6 = 0 is not allowed

A store buffer may not provide a local read operation early access to a value written by a semaphore operation. Therefore, the outcome r1 = 1, r3 = 1, r2 = 0, r4 = 0, r5 = 0, and r6 = 0 in the Table 2-13 execution is not allowed. The reasoning is similar to that used in the previous execution.

2.2.1.11 Ordered Cacheable Operations are Seen in the Same Order by All Observers

The Itanium memory ordering model requires that release stores and semaphore operations (both acquire and release forms) become visible to all observers in the coherence domain in a single total order with the exception that each processor may observe (via loads or acquire loads) its own update early. Thus, each observer in the coherence domain sees the same interleaving of release stores and semaphores (both acquire and release forms) from the other processors in the coherence domain except that each processor may observe its own release stores (via loads or acquire loads) prior to their being observed globally. Table 2-14 illustrates this behavior.

Table 2-14. Enforcing the Same Visibility Order to All Observers in a Coherence Domain

Processor #0	Processor #1	Processor #2	Processor #3

`st.rel [x] = 1// M1`	`ld.acq r1 = [x]//M2`
	`ld     r2 = [y]//M3`
		`st.rel [y] = 1// M4`	`ld.acq r3 = [y]//M5`
			`ld     r4 = [x]//M6`

Outcome: only r1 = 1, r3 = 1, r2 = 0, and r4 =0 is not allowed

The Itanium memory ordering model only disallows the outcome r1 = 1, r3 = 1, r2 = 0, and r4 = 0 in this execution. By the definition of the Itanium memory ordering semantics,

$$M2 \rightarrow M3$$
$$M5 \rightarrow M6$$

The Itanium memory ordering model does not permit the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome as this would require that Processors #1 and #3 observe the release stores to x and y in different orders. Specifically, assuming that the outcome is r1 = 1, r3 = 1, r2 = 0, and r4 = 0:

$$r1 = 1 \Rightarrow M1 \rightarrow M2$$
$$r3 = 1 \Rightarrow M4 \rightarrow M5$$
$$r2 = 0 \Rightarrow M3 \rightarrow M4 \Rightarrow M1 \rightarrow M4 \text{ because } M1 \rightarrow M2, M2 \rightarrow M3, \text{ and } M3 \rightarrow M4$$
$$r4 = 0 \Rightarrow M6 \rightarrow M1 \Rightarrow M4 \rightarrow M1 \text{ because } M4 \rightarrow M5, M5 \rightarrow M6, \text{ and } M6 \rightarrow M1$$

The final two statements are inconsistent since both $M1 \rightarrow M4$ and $M4 \rightarrow M1$ cannot be true unless Processors #1 and #3 are allowed to see the release stores to x and y in different orders.

The Itanium memory ordering model allows the r1 = 1, r3 = 1, r2 = 0, and r4 = 0 outcome if either one or both of the release stores M1 and M4 are unordered since unordered operations need not be seen in the same total order by all observers in the coherence domain. Thus, in a version of the execution shown in Table 2-14 with unordered stores, Processor #2 observes $M1 \rightarrow M4$ while Processor #4 observes $M4 \rightarrow M1$.

The Itanium memory ordering model also allows this outcome if the release stores M1 and M4 are replaced with a memory fence followed by an unordered store. From the standpoint of a single processor, a release store has equivalent ordering semantics on the local processor to a memory fence followed by an unordered store. However, because the store in the memory fence/unordered store pair is unordered, it does not have any ordering requirements with respect to a remote processor. Even when processors are allowed to construct different interleavings, the ordering of an individual processor's memory references within the interleaving must always respect the ordering constraints placed on those references.

2.2.1.12 Obeying Causality

As noted in Section 2.2.1.11, the Itanium memory ordering model requires that release stores and semaphore operations (both acquire and release forms) become visible to all observers in the coherence domain in a single total order with the exception that each processor may observe (via loads or acquire loads) its own update early. Thus, each observer in the coherence domain sees the same interleaving of release stores, and semaphores operations from the other processors in the coherence domain.

A consequence of this is the fact that the Itanium memory ordering model respects causality in a certain way. Specifically, if a release store or semaphore operation causally precedes any store or semaphore operation, then the two operations will become visible to all processors in the causality order. Table 2-1 illustrates this behavior. Suppose that M2 reads the value written by M1. In this case, there is a causal relationship from M1 to M3 (a control dependency could also establish such a relationship). The fact that the store to x is a release store implies that, since there is a causal relationship from M1 to M3, M1 must become visible to processor #2 before M3.

Table 2-15. Intel® Itanium® Architecture Obeys Causality

Processor #0	Processor #1	Processor #2
st.rel [x] = 1 // M1	ld.acq r1 = [x]// M2 st [y] = 1 // M3	ld.acq r2 = [y] // M4 ld r3 = [x] // M5

Outcome: only r1 = 1, r2 = 1, and r3 = 0 is not allowed

The Itanium memory ordering model disallows the outcome r1 = 1, r2 = 1, and r3 = 0 in this execution (all other outcomes are allowed). To see this, we note the following. If r1 = 1, then $\mathrm{M1} \rightarrow \mathrm{M2}$ at Processor #1. Because M2 is an acquire load and $\mathrm{M2} \gg \mathrm{M3}$, $\mathrm{M2} \rightarrow \mathrm{m3}$, where m3 represents the local visibility of memory operation 1 (see Section 2.2.1.8). Thus, $\mathrm{M1} \rightarrow \mathrm{m3}$. Since M1 is a release store, it appears to become visible to all processors at the same time. This fact and $\mathrm{m3} \rightarrow \mathrm{M3}$ together imply $\mathrm{M1} \rightarrow \mathrm{M3}$.

If r2 = 1, $\mathrm{M3} \rightarrow \mathrm{M4}$. Because M4 is an acquire load, $\mathrm{M4} \rightarrow \mathrm{M5}$. If r3 = 0, then $\mathrm{M5} \rightarrow \mathrm{M1}$. Together, these imply $\mathrm{M3} \rightarrow \mathrm{M1}$, which contradicts the observation from the previous paragraph. Thus, the outcome r1 = 1, r2 = 1, and r3 = 0 is disallowed.

The indicated outcome would also be disallowed if M1 were a semaphore operation because, like release stores, each semaphore must appear to become visible at all processors at the same time. The indicated outcome would be allowed if M1 were a weak store, as a weak store may appear to become visible at different times to different processors.

2.2.2 Memory Attributes

In addition to the ordering semantics and data dependencies, the memory attributes of the page that is being referenced also influence access ordering and visibility. Using memory attributes allows the Itanium architecture to match the performance and the usage model to the type of device (e.g. main memory, memory-mapped I/O device, frame buffer, locations with side-effects, etc.) that backs a page of memory. Typically, memory with side-effects is mapped uncacheable while memory without side-effects is mapped as write-back cacheable.

Section 4.4, "Memory Attributes" describes memory attributes in the Itanium architecture in greater depth.

Memory with the uncacheable UC or UCE attributes is sequential by definition. A processor based on the Itanium architecture ensures that accesses to sequential memory locations reach a peripheral domain (a platform-specific collection of uncacheable locations, colloquially known as "a device") in program order with respect to all other accesses to sequential locations in the same peripheral domain. The sequential behavior of UC or UCE memory is independent of the ordering semantics (i.e. acquire, release, fence, or unordered) attached to the accesses.

Other observers (e.g. processors or other peripheral domains) need not see references to UC or UCE memory in sequential order if at all. When multiple agents are writing to the same device, it is up to software to synchronize the accesses to the device to ensure the proper interleaving.

The ordering semantics of an access to sequential memory determines how the access becomes visible to the peripheral domain with respect to other operations. For example, consider the code sequence shown in Figure 2-2.

Figure 2-2. Interaction of Ordering and Accesses to Sequential Locations

```
sequential_example:
        st      [data_0] = 0        // M1: put data in cacheable mem
        st      [data_1] = 0        // M2: put data in cacheable mem
        st.rel  [ready] = 1         // M3: tell device to get ready
        st      [start] = 1         // M4: tell device to start
```

In this code, assume that `data_0` and `data_1` are cacheable locations and `start` and `ready` are an uncacheable UC or UCE locations.

Sequentiality ensures that M3 and M4 reach the peripheral domain in program order (i.e. M3 before M4). Further, the release semantics on M3 ensures that it is not made visible to the peripheral domain until after M1 and M2 are made visible to the coherence domain. The M1 and M2 accesses may become visible to the coherence domains in any order as they both have unordered semantics. Even though the memory ordering semantics allow M4 to become visible before M3, the processor must make M3 visible before M4 because both `ready` and `start` are sequential locations.

2.2.3 Understanding Other Ordering Models: Sequential Consistency and IA-32

To provide a point of reference, it is helpful to understand other memory ordering models. These ordering models affect not only the programmer's view of the system, but also the overall system performance and design. Processors with relaxed memory ordering models may achieve higher performance than those with strict ordering models.

The most intuitive memory ordering model is "sequential consistency" (SC) which Lamport formally defines in [L79]. In sequential consistency, all processors see the memory references from a given processor in program order, and, in addition, all processors see the same system-wide interleaving of memory references from each processor.

The SC model precludes many common optimizations made in modern microprocessors to enhance performance. For example, in an SC system, a load may not pass a prior store until that store becomes globally visible (because all memory operations must become visible in program order). This requirement prevents the SC system from using a store buffer to hide the latency of store traffic by allowing loads that hit the cache to be serviced under a prior store that miss the cache.

To address such performance issues, many memory ordering models have been developed that relax the constraints of sequential consistency. Adve categorizes these memory models by noting how they relax the ordering requirements between reads and writes and if they allow writes to be read early [AG95]. The Itanium architecture allows for relaxed ordering between reads and writes and also allows writes to be read early under certain circumstances.

Aside from disallowing any relaxation of memory references, sequential consistency has two other subtle differences from the Itanium memory ordering model. First, it requires a total order of operations whereas the Itanium memory ordering model only requires a total order for release stores and semaphores. Second, remote processors must always honor data dependencies since the local processor does not have the option of re-ordering such accesses as can occur.

The IA-32 memory ordering relaxes write to read ordering and allows a processor to read its own writes before they are globally visible. Further, IA-32 allows each processor in the coherence domain to interleave the reference streams from other processors in the coherence domain in a different order. The per-processor orders must meet some additional constraints to ensure they are consistent with each other (enumerating and explaining these constraints is beyond the scope of this document). For more information on the IA-32 ordering model see Section 6.2.3.2, "IA-32 Segmentation" on page 1:131.

2.3 Where the Intel® Itanium® Architecture Requires Explicit Synchronization

The Itanium architecture requires a memory synchronization (`sync.i`) and a memory fence (`mf`) during a context switch to ensure that all memory operations prior to the context switch are made visible before the context changes. Without this requirement, the ordering constraints may be violated if the process migrates to a different processor. For example, consider the example shown in Figure 2-3.

Figure 2-3. Why a Fence During Context Switches is Required in the Intel® Itanium® Architecture

```
// Process A begins executing on Processor #0...

       ld.acq     r1 = [x]           // load executes on processor #0

// 1) Context switch occurs
// 2) O/S migrates Process A from Processor #0 to Processor #1
// 3) Process A resumes at the instruction following the ld.acq

       st         [y] = r2           // store executes on processor #1
```

In this example, Processor #1 may make the unordered store visible to the coherence domain before Processor #0 makes the acquire load visible. This violates the ordering constraints. Executing a memory fence during the context switch handler ensures that this violation can not occur.

See Section 4.5, “Context Switching” on page 2:557 on context management in a processor based on the Itanium architecture.

Interruptions do not affect memory ordering. On entry to an interrupt handler, memory operations from the interrupted program may still be in-flight and not yet visible to other processors in the coherence domain. A handler that expects that all memory operations that precede the interruption to be visible must enforce this requirement by executing a memory fence at the beginning of the handler.

2.4 Synchronization Code Examples

There are many synchronization primitives that software uses in multiprocessor or multi-threaded environments to coordinate the activities of different code streams. In this section, we present several typical examples to illustrate how some common constructs translate to the Itanium instruction set. In addition, the discussions identify special considerations with various implementations.

The examples use the syntax “[`foo`]” to indicate the memory location that holds the variable `foo`. Actual Itanium architecture-based assembly language would first move the address of `foo` into a register and then use this register as an operand to a memory access instruction. The alternate syntax is chosen to simplify and clarify the examples.

2.4.1 Spin Lock

Software commonly uses spin locks to guard access to a critical region of code. In these locks, the software “spins” while waiting for a shared lock variable to indicate that the critical region can be safely accessed. Typically, the lock code uses atomic operations such as compare and exchange or fetch and add to update the shared lock variable. Figure 2-4 shows a spin lock based on the `cmpxchg` instruction.

Figure 2-4. Spin Lock Code

```
// available. If it is 1, another process is in the critical section.
//
spin_lock:
        mov            ar.ccv = 0                 // cmpxchg looks for avail (0)
        mov            r2 = 1                     // cmpxchg sets to held (1)

spin:
        ld8            r1 = [lock] ;;             // get lock in shared state
        cmp.ne         p1, p0 = r1, r2            // is lock held (ie, lock == 1)?
(p1)    br.cond.spnt   spin ;;                    // yes, continue spinning

        cmpxchg8.acq   r1 = [lock], r2, ar.ccv ;;// attempt to grab lock
        cmp.ne         p1, p0 = r1, r2            // was lock empty?
(p1)    br.cond.spnt   spin ;;                    // bummer, continue spinning

cs_begin:
        // critical section code goes here...
cs_end:

        st8.rel        [lock] = r0 ;;             // release the lock
```

The spin lock code first initializes `ar.ccv` and a register with the values that indicate that the lock is available and held, respectively. A compare and exchange obtains the lock by exchanging `lock` with `1` if it currently holds `0`. Next, the first loop ensures that the code spins in cache while the lock is held by someone else. Once this loop finds that the lock is available, a compare and exchange instruction attempts to obtain the lock. If this instruction fails (e.g. because someone else obtained the lock in the meantime), the code resumes spinning in the first loop.

Spinning using only the `cmpxchg`/`cmp`/`br` loop may generate excessive coherency traffic. For example, if the `cmpxchg` always stores to memory (even if the comparison fails) and the lock is highly-contested, the platform may have to generate a number of read for ownership transactions causing `lock` to move around the system. Using the first `ld8`/`cmp`/`br` loop avoids this problem by obtaining `lock` in a shared state. In the worst case, when `lock` is not contested, this loop adds only the overhead of the additional compare and branch.

The initial `ld8` need not be an acquire load because of the control-flow in the spin loop: this load must become visible before the `cmpxchg8` because the load must return data in order for the compare and branch to resolve. Further, the store that relinquishes the lock after the critical section uses release semantics to prevent memory references from the critical from moving after the reference that releases the lock. Finally, the branches use “static predict not taken” hints to optimize for the case where the lock is not highly contested.

2.4.2 Simple Barrier Synchronization

A barrier is a common synchronization primitive used to hold a set of processes at a particular point in the program (the barrier) until all processors reach the location. Once all processes arrive at the barrier, they may all continue to execute. Figure 2-5 shows a sense-reversing barrier synchronization based on the `fetchadd` instruction from Hennessy and Patterson [HP96].

This type of barrier prevents a process that races ahead to the next instance of the barrier from trapping other (slow) processors that are in the process of leaving the barrier.

Figure 2-5. Sense-reversing Barrier Synchronization Code

```
// The total shared variable is one less than the number of processors
// that wait at the barrier.
// The release shared variable indicates if the processor must wait at
// the barrier (initially, this variable is 0).
// local_sense is a per-processor local variable that indicates the
// "sense" of the barrier (initially, this variable is 0).

sr_barrier:
        fetchadd8.acq  r1 = [count], 1           // update counter
        ld8            r2 = [total]              // get number of procs - 1
        ld8            r3 = [local_sense] ;;     // get local “sense” variable
        xor            r3 = 1, r3                // local_sense =! local_sense
        cmp.eq         p1, p2 = r1, r2;;         // p1 => last proc to arrive
        st8            [local_sense] = r3        // save new value of local_sense
(p1)    st8            [count] = r0              // last resets count to 0
(p1)    st8.rel    [release] = r3 ;;             // last allows other to leave

wait_on_others:
(p2)    ld8            r1 = [release] ;;         // p2 => more procs to come
(p2)    cmp.ne.and     p0, p2 = r1, r3           // have all arrived yet?
(p2)    br.cond.sptk   wait_on_others ;;         // nope, continue waiting

        // This mf prevents memory operations that follow the barrier code
        // from moving ahead of memory operations that precede the barrier
        // code
        mf ;;
```

The barrier code begins by atomically updating the number of processors that are waiting at the barrier, `count`, using a `fetchadd` instruction. For the last processor that reaches the barrier, the `fetchadd` instruction returns the same value as the `total` shared variable, which is one less than the number of processors that wait at the barrier. Other processors each get a unique value on the interval [0, `total`) based on the order in which they arrive at the barrier.

All processors except the last processor wait in the `wait_on_others` loop for the signal that all have arrived at the barrier. The last processor to arrive at the barrier provides this signal.

The signal to leave the barrier is deduced from the value of the `release` shared variable and the `local_sense` local variable. Upon entering the barrier, each processor complements the value in its private `local_sense` variable. Once in the barrier, all processors always have the same value in their `local_sense` variables. This variable indicates the value that release must have before the processor can leave the barrier. The last processor to arrive at the barrier releases the other processors by setting release to the new local_sense value.

The mf instruction in Figure 2-5 is necessary only if the programmer wishes to ensure that memory operations performed before the barrier code are visible to memory operations performed by any processor after the barrier code.

2.4.3 Dekker's Algorithm

Dekker's algorithm [D65] is a common synchronization construct that arbitrates for a resource through the use of several shared variables that indicate which processor is using the resource. Each processor has its own flag variable that it shares with all other processors in the system. When a processor attempts to enter the critical section, it sets its flag to one and checks to make sure the flags for the other processors are all zero.

The code in Figure 2-6 illustrates the core of this algorithm for a two-way multiprocessor system. In this example, a processor makes a single attempt to acquire the resource; typically, this code would appear in a loop. Although there is an array of per-processor flag variables, the code uses flag_me and flag_you to indicate to the flag variables for the processor attempting to obtain the resource and the other remote processor, respectively.

Dekker's algorithm assumes a sequential consistency ordering model. Specifically, it assumes that loading zero from flag_you implies that a processor's load and stores to the flag variables occur before the other processor's load and store to the flag variables. If this is not the case, both processors can enter the critical section at the same time.

Using unordered loads or stores to access the flag_me and flag_you variables does not guarantee correct behavior as the processor may re-order the accesses as it sees fit. Using an acquire load and release store is also not sufficient to ensure correct behavior because the ordering semantics always allow acquire loads to move earlier and release stores to move later. In the absence of the mf, it is possible for the load from flag_you to occur before the store to flag_me; even with acquire and release operations.

The first ld8 need not be an acquire load because of the control-flow that skips the critical section: this load must become visible before any memory operations in the critical section because the load must return data in order for the compare and branch to resolve.

Figure 2-6. Dekker's Algorithm in a 2-way System

```
// The flag_me variable is zero if we are not in the
// synchronization and critical section code and non-zero
// otherwise; flag_you is similarly set for the other processor.
// This algorithm does not retry access to the
// resource if there is contention.
//
dekker:
        mov    r1 = 1 ;;          // my flag = 1 (i want access!)
        st8    [flag_me] = r1
        mf ;;                     // make st visible first
        ld8    r2 = [flag_you] ;; // is other's flag 0?
        cmp.ne p1, p0 = 0, r2
(p1)    br.cond.spnt   cs_skip ;; // if not, resource in use

cs_begin:
        // critical section code goes here...
cs_end:

cs_skip:
        st8.rel [flag_me] = r0 ;; // release lock
```

2.4.4 Lamport's Algorithm

Like Dekker's algorithm, Lamport's algorithm [L85] also provides mutual exclusion for critical sections of code. Lamport's algorithm is very simple and, in the case of non-contested locks, only requires two read and two write memory accesses to enter the critical section. The algorithm uses two shared variables, x and y, and a shared array, b, that identify the process entering and using the critical section. Figure 2-7 presents Lamport's algorithm 2 [L85].

Lamport's algorithm expects that a processor that enters the critical section performs the set of operations: S = {store x, load y, store y, load x}[1]. To enforce this ordering, the Itanium architecture requires a memory fence in the middle of the {store x, load y} sequence and the {store y, load x} sequence. No combination of ordered semantics on the operations in each of these sequences will guarantee the correct ordering.

It is not possible for the store y in the second sequence to pass the load y in the first sequence because of the data dependency from the load y to the compare and branch. If the processor reaches the store y in the second sequence, the load of y from the first sequence must be visible. Likewise, it is not possible for memory operations in the critical section to move ahead of the final load x because of the data dependency between this load and the compare and branch that guards the critical section.

The accesses to the b array allow the algorithm to correctly handle contention for the lock. In such cases, the algorithm backs off and re-trys.

[1] There are some additional operations on the b array that are interposed in this sequence when contention for the resource occurs.

Figure 2-7. Lamport’s Algorithm

```
// The proc_id variable holds a unique, non-zero id for the process that
// attempts access to the critical section. x and y are the synchronization
// variables that indicate who is in the critical section and who is
// attempting entry. ptr_b_1 and ptr_b_id point at the 1’st and id’th
// element of b[].
//
lamport:
        ld8            r1 = [proc_id] ;;          // r1 = unique process id
start:
        st8            [ptr_b_id] = r1            // b[id] = “true”
        st8            [x] = r1                   // x = process id
        mf                                        // MUST fence here!
        ld8            r2 = [y] ;;
        cmp.ne         p1, p0 = 0, r2;;           // if (y != 0) then...
(p1)    st8            [ptr_b_id] = r0            // ... b[id] = “false”
(p1)    br.cond.sptk   wait_y                     // ... wait until y == 0

        st8            [y] = r1                   // y = process id
        mf                                        // MUST fence here!
        ld8            r3 = [x] ;;
        cmp.eq         p1, p0 = r1, r3 ;;         // if (x == id) then...
(p1)    br.cond.sptk   cs_begin                   // ... enter critical section

        st8            [ptr_b_id] = r0            // b[id] = “false”
        ld8            r3 = [ptr_b_1]             // r3 = &b[1]
        mov            ar.lc = N-1 ;;             // lc = number of processors - 1
wait_b:
        ld8            r2 = [r3] ;;
        cmp.ne         p1, p0 = r1, r2            // if (b[j] != 0) then...
(p1)    br.cond.spnt   wait_b ;;                  // ... wait until b[j] == 0
        add            r3 = 8, r3                 // r3 = &b[j+1]
        br.cloop.sptk  wait_b ;;                  // loop over b[j] for each j

        ld8            r2 = [y] ;;
        cmp.ne         p1, p0 = r2, r1 ;;         // if (y != id) then...
(p1)    br.cond.sptk   cs_begin                   // ... enter critical section
wait_y:
        ld8            r2 = [y] ;;                // wait until y == 0
        cmp.ne         p1, p2 = 0, r2
(p1)    br.cond.spnt   wait_y
        br             start                      // back to start to try again

cs_begin:
        // critical section code goes here...
cs_end:

        st8            [y] = r0                   // release the lock
        st8.rel        [ptr_b_id] = r0;;          // b[id] = “false”
```

2.5 Updating Code Images

There are four general techniques for updating code images in order to modify the code stream of a local or remote processor.

- Self-modifying code or code that modifies its own image.
- Cross-modifying code or code that modifies the image of code running concurrently on another processor.

- Programmed I/O for paging of code pages.
- DMA for paging of code pages.

The next four sections discuss these techniques in greater depth.

To illustrate the code sequences for self- and cross-modifying code, the examples in this section use the syntax "`st [foo] = new`" to represent a group of aligned stores that change the instruction at address `foo` to the instruction "`new`". The Itanium architecture requires that the instruction stream see aligned stores atomically. In addition, the syntax "`fc.i foo`" represents a group of flush cache instructions that ensures the cache line addressed by `foo` is coherent with all the instruction caches. Updating more than one instruction simply requires the appropriate store/flush "pair" for each updated instruction[1].

2.5.1 Self-modifying Code

Figure 2-8 presents the Itanium instruction sequence necessary to update a code image location on the local processor only.

Figure 2-8. Updating a Code Image on the Local Processor

```
patch_local:
        st      [code] = new_inst          // write new instruction
        fc.i    code ;;                    // flush new instruction
        sync.i ;;                          // sync i stream with store
        srlz.i ;;                          // serialize

        // Local caches and pipeline are now coherent with new_inst...
```

This code fragment changes the instruction at the address `code` to the new instruction `new_inst`. After executing this code, the change is visible to both the local processor's caches and its pipeline.

The `st` instruction updates the code image and the `fc.i` instruction ensures the value stored is coherent with the instruction cache. The `fc.i` is necessary because the Itanium architecture does not require instruction caches to be coherent with data stores for Itanium architecture-based code. Next, the `sync.i` ensures that the code update is visible to the instruction stream of the local processor and orders the cache flush with respect to subsequent operations by waiting for the prior `fc.i` instructions to be made visible. Finally, the `srlz.i` instruction forces the pipeline to re-initiate any instruction group fetches it performed after the `srlz.i` and also waits for the `sync.i` to complete; effectively making the pipeline coherent with the updated code image.

The serialization instruction is not necessary if software can *guarantee* that the processor encounters an event that re-initiates code fetches performed after the `sync.i`, such as an interruption or an `rfi`, before executing the new code. Events such as an interrupt or `rfi` both perform an instruction serialization which in this example waits for the `sync.i` to complete and then re-initiates code fetches.

[1] This description hides some of the complexity involved. Specifically, the flush and store operations have different sizes. Whereas multiple store instructions are necessary to update a 16 byte instruction, a single cache line flush invalidates at least two 16 byte instructions.

2.5.2 Cross-modifying Code

Consider a multi-threaded program for a multiprocessor system that dynamically updates some procedure that any processor in the system may execute. The program maintains several disjoint buffers to hold the new code and requires a processor to execute an IP-relative branch instruction at some address `x` to reach the code. In this scenario, the program updates the procedure by emitting the new code into a different buffer and then patching the branch at address `x` to target this new buffer. By carefully writing the update code, software can ensure that any processor in the system sees either:

- The original branch at address `x` that targets the original code in the old buffer along with the original code, or
- The new branch at address `x` that targets the new code in the new buffer along with the new code.

The code in Figure 2-9 illustrates an optimized Itanium architecture-based code sequence that implements the cross-modifying code for this example.

Figure 2-9. Supporting Cross-modifying Code without Explicit Serialization

```
patch:
        st      [new_code] = new_inst  // write new instruction
        fc.i    new_code ;;            // flush new instruction
        sync.i ;;                      // sync i stream with store

// Update the target of the branch that jumps to the updated code.
// This branch MUST be ip-relative. Before executing the following
// store, the branch jumps to somewhere other than "new_code".
//
        st.rel  [x] = "branch <new_code>"

// If it is desired to propagate "branch <new_code>" to both
// the local processor and remote processor now, the following
// code is also necessary:
//
        fc.i    x ;;                   // flush branch
        sync.i ;;                      // sync i stream with store
        mf ;;                          // fence
```

To reach the new code at `new_code`, the processor executes the branch instruction at `x`. Initially, this branch jumps to an address other than `new_code`.

Note: The programmer needs to ensure that the branch to `new_code` is updated atomically. If an 8-byte store is used to update the branch, then the programmer needs to ensure that the branch to `new_code` is either in the first or last slot of the bundle.

The release store ensures a processor cannot see the new branch at address `x` and the original code at address `new_code`. That is, if a processor encounters "`branch <new_code>`" at address `x`, then the processor's instruction cache must be coherent with the code image updates applied before the release store that updates the branch.

If remote processors may see either the old or new code sequence, the final three instructions in Figure 2-9 are not necessary. In this case, the remote processors see the code image updates at some point in the future. In the meantime, they continue to execute the old code.

The release store ensures that the code image updates are made visible to the remote processors in the proper order (i.e. new_code is updated before the branch at address x is updated). Using the final three instructions ensures that the remote processors will see the new code the next time they execute the branch at address x.

On the local processor, the branch at address x also serves to force the pipeline to be coherent with the code image update the machine without requiring an interrupt, rfi instruction, or srlz.i instruction. Table 2-16 enumerates the potential pipeline behaviors to illustrate this point.

Table 2-16. Potential Pipeline Behaviors of the Branch at x from Figure 2-9

Pipeline Operation	Scenario #1	Scenario #2	Scenario #3	Scenario #4
Fetch branch at x	Old branch	Old branch	New branch	New branch
Predict branch at x	Old target	New target	Old target	New target
Code at target	Old instruction	“New” instruction (but could be stale)	Old instruction	New instruction
Retire branch at x	Old retires	Must flush due to misprediction	Must flush due to misprediction	New retires

In the first and fourth scenarios, the pipeline fetches and executes either the old branch and old target instruction or the new branch and new target instruction. Note that if the pipeline sees the new branch, it must also see the new target instruction by virtue of the way the code in Figure 2-9 is written. Either of these behaviors is consistent.

In the second and third scenarios, the pipeline obtains a mix of the old or new branch and the old or new target instruction. In these cases, the pipeline must flush because the predicted target will not agree with the branch instruction.

This behavior is not guaranteed unless the branch at address x is IP-relative and taken. The branch must be IP-relative to ensure that both the instruction and target address can be atomically updated (this is only possible with an IP-relative branch because in this type of branch, the target address is part of the instruction).

2.5.3 Programmed I/O

Programmed I/O requires that the CPU copy data from the device controller to main memory using load instructions to read from the device and store instructions to write data into cacheable memory (page-in).

To ensure correct operation, Itanium architecture-based software must exercise care in the presence of Programmed I/O due to two features of the architecture. First, the Itanium architecture does not require an implementation to maintain coherency between local instruction and data caches for Itanium architecture-based code. Second, the Itanium architecture allows aggressive instruction prefetching. Specifically, an implementation can move any location from a cacheable page into its instruction cache(s) any time a translation for the location indicates that the page is present (i.e. the p bit of the translation is set).

A system that performs Programmed I/O can use a sequence similar to that shown in Figure 2-8 to perform the data movement. Figure 2-10 presents a code sequence that updates a code image on both the local and remote processors.

Figure 2-10. Updating a Code Image on a Remote Processor

```
patch_l_and_r:
        st      [code] = new_inst          // write new instruction
        fc.i    code ;;                    // flush new instruction
        sync.i ;;                          // sync i stream with store

// If the local processor must ensure that remote processors see
// the preceding memory updates before any subsequent memory
// operations, the following code is also necessary.
//
        mf ;;                              // make store visible to others

// If the local processor is going to execute the code and cannot
// cannot ensure instruction stream serialization, the following
// code is also necessary,
//
        srlz.i ;;                          // serialize my pipeline

// Local caches and pipeline are now coherent with new_inst, remote
// caches are now coherent with new_inst...
```

This code fragment changes the instruction at the address `code` to the new instruction `new_inst`. After executing this code, the change is visible to the local and remote processor's caches and to the local processor's pipeline, but may not be visible to remote processor's pipelines.

The sequence in Figure 2-10 is similar to the code from Figure 2-8 except an `mf` instruction occurs between the `sync.i` and `srlz.i` instructions. The fence is necessary if software must ensure that the code image update is made visible to all remote processors before any subsequent memory operations from the local processor. Although the `sync.i`, which orders the `st`/`fc.i` pair, has unordered semantics, it is an orderable operation and thus obeys the release or fence semantics of subsequent instructions (unlike an `fc.i` instruction; see Section 4.4.7, "Sequentiality Attribute and Ordering" for more information).

Because the pipeline is not snooped, the code in Figure 2-10 cannot ensure that a remote processor's pipeline is coherent with the code image update. In the local case shown in Figure 2-8, the `srlz.i` instruction enforces this coherency. As a result, the remote processor must serialize its instruction stream before it executes the updated code in order to ensure that a stale copy of some of the updated code is not present in the pipeline. This can be accomplished by explicitly executing a `srlz.i` before executing the updated code or by forcing an event that re-initiates any code fetches performed after the `fc.i` is observed to occur, such as an interruption or `rfi`.

Several optimizations to this code are possible depending on how software uses the updated code. Specifically, the `mf` and `srlz.i` can be eliminated under certain circumstances.

The `srlz.i` is not necessary if the local processor that updates the code image does not ever execute the new code. In this case, the local processor does not require its pipeline to be coherent with the changes to the code image. The fence is not necessary if the code image update can be made visible to remote processors in any relationship with subsequent memory operations from the local processor.

Finally, software may also eliminate the `mf` or `srlz.i` instructions if it *guarantees* that these operations will take place elsewhere (e.g. in the operating system) before the processor attempts to execute the updated code. For example, context switch routines must contain a memory fence (see Section 2.3 on page page 2:526). Thus, the fence is not required if a context switch *always* occurs before any program can use the updated code.

2.5.4 DMA

Unlike Programmed I/O, which requires intervention from the CPU to move data from the device to main memory, data movement in DMA occurs without help from the CPU. A processor based on the Itanium architecture expects the platform to maintain coherency for DMA traffic. That is, the platform issues snoop cycles on the bus to invalidate cacheable pages that a DMA access modifies. These snoop cycles invalidate the appropriate lines in both instruction and data caches and thus maintain coherency. This behavior allows an operating system to page code pages without taking explicit actions to ensure coherency.

Software must maintain coherency for DMA traffic through explicit action if the platform does not maintain coherency for this traffic. Software can provide coherency by using the flush cache instruction, `fc`, to invalidate the instruction and data cache lines that a DMA transfer modifies. Code such as that shown in Figure 2-8 on page 2:532 and Figure 2-10 on page 2:535 accomplish this task.

2.6 References

[AG95] S. V. Adve and K. Gharachorloo. “Shared memory consistency models: A Tutorial,” Rice University ECE Technical Report 9512, September 1995.

[L79] L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, C-28(9):690-691, September 1979.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, second edition, Morgan-Kaufmann, 1996.

[D65] E. W. Dijkstra. “Cooperating sequential processes,” Eindhoven, the Netherlands, Technological University Technical Report EWD-123, 1965.

[L85] L. Lamport. “A Fast Mutual Exclusion Algorithm,” Compaq Systems Research Center Technical Report 7, November 1985.

§

Interruptions and Serialization 3

This chapter discusses the interruption and serialization model. Although the Itanium architecture is an explicitly parallel architecture, faults and traps are delivered in program order based on IP, and from left-to-right in each instruction group. In other words, faults and traps are reported precisely on the instruction that caused them.

3.1 Terminology

In the Itanium architecture, an **interruption** is an event which causes the hardware automatically to stop execution of the current instruction stream, and start execution at the instruction address corresponding to the **interruption handler** for that interruption. When this happens, we say that an interruption has been **delivered** to the processor core.

There are two classes of interruptions in the Itanium architecture. **IVA-based interruptions** are handled by the operating system (OS), at an address determined by the location of the interrupt vector table (IVT) and the particular interruption that has occurred. **PAL-based interruptions** are handled by the processor firmware. PAL-based interruptions are not visible to the OS, though PAL may notify the OS that a PAL-based interruption has occurred; see Section 13.3, "Event Handling in Firmware" on page 2:632.

The architecture supports several different types of interruptions. These are defined below:

- A **fault** occurs when OS intervention is required before the current instruction can be executed. For example, if the current instruction misses the TLBs on a data reference, a Data TLB Miss fault may be delivered by the processor. Faults are delivered precisely on the instruction that caused the fault. The faulting instruction and all subsequent instructions do not update any architectural state (with the possible exception of subsequent instructions which violate a resource dependency[1]). All instructions executed prior to the faulting instruction update all their architectural state before the fault handler begins execution.
- A **trap** occurs when OS intervention is required after the current instruction has completed. For example, if the last instruction executed was a branch and PSR.tb is 1, a Taken Branch trap will be delivered after the instruction completes. Traps are delivered precisely on the instruction following the trapping instruction. The trapping instruction and all prior instructions update all their architectural state before the trap handler begins execution. All instructions subsequent to the trapping instruction do not update any architectural state.[1]

[1] When an interruption is delivered on an instruction whose instruction group contains one or more illegal dependency violations, instructions which follow the interrupted instruction in program order and which violate the resource dependency may appear to complete before the interruption handler begins execution. Software cannot rely upon the value(s) written to the resource(s) whose dependencies have been violated; the value(s) are undefined. For details refer to Section 3.4, "Instruction Sequencing Considerations" on page 1:39.

- When an external or independent agent (I/O device, timer, another processor) requires attention from the processor, an **interrupt** occurs. There are several types of interrupts. An initialization interrupt occurs when the processor has received an initialization request. A **Platform Management Interrupt** (PMI) can be generated by the platform to request features such as power management. Initialization interrupts and PMIs are PAL-based interruptions. An **external interrupt** occurs when an agent in the system requires the OS to perform some service on its behalf. External interrupts are IVA-based interruptions. Interrupts are delivered asynchronously with respect to program execution. The instruction upon which an interrupt is delivered may or may not be related to the interrupt itself.
- An **abort** is generated by the processor when a malfunction (Machine Check) is detected, or when a processor reset occurs. Aborts are asynchronous with respect to program execution. If caused by a particular instruction, an abort may be delivered sometime after that instruction completes. Aborts are PAL-based interruptions.

An interruption handler returns from interruption when it executes an `rfi` instruction. The `rfi` instruction copies state from specific control registers known as **interruption registers** into their corresponding architectural state (e.g. IIP is copied into IP and execution begins at that instruction address). Whether or not the state that is restored by the `rfi` is the same state that was captured when the interruption occurred is up to the operating system.

3.2 Interruption Vector Table

The Interruption Vector Address (IVA) control register defines the base address of the interruption vector table (IVT). Each IVA-based interruption has its own architected offset into this table as defined in Section 5.7, “IVA-based Interruption Vectors” on page 2:113. For the remainder of this section, “interruption” refers to an IVA-based interruption, unless otherwise noted.

When an interruption occurs, the processor stops execution at the current IP, sets the current privilege level to 0, and begins fetching instructions from the address of the entry point to the interruption handler for the particular interruption that occurred. The address of this entry point is defined by the base address of the IVT contained in the IVA register and the architected offset into the table according to the interruption that occurred.

The IVT is 32Kbytes long and contains the code for the interruption handlers. Execution of the interruption handler begins at the entry point. The interruption handler may be contained entirely in the IVT, or the handler may branch to code outside the IVT if more space is needed.

When an interruption occurs, if the processor is operating with instruction address translation enabled (PSR.it is 1), then the address in IVA is treated as a virtual address; otherwise, it is treated as a physical address. Whenever an interruption may occur (i.e. whenever external interrupts are not masked or disabled, or whenever an instruction may raise a fault or trap), the software must ensure that the processor can safely reference the IVT. As a result, the IVT must be permanently resident in physical memory. If instruction address translation is enabled, the IVT must be mapped by an instruction translation register and must point at a valid physical page frame. When instruction address translation is disabled, the IVA register should contain the physical address of the base of the IVT. Software must further ensure that instruction and memory references from low-level interruption handlers do not generate additional interruptions until enough state has been saved and interruption collection can be re-enabled.

There are many more interruptions than there are interruption vectors in the IVT. As specified in Section 5.6, "Interruption Priorities" there is a many-to-one relationship between interruptions and interruption vectors. The interruptions that share a common interruption vector (and hence, the code for an interruption handler) can determine which interruption occurred by reading the Interruption Status Register (ISR) control register. See Chapter 8, "Interruption Vector Descriptions" and Chapter 9, "IA-32 Interruption Vector Descriptions" for details of the specific ISR settings for each unique interruption.

3.3 Interruption Handlers

3.3.1 Execution Environment

As defined in Section 5.5, "IVA-based Interruption Handling" on page 2:101, the processor automatically clears the PSR.i and PSR.ic bits when an interruption is delivered. This disables external interrupts and interrupt state collection, respectively. PMI delivery is also disabled while PSR.ic is 0; other PAL-based interruptions can be delivered at any point during the execution of the interruption handler, regardless of the state of PSR.i and PSR.ic.

In addition to clearing the PSR.i and PSR.ic bits, the processor also automatically clears the PSR.bn bit when an interruption is delivered, switching to bank 0 of general registers GR16 - GR31. This provides the interruption handler with its own set of registers which can be used without spilling any of the interrupted context's register state, effectively saving GR16 - GR31 of the interrupted context. (This assumes PSR.bn is 1 at the time of interruption; see Section 3.4.3, "Nested Interruptions" on page 2:546 for how to deal with the case where PSR.bn is 0 at the time of interruption.)

As specified in Section 3.3.7, "Banked General Registers" on page 2:42, GR24 - GR31 **of bank 0** should not be used while PSR.ic is 1. By firmware convention, PAL-based interruption handlers may use these registers without preserving their values when PSR.ic is 1. When PSR.ic is 0, software may safely use GR24 - GR31 of bank 0 as scratch register.

Several other PSR bits and the RSE.CFLE are modified by the hardware when an interruption is delivered. Table 3-1 summarizes the execution environment that interruption handlers operate in, and what each PSR bit and the RSE.CFLE values mean for the interruption handler.

Table 3-1. Interruption Handler Execution Environment (PSR and RSE.CFLE Settings)

PSR Bit	New Value	Effect on Low-level Interruption Handler
be	DCR.be	Byte order used by handler is determined by be-bit in DCR register.
ic & i	0	Disables interruption collection and external interrupts. Bank 0 is made active bank. This is discussed above
bn	0	
dt, rt, it, pk	unchanged	Instruction/Data/RSE address translation and protection key setting remain unchanged.
dfl & dfh	0	Floating-point registers are made accessible. This allows handlers to spill FP registers without having to toggle FP disable bits first. Modified bits indicate which registers were touched. See Section 4.2.2, “Preservation of Floating-point State in the OS” on page 2:553 for details.
mfl, mfh	unchanged	
pp	DCR.pp	Privileged Monitoring is determined by pp-bit in DCR register. By default, user counters are enabled and performance monitors are unsecured in handlers. See Chapter 12, “Performance Monitoring Support” for details.
up	unchanged	
sp	0	
di	0	Instruction set transitions are not intercepted.
si	0	Interval timer is unsecured.
ac	0	No alignment checks are performed.
db, lp, tb, ss	0	Debug breakpoints, lower-privilege interception, taken branch and single step trapping are disabled.
cpl	0	Current privilege level becomes most privileged.
is	0	Intel Itanium Instruction set. Handlers execute Intel Itanium instructions.
id, da, ia, dd, ed	0	Instruction/data debug, access bit and speculation deferral bits are disabled. For details, refer to Section 5.5.4, “Single Instruction Fault Suppression” on page 2:104 and Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:105.
ri	0	Interrupt handler starts at first instruction is bundle.
mc	unchanged	Software can mask delivery of some machine check conditions by setting PSR.mc to 1, but the processor hardware does not set this bit upon delivery of an IVA-based interruption. Delivery of resets and BINITs cannot be masked.
RSE.CFLE (not a PSR bit)	0	Allows interruption handler to service faults in presence of an incomplete current register stack frame. This can happen when a mandatory RSE load takes an exception during when RSE is servicing a register stack underflow. For details refer to Section 6.6, “RSE Interruptions” on page 2:144.

3.3.2 Interruption Register State

The Itanium architecture provides a set of hardware registers which, if interruption collection is enabled, capture relevant interruption state when an interruption occurs. The state of the PSR.ic bit at the time of an interruption controls whether collection is enabled. In this section, it is assumed that interruption collection is enabled (PSR.ic is 1); see Section 3.4.3, “Nested Interruptions” on page 2:546 for details on handling interruptions when collection is disabled (PSR.ic is 0). For details on collection of interruption resources for each interruption vector refer to Chapter 8, “Interruption Vector Descriptions” and Chapter 9, “IA-32 Interruption Vector Descriptions.”

A processor based on the Itanium architecture provides the following interruption registers for collecting information about the latest interruption or the state of the machine at the time of the interruption:

- IPSR – A copy of the processor status register (PSR) at the moment the interruption occurred. The OS can use the IPSR to determine the value of any PSR bit when the interruption occurred. The contents of IPSR are restored into the PSR when the OS executes an `rfi` instruction. If the OS wishes to change the PSR state of the interrupted process (e.g. to step over an instruction debug fault), it can do so by modifying the IPSR contents before executing the `rfi`. When an interruption occurs, the processor sets IPSR.ri to the slot number (0, 1, or 2) of the instruction that was interrupted.
- IIP – A copy of the instruction pointer (IP) where the interruption occurred. The instruction bundle address contained in IIP, along with the IPSR.ri field, defines the instruction whose execution was interrupted. This instruction has not completed (i.e. it has not retired), so when the OS returns to the interrupted context, typically this is the instruction at which execution of the interrupted context resumes[1]. When the OS executes an `rfi` instruction, the contents of IIP are copied into the IP register and the processor begins fetching instructions from this address.
- ISR – Contains extra information about the specific interruption that occurred. This register is useful for determining exactly which interruption occurred for interruptions which share the same IVT vector.
- IFA – Faults related to addressing (e.g. Data TLB fault) materialize the faulting address in this register.
- ITIR – Faults related to addressing materialize the default page size and permission key for the region to which the faulting address belongs in this register.
- IIPA – Contains the instruction bundle address of the last instruction to retire successfully while PSR.ic was 1. In conjunction with ISR.ei, IIPA can be used by software to locate the instruction that caused a trap or that was executed successfully prior to a fault or interrupt.
- IIM – Instructions that take a Speculation fault (e.g. `chk`) or a Break Instruction fault (e.g. `break.i`) write this register with their immediate field when taking these faults. For these cases, the IIM register can be used to emulate the instruction, or to pass information to the fault handler; for example, software can use a particular immediate field value in a break instruction to indicate to the operating system that a system call is being performed.
- IHA – Faults related to the VHPT place the VHPT hash address in this register. See Section 5.3, “Virtual Hash Page Table” on page 2:571 for details.
- IFS – This register can be used by software to save a copy of the interrupted context’s PFS register, but an interruption handler must do this explicitly; hardware only clears the valid bit (IFS.v) upon interruption. See below for details.
- IIB0, IIB1 – Contain the 16-byte instruction bundle related to the interruption. Note that the IIB registers do not provide bundle information for all interruptions and are not supported on all processor implementations; please refer to Chapter 8,

[1] When an instruction faults because it requires emulation by the OS, the OS will normally skip the emulated instruction by returning to the instruction bundle address and slot number that follows IIP in program order. It does so by writing the next in-order bundle address and slot number into IIP and IPSR.ri, respectively, before executing an `rfi` instruction. Details on emulation handlers is in Chapter 7, “Instruction Emulation and Other Fault Handlers.”

"Interruption Vector Descriptions" for details. Software can use the instruction bundle information for debug and emulation purposes.

No other architectural state is modified when an interruption occurs. Note that only IIP, IPSR, ISR, and IFS are written by all interruptions (assuming PSR.ic is 1 at the time of interruption); the other interruption control registers are only written by certain interruptions, and their values are undefined otherwise. For details on which faults update which interruption resources refer to Chapter 8, "Interruption Vector Descriptions" and Chapter 9, "IA-32 Interruption Vector Descriptions."

3.3.3 Resource Serialization of Interrupted State

As defined in Section 3.2, "Serialization" on page 2:17, Itanium control register updates do not take effect until software explicitly serializes the processor's data or instruction stream with a `srlz.d` or a `srlz.i` instruction, respectively. Control register updates that change a control register's value and that have not yet been serialized are termed "in-flight." Refer to Section 3.2.3, "Definition of In-flight Resources" on page 2:19 for a precise definition.

When an interruption is delivered and before execution begins in the interruption handler, the processor hardware automatically performs an instruction and data serialization on all "in-flight" resources. As described in Section 3.3.1 and Section 3.3.2 above, the following resources determine the execution environment of the interruption handler:

- CR[IVA] – determines new IP
- CR[DCR].be – determines new value of PSR.be
- CR[DCR].pp – determines new value of PSR.pp
- PSR.ic – determines whether interruption collection is enabled
- RR[7:0] – determines new value of CR[ITIR] and CR[IHA]
- CR[PTA] – determines new value of CR[IHA]

Although these resources are guaranteed to be serialized prior to interruption handler execution, there is no guarantee that they will be serialized prior to the determination of the handler's execution environment. If there is a value in-flight for any of these resources at the time of interruption delivery, either the old or new value may be used to generate the values of IP, PSR, CR[ITIR] and CR[IHA] seen by the handler.

As a result, if the handler requires the latest value of the listed resources to determine its execution environment, software must ensure that external interrupts are disabled and that no instruction or data references will take an exception until the resource updates have been appropriately serialized. Typically, the code toggling these resources is mapped by an instruction translation register to avoid TLB related faults.

Note that CR[IPSR] is guaranteed to get the latest value of the PSR on an interruption, even if there are PSR updates in-flight that have not been previously serialized by software.

For example, assume that GR2 contains the new value for IVA and that PSR.i is 1. To modify the IVA register, software would perform the following code sequence, where the code page is mapped by an instruction translation register or instruction translation is disabled:

```
rsm psr.i          // external interrupts disabled upon next instruction
mov cr[iva] = r2
;;
srlz.i             // writing IVA requires instruction serialization
;;
ssm psr.i          // external interrupts will be re-enabled after next srlz
```

3.3.4 Resource Serialization upon rfi

An `rfi` instruction also performs an instruction and a data serialization operation when it is executed. Any values that were written to processor register resources by instructions in an earlier instruction group than the `rfi` will be observed by the returned-to instruction, except for those register resources which are also written by the `rfi` itself, in which case the value written by the `rfi` will be observed. This makes the interruption handler more efficient by avoiding additional data and instruction serialization operations before returning to the interrupted context.

3.4 Interruption Handling

The Itanium architecture-based operating systems need to distinguish the following interruption handler types:

- Lightweight interruptions: Lightweight interruption handlers are allocated 1024 bytes (192 instructions) per handler in the IVT. These are discussed in Section 3.4.1.
- Heavyweight interruptions: Heavyweight interruption handlers are allocated only 256 bytes (48 instructions) per handler in the IVT. These are discussed in Section 3.4.2.
- Nested interruptions: If an interruption is taken when PSR.ic was 0 or was in-flight, a nested interruption occurs. Nested interruptions are discussed in Section 3.4.3.

3.4.1 Lightweight Interruptions

Lightweight interruption handlers are allocated 1024 bytes (192 instructions) per handler in the IVT. Typically, lightweight handlers are written in Itanium architecture-based assembly code, and run in their entirety with interruption collection turned off (PSR.ic = 0) and external interrupts disabled (PSR.i = 0). Because these lightweight handlers are usually very short and performance-critical, they are intended to fit entirely in the space allocated to them in the IVT. An example of a lightweight interruption handler is the Data TLB vector (offset 0x0800). The first 20 vectors in the IVT, offsets 0x0000 (VHPT Translation vector) through 0x4c00 (reserved), are lightweight vectors. Typical lightweight handlers deal with instruction, data or VHPT TLB Misses, protection key miss handling, and page table dirty or access bit updates.

A typical lightweight interruption handler can operate completely out of register bank 0. If the bank 0 registers provide sufficient storage for the handler, none of the interrupted context's register state need be saved to memory, and the handler does not need to use stacked registers. Assuming no stacked registers are needed, the lightweight interruption handler can operate with an incomplete current register stack frame, obviating the need for `cover` and `alloc` instructions in the handler. This also allows the TLB related handlers to service TLB misses that result from mandatory RSE loads to the current frame.

3.4.2 Heavyweight Interruptions

Heavyweight interruption handlers are allocated only 256 bytes (48 instructions) per handler in the IVT. This stub provides enough space to save minimal processor state, re-enable interruption collection and external interrupts, and branch to another routine to handle the interruption. Unlike a lightweight interruption handlers described above, heavyweight interruption handlers use general register bank 0 only until they can establish a safe memory context for spilling the interrupted context's state. This allows heavyweight handlers to be interruptible and to take exceptions.

A heavyweight handler stub (i.e. the portion of the handler that is located in the IVT) should determine exactly which type of interruption has occurred based on its offset in the IVT and the contents of the ISR control register. It can then branch out of the IVT to the actual interruption handler. For some heavyweight interruptions (e.g. Data Debug fault), these handlers are typically written in a high-level programming language; for others (e.g. emulation handlers) the interruption can be handled efficiently in Itanium architecture-based assembly code.

The sequence given below illustrates the steps that an Itanium architecture-based heavyweight handler needs to perform to save the interrupted context's state to memory and to create an interruptible execution environment. These steps assume that the low-level kernel code, the kernel backing store, and the kernel memory stack are pinned in the TLB (using a translation register), so that no TLB misses arise from referencing those memory pages. The ordering of the steps below is approximate and other operating system strategies are possible.

1. Copy the interruption resources (IIP, IPSR, IIPA, ISR, IFA, IIB0-1) into bank 0 of the banked registers. To avoid conflicts with processor firmware, use registers GR24-31 for this purpose. Both register bank 0 and the interruption control registers are accessible, since, as described in Section 3.3.1, the processor hardware, upon an interruption always switches to register bank 0, and clears PSR.ic and PSR.i.
2. Preserve the interrupted the predicate registers into bank 0 of the banked registers.
3. Determine whether interruption occurred in the operating system kernel or in user space by inspecting both IPSR.cpl and the memory stack pointer (GR12).
    a. If IPSR.cpl is zero and the interrupted context was already executing on a kernel stack, then no memory stack switch is required.
    b. Otherwise, software needs to switch to a kernel memory stack by preserving the interrupted memory stack pointer to a banked register in bank 0, and setting up a new kernel memory stack pointer in GR12.

4. Allocate a "trap frame" to store the interrupted context's state on the kernel memory stack, and move the interruption state (IIP, IPSR, IIPA, ISR, IFA, IFS, IIB0-1), the interrupted memory stack pointer and the interrupted predicate registers from the banked registers to the trap frame.

5. Save register stack and RSE state by following the steps outlined in Section 6.11.1, "Switch from Interrupted Context" on page 2:148.

   a. If IPSR.cpl is zero and the interrupted context was not executing on a kernel backing store (determined by inspecting BSPSTORE), then the new kernel BSPSTORE needs to be allocated such that enough space is provided for the RSE to spill all stacked registers. The architectural required maximum RSE spill area is 16KBytes. As a result, BSPSTORE should be offset from the base of the kernel backing store base by at least 16KBytes. This offset can be reduced if the kernel queries PAL for the actual implementation-specific number of stacked physical registers (RSE.N_STACK_PHYS). Based on RSE.N_STACK_PHYS, the required minimum offset in bytes is:

      ```
      8 * (RSE.N_STACK_PHYS + 1 + truncate((RSE.N_STACK_PHYS + 62)/63))
      ```

   Otherwise, the interrupted context was already executing on the kernel backing store. In this case, no new BSPSTORE pointer needs to be setup. The sequence in Section 6.11.1, "Switch from Interrupted Context" on page 2:148, is still required, however, step 6 in that sequence can be omitted.

   In either case, the interrupted register stack and RSE state (RSC, PFS, IFS, BSPSTORE, RNAT, and BSP) needs to be preserved, and should be saved either to the trap frame on the kernel memory stack, or to a newly allocated register stack frame.

6. Switch banked register to bank one and re-enable interruption collection as follows:

   ```
         ssm 0x2000 // Set PSR.ic
         bsw.1;;    // Switch to register bank 1
         srlz.d     // Serialize PSR.ic update
   ```

   With interruptions collection re-enabled, the kernel may now branch to paged code and may reference paged data structures.

7. Preserve branch register and application register state according to operating system conventions.

8. Preserve general and floating-point register state. If this is an involuntary interruption, e.g. an external interrupt or an exception, then software must save the interrupted context's volatile general register state (scratch registers) to the "trap frame" on the kernel memory stack, or to the newly allocated register stack frame. If this is a voluntary system call then there is no volatile register state. Preserved registers may or may not be spilled depending on operating system conventions. Additionally, the Itanium architecture provides mechanisms to reduce the amount of floating-point register spills and fills. More details on preservation of register context are given in Section 4.2, "Preserving Register State in the OS" on page 2:551.

9. At this point enough context has been saved to allow complete restoration of the interrupted context. Re-enable taking of external interrupts using the ssm instruction as follows:

```
ssm 0x4000 ;; // Set PSR.i
```

There is no need to explicitly serialize the PSR.i update, unless there is a requirement to force sampling of external interrupts right away. Without the serialization, the PSR.i update will occur at the very latest when the next exception causes an implicit instruction serialization to occur.

10. Dispatch interruption service routine (can be high-level programming language routine).
11. Return from interruption service routine.
12. Disable external interrupts as follows:

```
rsm 0x4000 ;; // Clear PSR.i
```

There is no need to explicitly serialize the PSR.i update, since clearing of the PSR.i bit with the `rsm` instruction takes effect at the next instruction group. For details refer to the `rsm` instruction page in Chapter 2, “Instruction Reference” in Volume 3.

13. Restore general and floating-point register state saved in step 8 above.
14. Restore branch register and application register state saved in step 7 above.
15. Disable collection of interruption resources and switch banked register to bank zero as follows:

```
rsm 0x2000    // Clear PSR.ic
bsw.0;;       // Switch to register bank 0
srlz.d        // Serialize PSR update
```

16. Restore register stack and RSE state by following the steps outlined in Section 6.11.2, “Return to Interrupted Context” on page 2:148.
17. Restore interrupted context’s interruption state (e.g., IIP, IPSR, IFS) from the “trap frame” on the kernel memory stack.
18. Restore interrupted context’s memory stack pointer and predicate registers from the trap frame on the kernel memory stack. This step essentially deallocates the trap frame from the kernel memory stack.
19. Return from interruption using the `rfi` instruction.

Many of the steps shown above are identical for different heavyweight interruptions, so unless there is a specific need to create a different handler for a particular interruption, a common handler can be used. Because external interrupt handlers use the Itanium external interrupt control registers to determine the specific external interrupt vector that needs servicing and to mask off other external interrupt vectors, an external interrupt handler looks somewhat different. Refer to Section 10.4, “External Interrupt Delivery” on page 2:606 for details on writing external interrupt handlers.

3.4.3 Nested Interruptions

The Itanium architecture provides a single set of interruption registers whose updates are controlled by PSR.ic. When an IVA-based interruption is delivered and PSR.ic is 0 or in-flight (e.g. during a lightweight interruption handler, or at the beginning of a heavyweight interruption handler), we say that a nested interruption has occurred. On a nested interruption (other than a Data Nested TLB fault) only ISR is updated by the hardware. All other interruption registers preserve their pre-interruption contents.

With the exception of the Data Nested TLB fault, the Itanium architecture does not support nested interruptions. Data Nested TLB faults are special and are discussed in Section 5.4.4, “Data Nested TLB Vector” on page 2:576. The remainder of this section does not apply to Data Nested TLB faults.

When a nested interruption occurs, the processor will update ISR as defined in Chapter 8, “Interruption Vector Descriptions” and it will set the ISR.ni bit to 1. A value of 1 in ISR.ni is the only indication to an interruption handler that a nested interruption has occurred. Since all other interruption registers are not updated, there is generally no way for the OS to recover from nested interruptions; the handler for the nested interruption has no context other than ISR for handling the nested interruption. If a nested interruption is detected, it is often useful for the handler to call some function in the OS that logs the state of ISR, IIP, and any other relevant register state to aid in debugging the problem.

§

Context Management 4

This chapter discusses specific context management considerations in the Itanium architecture. With 128 general registers and 128 floating-point registers, the architecture provides a comparatively large amount of state. This chapter discusses various context management and state preservation rules. This chapter introduces some architectural features that help an operating system limit the amount of register spill/fill and gives recommendations to system programmers as to how to use some of the instruction set features.

4.1 Preserving Register State across Procedure Calls

The Itanium Software and Runtime Architecture Conventions [SWC] define a contract on register preservation between procedures as follows:

- Scratch Registers (Caller Saves): GR2-3, GR8-11, GR14-GR15, and GR16-31 in register bank 1, FR6-15, and FR32-127. Code that expects scratch registers to hold their value across procedure calls is required to save and restore them.
- Preserved Registers (Callee Saves): GR4-7, FR2-5, and FR16-31. Procedures using these registers are required to preserve them for their callers.
- Stacked Registers: GR32-127, when allocated, are preserved by the RSE.
- Constant Register: GR0 is always 0. FR0 is always +0.0. FR1 is always +1.0.
- Special Use Registers: GR1, GR12, and GR13 have special uses.

Additional architectural register usage conventions apply to GR16-31 in register bank 0 which are used by low-level interrupt handlers and by processor firmware. For details refer to Section 3.3.1.

Itanium general registers and floating-point registers contain three state components: their register value, their control speculative (NaT/NaTVal) state, and their data speculative (ALAT) state. When software saves and restores these registers, all three state components need to be preserved. As described in Table 4-1, software is required to use different state preservation methods depending on the type of register. More details on register preservation are provided in the next two sections.

Table 4-1. Preserving Intel® Itanium® General and Floating-point Registers

State Components	General Registers		Floating-point Registers
	GR1-31 (static)	GR32-127 (stacked)	FR2-127
Register Value	`st8.spill` & `ld8.fill` preserve register value.	RSE automatically preserves register value.	`stf.spill` & `ldf.fill` preserve register value.
Control Speculative State (NaT/NaTVal)	`st8.spill` & `ld8.fill` preserve register NaT.	RSE automatically preserves register NaT.	`stf.spill` & `ldf.fill` preserve NaTVal.
Data Speculative State (ALAT)	Software must `invala.e` a register's ALAT state when restoring the register.	RSE and ALAT manage stacked register's ALAT state automatically.	Software must `invala.e` a register's ALAT state when restoring the register.

4.1.1 Preserving General Registers

The Itanium general register file is partitioned into two register sets: GR0-31 are termed the **static general registers** and GR32-127 are termed the **stacked general registers**. Typically, `st8.spill` and `ld8.fill` instructions are used to preserve the static GRs, and the processor's register stack engine (RSE) automatically preserves the stacked GRs.

Using the `st8.spill` and `ld8.fill` instructions, the general register value and its NaT bit are always preserved and restored in unison. However, these instructions do not save and restore a register's data speculative state in the Advanced Load Address Table (ALAT). To maintain the correct ALAT state, software is therefore required to explicitly invalidate a register's ALAT entry using the `invala.e` instruction when restoring a general register. The Itanium calling conventions avoid such explicit ALAT invalidations by disallowing data speculation to preserved registers (GR4-7) across procedure calls.

Spills and fills of general registers using `st8.spill` and `ld8.fill` cause implicit collection and restoration of the accompanying NaT bits to/from the User NaT collection application register (UNAT). The UNAT register needs to be preserved by software explicitly. The spill and fill instructions derive the UNAT bit index of a spilled/filled NaT bit from the spill/fill memory address and not from the spilled/filled register index. As a result, software needs to ensure that the 512-byte alignment offset[1] of the spill/fill memory address is preserved when a general register is restored. This can be an issue particularly for user context data structures that may be moved around in memory (e.g. a `setjmp()` jump buffer).

Unlike the `st8.spill` and `ld8.fill` instructions, the register stack engine (RSE) preserves not only register values and register NaT bits, but it also manages the stacked register's ALAT state by invalidating ALAT that could be reused by software when the physical register stack wraps. This automatic management of ALAT state across procedure calls permits compilers to use speculative advanced loads (`ld.sa`) to perform cross-procedure call control and data speculation in stacked general registers (GR32-127). Whenever software changes the virtual to physical register mapping of the stacked registers, the ALAT needs to be invalidated explicitly using the `invala` instruction. Typically this happens during process/thread context switches or in `longjmp()` when the register stack is reloaded with a new BSPSTORE. Refer to Section 4.5.1.1, "Non-local Control Transfers (setjmp/longjmp)" on page 2:557.

The RSE collects the NaT bits of the stacked general registers within the RNAT application register and automatically saves and restores accumulated RNAT collections to/from fixed locations within the register stack backing store. RNAT collections are placed on the backing store whenever BSPSTORE bits{8:3} are all one, which results in one RNAT collection for every 63 registers. When software copies a backing store to a new location, it is required to maintain the backing store's 512-byte alignment offset[2] to ensure that the RNAT collections get placed at the proper offset.

[1] The specific requirement is that (fill_address mod 512) must be equal to (spill_address mod 512).
[2] The specific requirement is that (old_bspstore mod 512) must be equal to (new_bspstore mod 512).

4.1.2 Preserving Floating-point Registers

The Itanium architecture encodes a floating-point register's control speculative state as a special unnormalized floating-point number called NaTVal. As a result, Itanium floating-point registers do not have a NaT bit. The architecture provides the `stf.spill` and `ldf.fill` instructions to save and restore floating-point register values and control speculative state. These instructions always generate a 16-byte memory image regardless of the precision of the floating-point number contained in the register.

Preservation of data speculative state associated with floating-point registers needs to be managed by software. As with the general registers, software is required to explicitly invalidate a register's ALAT entry using the `invala.e` instruction when restoring a floating-point register. The Itanium calling conventions avoid such explicit ALAT invalidations by disallowing data speculation to preserved floating-point registers (FR2-5, FR16-31) across procedure calls.

4.2 Preserving Register State in the OS

The software calling conventions described in the previous section apply to state preservation across procedure call boundaries. When entering the operating system kernel either voluntarily (for a system call) or involuntarily (for handling an exception or an external interrupt) additional concerns arise because the interrupted user's context needs to be preserved in its entirety.

The Itanium architecture defines a large register set: 128 general registers and 128 floating-point registers account for approximately 1 KByte and 2 KBytes of state, respectively. The architecture provides a variety of mechanisms to reduce the amount of state preservation that is needed on commonly executed code paths such as system calls and high frequency exceptions such as TLB miss handlers.

Additionally, Itanium architecture-based operating systems have opportunities to reduce the amount of context they need to save by distinguishing various kernel entry and exit points. For instance, when entering the kernel on behalf of a voluntary system call, the kernel need only preserve registers as outlined by the calling conventions. Furthermore, the operating system can be sensitive to whether the preserved context is coming from the IA-32 or Itanium instruction set, especially since the IA-32 register context is substantially smaller than the full Itanium register set. Ideally, an Itanium architecture-based operating system should use a single state storage structure which contains a field that indicates the amount of populated state.

Table 4-2 summarizes several key operating system points at which state preservation is needed.

Scratch GRs and FRs, the bulk of all state, only need to be preserved at involuntary interruptions resulting from unexpected external interrupts or from exceptions that need to call code written in a high-level programming language. The demarcation of floating-point registers FR32-127 as "scratch" along with architectural support for lazy state save/restore of the floating-point register file allows software to substantially reduce the overhead of preserving the scratch FRs. See Section 4.2.2 for details.

In principal, preserved GRs and FRs need not be spilled/filled when entering the kernel. Whatever function is called from the low-level interruption handler or the system call entry point will itself observe the calling conventions and preserve the registers. The only occasion when preserved registers need to be spilled/filled is on a process or thread context switch. However, many operating systems provide `get_context()` functions that provide user context upon demand. Although such functions are called infrequently, many operating systems prefer to pay the penalty of spilling preserved registers at system call and at interruption entry points to avoid the complexity of piecing together user state from various potentially unknown kernel stack locations on demand. Fortunately, the amount of preserved Itanium general register state is relatively small, and the Itanium architecture provides additional mechanisms for lazy floating-point state management. See Section 4.2.2 for details.

Table 4-2. Register State Preservation at Different Points in the OS

Register Type	Number of Registers	System Call (Voluntary)	Lightweight Interruptions[a] (Involuntary)	Heavyweight Interruptions[b] (Involuntary)	Process/Thread Context Switch (Voluntary)
Scratch GRs	23	no spill/fill required	Untouched (use banked registers)	spill/fill required	no spill/fill required (done at interruption)
Preserved GRs	4	no spill/fill required	Untouched (use banked registers)	no spill/fill required	spill/fill required
Stacked GRs	96	Backing Store Switch	Untouched	Backing Store Switch	Synchronous Backing Store Switch using flushrs[c]
Scratch FRs	106	no spill/fill required	Untouched	spill/fill required	no spill/fill required (done at interruption)
Preserved FRs	20	no spill/fill required	Untouched	no spill/fill required	spill/fill required

a. For details on lightweight interruption handlers refer to Section 3.4.1, “Lightweight Interruptions” on page 2:543.

b. For details on heavyweight interruption handlers refer to Section 3.4.2, “Heavyweight Interruptions” on page 2:544.

c. Refer to Section 6.11.3, “Synchronous Backing Store Switch” for details.

Stacked GRs are managed by the register stack engine (RSE). On process/thread context switches the operating system is required to completely flush the register stack to its backing store in memory (using the `flushrs` instruction). In cases where the operating system knows that it will return to the user process along the same path, e.g. in system calls and exception handling code, the Itanium architecture allows operating systems to switch the register stack backing store without having to flush all stacked registers to memory. This allows such kernel entry points to switch from the user’s to the kernel’s backing store without causing any memory traffic, as described in the next section.

4.2.1 Preservation of Stacked Registers in the OS

A switch from a thread of execution into the operating system kernel, whether on behalf of an involuntary interruption or a voluntary system call, requires preservation of the stacked registers. Instead of flushing all dirty stacked register’s to memory, the RSE can be used to automatically preserve the stacked registers of the interrupted context.

Automatic preservation offers performance benefits: the register stack may contain only a handful of dirty registers, system call parameters can be passed on the register stack, and, upon return to the interrupted context the `loadrs` instruction only needs to restore registers that were actually spilled to memory. Since system call rates scale with processor performance, the RSE offers a key method for reducing the kernel's execution time of a system call.

To ensure operating system integrity the RSE requires a valid backing store (i.e. one with a valid page mapping). The validity of the current backing store depends on the interrupted context. If the interrupted context is itself a kernel thread, then its backing store is in a known state, and no backing store switch is required (assuming that kernel interruptions are nested). If the interrupted context is a user process, then the backing store could be pointing at an invalid region of memory, and software is required to redirect the RSE at a kernel backing store. Section 6.11.1, "Switch from Interrupted Context" on page 2:148 describes the code sequence to switch the RSE backing store without causing memory traffic.

If the kernel redirects the backing store to a kernel memory region, then the kernel must restore the backing store of the interrupted context prior to resumption of the interrupted context. The kernel must also restore the register stack to its interrupted state by manually pulling the spilled registers from the backing store. The kernel uses the `loadrs` instruction to restore stacked registers from the backing store. The `loadrs` instruction requires the backing store pointer to align with any registers spilled from the interrupted context. Thus the kernel should have paired all function calls (`br.call` instructions) with function returns (`br.ret` instructions), or manually manipulated the kernel backing store pointer, so that all kernel contents have been removed from the kernel backing store prior to the `loadrs`. After loading the stacked registers, the kernel can switch to the backing store of the interrupted frame. This code sequence is described in Section 6.11.1, "Switch from Interrupted Context" on page 2:148.

The kernel may occasionally gather the complete interrupted user context, such as to satisfy a debugger request or to provide extended information to a user signal handler. To provide the preserved register stack contents, including NaT values, the kernel must extract the user context values from its backing store.

4.2.2 Preservation of Floating-point State in the OS

A full preservation of Itanium floating-point register file requires approximately 2 KBytes of memory. To reduce the frequency of such large register spills and fills, the Itanium architecture offers additional mechanisms for lazy floating-point state management. These features allow the system programmer to eliminate many unnecessary floating-point state spills and fills especially around voluntary and involuntary entries into the kernel, e.g. around system calls, external interrupts and exceptions. Lazy state preservation can provide a significant reduction of memory traffic and hence faster interrupt handlers and system calls, especially since most interrupt handlers and much system code rarely perform floating-point computations.

The 126 non-constant floating-point registers are architecturally divided into the lower set (FR2-31) and the higher set (FR32-127). The Itanium architecture provides two floating-point register set "modified" bits, PSR.mfl and PSR.mfh, which are set by hardware upon a write to any register in the lower and higher sets, respectively. The "modified" bits are accessible to a user process through the user mask. Additionally, two “disabled” bits, PSR.dfl and PSR.dfh, are accessible to the privileged software alone. Setting a “disabled” bit causes a fault into the disabled-fp vector upon first use (read or write) of the corresponding register set.

As mentioned earlier, an involuntary kernel entry (e.g. interruption) needs to preserve all scratch floating-point registers. Instead of blindly always spilling all registers, state spills can be conditionalized upon the “modified” bits in the PSR. Additionally, the “disabled” bits allow a deferred, or lazy, approach to both spills and fills. This is particularly useful for “on demand” state motion in an involuntary interruption handler that does not use many floating-point registers. To perform deferred spills on the high set, the handler sets PSR.dfh immediately upon entry. Any reference to a floating-point register in the high set will then fault into the disabled-fp vector which spills the corresponding state to a prearranged store before allowing use within the handler. Lazy state restoration is performed in a similar manner: the handler sets the “disabled” bit just before exit, causing the first reference by the interrupted context to the disabled set to fault into the kernel’s disabled floating-point vector which can then restore the appropriate state. Note the importance of agreeing upon prearranged stores for deferred spill/fill policies and the need for a mechanism to communicate a past fill or spill.

At process or thread context switches all preserved floating-point registers need to be context switched. The higher (scratch) set is also managed here if the context-switch was occasioned by an involuntary interruption (e.g. timer interrupt) which did not already spill the higher set. Use of the “modified” bits by the OS to determine if the appropriate register set is “dirty” with previously unsaved data can help avoid needless spills and fills.

The “modified” bits are intentionally accessible through the user mask so that a user process can provide hints to the OS code about its register liveness requirements. Clearing PSR.mfh, for instance, suggests that the user process does not see the higher register set as containing useful data anymore.

4.3 Preserving ALAT Coherency

As described in Section 4.4.5.3, “Detailed Functionality of the ALAT and Related Instructions” on page 1:65, software is required to explicitly invalidate the entire ALAT using the `invala` instruction whenever the virtual to physical register mapping is changed. Typically this occurs when the `clrrb` instruction is used, when a synchronous backing store switch is performed (e.g. in a user-level or kernel thread context switch), or when software “discontinuously” remaps the register to backing store mapping by resetting BSPSTORE (e.g. by calling `longjmp()`).

When returning to a user-process after servicing an involuntary interruptions, an Itanium architecture-based operating system is required to invalidate the entire ALAT using the `invala` instruction. This is required because the operating system may have targeted advanced loads at scratch registers, and thereby altered the user-visible ALAT state.

When returning from a system call, however, full ALAT invalidations can be avoided by using `invala.e` instructions to selectively invalidate ALAT entries of all preserved registers (GR4-7, FR2-5, and FR16-31), or by ensuring that these registers where never accessible to software during the system call (see Section 4.2.2 for details). This works, because at the system call entry user-code may not have any dependencies on the state of the scratch registers.

4.4 System Calls

Reducing the overhead associated with system calls becomes more important as processor efficiency increases. As processor frequencies and pipeline lengths increase, the typical overhead associated with flushing the processor pipeline to effect privilege domain crossings is increased. To reduce system call overhead, the Itanium architecture provides an efficient "enter privileged code" (epc) instruction (page 3:53) that can be paired with the demoting branch return. Additionally, the Itanium architecture provides the traditional break instruction (page 3:29) to enter privileged mode, that is typically paired with the rfi instruction (page 3:236) to return to user mode.

The epc instruction offers higher efficiency than the break instruction for invoking a kernel system call. Whereas a break instruction will always cause a pipeline flush to change privilege level, the epc is designed not to. The break instruction also passes the system call number as a parameter, and requires a table lookup with an indirect branch to the system call. With the epc instruction, the user application can directly branch to the system call code.

More information about epc-based system calls is provided in Section 4.4.1. More information about break-based system calls is provided in Section 4.4.2. Regardless of whether the epc or break instruction are used, an Itanium architecture-based operating system needs to check the integrity of system call parameters. In addition to traditional integrity checking of the passed parameter values, the system call handler should inspect system call parameters for set NaT bits as described in Section 4.4.3.

4.4.1 epc/Demoting Branch Return

To execute a system call with epc, a user system call stub branches to an execute-only kernel page containing the system call, using the br.call instruction. The kernel page executes an epc to raise the privilege level. The privilege level is raised to the privilege level of the page mapping corresponding to the instruction address of the epc instruction. The page mapping must be execute-only (see Section 4.1.1.6, "Page Access Rights" for details).

After the kernel completes its system call, it returns to the user system call stub with a br.ret instruction. The br.ret demotes the privilege level, by restoring the privilege level contained within the PFS application register (PFS.ppl). To ensure operating system integrity epc checks that the PFS.ppl field is no greater than the PSR.cpl at the time the epc is executed.

As described in Section 4.2.1, interruptions and system calls in a typical Itanium architecture-based operating system need to switch to the kernel register stack backing store upon kernel entry. The epc instruction does not disable interrupts nor does it switch the processor to the kernel backing store. As a result, code directly following the epc instruction that runs at increased privilege level is still running on the caller's backing store. It is recommended that software disable external interrupts right after the epc until the switch to the kernel backing store has been completed. Additionally, low-level operating system handlers should not only use IPSR.cpl, but should also check BSPSTORE, to determine whether they are running on the kernel backing store (imagine an external interrupt being delivered on the first instruction after the epc).

4.4.2 break/rfi

The break instruction, when issued in the *i*, *f*, and *m* syllables, specifies an arbitrary 21-bit immediate value. The kernel can choose a specific break immediate value to differentiate system calls from other usage of the break instruction (such as debug). The break instruction jumps to the break fault handler, which should be a valid address mapping for each user application, and raises the privilege mode to the most privileged level.

The system call number is an additional parameter passed to the kernel when invoking a system call via the break instruction. The system call number must reside in a fixed location. If stored within GR32, then the system call stub must rearrange its input parameters to map to the register stack starting at GR33. This register jostling can be avoided by passing the system call number through a scratch static general register or by using the break immediate itself. Additionally, the system call can utilize all eight input registers of the register stack for system call parameters.

4.4.3 NaT Checking for NaTs in System Calls

In addition to regular range/value checking on system call arguments, Itanium architecture-based operating systems need to additionally ensure that system call arguments passed in by a user application do not have any NaT bits set. The following code fragment can be used:

```
        mov mask = 0xff
        clrrrb
        ;;
// create register stack frame with only output registers for system call args
        alloc tmp = ar.pfs, 0, 0, 8, 0
        shl mask = mask, syscall_arg_count
        ;;
        mov pr = mask, 0xff00         // define p8 .. p15
        ;;
        cmp.eq p7 = r0, r0            // set p7 to true
        ;;
// test for NaT bits in the input arguments
(p8)    cmp.eq.and p7 = r32, r32      // and type compare clears p7 if r32 is NaT
(p9)    cmp.eq.and p7 = r33, r33
(p10)   cmp.eq.and p7 = r34, r34
(p11)   cmp.eq.and p7 = r35, r35
(p12)   cmp.eq.and p7 = r36, r36
(p13)   cmp.eq.and p7 = r37, r37
(p14)   cmp.eq.and p7 = r38, r38
(p15)   cmp.eq.and p7 = r39, r39
(p7)    br.cond.sptk ok_arguments     // No NaTs found
;;
// p7 was cleared by at least one NaT argument
```

4.5 Context Switching

This section discusses context switching at the user and kernel levels.

4.5.1 User-level Context Switching

4.5.1.1 Non-local Control Transfers (setjmp/longjmp)

A non-local control transfer such as the C language `setjmp()`/`longjmp()` pair requires software to correctly handle the register stack and the RSE. The register stack provides the BSP application register which always contains the backing store address of the current GR32. This permits execution of a `setjmp()` without having to manipulate any register stack or RSE state. All register stack and RSE manipulation is postponed to the much less frequent `longjmp()`.

In `setjmp()` only the RSC, PFS and BSP application registers have to be preserved. This can be accomplished by reading these registers, and without having to disable the RSE. The preserved values will be referred to as `setjmp_rsc`, `setjmp_pfs`, and `setjmp_bsp` further on.

In `longjmp()` restoration of the appropriate register stack and RSE state is more involved, and software needs to take the following steps:

1. Stop RSE by setting RSC.mode bits to zero.
2. Read current BSPSTORE (referred to as `current_bspstore` further down).
3. Find `setjmp()`'s RNAT collection (`rnat_value`).
    a. Compute the backing store location of `setjmp()`'s RNAT collection as follows:

       ```
       rnat_collection_address{63:0} = setjmp_bsp{63:0} | 0x1F8
       ```

       The RNAT location is computed by setting bits{8:3} of `setjmp()`'s BSP to all ones. This is where `setjmp()`'s RNAT collection will have been spilled to memory.

    b. If `(current_bspstore > rnat_collection_address)`, then the required RNAT collection has already been spilled to the backing store.
    c. Otherwise if `(current_bspstore <= rnat_collection_address)`, the required RNAT collection is incomplete and is still contained in the register stack. To materialize the complete RNAT collection, flush the register stack to the backing store using a `flushrs` instruction.
    d. Finally, load `rnat_value` from `rnat_collection_address` in memory.
4. Invalidate the contents of the register stack as follows:
    a. Allocate a zero size register stack frame using the `alloc` instruction.
    b. Write RSC.loadrs field with all zeros and execute a `loadrs` instruction.
    c. Invalidate the ALAT using the `invala` instruction.
5. Restore `setjmp()`'s register stack and RSE state as follows:
    a. Write BSPSTORE with `setjmp_bsp`.
    b. Write RNAT with `rnat_value`.

   c. Write RSC with `setjmp_rsc`.

   d. Write PFS with `setjmp_bsp`.

6. Restore `setjmp()`'s return IP into BR7.
7. Return from `longjmp()` into `setjmp()`'s caller using `br.ret` instruction.

4.5.1.2 User-level Co-routines

The following steps need to be taken to execute a voluntary user-level thread switch.

1. Save all preserved register state of outgoing thread to memory stack. Refer to Section 4.1 for details on preservation of general and floating-point registers.
2. Preserve predicate, branch, and application registers.
3. Flush outgoing register stack to backing store, and switch to incoming thread's backing store as described in Section 6.11.3, "Synchronous Backing Store Switch" on page 2:148. This code sequence includes ALAT invalidation.
4. Switch thread memory stack pointers.
5. Restore incoming thread's predicate, branch, and application registers.
6. Restore incoming thread's preserved register state.

4.5.2 Context Switching in an Operating System Kernel

4.5.2.1 Thread Switch within the Same Address Space

To switch between different threads in the same address space the following steps are required:

1. Application architecture state associated with each thread (GRs, FRs, PRs, BRs, ARs) are saved and restores as if this were a user-level coroutine. This is described in Section 4.5.1.2.
2. Memory Ordering: to preserve correct memory ordering semantics the context switch routine needs to fence all memory references and flush cache (`fc`, `fc.i`) operations by executing a `sync.i` and `mf` instruction. More details on memory ordering are given in Section 2.3.

4.5.2.2 Address Space Switching

When an operating system switches address spaces it needs to perform the same steps as a same address space thread switch (described in the previous section). Additionally, however between the saves of the outgoing and the restores of the incoming process, the operating system context switch handler is required to:

1. Save the contents of the protection key registers associated with the outbound context, and then invalidate the protection key registers.
2. Save the default control register (DCR) of the outbound context (if the DCR is maintained on a per-process basis).
3. Save the region registers of the outbound address space.
4. Restore the region registers of the inbound address space.

5. Restore the default control register (DCR) of the inbound context (if the DCR is maintained on a per-process basis).
6. Restore the contents of the protection key registers associated with the inbound context.

§

Memory Management 5

This chapter introduces various memory management mechanisms of the Itanium architecture: region register model, protection keys, and the virtual hash page table usage models are described. This chapter also discusses usage of the architecture translation registers and translation caches. Outlines are provided for common TLB and VHPT miss handlers.

5.1 Address Space Model

The Itanium architecture provides a byte-addressable 64-bit virtual address space. The address space is divided into 8 equally-sized sections called regions. Each region is 2^{61} bytes in size and is tagged with a unique region identifier (RID). As a result, the processor TLBs can hold translations from many different address spaces concurrently, and need not be flushed on address space switches. The regions provide the basic virtual memory architecture to support multiple address space (MAS) operating systems.

Additionally, each translation in the TLB contains a protection key that is matched against a set of software maintained protection key registers. The protection keys are orthogonal to the region model and allow efficient object sharing between different address spaces. The protection key registers provide the basic virtual memory architecture to support single address space (SAS) operating systems.

5.1.1 Regions

For each of the eight regions, there is a corresponding region register (RR), which contains a RID for that region. The operating system is responsible for managing the contents of the region registers. RIDs are between 18 and 24 bits wide, depending on the processor implementation. This allows an Itanium architecture-based operating system to uniquely address up to 2^{24} address spaces each of which can be up to 2^{61} bytes in virtual size. An address space is made accessible to software by loading its RID into one of the eight region registers.

Address Translation: The upper 3 bits of a 64-bit virtual address (bits 63:61) identify the region to which the address belongs; these are called the virtual region number (VRN) bits. When a virtual address is translated to a physical address, the VRN bits select a region register which provides the RID used for this translation. Each TLB entry contains the RID tag bits for the translation it maps; these are matched against the RID bits from the selected region register when the TLB is looked up during address translation. Address translation only succeeds if the RID and VPN bits from the virtual address match the RID and VPN bits from the TLB entry. Note that the VRN bits are used only to select the region register, are not matched against the TLB entries.

Inserting/Purging of Translations: When a translation is inserted into the processor TLBs (either by software, or by the processor's hardware page walker), the VRN bits of the virtual address translation being inserted are used only to index the corresponding region register; they are not inserted into the TLB. Likewise, when software purges a translation from the processor's TLBs, the VRN bits of the address used for the purge are used only to index the corresponding region register and are not used to find a matching translation. Only the RID and VPN bits are used to find overlapping translations in the TLBs.

The fact that the VRN bits are not contained in the processor TLB allows the same address space (identified by a RID) to be referenced through any of the eight region registers. In other words, the combination of RID and VPN establishes a unique 85-bit virtual address, regardless of which VRN (and region register) was used to form the pair. Independence of VRN allows easy creation of temporary virtual mappings of an address space and can accelerate cross-address space copying as described in Section 5.1.1.3.

5.1.1.1 RID Management

Before a RID that has been used for one address space can be reused for another address space, all TLB entries relating to the first address space have to be purged. In general, this will require a complete flush of the TLBs of all processors in the system. This can be accomplished by performing an IPI to all processors and executing the ptc.e loop described in Section 5.2.2.2.2 on each processor in the TLB coherence domain.

A more efficient alternative, depending on the size of the defunct address space, might be to perform a series of `ptc.ga` operations on one processor to tear down just the translations used by the recycled RID. Some processor implementations support an efficient region-wide purge page size such that this can be accomplished with a single `ptc.ga` operation.

The frequency of these global TLB flushes can be reduced by using a RID allocation strategy that maximizes the time between use and reuse of a RID. For example, RIDs could be assigned by using a counter that is as wide as the number of implemented RID bits and that is incremented after every assignment. Only when the RID counter wraps around it is necessary to do a global TLB flush. After the flush the operating system can either remember the in-use RIDs or it can re-assign new RIDs to all currently active address spaces.

5.1.1.2 Multiple Address Space Operating Systems

Multiple address space (MAS) operating systems provide a separate address space for each process. Typically, only when a process is running is its address space visible to software.

The application view of the virtual address space in the MAS OS model is a contiguous 64-bit address space, though normally not all of this virtual address space is accessible by the application. At least one of the 8 regions must be used to map the OS itself so that the OS can handle interruptions and system services invoked by the application.

The OS chooses a region ID and a region (e.g. region 7) into which to map itself during the boot process and usually does not change this mapping after enabling address translation. The other seven regions may be used to map process-private code and data; code and data that are shared amongst multiple processes; to map large files; temporary mappings to allow efficient cross-address space copies (see Section 5.1.1.3); and, for operating systems which use it, the long format VHPT.

In a MAS OS, the RID bits act as an address space identifier or tag. For each process-private region, a unique RID is assigned to that process by the OS. If a process needs multiple process-private regions (e.g. the process requires a private 64-bit address space), the OS assigns multiple unique RIDs for each such region. Because each translation in the processor's TLBs is tagged with its RID, the TLBs may contain translations from many different address spaces (RIDs) concurrently. This obviates the need for the OS to purge the processor's TLBs upon an address space switch. When the OS performs a context switch from process A to process B, the OS need only remove process A's private RIDs from the CPU's region registers and replace them with process B's private RIDs.

5.1.1.3 Cross-address Space Copies in a MAS OS

The use of regions, region registers, and RIDs provides a mechanism for efficient address space-to-address space copies. Because translations are tied to RIDs and not to a particular static region, a MAS OS can easily copy a memory range from one address space to another by temporarily remapping the target memory location to another region. This remapping is accomplished simply by placing the RID to which the target location belongs into a different region register and then performing the copy from source to target directly.

For example, assume a MAS OS wishes to copy and 8-byte buffer from virtual address 0x0000000000A00000 of the currently executing process (process A) to virtual address 0x0000000000A00000 of another process (process B):

```
         movl r2 = (2 << 61)
         mov r3 = process_b_rid
         movl r4 = 0x0000000000A00000
         movl r5 = 0x4000000000A00000;;;   // reference process B through RR[2]
         mov rr[r2] = r3 ;;                // put process B RID into RR[2]
         srlz.d                            // serialize RR write
copyloop:
         ld8 r6 = [r4] ;;                  // read buffer from process A addr space
         st8 [r5] = r6                     // store buffer into process B addr
space
    (p4)br copyloop                        // loop until done
         mov r3 = original_rr2_rid ;;
         mov rr[r2] = r3 ;;                // restore RR[2] RID
         srlz.d                            // serialize RR write
```

When the OS switches to process B and places process B’s RID into RR[0] and resumes execution of process B, the process can reference the message via virtual address 0x0000000000A00000. Note that no new translations need to be created to make the sequence shown above work; because translations are tagged by RID and not by region, all existing translations for process B’s address space are visible regardless of which region the reference is made to, as long as the region register for that region contains the correct process B RID. Note that the sequence shown above is intended for illustrative purposes only; the OS may need to perform other steps as well to perform a cross-address space copy.

5.1.2 Protection Keys

The Itanium architecture provides two mechanisms for applying protection to pages. The first mechanism is the access rights bits associated with each translation. These bits provide privilege level-granular access to a page. The second mechanism is the protection keys. Protection keys permit domain-granular access to a page. These are especially useful for mapping shared code and data segments in a globally shared region, and for implementing domains in a single address space (SAS) operating system.

Protection key checking is enabled via the PSR.pk bit. When PSR.pk is 1, instruction, data, and RSE references go through protection key access checks during the virtual-to-physical address translation process.

All processors based on the Itanium architecture implement at least 16 protection key registers (PKRs) in a protection key register cache. The OS is responsible for maintaining this cache and keeping track of which protection keys are present in the cache at any given time.

Each protection key register contains the following fields:

- v – valid bit. When 1, this register contains a valid key, and is checked during address translation whenever protection keys are enabled (PSR.pk is 1).
- wd – write disable. When 1, write permission is denied to translations which match this protection key, even if the data TLB access rights permit the write.
- rd – read disable. When 1, read permission is denied to translations which match this protection key, even if the data TLB access rights permit the read.
- xd – execute disable. When 1, execute permission is denied to translations which match this protection key, even if the instruction TLB access rights give execute permission.
- key – protection key. An 18- to 24-bit (depending on the processor implementation) unique key which tags a translation to a particular protection domain.

When protection key checking is enabled, the protection key tagged to a referenced translation is checked against all protection keys found in the protection key register cache. If a match is found, the protection rights specified by that key are applied to the translation. If the access being performed is allowed by the matching key, the access succeeds. If the access being performed is not allowed by the matching key (e.g. instruction fetch to a translation tagged with a key marked 'xd'), a Protection Key Permission fault is raised by the processor. The OS may then decide whether to terminate the offending program or grant it the requested access.

If no match is found, a Protection Key Miss fault is raised by the processor, and the OS must insert the correct protection key into the PKRs and retry the access.

Protection keys can be used to provide different access rights to shared translations to each process. For example, assume a shared data page is tagged with a protection key number of 0xA. Two processes share this data page: one is the producer of the data on this page, and the other is only a consumer. When the producer process is running, the OS will insert a valid PKR with the protection key 0xA and the 'wd' and 'rd' bits cleared, to allow this process to both read and write this page. When the consumer process is running, the OS will insert a valid PKR with the protection key 0xA and the ‘rd’ bit cleared, to allow this process to read from the page. However, the ‘wd’ bit for this PKR will be set when the consumer process is running to prevent it from writing the page.

The processor hardware has no notion of which protection keys belong to which process. The only check the hardware performs is to compare the protection key from the translation to any valid protection keys in the PKR cache. On a context switch, the OS must purge any valid protection keys from the PKRs which would provide access rights to the switched-to context that are not allowed. The OS may purge an existing PKR by performing a move to PKR instruction with the same key as the existing PKR, but with the PKR valid bit set to 0.

Protection keys can be read from the processor’s data TLBs via the `tak` instruction. However, instruction TLB key values cannot be read directly. Software must keep track of these values in its own data structures.

5.1.2.1 Single Address Space Operating Systems

Processes in a single address space (SAS) OS all cohabit a global address space. SAS operating systems running on a processor based on the Itanium architecture can view the RID bits as effectively extending the single virtual address space to between 79 and 85 bits (depending on the number of RID bits implemented by the processor). This address space is then divided into between 2^{18} and 2^{24} 61-bit regions, up to eight of which may be accessed concurrently.

Note that there is no “SAS OS” or “MAS OS” mode in the Itanium architecture. The processor behavior is the same, regardless of the address space model used by the OS. The difference between a SAS OS and a MAS OS is one of OS policy: specifically how the RIDs and protection keys are managed by the OS, and whether different processes are permitted to share RIDs for their private code and data. Multiple, unrelated processes in a SAS OS may share the same RID for their private pages; it is the responsibility of the OS to use protection keys and the protection key registers (PKRs) to enforce protection. In a MAS OS, the unique per-process RIDs enforce this protection.

Hybrid SAS/MAS models that combine unique RIDs for process-private regions and shared RIDs with protection keys for per-page memory protection in shared regions are also possible.

5.2 Translation Lookaside Buffers (TLBs)

All processors based on the Itanium architecture implement one or more translation lookaside buffers (TLBs) for fast virtual-to-physical address translation. The architecture provides instructions for managing instruction and data TLBs as separate structures.

Both the instruction and data TLBs are further divided into a set of translation registers (TRs), which are managed exclusively by software and are “locked down” to pin critical address translations (e.g. kernel memory); and a set of translation cache entries (TCs), which can be managed by both software and the processor hardware. The TRs are divided into slots, each of which are individually addressable on insertion by software.

The TCs are treated as a set associative cache and are not addressable by software. The TC replacement policy is determined by software. All processor models implement at least 8 instruction and 8 data TRs, and at least 1 instruction and 1 data TC entry.

Software inserts translations into the TLBs via insertion instructions. There are four variants of insertion instructions. `itr.i` and `itr.d` insert a translation into the specified instruction or data TR slot, respectively. `itc.i` and `itc.d` insert a translation into a hardware-selected instruction or data TC entry, respectively.

Software TR purge instructions also distinguish between the instruction and data TRs (`ptr.i`, `ptr.d`). TC purge instructions do not.

5.2.1 Translation Registers (TRs)

Once a translation is inserted by software into a TR, it remains in that TR until either the translation is overwritten by software, or the translation is purged. TRs are used by the OS to pin critical address translations; all memory references made to a TR translation will always hit the TLB and will never cause the processor's hardware page walker to walk the VHPT or raise a fault. Examples of memory areas that the OS might cover with one or more TRs are the Interruption Vector Table, critical interruption handlers not contained completely in the Interruption Vector Table, the root-level page table entries, the long format VHPT, and any other non-pageable kernel memory areas.

Two address translations are said to overlap when one or more virtual addresses are mapped by both translations. Software must ensure that translations in an instruction TR never overlap other instruction TR or TC translations; likewise, software must ensure that translations in a data TR never overlap other data TR or TC translations. If an overlap is created, the processor will raise a Machine Check Abort.

The processor hardware will never overwrite or purge a valid TR. TRs that are currently unused may be used by the processor hardware as extra TC entries, but if software subsequently inserts a translation into an unused a TR, the TC translation will be purged when the insertion is executed.

5.2.1.1 TR Insertion

To insert a translation into a TR, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[1]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.
4. Place the slot number of the instruction or data TR into which the translation is be inserted into a general register.
5. Place the base physical address of the translation into another general register.

[1] The upper 3 bits (VRN) of this address specify a region register whose contents are inserted along with the rest of the translation. See Section 5.1.1 for details.

6. Using the general registers from steps 4 and 5, execute the `itr.i` or `itr.d` instruction.

A data or instruction serialization operation must be performed after the insert (for `itr.d` or `itr.i`, respectively) before the inserted translation can be referenced.

Software may insert a new translation into a TR slot already occupied by another valid translation. However, software must perform a TR purge to ensure that the overwritten translation is no longer present in any of the processor's TLB structures.

Instruction TR inserts will purge any instruction TC entries which overlap the inserted translation, and may purge any data TC entries which overlap it. Data TR inserts will purge any data TC entries which overlap the inserted translation and may purge any instruction TC entries which overlap it.

Software may insert the same (or overlapping) translation into both the instruction TRs and the data TRs. This may be desirable for locked pages which contain both code and data, for example.

5.2.1.2 TR Purge

To purge a TR from the TLBs, software performs the following steps:

1. Place the base virtual address of the translation to be purged into a general register.[1]
2. Place the address range in bytes of the purge into bits {7:2} of a second general register.
3. Using these two GRs, execute the `ptr.d` or `ptr.i` instruction.

A data or instruction serialization operation must be performed after the purge (for `ptr.d` or `ptr.i`, respectively) before the translation is guaranteed to be purged from the processor's TLBs.

Note: The TR purge instruction operates independently of the slot into which the translation was originally inserted.

A `ptr.d` instruction will never purge an overlapping translation in an instruction TR, but may purge an overlapping translation in an instruction TC; likewise, a `ptr.i` instruction will never purge an overlapping translation in a data TR, but may purge an overlapping translation in a data TC.

A TR purge does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

5.2.2 Translation Caches (TCs)

The TC array acts as a cache of the dynamic working set for data and instruction translations. It is managed by software (via `itc` and `ptc` instructions) and, optionally by hardware, if the processor provides a hardware page walker (HPW) and the walker is enabled. See Section 5.3 below.

[1] The upper 3 bits (VRN) of this address specify a region register whose contents are used as part of the translation to be purged. See Section 5.1.1 for details.

The size, associativity, and replacement policy of the TC array are implementation-dependent. With the exception of the forward progress rules defined in Section 4.1.1.2, "Translation Cache (TC)" on page 2:49, software cannot depend on the existence or life-span of a TC translation, as a TC entry may be replaced or invalidated by the hardware at any time.

5.2.2.1 TC Insertion

To insert a TC entry, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[1]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.
4. Place the base physical address of the translation into a general register.
5. Using the general register from step 4, execute the `itc.i` or `itc.d` instruction.

A data or instruction serialization operation must be performed after the insert (for `itc.d` or `itc.i`, respectively) before the inserted translation can be referenced.

Instruction TC inserts always purge overlapping instruction TCs and may purge overlapping data TCs. Likewise, data TC inserts always purge overlapping data TCs and may purge overlapping instruction TCs.

5.2.2.2 TC Purge

There are several types of TC purge instructions. Unlike the other TLB management instructions, the TC purge instructions do not distinguish between instruction and data translations; they will purge any matching translations in either the data or instruction TC arrays.

5.2.2.2.1 ptc.l

The most basic TC purge is the local TC purge instruction (`ptc.l`). To purge a TC from the local processor TLBs, software performs the following steps:

1. Place the base virtual address of the translation to be purged into a general register.[2]
2. Place the address range in bytes of the purge into bits {7:2} of a second general register.
3. Using these two GRs, execute the `ptc.l` instruction.

[1] The upper 3 bits (VRN) of this address specify a region register whose contents are inserted along with the rest of the translation. See Section 5.1.1 for details.

[2] The upper 3 bits (VRN) of this address specify a region register whose contents are used as part of the translation to be purged. See Section 5.1.1 for details.

A data or instruction serialization operation must be performed after the `ptc.l` before the translation is guaranteed to be no longer visible to the local data or instruction stream, respectively.

The `ptc.l` instruction does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

The `ptc.l` instruction ensures that all prior stores are made locally visible before the actual purge operation is performed. Consider the following code sequence:

```
st8 [VHPT] = <new_translation>
ptc.l <old_translation>
srlz.i
```

The `ptc.l` instruction will purge the translation only after the local store update is seen. If there was a hardware-initiated VHPT walk for the same translation, it would either insert the *old_translation* in the TLB before the `ptc.l` executes and then get purged by the `ptc.l`, or insert the *new_translation* after both the local store update and `ptc.l` purge are complete.

5.2.2.2.2 ptc.e

To purge all TC entries from the local processor's TLBs, software uses a series of `ptc.e` instructions. Software must call the PAL_PTCE_INFO PAL routine at boot time to determine the parameters needed to use the `ptc.e` instruction. Specifically, PAL_PTCE_INFO returns:

- tc_base – an unsigned 64-bit integer denoting the beginning address to be used by the first `ptc.e` instruction in the purge loop.
- tc_counts – two unsigned 32-bit integers packed into a 64-bit parameter denoting the loop counts of the outer and inner purge loops. count1 (outer loop) is contained in bits {63:32} of the parameter, and count2 (inner loop) is contained in bits {31:0} of the parameter.
- tc_strides – two unsigned 32-bit integers packed into a 64-bit parameter denoting the loop stride of the outer and inner purge loops. stride1 (outer loop) is contained in bits {63:32} of the parameter, and stride2 (inner loop) is contained in bits {31:0} of the parameter.

Software then executes the following sequence:

```
disable_interrupts();
addr = tc_base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
        ptc.e addr;
        addr += stride2;
    }
    addr += stride1;
}
enable_interrupts();
```

A data or instruction serialization operation must be performed after the sequence shown above before the translations are guaranteed to be no longer visible to the local data or instruction stream, respectively.

The `ptc.e` instruction does not modify the page tables nor any other memory location, nor does it affect the TLB state of any processor other than the one on which it is executed.

5.2.2.2.3 ptc.g, ptc.ga

The Itanium architecture supports efficient global TLB shootdowns via the ptc.g and ptc.ga instructions. These instructions obviate the need for performing inter-processor interrupts to maintain TLB coherence in a multiprocessor system. A TLB coherence domain is defined as a group of processors in a multiprocessor system which maintain TLB coherence via hardware.

For the remainder of this section, **`ptc.g`** refers to both the `ptc.g` and `ptc.ga` instructions, except where otherwise noted.

The number of `ptc.g` operations that can be in progress at any time is implementation dependent, and can be determined from the *max_purges* return parameter of PAL_VM_SUMMARY. Attempting to execute more than the maximum allowed number of simultaneous `ptc.g` purges will have undefined effects, including possibly raising a Machine Check Abort on one or more processors. Software should implement some semaphoring mechanism to ensure that not more than the maximum `ptc.g` purges allowed are in flight at any one time.

A `ptc.g` instruction is a release operation; all memory references that precede a `ptc.g` in program order are made visible to all other processors before the `ptc.g` is made visible. To guarantee visibility of the `ptc.g` prior to a particular point in program execution, software must use another release operation or a memory fence.

To purge a translation from all TLBs in the coherence domain, software performs the following steps:

1. Acquire the semaphore.
2. Place the base virtual address of the translation to be purged into a general register.
3. Place the address range in bytes of the purge into bits {7:2} of a second general register.
4. Using these two GRs, execute the ptc.g instruction. Note that the `ptc.g` instruction must be followed by a stop.
5. Release the semaphore.

Global purges can be batched together by performing multiple `ptc.g` instructions prior to releasing the lock.

A data or instruction serialization operation must be performed after the sequence shown above before the translations are guaranteed to be no longer visible to the local data or instruction stream, respectively. To guarantee the translations are no longer visible on remote processors, a release operation or memory fence instruction is required after the `ptc.g` instruction.

The `ptc.g` instruction does not modify the page tables nor any other memory location. It affects both the local and all remote TC entries in the TLB coherence domain. It does not remove translations from either local or remote TR entries. If a ptc.g overlaps a translation contained in a TR on the local processor, the local processor will raise a Machine Check Abort; if the `ptc.g` overlaps a translation contained in a TR on any remote processor in the coherence domain, no Machine Check Abort is raised.

The ptc.ga variant of the global purge instruction behaves just like the ptc.g variant, but it also removes any ALAT entries which fall into the address range specified by the global shootdown from all remote processors' ALATs. The ptc.ga variant is intended to be used whenever a translation is remapped to a different physical address to ensure that any stale ALAT entries are invalidated. Note that the ptc.ga is not guaranteed to affect the issuing processor's ALAT; processor implementations may optionally remove matching entries from the local ALAT, therefore software must perform a local ALAT invalidation via the invala instruction on the processor issuing the ptc.ga to ensure the local ALAT is coherent.

Note that processors based on the Itanium architecture may support one or more implementation-dependent purge sizes; some implementations may include a region-wide purge. The PAL_VM_PAGE_SIZE firmware call returns the supported page sizes for purges for a particular processor implementation. Refer to Section 11.10.1, "PAL Procedure Summary" for details. When software wishes to purge an address range that is much larger than the largest supported purge size from all TCs in the coherence domain, performance may be enhanced by issuing inter-processor interrupts to all processors and using the ptc.e loop described in Section 5.2.2.2.2 on each processor, instead of issuing many ptc.g instructions from one processor.

ptc.g instructions do not apply to processors outside the coherence domain of the processor issuing the ptc.g instruction. Systems with multiple coherence domains must use a platform-specific method for maintaining TLB coherence across coherence domains.

5.3 Virtual Hash Page Table

The Itanium architecture defines a data structure that allows for the insertion of TLB entries by a hardware mechanism. The data structure is called the "virtual hash page table" (VHPT) and the hardware mechanism is called the VHPT walker.

Unlike the IA-32 page tables, the Itanium VHPT itself is virtually mapped, i.e. VHPT walker references can take TLB faults themselves. Virtual mapping of the page tables is needed because the page tables for 2^{64} address space are quite large and typically do not fit into physical memory.

The Itanium architecture prescribes the format of a leaf-node page table entry (PTE) seen by the VHPT walker, but does not impose an OS page table data structure itself. As summarized in Table 5-1, the architecture support two different VHPT formats:

- **Short** format uses 8-byte PTEs, and is a linear page table. The short format VHPT does not contain protection key information (there are not enough PTE bits for that). Short format is a per-region linear page table, i.e. the PTEs and hash function are independent of the RID. The short format prefers use of a self-mapped page table. The short format VHPT is an efficient representation for address spaces that contain only a few large clusters of pages, like the text, data, and stack segments of applications running on a MAS operating system.
- **Long** format uses 32-byte PTEs, and is a hashed page table. The hash function embedded in hardware. The long format supports protection keys and the use of multiple page sizes in a region. The long format hash and tag functions incorporate the RID, and allows multiple address space translations to be present in the same VHPT. The long format is expected to be used either as a cache of the real OS page

tables, or as a primary page table with collision chains. The long format VHPT is a much better representation for address spaces that are sparsely populated, since the short format VHPT has a linear layout and would consume a large amount of memory. Single address space operating systems may prefer the long format VHPT for this reason.

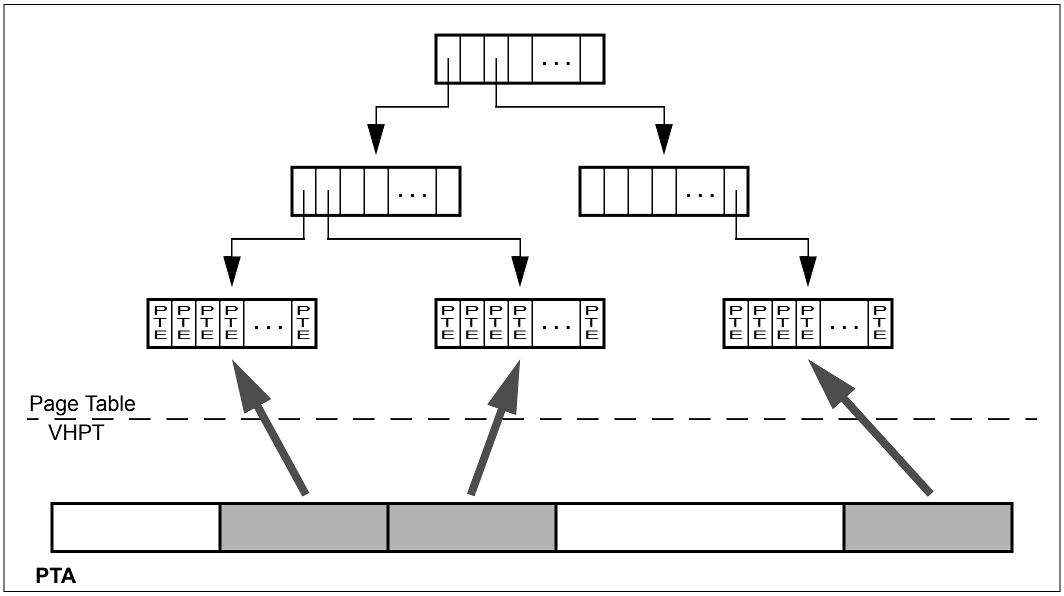
Table 5-1. Comparison of VHPT Formats

Attribute	Short Format	Long Format
Entry Size	8 Byte	32 Byte
Lookup	Linear	Hashed
Protection Keys	No	Yes
Page Size	per region	per entry

5.3.1 Short Format

The short format VHPT is a per-region linear table that contains translation entries for every page in the region's virtual address space. This makes the VHPT very large, but since the VHPT itself lives in virtual address space only those parts of the VHPT that actually contain valid translation entries have to be present in physical memory. If the operating system's page table is a hierarchical data structure and the last level of the hierarchy is a linear list of translations, the VHPT can be mapped directly onto the page table as shown in Figure 5-1.

Figure 5-1. Self-mapped Page Table



If the VHPT walker tries to access a location in the VHPT for which no translation is present in the TLB, a VHPT Translation fault is raised. The original address for which the VHPT walker was trying to find an entry in the VHPT is supplied to the fault handler in the IFA register. The fault handler can use this address to traverse the page table and insert a translation into the TLB that maps the address the VHPT walker tried to access (in IHA) to the page that contains the corresponding leaf page table.

5.3.2 Long Format

The long format VHPT is organized as a hash table which contains a subset of all translation entries. The long format VHPT entries contain a 8-byte field that is ignored by the VHPT walker and can be used by the operating system to link VHPT entries to software-walkable hash collision chains if it uses the VHPT as its primary page table. The size of the long format VHPT is usually kept small enough to keep a mapping for it in one of the translation registers (TRs), so it is not necessary to handle VHPT translation faults.

The long format hash algorithm is based on the per-region preferred page size, but a translation for a larger page can still be entered into the VHPT by subdividing the large page into multiple smaller pages with the preferred page size and placing an entry for the large page at all VHPT locations that correspond to the smaller pages.

5.3.3 VHPT Updates

Visibility of VHPT updates to a VHPT walker on another processor follows the rules outlined in Section 4.1.7, "VHPT Environment" on page 2:67. Since a global TLB purge has release semantics, prior modifications to the VHPT will be visible to operations that occur after the TLB purge operation.

Atomic updates to short format VHPT entries can easily be done through 8-byte stores. For atomic updates of long format VHPT entries, the "ti" flag in bit 63 of the tag field can be utilized as follows:

- Set the "ti" bit to 1.
- Issue a memory fence.
- Update the entry.
- Clear the "ti" bit through a store with release semantics.

5.4 TLB Miss Handlers

The Itanium architecture enables lightweight TLB fault handlers by providing individual entry points for different excepting conditions and by pre-setting the translation insertion registers for the various types of TLB faults. The following subsections list the typical steps for resolving each kind of fault.

5.4.1 Data/Instruction TLB Miss Vectors

These faults occur when the data or instruction TLB required for a data access or instruction fetch is not found in the processor TLBs, the VHPT walker is enabled, and:

- Either the VHPT walker aborted the walk (for any reason and at any time), or
- The VHPT walker found the translation but the insert failed (due to tag mismatch in the long format or badly formed PTE), or
- The walker is not implemented on this processor.

There is a separate vector for each fault type (data and instruction).

Since the VHPT walker may abort a walk at any time and raise these faults, software must always be able to handle all TLB faults, even when the VHPT walker is enabled. Upon entry to these fault handlers, the IHA, ITIR, and IFA control registers are initialized by the hardware as follows:

- IHA – contains the virtual address of the hashed page table address corresponding to the reference which raised the fault.
- ITIR – contains the default translation information for the reference which raised the fault (i.e. for the virtual address contained in IFA). The access key field is set to the region ID from the RR corresponding to the faulting address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the faulting address.
- IFA – the virtual address of the bundle (for instruction faults) or data reference (for data faults) which missed the TLB.

The fault handler for a short format VHPT performs the following steps, at a minimum, to handle the fault:

1. Move IHA into a general register, chosen by convention to match the register expected by the nested TLB fault handler.
2. Perform an 8-byte load into another general register from the address contained in this general register to grab the VHPT entry. Note that the format of these first 8 bytes is identical to the format required for TLB insertion. If the VHPT is not mapped by a TR, software must be prepared to handle a nested TLB fault when performing this load.
3. Using the general register from step 2 that holds the contents of the VHPT entry, perform a TC insert (`itc.i` for instruction faults, `itc.d` for data faults).
4. In an MP environment, reload the VHPT entry from step 2 into a third general register and compare the value to the one loaded in step 2. If the values are not the same, then the VHPT has been modified by another processor between steps 2 and 3, and the entry will have to be re-inserted. In this case, purge the entry just inserted using a `ptc.l` instruction. The fault will re-occur after the `rfi` in step 5 (unless the VHPT walker succeeds on the next TLB miss) and the fault handler will re-attempt the insertion. (Uniprocessor environments may skip this step.)
5. `rfi`.

For a long format VHPT, additional steps are required to load bytes 16-23 of the VHPT entry and check for the correct tag (the correct tag for the reference can be generated using the ttag instruction). If the tags do not match, this indicates a VHPT collision, and the handler must proceed to walk the operating system's collision chain manually to find the correct entry. The handler may then choose to swap places between the correct entry and the VHPT entry. Note that the pointers for a collision chain can be stored in bytes 24-31 of the VHPT entry format since these bytes are ignored by the VHPT walker.

If the default page size and key are not sufficient, the handler must also perform additional steps to load the correct page size and key into the ITIR register before performing the TC insert in step 3 of the sequence shown above.

5.4.2 VHPT Translation Vector

Processors based on the Itanium architecture does not perform recursive TLB hardware page walks. Since the VHPT is itself a virtually addressed structure, each reference performed by the walker itself goes through the TLBs and may miss. These faults are raised when the VHPT walker is enabled, but the walker misses the TLBs when attempting to service a TLB miss caused by the program.

There is a separate vector for each fault type (data and instruction).

Upon entry to this fault handler, the IHA, IFA, and ITIR control registers are initialized by the hardware as follows:

- IHA – contains the virtual address of the hashed page table address corresponding to the reference which raised the fault.
- ITIR – contains the default translation information for the VHPT address which missed the TLBs (i.e. for the virtual address contained in IHA). The access key field is set to the region ID from the RR corresponding to the VHPT address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the VHPT address.
- IFA – contains the original faulting address that the VHPT walker was attempting to resolve.

The fault handler for a short format VHPT performs the following steps, at a minimum, to handle the fault:

1. Move the IHA register into a general register.
2. Perform a thash instruction using the general register from step 1 This will produce, in the target register, the VHPT address of the VHPT entry that maps the VHPT entry corresponding to the original faulting address (i.e. the address in IFA).
3. Using the target general register of the thash from step 2 as the load address, perform an 8-byte load from the VHPT. Note that the format of these first 8 bytes is identical to the format required for TLB insertion. Software must be prepared to take a nested TLB fault if this load misses the TLBs.
4. Move the IHA value from the general register written in step 1 into the IFA register.
5. Using the general register from step 3 that holds the contents of the VHPT entry, perform a data TC insert using the `itc.d` instruction. (VHPT references always go through the data TLBs.)
6. In an MP environment, reload the VHPT entry from step 3 into a different general register and compare the value to the one loaded in step 3. If the values are not the same, then the VHPT has been modified by another processor between steps 3 and 4, and the entry will have to be re-inserted. In this case, purge the entry just inserted using a `ptc.l` instruction. The fault will re-occur after the `rfi` in step 7 (unless the VHPT walker succeeds on the next TLB miss) and the fault handler will re-attempt the insertion. (Uniprocessor environments may skip this step.)
7. `rfi`.

For a long format VHPT, additional steps are required to load bytes 16-23 of the VHPT entry and check for the correct tag; see Section 5.4.1 for more details.

A separate structure other than the VHPT may be used to back VHPT translations, in which case the handler would not use the thash instruction to generate the address of the translation mapping the VHPT entry corresponding to the original faulting address. Instead, the handler would use the operating system's own mechanism for finding VHPT back-mappings. Other schemes for handling VHPT misses are also possible, but are beyond the scope of this document.

5.4.3 Alternate Data/Instruction TLB Miss Vectors

These faults are raised when an instruction or data reference misses the processor's TLBs and the VHPT walker is not enabled for the faulting address, i.e. TLB misses are handled entirely in software. Operating systems which do not wish to use the VHPT walker can disable the walker and use these fault vectors for software TLB fill handlers. The OS may also choose to enable the walker on a per-region basis and use these vectors to handle misses in regions where the walker is disabled.

Upon entry to these fault handlers, the IFA and ITIR registers are initialized by the hardware as follows:

- ITIR – contains the default translation information for the reference which raised the fault (i.e. for the virtual address contained in IFA). The access key field is set to the region ID from the RR corresponding to the faulting address. The page size field is set to the preferred page size (RR.ps) from the RR corresponding to the faulting address.
- IFA – the virtual address of the bundle (for instruction faults) or data reference (for data faults) which missed the TLB.

The OS needs to lookup the PTE for the faulting address in the OS page table, convert it to the architected insertion format (see Section 4.1.1.5, "Translation Insertion Format"), and insert it into the TLB. The mechanism used to handle these faults is OS specific and is beyond the scope of this document.

5.4.4 Data Nested TLB Vector

To enable efficient handling of software TLB fills, the Itanium architecture provides a dedicated Data Nested TLB fault vector. The Data Nested TLB fault handler is intended to be used by the Data TLB fault handler, which allows the OS to page the page tables themselves. When PSR.ic is 0, any data reference that misses the TLB and would normally raise a Data TLB Miss fault (e.g. a load performed by the Data TLB fault handler to the page tables) will vector to the Data Nested TLB fault handler instead. Because IFA is not updated when PSR.ic is 0, the Data Nested TLB fault handler must get the faulting address from the general register used as the load address in the Data TLB fault handler[1]. Unlike other nested interruptions, the hardware does *not* update ISR when a Data Nested TLB fault is delivered.

[1] This requires a register usage convention between all TLB miss handlers and the Data Nested TLB miss handler.

The processor will not deliver a Data Nested TLB fault when PSR.ic is in-flight; Data Nested TLB faults are only delivered when PSR.ic is 0. If PSR.ic is in-flight, any data references which miss the TLB and trigger a fault will raise a Data TLB fault, and the processor will set ISR.ni to 1.

5.4.5 Dirty Bit Vector

The operating system is expected to lookup the PTE for the faulting address in the OS page table and load the PTE into a general register r_x. It can then set the "dirty" bit in r_x and write the updated PTE back to the page table. To continue execution, the OS must insert the updated PTE into the data TLB or update the PTE memory image and let the VHPT walker perform the insertion.

5.4.6 Data/Instruction Access Bit Vector

The operating system is expected to lookup the PTE for the faulting address in the OS page table and load the PTE into a general register r_x. It can then set the "access" bit in r_x and to continue execution, the OS must either:

- Write the updated PTE back to the page table, and have the VHPT walker pick it up, or
- Insert the updated PTE into the TLB using `itc.i` r_x for instruction pages, and `itc.d` r_x for data pages, or
- Step over the instruction/data access bit fault by setting the IPSR.ia or IPSR.da bits prior to performing an `rfi`.

5.4.7 Page Not Present Vector

Forward the fault to the operating system's virtual memory subsystem.

5.4.8 Data/Instruction Access Rights Vector

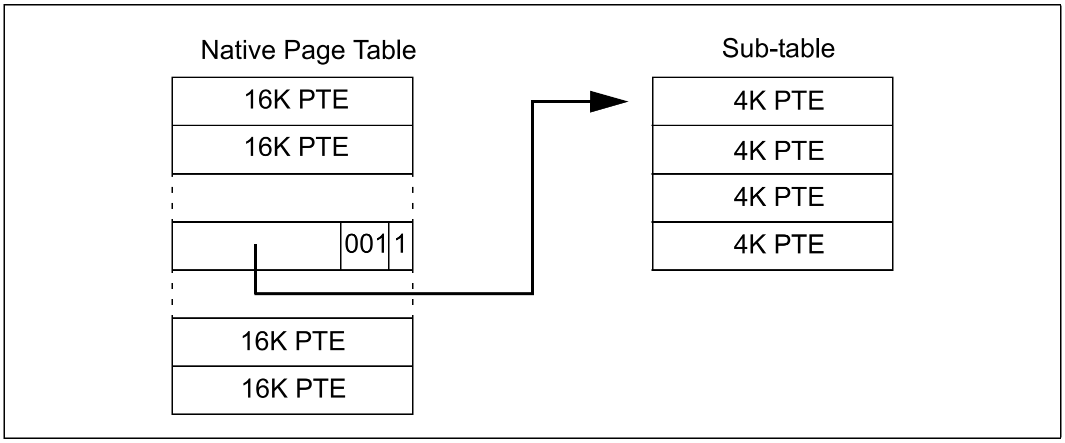
Forward the fault to the operating system's virtual memory subsystem.

5.5 Subpaging

The native page size an Itanium architecture-based operating system will choose for its page tables is likely be larger than the architectural minimum page size of 4 KB. Some legacy IA-32 applications, however, expect a page protection granularity of 4 KB. The following technique allows support for these applications with minimal impact on the native, larger page size paging mechanism.

A special type of entry is used in the native page table to mark pages that are subdivided into smaller 4 KByte units. The entry must have its memory attribute field set to the architecturally "software reserved" encoding (binary 001), and it carries a pointer to an array of 4 KB subentries in its most significant 59 bits. An example using a native page size of 16 KB is shown in Figure 5-2. The use of the "software reserved" memory attribute prevents the VHPT walker from attempting to insert the entry into the TLB.

Figure 5-2. Subpaging



When one of the subdivided pages is referenced and does not have a translation in the TLB, a TLB miss will occur. The handler for this fault can then use the faulting address to calculate the appropriate offset into the sub-table and insert the corresponding 4KByte PTE into the TLB.

Some care is required to ensure forward progress for IA-32 instructions. Each IA-32 instruction can reference up to 8 distinct memory pages during its execution (see also Section 10.6.3, "IA-32 TLB Forward Progress Requirements"). This means that the fault handler not only has to insert the PTE for the current fault into the TLB, but also the PTEs for up to seven faults that occurred before, if these faults originate from the same IA-32 instruction. This can be accomplished by maintaining a buffer for the most recent faulting IIP and for the parameters of up to 7 TLB insertions. If a TLB fault occurs while executing in IA-32 mode and the IIP matches the most recent IIP, all TLB insertions in the buffer have to be repeated and the parameters for the new TLB fault must be added to the buffer. Otherwise, the buffer can be cleared out and the most recent IIP can be updated. The buffer also has to be cleared out when a TLB purge occurs.

§

Runtime Support for Control and Data Speculation 6

An Itanium architecture-based operating system needs to handle exceptions generated by control speculative loads (`ld.s` or `ld.sa`), data speculative loads (`ld.a`) and architectural loads (`ld`) in different ways.

Software does not have to worry about control or data speculative loads potentially hitting uncacheable memory with side-effects, since `ld.s`, `ld.sa`, and `ld.a` instructions to non-speculative memory are always deferred by the processor for details refer to Section 4.4.6, "Speculation Attributes" on page 2:79. As a result, compilers can freely use control and data speculation to all program variables.

Control speculative loads require special exception handling and the Itanium architecture provides a variety of deferral mechanisms for handling of control speculative exception handling. This is discussed in Section 6.1.

The Itanium architecture supports different control speculation recovery models. These are discussed in Section 6.2.

Handling of exceptions caused by architectural and data speculative loads is the same, except for emulation of unaligned data speculative references, which require special unaligned emulation handling. This is discussed in Section 6.3.1.

6.1 Exception Deferral of Control Speculative Loads

Exceptions that occur on control speculative loads (`ld.s` or `ld.sa`) can be handled by the operating system in different ways. The operating system can configure a processor based on the Itanium architecture in three ways:

- Hardware-Only Deferral: automatic hardware deferral of all control speculative exceptions. In this case, the processor hardware will always defer excepting control speculative loads without invoking the operating system.
- Combined Hardware/Software Deferral: automatic deferral of some control speculative exceptions, but deliver others to software. In this case, some exceptions will result in hardware deferral as described above, other exceptions will be reported to the operating system. The operating system fault handlers can identify that an exception has been caused by a control speculative load (ISR.sp will be 1). Furthermore, OS handlers can software-defer an exception on a control speculative load by setting IPSR.ed to 1 prior to `rfi`-ing back to the `ld.s` or `ld.sa`. This allows an operating system to service "cheap" non-fatal exceptions (e.g. simple TLB misses), while software-deferring both "expensive" non-fatal (e.g. page faults) as well as fatal exceptions (e.g. non-recovery protection violation).
- Software-Only Deferral: processor is configured to deliver all control speculative exceptions to software. In this case, operating system software handles all non-fatal control speculative exceptions, and software-defers all fatal control speculative exceptions.

Details on these three models are discussed in the next three sections as well as in Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:105.

6.1.1 Hardware-only Deferral

Hardware only deferral is configured by setting all speculation deferral bits in the DCR register (dd, da, dr, dx, dk, dp and dm) to 1. All excepting control speculative loads are automatically deferred by the processor. As a result, all excepting control speculative loads that hit non-fatal exceptions, e.g. a TLB miss or a page fault, will be deferred by the processor hardware, and will cause speculation recovery code to be invoked. This can cause speculation recovery code to be invoked more often than strictly necessary.

6.1.2 Combined Hardware/Software Deferral

Setting of a DCR deferral bit to 1 results in hardware deferral by the processor, whereas clearing of a deferral bit causes exceptions to be delivered to software. The operating system may want to configure the processor to deliver control speculative exceptions to its handlers for certain non-fatal faults such as TLB misses or protection key misses. Early handling of these exceptions avoids unnecessary invocation of speculation recovery code, and the associated performance penalty. This is especially useful for exceptions handlers whose overhead is small. Note that handlers will also be invoked for excepting control speculative loads that have been hoisted from not taken paths, and therefore are not needed. As a result, software handling of control speculative exceptions is recommended only for statistically infrequent light weight fault handlers such as TLB miss or protection key miss handlers. If, while handling the exception, the operating system determines that this instance of the exception may require too much effort, e.g. a TLB miss turns out to be a page fault, the handler still has the choice of software-deferring the exception.

6.1.3 Software-only Deferral

Software only deferral is configured by clearing all speculation deferral bits in the DCR register (dd, da, dr, dx, dk, dp and dm) to 0. Control speculative loads that hit any Debug, Access Bit, Access Rights, Key Permissions, Key Miss, or Not Present fault, or that suffer a TLB miss or a VHPT Translation fault will be delivered to software.

6.2 Speculation Recovery Code Requirements

As described by Table 6-1, code generators for the Itanium architecture are not always required to generate speculation recovery code for all forms of speculation. Compilers and operating systems can collaborate to provide two models for handling of recovery from failed control speculation:

- ITLB.ed=1 (application with recovery code – the default): The compiler generates appropriate recovery code for all `ld.s` instructions, as well as for `ld.sa` and `ld.a` instructions that have speculatively executed uses. Speculation failure of `ld.sa` and `ld.a` instructions that have no speculatively executed uses can be recovered by a `ld.c` instruction, and hence do not require recovery code. The operating system may defer non-fatal exceptions.

- ITLB.ed=0 (no control speculative recovery code): The compiler generates recovery code only for ld.sa and ld.a instructions that have speculatively executed uses. Speculation failure of `ld.sa` and `ld.a` instructions that have no speculatively executed uses can be recovered by a `ld.c` instruction, and hence do not require recovery code. Speculation failure of `ld.s` instructions does not require recovery code, because, in this model, the operating system must guarantee that only fatal exceptions will be deferred. This requires software-only deferral of all potential non-fatal exceptions. The motivation for this model is that the absence of `chk.s` instructions and their associated recovery code may make for shorter and more compact in-line code, especially in loops with tight instruction schedules.

Table 6-1. Speculation Recovery Code Requirements

Usage Model	OS May Defer Non-fatal Exceptions on Control Speculative Loads (ITLB.ed=1)	OS Must Not Defer Non-fatal Exceptions on Control Speculative Loads (ITLB.ed=0)
No Speculative Load Uses		
`ld.s`	Recovery code required; Invoked by `chk.s` or non-speculative use of speculative value recovers from failed control speculation.	No recovery code required; OS handles all non-fatal exceptions speculatively.
`ld.sa,ld.a`	No recovery code required; `ld.c` recovers from failed data speculation.	
With Speculative Load Uses		
`ld.s`	Recovery code required; invoked by `chk.s` or non-speculative use of speculative value recovers from failed control speculation.	No recovery code required; OS handles all non-fatal exceptions speculatively.
`ld.sa,ld.a`	Recovery code required; `chk.a` recovers from failed data speculation.	

Presence or lack of control speculation recovery code is communicated from the compiler and the runtime system to the operating system by marking the code page's page table entry ed-bit appropriately (this bit is referred to as ITLB.ed). When ITLB.ed is 1, the operating system will expect recovery code to be present; when ITLB.ed is 0 no recovery code is expected. When a control speculative load takes an exception, the code page's ITLB.ed bit is copied into ISR.ed and is made available to the operating system exception handler. Furthermore, a set ISR.sp bit indicates that an exception was caused by a control speculative load.

6.3 Speculation Related Exception Handlers

6.3.1 Unaligned Handler

Misaligned control and data speculative loads, as well as architectural loads, are not required to be handled by the processor. As a result, the operating system's unaligned reference handler has to be prepared to emulate such misaligned memory references, especially in cases where the application has not provided any recovery code (see Section 6.2 for details). Furthermore, misaligned data speculative loads (`ld.sa` or `ld.a`) must be forced failed by the unaligned emulation handler, because the ALAT cannot track all sizes of misalignment for store conflict detection.

The following pseudo code outlines the basic steps for an unaligned reference handler:

1. Ensure that only ISR.r is 1, and that ISR.w, ISR.x, and ISR.na are 0.
2. Inspect the ISR.sp and ISR.ed. If both are 1, then defer this control speculative load by setting IPSR.ed and `rfi`-ing.
3. Crack the instruction opcode to determine:
    a. Size of the load: 1, 2, 4, 8, 10 bytes
    b. Type of the load: `ld.sa`, `ld.s`, `ld.a`, `ld.c.clr`, `ld.c.nc` or `ld`
    c. Target, source and post-increment registers of the load
4. If this is a data speculative load (`ld.sa`, or `ld.a`), invalidate the target register's ALAT entry using an `invala.e` instruction, and `rfi`.
5. If this is a `ld.c.clr` instruction invalidate the target register's ALAT entry using an `invala.e` instruction.
6. Emulate the memory read of the load instruction by updating the target register as follows:
    a. Validate that emulated code has the access rights to the target memory location at the privilege level that it was running prior to taking the alignment fault. The regular_form `probe` instruction can be used on the first and the last byte of the unaligned memory reference. If both probes succeed the memory reference may proceed.
    b. Using architectural `ld` instructions if the emulated operation is a `ld` or a `ld.c` (either clear or no clear flavor).
    c. Using `ld.s` instructions if the emulated operation is a `ld.s`. The result in the target register may end up with its NaT bit or NaTVal set, if one of the parts of emulation causes an exception. If ITLB.ed is 0 (no control speculation recovery code), then the misaligned `ld.s` may only be deferred if a fatal exception occurred on either half or the `ld.s` emulation.
7. If this is a post-increment load, compute the new value for the source register.

§

Instruction Emulation and Other Fault Handlers 7

This chapter introduces several common emulation handlers that an Itanium architecture-based operating system must support. A general overview is given for:

- Unaligned Reference Handler – emulation of misaligned memory references that the processor hardware cannot handle, or has been configured to fault on.
- Unsupported Data Reference Handler – emulation of memory operations that the processor hardware does not support. Examples are semaphore, `ldfe` or `stfe` operations to uncacheable memory.
- Illegal Dependency Fault Handler – this is a fatal condition that operating system needs to provide error logging functionality for.
- Long Branch Handler – the Itanium processor does not implement the long branch instruction. When encountered on the Itanium processor, long branches must be emulated by the operating system.

Floating-point software assist emulation handlers are not discussed here, but are presented in Chapter 8, “Floating-point System Software.” Additionally, Section 5.5.1, “Efficient Interruption Handling” on page 2:102 discusses more details about emulation code in the Itanium architecture.

7.1 Unaligned Reference Handler

Misaligned memory references that are not supported by the processor cause Unaligned Reference Faults. This behavior is implementation specific but typically occurs in cases where the access crosses a cache line or page boundary. In cases where the operating system chooses to emulate misaligned operations, some special cases need to be considered:

- Emulation of control and data speculative loads as well as advanced check and “regular” loads requires special attention. For details consult Section 6.3.1, “Unaligned Handler” on page 2:581.
- Emulation of unaligned semaphores, especially when interacting with IA-32 code require special attention. For details consult Section 2.1.3.2, “Behavior of Uncacheable and Misaligned Semaphores” on page 2:509.

IA-32 programs do not use the Itanium architecture-based handler to support unaligned references. The hardware that supports IA-32 execution provides the appropriate behavior if alignment checking is disabled through EFLAGS.ac. If an unaligned reference occurs in IA-32 code when EFLAGS.ac is set to enable alignment checking, alignment faults are delivered to a different vector from the unaligned reference handler. Specifically they are delivered to the IA_32_Exception(AlignmentCheck) vector; see Chapter 9, “IA-32 Interruption Vector Descriptions” for details.

7.2 Unsupported Data Reference Handler

Processors based on the Itanium architecture do not support all types of memory references to all memory attributes. In particular:

- Semaphore operations to uncacheable memory are not supported. For details consult Section 2.1.3.2, “Behavior of Uncacheable and Misaligned Semaphores” on page 2:509.
- A 10-byte memory access, e.g. `ldfe` or `stfe`, to uncacheable memory are not supported by all implementations.

The handler for 10-byte memory accesses must go through the following steps to emulate the `ldfe` or `stfe` instructions:

- Determine that the opcode at the faulting address is an `ldfe` or `stfe`. On control-speculative flavors of these instructions (`ldfe.s` or `ldfe.sa`) processor hardware always defers the unsupported data reference fault. In other words, software does not have to emulate control-speculative fault deferral.
- If the instruction is an advanced load `ldfe.a` then the emulation handler should invalidate the ALAT entry of the appropriate floating-point target register using the `invala.e` instruction. Furthermore, a zero should be returned in the floating-point target register.
- If the instruction is a regular `ldfe` or `stfe`, then software must emulate the load or store behavior of the instruction taking the appropriate faults if necessary.
- If the instruction is the base register update form, update the appropriate base register.

A number of these steps may require the use of self-modifying code to patch instructions with the appropriate operands (for example, the target register of the `inval.e` must be patched to the destination register of the `ldfe` or `stfe`). See Section 2.5, “Updating Code Images” on page 2:531 for more information.

7.3 Illegal Dependency Fault

The Itanium instruction sequencing rules specify that, generally speaking, instructions within an instruction group are free of dependencies as described in Section 3.4, “Instruction Sequencing Considerations” on page 1:39. A dependency violation occurs anytime a program violates read-after-write (RAW), write-after-write (WAW) or write-after-read (WAR) resource dependency rules within an instruction group.

As Section 3.4.4, “Processor Behavior on Dependency Violations” on page 1:44 describes, an implementation may provide hardware to detect and report dependency violations. It is important to note that the presence and capabilities of such hardware is implementation specific. A processor based on the Itanium architecture reports dependency violations through the General Exception Vector with an ISR.code of 8.

It is recommended that operating systems log the dependency violation and then terminate the offending application, as hardware behavior is undefined when a dependency violation occurs.

7.4 Long Branch

The Itanium architecture supports “long” branches with a 64-bit offset. This provides IP-relative conditional- and call-type branches that can reach any address in a 64-bit address space. These instructions use the MLX template, and similar to the move long instruction (`movl`), they encode their immediate in the L and the X slot of the bundle.

The Intel Itanium processor does not support the long branch instruction, `brl`, and requires the operating system to emulate its behavior. When an Itanium processor encounters a `brl` instruction, it vectors to the Illegal Operation Fault handler, regardless of the branches’ qualifying predicate. This handler is expected to emulate the long branch instruction in software. A general outline of the long branch emulation handler is as follows:

- The emulation handler reads the IIP, IPSR, and predicates at the time of the fault.
- If the fault occurred in IA-32 code or if the fault did not occur in slot 2 of a bundle (IPSR.ri is not 2), the handler passes the fault to regular illegal operation fault handler.
- Two floating-point registers are spilled into the integer register file to get ready to load the bundle.
- The emulation handler speculatively loads the 128-bit bundle at the faulting IP using the integer form of the floating-point load pair instruction. This instruction is chosen because it operates atomically (see Section 4.5, “Memory Datum Alignment and Atomicity”). Using two 64-bit integer loads would require the handler to ensure that another agent does not update the bundle between the two reads.
- If the speculation fails, the recovery code re-issues the load. Before re-issuing an architectural load, the processor must first re-enable PSR.ic to be able to handle potential TLB misses when reading the opcode from memory. In other words, this becomes a heavyweight handler. For details see Section 3.4.2, “Heavyweight Interruptions” on page 2:544. Once the opcode has been read from memory successfully flow of the emulation continues at the next step.
- The 128-bit bundle is moved from the FP register file into two integer registers and the FP registers are restored to their contents at the time of the fault.
- The handler extracts the fields necessary to decode the instruction (specifically, the qp, template, major opcode, and btype or b_1 fields of slot 2). It also determines the value of the qualifying predicate of the instruction in slot 2 from the contents of the predicate register at the time of the fault. Itanium instruction are always stored in memory in little-endian memory format. When extracting bit fields from the loaded opcode current processor endianness (PSR.be) must be taken into account.
- The emulation handler passes the fault off to the regular illegal operation fault handler if the bundle is not an MLX or if the faulting instruction is not a `brl.cond` or `brl.call`.
- If the faulting instruction is a not-taken `brl.cond` or `brl.call`, the code prepares to change the IIP to the address of the sequential successor of the faulting branch (i.e. IIP + 16) and jumps ahead to the trap detection code mentioned below.
- If the faulting instruction is a taken `brl.call`, the handler emulates the appropriate behavior of the call. The code uses a `br.call` to move the appropriate values into CFM and AR[PFS]. There are several details, however. First, the branch register update from the call must be backed out (as it is not the correct update for the `brl.call`). Second, AR[PFS].ppl must be set based on the cpl at the time of the fault (which is given by IPSR.cpl). Finally, the code must update the branch register

specified in the brl.call instruction with the IP of the successor of the brl.call (predication helps here as the Itanium instruction set does not provide an indirect move to branch register instruction).

- The handler forms the 60-bit immediate IP-offset for the brl target from the i and imm20 fields from the X syllable of the bundle (the brl instruction) and the imm39 field from the L syllable of the bundle.
- The handler checks to see if there are any traps to be taken. Specifically, it verifies that the next IP is at an implemented address (the specific test depends on whether the processor was in virtual or physical mode at the time of the fault as IPSR.it indicates), that taken branch traps are not enabled if the branch is taken, and that single stepping is not enabled.
- If a trap condition is detected, the ISR.code and ISR.vector fields are set up as appropriate and the handler jumps to the appropriate operating system entry point after restoring the predicates at the time of the fault and setting the IIP to the appropriate address.
- If no trap occurs, the handler restores the predicates and returns to the faulting code at the appropriate IP.

A processor based on the Itanium architecture typically does not fault on instructions with false qualifying predicates. However, an implementation may take an Illegal Operation Fault on an MLX instruction with a false predicate; the Itanium processor is such an implementation. This implies that the brl emulation handler must also provide the means to skip the faulting instruction when its qualifying predicate is false.

§

Floating-point System Software 8

This chapter details the way floating-point exceptions are handled in the Itanium architecture and how the architecture can be used to implement the ANSI/IEEE Std. 754-1985 for Binary Floating-point Arithmetic (IEEE-754). It is useful in creating and maintaining floating-point exception handling software by operating system writers.

8.1 Floating-point Exceptions in the Intel® Itanium® Architecture

Floating-point exception handling in the Itanium architecture has two major responsibilities. The first responsibility is to assist a hardware implementation to conform to the Itanium floating-point architecture specification. The Floating-point Software Assistance (FP SWA) Exception handler supports this conformance and is included as a driver in the Unified Extensible Firmware Interface (UEFI). The second responsibility is to provide conformance to the IEEE-754 standard. The IEEE Floating-point Exception Filter (IEEE Filter) supports providing this conformance.

When a floating-point exception occurs, a minimal amount of processor state information is saved in interruption control registers. Additional information is contained in the Floating-point Status Register (FPSR), i.e. application register (AR40). This register contains the IEEE exception enable controls, the IEEE rounding controls, the IEEE status flags, and information to determine the dynamic precision and range of the result to be produced.

When a floating-point exception occurs, execution is transferred to the appropriate interruption vector, either the Floating-point Fault Vector (at vector address 0x5c00) or the Floating-point Trap Vector (at vector address 0x5d00.) There the operating system may handle the exception or save additional processor information and arrange for handling of the exception elsewhere in the operating system. Floating-point exception faults must be handled differently than other faults. Correcting the condition that caused the fault (e.g. a page not present is brought into memory) and re-executing the instruction is how most other faults are handled. For floating-point faults, software is required to emulate the operation and continue execution at the next instruction as is normally done for traps. Part of this emulation needs to include a check for any lower priority traps that would have been raised if the instruction hadn’t faulted, e.g. a single-step trap.

8.1.1 Software Assistance Exceptions (Faults and Traps)

There are three categories of Software Assistance (SWA) exceptions that must handled by the operating system. The first two categories, SWA Faults and SWA Traps, are implementation dependent and could be generated by any Itanium floating-point arithmetic instruction that contains a status field specifier in the instruction's encoding. An implementation may choose to raise a SWA Fault as needed. The SWA Trap can only be raised under special circumstances. The third category, architecturally mandated

SWA Faults, is limited to the scalar reciprocal and scalar reciprocal square-root approximation instructions and is not implementation dependent. It is required for the correctness of the divide and square root algorithms.

8.1.1.1 SWA Faults

The Itanium architecture allows an implementation to raise SWA faults as required. Therefore an implementation-independent operating system must be able to emulate the architectural behavior of all FP instructions that can raise a floating-point exception. However, hardware implementations will limit the cases that raise SWA Faults for performance reasons. The most likely cases would be for the consumption of denormalized or unnormalized operands and production of denormalized results.

The general flow of the SWA Fault handler is as follows:

1. From the interruption instruction bundle pointer (IIP) and faulting instruction index (IPSR.ri), determine the FP instruction that faulted.
2. From the instruction, decode the opcode, static precision, status field and input/output register specifiers.
3. Read the data from the input registers.
4. From the opcode and the FPSR’s status field, decode the result range and precision.
5. From the ISR.code, determine that a SWA Fault has occurred, if not go to the last step.
6. From the FPSR, determine if the trap disabled or trap enabled result is wanted.
7. Emulate the Itanium instruction to produce the Itanium architecture specified result.
8. Place the result(s) in the correct FR and/or PR registers, if required.
9. Update the flags in the appropriate status field of the FPSR, if required.
10. Update the ISR.code if required. (This is required if the SWA fault has been translated into an IEEE fault or trap.)
11. Check to see if an IEEE fault or trap needs to be raised. If so, then queue it to the IEEE Filter, otherwise continue checking for lower priority traps that may need to be raised and if required invoke their handler. When finished, continue execution at the next instruction.

8.1.1.2 SWA Traps

SWA traps are allowed in the Itanium architecture as an optimization for cases when the hardware implementation has produced the result of the first (exponent unbounded) IEEE rounding[1] and can't continue with the second (exponent bounded) IEEE rounding to produce the final result. One option for the implementation would be to throw away the first IEEE rounding result and raise the SWA Fault. The SWA Fault handler would then have to redo the computation of the first IEEE rounding. A potentially more efficient option would be for the implementation to return the first IEEE rounding result and raise a SWA trap. Returning the first IEEE rounded result is

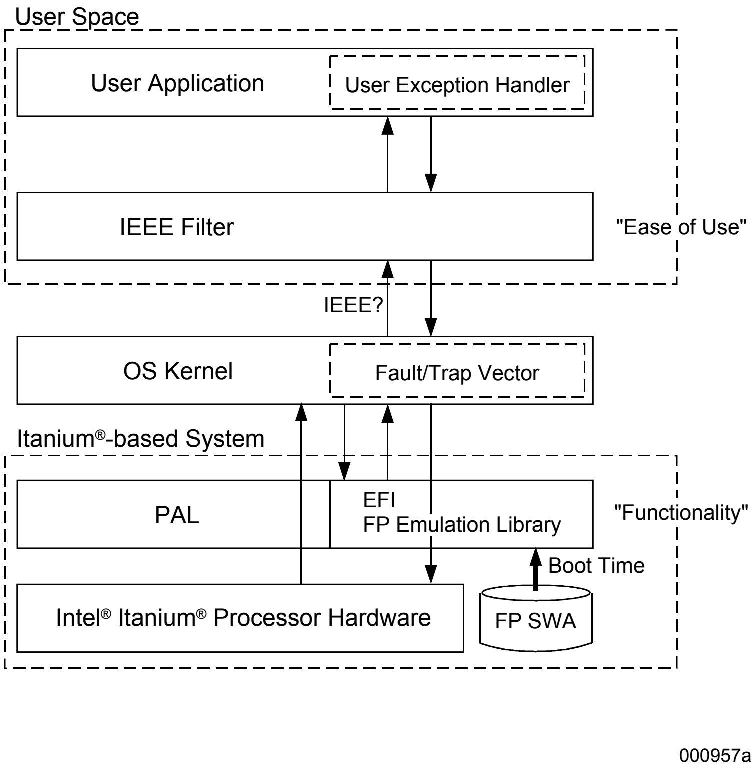
[1] ANSI/IEEE Std 754-1985 sections 7.3 Overflow and 7.4 Underflow.

the same as what is done when the IEEE Overflow or Underflow exceptions are enabled. However, hardware implementations will limit the cases that raise SWA Traps for performance reasons. The most likely case would be for the production of denormalized results.

For tiny[1] results, the SWA Trap handler has the simpler task of taking the intermediate result of the first IEEE rounding, the ISR.fpa and ISR.i status bits and producing the correctly rounded and signed minimum normal, denormal or zero. For huge[2] results, the SWA Trap handler has the even simpler task of taking the intermediate result of the first rounding and producing the correctly signed maximum representable normal or infinity, based on the sign of the result, the rounding direction, and the result precision and range.

Note: The Itanium architecture also allows for SWA Traps to be raised when the result is just Inexact. This is a trivial case for the SWA Trap handler, since result of the second IEEE rounding is identical to the first IEEE rounding.

Figure 8-1. Overview of Floating-point Exception Handling in the Intel® Itanium® Architecture



The general flow of the SWA Trap handler is as follows:

1. From the interruption instruction previous address (IIPA) and exception instruction index (ISR.ei), determine the FP instruction that trapped.
2. From the instruction, decode the opcode, static precision, status field and

[1] Tiny numbers are non-zero values with a magnitude smaller than the smallest normal floating-point number.

[2] Huge numbers have values larger in magnitude than the largest normal floating-point number.

input/output register specifiers.

3. From the ISR.code and FPSR trap enable controls, determine if a SWA Trap has occurred, if not go to the last step.
4. Read the first IEEE rounded result from the FR output register.
5. From the opcode and the status field, decode the result range and precision.
6. From the ISR.code's FPA, O, U, and I status bits and the intermediate result, produce the Itanium architecture specified result.
7. Place the result in the output FR register.
8. Update the flags in the appropriate status field of the FPSR, if required.
9. Update the ISR.code if required. (This is required if the SWA trap has been translated into an IEEE trap.)
10. Check to see if an IEEE trap needs to be raised. If so, then queue it to the IEEE Filter, otherwise continue checking for lower priority traps that may need to be raised and if required invoke their handler. When finished, continue execution at the next instruction.

8.1.1.3 Approximation Instructions and Architecturally Mandated SWA Faults

The scalar approximation instructions, `frcpa` and `frsqrta`, can raise architecturally mandated SWA Faults. This occurs when their input operands are such that they are potentially prevented from generating the correct result by the usual software algorithms that are employed for divide and square root. The reasons for this are that these algorithms may suffer from underflow, overflow, or loss of precision, because the inputs or result are at the extremes of their range. For these special cases, the SWA Fault handler must use alternate algorithms to provide the correct quotient or square root and place that result in the floating-point destination register. The predicate destination register is also cleared to indicate the result is not an approximation that needs to be improved via the iterative algorithm.

The parallel approximation instructions `fprcpa` and `fprsqrta` have situations similar to the scalar approximation instruction's architecturally mandated SWA Faults. This occurs when their input operands are such that they are potentially prevented from generating the correct result by the usual software algorithms that are employed for divide and square root. For these special cases, instead of generating a SWA Fault, the parallel approximation instructions indicate that software must use alternate algorithms to provide the correct reciprocal or square-root reciprocal by clearing the destination predicate register. The cleared predicate is the indication to the parallel IEEE-754 divide and square root software algorithms that alternative algorithms are required to produce the correct IEEE-754 quotient or square root.

8.1.2 The IEEE Floating-point Exception Filter

The Itanium architecture supports the reporting of the five IEEE-754 standard floating-point exceptions and the IA-32 Denormal Operand exception. In the Itanium architecture the Denormal Operand exception is expanded to the Denormal/Unnormal Operand exception. When referring to the IEEE-754 exceptions in the Itanium architecture the Denormal/Unnormal Operand exception is included.

At the application level, a user floating-point exception handler could handle the Itanium floating-point exception directly. This is the traditional operating system approach of providing a signal handler with a pointer to a machine-dependent data structure. It would be more convenient for the application developer if the operating system were to first transform the results to make them IEEE-754 conforming and then present the exception to the user in an abstracted manner. It is recommended that the operating system include such a software layer to enable application developers that want to handle floating-point exceptions in their application. The IEEE Floating-point Exception Filter provides this convenience to the developer through three functions.

- The first function of the IEEE Filter is to map the Itanium architecture's result to the IEEE-754 conforming result. This includes the wrapping of the exponent for Overflow and Underflow exceptions. The Itanium architecture keeps the exponent in the 17-bit format, which is not wrapped (i.e. scaled) with the appropriate value for the destination precision.
- The second function of an IEEE Filter is to transform the interruption information to a format that is easier to interpret and to invoke a user handler for the exception. The user's handler may then provide a value to be substituted for the IEEE default result, based on the operation, exception and inputs.
- The third function of the filter is to hide the complexities of the parallel instructions from the user. If a floating-point fault occurs in the high half of a parallel floating-point instruction and there is a user handler provided, the parallel instruction is split into two scalar instructions. The result for the high half comes from the user handler, while the low half is emulated by the IEEE Filter. The two results are combined back into a parallel result and execution is continued. More complicated cases can also occur with multiple faults and/or traps occurring in the same instruction.

Note: Usage of the IEEE Filter should not be compulsory – the user should be able to choose to handle enabled floating-point exceptions directly. The IEEE filter just hides the details of the instruction set and frees the user handler from having to emulate instructions directly and potentially incorrectly.

8.1.2.1 Invalid Operation Exception (Fault)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Invalid Operation exception is to leave the operands unchanged and to set the V bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.2 Divide by Zero Exception (Fault)

The exception-enabled response of an Itanium floating-point arithmetic instruction to a Divide-by-Zero exception is to leave the operands unchanged and to set the Z bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.3 Denormal/Unnormal Operand Exception (Fault)

The exception-enabled response of the Itanium arithmetic instruction to a Denormal/Unnormal Operand exception is to leave the operands unchanged and to set the D bit in the ISR.code field of the ISR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.1.2.4 Overflow Exception (Trap)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Overflow exception is to deliver the first (exponent unbounded) IEEE rounded result, and to set the O bit (and possibly the I and FPA bits) in the ISR.code field of the ISR register and the Overflow flags (and possibly the Inexact flag) in the appropriate status field of the FPSR register.

The IEEE-754 standard requires that, when raising an overflow exception, the user handler should be provided with the result rounded to the destination precision with the exponent range unbounded. For the huge result to fit in the destination's range, it must be scaled down by a factor equal to 2.0^{a} (with a equal to $3*2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result.) This scaling down will bring the result close to the middle of the range covered by the particular format. The exponent adjustment factors to do the scaling for the various formats are determined as follows:

- 8-bit (single) exponents are adjusted by $3*2^{6}$ = 0xc0 = 192.
- 11-bit (double) exponents are adjusted by $3*2^{9}$ = 0x600 = 1536.
- 15-bit (double-extended) exponents are adjusted by $3*2^{13}$ = 0x6000 = 24576.
- 17-bit (register) exponents are adjusted by $3*2^{15}$ = 0x18000 = 98304.

The actual scaling of the result is not performed by the Itanium architecture. The IEEE filter that is invoked before calling the user floating-point exception handler typically performs the scaling.

8.1.2.5 Underflow Exception (Trap)

The exception-enabled response of an Itanium floating-point arithmetic instruction to an Underflow exception is to deliver the first (exponent unbounded) IEEE rounded result, and to set the U bit (and possibly the I and FPA bits) in the ISR.code field of the ISR register and the Underflow flag (and possibly the Inexact flag) in the appropriate status field of the FPSR register.

The IEEE-754 standard requires that, when raising an underflow exception, the user handler should be provided with the result rounded to the destination precision with the exponent range unbounded. For the tiny result to fit in the destination's range, it must be scaled up by a factor equal to 2.0^{a} (with a equal to $3*2^{n-2}$, where n is the number of bits in the exponent of the floating-point format used to represent the result). The scaling up will bring result close to the middle of the range covered by the particular format. The exponent adjustment factors to do this scaling for the various formats are the same as those for enabled overflow exceptions, listed above.

Just as for overflow, the actual scaling of the result is not performed by the Itanium architecture. It is typically performed by the IEEE Filter, which is invoked before calling the user floating-point exception handler.

8.1.2.6 Inexact Exception (Trap)

The exception-enabled response of an Itanium arithmetic instruction to an Inexact exception is to set the I bit (and possibly the FPA bit) in the ISR.code field of the ISR register and the Inexact flag in the appropriate status field of the FPSR register. The operating system kernel, reached via the floating-point fault vector, will then invoke the user floating-point exception handler, if one has been registered.

8.2 IA-32 Floating-point Exceptions

IA-32 floating-point exceptions may occur when executing code in IA-32 mode. When this happens, execution is transferred to the Itanium interruption vector for IA-32 Exceptions (at vector address 0x6900.) For classic IA-32 floating-point instructions, they are raised via the "IA_32_Exception(FPError) – Pending Floating-point Error." For SSE instructions, they are raised via the "IA_32_Exception(StreamingSIMD) – SSE Numeric Error Fault." The operating system may schedule Itanium architecture-based and/or IA-32 exception handlers for these exceptions.

§

IA-32 Application Support 9

The Itanium architecture enables Itanium architecture-based operating systems to host IA-32 applications, Itanium architecture-based applications, as well as mixed IA-32/Itanium architecture-based applications. Unless the operating system explicitly intercepts ISA transfers (using the PSR.di), user-level code can transition between the two instruction sets without operating system intervention. This allows IA-32 programs to call Itanium architecture-based subroutines or vice-versa. Itanium architecture-based and IA-32 code can share data through registers and/or memory. Multi-threaded IA-32 and Itanium architecture-based applications can easily communicate with each other or the Itanium architecture-based operating system using shared memory. The Itanium architecture does not support execution of Itanium architecture-based programs on an IA-32 operating system. While the architecture does not prevent IA-32 code from executing as part of an Itanium architecture-based operating system, it is strongly recommended that Itanium architecture-based operating systems do **not** contain IA-32 code.

One of the most compelling motivations for executing IA-32 code on an Itanium architecture-based operating system is the ability to run existing unmodified IA-32 application binaries. Because IA-32 performs 32-bit instruction/memory references that are zero-extended into 64-bit virtual addresses, Itanium architecture-based operating systems must ensure that all IA-32 code and data is located in the lower 4GBytes of the virtual address space. Compute intensive IA-32 applications can improve their performance substantially by migrating compute kernels from IA-32 to Itanium architecture-based code while preserving the bulk of the application's IA-32 binary code. If mixed IA-32/Itanium architecture-based applications are supported, care has to be taken that the data accessible to IA-32 portions of the application is located in the lower 4GBytes of the virtual address space.

While processors based on the Itanium architecture are capable of supporting a wide range of Itanium architecture-based/IA-32 code mixing, Itanium architecture-based operating systems need to provide a software support infrastructure to enable full interoperability between the IA-32 and Itanium instruction set. Most Itanium architecture-based operating systems are expected to support user-level IA-32 applications, and, as a result, must be able to provide the full range of operating system services through a 32-bit system call interface. However, different operating systems and runtime conventions may reduce the set of interoperability modes as desired by the operating system vendor.

While it is an interesting topic, this chapter does not discuss 32-bit application binary interfaces provided by specific operating systems. Instead, this chapter focusses on what services are required from an Itanium architecture-based operating system by a processor based on the Itanium architecture that is executing IA-32 code. In other words, the focus of this chapter is the low-level processor / operating system interface rather than the IA-32 software / operating system (application binary) interface.

9.1 Transitioning between Intel® Itanium® and IA-32 Instruction Sets

As mentioned earlier, user-level code can transition from Itanium to IA-32 (or back) instruction sets without operating system intervention. As described in Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium® System Environment" in Volume 1, two instructions are provided for this purpose: `br.ia` (an Itanium unconditional branch), and JMPE (an IA-32 register indirect and absolute jump). Prior to executing any IA-32 instructions, however, the Itanium architecture-based operating system needs to setup an execution environment for executing IA-32 code.

9.1.1 IA-32 Code Execution Environments

Processors based on the Itanium architecture are capable of executing IA-32 code in real mode, VM86 mode or protected mode. When segmentation is enabled both 16 and 32-bit code are supported. Prior to transferring control to IA-32 code, an Itanium architecture-based application and/or operating system is expected to setup the complete IA-32 execution environment in Itanium registers.

In particular, Itanium architecture-based software must setup IA-32 segment descriptor and selector registers in Itanium application registers, and must ensure that code and stack segment descriptors (CSD, SSD) are pointing at valid and correctly aligned memory areas. It is also worth noting that the IA-32 GDT and LDT descriptors are maintained in GR30 and GR31, and are unprotected from Itanium architecture-based user-level code. For more details on the IA-32 execution environment please refer to Section 6.2.2, "IA-32 Application Register State Model" on page 1:113.

Some IA-32 execution environments may need support from an Itanium architecture-based operating system. Which IA-32 software environments are supported by an Itanium architecture-based operating system is determined by the operating system vendor. Itanium architecture-based platform firmware (SAL) provides a runtime environment that allows execution of real-mode IA-32 code found in PCI configuration option ROMs.

9.1.2 br.ia

`br.ia` is an unconditional indirect branch that transitions from Itanium to IA-32 instruction set. Prior to entering IA-32 code with `br.ia`, software is also required to flush the register stack. `br.ia` sets the size of the current register stack frame to zero. The register stack is disabled during IA-32 code execution. Because IA-32 code execution uses Itanium registers, much of the Itanium register state is overwritten and left in an undefined state when IA-32 code is run. As a result, software can not rely on the value of such registers across an instruction set transition. Execution of IA-32 code also invalidates the ALAT. For more details refer to Table 6-2, "IA-32 Segment Register Fields" on page 1:118.

For best performance, the following code sequence is recommended for transitioning from Itanium to IA-32 instruction set:

```
{.mii
    flushrs             // flush register stack
    mov b7 = rTarget    // Setup IA-32 target address
    nop.i               // nop.i or other instruction
    ;;
{.mib
    nop.m               // nop.m or other instruction
    nop.i               // nop.i or other instruction
    br.ia.sptk b7       // branch to IA-32 target defined by
                        // lower 32-bits of branch register b7
    ;;
```

Key to performance is that the register stack flush (`flushrs`) and the `br.ia` instruction are separated by a single cycle, and that the `br.ia` instruction is the first B-slot in the bundle directly following the `flushrs`. The `nop` instruction slots in the code example may be used for other instructions.

9.1.3 JMPE

JMPE is an IA-32 instruction that comes in a register indirect and absolute branch flavors. The code segment descriptor base is held in the CSD application register (ar.csd).

- JMPE reg16/32 computes the target of the Itanium instruction set as
  `IP = ([reg16/32] + CSD.base) & 0xfffffff0`
- JMPE disp16/32 computes the target of the Itanium instruction set as
  `IP = (disp16/32 + CSD.base) & 0xfffffff0`

Targets of the IA-32 JMPE instruction are forced to be 16-byte aligned, and are constrained to the lower 4Gbytes of the 64-bit virtual address space. The JMPE instruction leaves the IA-32 return address (address of the IA-32 instruction following the JMPE itself) in IA_64 register GR1.

9.1.4 Procedure Calls between Intel® Itanium® and IA-32 Instruction Sets

If procedure call linkage is required between Itanium architecture-based and IA-32 subroutines, software needs to perform additional work as described in the next two sections.

9.1.4.1 Itanium® Architecture-based Caller to IA-32 Callee

This section outlines what steps an Itanium architecture-based caller of an IA-32 procedure needs to perform. The ordering of the steps is approximate and need not be executed exactly in the order presented.

1. Setup IA-32 execution environment, if not already done (see Section 9.1.2 for details). Ensure that no NaTed registers are used to setup IA-32 environment nor that they are passed as procedure call arguments to IA-32 code.
2. Marshall arguments from the register stack to memory stack according to IA-32 software conventions.
3. Set up exception handle unwind data structures according to OS convention.

4. Make sure JMPE knows where to return to, e.g. deposit return address for the JMPE on memory stack or pass it in an IA-32 visible register.
5. Setup IA-32 branch target in branch register.
6. Flush register stack, but no other RSE updates.
7. `br.ia` is an indirect branch to IA-32 code. There is no need to preserve Itanium only application registers, since IA-32 code execution leaves them unmodified.
8. Run in the IA-32 callee until it executes a JMPE instruction.
9. JMPE instruction is an unconditional jump to Itanium architecture-based code. JMPE should use the return address specified in step 4.
10. Move return values from memory stack to static Itanium register used for procedure return value according to Itanium calling conventions.
11. Ensure that IA-32 code correctly unwound memory stack, and that memory stack pointer is correctly aligned.
12. Update exception handle unwind data structures according to OS convention.
13. `br.ret` returns to Itanium architecture-based caller.

9.1.4.2 IA-32 Caller to Itanium® Architecture-based Callee

This section outlines what steps an IA-32 caller of an Itanium architecture-based procedure needs to perform. The ordering of the steps is approximate and need not be executed exactly in the order presented.

1. Caller deposits arguments on memory stack, and calls Itanium architecture-based transition stub using the JMPE instruction.
2. Execute JMPE instruction as an unconditional branch to Itanium architecture-based code. The JMPE instruction will leave the address of the IA-32 instruction following the JMPE itself in Itanium register GR1. This address may be used as a return address later.
3. Allocate a register stack frame with the `alloc` instruction.
4. Load procedure arguments from memory stack into Itanium stacked registers. Preserve IA-32 return address in memory or register stack.
5. Set up exception handle unwind data structures according to OS convention.
6. `br.call` to target Itanium architecture-based callee.
7. Execute Itanium architecture-based code until it returns using `br.ret`.
8. Move return value from static Itanium register to memory stack.
9. Load IA-32 return address from step 4 into branch register.
10. Instead of flushing the register stack to memory, the contents of the register stack can be discarded at this point since IA-32 code execution will overwrite it anyway. Invalidate register stack by:
    a. Allocating a zero-size stack frame using the `alloc` instruction.
    b. Writing zero into RSC application register, and executing a `loadrs` instruction.
    c. Restore RSC application register to its original value in preparation for the next call from IA-32 to Itanium instruction set.

11. Ensure memory stack pointer is correctly aligned prior to returning to IA-32 code.
12. `br.ia` returns to IA-32 caller.

9.2 IA-32 Architecture Handlers

An Itanium architecture-based operating system needs to be prepared to handle exceptions from Itanium architecture-based and IA-32 code. Depending on the exception cause, exception vectors can be:

- Shared Itanium/IA-32 Exception Vectors: all virtual memory related instruction and data reference faults share a common exception vector, regardless of whether they were caused by Itanium architecture-based or IA-32 code.
- Unique Itanium Exception vectors: these are conditions that only Itanium architecture-based code can cause. Examples are: Instruction Breakpoint fault, Illegal Operation fault, Illegal Dependency fault, Unimplemented Data Address fault, etc.
- Unique IA-32 Exception Vectors: these conditions can occur only from IA-32 instructions.

A detailed break-down of which exceptions occur on which interruption vector and from which instruction set is given in Table 5-6. Table 9-1 shown below summarizes all IA-32 related exceptions that an Itanium architecture-based operating system needs to be ready to handle. These IA-32 specific interrupts are grouped into three vectors: the IA-32 Exception vector, the IA-32 Intercept, and the IA-32 Interrupt vector. Within each of these vectors the interrupt status register (ISR) provides detailed codes as to the origin of this exception. Details on the IA-32 vectors is provided in Chapter 9, “IA-32 Interruption Vector Descriptions.” More details on debug related IA-32 exceptions is given in the following section of this document.

Table 9-1. IA-32 Vectors that need Itanium® Architecture-based OS Support

Vector (IVA offset)	Exception Name	Exception Related To	Expected OS Behavior
IA-32 Exception vector (0x6900)	IA-32 Instruction Debug fault	Debug	Relay to debugger.
	IA-32 Code Fetch fault	Segmentation	Signal application.
	IA-32 Instruction Length > 15 bytes fault	Bad Opcode	Signal application.
	IA-32 Device Not Available fault	Numeric	Signal application.
	IA-32 FP Error fault	Numeric	Signal application.
	IA-32 Segment Not Present fault	Segmentation	Signal application.
	IA-32 Stack Exception fault	Segmentation	Signal application.
	IA-32 General Protection fault	Segmentation	Signal application.
	IA-32 Divide by Zero fault	Numeric	Signal application.
	IA-32 Alignment Check fault	Misaligned IA-32 Memory Reference with alignment checking enabled.	Depends on convention.
	IA-32 Bound fault	Segmentation	Signal application.
	IA-32 SSE Numeric Error Fault	Numeric	Signal application.
	IA-32 INTO Overflow trap	Numeric	Signal application.
	IA-32 Breakpoint (INT 3) trap	Software Breakpoint	Depends on convention.
	IA-32 Data Breakpoint trap	Debug	Relay to debugger.

Table 9-1. IA-32 Vectors that need Itanium® Architecture-based OS Support (Continued)

Vector (IVA offset)	Exception Name	Exception Related To	Expected OS Behavior
	IA-32 Taken Branch trap	Debug	Relay to debugger.
	IA-32 Single Step trap	Debug	Relay to debugger.
	IA-32 Invalid Opcode fault	Bad Opcode	Signal application.
IA-32 Intercept vector (0x6a00)	IA-32 Instruction Intercept fault	Attempted to access IA-32 paging, MTRRs, IDT, IA-32 control registers, IA-32 debug registers or attempted to execute IA-32 privileged instructions.	This is not supported on an Itanium architecture-based OS. Signal application.
	IA-32 Locked Data Reference fault	Attempt to reference misaligned or uncacheable semaphore.	Emulation handler if needed. Refer to Section 2.1.3.2, "Behavior of Uncacheable and Misaligned Semaphores" on page 2:509.
	IA-32 System Flag Intercept trap	System Flag intercept	Depends on convention.
	IA-32 Gate Intercept trap	Gate/Task transfer intercept	Depends on convention.
IA-32 Interrupt vector (0x6b00)	IA-32 Software Interrupt (INT) trap	Software Interrupt	Depends on convention.
Cannot happen in Itanium architecture-based operating system	IA-32 Double Fault IA-32 Invalid TSS Fault, IA-32 Page Fault, IA-32 Machine Check	N/A	Don't worry,

9.3 Debugging IA-32 and Itanium® Architecture-based Code

Itanium architecture-based operating systems that want to provide debug support for both IA-32 and Itanium architecture-based applications, need to be aware of the differences between taking instruction and data breakpoint exceptions as well as single step or taken branch traps on Itanium and IA-32 instructions.

9.3.1 Instruction Breakpoints

If an Itanium instruction matches an instruction breakpoint register (IBR) then an Instruction Debug Fault is delivered on the Itanium Debug vector. To step across a single Itanium instruction, IPSR.id must be set to one. An IA-32 instruction, however, that matches an IBR causes an IA-32 Instruction Breakpoint fault which is delivered to the IA-32 Exception vector (Debug). To step across a single IA-32 instruction, either IPSR.id or EFLAGS.rf must be set to one.

9.3.2 Data Breakpoints

If an Itanium memory reference matches a data breakpoint register (DBR) then a Data Debug Fault is delivered on the Itanium Debug vector. To step across a single data breakpoint, IPSR.dd must be set to one. An IA-32 instruction, however, that matches a DBR causes an IA-32 Data Breakpoint *trap* which is delivered to the IA-32 Exception vector (Debug). In other words, the debugger only gets control after the instruction making the reference has completed. Since IA-32 instruction can make multiple memory references, a single IA-32 instruction may cause multiple data break points to trigger. Details on how this is communicated to software in the interrupt status register (ISR) is given in Section 9.1, “IA-32 Trap Code” on page 2:213. Since IA-32 data breakpoints are traps, there is no need to step over them.

9.3.3 Single Step Traps

When PSR.ss enables single stepping of Itanium architecture-based applications, each instruction that is stepped will stop at the Single Step trap handler. When PSR.ss or EFLAG.tf enable single stepping of IA-32 applications, an IA_32_Exception(Debug) trap is taken after each IA-32 instruction. For more details refer to Section 9.1, “IA-32 Trap Code” on page 2:213.

9.3.4 Taken Branch Traps

When PSR.tb enables taken branch trapping on Itanium architecture-based applications, each taken branch will transfer control to the Taken Branch Trap handler. When PSR.tb is set, taken IA-32 branches transfer control to the IA_32_Exception(Debug) trap handler taken after each IA-32 instruction. For more details refer to Section 9.1, “IA-32 Trap Code” on page 2:213.

§

External Interrupt Architecture 10

The Itanium architecture provides a high performance external interrupt architecture. While IA-32 processors commonly use a three wire shared APIC bus, processors based on the Itanium architecture utilize a high performance, message-based, point-to-point protocol between processors and multiple I/O interrupt controllers. To ensure that processors based on the Itanium architecture can fully leverage the large set of existing platform infrastructure and I/O devices, compatibility with existing platform infrastructure is provided in the form of direct support for Intel 8259A compatible interrupt controllers and limited support for level sensitive interrupts.

This chapter introduces the basic external interrupt mechanism provided by the architecture, while Section 5.8, "Interrupts" provides the complete architectural definition for the Itanium external interrupt architecture.

10.1 External Interrupt Basics

Interrupts are identified by their vector number. The vector number implies interrupt priority, and also determines whether the interrupt is delivered to processor firmware as a "PAL-based" interrupt, or whether it is delivered to the operating system as an "IVA-based" external interrupt.

This chapter discusses asynchronous external interrupts only. PAL-based platform management interrupts (PMI) are not discussed here. External interrupts are IVA-based and are delivered to the operating system by transferring control to code located at address CR[IVA]+0x3000. This code location is also known as the external interrupt vector and is described on page 2:186.

Software can distinguish interrupts based on their vector number. Vector numbers range from 0 to 255. Vector numbers also establish interrupt priorities as follows:

- Vector numbers below 16 are special, and are architecturally defined in Section 5.8.1, "Interrupt Vectors and Priorities" on page 2:118. The non-maskable interrupt (NMI) is always vector 2 and is higher priority than all in-service external interrupts. ExtINT, Intel 8259A compatible external interrupt controller interrupt, is always vector 0. Vector numbers below 16 have higher priority than vectors above 16. Vector 15 is used to indicate that the highest priority pending interrupt in the processor is at a priority level that is currently masked or there are no pending external interrupts.
- For vector numbers between 16 and 255, higher vector numbers imply higher priority. In this range, vectors are freely assignable by software. This is achieved by programming of interrupt controllers and the processor internal interrupt configuration registers.

10.2 Configuration of External Interrupt Vectors

As defined in Section 5.8, “Interrupts” on page 2:114, external interrupts originate from one of four sources:

- From external sources, e.g. external interrupt controllers or intelligent external I/O devices, or
- From the processor’s LINT0 or LINT1 pins[1] (typically connected to an Intel 8259A compatible interrupt controller), or
- From internal processor sources, e.g. timers or performance monitors, or
- From other processors, e.g. inter-processor interrupts (IPIs).

All interrupts are point-to-point communications. There is no facility for broadcasting of interrupts. The interrupt message protocol used by the processor-to-processor and the external source-to-processor is not defined architecturally, and is not visible to software.

A number of external interrupt control registers (LID,TPR, ITV, PMV, CMCV, LRR0 and LRR1) allow software to directly configure the processor interrupt resources. The Local ID register (LID) establishes a processor’s unique physical interrupt identifier. The Task Priority Register (TPR) allows masking of external interrupts based on vector priority classes. The ITV, PMV, CMCV, LRR0 and LRR1 external interrupt control registers configure the vector number for the processor’s local interrupt sources. Configuration of the external controllers and devices is controller-/device-specific, and is beyond the scope of this document.

10.3 External Interrupt Masking

The Itanium architecture provides four mechanisms to prevent external interrupts from being delivered to a processor: a bit in the processor status register (PSR.i), the interrupt vector register (IVR) and the end-of-interrupt (EOI) register, the task priority register (TPR), and the external task priority register (XTPR). The next four sections discuss these mechanisms.

10.3.1 PSR.i

When PSR.i is zero, the processor does not accept any external interrupts. However, interrupts continue to be pended by the processor. Software can use PSR.i to temporarily disable taking of external interrupts, e.g. to ensure uninterruptable execution of critical code sections. Since clearing of PSR.i takes effect immediately (refer to the rsm instruction page), software is not necessarily required to explicitly serialize clearing of PSR.i (unless another processor resource requires serialization). On

[1] Processors optionally support two external interrupt pins. Software can query for the presence of LINT pins via the PAL_PROC_GET_FEATURES procedure call.

the way out of an uninterruptable code section software is not required to serialize the setting of PSR.i either, unless it is of interest to software to be able to take interrupts in the very next instruction group. A code example for this case is given below:

```
rsm i ;;
// rsm of PSR.i takes effect on the next instruction

// uninterruptable code sequence here

ssm i ;;
// ssm of PSR.i does require data serialization, if we need to ensure
// that external interrupts are enabled at the very next instruction. If
// data serialization is omitted, PSR.i is set to 1 at the latest when
// the next exception is taken.
```

By avoiding the serialization operations on PSR.i the performance of such uninterruptable code sections is improved.

10.3.2 IVR Reads and EOI Writes

As described in Section 10.4, IVR reads return the highest priority, pending, unmasked vector, and places this vector "in-service." Additionally, IVR reads have the side-effect of masking all vectors that have equal or lower priority than one that is returned by the IVR read. Correspondingly, writes to the EOI register unmask all vectors with equal or lower priority than the highest priority "in-service" vector. Due to nesting of higher priority interrupts, it is possible to have multiple vectors in the "in-service" state.

10.3.3 Task Priority Register (TPR)

The Task Priority Register (TPR) provides an additional interrupt masking capability. It allows software to mask interrupt "priority classes" of 16 vectors each by specifying the mask priority class in the TPR.mic field. The TPR.mmi field allows masking of all maskable external interrupts (essentially all but NMI).

An example of TPR use is shown in Section 10.5.2, "TPR and XPTR Usage Example" on page 2:608.

10.3.4 External Task Priority Register (XTPR)

The External Task Priority Register (XTPR) is a per-processor resource that can be provided by external bus logic in some Itanium architecture-based platforms. If supported by the platform, XTPR can be used by the operating system to redirect external interrupts to other processors in a multiprocessor system.

The XTPR is updated by performing a 1-byte store to the XTP byte which is located at an offset of 0x1e0008 in the Processor Interrupt Block (see Section 5.8.4, "Processor Interrupt Block" for details). Since the timing of the modification of the XTP register is not time critical there is no serialization required. Effects of the one byte store operation are platform specific. Typically, it will generate a transaction on the system bus identifying it as an XTP register update transaction, and will indicate which processor generated the transaction as well as the stored data.

An example of XTPR use is included in Section 10.5.2, "TPR and XPTR Usage Example" on page 2:608.

10.4 External Interrupt Delivery

The architectural interrupt model in Section 5.8 defines how each interrupt vector cycles through one of four states:

- *Inactive*: there is no interrupt *pending* on this vector.
- *Pending*: an interrupt has been received by the processor on this vector, but has not been *accepted* by the processor and has not been *acquired* by software. The processor hardware will *accept* the interrupt when this vector's priority level is higher than the highest currently in-service vector, PSR.i is one, and TPR settings do not mask the interrupt. This will cause the processor to transfer control flow to the external interrupt handler. Software can then *acquire* the highest priority, pending, unmasked vector by reading the IVR control register. The IVR read returns the 8-bit vector number in a register and masks all vectors that have equal or lower priority. This vector now enters the In-Service/None Pending state.
- *In-Service/None Pending*: an interrupt has been received by the processor on this vector, and has been acquired by software (by reading the IVR control register), but software has not *completed servicing* this interrupt. In this state, the processor masks all vectors that have equal or lower priority. In this state, the processor can receive and remember a second interrupt on this vector. If this happens, the processor transitions this vector to the "In-Service/One Pending" state. If software *completes the interrupt* service routine (indicated to the processor by writing the EOI register) before another interrupt is received on this vector, then the processor returns this vector to the Inactive state, and all vectors with equal or lower priority are unmasked.
- *In-Service/One Pending*: an interrupt has been received by the processor on this vector, and has been acquired by software (by reading the IVR control register), and software has not completed servicing this interrupt. Additionally, the processor received a second interrupt on this vector, which is now held pending. If additional interrupts on this vector are received by the processor while this vector is in the "In-Service/One Pending" state, those additional interrupts are not distinguishable by the processor hardware. When software completes the interrupt service routine for the original interrupt on this vector (indicated to the processor by writing the EOI register), then the processor returns this interrupt vector to the Pending state for the second interrupt that was received on this vector. Additionally, all vectors with equal or lower priority are unmasked.

It is recommended the following structure for an Itanium architecture-based external interrupt handler:

1. Read and Save TPR, i.e. save Old Task Priority variable (optional).
2. External Interrupt Harvest Loop:
    a. Read the IVR control register to determine which vector is being delivered. If the returned IVR value is 15, then this is a spurious interrupt and it can be can ignored; software can now clear PSR.ic, restore IPSR and IIP and then `rfi` to the interrupted context. If the returned IVR value is not 15, continue with step 2b.

    b. Raise TPR register to the interrupt class to which the level read out of IVR belongs (optional).

c. Software must preserve IIP and IPSR prior to re-enabling PSR.ic and PSR.i which will re-enable taking of exceptions and higher priority external interrupts.

d. Issue a `srlz.d` instruction. This ensures that updated PSR.ic and PSR.i settings are visible, and it also makes sure that the IVR read side effect of masking lower or equal priority interrupts is visible when PSR.i becomes 1.

e. Dispatch the appropriate interrupt service routine.

f. Disable external interrupts by clearing PSR.i with an `rsm 0x4000` instruction.This ensures that external interrupts are disabled prior to the EOI write in the next step.

g. Notify the processor that interrupt handling for this vector is completed by writing to the EOI register. This will unmask any pending lower priority interrupts. If this was a level triggered interrupt, write to the I/O SAPIC EOI register.

h. Lower TPR register to Old Task Priority (optional).

i. Issue a `srlz.d` instruction. This ensures that ensure the EOI write from step 2g is reflected in the future IVR read (in step 2a). It also ensures that the TPR update from step 2h unmasks any interrupts in the priority classes (including the current task priority level) that were masked by the previous value of TPR.

j. Return to top of loop (step 2a).

These steps assume that the routine's caller already performed the required state preservation of interruption resources. Therefore the focus of the steps above is to check the IVR to acquire the vector so the operating system can determine what device the interrupt is associated with. The code is setup to loop, servicing interrupts until the spurious interrupt vector (15) is returned. Looping and harvesting outstanding interrupts reduces the time wasted by returning to the previous state just to get interrupted again. The benefit of interrupt harvesting is that the processor pipeline is not unnecessarily flushed and that the interrupted context is only saved/restored once for a sequence of external interrupts. Once the vector is obtained the specific interrupt service routine is called to service the device request. Upon return from the interrupt service routine, an EOI is written and the IVR is checked once again.

If the operating system does not implement priority levels then there is no need to save and restore the task priority level (steps 1, 2b, and 2h are optional). As described in Section 10.3 above, an IVR read automatically masks interrupts at the current in-service level and below until the corresponding EOI is issued. For level triggered interrupts, the programmer must not only inform the processor, but the external interrupt controller that the level triggered interrupt has been serviced.

10.5 Interrupt Control Register Usage Examples

The examples in this section provide an overview of using the Itanium external interrupt control registers. Actual and pseudo code fragments are listed to aid in the development of OS code which will utilize these registers. It is up to the operating system and its writer to determine what minimum set of control registers are required to be used.

10.5.1 Notation

Preprocessor macros for function ENTRY and END are used in the examples to reduce duplication of code and reduce document space requirements.

```
#define ENTRY(label) \
     .text; \
     .align 32;; \
     .global label; \
     .proc label; \
label::

#define END(label) .endp
```

10.5.2 TPR and XPTR Usage Example

This code will allow certain interrupts to be masked by increasing/decreasing the task priority register. If you don’t want to mask all external interrupts, you can raise the priority level to mask out only the interrupts that have higher priority (and no effect on your current critical section).

We also take the expensive route here by updating not only the processor TPR, but the External Task Priority Register used by the chipset (if supported) as a hint to what processor should receive the next external interrupt.

```
//
// routine to set the task priority register to mask
// interrupts at the specific level or below
//
// INPUT: SPL level
//

TPR_MIC=4
TPR_MIC_LEN=4

.global external_task_pri_reg// address points to Interrupt Delivery block

ENTRY(set_spl)
    alloc r18=ar.pfs,1,0,0,0
    dep.z r22=r32,TPR_MIC,TPR_MIC_LEN
    movl r19=external_task_pri_reg
    ;;
    mov cr.tpr=r22
    ld8 r20=[r19]   // get address of EXt. TASK Priority Register
    ;;
    srlz.d          // srlz.d only required if want TPR update effective
immediately
    st1 [r20]=r32   // if supported by platform: update eXternal Task Priority
(XTP)
    br.ret.sptk b0
    ;;
END(set_spl)
```

10.5.3 EOI Usage Example

This example is a typical return from an interrupt service routine to the generic interrupt handler. Interrupts are disabled before returning to the main trap handler in preparation for returning from kernel space.

```
return_from_interrupt:
// disable interrupts here

    rsm 0x4000              // make sure interrupts disabled


// interrupt_eoi# clear the sapic/pic interrupt
sapic_eoi:
    mov cr.eoi=r0           // issue and eoi
    ;;
    srlz.d                  // make sure it takes effect


// issue the appropriate EOI sequence to the external interrupt
// controller here.
```

For level trigger interrupts, the OS is required to issue an EOI not only to the processor, but also the external interrupt controller where the interrupt originated. This forces the OS to keep track of whether the vector is associated with a level or an edge trigger interrupt line.

10.5.4 IRR Usage Example

Waiting on an interrupt with interrupts disabled.

```
my_interrupt_loop::
//
// check for vector 192 (0xc0) via irr3
//

        mov     r3=cr.irr3
        ;;
        and     r3=0x1,r3
        ;;
        cmp.eq p6,p7=0x1,r3
    (p7)br.cond.sptk.few   my_interrupt_loop
        ;;
        mov     r4=cr.ivr       // read the vector
        ;;
        mov     cr.eoi=r0       // clear it
        ;;
```

10.5.5 Interval Timer Usage Example

The Itanium architecture provides a 64 bit interval timer for elapsed time notification interrupts. It is similar to the IA-32 Time Stamp Counter (TSC). Programming the Itanium interval timer consists of initializing the ITV (CR 72), ITM (CR 1), and ITC (AR 44).

The Interval Timer Vector (ITV) specifies the external interrupt vector number for the Interval Timer Interrupts. The code examples below show how to clear and initialize the timers vector, match register, and count registers.

The Interval Time Counter (ITC) gets updated at a fixed relation to the processor clock. The ITM, Interval Timer Match, is used to determine when a interval timer interrupt is generated. When the ITC matches the ITM and the timer is unmasked via ITV then an interrupt will be generated.

```
//
// routine to reset the interval timer to zero..
//

ENTRY(em_timer_reinit)
    mov    ar.itc=r0              // reset itimer counter
    br.ret.spnt.few rp
END(em_timer_reinit)


//
// routine to setup the interval timer.
//
//  1) setup the interval timer vector
//  2) initialize the time counter to zero
//  3) initialize the match register
//
//  INPUTS: timermatch -- value to initialize ITM register with.
//          vector number -- vector to interrupt with
//  OUTPUTS: none
//
ENTRY(enable_minterval)
    alloc   r14=ar.pfs,0x2,0,0,0  // get ready for input parameters
    mov     ar.itc=r0             // initialize counter to zero
    ;;
    mov     cr.itm=r32            // set match register
    ;;
    srlz.d
    mov     cr.itv=r33            // set interval timer vector
    ;;
    srlz.d                        // make sure it goes through
    br.ret.sptk.few rp            // return
    .endp
```

Since the ITC gets updated at a fixed relation to the processor clock, in order to find out the frequency at run time, one can use a firmware call to obtain the input frequency information to the interval time. Using this frequency information the ITM can be set to deliver an interrupt at a specific time interval (i.e. for operating system scheduling purposes). Assuming the frequency information returned by the firmware is in ticks per second, the programmer could use a time-out delta for delivering a timer interrupt every 10 milliseconds as follows:

```
timeout_delta=ticks_per_second/100;
```

where `ticks_per_second` is the frequency value returned by the firmware and `timeout_delta` will be the value added to the ITC for setting the next ITM. Therefore, the ITC is left free running, but the ITM must be updated upon every timer interrupt with its next time out match value, i.e. ITM = ITC + `timeout_delta`.

The only issue with this setup is if the timer interrupt delivery is delayed beyond the point of the original intended delivery time (i.e. ITC > ITM). This could happen if interrupts were disabled or blocked by the operating system/device driver longer than the time-out value. In this case the ITM has to be adjusted in order for the next ITM to be accurate. The following algorithm could be used to adjust the next ITM before returning from the timer interrupt handler.

```
for (;;) {
    itm_next = itm_next + timeout_delta + (read current ITC - read current ITM);
    if (itm_next < current ITC) {
        /* we missed the next interrupt already, continue */
    } else {
        set_itm(itm_next);
        break;
    }
}
```

where `itm_next` was initialized to current ITC + `timeout_delta`, and `set_itm` in Itanium architecture-based assembly would look like:

```
.global set_itm
.proc set_itm
set_itm:
    alloc r18=ar.pfs,1,0,0,0
    mov cr.itm=r32
    ;;
    srlz.d
    br.ret.sptk b0
    ;;
.endp set_itm
```

10.5.6 Resource Utilization Counter Usage Example

The Itanium architecture provides a 64-bit counter to provide information on how many execution cycles a given logical processor is getting. It is similar to the Interval Timer (ITC, AR 44), except that it is clocked only when the logical processor is active. Optimizations such as hardware multi-threading and processor virtualization may cause a logical processor to sometimes be inactive. The Resource Utilization Counter allows for better cycle accounting for logical processors, given these types of optimizations.

RUC should only be written by Virtual Machine Monitors; other Operating Systems should not write to RUC, but should only read it.

10.5.7 Local Redirection Example

The Local Redirection Registers (LRR0-1) serves to steer external signal-based interrupts that are directly connected to the processor. LRR0 and LRR1 control the external interrupt signals (pins) referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1) respectively. The example below shows how to mask interrupt delivery on LINT0.

```
movl r18=(1<<16)
;;
mov cr.lrr0=r18
;;
srlz.d   // srlz.d is required after LRR write to ensure write effect
```

Note: LINT0 and LINT1 pins are not required to be supported. Writes to LRR0-1 control registers would have not effect, and reads from LRR0-1 control registers would return 0.

10.5.8 Inter-processor Interrupts Layout and Example

A processor generates an inter-processor interrupt (IPI) by storing a 64-bit interrupt command to an 8-byte aligned address in the Interrupt delivery region of the Processor Interrupt block. The address being stored to determines what target processor receives the IPI. The example below is an example of sending an interrupt to a specific processor based on the destination ID passed in. The destination ID consists of the Local interrupt ID and the Extended interrupt ID.

Writing to improperly aligned addresses in the delivery region or failure to store less than 64 bits can result in an invalid operation fault. The access must be uncacheable in order to generate an IPI.

```
//
// send_ipi_physical (dest_id, vector)
//
// inputs:     processor destination ID vector to send
//             (Local ID (8 bits << 8)| EID ( 8 bits))
//
//
//

.global ipi_block             // pointer to processor I/O block

IPI_DEST_EID=0x4

ENTRY(send_ipi_physical)
        alloc r19=ar.pfs,2,0,0,0
        movl r17=ipi_block;;
        ld8 r17=[r17]         // get pointer to processor block
        shl r21=r32,IPI_DEST_EID;;
        add r20=r21,r17;;     // point to proper processor
        st8.rel [r20]=r33     // send the IPI
        br.ret.sptk b0;;

END(send_ipi_physical)
```

10.5.9 INTA Example

External interrupt controllers, that are compatible with the Intel 8259A interrupt controller can not issue interrupt messages, so the vector number is not available at the time of the interrupt request. When an interrupt is accepted the software must check to see if it came from an external controller by the vector number (via IVR) to see if it is the ExtINT vector.

Once the software determines it is an ExtINT, it must obtain the actual vector by doing an uncached 1-byte load from the INTA byte located in the upper half of the processor interrupt block, offset 0x1e0000 from the base.

```
EXTINT=r0
INTA_PHYS_ADDRESS=0x80000000fefe0000
inta_address=r31

        movl inta_address=INTA_PHYS_ADDRESS
        ;;
        srlz.d             // make sure everything is up to date
        mov r14 = cr.ivr   // read ivr
        ;;
        srlz.d             // serialize before the EOI is written...
        ;;
        cmp.ne  p1,p2 = EXTINT,r14 ;;
    (p1)br.cond.sptk process_interrupt
        ;;

//
// A single byte load from the INTA address should cause
// the processor to emit the INTA cycle on the processor
// system bus. Any Intel 8259A compatible external interrupt
// controller must respond with the actual interrupt
// vector number as the data to be loaded.
//
//
        ld1 r17 = [inta_address]   // get the real vector..
        ;;
// vector obtained

process_interrupt:
```

§

I/O Architecture 11

I/O devices can be accessed from Itanium architecture-based programs using regular loads and stores to uncacheable space. While cacheable Itanium memory references may be reordered by the processor, uncacheable I/O references are always presented to the platform in program order. This "sequentiality" of uncacheable references is discussed in Section 2.2.2, "Memory Attributes" on page 2:524 and in more detail in Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82.

Additionally, uncacheable memory pages are defined to be "non-speculative" which causes all data and control speculative loads to uncacheable pages to defer. Control speculative loads to uncacheable memory return a NaT/NaTVal to their target register. Data speculative loads to uncacheable memory return zero to their target register. For details, refer to Section 4.4.6, "Speculation Attributes" on page 2:79.

When configuring chipset registers or setting up device registers, it is sometimes required to know when a memory transaction has been completed. Completion means the processor received acknowledgment that the transaction finished successfully in the platform, and that all its side-effects have occurred and will be visible to the next memory operation (issued by the same processor). To ensure completion of prior accesses on the platform, the Itanium architecture provides the `mf.a` instruction. Unlike the `mf` instruction that waits for **visibility** of prior operations, the `mf.a` waits for **completion** of prior operations on the platform. More details in Section 11.1.

To fully leverage the large set of existing platform infrastructure and I/O devices, the architecture also supports the IA-32 platform I/O port space. The Itanium instruction set does not provide IN and OUT instructions, but they can be emulated. The I/O port space can be mapped into user-space, and IA-32 applications can use IN and OUT instructions to directly communicate with the I/O port space. More details in Section 11.2.

The Itanium architecture provides a high-performance, high-bandwidth uncacheable memory attribute that supports write-coalescing. This allows the processor to burst writes to uncacheable locations at much higher bandwidth. The Itanium architecture does **not** guarantee the FIFO delivery of write-coalescing stores. More details in Section 4.4.5, "Coalescing Attribute" on page 2:78.

11.1 Memory Acceptance Fence (mf.a)

An `mf` instruction ensures that all cache coherent agents have observed all prior memory operations made by the processor issuing the `mf`. However, it does **not** ensure that those operations have completed, in the Itanium architecture parlance it does not ensure that they have been "accepted" by the external platform. For instance, a load may have been made visible to all processors by snooping their caches, but the data return may still be in progress. Such a load would be visible, but not complete.

The mf.a instruction on the other hand ensures that all prior data memory references made by the processor issuing the mf.a have been “accepted” by the external platform. However by itself the mf.a does not guarantee that all cache coherent agents have observed all prior memory operations. For instance, an uncacheable store to a chipset register may have completed on the system bus, however, that does not entail that all prior cacheable transactions (from the processor issuing the store) have been observed by all other processors in the coherence domain.

If software needs to ensure that all prior memory operations have been accepted by the platform **and** have been observed by all cache coherent agents, both an `mf.a` and an `mf` instruction must be issued. The `mf.a` must be issued first, and the `mf` must be issued second. For more details on memory ordering between cache coherent agents please refer to Chapter 2, “MP Coherence and Synchronization.”

Typically `mf.a` is used to configure a system’s I/O space, e.g. to setup chipset registers that affect all subsequent memory operations. Specifically, the mf.a instruction restrains further data accesses from initiating on the external platform interface until:

1. All previous sequential (i.e. non write-coalescing uncacheable) loads have been returned data, and
2. All previous stores have been “accepted” by the platform. Typically acceptance is indicated by a bus-specific signals/phase, e.g. completion of response phase on the system bus.

Architecturally, the definition of “acceptance” is platform dependent. The next section discusses the usage of the `mf.a` instruction in the context of the I/O port space.

11.2 I/O Port Space

IA-32 processors support two I/O models: memory mapped I/O and the 64KB I/O port space. To support IA-32 platforms, the Itanium architecture allows operating systems to map the 64KB I/O port space into the 64-bit virtual address space. This allows Itanium architecture-based operating systems to see all I/O devices as a single unified memory mapped I/O model, and permits “normal” Itanium load and store instructions as well as IA-32 IN and OUT instructions to directly access the I/O port space.

As described in Section 10.7, “I/O Port Space Model” on page 2:267, Itanium architecture-based operating systems can map the physical 64KB I/O port space into a spread-out 64MB block of virtual address space. The virtual base address of the I/O port space (IOBase) is maintained by the operating system in kernel register KR0. When the processor issues Itanium load and stores accesses to the I/O port space, a port’s virtual address is computed as:

```
port_virtual_address = IOBase | (port{15:2}<<12) | port{11:0}
```

For Itanium loads and stores, this address computation places four 1-byte ports on each 4KB page and expands the space to 64MB, with the ports being at a relative offset specified by port{11:0} within each 4KB virtual page. When executing an IA-32 IN or OUT instruction a processor based on the Itanium architecture automatically converts the IA-32 address to the appropriate expanded I/O port space address.

As a result of the spreading-out of the I/O ports into individual 4KB pages, Itanium architecture-based operating system code can control IA-32 IN, OUT instruction and IA-32 or Itanium load/store accessibility to blocks of 4 virtual I/O ports using the TLBs. This allows Itanium architecture-based operating systems to securely map devices that inhabit the I/O port space to different Itanium architecture-based device drivers or to user-space Itanium architecture-based applications.

Itanium architecture-based operating systems must ensure that the I/O port space is always mapped as uncacheable memory, and that Itanium architecture-based software only issues aligned 1, 2 or 4 byte references to I/O port space, otherwise device behavior is undefined.

When porting an IA-32 device driver to the Itanium architecture it can be useful to emulate the behavior of IA-32 IN and OUT instructions. The following code examples should be used for this purpose, since they enforce the strict memory ordering and platform acceptance requirements that IA-32 IN and OUT instructions are subject to. The following Itanium architecture-based assembly code outb (out byte) and inb (in byte) examples assume that the io_port_base is the virtual address mapping pointer set up by the IA_64 operating system. An `mf.a` instruction is used to verify acceptance by the platform before returning to the calling routine. Interrupts would expected to be disabled if these routines are called from user mode. This is for possible issues with process migration after servicing an interrupt.

```
//
// void outb(unsigned char *io_port,unsigned char byte)
//
//Output a byte to an I/O port.
//
ENTRY(outb)
    base_addr = r16
    port_addr = r17
    port_offset = r18
    mask = r19

    alloc   r13 = ar.pfs, 2, 0, 0, 0        // 2 in, 0 local, 0 out, 0 rot
    movl    base_addr = io_port_base
    extr.u  port_offset = in0, 2, 14
    mov     mask = 0xfff
    ;;
    ld8     port_addr = [base_addr]
    shl     port_offset = port_offset, 12
    and     in0 = mask, in0
    ;;
    add     port_offset = port_offset, in0
    ;;
    mf
    add     port_addr = port_addr, port_offset
    ;;
    st1.rel [port_addr] = in1
    mf.a
    mf
    br.ret.spnt.few rp
END(outb)

//
// unsigned char inb(unsigned char *io_port)
//
// Input a byte from an I/O port.
//
ENTRY(inb)
    base_addr = r16
    port_addr = r17
    port_offset = r18
    mask = r19

    alloc   r13 = ar.pfs, 2, 0, 0, 0        // 2 in, 0 local, 0 out, 0 rot
    movl    base_addr = io_port_base
    extr.u  port_offset = in0, 2, 14
    mov     mask = 0xfff
    ;;
    ld8     port_addr = [base_addr]
    shl     port_offset = port_offset, 12
    and     in0 = mask, in0
    ;;
    add     port_offset = port_offset, in0
    ;;
    mf
    add     port_addr = port_addr, port_offset
    ;;
    ld1.acq r8 = [port_addr]
    mf.a
    mf
    br.ret.spnt.few rp
END(inb)
```

§

Performance Monitoring Support 12

Processors based on the Itanium architecture include a minimum of four performance counters which can be programmed to count processor events. These event counts can be used to analyze both hardware and software performance. Performance counters can be configured to generate a counter overflow interrupt. This interrupt can be used for event- or time-based profiling. For hot-spot analysis of running code, performance monitor interrupts can be used to create a profile of frequently occurring instruction pointers (IP). Another common use of event counts is to compute processor performance metrics such as cycles per instructions (CPI), the current branch, cache or TLB miss rates, etc.

The Itanium architecture provides architected support for context switching of performance monitors by an Itanium architecture-based operating system. If supported by the operating system, this allows performance counter events to be broken down per thread or per process which is important for effective performance tuning of Itanium architecture-based applications.

The remainder of this chapter reviews the architected performance monitoring mechanisms. It also discusses the Itanium architecture-based operating system support needed for two monitoring usage models: per process/thread and system-wide event monitoring.

12.1 Architected Performance Monitoring Mechanisms

As defined in Section 7.2, “Performance Monitoring” on page 2:155, processors based on the Itanium architecture provide a minimum of four generic performance counter pairs (PMC/PMD[4..7]). The performance monitor control (PMC) registers are used to select the event to be counted, and to define under what conditions the event should qualify for being counted (for details refer to Section 7.2.1, “Generic Performance Counter Registers” on page 2:156). The performance monitor data (PMD) registers contain the event count or data.

The PMC/PMD registers can only be written by privileged software (PSR.cpl must be zero). A counter can be configured as a “privileged” counter or a “user-level” counter by setting of the PMC[i].pm bit. Privileged counters can only read at privilege level 0, while user-level counters can by read by user mode code (unless the operating system has explicitly disabled the user-level monitor reads using PSR.sp).

Once the PMC/PMD registers have been configured, counting is enabled and disabled by setting bits in the PSR. User-level counters can be controlled at user-level using the rum and sum instructions to toggle PSR.up. Privileged counters are controlled by privileged software using the rsm, ssm, mov from/to PSR instructions to toggle PSR.pp. Counting for all counters is further controlled by the PMC[0] freeze bit. When PMC[0].fr is 0, all counters are disabled. When PMC[0].fr is 1, counting is enabled based on PMC[i].pm, PSR.pp and PSR.up. For more details on controlling of the performance monitors please refer to Section 7.2.1, “Generic Performance Counter Registers” on page 2:156.

The PAL firmware provides information about the performance monitor registers that are implemented on the processor through the PAL_PERF_MON_INFO PAL call. Information provided by the PAL includes bit masks which indicate which PMC/PMD registers are implemented on this processor model, as well as the implemented number of generic PMC/PMD pairs, and the counter width of the generic counters.

12.2 Operating System Support

The monitoring mechanisms discussed in the previous section support two performance monitoring usage models that need support from an Itanium architecture-based operating system.

- Per Thread/Process Event Monitoring

To monitor processor events per thread the operating system needs to save and restore performance monitor state at thread/process context switches. This save/restore of PMC and PMD registers only needs to be done for monitored threads. The effect of the save/restore is that when a monitored thread is running, PMD reads will reflect events for the monitored thread/process only. Section 7.2.4.2, "Performance Monitor Context Switch" defines the steps required for per-thread context switch of performance monitors. It is worth noting that the PMC/PMD masks returned from PAL_PERF_MON_INFO indicate which PMC/PMD registers are implemented. The context switch routine can use the mask to save/restore implemented monitors without knowing the function of the monitors.

- System Wide Event Monitoring

To monitor processor events system wide (across all processes and the operating system kernel itself), a monitor must be enabled continuously across all contexts. This can be achieved by configuring a privileged monitor (PMC.pm=1), and by ensuring that PSR.pp and DCR.pp remain set for the duration of the monitor session. Since the operating system typically reloads PSR and possibly DCR on context switch, this requires the operating system to set PSR.pp and DCR.pp for all contexts that are active during the monitoring session. One way to accomplish this is to have code in the context switch routine to always set PSR.pp and DCR.pp when system wide monitoring is in effect. Another technique is to set the initial state for all new threads/processes to PSR.pp=1, PSR.up=0, PSR.sp=0 and DCR.pp=1. Setting the per thread PSR and DCR in this way ensures that privileged monitors will be enabled across all contexts. When system wide monitoring is in effect, PSR.pp, DCR.pp as well as the PMC and PMD registers should not be altered by the context switch routine.

To support both per thread and system wide monitoring, the operating system needs to be aware which type of monitoring is being performed at any given moment. If per thread/process monitoring is active, then the operating system must save/restore monitor state for monitored threads. If system wide monitoring is active, then the operating system must ensure that PSR.pp and DCR.pp remain set.

The preferred approach for performance monitoring is for Itanium architecture-based operating systems to provide a set of kernel mode services that allow performance monitoring software to be implemented in a loadable device driver. Such a loadable device driver can support various usage monitoring models, can be adapted to model-specific processor monitoring capabilities, and is a well-defined isolated and easily replaceable software component. The following operating system services allow a kernel mode device driver to take full advantage of the performance monitors:

- Allocation/Free Performance monitors – operating system should delegate management of the performance monitor resources to device driver.
- Process create/terminate notification – operating system should notify driver on process create/terminate.
- Thread create/terminate notification – operating system should notify driver on thread create/terminate.
- Context switch notification – operating system should notify driver on thread and process context switch. The driver will perform the required save/restore depending on the currently active usage model.
- Performance counter overflow interrupt – operating system should notify driver when a performance monitor overflow interrupt occurs.
- Get Current Process Identifier – returns a unique identifier for the current process or address space. This should be callable in any context, e.g. by an interrupt handler.
- Get Current Thread Identifier – returns a unique identifier for the current thread of execution. This should be callable in any context, e.g. by an interrupt handler.

One of the challenges when doing instruction pointer (IP) profiling is to relate the current IP to an executable binary module and to an instruction within that module. If appropriate symbol information is available, the IP can be mapped to a line of source code.

To support this IP to module mapping, it is recommended that the OS provide services to enumerate all kernel and user mode modules in memory, and to allow a kernel mode driver to be notified of each module load. The following services are recommended:

- Enumerate kernel mode modules – provides information each kernel mode module currently loaded in memory.
- Enumerate threads/processes – provides a list of current threads/processes. The list should include the unique identifier for each thread/process.
- Enumerate all user mode modules – provides information on each user mode module that is currently loaded in memory (all processes).
- Enumerate modules for a process – provides information on each user mode module that is currently loaded in memory for the selected process.
- Module load notification – OS should notify a driver when the OS loads a kernel or user mode module into memory for execution. The notification should occur before the module begins execution.

In the above services for module enumeration and load notification, the module information provided for a module should include module name, load address, size in bytes, section number (if a section of a module is loaded non-contiguously), and a process/thread identifier that identifies the process into which the module is loaded.

§

Firmware Overview 13

Itanium-based systems make use of several firmware components: Processor Abstraction Layer (PAL), System Abstraction Layer (SAL), Unified Extensible Firmware Interface (UEFI) and Advanced Configuration and Power Interface (ACPI).

The PAL and SAL components work together to handle the reset abort event. The reset abort handling performs processor and system initialization for operating system (OS) boot and provides an API to the operating system loader. The PAL and SAL firmware layers work together to handle machine check aborts (MCA), initialization events (INIT), and platform management interrupt (PMI) handling. All firmware components also provide runtime procedure calls to abstract processor and platform functions that may vary across implementations.

This chapter will provide an overview of the firmware components and how the firmware components interact with each other as well as with the operating system. For the full architecture specifications of the PAL firmware please refer to Chapter 11, "Processor Abstraction Layer." For full architecture specifications on SAL, UEFI and ACPI firmware components please refer to Section 1.2, "Related Documents" on page 2:505.

The PAL layer is developed by Intel Corporation and delivered with the processor. The SAL, UEFI and ACPI firmware is developed by the platform manufacturer and provide a means of supporting value added platform features from different vendors.

The interaction of the various functional firmware blocks with the processor, platform and operating system is shown in Figure 13-1, "Firmware Model" on page 2:624.

13.1 Processor Boot Flow Overview

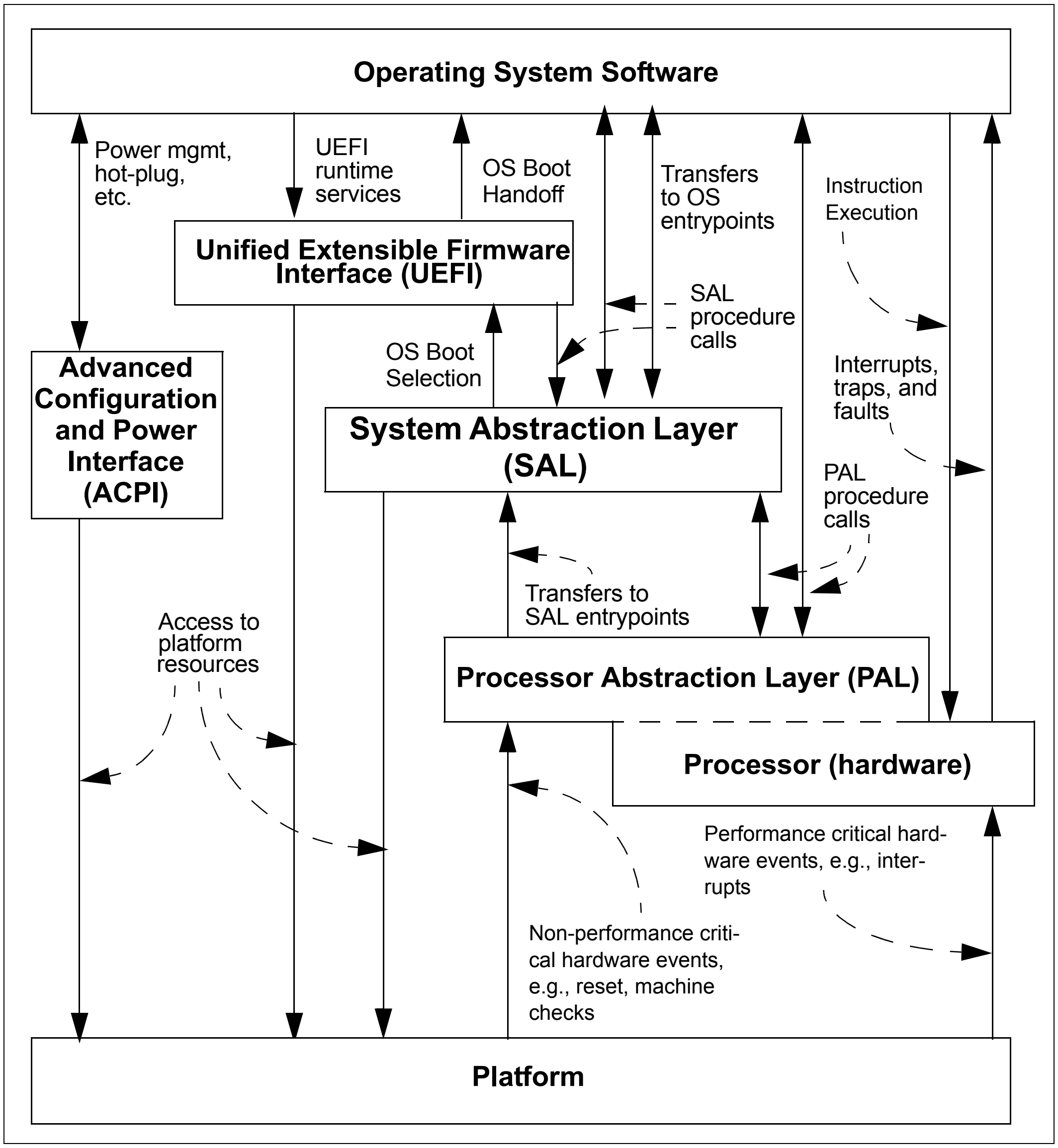
13.1.1 Firmware Boot Flow

Upon detection of a reset event on a processor based on the Itanium architecture, execution begins at an architected entry point inside of PAL. This PAL code will verify the integrity of the PAL code and may perform some basic processor testing. PAL will then branch to an entry point within the SAL firmware. This first branch to SAL is to determine if a firmware update is needed requiring re-programming of the firmware code. If no firmware update is needed SAL will branch back to PAL.

PAL now performs additional processor testing and initialization. These first processor tests are performed without platform memory. PAL indicates the outcome of the testing and branches to an entry point within SAL firmware for the second time. SAL will now begin platform testing and initialization. The exact division of work between SAL and UEFI from that point on is platform implementation dependent. It is required that the SAL runtime services, the UEFI boot and runtime services, and the ACPI tables and control methods be exposed to the operating systems for correct operation.

The order of steps within the UEFI/SAL firmware is platform implementation dependent and may vary. In general, the UEFI/SAL firmware selects a Bootstrap processor (BSP) in multiprocessor (MP) configurations early in the boot sequence. Next, UEFI/SAL will find and initialize memory and invoke PAL procedures to conduct additional processor tests to ensure the health of the processors. UEFI/SAL then initializes the system fabric and platform devices.

Figure 13-1. Firmware Model



Operating System Software
Power mgmt, hot-plug, etc.
UEFI runtime services
OS Boot Handoff
Transfers to OS entrypoints
Instruction Execution
Unified Extensible Firmware Interface (UEFI)
SAL procedure calls
Advanced Configuration and Power Interface (ACPI)
OS Boot Selection
Interrupts, traps, and faults
System Abstraction Layer (SAL)
PAL procedure calls
Transfers to SAL entrypoints
Access to platform resources
Processor Abstraction Layer (PAL)
Processor (hardware)
Performance critical hardware events, e.g., interrupts
Non-performance critical hardware events, e.g., reset, machine checks
Platform

The UEFI firmware may incorporate a Boot Manager. The UEFI firmware specification [UEFI] enables booting from a variety of mass storage devices such as hard disk, CD, DVD as well as remote boot via a network. At a minimum, one of the mass storage devices contains an UEFI system partition.

The UEFI Boot Manager displays the list of operating system choices and permits the user to select the operating system for booting. To support this functionality, the OS setup program stores the boot paths of the OS loaders and boot options in non-volatile storage managed by the UEFI firmware. The UEFI reserves the environment variables Boot#### (#### represents values 0000 to 0xFFFF) for this purpose. The OS setup program must also store the OS loader binary images within the UEFI System Partition. The UEFI Boot Manager will also allow the user to add boot options, delete boot options, launch an UEFI application, and set the auto-boot time out value.

The UEFI System Partition also contains UEFI drivers that may be loaded by the UEFI firmware prior to transfer of control to an OS loader. The floating-point software assist (FPSWA) library is included in a UEFI runtime driver. The FPSWA library may be invoked by the OS during floating-point exception faults and traps. Please see Section 8.1.1, "Software Assistance Exceptions (Faults and Traps)" on page 2:587 for more information on the usage of this library.

If the user elects to boot an Itanium architecture-based operating system, the UEFI loads the appropriate OS loader from the UEFI System Partition and passes control to it. The OS loader will load other files including the OS kernel from an OS partition using the UEFI boot services which provides an API interface to the OS loader.

The OS loader can obtain information about the memory map usage of the firmware by making the UEFI procedure call GetMemoryMap(). This procedure provides information related to the size and attributes of the memory regions currently used by firmware.

The OS loader will then jump to the OS kernel that takes control of the system. Until this point, system firmware retained control of key system resources such as the Interrupt Vector Table and provided the necessary interrupt, trap and fault handlers.

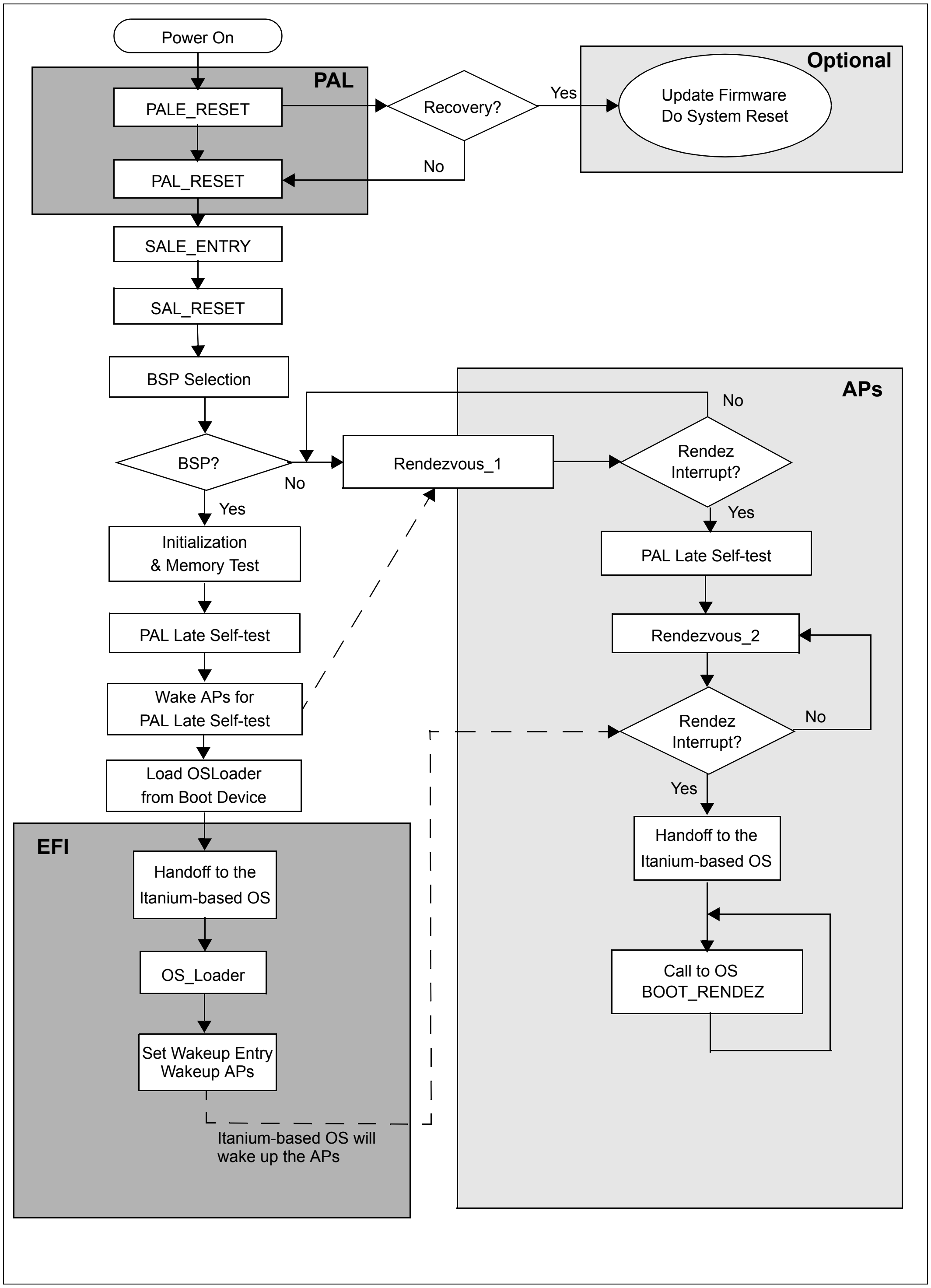
Figure 13-2, "Control Flow of Boot Process in a Multiprocessor Configuration" on page 2:626 depicts the booting steps in a MP configuration.

13.1.2 Operating System Boot Steps

The firmware will initialize the processor(s) and platform to a specific state before handing off to the operating system boot loader. The boot loader is then responsible for copying the operating system from some storage medium into memory for running. Once this is done the operating system will need to initialize some key registers before entering into a higher level language code such as C. This section will describe code that an OS will need to execute in order to initialize system registers for preparing an OS to run in virtual mode and handle interrupts. Appendix A, "Code Examples" provides the Itanium architecture-based sample assembly code described in this section.

Assuming the specific operating system boot loader hands off to the OS kernel in physical mode, the operating system should first disable interrupts and interrupt collection via the PSR. This is done to avoid taking external interrupts from timers, etc and also prepares for writing specific system registers that require PSR.ic to be 0 when written.

Figure 13-2. Control Flow of Boot Process in a Multiprocessor Configuration



Next the operating system startup code invalidates the ALAT via the `invala` instruction. The invala in complete form will invalidate all entries in the ALAT.

The register stack should be invalidated. This can be done by setting the Register Stack Configuration Register (RSC) to zero followed by a loadrs instruction. Setting the RSC to zero will put the register stack in enforced lazy mode and set the RSC.loadrs, load distance to tear point, to zero. The loadrs will invalidate all stacked registers outside current frame.

The region registers and protection key registers are then initialized with operating system implementation dependent values. For example, the OS will initialize the region register with a preferred page size. It would also disable the VHPT until it was ready for it. In the example, all region registers are initialized with an 8-KB page size.

An OS must setup a kernel stack pointer and backing store pointer for the register stack. The stack pointer (GR12) is set to the OS kernel stack area with scratch space to cover calling conventions. AR.RSC must be set to enforced lazy mode before writing to the bspstore register. Initializing the bspstore has effects on all three RSE pointers (BSP, BSPSTORE, and RSE.BspLoad).

In order for the operating systems to handle interruptions, the operating system interrupt vector table base address must be set up. The size of the vector table is 32K bytes and is 32K byte aligned. Setting the location of the table is accomplished by moving the address into CR.IVA.

Operating systems setup system address translations for the kernel text and data by using the translation insertion format described in Section 4.1.1.5, "Translation Insertion Format" on page 2:53. A combination of a general register, Interruption TLB Insertion Register (ITIR), and the Interruption Faulting Address register (IFA) are used to insert entries into the TLB. To void TLB faults on specific text and data areas the operating system can lock critical virtual memory translations in the TLB by use of Translation Register (TR) section of the TLB. The entries are placed into a TR via the Insert Translation Register (itr) instruction. The translation will remain unless the software issues the Purge Translation (ptr) instruction. Other important areas might be locked also, such as entries for memory mapped I/O, etc.

After the initial translations have been entered, the OS can make final preparations for enabling virtual addressing. The OS needs to set several important bits in the IPSR, such as data address translation (dt), register stack translation (rt), instruction address translation (it), enabling interruption collection (ic), and setting the specific register bank (bn).

The Default Control Register (DCR) specifies the default parameters for PSR values on interruption, some additional global controls, and whether speculative load faults can be deferred. The example defers all speculation faults. Also, if the operating system is utilizing the performance monitors then the DCR.pp bit should be set so that on interruption the PSR.pp bit will be set.

The global pointer (GR1) should point to the global data area. It must be setup properly before using higher level languages such as C. The startup code should also set the following registers to zero, the Interruption Function State (CR.IFS, to set frame marker to zero), and AR.RNAT (to make sure no NaT bits are set before OS kernel begins using the RSE.

Before enabling virtual addressing, the Interruption Instruction Bundle Pointer (IIP) is set to point a virtual address. This is done so when the return from interruption instruction (`rfi`) is executed the instruction fetched will have a virtual address. The rfi will switch modes based on IPSR values which are moved into the PSR. The IIP value becomes the new IP.

13.2 Runtime Procedure Calls

The PAL, SAL, and UEFI firmware components provide entry points as runtime interfaces to the OS. These runtime interfaces allow the OS to obtain information about the processor and platform as well as perform implementation-specific functions on the processor and platform.

The calling conventions for these runtime procedures are documented in the respective firmware architecture specifications. For PAL and SAL, the first input argument to the procedure call specifies the index of the procedure within the list of supported procedures for each firmware layer.

13.2.1 PAL Procedure Calls

PAL procedure calls are classified into two types: static and stacked. The static calls are intended for boot-time use before main memory is available or in error recovery situations where memory or the RSE may not be reliable. All parameters will be passed in the general registers GR28 to GR31 of Bank 1. The stacked registers (GR32 to GR127) will not be used for these calls. The static calls can be called at both boot-time and runtime.

Stacked register calls are intended for use after memory has been made available. The stacked registers are used for parameter passing and local variable allocation. These calls also allow memory pointers may be passed as arguments. These calls can be made at boot-time after memory has been tested and initialized as well as runtime.

For a listing of all the PAL procedures and their classification please see Section 11.10.1, "PAL Procedure Summary" on page 2:354.

All PAL calls are re-entrant and can be executed simultaneously on multiple processors.

13.2.1.1 Making a Static PAL Call

Since the static PAL calls do not use stacked registers, these calls are made as a pure jump with branch register B0 containing the address of the bundle to which control will return. The following code example describes how to make a static PAL call:

```
GetFeaturesCall:

mov r14 = ip // Get the ip of the current bundle
movl r28 = PAL_PROC_GET_FEATURES// Index of the PAL procedure
movl r4 = AddressOfPALProc;;// Address of the PAL proc entry point
ld8 r4 = [r4];;// Read address from local pointer
mov b5 = r4    // Move address into a branch register

// Compute the return address in a position independent manner

addl r14 = (BackHome - GetFeaturesCall),r14;;
mov b0 = r14 // b0 is the return link
mov r29 = r0 // Initialize rest of input arguments
mov r30 = r0 // to zero as required by the
mov r31 = r0 // architecture.

br.sptk b5;; // Make the PAL call.

// PAL will return here when the call is completed

BackHome:
```

The sample code is position independent and functions in both physical and virtual addressing modes. Since the return address is evaluated by using the runtime instruction pointer (IP value), it will run from any address. This attribute is important for any relocatable code.

The address of the PAL procedure entry point is passed to SAL at the hand-off from PAL to SAL during reset. SAL will pass this information on to the OS during OS boot as well.

13.2.1.2 Making a Stacked PAL Call

A stacked PAL call uses the stacked registers for argument passing and local variable allocation. The stacked PAL calls conform to the calling conventions document [SWC], with the exception that general register GR28 must also contain the function index input argument. The following code example describes how to make a stacked PAL call.

```
movl r4 = AddressOfPALProc;;// Address of the PAL proc entry point
ld8 r4 = [r4];;// Read address from local pointer
mov b5 = r4  // Move address into a branch register

// Make the PAL_HALT_INFO procedure call. PAL_HALT_INFO uses stacked
register
// convention and parameters are passed with in0-in3

mov r28 = PAL_HALT_INFO;;// Index of the PAL procedure
mov out0 = r28// r28 and in0 must both contain the
               // index value for stacked PAL calls.
mov out1 = ScratchMem_Pointer// Pointer to the memory argument
mov out2 = 0x0// Write zero to unused input arguments
mov out3 = 0x0

br.call.sptk.few b0 = b5;;// PAL stacked call

// PAL will return here when the call is completed
```

13.2.1.3 PAL Procedure Calls and Performance

PAL procedure calls are designed for a number of different functions varying from boot-time usage before platform memory is available to processor-specific functions used during runtime by the OS. PAL runtime procedure calls made by the OS are designed to be flexible with minimal overhead. The following features aid in this goal:

- PAL procedure calls are relocatable. This feature is useful for platforms that have PAL stored in non-volatile storage, such as flash. During OS boot the PAL procedures are copied into RAM which will reduce the memory latency.
- A number of PAL procedure calls are defined to be called in both physical and virtual addressing. This allows the caller to make the call in its currently executing addressing mode, thus reducing the need to switch between physical and virtual addressing.

13.2.2 SAL Procedure Calls

All SAL procedure calls use the stacked register calling convention. SAL follows the floating-point register conventions specified in the calling conventions document [SWC], with the exception that SAL does not use the floating-point registers FR32 to FR127. This exception eliminates the need for the OS to save these registers across SAL procedure calls.

SAL procedures are non re-entrant. The OS is required to enforce single threaded access to the SAL procedures except for the following procedures:

- SAL_MC_RENDEZ, SAL_CACHE_INIT, SAL_CACHE_FLUSH

13.2.3 UEFI Procedure Calls

UEFI procedure calls are classified into the following two categories: boot services and runtime services. The UEFI boot services execute in physical addressing mode only. The runtime services can execute in either physical or virtual addressing mode. The UEFI boot services are only available during the boot process and are terminated by a call to the EfiExitBootServices() procedure. After this call, UEFI boot services may no longer be invoked by the OS. The UEFI runtime services execute in physical mode until the OS invokes the EFISetVirtualAddress() function to switch the UEFI to virtual mode. After this point, the UEFI runtime services may be invoked in virtual mode only. For full information on all the UEFI boot and runtime services please refer to the UEFI specification [UEFI].

13.2.4 ACPI Control Methods

Advanced Configuration and Power Interface (ACPI) firmware provides a method of reporting system resources (up to the boundary of the box) to the operating systems. ACPI uses tables to describe system information, features, and methods for controlling those features. The ACPI tables list devices on the system board, devices that cannot be detected by bus walks, and devices which require the OS for power or temperature management. The ACPI control methods use a pseudo-code language called AML (ACPI Machine Language). AML is a tokenized language. The OS contains and uses an AML interpreter that interprets and executes these methods stored in the ACPI tables.

13.2.5 Physical and Virtual Addressing Mode Considerations

All of the PAL procedures can be called in the physical addressing mode. A subset of PAL calls can be made using the virtual addressing mode. For PAL calls that can be invoked using virtual addressing mode, it is the responsibility of the caller to map these PAL procedures with an ITR as well as either a DTR or DTC. If the caller chooses to map the PAL procedures using a DTC it must be able to handle TLB faults that could occur. See Section 11.10.1, “PAL Procedure Summary” for a summary of all PAL procedures and the calling conventions.

The SAL and UEFI firmware layers have been designed to operate in virtual addressing mode. UEFI provides an interface to the OS loader that describes the physical memory addresses used by firmware and indicates whether the virtual address of such areas need to be registered by the OS with UEFI. The UEFI Specification [UEFI] also provides the interfaces for the OS to register the virtual address mappings. In a MP configuration, the virtual addresses registered by the OS must be valid globally on all the processors in the system.

The SAL runtime services may be called either in virtual or physical addressing mode. SAL procedures that execute during machine check, INIT, and PMI handling must be invoked in physical addressing mode.

The parameters passed to the firmware runtime services must be consistent with the addressing environment, i.e. PSR.dt, PSR.rt setting. Additionally, the global pointer (gp) register [SWC] must contain the physical or virtual address for use by the firmware.

13.2.5.1 SAL Procedures that Invoke PAL Procedures

Some of the SAL runtime services, e.g. SAL_CACHE_FLUSH, will need to invoke PAL procedures. While invoking these SAL procedures in virtual mode, the OS must provide the appropriate translation resources required by PAL (i.e. ITR and DTC covering the PAL code area).

In general, if SAL needs to invoke a PAL procedure, it will do so in the same addressing mode in which it was called by the OS (i.e. without changing the PSR.dt, PSR.rt, and PSR.it bits). If a particular PAL procedure can only be invoked in physical mode, SAL will turn off translations and then invoke the PAL procedure. SAL will then restore translations before returning to the caller. The PAL_CACHE_INIT procedure invoked by the SAL_CACHE_INIT is an example of a procedure that would require such an addressing mode transition.

13.3 Event Handling in Firmware

The PAL and SAL firmware layers are responsible for handling three events. These events are the machine check abort (MCA), the initialization event (INIT) and the platform management interrupt (PMI). When the processor detects these events it will pass control to PAL for handling. The following sections describe the high level overview of the firmware handling of these events.

13.3.1 Machine Check Abort (MCA) Flows

In order to have a highly reliable and fault tolerant computing environment a great deal of coordination and cooperation between the system entities (i.e. the processor, platform, and system software) is required. The PAL firmware, the SAL firmware, and the operating system all work together to meet this goal. This section will provide an overview of the machine check abort handling.

When the processor detects an error, control is transferred to the PAL_CHECK entrypoint. PAL_CHECK will perform error analysis and processor error correction where possible. Subsequently, PAL either returns to the interrupted context or hands off control to the SAL_CHECK component. The level of recovery provided by PAL_CHECK is implementation dependant and is beyond the scope of this specification. SAL_CHECK will perform error logging and platform error correction where possible. Errors that are corrected by PAL and SAL firmware are logged and control is transferred back to the interrupted process/context. For corrected errors, no OS intervention is required for error handling, but the OS is notified of the event for logging purposes through a low priority asynchronous corrected machine check interrupt (CMCI). See Section 5.8.3.8, "Corrected Machine Check Vector (CMCV – CR74)" for more information on the CMCI. If the error was not corrected by firmware, SAL hands off control to the OS_MCA handler.

Within the firmware the entire machine check is handled with virtual address translations disabled. However, the OS machine check handler may optionally enable virtual addressing and execute most of MCA handler in virtual mode.

Figure 13-3 and Figure 13-4 depict an overview of Itanium machine check processing. The control flows are slightly different for corrected and uncorrected machine checks.

Figure 13-3. Correctable Machine Check Code Flow

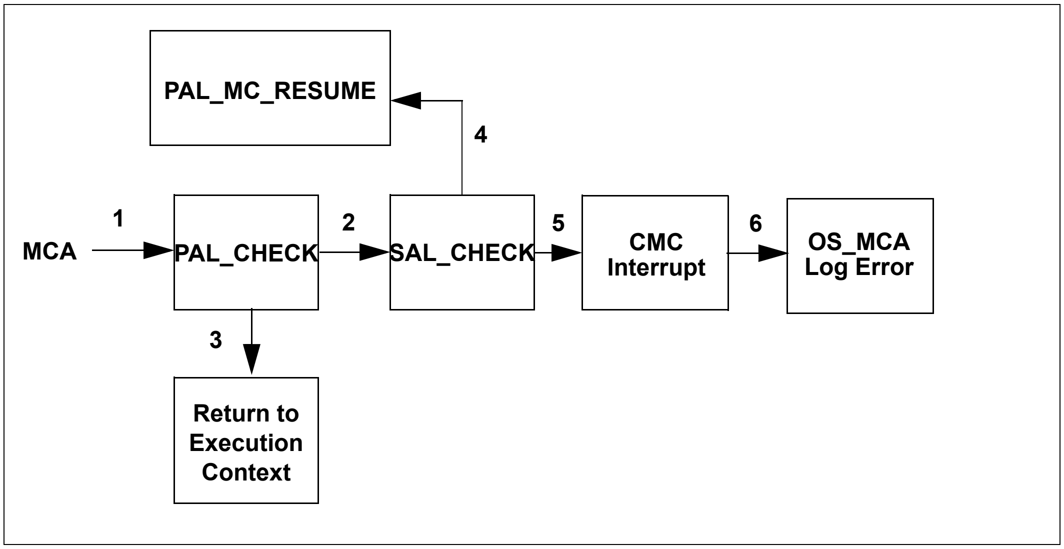
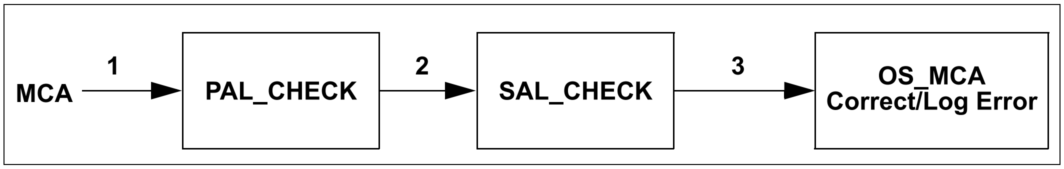


Figure 13-4. Uncorrectable Machine Check Code Flow



For multiprocessor systems, machine checks are classified as local and global. A global MCA implies a system wide broadcast by hardware of an error condition. During a global MCA condition, all the processors in the system will be notified of the MCA, detected by one or more system components, and each of the processors in the system will start processing the MCA in their respective handlers. The SAL firmware and OS layers will coordinate the handling of the error among the processors.

A local MCA has a scope of influence that is limited to the particular processor which encountered the error. This local MCA will not be broadcast to other processors in the system and will be handled on an individual processor basis. At any point in time, more than one processor in the system may experience a local MCA and handle it without notifying other processors in the system.

The next sections will provide an overview of the responsibilities that the PAL, SAL and OS have for handling machine checks. These sections are not an exhaustive description of the functionality of the handlers but provides a high level description of how the MCA handling is split among the different components.

13.3.1.1 Machine Check Handling in PAL

All machine check abort events are first handled in the PAL firmware layer. The following provides a brief description of some of the functions of the PAL machine check handler:

- Correct processor errors if possible.

- Attempt to contain the error by requesting a rendezvous for all processors in the system if needed.
- Hand off control to SAL for further processing, such as error logging.
- Return processor error log information upon request by SAL.
- Return to the interrupted context by restoring the state of the processor.
- Notify the OS about corrected machine check conditions through the CMC interrupt.

13.3.1.2 Machine Check Handling in SAL

Before SAL is ready to handle machine checks, it must register with PAL an uncacheable memory buffer that PAL can use to save away processor state. This area is known as the min-state save area. If a machine check occurs before this memory location has been registered, return to the interrupted context is not possible and the machine check is not recoverable.

The following provides a description of some of the functions of the SAL machine check handler.

- Attempt to rendezvous the other processors in the system on a PAL request.
- Process MCA handling after handoff from PAL.
- Retrieve processor error log information via PAL procedure calls and store this information for logging purposes.
- Issue a PAL clear log request to clear the processor error logs, which enables further logging.
- Log platform state for MCA and retain it until it is retrieved by the OS.
- Attempt to correct processor machine check errors which are not corrected by PAL.
- Attempt to correct platform machine check errors.
- Branch to the OS MCA handler for uncorrected errors or optionally reset the system.
- Return to the interrupted context via a PAL procedure call.

13.3.1.3 Machine Check Abort Handling in OS

Before the OS kernel is ready to handle machine checks, it must register the address of the OS_MCA entry point and the GP [SWC] value for the OS_MCA handler with SAL. If the OS does not register its entry point, the occurrence of a machine check will cause a system reset. In MP configurations, the OS must also register with SAL:

- A rendezvous interrupt vector which SAL firmware can use to rendezvous the processors.
- The mechanism that the OS will employ to wake up the processors at the end of machine check processing.

When the OS registers the OS_MCA entry point with SAL, it also supplies the length of the code (or at least the length of the first level OS_MCA handler). SAL computes and saves the checksum of this code area. Prior to entering OS_MCA, SAL ensures that the OS_MCA vector is valid by verifying the checksum of the OS_MCA code. Hence, the OS_MCA code must not contain any self modifying code.

When an uncorrected machine check event occurs, SAL will invoke the OS_MCA handler. The functionality of this handler is dependent on the OS. At a minimum, it must call a SAL procedure to retrieve the error logging and state information and then call another SAL procedure to release these resources for future error logging and state save.

When the OS_MCA code completes, it decides whether or not to return to the interrupted context. The OS must take into account the state information retrieved from the SAL with respect to the continuability of the processor and system. Thus, even if the OS could correct the error, if PAL or SAL reports that it did not capture the entire processor context, resumption of the interrupted context will not be possible.

The OS must also determine from values stored by PAL in the min-state save area whether the machine check occurred while operating with PSR.ic set to 0 and whether the processor supports recovery for this case. Please refer to Section 11.3.1.1, "Resources Required for Machine Check and Initialization Event Recovery" for more information on processor recovery under this condition.

To provide better software error handling, some operating systems build mechanisms to identify whether machine checks occurred during execution of the OS kernel code or in the application context. One technique to achieve this is to call the PAL_MC_DRAIN procedure when an application makes a system call to the OS. This procedure completes all outstanding transactions within the processor and reports any pending machine checks. This technique impacts system call and interrupt handling performance significantly, but will improve system reliability by allowing the OS to recover from more errors than if this mechanism was not included.

13.3.2 INIT Flows

INIT is an initialization event generated by the platform or by software through an inter-processor interrupt message. The INIT can be due to a platform INIT event or due to a failed rendezvous on an application processor.

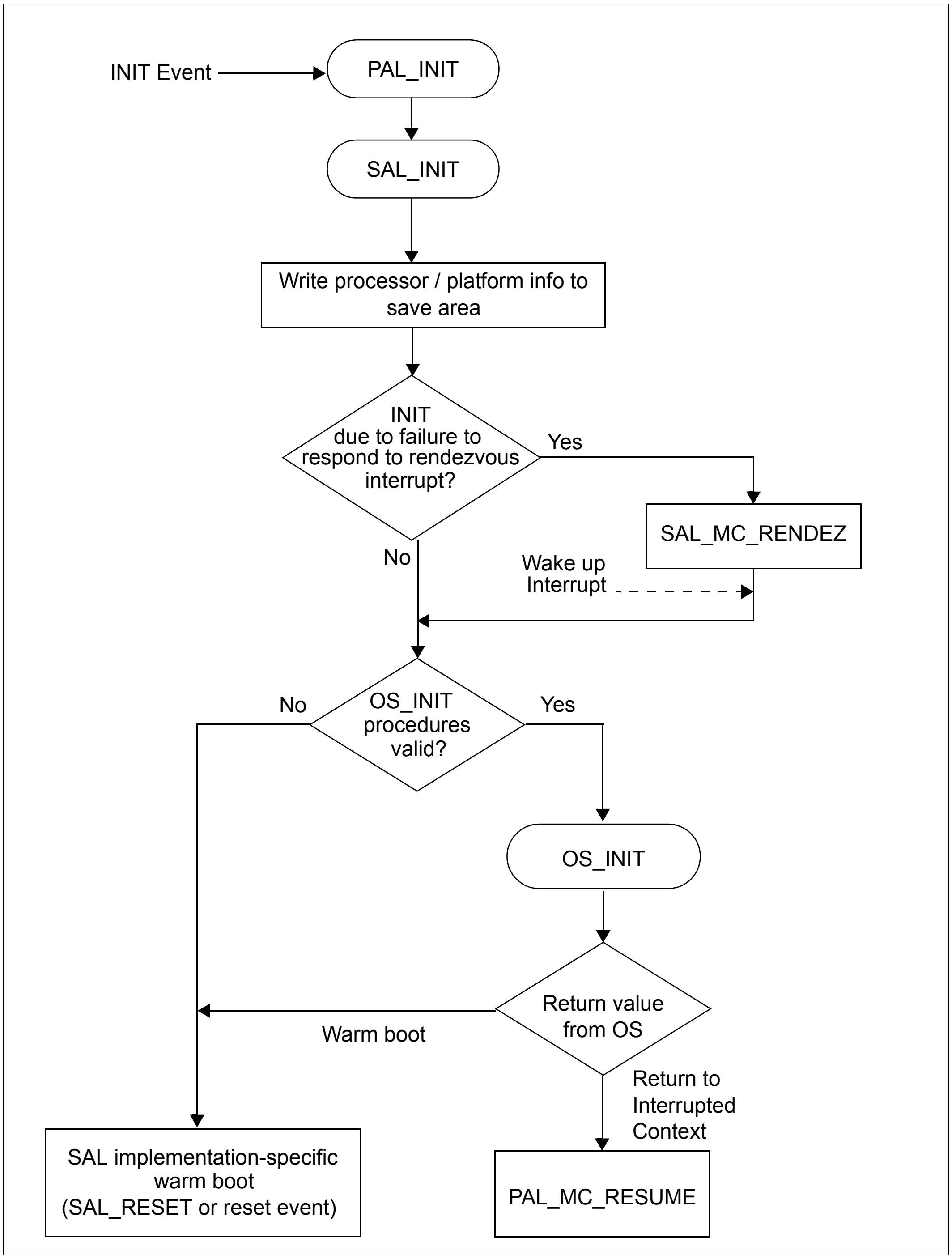
The INIT event will pass control to the PAL firmware INIT handler. The PAL INIT handler saves processor state to the registered min-state save area and sets up the architected hand off state before branching to SAL. See Section 11.5, "Platform Management Interrupt (PMI)" for more information on the PAL INIT handling.

The SAL INIT handler logs processor state and platform state information and then calls the OS_INIT handler if one is registered. The OS_INIT handler gains control in physical mode but may switch to virtual mode if necessary. The OS may choose to implement a crash dump or an interactive debugger within the OS_INIT handler.

The OS must register the OS_INIT entry point with SAL, otherwise the occurrence of an INIT event will cause a system reset. At the end of OS_INIT handling, the OS must return to SAL with the appropriate exit status.

Figure 13-5 illustrates the flow of control during INIT processing.

Figure 13-5. INIT Flow


INIT Event
PAL_INIT
SAL_INIT
Write processor / platform info to save area
INIT due to failure to respond to rendezvous interrupt?
Yes
SAL_MC_RENDEZ
No
Wake up Interrupt
OS_INIT procedures valid?
No
Yes
OS_INIT
Return value from OS
Warm boot
Return to Interrupted Context
SAL implementation-specific warm boot (SAL_RESET or reset event)
PAL_MC_RESUME

13.3.3 PMI Flows

Processors based on the Itanium architecture implement the Platform Management Interrupt (PMI) to enable platform developers to provide high level system functions, such as power management and security, in a manner that is transparent not only to the application software but also to the operating system.

When the processor detects a PMI event it will transfer control to the registered PAL PMI entrypoint. PAL will set up the hand off state which includes the vector information for the PMI and hand off control to the registered SAL PMI handler. To reduce the PMI overhead time, the PAL PMI handler will not save any processor architectural state to memory. Please see Section 11.5, "Platform Management Interrupt (PMI)" for more information on PAL PMI handling.

The SAL PMI handler may choose to save some additional register state to SAL allocated memory to handle the specific platform event that generated the PMI.

The OS will not see the PMI events generated by the platform. The platform developer can use PMI interrupts to provide features to differentiate their platform.

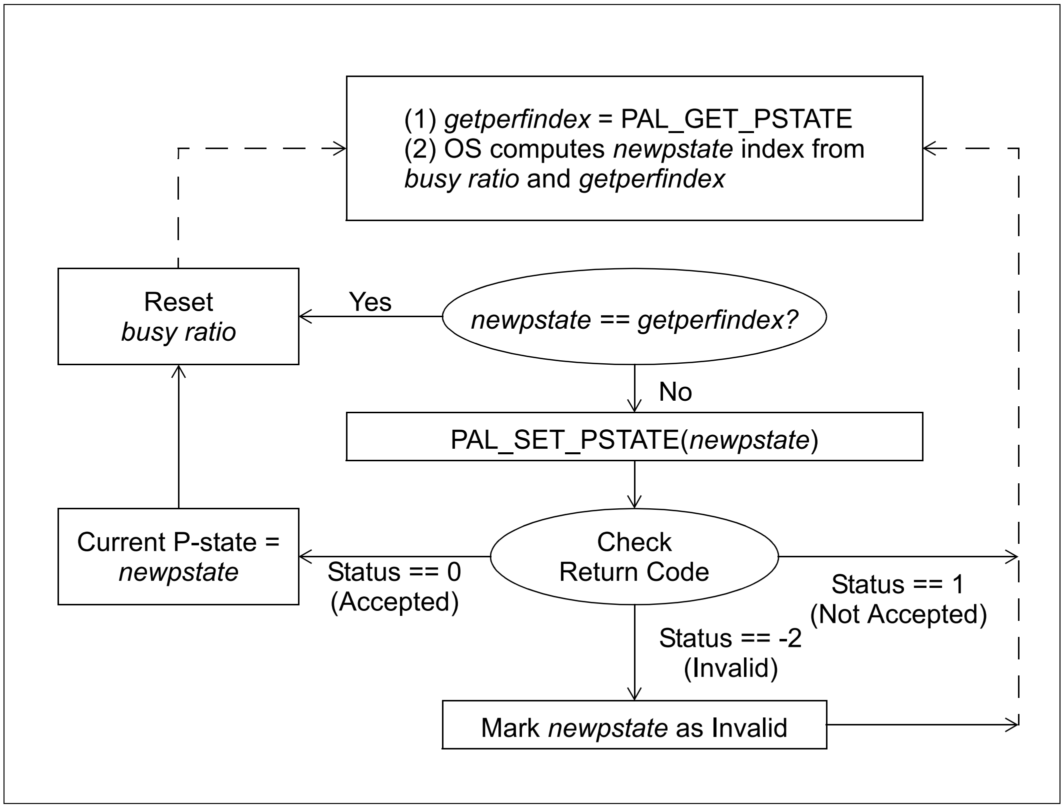
PMI handling was designed to be executed with minimal overhead. The SAL firmware code copies the PAL and SAL PMI handlers to RAM during system reset and registers these entry-points with the processor. This code is then run with the cacheable memory attribute to improve performance.

Depending on the implementation and the platform, there may be no special hardware protection of the PMI code's memory area in RAM, and the protection of this code space may be through the OS memory management's paging mechanism. SAL sets the correct attributes for this memory space and passes this information to the OS through the Memory Descriptor Table from EfiGetMemoryMap() [UEFI].

13.3.4 P-state Feedback Mechanism Flow Diagram

The example flowchart shown below illustrates how the caller can utilize the PAL_SET_PSTATE and the PAL_GET_PSTATE procedures to manage system utilization and power consumption, for a processor implementation that belongs to either a hardware-coordinated dependency domain or a hardware-independent dependency domain. At the beginning of the loop, PAL_GET_PSTATE gives the performance characteristics of the processor over the last time period. It is assumed that the caller maintains an internal count for determining the busy ratio of the logical processor (busy ratio can be defined as the percentage of time the processor was busy executing instructions and not idle). The caller then seeks to adjust the P-state for the next time period to match the busy ratio from the previous time period. For example, if the busy ratio for a given period was 100%, and the *performance_index* returned by PAL_GET_PSTATE was 60, then this indicates that the P-state for the next time period should be P0 (which has performance index of 100). The caller would then call the PAL_SET_PSTATE procedure to transition the processor to the P0 state. In essence, if the busy ratio is greater than the *performance_index* returned by PAL_GET_PSTATE, the caller responds to the increased demand requirement of the workload by transitioning the processor to a higher-performance P-state. Alternatively, if the busy ratio is lower than the *performance_index* returned by PAL_GET_PSTATE, the caller responds by transitioning the processor to a lower performance P-state, which consumes less power and operates at reduced performance.

Figure 13-6. Flowchart Showing P-state Feedback Policy


(1) *getperfindex* = PAL_GET_PSTATE
(2) OS computes *newpstate* index from
busy ratio and *getperfindex*
Reset
busy ratio
Yes
newpstate == getperfindex?
No
PAL_SET_PSTATE(*newpstate*)
Current P-state =
newpstate
Check
Return Code
Status == 0
(Accepted)
Status == 1
(Not Accepted)
Status == -2
(Invalid)
Mark *newpstate* as Invalid

Such an adaptive policy implemented by the caller to dynamically respond to system workload characteristics using P-states allows for efficient power utilization – the processor consumes additional power by operating at a higher performance level only when the current workload requires it to do so.

§

Code Examples A

A.1 OS Boot Flow Sample Code

The sample code given below is a example of setting up operating system register state to prepare the processor for running in virtual mode as described in Section 13.1.2, “Operating System Boot Steps” on page 2:625.

```
// This code will perform the following steps:
//1.Initialize PSR with interrupt disabled (bit 13)
//2.Invalidate ALAT via invala instruction
//3.Invalidate register stack
//4.Set region registers rr[r0] - rr[r7] to RID=0, PS=8K, E=0.
//5.Disable the VHPT
//6.Initialize protection key registers
//7.Initialize SP
//8.Initialize BSP
//9.Enable register stack engine.
//10.Setup IVA
//11.Setup virtual->physical address translation
//12.Setup GP.

.file“start.s”

// globals

        .global main
        .type main, @function     // C function we will return to

    .global __GLOB_DATA_PTR       // External pointer to Global Data area
    .global IVT_BASE              // External pointer to IVT_BASE

        .text

// This is the entry point where primary boot loader
// passes control.

pstart::

    mov     psr.l = r0            // Initialize psr.l
    ;;
    invala                        // Invalidate ALAT
    mov     ar.rsc = r0           // Invalidate register stack
    ;;
    loadrs

// Initialize Region Registers

    mov     r2 = (13 << 2)        // 8K page size
    mov     r3 = r0
    mov     r4 = 61
    ;;

Loader_RRLoop:
    shl     r10 = r3, r4
    ;;
    mov     rr[r10] = r2
    add     r3 = 1, r3
    ;;
    cmp4.geu p6, p7 = 8, r3
```

```
(p6)br.cond.sptk.few.clr Loader_RRLoop
    ;;

// Disable the VHPT walker and set up the minimum size for it (32K) by writing
// to the page table address register (cr.pta)


    mov r2 = (15<<2)
        ;;
    mov cr.pta = r2


// Initialize the protection key registers for kernel

    mov r2 = (1<< 0)
    mov r3 = r0
    ;;
    mov pkr[r3] = r2                  // validate pkr[zero]
    ;;
    mov r2 = r0
    ;;

pkr_loop:
    add  r3=r3,r0, 1                  // start with index 1
    ;;
    cmp.gtu p6,p7 = 8,r3
    ;;
(p6)mov pkr[r3] = r2
(p6)br.cond.sptk.few.clr pkr_loop     // loop until 8

// Setup kernel stack pointer (r12)

    movl    sp = kstack + (64*1024)   // 64K stack
    ;;

// Set up the scratch area on stack

    add     sp = - 32, sp

// Setup the Register stack backing store
//
// 1st deal with Register Stack Configuration register
//
// NOTE: the RSC mode must be enforced lazy (00) to write to bspstore
//
// mode: = enforced lazy
// be = little endian

    mov  ar.rsc = r0
    ;;

//Now have to setup the RSE backing store pointer
//
//NOTE: initializing the bspstore has effects on all 3 RSE pointers
//  (BSP, BSPSTORE, and RSE.BspLoad)

    movl r2 = kstack + ((96 + (96/63))*8)
    ;;
    mov  ar.bspstore = r2

// Need to setup base address for interrupt vector table...

    movl r3 = IVT_BASE
    ;;
    mov  cr.iva = r3

// Setup system address translation for the kernel
```

```
//
//      The Translation Insertion Format looks like the following...
//
//      Below is the register interface to insert entries into the TLB
//
//1) A general register contains an address,attributes,and permissions
//2) ITIR: additional info such as protection key page size info
//3) IFA: specifies the virtual page number for instruction and data
//  TLB inserts
//
//Registers used:
//---------------
//      | 63 53 | 52 | 51 50 | 49 12 | 11 9| 8 7 | 6 | 5 |4 1| 0 |
//GR    |    ig | ed |    rv  |    ppn | ar  | pl | d | a | ma | p |
//
// ITIR | rv {63:32} | key {31:8} | ps {7:2} | rv {1:0}|
//
//IFA   | vpn {63:12}| ignored {11:0} |
//
//RR[vrn] | reserved{63:32} | rid {31:8}| ignored {7:2) | rv{1} | ignored {0}|
//
//
//where
//ig = ignored bits
//rv= reserved bits
//p = present bit
//ma = memory attribute
//a = accessed bit
//d = dirty bit
//pl= privilege level
//ar= access rights
//ppn= physical page number
//ed= exception deferral
//ps= page size of mapping (2**ps)
//vpn= virtual page number
//
// Setup virtual page number
//
// NOTE:The virtual page number depends on a translation’s
//page size.
//
// Add entry for TEXT section

    movl r2 = 0x0
    ;;
    mov  cr.ifa = r2

//setup ITIR (Interruption TLB Insertion Register)

    movl r3=( ( 24 << 2 ) | ( 0 << 8 ) ) // set page size to 16 MB
    ;;
    mov  cr.itir = r3

//now setup the general register to use with itr (insert translation
//register), use physical page of zero

    movl r10 =((1 << 52 )| ( 0x00000000 << 12 )|( 3 << 9 )|( 0 << 7 )| \
                (1 <<6 ) | ( 1 << 5 ) | ( 1 << 0 ))
    mov r11 = r0
    ;;
    itr.i itr[r11] = r10      // Insert translation register


//Entry for OS Data section

    add r11 = 1, r11              // skip to tr next index
    movl r2 = 0x0                 // use vpn 0
    ;;
    mov  cr.ifa = r2

//Setup ITIR (Interruption TLB Insertion Register)

    movl r3 = ( ( 24 << 2 ) | ( 0 << 8 ) )   // 16 MB
    ;;
    mov  cr.itir = r3

//Now setup the general register to use with itr (insert translation
//register)

    movl r10 =( (1 << 52 ) | (0x0 << 12 ) | (3 << 9 ) | (0 << 7) |\
               (1 << 6) | ( 1 << 5 ) | (1 << 0))
    ;;
    itr.d dtr[r11] = r10              // Insert translation register
    ;;

//It is now time to set the appropriate bits in the PSR (processor
//status register)

    movl r3 = ( (1 << 44) | (1 << 36) |(1 << 38) |(1 << 27) |(1 << 17) | \
               (1 << 15) | (1 << 14) | (1 <<      13))
    ;;
    mov cr.ipsr = r3

//Initialize DCR to defer all speculation faults

    movl r2 = 0x7f00
    ;;
    mov cr.dcr = r2

// Initialize the global pointer (gp = r1)

    movl gp = __GLOB_DATA_PTR

// Clear out ifs


    mov cr.ifs=r0

// Need to do a “rfi” in order to synchronize above instructions and set
// “it” and “ed” bits in the PSR.

    movl r3 = main                // Setup for main, C code
    ;;
    mov cr.iip = r3               // Setup iip to hit main
    ;;
    rfi
    ;;

// Setup kernel stack

.data
.globalkstack
.align 16
kstack:
.skip(64*1024)
```

§

Intel® Itanium® Architecture
Software Developer's Manual

Revision 2.3

Volume 3: Intel® Itanium® Instruction Set

Intel® Itanium® Architecture Software Developer's Manual

Volume 3: Intel® Itanium® Instruction Set Reference

Revision 2.3

May 2010

Document Number: 323207

Contents

Figures

Tables

§

About this Manual 1

The Intel® Itanium® architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The Itanium architecture features a revolutionary 64-bit instruction set architecture (ISA) which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the Itanium architecture is IA-32 instruction set compatibility.

The *Intel® Itanium® Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of the features of the Itanium architecture to help them optimize code.

1.1 Overview of Volume 1: Application Architecture

This volume defines the Itanium application architecture, including application level resources, programming environment, and the IA-32 application interface. This volume also describes optimization techniques used to generate high performance software.

1.1.1 Part 1: Application Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

Chapter 2, "Introduction to the Intel® Itanium® Architecture" provides an overview of the architecture.

Chapter 3, "Execution Environment" describes the Itanium register set used by applications and the memory organization models.

Chapter 4, "Application Programming Model" gives an overview of the behavior of Itanium application instructions (grouped into related functions).

Chapter 5, "Floating-point Programming Model" describes the Itanium floating-point architecture (including integer multiply).

Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium® System Environment" describes the operation of IA-32 instructions within the Itanium System Environment from the perspective of an application programmer.

1.1.2 Part 2: Optimization Guide for the Intel® Itanium® Architecture

Chapter 1, "About the Optimization Guide" gives an overview of the optimization guide.

Chapter 2, “Introduction to Programming for the Intel® Itanium® Architecture” provides an overview of the application programming environment for the Itanium architecture.

Chapter 3, “Memory Reference” discusses features and optimizations related to control and data speculation.

Chapter 4, “Predication, Control Flow, and Instruction Stream” describes optimization features related to predication, control flow, and branch hints.

Chapter 5, “Software Pipelining and Loop Support” provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 6, “Floating-point Applications” discusses current performance limitations in floating-point applications and features that address these limitations.

1.2 Overview of Volume 2: System Architecture

This volume defines the Itanium system architecture, including system level resources and programming state, interrupt model, and processor firmware interface. This volume also provides a useful system programmer's guide for writing high performance system software.

1.2.1 Part 1: System Architecture Guide

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Intel® Itanium® System Environment” introduces the environment designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications.

Chapter 3, “System State and Programming Model” describes the Itanium architectural state which is visible only to an operating system.

Chapter 4, “Addressing and Protection” defines the resources available to the operating system for virtual to physical address translation, virtual aliasing, physical addressing, and memory ordering.

Chapter 5, “Interruptions” describes all interruptions that can be generated by a processor based on the Itanium architecture.

Chapter 6, “Register Stack Engine” describes the architectural mechanism which automatically saves and restores the stacked subset (GR32 – GR 127) of the general register file.

Chapter 7, “Debugging and Performance Monitoring” is an overview of the performance monitoring and debugging resources that are available in the Itanium architecture.

Chapter 8, “Interruption Vector Descriptions” lists all interruption vectors.

Chapter 9, “IA-32 Interruption Vector Descriptions” lists IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment.

Chapter 10, “Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications” defines the operation of IA-32 instructions within the Itanium System Environment from the perspective of an Itanium architecture-based operating system.

Chapter 11, “Processor Abstraction Layer” describes the firmware layer which abstracts processor implementation-dependent features.

1.2.2 Part 2: System Programmer’s Guide

Chapter 1, “About the System Programmer’s Guide” gives an introduction to the second section of the system architecture guide.

Chapter 2, “MP Coherence and Synchronization” describes multiprocessing synchronization primitives and the Itanium memory ordering model.

Chapter 3, “Interruptions and Serialization” describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 4, “Context Management” describes how operating systems need to preserve Itanium register contents and state. This chapter also describes system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 5, “Memory Management” introduces various memory management strategies.

Chapter 6, “Runtime Support for Control and Data Speculation” describes the operating system support that is required for control and data speculation.

Chapter 7, “Instruction Emulation and Other Fault Handlers” describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

Chapter 8, “Floating-point System Software” discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

Chapter 9, “IA-32 Application Support” describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

Chapter 10, “External Interrupt Architecture” describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 11, “I/O Architecture” describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

Chapter 12, "Performance Monitoring Support" describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

Chapter 13, "Firmware Overview" introduces the firmware model, and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

Appendix A, "Code Examples" provides OS boot flow sample code.

1.3 Overview of Volume 3: Intel® Itanium® Instruction Set Reference

This volume is a comprehensive reference to the Itanium instruction set, including instruction format/encoding.

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

Chapter 2, "Instruction Reference" provides a detailed description of all Itanium instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, "Pseudo-Code Functions" provides a table of pseudo-code functions which are used to define the behavior of the Itanium instructions.

Chapter 4, "Instruction Formats" describes the encoding and instruction format instructions.

Chapter 5, "Resource and Dependency Semantics" summarizes the dependency rules that are applicable when generating code for processors based on the Itanium architecture.

1.4 Overview of Volume 4: IA-32 Instruction Set Reference

This volume is a comprehensive reference to the IA-32 instruction set, including instruction format/encoding.

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

Chapter 2, "Base IA-32 Instruction Reference" provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-32 Intel® MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 4, “IA-32 SSE Instruction Reference” provides a detailed description of all IA-32 SSE instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.5 Terminology

The following definitions are for terms related to the Itanium architecture and will be used throughout this document:

Instruction Set Architecture (ISA) – Defines application and system level resources. These resources include instructions and registers.

Itanium Architecture – The new ISA with 64-bit instruction capabilities, new performance- enhancing features, and support for the IA-32 instruction set.

IA-32 Architecture – The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer’s Manual***.

Itanium System Environment – The operating system environment that supports the execution of both IA-32 and Itanium architecture-based code.

Itanium® Architecture-based Firmware – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

Processor Abstraction Layer (PAL) – The firmware layer which abstracts processor features that are implementation dependent.

System Abstraction Layer (SAL) – The firmware layer which abstracts system features that are implementation dependent.

1.6 Related Documents

The following documents can be downloaded at the Intel’s Developer Site at http://developer.intel.com:

- ***Dual-Core Update to the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization***– Document number 308065 provides model-specific information about the dual-core Itanium processors.
- ***Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*** – This document (Document number 251110) describes model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.
- ***Intel® Itanium® Processor Reference Manual for Software Development*** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.

- ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.
- ***Intel® Itanium® Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.

The following document can be downloaded at the Unified EFI Forum website at http://www.uefi.org:

- ***Unified Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware.

1.7 Revision History

Date of Revision	Revision Number	Description
March 2010	2.3	Added information about illegal virtualization optimization combinations and IIPA requirements. Added Resource Utilization Counter and PAL_VP_INFO. PAL_VP_INIT and VPD.vpr changes. New PAL_VPS_RESUME_HANDLER parameter to indicate RSE Current Frame Load Enable setting at the target instruction. PAL_VP_INIT_ENV implementation-specific configuration option. Minimum Virtual address increased to 54 bits. New PAL_MC_ERROR_INFO health indicator. New PAL_MC_ERROR_INJECT implementation-specific bit fields. MOV-to_SR.L reserved field checking. Added virtual machine disable. Added variable frequency mode additions to ACPI P-state description. Removed *pal_proc_vector* argument from PAL_VP_SAVE and PAL_VP_RESTORE. Added PAL_PROC_SET_FEATURES data speculation disable. Added Interruption Instruction Bundle registers. Min-state save area size change. PAL_MC_DYNAMIC_STATE changes. PAL_PROC_SET_FEATURES data poisoning promotion changes. ACPI P-state clarifications. Synchronization requirements for virtualization opcode optimization. New priority hint and multi-threading hint recommendations.

Date of Revision	Revision Number	Description
August 2005	2.2	Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 “Interruptions” and Section 11.2.2 PALE_RESET Exit State for details. Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10. Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of `ptc.l` instruction in Vol 3 for details. Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details. Introduced Unimplemented Instruction Address fault. Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details. Architecture extensions for processor virtualization. All instructions which must be last in an instruction group results in undefined behavior when this rule is violated. Added architectural sequence that guarantees increasing ITC and PMD values on successive reads. Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures. Allows IPI-redirection feature to be optional. Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block. Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details. “Bus parking” feature is now optional for PAL_BUS_GET_FEATURES. Introduced low-power synchronization primitive using `hint` instruction. FR32-127 is now preserved in PAL calling convention. New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges. Performance Monitor Data (PMD) registers are no longer sign-extended. New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details. Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure. Clarified PMU interrupts as edge-triggered. Modified ‘proc_number’ parameter in PAL_LOGICAL_TO_PHYSICAL procedure. Modified pal_copy_info alignment requirements. New bit in PAL_PROC_GET_FEATURES for variable P-state performance. Clarified descriptions for check_target_register and check_target_register_sof. Various fixes in dependency tables in Vol 3 Ch 5. Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5. Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3. Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.

Date of Revision	Revision Number	Description
August 2002	2.1	Added Predicate Behavior of `alloc` Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3). Added New `fc.i` Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3). Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2). Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2). Added Spontaneous NaT Generation on Speculative Load (`ld.s`) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3). Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2). Added Freeze Bit Functionality in Context Switching and Interrupt Generation Clarification (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Volume 2) Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2). Added capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Volume 2). Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2). Added Fault Handling Semantics for `lfetch.fault` Instruction (Section 2.2, Part I, Volume 3).
December 2001	2.0	Volume 1: Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1). IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3). Load instructions change (Section 4.4.1).

Date of Revision	Revision Number	Description
		Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Chapter 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4). Memory attribute transitions – added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation – changed behavior of DCR ‘lc’ bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2).
		Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3).

Date of Revision	Revision Number	Description
		Volume 2: Clarifications regarding “reserved” fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10). FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4). Clarification regarding ordering data dependency. Out-of-order IPI delivery is now allowed (Chapters 4 and 5). Content of EFLAG field changed in IIM (p. 9-24). PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11). PAL_CHECK processor state parameter changes (Chapter 11). PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11). PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11). PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11). PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11). PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11). Clarified memory ordering changes (Chapter 13). Clarification in dependence violation table (Appendix A).
		Volume 3: fmix instruction page figures corrected (Chapter 2). Clarification of “reserved” fields in ITIR (Chapters 2 and 3). Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/ instruction group (Chapters 2 and 4). IA-32 JMPE instruction page typo fix (p. 5-238). Processor Serial Number feature removed (Chapter 5).
January 2000	1.0	Initial release of document.

§

Instruction Reference 2

This chapter describes the function of each Itanium instruction. The pages of this chapter are sorted alphabetically by assembly language mnemonic.

2.1 Instruction Page Conventions

The instruction pages are divided into multiple sections as listed in Table 2-1. The first three sections are present on all instruction pages. The last three sections are present only when necessary. Table 2-2 lists the font conventions which are used by the instruction pages.

Table 2-1. Instruction Page Description

Section Name	Contents
Format	Assembly language syntax, instruction type and encoding format
Description	Instruction function in English
Operation	Instruction function in C code
FP Exceptions	IEEE floating-point traps
Interruptions	Prioritized list of interruptions that may be caused by the instruction
Serialization	Serializing behavior or serialization requirements

Table 2-2. Instruction Page Font Conventions

Font	Interpretation
regular	(Format section) Required characters in an assembly language mnemonic
italic	(Format section) Assembly language field name that must be filled with one of a range of legal values listed in the Description section
`code`	(Operation section) C code specifying instruction behavior
`code_italic`	(Operation section) Assembly language field name corresponding to a *italic* field listed in the Format section

In the Format section, register addresses are specified using the assembly mnemonic field names given in the third column of Table 2-3. For instructions that are predicated, the Description section assumes that the qualifying predicate is true (except for instructions that modify architectural state when their qualifying predicate is false). The test of the qualifying predicate is included in the Operation section (when applicable).

In the Operation section, registers are addressed using the notation `reg[`*`addr`*`].field`. The register file being accessed is specified by `reg`, and has a value chosen from the second column of Table 2-3. The *`addr`* field specifies a register address as an assembly language field name or a register mnemonic. For the general, floating-point, and predicate register files which undergo register renaming, *`addr`* is the register address prior to renaming and the renaming is not shown. The `field` option specifies a named bit field within the register. If `field` is absent, then all fields of the register are accessed. The only exception is when referencing the data field of the general registers

(64-bits not including the NaT bit) where the notation GR[addr] is used. The syntactical differences between the code found in the Operation section and ANSI C is listed in Table 2-4.

Table 2-3. Register File Notation

Register File	C Notation	Assembly Mnemonic	Indirect Access
Application registers	AR	ar	
Branch registers	BR	b	
Control registers	CR	cr	
CPU identification registers	CPUID	cpuid	Y
Data breakpoint registers	DBR	dbr	Y
Instruction breakpoint registers	IBR	ibr	Y
Data TLB translation cache	DTC	N/A	
Data TLB translation registers	DTR	dtr	Y
Floating-point registers	FR	f	
General registers	GR	r	
Instruction TLB translation cache	ITC	N/A	
Instruction TLB translation registers	ITR	itr	Y
Protection key registers	PKR	pkr	Y
Performance monitor configuration registers	PMC	pmc	Y
Performance monitor data registers	PMD	pmd	Y
Predicate registers	PR	p	
Region registers	RR	rr	Y

Table 2-4. C Syntax Differences

Syntax	Function
{msb:lsb}, {bit}	Bit field specifier. When appended to a variable, denotes a bit field extending from the most significant bit specified by “msb” to the least significant bit specified by “lsb” including bits “msb” and “lsb.” If “msb” and “lsb” are equal then a single bit is accessed. The second form denotes a single bit.
u>, u>=, u<, u<=	Unsigned inequality relations. Variables on either side of the operator are treated as unsigned.
u>>, u>>=	Unsigned right shift. Zeroes are shifted into the most significant bit position.
u+	Unsigned addition. Operands are treated as unsigned, and zero-extended.
u*	Unsigned multiplication. Operands are treated as unsigned.

The Operation section contains code that specifies only the execution semantics of each instruction and does not include any behavior relating to instruction fetch (e.g., interrupts and faults caused during fetch). The Interruptions section does not list any faults that may be caused by instruction fetch or by mandatory RSE loads. The code to raise certain pervasive faults and actions is not included in the code in the Operation section. These faults and actions are listed in Table 2-5. The Single step trap applies to all instructions and is not listed in the Interruptions section.

Table 2-5. Pervasive Conditions Not Included in Instruction Description Code

Condition	Action
Read of a register outside the current frame.	An undefined value is returned (no fault).
Access to a banked general register (GR 16 through GR 31).	The GR bank specified by PSR.bn is accessed.
PSR.ss is set.	A Single Step trap is raised.

2.2 Instruction Descriptions

The remainder of this chapter provides a description of each of the Itanium instructions.

add — Add

Format:

(*qp*) add $r_1 = r_2, r_3$	register_form	A1
(*qp*) add $r_1 = r_2, r_3$, 1	plus1_form, register_form	A1
(*qp*) add $r_1 = imm, r_3$	pseudo-op	
(*qp*) adds $r_1 = imm_{14}, r_3$	imm14_form	A4
(*qp*) addl $r_1 = imm_{22}, r_3$	imm22_form	A5

Description: The two source operands (and an optional constant 1) are added and the result placed in GR r_1. In the register form the first operand is GR r_2; in the imm_14 form the first operand is taken from the sign-extended imm_{14} encoding field; in the imm22_form the first operand is taken from the sign-extended imm_{22} encoding field. In the imm22_form, GR r_3 can specify only GRs 0, 1, 2 and 3.

The plus1_form is available only in the register_form (although the equivalent effect in the immediate forms can be achieved by adjusting the immediate).

The immediate-form pseudo-op chooses the imm14_form or imm22_form based on the size of the immediate operand and the value of r_3.

Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);

    if (register_form)                                  // register form
        tmp_src = GR[$r_2$];
    else if (imm14_form)                                // 14-bit immediate form
        tmp_src = sign_ext($imm_{14}$, 14);
    else                                                // 22-bit immediate form
        tmp_src = sign_ext($imm_{22}$, 22);

    tmp_nat = (register_form ? GR[$r_2$].nat : 0);

    if (plus1_form)
        GR[$r_1$] = tmp_src + GR[$r_3$] + 1;
    else
        GR[$r_1$] = tmp_src + GR[$r_3$];

    GR[$r_1$].nat = tmp_nat || GR[$r_3$].nat;
}
```

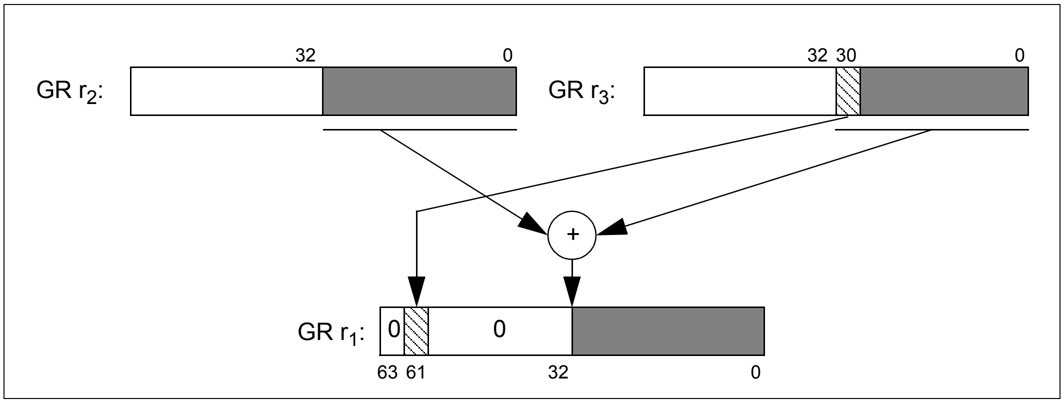
Interruptions: Illegal Operation fault

addp4 — Add Pointer

Format: (qp) addp4 $r_1 = r_2, r_3$ register_form A1
(qp) addp4 $r_1 = imm_{14}, r_3$ imm14_form A4

Description: The two source operands are added. The upper 32 bits of the result are forced to zero, and then bits {31:30} of GR r_3 are copied to bits {62:61} of the result. This result is placed in GR r_1. In the register_form the first operand is GR r_2; in the imm14_form the first operand is taken from the sign-extended imm_{14} encoding field.

Figure 2-1. Add Pointer



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm14, 14));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    tmp_res = tmp_src + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

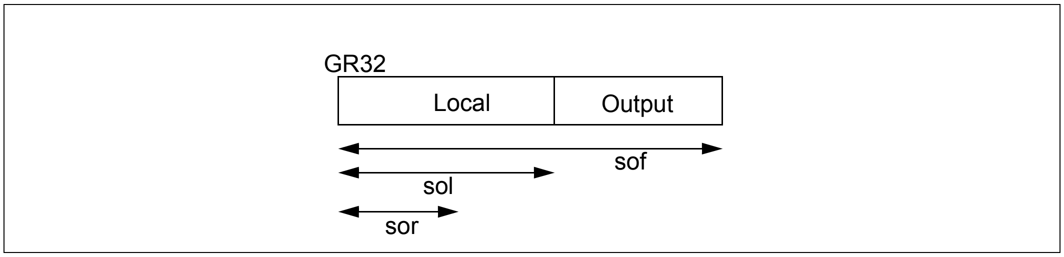
alloc — Allocate Stack Frame

Format: (*qp*) alloc r_1 = ar.pfs, *i*, *l*, *o*, *r* M34

Description: A new stack frame is allocated on the general register stack, and the Previous Function State register (PFS) is copied to GR r_1. The change of frame size is immediate. The write of GR r_1 and subsequent instructions in the same instruction group use the new frame.

The four parameters, *i* (size of inputs), *l* (size of locals), *o* (size of outputs), and *r* (size of rotating) specify the sizes of the regions of the stack frame.

Figure 2-2. Stack Frame



The size of the frame (sof) is determined by *i* + *l* + *o*. Note that this instruction may grow or shrink the size of the current register stack frame. The size of the local region (sol) is given by *i* + *l*. There is no real distinction between inputs and locals. They are given as separate operands in the instruction only as a hint to the assembler about how the local registers are to be used.

The rotating registers must fit within the stack frame and be a multiple of 8 in number. If this instruction attempts to change the size of CFM.sor, and the register rename base registers (CFM.rrb.gr, CFM.rrb.fr, CFM.rrb.pr) are not all zero, then the instruction will cause a Reserved Register/Field fault.

Although the assembler does not allow illegal combinations of operands for alloc, illegal combinations can be encoded in the instruction. Attempting to allocate a stack frame larger than 96 registers, or with the rotating region larger than the stack frame, or with the size of locals larger than the stack frame, or specifying a qualifying predicate other than PR 0, will cause an Illegal Operation fault.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined.

If insufficient registers are available to allocate the desired frame `alloc` will stall the processor until enough dirty registers are written to the backing store. Such mandatory RSE stores may cause the data related faults listed below.

Operation:

```
// tmp_sof, tmp_sol, tmp_sor are the fields encoded in the instruction
tmp_sof = i + l + o;
tmp_sol = i + l;
tmp_sor = r u>> 3;
check_target_register_sof(r1, tmp_sof);
if (tmp_sof u> 96 || r u> tmp_sof || tmp_sol u> tmp_sof || qp != 0)
   illegal_operation_fault();
if (tmp_sor != CFM.sor &&
            (CFM.rrb.gr != 0 || CFM.rrb.fr != 0 || CFM.rrb.pr != 0))
   reserved_register_field_fault();

alat_frame_update(0, tmp_sof - CFM.sof);
rse_new_frame(CFM.sof, tmp_sof);// Make room for new registers; Mandatory
                                // RSE stores can raise faults listed below.
CFM.sof = tmp_sof;
CFM.sol = tmp_sol;
CFM.sor = tmp_sor;

GR[r1] = AR[PFS];
GR[r1].nat = 0;
```

Interruptions:

Illegal Operation fault
Reserved Register/Field fault
Unimplemented Data Address fault
VHPT Data fault
Data Nested TLB fault
Data TLB fault
Alternate Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

and — Logical And

Format: (*qp*) and $r_1 = r_2, r_3$ register_form A1
(*qp*) and $r_1 = imm_8, r_3$ imm8_form A3

Description: The two source operands are logically ANDed and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

andcm — And Complement

Format:	(qp) andcm $r_1 = r_2, r_3$	register_form	A1
	(qp) andcm $r_1 = imm_8, r_3$	imm8_form	A3

Description: The first source operand is logically ANDed with the 1’s complement of the second source operand and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src & ~GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

br — Branch

Format:			
	(qp) br.*btype.bwh.ph.dh* $target_{25}$	ip_relative_form	B1
	(qp) br.*btype.bwh.ph.dh* b_1 = $target_{25}$	call_form, ip_relative_form	B3
	br.*btype.bwh.ph.dh* $target_{25}$	counted_form, ip_relative_form	B2
	br.*ph.dh* $target_{25}$	pseudo-op	
	(qp) br.*btype.bwh.ph.dh* b_2	indirect_form	B4
	(qp) br.*btype.bwh.ph.dh* b_1 = b_2	call_form, indirect_form	B5
	br.*ph.dh* b_2	pseudo-op	

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

Branches can be either IP-relative, or indirect. For IP-relative branches, the $target_{25}$ operand, in assembly, specifies a label to branch to. This is encoded in the branch instruction as a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction (imm_{21} = $target_{25}$ - IP >> 4). For indirect branches, the target address is taken from BR b_2.

Table 2-6. Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-rel or Indirect
call	Conditional procedure call	Qualifying predicate	IP-rel or Indirect
ret	Conditional procedure return	Qualifying predicate	Indirect
ia	Invoke IA-32 instruction set	Unconditional	Indirect
cloop	Counted loop branch	Loop count	IP-rel
ctop, cexit	Mod-scheduled counted loop	Loop count and epilog count	IP-rel
wtop, wexit	Mod-scheduled while loop	Qualifying predicate and epilog count	IP-rel

There are two pseudo-ops for unconditional branches. These are encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For the basic branch types, the branch condition is simply the value of the specified predicate register. These basic branch types are:

- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
    - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
    - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
    - The rotation rename base registers in the CFM are reset to 0.
    - A return link value is placed in BR b_1.
- **return:** If the qualifying predicate is 1, the branch is taken and the following occurs:
    - CFM, EC, and the current privilege level are restored from PFS. (The privilege level is restored only if this does not increase privilege.)
    - The caller's stack frame is restored.
    - If the return lowers the privilege, and PSR.lp is 1, then a Lower-Privilege Transfer trap is taken.
- **ia:** The branch is taken unconditionally, if it is not intercepted by the OS. The effect of the branch is to invoke the IA-32 instruction set (by setting PSR.is to 1) and begin processing IA-32 instructions at the virtual linear target address contained in BR b_2\{31:0\}. If the qualifying predicate is not PR 0, an Illegal Operation fault is raised. If instruction set transitions are disabled (PSR.di is 1), then a Disabled Instruction Set Transition fault is raised.

  The IA-32 target effective address is calculated relative to the current code segment, i.e. EIP\{31:0\} = BR b_2\{31:0\} - CSD.base. The IA-32 instruction set can be entered at any privilege level, provided PSR.di is 0. If PSR.dfh is 1, a Disabled FP Register fault is raised on the target IA-32 instruction. No register bank switch nor change in privilege level occurs during the instruction set transition.

  Software must ensure the code segment descriptor (CSD) and selector (CS) are loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA_32_Exception(GPFault) is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if BR b_2 is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction. EFLAG.rf is unmodified until the successful completion of the first IA-32 instruction. PSR.da, PSR.id, PSR.ia, PSR.dd, and PSR.ed are cleared to zero after `br.ia` completes execution and before the first IA-32 instruction begins execution. EFLAG.rf is not cleared until the target IA-32 instruction successfully completes.

  Software must set PSR properly before branching to the IA-32 instruction set; otherwise processor operation is undefined. See Table 3-2, "Processor Status Register Fields" on page 2:24 for details.

  Software must issue a `mf` instruction before the branch if memory ordering is required between IA-32 processor consistent and Itanium unordered memory references. The processor does not ensure Itanium-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instruction fetches. `br.ia` does not perform an instruction serialization operation. The processor does ensure that prior writes (even in the same instruction group) to GRs and FRs are observed by the first IA-32 instruction. Writes to ARs within the same instruction group as `br.ia` are not allowed, since `br.ia` may implicitly reads all ARs. If an illegal RAW dependency is present between an AR write and `br.ia`, the first IA-32 instruction fetch and execution may or may not see the updated AR value.

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT values being preserved across an instruction set transition. All registers left in the current register stack frame are undefined across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. If the register stack contains any dirty registers, an Illegal Operation fault is raised on the `br.ia` instruction. The current register stack frame is forced to zero. To flush the register file of dirty registers, the `flushrs` instruction must be issued in an instruction group preceding the `br.ia` instruction. To enhance the performance of the instruction set transition, software can start the register stack flush in parallel with starting the IA-32 instruction set by 1) ensuring `flushrs` is exactly one instruction group before the `br.ia`, and 2) `br.ia` is in the first B-slot. `br.ia` should always be executed in the first B-slot with a hint of "static-taken" (default), otherwise processor performance will be degraded.

If a `br.ia` causes any Itanium traps (e.g., Single Step trap, Taken Branch trap, or Unimplemented Instruction Address trap), IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

Another branch type is provided for simple counted loops. This branch type uses the Loop Count application register (LC) to determine the branch condition, and does not use a qualifying predicate:

- **cloop:** If the LC register is not equal to zero, it is decremented and the branch is taken.

In addition to these simple branch types, there are four types which are used for accelerating modulo-scheduled loops (see also Section 4.5.1, "Modulo-scheduled Loop Support" on page 1:75). Two of these are for counted loops (which use the LC register), and two for while loops (which use the qualifying predicate). These loop types use register rotation to provide register renaming, and they use predication to turn off instructions that correspond to empty pipeline stages.

The Epilog Count application register (EC) is used to count epilog stages and, for some while loops, a portion of the prolog stages. In the epilog phase, EC is decremented each time around and, for most loops, when EC is one, the pipeline has been drained, and the loop is exited. For certain types of optimized, unrolled software-pipelined loops, the target of a `br.cexit` or `br.wexit` is set to the next sequential bundle. In this case, the pipeline may not be fully drained when EC is one, and continues to drain while EC is zero.

For these modulo-scheduled loop types, the calculation of whether the branch is taken or not depends on the kernel branch condition (LC for counted types, and the qualifying predicate for while types) and on the epilog condition (whether EC is greater than one or not).

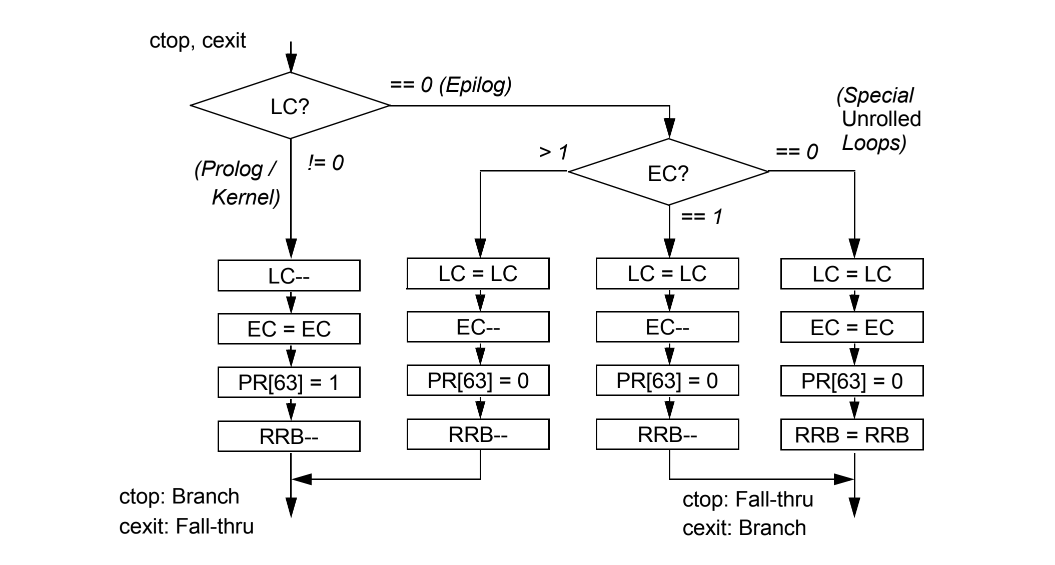
These branch types are of two categories: top and exit. The top types (ctop and wtop) are used when the loop decision is located at the bottom of the loop body and therefore a taken branch will continue the loop while a fall through branch will exit the loop. The exit types (cexit and wexit) are used when the loop decision is located somewhere other than the bottom of the loop and therefore a fall though branch will continue the loop and a taken branch will exit the loop. The exit types are also used at intermediate points in an unrolled pipelined loop. (For more details, see Section 4.5.1, "Modulo-scheduled Loop Support" on page 1:75).

The modulo-scheduled loop types are:

- **ctop** and **cexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.ctop`, the branch is taken if either LC is non-zero or EC is greater than one. For `br.cexit`, the opposite is true. It is not taken if either LC is non-zero or EC is greater than one and is taken otherwise.

  These branch types also use LC and EC to control register rotation and predicate initialization. During the prolog and kernel phase, when LC is non-zero, LC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero, the epilog phase is entered, and EC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If LC and EC are equal to zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-3.

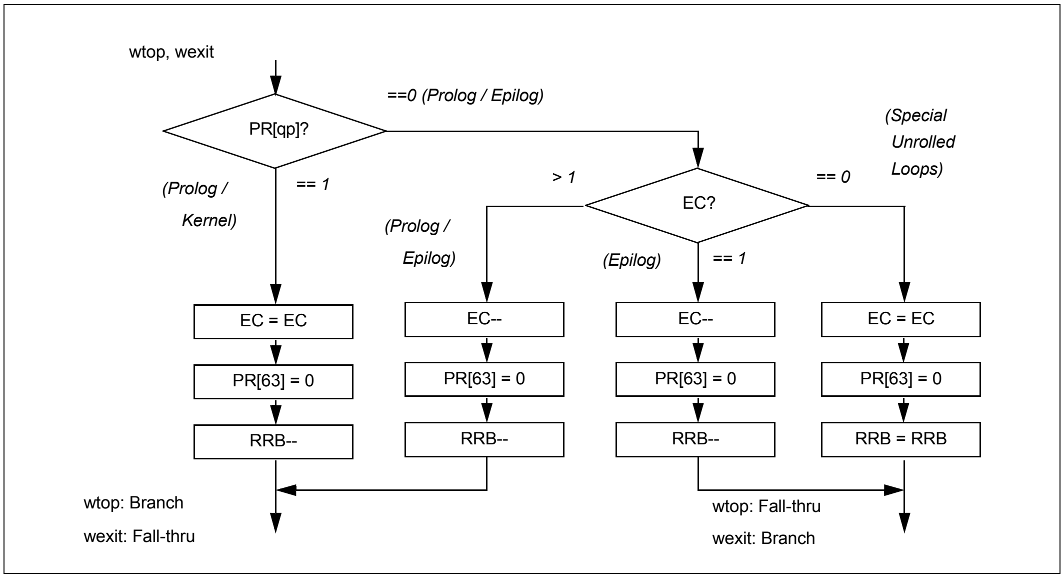
Figure 2-3. Operation of br.ctop and br.cexit



wtop and **wexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.wtop`, the branch is taken if either the qualifying predicate is one or EC is greater than one. For `br.wexit`, the opposite is true. It is not taken if either the qualifying predicate is one or EC is greater than one, and is taken otherwise.

These branch types also use the qualifying predicate and EC to control register rotation and predicate initialization. During the prolog phase, the qualifying predicate is either zero or one, depending upon the scheme used to program the loop. During the kernel phase, the qualifying predicate is one. During the epilog phase, the qualifying predicate is zero, and EC counts down. When `br.wtop` or `br.wexit` is executed with the qualifying predicate equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If the qualifying predicate and EC are zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-4.

Figure 2-4. Operation of br.wtop and br.wexit



The loop-type branches (`br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, and `br.wexit`) are only allowed in instruction slot 2 within a bundle. Executing such an instruction in either slot 0 or 1 will cause an Illegal Operation fault, whether the branch would have been taken or not.

Read after Write (RAW) and Write after Read (WAR) dependency requirements are slightly different for branch instructions. Changes to BRs, PRs, and PFS by non-branch instructions are visible to a subsequent branch instruction in the same instruction group (i.e., a limited RAW is allowed for these resources). This allows for a low-latency compare-branch sequence, for example. The normal RAW requirements apply to the LC and EC application registers, and the RRBs.

Within an instruction group, a WAR dependency on PR 63 is not allowed if both the reading and writing instructions are branches. For example, a `br.wtop` or `br.wexit` may not use PR[63] as its qualifying predicate and PR[63] cannot be the qualifying predicate for any branch preceding a `br.wtop` or `br.wexit` in the same instruction group.

For dependency purposes, the loop-type branches effectively always write their associated resources, whether they are taken or not. The cloop type effectively always writes LC. When LC is 0, a cloop branch leaves it unchanged, but hardware may implement this as a re-write of LC with the same value. Similarly, `br.ctop` and `br.cexit` effectively always write LC, EC, the RRBs, and PR[63]. `br.wtop` and `br.wexit` effectively always write EC, the RRBs, and PR[63].

Values for various branch hint completers are shown in the following tables. Whether Prediction Strategy hints are shown in Table 2-7. Sequential Prefetch hints are shown in Table 2-8. Branch Cache Deallocation hints are shown in Table 2-9. See Section 4.5.2, “Branch Prediction Hints” on page 1:78.

Table 2-7. Branch Whether Hint

bwh Completer	Branch Whether Hint
spnt	Static Not-Taken
sptk	Static Taken
dpnt	Dynamic Not-Taken
dptk	Dynamic Taken

Table 2-8. Sequential Prefetch Hint

ph Completer	Sequential Prefetch Hint
few or *none*	Few lines
many	Many lines

Table 2-9. Branch Cache Deallocation Hint

dh Completer	Branch Cache Deallocation Hint
none	Don’t deallocate
clr	Deallocate branch information

Operation:

```
if (ip_relative_form)                         // determine branch target
    tmp_IP = IP + sign_ext((imm21 << 4), 25);
else // indirect_form
    tmp_IP = BR[b2];

if (btype != ‘ia’)                        // for Itanium branches,
    tmp_IP = tmp_IP & ~0xf;               //  ignore bottom 4 bits of target

lower_priv_transition = 0;

switch (btype) {
    case ‘cond’:                          // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case ‘call’:                          // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;            // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;           // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        }
        break;

    case ‘ret’:                           // return restores stack frame
    tmp_taken = PR[qp];
    if (tmp_taken) {
        // tmp_growth indicates the amount to move logical TOP *up*:
        // tmp_growth = sizeof(previous out) - sizeof(current frame)
        // a negative amount indicates a shrinking stack
        tmp_growth = (AR[PFS].pfm.sof - AR[PFS].pfm.sol) - CFM.sof;
        alat_frame_update(-AR[PFS].pfm.sol, 0);
        rse_fatal = rse_restore_frame(AR[PFS].pfm.sol,
                                    tmp_growth, CFM.sof);
        if (rse_fatal) {
        // See Section 6.4, “RSE Operation” on page 2:137
            CFM.sof = 0;
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        } else // normal branch return
            CFM = AR[PFS].pfm;

        rse_enable_current_frame_load();
        AR[EC] = AR[PFS].pec;
        if (PSR.cpl u< AR[PFS].ppl) {   // ... and restores privilege
            PSR.cpl = AR[PFS].ppl;
            lower_priv_transition = 1;
        }
    }
    break;

case ‘ia’:                             // switch to IA mode
    tmp_taken = 1;
    if (PSR.ic == 0 || PSR.dt == 0 || PSR.mc == 1 || PSR.it == 0)
        undefined_behavior();
    if (qp != 0)
        illegal_operation_fault();
    if (AR[BSPSTORE] != AR[BSP])
        illegal_operation_fault();
    if (PSR.di)
        disabled_instruction_set_transition_fault();
    PSR.is = 1;                        // set IA-32 Instruction Set Mode
    CFM.sof = 0;                       //force current stack frame
    CFM.sol = 0;                       //to zero
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
//compute effective instruction pointer
    EIP{31:0} = tmp_IP{31:0} - AR[CSD].Base;

// Note the register stack is disabled during IA-32 instruction
// set execution
    break;

case ‘cloop’:                          // simple counted loop
    if (slot != 2)
```

```
            illegal_operation_fault();
        tmp_taken = (AR[LC] != 0);
        if (AR[LC] != 0)
            AR[LC]--;
        break;

    case ‘ctop’:
    case ‘cexit’:                             // SW pipelined counted loop
        if (slot != 2)
            illegal_operation_fault();
        if (btype == ‘ctop’) tmp_taken =  ((AR[LC] != 0) || (AR[EC] u> 1));
        if (btype == ‘cexit’)tmp_taken = !((AR[LC] != 0) || (AR[EC] u> 1));
        if (AR[LC] != 0) {
            AR[LC]--;
            AR[EC] = AR[EC];
            PR[63] = 1;
            rotate_regs();
        } else if (AR[EC] != 0) {
            AR[LC] = AR[LC];
            AR[EC]--;
            PR[63] = 0;
            rotate_regs();
        } else {
            AR[LC] = AR[LC];
            AR[EC] = AR[EC];
            PR[63] = 0;
            CFM.rrb.gr = CFM.rrb.gr;
            CFM.rrb.fr = CFM.rrb.fr;
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;

    case ‘wtop’:
    case ‘wexit’:                             // SW pipelined while loop
        if (slot != 2)
            illegal_operation_fault();
        if (btype == ‘wtop’) tmp_taken =  (PR[qp] || (AR[EC] u> 1));
        if (btype == ‘wexit’)tmp_taken = !(PR[qp] || (AR[EC] u> 1));
        if (PR[qp]) {
            AR[EC] = AR[EC];
            PR[63] = 0;
            rotate_regs();
        } else if (AR[EC] != 0) {
            AR[EC]--;
            PR[63] = 0;
            rotate_regs();
        } else {
            AR[EC] = AR[EC];
            PR[63] = 0;
            CFM.rrb.gr = CFM.rrb.gr;
            CFM.rrb.fr = CFM.rrb.fr;
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;
}
if (tmp_taken) {
        taken_branch = 1;
        IP = tmp_IP;                            // set the new value for IP
        if (!impl_uia_fault_supported() &&
            ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
             || (!PSR.it && unimplemented_physical_address(tmp_IP))))
            unimplemented_instruction_address_trap(lower_priv_transition,
                                                    tmp_IP);
        if (lower_priv_transition && PSR.lp)
            lower_privilege_transfer_trap();
        if (PSR.tb)
            taken_branch_trap();
    }
```

Interruptions: Illegal Operation fault
Disabled Instruction Set Transition fault
Unimplemented Instruction Address trap
Lower-Privilege Transfer trap
Taken Branch trap

Additional Faults on IA-32 target instructions:
IA_32_Exception(GPFault)
Disabled FP Reg Fault if PSR.dfh is 1

break — Break

Format:

(*qp*) break *imm_{21}*	pseudo-op	
(*qp*) break.i *imm_{21}*	i_unit_form	I19
(*qp*) break.b *imm_{21}*	b_unit_form	B9
(*qp*) break.m *imm_{21}*	m_unit_form	M37
(*qp*) break.f *imm_{21}*	f_unit_form	F15
(*qp*) break.x *imm_{62}*	x_unit_form	X1

Description: A Break Instruction fault is taken. For the i_unit_form, f_unit_form and m_unit_form, the value specified by *imm_{21}* is zero-extended and placed in the Interruption Immediate control register (IIM).

For the b_unit_form, *imm_{21}* is ignored and the value zero is placed in the Interruption Immediate control register (IIM).

For the x_unit_form, the lower 21 bits of the value specified by *imm_{62}* is zero-extended and placed in the Interruption Immediate control register (IIM). The L slot of the bundle contains the upper 41 bits of *imm_{62}*.

A `break.i` instruction may be encoded in an MLI-template bundle, in which case the L slot of the bundle is ignored.

This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Operation:

```
if (PR[qp]) {
    if (b_unit_form)
        immediate = 0;
    else if (x_unit_form)
        immediate = zero_ext($imm_{62}$, 21);
    else // i_unit_form || m_unit_form || f_unit_form
        immediate = zero_ext($imm_{21}$, 21);

    break_instruction_fault(immediate);
}
```

Interruptions: Break Instruction fault

brl — Branch Long

Format:

(*qp*) brl.*btype.bwh.ph.dh* *target*$_{64}$		X3
(*qp*) brl.*btype.bwh.ph.dh* b_1 = *target*$_{64}$	call_form	X4
brl.*ph.dh* *target*$_{64}$	pseudo-op	

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

Long branches are always IP-relative. The *target*$_{64}$ operand, in assembly, specifies a label to branch to. This is encoded in the long branch instruction as an immediate displacement (*imm*$_{60}$) between the target bundle and the bundle containing this instruction (*imm*$_{60}$ = *target*$_{64}$ - IP >> 4). The L slot of the bundle contains 39 bits of *imm*$_{60}$.

Table 2-10. Long Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-relative
call	Conditional procedure call	Qualifying predicate	IP-relative

There is a pseudo-op for long unconditional branches, encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For all long branch types, the branch condition is simply the value of the specified predicate register:

- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR b_1.

Read after Write (RAW) and Write after Read (WAR) dependency requirements for long branch instructions are slightly different than for other instructions but are the same as for branch instructions. See page 3:24 for details.

This instruction must be immediately followed by a stop; otherwise its behavior is undefined.

Values for various branch hint completers are the same as for branch instructions. Whether Prediction Strategy hints are shown in Table 2-7 on page 3:25, Sequential Prefetch hints are shown in Table 2-8 on page 3:25, and Branch Cache Deallocation hints are shown in Table 2-9 on page 3:25. See Section 4.5.2, "Branch Prediction Hints" on page 1:78.

This instruction is not implemented on the Itanium processor, which takes an Illegal Operation fault whenever a long branch instruction is encountered, regardless of whether the branch is taken or not. To support the Itanium processor, the operating system is required to provide an Illegal Operation fault handler which emulates taken and not-taken long branches. Presence of this instruction is indicated by a 1 in the lb bit of CPUID register 4. See Section 3.1.11, “Processor Identification Registers” on page 1:34.

Operation:

```
tmp_IP = IP + (imm60 << 4);                // determine branch target
if (!followed_by_stop())
    undefined_behavior();
if (!instruction_implemented(BRL))
    illegal_operation_fault();

switch (btype) {
    case ‘cond’:                           // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case ‘call’:                           // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;             // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;            // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                           // set the new value for IP
    if (!impl_uia_fault_supported() &&
        ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
         || (!PSR.it && unimplemented_physical_address(tmp_IP))))
        unimplemented_instruction_address_trap(0,tmp_IP);
    if (PSR.tb)
        taken_branch_trap();
}
```

Interruptions: Illegal Operation fault; Taken Branch trap; Unimplemented Instruction Address trap

brp — Branch Predict

Format:

brp.*ipwh.ih* $target_{25}$, tag_{13}	ip_relative_form	B6
brp.*indwh.ih* b_2, tag_{13}	indirect_form	B7
brp.ret.*indwh.ih* b_2, tag_{13}	return_form, indirect_form	B7

Description: This instruction can be used to provide to hardware early information about a future branch. It has no effect on architectural machine state, and operates as a `nop` instruction except for its performance effects.

The tag_{13} operand, in assembly, specifies the address of the branch instruction to which this prediction information applies. This is encoded in the branch predict instruction as a signed immediate displacement ($timm_9$) between the bundle containing the presaged branch and the bundle containing this instruction ($timm_9$ = tag_{13} - IP >> 4).

The $target_{25}$ operand, in assembly, specifies the label that the presaged branch will have as its target. This is encoded in the branch predict instruction exactly as in branch instructions, with a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction (imm_{21} = $target_{25}$ - IP >> 4). The indirect_form can be used to presage an indirect branch. In the indirect_form, the target of the presaged branch is given by BR b_2.

The return_form is used to indicate that the presaged branch will be a return.

Other hints can be given about the presaged branch. Values for various hint completers are shown in the following tables. For more details, refer to Section 4.5.2, "Branch Prediction Hints" on page 1:78.

The *ipwh* and *indwh* completers provide information about how best the branch condition should be predicted, when the branch is reached.

Table 2-11. IP-relative Branch Predict Whether Hint

ipwh Completer	IP-relative Branch Predict Whether Hint
sptk	Presaged branch should be predicted Static Taken
loop	Presaged branch will be `br.cloop`, `br.ctop`, or `br.wtop`
exit	Presaged branch will be `br.cexit` or `br.wexit`
dptk	Presaged branch should be predicted Dynamically

Table 2-12. Indirect Branch Predict Whether Hint

indwh Completer	Indirect Branch Predict Whether Hint
sptk	Presaged branch should be predicted Static Taken
dptk	Presaged branch should be predicted Dynamically

The *ih* completer can be used to mark a small number of very important branches (e.g., an inner loop branch). This can signal to hardware to use faster, smaller prediction structures for this information.

Table 2-13. Importance Hint

ih Completer	Branch Predict Importance Hint
none	Less important
imp	More important

Operation:

```
tmp_tag = IP + sign_ext((timm_9 << 4), 13);
if (ip_relative_form) {
    tmp_target = IP + sign_ext((imm_21 << 4), 25);
    tmp_wh = ipwh;
} else { // indirect_form
    tmp_target = BR[b_2];
    tmp_wh = indwh;
}
branch_predict(tmp_wh, ih, return_form, tmp_target, tmp_tag);
```

Interruptions: None

bsw — Bank Switch

Format:	bsw.0	zero_form	B8
	bsw.1	one_form	B8

Description: This instruction switches to the specified register bank. The zero_form specifies Bank 0 for GR16 to GR31. The one_form specifies Bank 1 for GR16 to GR31. After the bank switch the previous register bank is no longer accessible but does retain its current state. If the new and old register banks are the same, `bsw` is effectively a `nop`, although there may be a performance degradation.

A `bsw` instruction must be the last instruction in an instruction group; otherwise, operation is undefined. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    undefined_behavior();

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (PSR.vm == 1)
    virtualization_fault();

if (zero_form)
    PSR.bn = 0;
else // one_form
    PSR.bn = 1;
```

Interruptions: Privileged Operation fault   Virtualization fault

Serialization: This instruction does not require any additional instruction or data serialization operation. The bank switch occurs synchronously with its execution.

chk — Speculation Check

Format:

(*qp*) chk.s r_2, $target_{25}$	pseudo-op	
(*qp*) chk.s.i r_2, $target_{25}$	control_form, i_unit_form, gr_form	I20
(*qp*) chk.s.m r_2, $target_{25}$	control_form, m_unit_form, gr_form	M20
(*qp*) chk.s f_2, $target_{25}$	control_form, fr_form	M21
(*qp*) chk.a.*aclr* r_1, $target_{25}$	data_form, gr_form	M22
(*qp*) chk.a.*aclr* f_1, $target_{25}$	data_form, fr_form	M23

Description: The result of a control- or data-speculative calculation is checked for success or failure. If the check fails, a branch to $target_{25}$ is taken.

In the control_form, success is determined by a NaT indication for the source register. If the NaT bit corresponding to GR r_2 is 1 (in the gr_form), or FR f_2 contains a NaTVal (in the fr_form), the check fails.

In the data_form, success is determined by the ALAT. The ALAT is queried using the general register specifier r_1 (in the gr_form), or the floating-point register specifier f_1 (in the fr_form). If no ALAT entry matches, the check fails. An implementation may optionally cause the check to fail independent of whether an ALAT entry matches. A `chk.a` with general register specifier r0 or floating-point register specifiers f0 or f1 always fails.

The $target_{25}$ operand, in assembly, specifies a label to branch to. This is encoded in the instruction as a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction (imm_{21} = $target_{25}$ - IP >> 4).

The branching behavior of this instruction can be optionally unimplemented. If the instruction would have branched, and the branching behavior is not implemented, then a Speculative Operation fault is taken and the value specified by imm_{21} is zero-extended and placed in the Interruption Immediate control register (IIM). The fault handler emulates the branch by sign-extending the IIM value, adding it to IIP and returning.

The control_form of this instruction for checking general registers can be encoded on either an I-unit or an M-unit. The pseudo-op can be used if the unit type to execute on is unimportant.

For the data_form, if an ALAT entry matches, the matching ALAT entry can be optionally invalidated, based on the value of the *aclr* completer (See Table 2-14).

Table 2-14. ALAT Clear Completer

aclr Completer	Effect on ALAT
clr	Invalidate matching ALAT entry
nc	Don’t invalidate

Note that if the *clr* value of the *aclr* completer is used and the check succeeds, the matching ALAT entry is invalidated. However, if the check fails (which may happen even if there is a matching ALAT entry), any matching ALAT entry may optionally be invalidated, but this is not required. Recovery code for data speculation, therefore, cannot rely on the absence of a matching ALAT entry.

Operation:

```
if (PR[$qp$]) {
    if (control_form) {
        if (fr_form && (tmp_isrcode = fp_reg_disabled($f_2$, 0, 0, 0)))
            disabled_fp_register_fault(tmp_isrcode, 0);
        check_type = gr_form ? CHKS_GENERAL : CHKS_FLOAT;
        fail = (gr_form && GR[$r_2$].nat) || (fr_form && FR[$f_2$] == NATVAL);
    } else {                                         // data_form
        if (gr_form) {
            reg_type   = GENERAL;
            check_type = CHKA_GENERAL;
            alat_index = $r_1$;
            always_fail = (alat_index == 0);
        } else {                                     // fr_form
            reg_type   = FLOAT;
            check_type = CHKA_FLOAT;
            alat_index = $f_1$;
            always_fail = ((alat_index == 0) || (alat_index == 1));
        }
        fail = (always_fail || (!alat_cmp(reg_type, alat_index)));
    }
    if (fail) {
        if (check_branch_implemented(check_type)) {
            taken_branch = 1;
            IP = IP + sign_ext(($imm_{21}$ << 4), 25);
            if (!impl_uia_fault_supported() &&
                ((PSR.it && unimplemented_virtual_address(IP, PSR.vm))
                || (!PSR.it && unimplemented_physical_address(IP))))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(check_type, zero_ext($imm_{21}$, 21));
    } else if (data_form && ($aclr$ == 'clr'))
        alat_inval_single_entry(reg_type, alat_index);
}
```

Interruptions:

Disabled Floating-point Register fault
Speculative Operation fault
Unimplemented Instruction Address trap
Taken Branch trap

clrrrb — Clear RRB

Format:	clrrrb	all_form	B8
	clrrrb.pr	pred_form	B8

Description: In the all_form, the register rename base registers (CFM.rrb.gr, CFM.rrb.fr, and CFM.rrb.pr) are cleared. In the pred_form, the single register rename base register for the predicates (CFM.rrb.pr) is cleared.

This instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    undefined_behavior();

if (all_form) {
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
} else { // pred_form
    CFM.rrb.pr = 0;
}
```

Interruptions: None

clz — Count Leading Zeros

Format: (qp) clz r_1 = r_3 I9

Description: The number of leading zeros in GR r_3 is placed in GR r_1.

An Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See Section 3.1.11, “Processor Identification Registers” on page 1:34 for details. This capability may also be determined using the test feature (`tf`) instruction using the `@clz` operand.

Operation:

```
if (PR[qp])
    if (!instruction_implemented(CLZ))
        illegal_operation_fault();
    check_target_register(r1);

    tmp_val = 0;

    do {
        if (GR[r3]{63 - tmp_val} != 0) break;
    } while (tmp_val++ < 63);

    GR[r1] = tmp_val;
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

cmp — Compare

Format:

(*qp*)	cmp.*crel*.*ctype* $p_1, p_2 = r_2, r_3$	register_form	A6
(*qp*)	cmp.*crel*.*ctype* $p_1, p_2 = imm_8, r_3$	imm8_form	A8
(*qp*)	cmp.*crel*.*ctype* $p_1, p_2 = \text{r0}, r_3$	parallel_inequality_form	A7
(*qp*)	cmp.*crel*.*ctype* $p_1, p_2 = r_3, \text{r0}$	pseudo-op	

Description: The two source operands are compared for one of ten relations specified by *crel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*.

The compare types describe how the predicate targets are updated based on the result of the comparison. The normal type simply writes the compare result to one target, and the complement to the other. The parallel types update the targets only for a particular comparison result. This allows multiple simultaneous OR-type or multiple simultaneous AND-type compares to target the same predicate register.

The unc type is special in that it first initializes both predicate targets to 0, *independent of the qualifying predicate*. It then operates the same as the normal type. The behavior of the compare types is described in Table 2-15. A blank entry indicates the predicate target is left unchanged.

Table 2-15. Comparison Types

ctype	Pseudo-op of	PR[*qp*]==0		PR[*qp*]==1					
				Result==0, No Source NaTs		Result==1, No Source NaTs		One or More Source NaTs	
		PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none				0	1	1	0	0	0
unc		0	0	0	1	1	0	0	0
or						1	1		
and				0	0			0	0
or.andcm						1	0		
orcm	or			1	1				
andcm	and					0	0	0	0
and.orcm	or.andcm			0	1				

In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the sign-extended imm_8 encoding field; and in the parallel_inequality_form the first operand must be GR 0. The parallel_inequality_form is only used when the compare type is one of the parallel types, and the relation is an inequality (>, >=, <, <=). See below.

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is 1, or if the compare type is unc.

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation. For some of the pseudo-op compares in the imm8_form, the assembler subtracts 1 from the immediate value, making the allowed immediate range slightly different. Of the six parallel compare types, three of the types are actually pseudo-ops. The assembler simply uses the negative relation with an implemented type. The implemented relations and how the pseudo-ops map onto them are shown in Table 2-16 (for normal and unc type compares), and Table 2-17 (for parallel type compares).

Table 2-16. 64-bit Comparison Relations for Normal and unc Compares

crel	Compare Relation (*a* rel *b*)	Register Form is a pseudo-op of	Immediate Form is a pseudo-op of	Immediate Range
eq	$a == b$			-128 .. 127
ne	$a != b$	eq $p_1 \leftrightarrow p_2$	eq $p_1 \leftrightarrow p_2$	-128 .. 127
lt	$a < b$ signed			-128 .. 127
le	$a <= b$	lt $a \leftrightarrow b$ $p_1 \leftrightarrow p_2$	lt a-1	-127 .. 128
gt	$a > b$	lt $a \leftrightarrow b$	lt a-1 $p_1 \leftrightarrow p_2$	-127 .. 128
ge	$a >= b$	lt $p_1 \leftrightarrow p_2$	lt $p_1 \leftrightarrow p_2$	-128 .. 127
ltu	$a < b$ unsigned			0 .. 127, 2^{64}-128 .. 2^{64}-1
leu	$a <= b$	ltu $a \leftrightarrow b$ $p_1 \leftrightarrow p_2$	ltu a-1	1 .. 128, 2^{64}-127 .. 2^{64}
gtu	$a > b$	ltu $a \leftrightarrow b$	ltu a-1 $p_1 \leftrightarrow p_2$	1 .. 128, 2^{64}-127 .. 2^{64}
geu	$a >= b$	ltu $p_1 \leftrightarrow p_2$	ltu $p_1 \leftrightarrow p_2$	0 .. 127, 2^{64}-128 .. 2^{64}-1

The parallel compare types can be used only with a restricted set of relations and operands. They can be used with equal and not-equal comparisons between two registers or between a register and an immediate, or they can be used with inequality comparisons between a register and GR 0. Unsigned relations are not provided, since they are not of much use when one of the operands is zero. For the parallel inequality comparisons, hardware only directly implements the ones where the first operand (GR r_2) is GR 0. Comparisons where the second operand is GR 0 are pseudo-ops for which the assembler switches the register specifiers and uses the opposite relation.

Table 2-17. 64-bit Comparison Relations for Parallel Compares

crel	Compare Relation (*a* rel *b*)	Register Form is a pseudo-op of	Immediate Range
eq	$a == b$		-128 .. 127
ne	$a != b$		-128 .. 127
lt	$0 < b$ signed		no immediate forms
lt	$a < 0$	gt $a \leftrightarrow b$	
le	$0 <= b$		
le	$a <= 0$	ge $a \leftrightarrow b$	
gt	$0 > b$		
gt	$a > 0$	lt $a \leftrightarrow b$	
ge	$0 >= b$		
ge	$a >= 0$	le $a \leftrightarrow b$	

Operation:

```
if (PR[qp]) {
   if (p_1 == p_2)
      illegal_operation_fault();

   tmp_nat = (register_form ? GR[r_2].nat : 0) || GR[r_3].nat;
   if (register_form)
      tmp_src = GR[r_2];
   else if (imm8_form)
      tmp_src = sign_ext(imm_8, 8);
   else // parallel_inequality_form
      tmp_src = 0;

   if      (crel == 'eq')  tmp_rel = tmp_src == GR[r_3];
   else if (crel == 'ne')  tmp_rel = tmp_src != GR[r_3];
   else if (crel == 'lt')  tmp_rel = lesser_signed(tmp_src,  GR[r_3]);
   else if (crel == 'le')  tmp_rel = lesser_equal_signed(tmp_src, GR[r_3]);
   else if (crel == 'gt')  tmp_rel = greater_signed(tmp_src,  GR[r_3]);
   else if (crel == 'ge')  tmp_rel = greater_equal_signed(tmp_src, GR[r_3]);
   else if (crel == 'ltu') tmp_rel = lesser(tmp_src, GR[r_3]);
   else if (crel == 'leu') tmp_rel = lesser_equal(tmp_src, GR[r_3]);
   else if (crel == 'gtu') tmp_rel = greater(tmp_src, GR[r_3]);
   else                    tmp_rel = greater_equal(tmp_src, GR[r_3]);//'geu'

   switch (ctype) {
      case 'and':                                   // and-type compare
         if (tmp_nat || !tmp_rel) {
            PR[p_1] = 0;
            PR[p_2] = 0;
         }
         break;
      case 'or':                                    // or-type compare
         if (!tmp_nat && tmp_rel) {
            PR[p_1] = 1;
            PR[p_2] = 1;
         }
         break;
      case 'or.andcm':                              // or.andcm-type compare
         if (!tmp_nat && tmp_rel) {
            PR[p_1] = 1;
            PR[p_2] = 0;
         }
         break;
      case 'unc':                                   // unc-type compare
      default:                                      // normal compare
         if (tmp_nat) {
            PR[p_1] = 0;
            PR[p_2] = 0;
         } else {
            PR[p_1] = tmp_rel;
            PR[p_2] = !tmp_rel;
         }
         break;
   }
} else {
   if (ctype == 'unc') {
      if (p_1 == p_2)
```

```
                illegal_operation_fault();
            PR[$p_1$] = 0;
            PR[$p_2$] = 0;
        }
    }
```

Interruptions: Illegal Operation fault

cmp4 — Compare 4 Bytes

Format:

(qp) cmp4.*crel*.*ctype* p_1, p_2 = r_2, r_3	register_form	A6
(qp) cmp4.*crel*.*ctype* p_1, p_2 = imm_8, r_3	imm8_form	A8
(qp) cmp4.*crel*.*ctype* p_1, p_2 = r0, r_3	parallel_inequality_form	A7
(qp) cmp4.*crel*.*ctype* p_1, p_2 = r_3, r0	pseudo-op	

Description: The least significant 32 bits from each of two source operands are compared for one of ten relations specified by *crel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:39.

In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the sign-extended imm_8 encoding field; and in the parallel_inequality_form the first operand must be GR 0. The parallel_inequality_form is only used when the compare type is one of the parallel types, and the relation is an inequality (>, >=, <, <=). See the Compare instruction and Table 2-17 on page 3:40.

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is 1, or if the compare type is unc.

Of the ten relations, not all are directly implemented in hardware. Some are actually pseudo-ops. See the Compare instruction and Table 2-16 and Table 2-17 on page 3:40. The range for immediates is given below.

Table 2-18. Immediate Range for 32-bit Compares

crel	Compare Relation (*a* rel *b*)	Immediate Range
eq	*a* == *b*	-128 .. 127
ne	*a* != *b*	-128 .. 127
lt	*a* < *b* signed	-128 .. 127
le	*a* <= *b*	-127 .. 128
gt	*a* > *b*	-127 .. 128
ge	*a* >= *b*	-128 .. 127
ltu	*a* < *b* unsigned	0 .. 127, 2^{32}-128 .. 2^{32}-1
leu	*a* <= *b*	1 .. 128, 2^{32}-127 .. 2^{32}
gtu	*a* > *b*	1 .. 128, 2^{32}-127 .. 2^{32}
geu	*a* >= *b*	0 .. 127, 2^{32}-128 .. 2^{32}-1

Operation:

```
if (PR[qp]) {
   if (p1 == p2)
      illegal_operation_fault();

   tmp_nat = (register_form ? GR[r2].nat : 0) || GR[r3].nat;

   if (register_form)
      tmp_src = GR[r2];
   else if (imm8_form)
      tmp_src = sign_ext(imm8, 8);
   else // parallel_inequality_form
      tmp_src = 0;

   if      (crel == ‘eq’)   tmp_rel = tmp_src{31:0} == GR[r3]{31:0};
   else if (crel == ‘ne’)   tmp_rel = tmp_src{31:0} != GR[r3]{31:0};
   else if (crel == ‘lt’)
      tmp_rel = lesser_signed(sign_ext(tmp_src, 32),
                              sign_ext(GR[r3], 32));
   else if (crel == ‘le’)
      tmp_rel = lesser_equal_signed(sign_ext(tmp_src, 32),
                              sign_ext(GR[r3], 32));
   else if (crel == ‘gt’)
      tmp_rel = greater_signed(sign_ext(tmp_src, 32),
                              sign_ext(GR[r3], 32));
   else if (crel == ‘ge’)
      tmp_rel = greater_equal_signed(sign_ext(tmp_src, 32),
                              sign_ext(GR[r3], 32));
   else if (crel == ‘ltu’)
      tmp_rel = lesser(zero_ext(tmp_src, 32),
                              zero_ext(GR[r3], 32));
   else if (crel == ‘leu’)
      tmp_rel = lesser_equal(zero_ext(tmp_src, 32),
                              zero_ext(GR[r3], 32));
   else if (crel == ‘gtu’)
      tmp_rel = greater(zero_ext(tmp_src, 32),
                              zero_ext(GR[r3], 32));
   else        // ‘geu’
      tmp_rel = greater_equal(zero_ext(tmp_src, 32),
                              zero_ext(GR[r3], 32));

   switch (ctype) {
      case ‘and’:                                  // and-type compare
         if (tmp_nat || !tmp_rel) {
            PR[p1] = 0;
            PR[p2] = 0;
         }
         break;
      case ‘or’:                                   // or-type compare
         if (!tmp_nat && tmp_rel) {
            PR[p1] = 1;
            PR[p2] = 1;
         }
         break;
      case ‘or.andcm’:                             // or.andcm-type compare
         if (!tmp_nat && tmp_rel) {
            PR[p1] = 1;
```

```
                PR[$p_2$] = 0;
            }
            break;
        case ‘unc’:                                   // unc-type compare
        default:                                      // normal compare
            if (tmp_nat) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            } else {
                PR[$p_1$] = tmp_rel;
                PR[$p_2$] = !tmp_rel;
            }
            break;
    }
} else {
    if ($ctype$ == ‘unc’) {
        if ($p_1$ == $p_2$)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

Interruptions: Illegal Operation fault

cmpxchg — Compare and Exchange

Format: (*qp*) cmpxchg*sz*.*sem*.*ldhint* r_1 = [r_3], r_2, ar.ccv M16

(*qp*) cmp8xchg16.*sem*.*ldhint* r_1 = [r_3], r_2, ar.csd, ar.ccv sixteen_byte_form M16

Description: A value consisting of *sz* bytes (8 bytes for `cmp8xchg16`) is read from memory starting at the address specified by the value in GR r_3. The value is zero extended and compared with the contents of the `cmpxchg` Compare Value application register (AR[CCV]). If the two are equal, then the least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. For `cmp8xchg16`, if the two are equal, then 8-bytes from GR r_2 are stored at the specified address ignoring bit 3 (GR r_3 & ~0x8), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR r_3 & ~0x8) + 8). The zero-extended value read from memory is placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared.

The values of the *sz* completer are given in Table 2-19. The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-20. See Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82 for details on memory ordering.

Table 2-19. Memory Compare and Exchange Size

sz Completer	Bytes Accessed
1	1
2	2
4	4
8	8

Table 2-20. Compare and Exchange Semaphore Types

sem Completer	Ordering Semantics	Semaphore Operation
acq	Acquire	The memory read/write is made visible prior to all subsequent data memory accesses.
rel	Release	The memory read/write is made visible after all previous data memory accesses.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register). For the `cmp8xchg16` instruction, the address specified must be 8-byte aligned.

The memory read and write are guaranteed to be atomic. For the `cmp8xchg16` instruction, the 8-byte memory read and the 16-byte memory write are guaranteed to be atomic.

Both read and write access privileges for the referenced page are required. The write access privilege check is performed whether or not the memory write is performed.

This instruction is only supported to cacheable pages with write-back write policy. Accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34 on page 3:152. Locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69 for details.

For `cmp8xchg16`, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See Section 3.1.11, “Processor Identification Registers” on page 1:34 for details.

Operation:

```
if (PR[qp]) {
    size = sixteen_byte_form ? 16 : sz;

    if (sixteen_byte_form && !instruction_implemented(CMP8XCHG16))
        illegal_operation_fault();
    check_target_register(r1);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], size, SEMAPHORE, PSR.cpl, &mattr,
                         &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sixteen_byte_form) {
        if (sem == ‘acq’)
            val = mem_xchg16_cond(AR[CCV], GR[r2], AR[CSD], paddr, UM.be,
                                mattr, ACQUIRE, ldhint);
        else // ‘rel’
            val = mem_xchg16_cond(AR[CCV], GR[r2], AR[CSD], paddr, UM.be,
                                mattr, RELEASE, ldhint);
    } else {
        if (sem == ‘acq’)
            val = mem_xchg_cond(AR[CCV], GR[r2], paddr, size, UM.be, mattr,
                                ACQUIRE, ldhint);
        else // ‘rel’
            val = mem_xchg_cond(AR[CCV], GR[r2], paddr, size, UM.be, mattr,
                                RELEASE, ldhint);
        val = zero_ext(val, size * 8);
    }

    if (AR[CCV] == val)
        alat_inval_multiple_entries(paddr, size);

    GR[r1] = val;
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

cover — Cover Stack Frame

Format: cover B8

Description: A new stack frame of zero size is allocated which does not include any registers from the previous frame (as though all output registers in the previous frame had been locals). The register rename base registers are reset. If interruption collection is disabled (PSR.ic is zero), then the old value of the Current Frame Marker (CFM) is copied to the Interruption Function State register (IFS), and IFS.v is set to one.

A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

This instruction cannot be predicated.

Operation:

```
if (!followed_by_stop())
    undefined_behavior();

if (PSR.cpl == 0 && PSR.vm == 1)
    virtualization_fault();

alat_frame_update(CFM.sof, 0);
rse_preserve_frame(CFM.sof);
if (PSR.ic == 0) {
    CR[IFS].ifm = CFM;
    CR[IFS].v = 1;
}

CFM.sof = 0;
CFM.sol = 0;
CFM.sor = 0;
CFM.rrb.gr = 0;
CFM.rrb.fr = 0;
CFM.rrb.pr = 0;
```

Interruptions: Virtualization fault

czx — Compute Zero Index

Format:

(*qp*) czx1.l $r_1 = r_3$	one_byte_form, left_form	I29
(*qp*) czx1.r $r_1 = r_3$	one_byte_form, right_form	I29
(*qp*) czx2.l $r_1 = r_3$	two_byte_form, left_form	I29
(*qp*) czx2.r $r_1 = r_3$	two_byte_form, right_form	I29

Description: GR r_3 is scanned for a zero element. The element is either an 8-bit aligned byte (one_byte_form) or a 16-bit aligned pair of bytes (two_byte_form). The index of the first zero element is placed in GR r_1. If there are no zero elements in GR r_3, a default value is placed in GR r_1. Table 2-21 gives the possible result values. In the left_form, the source is scanned from most significant element to least significant element, and in the right_form it is scanned from least significant element to most significant element.

Table 2-21. Result Ranges for czx

Size	Element Width	Range of Result if Zero Element Found	Default Result if No Zero Element Found
1	8 bit	0-7	8
2	16 bit	0-3	4

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        if (left_form) {            // scan from most significant down
            if      ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        } else { // right_form      scan from least significant up
            if      ((GR[r3] & 0x00000000000000ff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x000000000000ff00) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x0000000000ff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x00000000ff000000) == 0) GR[r1] = 3;
            else if ((GR[r3] & 0x000000ff00000000) == 0) GR[r1] = 4;
            else if ((GR[r3] & 0x0000ff0000000000) == 0) GR[r1] = 5;
            else if ((GR[r3] & 0x00ff000000000000) == 0) GR[r1] = 6;
            else if ((GR[r3] & 0xff00000000000000) == 0) GR[r1] = 7;
            else GR[r1] = 8;
        }
    } else { // two_byte_form
        if (left_form) {            // scan from most significant down
            if      ((GR[r3] & 0xffff000000000000) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x0000ffff00000000) == 0) GR[r1] = 1;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 2;
            else if ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 3;
            else GR[r1] = 4;
        } else { // right_form      scan from least significant up
            if      ((GR[r3] & 0x000000000000ffff) == 0) GR[r1] = 0;
            else if ((GR[r3] & 0x00000000ffff0000) == 0) GR[r1] = 1;
            else if ((GR[$r_3$] & 0x0000ffff00000000) == 0) GR[$r_1$] = 2;
            else if ((GR[$r_3$] & 0xffff000000000000) == 0) GR[$r_1$] = 3;
            else GR[$r_1$] = 4;
        }
    }
    GR[$r_1$].nat = GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

dep — Deposit

Format:	(*qp*) dep $r_1 = r_2, r_3, pos_6, len_4$	merge_form, register_form	I15
	(*qp*) dep $r_1 = imm_1, r_3, pos_6, len_6$	merge_form, imm_form	I14
	(*qp*) dep.z $r_1 = r_2, pos_6, len_6$	zero_form, register_form	I12
	(*qp*) dep.z $r_1 = imm_8, pos_6, len_6$	zero_form, imm_form	I13

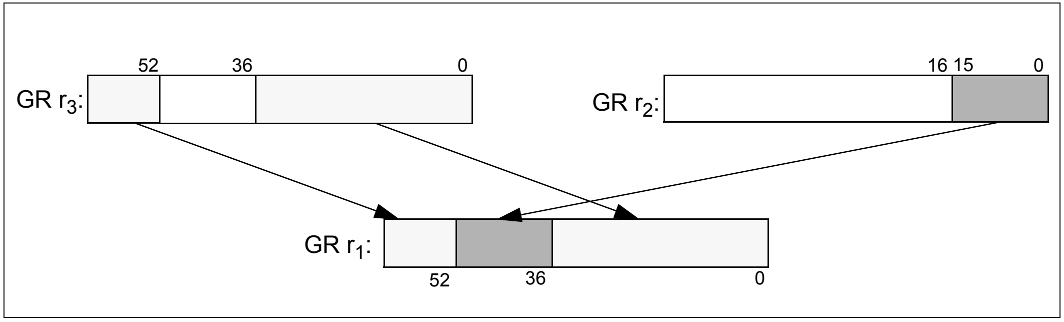
Description: In the merge_form, a right justified bit field taken from the first source operand is deposited into the value in GR r_3 at an arbitrary bit position and the result is placed in GR r_1. In the register_form the first source operand is GR r_2; and in the imm_form it is the sign-extended value specified by imm_1 (either all ones or all zeroes). The deposited bit field begins at the bit position specified by the pos_6 immediate and extends to the left (towards the most significant bit) a number of bits specified by the *len* immediate. Note that *len* has a range of 1-16 in the register_form and 1-64 in the imm_form. The pos_6 immediate has a range of 0 to 63.

In the zero_form, a right justified bit field taken from either the value in GR r_2 (in the register_form) or the sign-extended value in imm_8 (in the imm_form) is deposited into GR r_1 and all other bits in GR r_1 are cleared to zero. The deposited bit field begins at the bit position specified by the pos_6 immediate and extends to the left (towards the most significant bit) a number of bits specified by the *len* immediate. The *len* immediate has a range of 1-64 and the pos_6 immediate has a range of 0 to 63.

In the event that the deposited bit field extends beyond bit 63 of the target, i.e., *len* + pos_6 > 64, the most significant *len* + pos_6 - 64 bits of the deposited bit field are truncated. The *len* immediate is encoded as *len* minus 1 in the instruction.

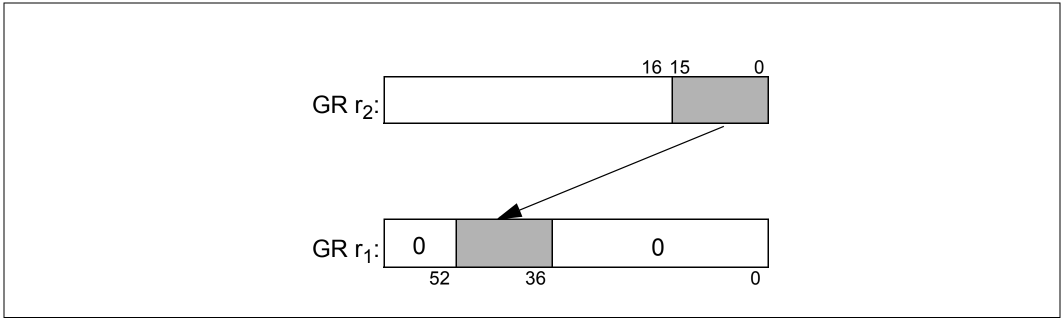
The operation of `dep` r_1 `=` r_2`,` r_3`, 36, 16` is illustrated in Figure 2-5.

Figure 2-5. Deposit Example (merge_form)



The operation of `dep.z r1 = r2, 36, 16` is illustrated in Figure 2-6.

Figure 2-6. Deposit Example (zero_form)



Operation:

```
if (PR[$qp$]) {
    check_target_register($r_1$);

    if (imm_form) {
        tmp_src = (merge_form ? sign_ext($imm_1$,1) : sign_ext($imm_8$, 8));
        tmp_nat = merge_form ? GR[$r_3$].nat : 0;
        tmp_len = $len_6$ ;
    } else {                                            // register_form
        tmp_src = GR[$r_2$];
        tmp_nat = (merge_form ? GR[$r_3$].nat : 0) || GR[$r_2$].nat;
        tmp_len = merge_form ? $len_4$ : $len_6$ ;
    }
    if ($pos_6$ + tmp_len u> 64)
        tmp_len = 64 - $pos_6$;

    if (merge_form)
        GR[$r_1$] = GR[$r_3$];
    else // zero_form
        GR[$r_1$] = 0;

    GR[$r_1$]{($pos_6$ + tmp_len - 1):$pos_6$} = tmp_src{(tmp_len - 1):0};
    GR[$r_1$].nat = tmp_nat;
}
```

Interruptions: Illegal Operation fault

epc — Enter Privileged Code

Format: epc B8

Description: This instruction increases the privilege level. The new privilege level is given by the TLB entry for the page containing this instruction. This instruction can be used to implement calls to higher-privileged routines without the overhead of an interruption.

Before increasing the privilege level, a check is performed. The PFS.ppl (previous privilege level) is checked to ensure that it is not more privileged than the current privilege level. If this check fails, the instruction takes an Illegal Operation fault.

If the check succeeds, then the privilege is increased as follows:

- If instruction address translation is enabled and the page containing the `epc` instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the `epc` instruction. This instruction can promote but cannot demote, and the new privilege comes from the TLB entry.

  If instruction address translation is disabled, then the current privilege level is set to 0 (most privileged).

  Instructions after the `epc` in the same instruction group may be executed at the old privilege level or the new, higher privilege level. Instructions in subsequent instruction groups will be executed at the new, higher privilege level.
- If the page containing the `epc` instruction has any other access rights besides execute-only, or if the privilege level assigned to the page is lower or equal to (numerically greater than or equal to) the current privilege level, then no action is taken (the current privilege level is unchanged).

Note that the ITLB is actually only read once, at instruction fetch. Information from the access rights and privilege level fields from the translation is then used in executing this instruction.

This instruction cannot be predicated.

Operation:

```
if (AR[PFS].ppl u< PSR.cpl)
    illegal_operation_fault();

if (PSR.it)
    PSR.cpl = tlb_enter_privileged_code();
else
    PSR.cpl = 0;
```

Interruptions: Illegal Operation fault

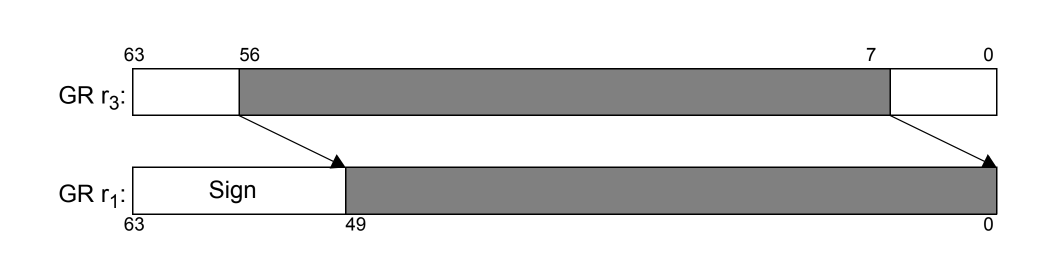
extr — Extract

Format: (qp) extr $r_1 = r_3, pos_6, len_6$ signed_form I11
(qp) extr.u $r_1 = r_3, pos_6, len_6$ unsigned_form I11

Description: A field is extracted from GR r_3, either zero extended or sign extended, and placed right-justified in GR r_1. The field begins at the bit position given by the second operand and extends len_6 bits to the left. The bit position where the field begins is specified by the pos_6 immediate. The extracted field is sign extended in the signed_form or zero extended in the unsigned_form. The sign is taken from the most significant bit of the extracted field. If the specified field extends beyond the most significant bit of GR r_3, the sign is taken from the most significant bit of GR r_3. The immediate value len_6 can be any number in the range 1 to 64, and is encoded as len_6-1 in the instruction. The immediate value pos_6 can be any value in the range 0 to 63.

The operation of `extr r1 = r3, 7, 50` is illustrated in Figure 2-7.

Figure 2-7. Extract Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_len = len6;

    if (pos6 + tmp_len u> 64)
        tmp_len = 64 - pos6;

    if (unsigned_form)
        GR[r1] = zero_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);
    else // signed_form
        GR[r1] = sign_ext(shift_right_unsigned(GR[r3], pos6), tmp_len);

    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

fabs — Floating-point Absolute Value

Format: (*qp*) fabs f_1 = f_3 pseudo-op of: (*qp*) fmerge.s f_1 = f0, f_3

Description: The absolute value of the value in FR f_3 is computed and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See “fmerge — Floating-point Merge” on page 3:80.

fadd — Floating-point Add

Format: (qp) fadd.pc.sf $f_1 = f_3, f_2$ pseudo-op of: (qp) fma.pc.sf $f_1 = f_3$, f1, f_2

Description: FR f_3 and FR f_2 are added (computed to infinite precision), rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1. If either FR f_3 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22. The mnemonic values for *sf* are given in Table 2-23. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Table 2-22. Specified *pc* Mnemonic Values

pc Mnemonic	Precision Specified
.s	single
.d	double
none	dynamic (i.e. use pc value in status field)

Table 2-23. *sf* Mnemonic Values

sf Mnemonic	Status Field Accessed
.s0 or *none*	sf0
.s1	sf1
.s2	sf2
.s3	sf3

Operation: See "fma — Floating-point Multiply Add" on page 3:77.

famax — Floating-point Absolute Maximum

Format: (*qp*) famax.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the larger absolute value is placed in FR f_1. If the magnitude of FR f_2 equals the magnitude of FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_right = fp_reg_read(FR[f2]);
        tmp_left = fp_reg_read(FR[f3]);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

famin — Floating-point Absolute Minimum

Format: (qp) famin.sf $f_1 = f_2, f_3$ F8

Description: The operand with the smaller absolute value is placed in FR f_1. If the magnitude of FR f_2 equals the magnitude of FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_left = fp_reg_read(FR[f2]);
        tmp_right = fp_reg_read(FR[f3]);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }

    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fand — Floating-point Logical And

Format: (*qp*) fand $f_1 = f_2, f_3$ F9

Description: The bit-wise logical AND of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fandcm — Floating-point And Complement

Format: (*qp*) fandcm $f_1 = f_2, f_3$ F9

Description: The bit-wise logical AND of the significand field of FR f_2 with the bit-wise complemented significand field of FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand & ~FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fc — Flush Cache

Format:	(qp) fc r_3	invalidate_line_form	M28
	(qp) fc.i r_3	instruction_cache_coherent_form	M28

Description: In the invalidate_line form, the cache line associated with the address specified by the value of GR r_3 is invalidated from all levels of the processor cache hierarchy. The invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory it is written to memory before invalidation. The line size affected is at least 32-bytes (aligned on a 32-byte boundary). An implementation may flush a larger region.

In the instruction_cache_coherent form, the cache line specified by GR r_3 is flushed in an implementation-specific manner that ensures that the instruction caches are coherent with the data caches. The `fc.i` instruction is not required to invalidate the targeted cache line nor write the targeted cache line back to memory if it is inconsistent with memory, but may do so if this is required to make the instruction caches coherent with the data caches. The `fc.i` instruction is broadcast throughout the coherence domain if necessary to make all instruction caches coherent. The line size affected is at least 32-bytes (aligned on a 32-byte boundary). An implementation may flush a larger region.

When executed at privilege level 0, `fc` and `fc.i` perform no access rights or protection key checks. At other privilege levels, `fc` and `fc.i` perform access rights checks as if they were 1-byte reads, but do not perform any protection key checks (regardless of PSR.pk).

The memory attribute of the page containing the affected line has no effect on the behavior of these instructions. The `fc` instruction can be used to remove a range of addresses from the cache by first changing the memory attribute to non-cacheable and then flushing the range.

These instructions follow data dependency ordering rules; they are ordered only with respect to previous load, store or semaphore instructions to the same line. `fc` and `fc.i` have data dependencies in the sense that any prior stores by this processor will be included in the flush operation. Subsequent memory operations to the same line need not wait for prior `fc` or `fc.i` completion before being globally visible. `fc` and `fc.i` are unordered operations, and are not affected by a memory fence (`mf`) instruction. These instructions are ordered with respect to the `sync.i` instruction.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|FC|READ;
    if (GR[r3].nat)
        register_nat_consumption_fault(itype);
    tmp_paddr = tlb_translate_nonaccess(GR[r3], itype);

    if (invalidate_line_form)
        mem_flush(tmp_paddr);
    else // instruction_cache_coherent_form
        make_icache_coherent(tmp_paddr);
}
```

Interruptions: Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Access Rights fault

fchkf — Floating-point Check Flags

Format: (*qp*) fchkf.*sf* *target*$_{25}$ F14

Description: The flags in FPSR.*sf*.flags are compared with FPSR.s0.flags and FPSR.traps. If any flags set in FPSR.*sf*.flags correspond to FPSR.traps which are enabled, or if any flags set in FPSR.*sf*.flags are not set in FPSR.s0.flags, then a branch to *target*$_{25}$ is taken.

The *target*$_{25}$ operand, specifies a label to branch to. This is encoded in the instruction as a signed immediate displacement (*imm*$_{21}$) between the target bundle and the bundle containing this instruction (*imm*$_{21}$ = *target*$_{25}$ - IP >> 4).

The branching behavior of this instruction can be optionally unimplemented. If the instruction would have branched, and the branching behavior is not implemented, then a Speculative Operation fault is taken and the value specified by *imm*$_{21}$ is zero-extended and placed in the Interruption Immediate control register (IIM). The fault handler emulates the branch by sign-extending the IIM value, adding it to IIP and returning.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    switch (sf) {
        case ‘s0’:
            tmp_flags = AR[FPSR].sf0.flags;
            break;
        case ‘s1’:
            tmp_flags = AR[FPSR].sf1.flags;
            break;
        case ‘s2’:
            tmp_flags = AR[FPSR].sf2.flags;
            break;
        case ‘s3’:
            tmp_flags = AR[FPSR].sf3.flags;
            break;
    }
    if ((tmp_flags & ~AR[FPSR].traps) || (tmp_flags & ~AR[FPSR].sf0.flags)) {
        if (check_branch_implemented(FCHKF)) {
            taken_branch = 1;
            IP = IP + sign_ext((imm21 << 4), 25);
            if (!impl_uia_fault_supported() &&
                ((PSR.it && unimplemented_virtual_address(IP, PSR.vm))
                || (!PSR.it && unimplemented_physical_address(IP)))
                unimplemented_instruction_address_trap(0, IP);
            if (PSR.tb)
                taken_branch_trap();
        } else
            speculation_fault(FCHKF, zero_ext(imm21, 21));
    }
}
```

FP Exceptions: None

Interruptions: Speculative Operation fault
Unimplemented Instruction Address trap
Taken Branch trap

fclass — Floating-point Class

Format: (*qp*) fclass.*fcrel*.*fctype* p_1, p_2 = f_2, *fclass*$_9$ F5

Description: The contents of FR f_2 are classified according to the *fclass*$_9$ completer as shown in Table 2-25. This produces a boolean result based on whether the contents of FR f_2 agrees with the floating-point number format specified by *fclass*$_9$, as specified by the *fcrel* completer. This result is written to the two predicate register destinations, p_1 and p_2. The result written to the destinations is determined by the compare type specified by *fctype*.

The allowed types are Normal (or *none*) and unc. See Table 2-26 on page 3:67. The assembly syntax allows the specification of membership or non-membership and the assembler swaps the target predicates to achieve the desired effect.

Table 2-24. Floating-point Class Relations

fcrel	Test Relation
m	FR f_2 agrees with the pattern specified by *fclass*$_9$ (is a member)
nm	FR f_2 does not agree with the pattern specified by *fclass*$_9$ (is not a member)

A number agrees with the pattern specified by *fclass*$_9$ if:

- the number is NaTVal and *fclass*$_9$ {8} is 1, or
- the number is a quiet NaN and *fclass*$_9$ {7} is 1, or
- the number is a signaling NaN and *fclass*$_9$ {6} is 1, or
- the sign of the number agrees with the sign specified by one of the two low-order bits of *fclass*$_9$, and the type of the number (disregarding the sign) agrees with the number-type specified by the next four bits of *fclass*$_9$, as shown in Table 2-25.

Note: An *fclass*$_9$ of 0x1FF is equivalent to testing for any supported operand.

The class names used in Table 2-25 are defined in Table 5-2, “Floating-point Register Encodings” on page 1:86.

Table 2-25. Floating-point Classes

fclass$_9$	Class	Mnemonic
Either these cases can be tested for		
0x0100	NaTVal	@nat
0x080	Quiet NaN	@qnan
0x040	Signaling NaN	@snan
or the OR of the following two cases		
0x001	Positive	@pos
0x002	Negative	@neg
AND’ed with OR of the following four cases		
0x004	Zero	@zero
0x008	Unnormalized	@unorm
0x010	Normalized	@norm
0x020	Infinity	@inf

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    tmp_rel = ((fclass9{0} && !FR[f2].sign || fclass9{1} && FR[f2].sign)
                &&  ((fclass9{2} && fp_is_zero(FR[f2]))||
                     (fclass9{3} && fp_is_unorm(FR[f2])) ||
                     (fclass9{4} && fp_is_normal(FR[f2])) ||
                     (fclass9{5} && fp_is_inf(FR[f2]))
                    )
                )
            || (fclass9{6} && fp_is_snan(FR[f2]))
            || (fclass9{7} && fp_is_qnan(FR[f2]))
            || (fclass9{8} && fp_is_natval(FR[f2]));

    tmp_nat = fp_is_natval(FR[f2]) && (!fclass9{8});

    if (tmp_nat) {
        PR[p1] = 0;
        PR[p2] = 0;
    } else {
        PR[p1] = tmp_rel;
        PR[p2] = !tmp_rel;
    }
} else {
    if (fctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault   Disabled Floating-point Register fault

fclrf — Floating-point Clear Flags

Format: (*qp*) fclrf.*sf* F13

Description: The status field's 6-bit flags field is reset to zero.
The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_set_sf_flags(sf, 0);
}
```

FP Exceptions: None

Interruptions: None

fcmp — Floating-point Compare

Format: (qp) fcmp.*frel.fctype.sf* p_1, p_2 = f_2, f_3 F4

Description: The two source operands are compared for one of twelve relations specified by *frel*. This produces a boolean result which is 1 if the comparison condition is true, and 0 otherwise. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *fctype*. The allowed types are Normal (or *none*) and unc.

Table 2-26. Floating-point Comparison Types

fctype	PR[*qp*]==0		PR[*qp*]==1					
			Result==0, No Source NaTVals		Result==1, No Source NaTVals		One or More Source NaTVals	
	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]	PR[p_1]	PR[p_2]
none			0	1	1	0	0	0
unc	0	0	0	1	1	0	0	0

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

The relations are defined for each of the comparison types in Table 2-27. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate target specifiers and uses an implemented relation.

Table 2-27. Floating-point Comparison Relations

frel	*frel* Completer Unabbreviated	Relation	Pseudo-op of	Quiet NaN as Operand Signals Invalid
eq	equal	$f_2 == f_3$		No
lt	less than	$f_2 < f_3$		Yes
le	less than or equal	$f_2 <= f_3$		Yes
gt	greater than	$f_2 > f_3$	lt $f_2 \leftrightarrow f_3$	Yes
ge	greater than or equal	$f_2 >= f_3$	le $f_2 \leftrightarrow f_3$	Yes
unord	unordered	$f_2 ? f_3$		No
neq	not equal	$!(f_2 == f_3)$	eq $p_1 \leftrightarrow p_2$	No
nlt	not less than	$!(f_2 < f_3)$	lt $p_1 \leftrightarrow p_2$	Yes
nle	not less than or equal	$!(f_2 <= f_3)$	le $p_1 \leftrightarrow p_2$	Yes
ngt	not greater than	$!(f_2 > f_3)$	lt $f_2 \leftrightarrow f_3$ $p_1 \leftrightarrow p_2$	Yes
nge	not greater than or equal	$!(f_2 >= f_3)$	le $f_2 \leftrightarrow f_3$ $p_1 \leftrightarrow p_2$	Yes
ord	ordered	$!(f_2 ? f_3)$	unord $p_1 \leftrightarrow p_2$	No

Operation:

```
if (PR[qp]) {
    if ($p_1$ == $p_2$)
        illegal_operation_fault();

    if (tmp_isrcode = fp_reg_disabled($f_2$, $f_3$, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$])) {
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    } else {
        fcmp_exception_fault_check($f_2$, $f_3$, frel, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = fp_reg_read(FR[$f_2$]);
        tmp_fr3 = fp_reg_read(FR[$f_3$]);

        if       (frel == ‘eq’)  tmp_rel = fp_equal(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘lt’)  tmp_rel = fp_less_than(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘le’)  tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘gt’)  tmp_rel = fp_less_than(tmp_fr3,
                                                      tmp_fr2);
        else if (frel == ‘ge’)  tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                      tmp_fr2);
        else if (frel == ‘unord’)tmp_rel = fp_unordered(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘neq’) tmp_rel = !fp_equal(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘nlt’) tmp_rel = !fp_less_than(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘nle’) tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                      tmp_fr3);
        else if (frel == ‘ngt’) tmp_rel = !fp_less_than(tmp_fr3,
                                                      tmp_fr2);
        else if (frel == ‘nge’) tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                      tmp_fr2);
        else                    tmp_rel = !fp_unordered(tmp_fr2,
                                                      tmp_fr3); //‘ord’

        PR[$p_1$] = tmp_rel;
        PR[$p_2$] = !tmp_rel;

        fp_update_fpsr(sf, tmp_fp_env);
    }
} else {
    if (fctype == ‘unc’) {
        if ($p_1$ == $p_2$)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fcvt.fx — Convert Floating-point to Integer

Format:

(*qp*) fcvt.fx.*sf* $f_1 = f_2$	signed_form	F10
(*qp*) fcvt.fx.trunc.*sf* $f_1 = f_2$	signed_form, trunc_form	F10
(*qp*) fcvt.fxu.*sf* $f_1 = f_2$	unsigned_form	F10
(*qp*) fcvt.fxu.trunc.*sf* $f_1 = f_2$	unsigned_form, trunc_form	F10

Description: FR f_2 is treated as a register format floating-point value and converted to a signed (signed_form) or unsigned integer (unsigned_form) using either the rounding mode specified in the FPSR.*sf.rc*, or using Round-to-Zero if the trunc_form of the instruction is used. The result is placed in the 64-bit significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). If the result of the conversion cannot be represented as a 64-bit integer, the 64-bit integer indefinite value 0x8000000000000000 is used as the result, if the IEEE Invalid Operation Floating-point Exception fault is disabled.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fcvt_exception_fault_check(f2, signed_form,
                                        trunc_form, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1].significand = INTEGER_INDEFINITE;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        } else {
            tmp_res = fp_ieee_rnd_to_int(fp_reg_read(FR[f2]), &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            FR[f1].significand = tmp_res.significand;
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = FP_SIGN_POSITIVE;
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:	Invalid Operation (V)	Inexact (I)
	Denormal/Unnormal Operand (D)	
	Software Assist (SWA) fault	
Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

fcvt.xf — Convert Signed Integer to Floating-point

Format: (*qp*) fcvt.xf $f_1 = f_2$ F11

Description: The 64-bit significand of FR f_2 is treated as a signed integer and its register file precision floating-point representation is placed in FR f_1.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation is always exact and is unaffected by the rounding mode.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$])) {
        FR[$f_1$] = NATVAL;
    } else {
        tmp_res = FR[$f_2$];
        if (tmp_res.significand{63}) {
            tmp_res.significand = (~tmp_res.significand) + 1;
            tmp_res.sign = 1;
        } else
            tmp_res.sign = 0;

        tmp_res.exponent = FP_INTEGER_EXP;
        tmp_res = fp_normalize(tmp_res);

        FR[$f_1$].significand = tmp_res.significand;
        FR[$f_1$].exponent = tmp_res.exponent;
        FR[$f_1$].sign = tmp_res.sign;
    }
    fp_update_psr($f_1$);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault

Disabled Floating-point Register fault

fcvt.xuf — Convert Unsigned Integer to Floating-point

Format: (*qp*) fcvt.xuf.*pc*.*sf* $f_1 = f_3$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, f0

Description: FR f_3 is multiplied with FR 1, rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

Note: Multiplying FR f_3 with FR 1 (a 1.0) normalizes the canonical representation of an integer in the floating-point register file producing a normal floating-point value.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation: See "fma — Floating-point Multiply Add" on page 3:77.

fetchadd — Fetch and Add Immediate

Format: (qp) fetchadd4.*sem.ldhint* r_1 = [r_3], inc_3 four_byte_form M17
(qp) fetchadd8.*sem.ldhint* r_1 = [r_3], inc_3 eight_byte_form M17

Description: A value consisting of four or eight bytes is read from memory starting at the address specified by the value in GR r_3. The value is zero extended and added to the sign-extended immediate value specified by inc_3. The values that may be specified by inc_3 are: -16, -8, -4, -1, 1, 4, 8, 16. The least significant four or eight bytes of the sum are then written to memory starting at the address specified by the value in GR r_3. The zero-extended value read from memory is placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared.

The *sem* completer specifies the type of semaphore operation. These operations are described in Table 2-28. See Section 4.4.7, "Sequentiality Attribute and Ordering" on page 2:82 for details on memory ordering.

Table 2-28. Fetch and Add Semaphore Types

sem Completer	Ordering Semantics	Semaphore Operation
acq	Acquire	The memory read/write is made visible prior to all subsequent data memory accesses.
rel	Release	The memory read/write is made visible after all previous data memory accesses.

The memory read and write are guaranteed to be atomic for accesses to pages with cacheable, writeback memory attribute. For accesses to other memory types, atomicity is platform dependent. Details on memory attributes are described in Section 4.4, "Memory Attributes" on page 2:75.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register).

Both read and write access privileges for the referenced page are required. The write access privilege check is performed whether or not the memory write is performed.

Only accesses to UCE pages or cacheable pages with write-back write policy are permitted. Accesses to NaTPages result in a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

On a processor model that supports exported `fetchadd`, a `fetchadd` to a UCE page causes the fetch-and-add operation to be exported outside of the processor; if the platform does not support exported `fetchadd`, the operation is undefined. On a processor model that does not support exported `fetchadd`, a `fetchadd` to a UCE page causes an Unsupported Data Reference fault. See Section 4.4.9, "Effects of Memory Attributes on Memory Reference Instructions" on page 2:86.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34 on page 3:152. Locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat)
        register_nat_consumption_fault(SEMAPHORE);

    size = four_byte_form ? 4 : 8;

    paddr = tlb_translate(GR[r3], size, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);
    if (!ma_supports_fetchadd(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    if (sem == 'acq')
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, ACQUIRE, ldhint);
    else // 'rel'
        val = mem_xchg_add(inc3, paddr, size, UM.be, mattr, RELEASE, ldhint);

    alat_inval_multiple_entries(paddr, size);

    GR[r1] = zero_ext(val, size * 8);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

flushrs — Flush Register Stack

Format: flushrs M25

Description: All stacked general registers in the dirty partition of the register stack are written to the backing store before execution continues. The dirty partition contains registers from previous procedure frames that have not yet been saved to the backing store. For a description of the register stack partitions, refer to Chapter 6, "Register Stack Engine" in Volume 2. A pending external interrupt can interrupt the RSE store loop when enabled.

After this instruction completes execution BSPSTORE is equal to BSP.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

```
while (AR[BSPSTORE] != AR[BSP]) {
    rse_store(MANDATORY);              // increments AR[BSPSTORE]
    deliver_unmasked_pending_external_interrupt();
}
```

Interruptions:

Unimplemented Data Address fault
VHPT Data fault
Data Nested TLB fault
Data TLB fault
Alternate Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

fma — Floating-point Multiply Add

Format: (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision and then FR f_2 is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and add. See "fmpy — Floating-point Multiply" on page 3:85.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fma_exception_fault_check(f2, f3, f4,
                                                         pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

fmax — Floating-point Maximum

Format: (*qp*) fmax.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the larger value is placed in FR f_1. If FR f_2 equals FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f3]),
                                    fp_reg_read(FR[f2]));
        FR[f1] = (tmp_bool_res ? FR[f2] : FR[f3]);

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fmerge — Floating-point Merge

Format:	(*qp*) fmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
	(*qp*) fmerge.s $f_1 = f_2, f_3$	sign_form	F9
	(*qp*) fmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

Description: Sign, exponent and significand fields are extracted from FR f_2 and FR f_3, combined, and the result is placed in FR f_1.

For the neg_sign_form, the sign of FR f_2 is negated and concatenated with the exponent and the significand of FR f_3. This form can be used to negate a floating-point number by using the same register for FR f_2 and FR f_3.

For the sign_form, the sign of FR f_2 is concatenated with the exponent and the significand of FR f_3.

For the sign_exp_form, the sign and exponent of FR f_2 is concatenated with the significand of FR f_3.

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-8. Floating-point Merge Negative Sign Operation

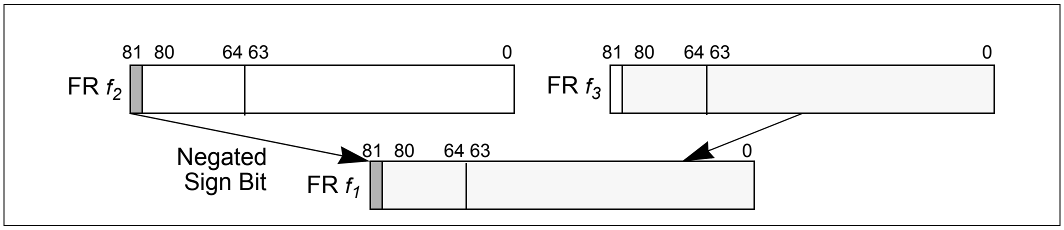


Figure 2-9. Floating-point Merge Sign Operation

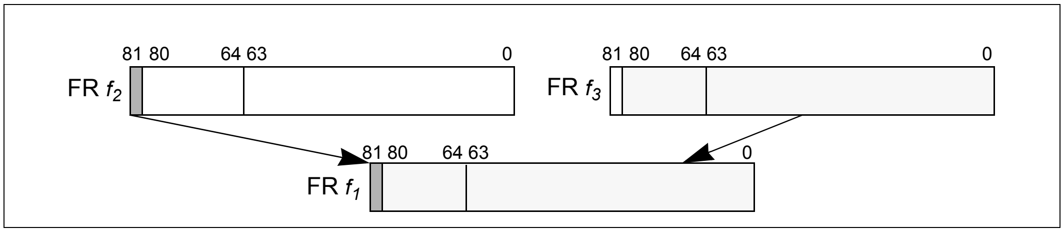
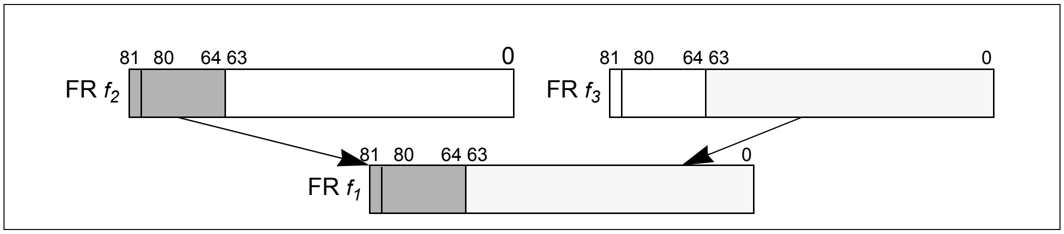


Figure 2-10. Floating-point Merge Sign and Exponent Operation



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f3].significand;
        if (neg_sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = !FR[f2].sign;
        } else if (sign_form) {
            FR[f1].exponent = FR[f3].exponent;
            FR[f1].sign = FR[f2].sign;
        } else {                                     // sign_exp_form
            FR[f1].exponent = FR[f2].exponent;
            FR[f1].sign = FR[f2].sign;
        }
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fmin — Floating-point Minimum

Format: (qp) fmin.*sf* $f_1 = f_2, f_3$ F8

Description: The operand with the smaller value is placed in FR f_1. If FR f_2 equals FR f_3, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaN, FR f_1 gets FR f_3.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as the `fcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_bool_res = fp_less_than(fp_reg_read(FR[f2]),
                                    fp_reg_read(FR[f3]));
        FR[f1] = tmp_bool_res ? FR[f2] : FR[f3];

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions:

Illegal Operation fault	Floating-point Exception fault
Disabled Floating-point Register fault	

fmix — Floating-point Mix

Format:

(*qp*) fmix.l $f_1 = f_2, f_3$	mix_l_form	F9
(*qp*) fmix.r $f_1 = f_2, f_3$	mix_r_form	F9
(*qp*) fmix.lr $f_1 = f_2, f_3$	mix_lr_form	F9

Description: For the mix_l_form (mix_r_form), the left (right) single precision value in FR f_2 is concatenated with the left (right) single precision value in FR f_3. For the mix_lr_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-11. Floating-point Mix Left

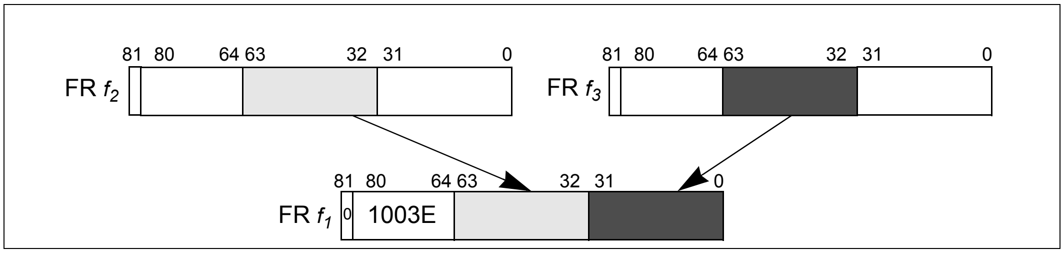


Figure 2-12. Floating-point Mix Right

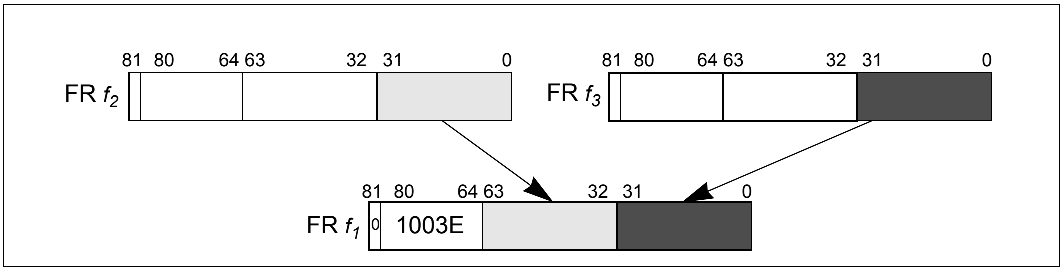
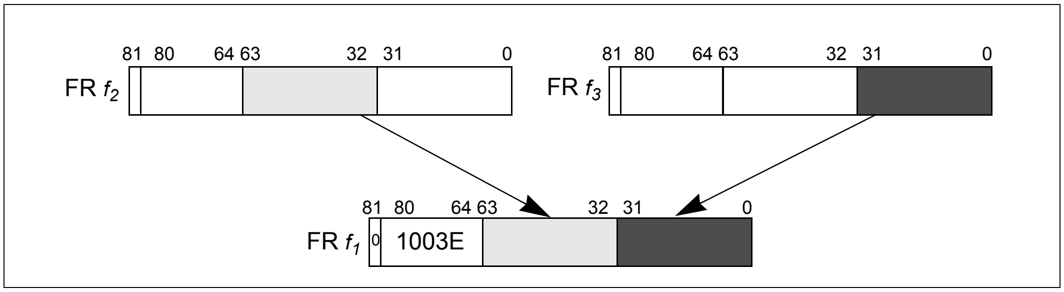


Figure 2-13. Floating-point Mix Left-Right



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f_1);
    if (tmp_isrcode = fp_reg_disabled(f_1, f_2, f_3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f_2]) || fp_is_natval(FR[f_3])) {
        FR[f_1] = NATVAL;
    } else {
        if (mix_l_form) {
            tmp_res_hi = FR[f_2].significand{63:32};
            tmp_res_lo = FR[f_3].significand{63:32};
        } else if (mix_r_form) {
            tmp_res_hi = FR[f_2].significand{31:0};
            tmp_res_lo = FR[f_3].significand{31:0};
        } else {                                    // mix_lr_form
            tmp_res_hi = FR[f_2].significand{63:32};
            tmp_res_lo = FR[f_3].significand{31:0};
        }
        FR[f_1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f_1].exponent = FP_INTEGER_EXP;
        FR[f_1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f_1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault   Disabled Floating-point Register fault

fmpy — Floating-point Multiply

Format: (*qp*) fmpy.*pc*.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4$, f0

Description: The product FR f_3 and FR f_4 is computed to infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
See "fma — Floating-point Multiply Add" on page 3:77.
```

fms — Floating-point Multiply Subtract

Format: (qp) fms.pc.sf $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision and then FR f_2 is subtracted from this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf.pc* and FPSR.*sf.wre*) using the rounding mode specified by FPSR.*sf.rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, a NaTVal is placed in FR f_1 instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed instead of a multiply and subtract. See "fmpy — Floating-point Multiply" on page 3:85.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                             pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_fr2 = fp_reg_read(FR[f2]);
            tmp_fr2.sign = !tmp_fr2.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, tmp_fr2, tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)	Underflow (U)
Denormal/Unnormal Operand (D)	Overflow (O)
Software Assist (SWA) fault	Inexact (I)
	Software Assist (SWA) trap

Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

fneg — Floating-point Negate

Format: (*qp*) fneg $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.ns $f_1 = f_3, f_3$

Description: The value in FR f_3 is negated and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: `See “fmerge — Floating-point Merge” on page 3:80.`

fnegabs — Floating-point Negate Absolute Value

Format: (*qp*) fnegabs $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.ns f_1 = f0, f_3

Description: The absolute value of the value in FR f_3 is computed, negated, and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: `See “fmerge — Floating-point Merge” on page 3:80.`

fnma — Floating-point Negative Multiply Add

Format: (*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The product of FR f_3 and FR f_4 is computed to infinite precision, negated, and then FR f_2 is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

If f_2 is f0, an IEEE multiply operation is performed, followed by negation of the product. See "fnmpy — Floating-point Negative Multiply" on page 3:92.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result = fms_fnma_exception_fault_check(f2, f3, f4,
                                                    pc, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
        } else {
            tmp_res = fp_mul(fp_reg_read(FR[f3]), fp_reg_read(FR[f4]));
            tmp_res.sign = !tmp_res.sign;
            if (f2 != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read(FR[f2]), tmp_fp_env);
            FR[f1] = fp_ieee_round(tmp_res, &tmp_fp_env);
        }

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

fnmpy — Floating-point Negative Multiply

Format: (*qp*) fnmpy.*pc*.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4$,f0

Description: The product FR f_3 and FR f_4 is computed to infinite precision and then negated. The resulting value is then rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*. The rounded result is placed in FR f_1.

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
See "fnma — Floating-point Negative Multiply Add" on page 3:90.
```

fnorm — Floating-point Normalize

Format: (qp) fnorm.*pc*.*sf* $f_1 = f_3$ pseudo-op of: (qp) fma.*pc*.*sf* $f_1 = f_3$, f1, f0

Description: FR f_3 is normalized and rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
See "fma — Floating-point Multiply Add" on page 3:77.
```

for — Floating-point Logical Or

Format: (qp) for $f_1 = f_2, f_3$ F9

Description: The bit-wise logical OR of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand | FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fpabs — Floating-point Parallel Absolute Value

Format: (*qp*) fpabs $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.s f_1 = f0, f_3

Description: The absolute values of the pair of single precision values in the significand field of FR f_3 are computed and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "fpmerge — Floating-point Parallel Merge" on page 3:111.

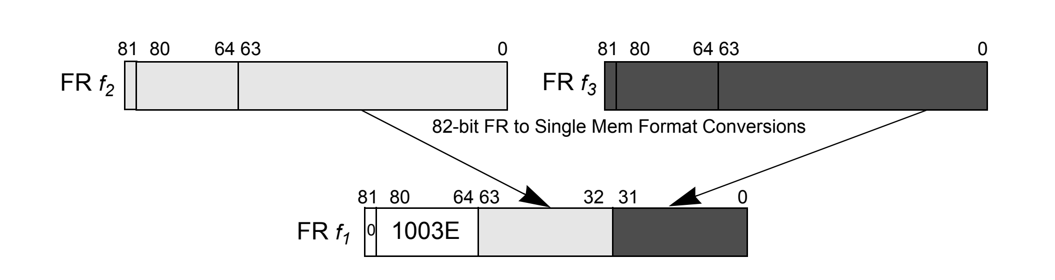
fpack — Floating-point Pack

Format: (*qp*) fpack $f_1 = f_2, f_3$ pack_form F9

Description: The register format numbers in FR f_2 and FR f_3 are converted to single precision memory format. These two single precision numbers are concatenated and stored in the significand field of FR f_1 . The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-14. Floating-point Pack



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        tmp_res_hi = fp_single(FR[f2]);
        tmp_res_lo = fp_single(FR[f3]);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }
    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fpamax — Floating-point Parallel Absolute Maximum

Format: (*qp*) fpamax.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 and FR f_3 are compared. The operands with the larger absolute value are returned in the significand field of FR f_1.

If the magnitude of high (low) FR f_3 is less than the magnitude of high (low) FR f_2, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, and neither FR f_2 or FR f_3 is a NaTVal, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, $f_3$, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$])) {
        FR[$f_1$] = NATVAL;
    } else {
        fpminmax_exception_fault_check($f_2$, $f_3$, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi($f_2$);
        tmp_fr3 = tmp_left = fp_reg_read_hi($f_3$);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo($f_2$);
        tmp_fr3 = tmp_left = fp_reg_read_lo($f_3$);
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr($f_1$);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fpamin — Floating-point Parallel Absolute Minimum

Format: (*qp*) fpamin.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the smaller absolute value is returned in the significand of FR f_1.

If the magnitude of high (low) FR f_2 is less than the magnitude of high (low) FR f_3, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, and neither FR f_2 or FR f_3 is a NaTVal, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_left.sign = FP_SIGN_POSITIVE;
        tmp_right.sign = FP_SIGN_POSITIVE;
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fpcmp — Floating-point Parallel Compare

Format: (qp) fpcmp.*frel.sf* $f_1 = f_2, f_3$ F8

Description: The two pairs of single precision source operands in the significand fields of FR f_2 and FR f_3 are compared for one of twelve relations specified by *frel*. This produces a boolean result which is a mask of 32 1's if the comparison condition is true, and a mask of 32 0's otherwise. This result is written to a pair of 32-bit integers in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Table 2-29. Floating-point Parallel Comparison Results

PR[*qp*]==0	PR[*qp*]==1		
	Result==false, No Source NaTVals	**Result==true, No Source NaTVals**	**One or More Source NaTVals**
unchanged	0...0	1...1	NaTVal

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

The relations are defined for each of the comparison types in Table 2-29. Of the twelve relations, not all are directly implemented in hardware. Some are actually pseudo-ops. For these, the assembler simply switches the source operand specifiers and/or switches the predicate type specifiers and uses an implemented relation.

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Table 2-30. Floating-point Parallel Comparison Relations

frel	*frel* Completer Unabbreviated	Relation	Pseudo-op of		Quiet NaN as Operand Signals Invalid
eq	equal	$f_2 == f_3$			No
lt	less than	$f_2 < f_3$			Yes
le	less than or equal	$f_2 <= f_3$			Yes
gt	greater than	$f_2 > f_3$	lt	$f_2 \leftrightarrow f_3$	Yes
ge	greater than or equal	$f_2 >= f_3$	le	$f_2 \leftrightarrow f_3$	Yes
unord	unordered	$f_2 ? f_3$			No
neq	not equal	$!(f_2 == f_3)$			No
nlt	not less than	$!(f_2 < f_3)$			Yes
nle	not less than or equal	$!(f_2 <= f_3)$			Yes
ngt	not greater than	$!(f_2 > f_3)$	nlt	$f_2 \leftrightarrow f_3$	Yes
nge	not greater than or equal	$!(f_2 >= f_3)$	nle	$f_2 \leftrightarrow f_3$	Yes
ord	ordered	$!(f_2 ? f_3)$			No

Operation:

```
if (PR[qp]) {
   fp_check_target_register(f1);
   if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
      disabled_fp_register_fault(tmp_isrcode, 0);

   if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
      FR[f1] = NATVAL;
   } else {
      fpcmp_exception_fault_check(f2, f3, frel, sf, &tmp_fp_env);

      if (fp_raise_fault(tmp_fp_env))
         fp_exception_fault(fp_decode_fault(tmp_fp_env));

      tmp_fr2 = fp_reg_read_hi(f2);
      tmp_fr3 = fp_reg_read_hi(f3);

      if      (frel == 'eq')   tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
      else if (frel == 'lt')   tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
      else if (frel == 'le')   tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                     tmp_fr3);
      else if (frel == 'gt')   tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
      else if (frel == 'ge')   tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                     tmp_fr2);
      else if (frel == 'unord')tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
      else if (frel == 'neq')  tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
      else if (frel == 'nlt')  tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
      else if (frel == 'nle')  tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                     tmp_fr3);
      else if (frel == 'ngt')  tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
      else if (frel == 'nge')  tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                     tmp_fr2);
      else                     tmp_rel = !fp_unordered(tmp_fr2,
                                                     tmp_fr3); //'ord'

      tmp_res_hi = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

      tmp_fr2 = fp_reg_read_lo(f2);
      tmp_fr3 = fp_reg_read_lo(f3);

      if      (frel == 'eq')   tmp_rel = fp_equal(tmp_fr2, tmp_fr3);
      else if (frel == 'lt')   tmp_rel = fp_less_than(tmp_fr2, tmp_fr3);
      else if (frel == 'le')   tmp_rel = fp_lesser_or_equal(tmp_fr2,
                                                     tmp_fr3);
      else if (frel == 'gt')   tmp_rel = fp_less_than(tmp_fr3, tmp_fr2);
      else if (frel == 'ge')   tmp_rel = fp_lesser_or_equal(tmp_fr3,
                                                     tmp_fr2);
      else if (frel == 'unord')tmp_rel = fp_unordered(tmp_fr2, tmp_fr3);
      else if (frel == 'neq')  tmp_rel = !fp_equal(tmp_fr2, tmp_fr3);
      else if (frel == 'nlt')  tmp_rel = !fp_less_than(tmp_fr2, tmp_fr3);
      else if (frel == 'nle')  tmp_rel = !fp_lesser_or_equal(tmp_fr2,
                                                     tmp_fr3);
      else if (frel == 'ngt')  tmp_rel = !fp_less_than(tmp_fr3, tmp_fr2);
      else if (frel == 'nge')  tmp_rel = !fp_lesser_or_equal(tmp_fr3,
                                                     tmp_fr2);
      else                     tmp_rel = !fp_unordered(tmp_fr2,
                                                     tmp_fr3); //'ord'
```

```
        tmp_res_lo = (tmp_rel ? 0xFFFFFFFF : 0x00000000);

        FR[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr($f_1$);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fpcvt.fx — Convert Parallel Floating-point to Integer

Format:	(*qp*) fpcvt.fx.*sf* $f_1 = f_2$	signed_form	F10
	(*qp*) fpcvt.fx.trunc.*sf* $f_1 = f_2$	signed_form, trunc_form	F10
	(*qp*) fpcvt.fxu.*sf* $f_1 = f_2$	unsigned_form	F10
	(*qp*) fpcvt.fxu.trunc.*sf* $f_1 = f_2$	unsigned_form, trunc_form	F10

Description: The pair of single precision values in the significand field of FR f_2 is converted to a pair of 32-bit signed integers (signed_form) or unsigned integers (unsigned_form) using either the rounding mode specified in the FPSR.*sf.rc*, or using Round-to-Zero if the trunc_form of the instruction is used. The result is written as a pair of 32-bit integers into the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). If the result of the conversion cannot be represented as a 32-bit integer, the 32-bit integer indefinite value 0x80000000 is used as the result, if the IEEE Invalid Operation Floating-point Exception fault is disabled.

If FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpcvt_exception_fault_check(f2,
                                    signed_form, trunc_form, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_hi(f2), HIGH,
                                                &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            tmp_res_hi = tmp_res.significand{31:0};
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = INTEGER_INDEFINITE_32_BIT;
        } else {
            tmp_res = fp_ieee_rnd_to_int_sp(fp_reg_read_lo(f2), LOW,
                                                &tmp_fp_env);
            if (tmp_res.exponent)
                tmp_res.significand = fp_U64_rsh(
                    tmp_res.significand, (FP_INTEGER_EXP - tmp_res.exponent));
            if (signed_form && tmp_res.sign)
                tmp_res.significand = (~tmp_res.significand) + 1;

            tmp_res_lo = tmp_res.significand{31:0};
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) Fault

Inexact (I)

Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	Floating-point Exception trap

fpma — Floating-point Parallel Multiply Add

Format: (*qp*) fpma.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then the pair of single precision values in the significand field of FR f_2 is added to these products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

Note: If f_2 is f0 in the fpma instruction, just the IEEE multiply operation is performed. (See "fpmpy — Floating-point Parallel Multiply" on page 3:115.) FR f1, as an operand, is not a packed pair of 1.0 values, it is just the register file format's 1.0 value.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, $f_3$, $f_4$))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$]) ||
        fp_is_natval(FR[$f_4$])) {
        FR[$f_1$] = NATVAL;
        fp_update_psr($f_1$);
    } else {
        tmp_default_result_pair = fpma_exception_fault_check($f_2$,
                                        $f_3$, $f_4$, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi($f_3$), fp_reg_read_hi($f_4$));
            if ($f_2$ != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi($f_2$), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo($f_3$), fp_reg_read_lo($f_4$));
            if ($f_2$ != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo($f_2$), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr($f_1$);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) Fault
Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

fpmax — Floating-point Parallel Maximum

Format: (*qp*) fpmax.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the larger value is returned in the significand of FR f_1.

If the value of high (low) FR f_3 is less than the value of high (low) FR f_2, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_right = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_left = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        tmp_fr2 = tmp_right = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_left = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2 : tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fpmerge — Floating-point Parallel Merge

Format:

(*qp*) fpmerge.ns $f_1 = f_2, f_3$	neg_sign_form	F9
(*qp*) fpmerge.s $f_1 = f_2, f_3$	sign_form	F9
(*qp*) fpmerge.se $f_1 = f_2, f_3$	sign_exp_form	F9

Description: For the neg_sign_form, the signs of the pair of single precision values in the significand field of FR f_2 are negated and concatenated with the exponents and the significands of the pair of single precision values in the significand field of FR f_3 and stored in the significand field of FR f_1. This form can be used to negate a pair of single precision floating-point numbers by using the same register for f_2 and f_3.

For the sign_form, the signs of the pair of single precision values in the significand field of FR f_2 are concatenated with the exponents and the significands of the pair of single precision values in the significand field of FR f_3 and stored in FR f_1.

For the sign_exp_form, the signs and exponents of the pair of single precision values in the significand field of FR f_2 are concatenated with the pair of single precision significands in the significand field of FR f_3 and stored in the significand field of FR f_1.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-15. Floating-point Parallel Merge Negative Sign Operation

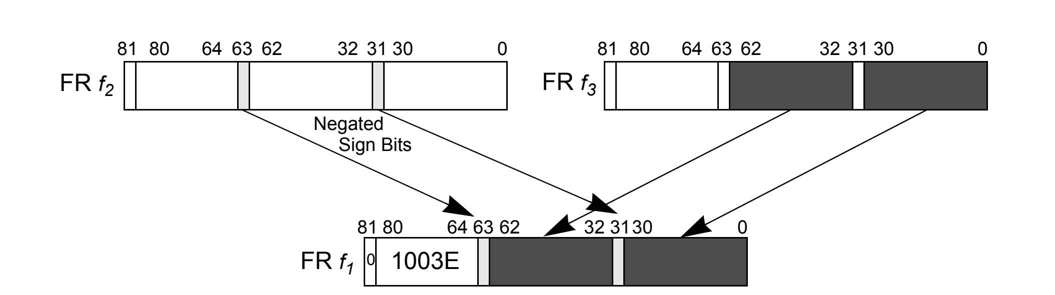


Figure 2-16. Floating-point Parallel Merge Sign Operation

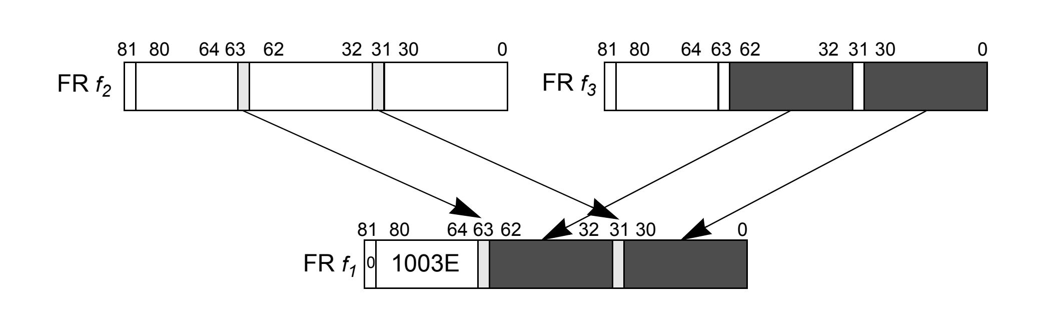
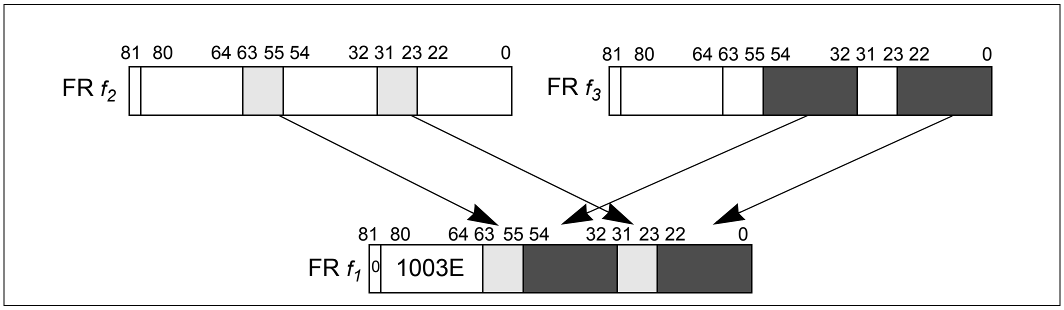


Figure 2-17. Floating-point Parallel Merge Sign and Exponent Operation



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (neg_sign_form) {
            tmp_res_hi = (!FR[f2].significand{63} << 31)
                         | (FR[f3].significand{62:32});
            tmp_res_lo = (!FR[f2].significand{31} << 31)
                         | (FR[f3].significand{30:0});
        } else if (sign_form) {
            tmp_res_hi = (FR[f2].significand{63} << 31)
                         | (FR[f3].significand{62:32});
            tmp_res_lo = (FR[f2].significand{31} << 31)
                         | (FR[f3].significand{30:0});
        } else {                                       // sign_exp_form
            tmp_res_hi = (FR[f2].significand{63:55} << 23)
                         | (FR[f3].significand{54:32});
            tmp_res_lo = (FR[f2].significand{31:23} << 23)
                         | (FR[f3].significand{22:0});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault   Disabled Floating-point Register fault

fpmin — Floating-point Parallel Minimum

Format: (*qp*) fpmin.*sf* $f_1 = f_2, f_3$ F8

Description: The paired single precision values in the significands of FR f_2 or FR f_3 are compared. The operands with the smaller value is returned in significand of FR f_1.

If the value of high (low) FR f_2 is less than the value of high (low) FR f_3, high (low) FR f_1 gets high (low) FR f_2. Otherwise high (low) FR f_1 gets high (low) FR f_3.

If high (low) FR f_2 or high (low) FR f_3 is a NaN, high (low) FR f_1 gets high (low) FR f_3.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

This operation does not propagate NaNs the same way as other arithmetic floating-point instructions. The Invalid Operation is signaled in the same manner as for the `fpcmp.lt` operation.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        fpminmax_exception_fault_check(f2, f3, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        tmp_fr2 = tmp_left = fp_reg_read_hi(f2);
        tmp_fr3 = tmp_right = fp_reg_read_hi(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_hi = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        tmp_fr2 = tmp_left = fp_reg_read_lo(f2);
        tmp_fr3 = tmp_right = fp_reg_read_lo(f3);
        tmp_bool_res = fp_less_than(tmp_left, tmp_right);
        tmp_res_lo = fp_single(tmp_bool_res ? tmp_fr2: tmp_fr3);

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

fpmpy — Floating-point Parallel Multiply

Format: (*qp*) fpmpy.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fpma.*sf* $f_1 = f_3, f_4$, f0

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_3, or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation: `See "fpma — Floating-point Parallel Multiply Add" on page 3:107.`

fpms — Floating-point Parallel Multiply Subtract

Format: (qp) fpms.sf $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then the pair of single precision values in the significand field of FR f_2 is subtracted from these products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Note: If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

Mapping: If f_2 is f0 in the fpms instruction, just the IEEE multiply operation is performed.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, f4))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3]) ||
        fp_is_natval(FR[f4])) {
        FR[f1] = NATVAL;
        fp_update_psr(f1);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check(f2, f3,
                                                      f4, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi(f3), fp_reg_read_hi(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_hi(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo(f3), fp_reg_read_lo(f4));
            if (f2 != 0) {
                tmp_sub = fp_reg_read_lo(f2);
                tmp_sub.sign = !tmp_sub.sign;
                tmp_res = fp_add(tmp_res, tmp_sub, tmp_fp_env);
            }
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr(f1);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault
Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

fpneg — Floating-point Parallel Negate

Format: (*qp*) fpneg $f_1 = f_3$ pseudo-op of: (*qp*) fpmerge.ns $f_1 = f_3, f_3$

Description: The pair of single precision values in the significand field of FR f_3 are negated and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See "fpmerge — Floating-point Parallel Merge" on page 3:111.

fpnegabs — Floating-point Parallel Negate Absolute Value

Format: (*qp*) fpnegabs f_1 = f_3 pseudo-op of: (*qp*) fpmerge.ns f_1 = f0, f_3

Description: The absolute values of the pair of single precision values in the significand field of FR f_3 are computed, negated and stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation: See “fpmerge — Floating-point Parallel Merge” on page 3:111.

fpnma — Floating-point Parallel Negative Multiply Add

Format: (*qp*) fpnma.*sf* $f_1 = f_3, f_4, f_2$ F1

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision, negated, and then the pair of single precision values in the significand field of FR f_2 are added to these (negated) products, again in infinite precision. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Note: If f_2 is f0 in the fpnma instruction, just the IEEE multiply operation (with the product being negated before rounding) is performed.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, $f_3$, $f_4$))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$]) ||
        fp_is_natval(FR[$f_4$])) {
        FR[$f_1$] = NATVAL;
        fp_update_psr($f_1$);
    } else {
        tmp_default_result_pair = fpms_fpnma_exception_fault_check($f_2$, $f_3$,
                                                        $f_4$, sf, &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
        } else {
            tmp_res = fp_mul(fp_reg_read_hi($f_3$), fp_reg_read_hi($f_4$));
            tmp_res.sign = !tmp_res.sign;
            if ($f_2$ != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_hi($f_2$), tmp_fp_env);
            tmp_res_hi = fp_ieee_round_sp(tmp_res, HIGH, &tmp_fp_env);
        }

        if (fp_is_nan_or_inf(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
        } else {
            tmp_res = fp_mul(fp_reg_read_lo($f_3$), fp_reg_read_lo($f_4$));
            tmp_res.sign = !tmp_res.sign;
            if ($f_2$ != 0)
                tmp_res = fp_add(tmp_res, fp_reg_read_lo($f_2$), tmp_fp_env);
            tmp_res_lo = fp_ieee_round_sp(tmp_res, LOW, &tmp_fp_env);
        }

        FR[$f_1$].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;

        fp_update_fpsr(sf, tmp_fp_env);
        fp_update_psr($f_1$);
        if (fp_raise_traps(tmp_fp_env))
            fp_exception_trap(fp_decode_trap(tmp_fp_env));
    }
}
```

FP Exceptions:

Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault
Underflow (U)
Overflow (O)
Inexact (I)
Software Assist (SWA) trap

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault
Floating-point Exception trap

fpnmpy — Floating-point Parallel Negative Multiply

Format: (*qp*) fpnmpy.*sf* $f_1 = f_3, f_4$ pseudo-op of: (*qp*) fpnma.*sf* $f_1 = f_3, f_4$,f0

Description: The pair of products of the pairs of single precision values in the significand fields of FR f_3 and FR f_4 are computed to infinite precision and then negated. The resulting values are then rounded to single precision using the rounding mode specified by FPSR.*sf.rc*. The pair of rounded results are stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either FR f_3 or FR f_4 is a NaTVal, FR f_1 is set to NaTVal instead of the computed results.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.
The encodings and interpretation for the status field's *rc* are given in Table 5-6 on page 1:90.

Operation: See "fpnma — Floating-point Parallel Negative Multiply Add" on page 3:120.

fprcpa — Floating-point Parallel Reciprocal Approximation

Format: (*qp*) fprcpa.*sf* f_1, p_2 = f_2, f_3 F6

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- Each half of the significand of FR f_1 is either set to an approximation (with a relative error < $2^{-8.886}$) of the reciprocal of the corresponding half of FR f_3, or set to the IEEE-754 mandated response for the quotient FR f_2/FR f_3 of the corresponding half — if that half of FR f_2 or of FR f_3 is in the set {-Infinity, -0, +0, +Infinity, NaN}.
- If either half of FR f_1 is set to the IEEE-754 mandated quotient, or is set to an approximation of the reciprocal which may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 divide result, then PR p_2 is set to 0, otherwise it is set to 1.

  For correct IEEE divide results, when PR p_2 is cleared, user software is expected to compute the quotient (FR f_2/FR f_3) for each half (using the non-parallel `frcpa` instruction), and merge the results into FR f_1, keeping PR p_2 cleared.
- The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).
- If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprcpa_exception_fault_check(f2, f3, sf,
                                            &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result_pair.hi) ||
            limits_check.hi_fr3) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            num = fp_normalize(fp_reg_read_hi(f2));
            den = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_hi = 0;
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.hi_fr2_or_quot)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
            }
            tmp_res_hi = fp_single(tmp_res);
        }
        if (fp_is_nan_or_inf(tmp_default_result_pair.lo) ||
            limits_check.lo_fr3) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            num = fp_normalize(fp_reg_read_lo(f2));
            den = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                tmp_res = FP_ZERO;
                tmp_res.sign = num.sign ^ den.sign;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip(den);
                if (limits_check.lo_fr2_or_quot)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP Exceptions: Invalid Operation (V)
Zero Divide (Z)

Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Floating-point Exception fault
Disabled Floating-point Register fault

fprsqrta — Floating-point Parallel Reciprocal Square Root Approximation

Format: (*qp*) fprsqrta.*sf* f_1, p_2 = f_3 F7

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- Each half of the significand of FR f_1 is either set to an approximation (with a relative error < $2^{-8.831}$) of the reciprocal square root of the corresponding half of FR f_3, or set to the IEEE-754 compliant response for the reciprocal square root of the corresponding half of FR f_3 — if that half of FR f_3 is in the set {-Infinity, -Finite, -0, +0, +Infinity, NaN}.
- If either half of FR f_1 is set to the IEEE-754 mandated reciprocal square root, or is set to an approximation of the reciprocal square root which may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 square root result, then PR p_2 is set to 0, otherwise it is set to 1.

  For correct IEEE square root results, when PR p_2 is cleared, user software is expected to compute the square root for each half (using the non-parallel `frsqrta` instruction), and merge the results in FR f_1, keeping PR p_2 cleared.
- The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).
- If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result_pair = fprsqrta_exception_fault_check(f3, sf,
                                            &tmp_fp_env, &limits_check);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result_pair.hi)) {
            tmp_res_hi = fp_single(tmp_default_result_pair.hi);
            tmp_pred_hi = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_hi(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_hi = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_hi = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.hi)
                    tmp_pred_hi = 0;
                else
                    tmp_pred_hi = 1;
            }
            tmp_res_hi = fp_single(tmp_res);
        }

        if (fp_is_nan(tmp_default_result_pair.lo)) {
            tmp_res_lo = fp_single(tmp_default_result_pair.lo);
            tmp_pred_lo = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read_lo(f3));
            if (fp_is_zero(tmp_fr3)) {
                tmp_res = FP_INFINITY;
                tmp_res.sign = tmp_fr3.sign;
                tmp_pred_lo = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                tmp_res = FP_ZERO;
                tmp_pred_lo = 0;
            } else {
                tmp_res = fp_ieee_recip_sqrt(tmp_fr3);
                if (limits_check.lo)
                    tmp_pred_lo = 0;
                else
                    tmp_pred_lo = 1;
            }
            tmp_res_lo = fp_single(tmp_res);
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
        PR[p2] = tmp_pred_hi && tmp_pred_lo;

        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

frcpa — Floating-point Reciprocal Approximation

Format: (*qp*) frcpa.*sf* f_1, p_2 = f_2, f_3 F6

Description: If PR *qp* is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR *qp* is 1, the following will occur:

- FR f_1 is either set to an approximation (with a relative error < $2^{-8.886}$) of the reciprocal of FR f_3, or to the IEEE-754 mandated quotient of FR f_2/FR f_3 — if either FR f_2 or FR f_3 is in the set {-Infinity, -0, Pseudo-zero, +0, +Infinity, NaN, Unsupported}.
- If FR f_1 is set to the approximation of the reciprocal of FR f_3, then PR p_2 is set to 1; otherwise, it is set to 0.
- If FR f_2 and FR f_3 are such that the approximation of FR f_3's reciprocal may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 result of FR f_2/FR f_3, then a Floating-point Exception fault for Software Assist occurs.

  System software is expected to compute the IEEE-754 quotient (FR f_2/FR f_3), return the result in FR f_1, and set PR p_2 to 0.
- If either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frcpa_exception_fault_check(f2, f3, sf,
                                                          &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan_or_inf(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            num = fp_normalize(fp_reg_read(FR[f2]));
            den = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_inf(num) && fp_is_finite(den)) {
                FR[f1] = FP_INFINITY;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_finite(num) && fp_is_inf(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else if (fp_is_zero(num) && fp_is_finite(den)) {
                FR[f1] = FP_ZERO;
                FR[f1].sign = num.sign ^ den.sign;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip(den);
                PR[p2] = 1;
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip()

fp_ieee_recip(den)
{
    RECIP_TABLE[256] = {
        0x3fc, 0x3f4, 0x3ec, 0x3e4, 0x3dd, 0x3d5, 0x3cd, 0x3c6,
        0x3be, 0x3b7, 0x3af, 0x3a8, 0x3a1, 0x399, 0x392, 0x38b,
        0x384, 0x37d, 0x376, 0x36f, 0x368, 0x361, 0x35b, 0x354,
        0x34d, 0x346, 0x340, 0x339, 0x333, 0x32c, 0x326, 0x320,
        0x319, 0x313, 0x30d, 0x307, 0x300, 0x2fa, 0x2f4, 0x2ee,
        0x2e8, 0x2e2, 0x2dc, 0x2d7, 0x2d1, 0x2cb, 0x2c5, 0x2bf,
        0x2ba, 0x2b4, 0x2af, 0x2a9, 0x2a3, 0x29e, 0x299, 0x293,
        0x28e, 0x288, 0x283, 0x27e, 0x279, 0x273, 0x26e, 0x269,
        0x264, 0x25f, 0x25a, 0x255, 0x250, 0x24b, 0x246, 0x241,
        0x23c, 0x237, 0x232, 0x22e, 0x229, 0x224, 0x21f, 0x21b,
        0x216, 0x211, 0x20d, 0x208, 0x204, 0x1ff, 0x1fb, 0x1f6,
        0x1f2, 0x1ed, 0x1e9, 0x1e5, 0x1e0, 0x1dc, 0x1d8, 0x1d4,
        0x1cf, 0x1cb, 0x1c7, 0x1c3, 0x1bf, 0x1bb, 0x1b6, 0x1b2,
        0x1ae, 0x1aa, 0x1a6, 0x1a2, 0x19e, 0x19a, 0x197, 0x193,
        0x18f, 0x18b, 0x187, 0x183, 0x17f, 0x17c, 0x178, 0x174,
        0x171, 0x16d, 0x169, 0x166, 0x162, 0x15e, 0x15b, 0x157,
        0x154, 0x150, 0x14d, 0x149, 0x146, 0x142, 0x13f, 0x13b,
        0x138, 0x134, 0x131, 0x12e, 0x12a, 0x127, 0x124, 0x120,
        0x11d, 0x11a, 0x117, 0x113, 0x110, 0x10d, 0x10a, 0x107,
        0x103, 0x100, 0x0fd, 0x0fa, 0x0f7, 0x0f4, 0x0f1, 0x0ee,
        0x0eb, 0x0e8, 0x0e5, 0x0e2, 0x0df, 0x0dc, 0x0d9, 0x0d6,
        0x0d3, 0x0d0, 0x0cd, 0x0ca, 0x0c8, 0x0c5, 0x0c2, 0x0bf,
        0x0bc, 0x0b9, 0x0b7, 0x0b4, 0x0b1, 0x0ae, 0x0ac, 0x0a9,
        0x0a6, 0x0a4, 0x0a1, 0x09e, 0x09c, 0x099, 0x096, 0x094,
        0x091, 0x08e, 0x08c, 0x089, 0x087, 0x084, 0x082, 0x07f,
        0x07c, 0x07a, 0x077, 0x075, 0x073, 0x070, 0x06e, 0x06b,
        0x069, 0x066, 0x064, 0x061, 0x05f, 0x05d, 0x05a, 0x058,
        0x056, 0x053, 0x051, 0x04f, 0x04c, 0x04a, 0x048, 0x045,
        0x043, 0x041, 0x03f, 0x03c, 0x03a, 0x038, 0x036, 0x033,
        0x031, 0x02f, 0x02d, 0x02b, 0x029, 0x026, 0x024, 0x022,
        0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x015, 0x013, 0x011,
        0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
    };

    tmp_index = den.significand{62:55};
    tmp_res.significand = (1 << 63) | (RECIP_TABLE[tmp_index] << 53);
    tmp_res.exponent = FP_REG_EXP_ONES - 2 - den.exponent;
    tmp_res.sign = den.sign;
    return (tmp_res);
}
```

FP Exceptions: Invalid Operation (V)
Zero Divide (Z)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Floating-point Exception fault

frsqrta — Floating-point Reciprocal Square Root Approximation

Format: (qp) frsqrta.sf f_1, p_2 = f_3 F7

Description: If PR qp is 0, PR p_2 is cleared and FR f_1 remains unchanged.

If PR qp is 1, the following will occur:

- FR f_1 is either set to an approximation (with a relative error < $2^{-8.831}$) of the reciprocal square root of FR f_3, or set to the IEEE-754 mandated square root of FR f_3 — if FR f_3 is in the set {-Infinity, -Finite, -0, Pseudo-zero, +0, +Infinity, NaN, Unsupported}.
- If FR f_1 is set to an approximation of the reciprocal square root of FR f_3, then PR p_2 is set to 1; otherwise, it is set to 0.
- If FR f_3 is such the approximation of its reciprocal square root may cause the Newton-Raphson iterations to fail to produce the correct IEEE-754 square root result, then a Floating-point Exception fault for Software Assist occurs.

  System software is expected to compute the IEEE-754 square root, return the result in FR f_1, and set PR p_2 to 0.
- If FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result, and PR p_2 is cleared.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f3, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
        PR[p2] = 0;
    } else {
        tmp_default_result = frsqrta_exception_fault_check(f3, sf,
                                                      &tmp_fp_env);
        if (fp_raise_fault(tmp_fp_env))
            fp_exception_fault(fp_decode_fault(tmp_fp_env));

        if (fp_is_nan(tmp_default_result)) {
            FR[f1] = tmp_default_result;
            PR[p2] = 0;
        } else {
            tmp_fr3 = fp_normalize(fp_reg_read(FR[f3]));
            if (fp_is_zero(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else if (fp_is_pos_inf(tmp_fr3)) {
                FR[f1] = tmp_fr3;
                PR[p2] = 0;
            } else {
                FR[f1] = fp_ieee_recip_sqrt(tmp_fr3);
                PR[p2] = 1;
            }
        }
        fp_update_fpsr(sf, tmp_fp_env);
    }
    fp_update_psr(f1);
} else {
    PR[p2] = 0;
}

// fp_ieee_recip_sqrt()

fp_ieee_recip_sqrt(root)
{
    RECIP_SQRT_TABLE[256] = {
        0x1a5, 0x1a0, 0x19a, 0x195, 0x18f, 0x18a, 0x185, 0x180,
        0x17a, 0x175, 0x170, 0x16b, 0x166, 0x161, 0x15d, 0x158,
        0x153, 0x14e, 0x14a, 0x145, 0x140, 0x13c, 0x138, 0x133,
        0x12f, 0x12a, 0x126, 0x122, 0x11e, 0x11a, 0x115, 0x111,
        0x10d, 0x109, 0x105, 0x101, 0x0fd, 0x0fa, 0x0f6, 0x0f2,
        0x0ee, 0x0ea, 0x0e7, 0x0e3, 0x0df, 0x0dc, 0x0d8, 0x0d5,
        0x0d1, 0x0ce, 0x0ca, 0x0c7, 0x0c3, 0x0c0, 0x0bd, 0x0b9,
        0x0b6, 0x0b3, 0x0b0, 0x0ad, 0x0a9, 0x0a6, 0x0a3, 0x0a0,
        0x09d, 0x09a, 0x097, 0x094, 0x091, 0x08e, 0x08b, 0x088,
        0x085, 0x082, 0x07f, 0x07d, 0x07a, 0x077, 0x074, 0x071,
        0x06f, 0x06c, 0x069, 0x067, 0x064, 0x061, 0x05f, 0x05c,
        0x05a, 0x057, 0x054, 0x052, 0x04f, 0x04d, 0x04a, 0x048,
        0x045, 0x043, 0x041, 0x03e, 0x03c, 0x03a, 0x037, 0x035,
        0x033, 0x030, 0x02e, 0x02c, 0x029, 0x027, 0x025, 0x023,
        0x020, 0x01e, 0x01c, 0x01a, 0x018, 0x016, 0x014, 0x011,
        0x00f, 0x00d, 0x00b, 0x009, 0x007, 0x005, 0x003, 0x001,
        0x3fc, 0x3f4, 0x3ec, 0x3e5, 0x3dd, 0x3d5, 0x3ce, 0x3c7,
        0x3bf, 0x3b8, 0x3b1, 0x3aa, 0x3a3, 0x39c, 0x395, 0x38e,
        0x388, 0x381, 0x37a, 0x374, 0x36d, 0x367, 0x361, 0x35a,
        0x354, 0x34e, 0x348, 0x342, 0x33c, 0x336, 0x330, 0x32b,
        0x325, 0x31f, 0x31a, 0x314, 0x30f, 0x309, 0x304, 0x2fe,
        0x2f9, 0x2f4, 0x2ee, 0x2e9, 0x2e4, 0x2df, 0x2da, 0x2d5,
        0x2d0, 0x2cb, 0x2c6, 0x2c1, 0x2bd, 0x2b8, 0x2b3, 0x2ae,
        0x2aa, 0x2a5, 0x2a1, 0x29c, 0x298, 0x293, 0x28f, 0x28a,
        0x286, 0x282, 0x27d, 0x279, 0x275, 0x271, 0x26d, 0x268,
        0x264, 0x260, 0x25c, 0x258, 0x254, 0x250, 0x24c, 0x249,
        0x245, 0x241, 0x23d, 0x239, 0x235, 0x232, 0x22e, 0x22a,
        0x227, 0x223, 0x220, 0x21c, 0x218, 0x215, 0x211, 0x20e,
        0x20a, 0x207, 0x204, 0x200, 0x1fd, 0x1f9, 0x1f6, 0x1f3,
        0x1f0, 0x1ec, 0x1e9, 0x1e6, 0x1e3, 0x1df, 0x1dc, 0x1d9,
        0x1d6, 0x1d3, 0x1d0, 0x1cd, 0x1ca, 0x1c7, 0x1c4, 0x1c1,
        0x1be, 0x1bb, 0x1b8, 0x1b5, 0x1b2, 0x1af, 0x1ac, 0x1aa,
    };

    tmp_index = (root.exponent{0} << 7) | root.significand{62:56};
    tmp_res.significand = (1 << 63) | (RECIP_SQRT_TABLE[tmp_index] << 53);
    tmp_res.exponent = FP_REG_EXP_HALF -
                         ((root.exponent - FP_REG_BIAS) >> 1);
    tmp_res.sign = FP_SIGN_POSITIVE;
    return (tmp_res);
}
```

FP Exceptions: Invalid Operation (V)
Denormal/Unnormal Operand (D)
Software Assist (SWA) fault

Interruptions:	Illegal Operation fault	Floating-point Exception fault
	Disabled Floating-point Register fault	

fselect — Floating-point Select

Format: (*qp*) fselect $f_1 = f_3, f_4, f_2$ F3

Description: The significand field of FR f_3 is logically AND-ed with the significand field of FR f_2 and the significand field of FR f_4 is logically AND-ed with the one's complement of the significand field of FR f_2. The two results are logically OR-ed together. The result is placed in the significand field of FR f_1.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E). The sign bit field of FR f_1 is set to positive (0).

If any of FR f_3, FR f_4, or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, $f_3$, $f_4$))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$]) ||
        fp_is_natval(FR[$f_4$])) {
        FR[$f_1$] = NATVAL;
    } else {
        FR[$f_1$].significand   = (FR[$f_3$].significand & FR[$f_2$].significand)
                                | (FR[$f_4$].significand & ~FR[$f_2$].significand);
        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr($f_1$);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fsetc — Floating-point Set Controls

Format: (*qp*) fsetc.*sf* *amask*$_7$, *omask*$_7$ F12

Description: The status field's control bits are initialized to the value obtained by logically AND-ing the sf0.controls and *amask*$_7$ immediate field and logically OR-ing the *omask*$_7$ immediate field.

The mnemonic values for *sf* are given in Table 2-23 on page 3:56.

Operation:

```
if (PR[qp]) {
    tmp_controls = (AR[FPSR].sf0.controls & amask7) | omask7;
    if (is_reserved_field(FSETC, sf, tmp_controls))
        reserved_register_field_fault();
    fp_set_sf_controls(sf, tmp_controls);
}
```

FP Exceptions: None

Interruptions: Reserved Register/Field fault

fsub — Floating-point Subtract

Format: (qp) fsub.*pc*.*sf* $f_1 = f_3, f_2$ pseudo-op of: (qp) fms.*pc*.*sf* $f_1 = f_3$, f1, f_2

Description: FR f_2 is subtracted from FR f_3 (computed to infinite precision), rounded to the precision indicated by *pc* (and possibly FPSR.*sf*.*pc* and FPSR.*sf*.*wre*) using the rounding mode specified by FPSR.*sf*.*rc*, and placed in FR f_1.

If either FR f_3 or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

The mnemonic values for the opcode's *pc* are given in Table 2-22 on page 3:56. The mnemonic values for *sf* are given in Table 2-23 on page 3:56. For the encodings and interpretation of the status field's *pc*, *wre*, and *rc*, refer to Table 5-5 and Table 5-6 on page 1:90.

Operation:

```
See "fms — Floating-point Multiply Subtract" on page 3:86.
```

fswap — Floating-point Swap

Format:	(qp) fswap $f_1 = f_2, f_3$	swap_form	F9
	(qp) fswap.nl $f_1 = f_2, f_3$	swap_nl_form	F9
	(qp) fswap.nr $f_1 = f_2, f_3$	swap_nr_form	F9

Description: For the swap_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped.

For the swap_nl_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the left single precision value is negated.

For the swap_nr_form, the left single precision value in FR f_2 is concatenated with the right single precision value in FR f_3. The concatenated pair is then swapped, and the right single precision value is negated.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-18. Floating-point Swap

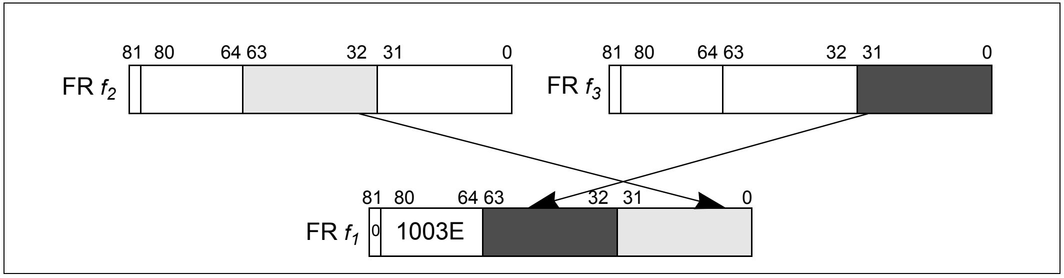


Figure 2-19. Floating-point Swap Negate Left

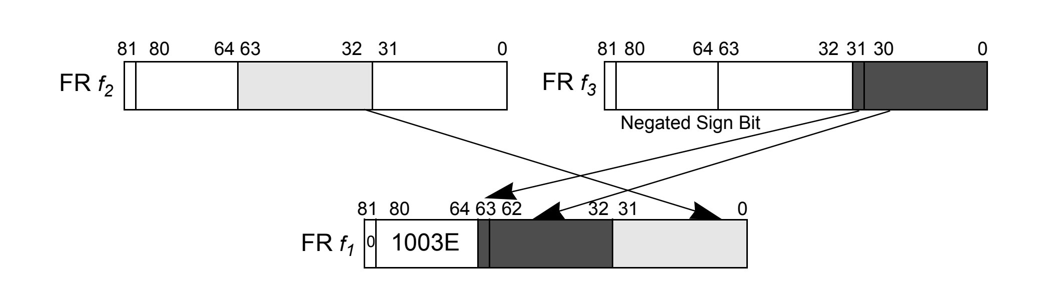
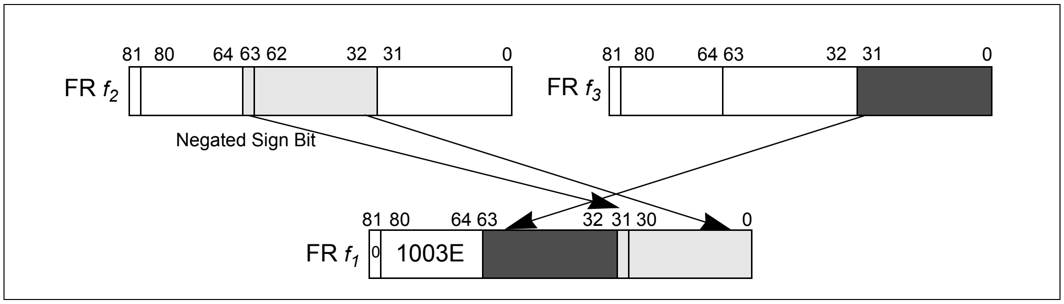


Figure 2-20. Floating-point Swap Negate Right



Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        if (swap_form) {
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = FR[f2].significand{63:32};
        } else if (swap_nl_form) {
            tmp_res_hi = (!FR[f3].significand{31} << 31)
                        | (FR[f3].significand{30:0});
            tmp_res_lo = FR[f2].significand{63:32};
        } else { // swap_nr_form
            tmp_res_hi = FR[f3].significand{31:0};
            tmp_res_lo = (!FR[f2].significand{63} << 31)
                        | (FR[f2].significand{62:32});
        }

        FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault   Disabled Floating-point Register fault

fsxt — Floating-point Sign Extend

Format:	(*qp*) fsxt.l $f_1 = f_2, f_3$	sxt_l_form	F9
	(*qp*) fsxt.r $f_1 = f_2, f_3$	sxt_r_form	F9

Description: For the sxt_l_form (sxt_r_form), the sign of the left (right) single precision value in FR f_2 is extended to 32-bits and is concatenated with the left (right) single precision value in FR f_3.

For all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For all forms, if either FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Figure 2-21. Floating-point Sign Extend Left

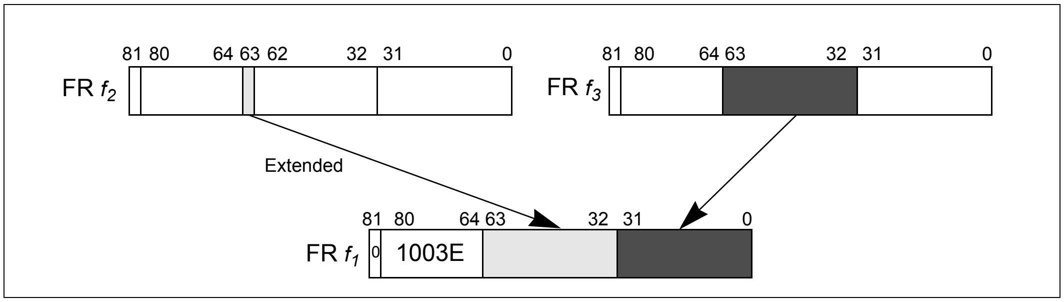
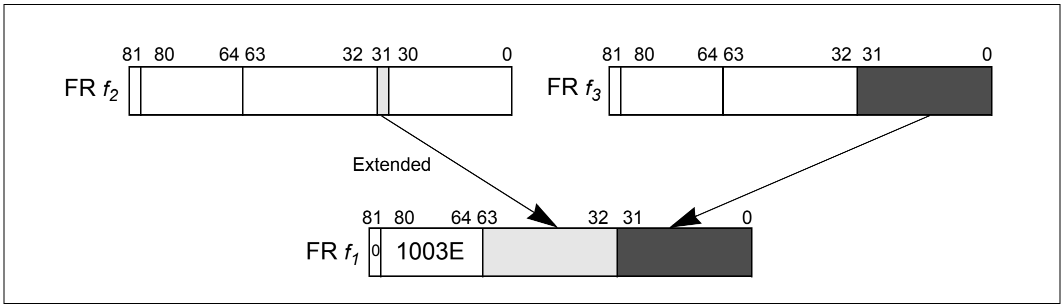


Figure 2-22. Floating-point Sign Extend Right



Operation:

```
if (PR[qp]) {
   fp_check_target_register(f1);
   if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
      disabled_fp_register_fault(tmp_isrcode, 0);

   if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
      FR[f1] = NATVAL;
   } else {
      if (sxt_l_form) {
         tmp_res_hi = (FR[f2].significand{63} ? 0xFFFFFFFF : 0x00000000);
         tmp_res_lo = FR[f3].significand{63:32};
      } else {                                   // sxt_r_form
         tmp_res_hi = (FR[f2].significand{31} ? 0xFFFFFFFF : 0x00000000);
         tmp_res_lo = FR[f3].significand{31:0};
      }

      FR[f1].significand = fp_concatenate(tmp_res_hi, tmp_res_lo);
      FR[f1].exponent = FP_INTEGER_EXP;
      FR[f1].sign = FP_SIGN_POSITIVE;
   }

   fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

fwb — Flush Write Buffers

Format: (*qp*) fwb M24

Description: The processor is instructed to expedite flushing of any pending stores held in write or coalescing buffers. Since this operation is a hint, the processor may or may not take any action and actually flush any outstanding stores. The processor gives no indication when flushing of any prior stores is completed. An `fwb` instruction does not ensure ordering of stores, since later stores may be flushed before prior stores.

To ensure prior coalesced stores are made visible before later stores, software must issue a release operation between stores (see Table 4-15 on page 2:83 for a list of release operations).

This instruction can be used to help ensure stores held in write or coalescing buffers are not delayed for long periods or to expedite high priority stores out of the processors.

Operation:

```
if (PR[qp]) {
    mem_flush_pending_stores();
}
```

Interruptions: None

fxor — Floating-point Exclusive Or

Format: (*qp*) fxor $f_1 = f_2, f_3$ F9

Description: The bit-wise logical exclusive-OR of the significand fields of FR f_2 and FR f_3 is computed. The resulting value is stored in the significand field of FR f_1. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

If either of FR f_2 or FR f_3 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, f3, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[f2]) || fp_is_natval(FR[f3])) {
        FR[f1] = NATVAL;
    } else {
        FR[f1].significand = FR[f2].significand ^ FR[f3].significand;
        FR[f1].exponent = FP_INTEGER_EXP;
        FR[f1].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr(f1);
}
```

FP Exceptions: None

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

getf — Get Floating-point Value or Exponent or Significand

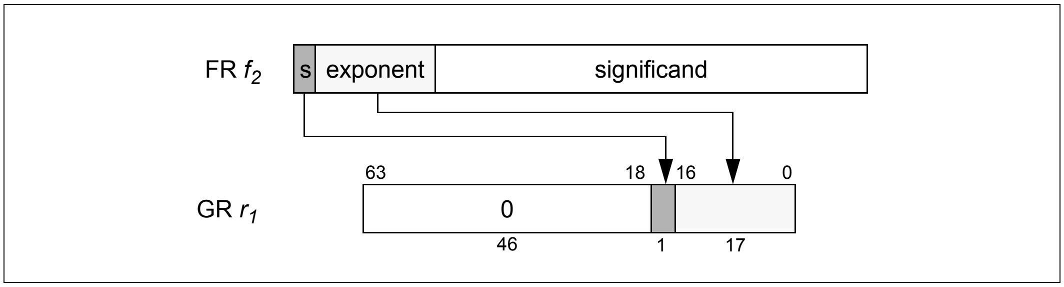
Format:

(qp) getf.s $r_1 = f_2$	single_form	M19
(qp) getf.d $r_1 = f_2$	double_form	M19
(qp) getf.exp $r_1 = f_2$	exponent_form	M19
(qp) getf.sig $r_1 = f_2$	significand_form	M19

Description: In the single and double forms, the value in FR f_2 is converted into a single precision (single_form) or double precision (double_form) memory representation and placed in GR r_1, as shown in Figure 5-7 and Figure 5-8 on page 1:95, respectively. In the single_form, the most-significant 32 bits of GR r_1 are set to 0.

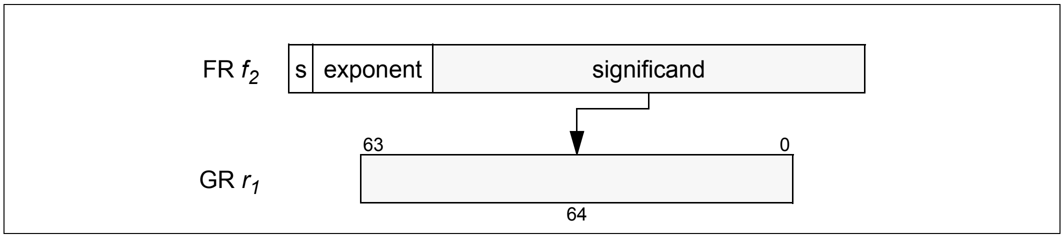
In the exponent_form, the exponent field of FR f_2 is copied to bits 16:0 of GR r_1 and the sign bit of the value in FR f_2 is copied to bit 17 of GR r_1. The most-significant 46-bits of GR r_1 are set to zero.

Figure 2-23. Function of getf.exp



In the significand_form, the significand field of the value in FR f_2 is copied to GR r_1

Figure 2-24. Function of getf.sig



For all forms, if FR f_2 contains a NaTVal, then the NaT bit corresponding to GR r_1 is set to 1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);
    if (tmp_isrcode = fp_reg_disabled(f2, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (single_form) {
        GR[r1]{31:0} = fp_fr_to_mem_format(FR[f2], 4, 0);
        GR[r1]{63:32} = 0;
    } else if (double_form) {
        GR[r1] = fp_fr_to_mem_format(FR[f2], 8, 0);
    } else if (exponent_form) {
        GR[r1]{63:18} = 0;
        GR[r1]{16:0} = FR[f2].exponent;
        GR[r1]{17} = FR[f2].sign;
    } else // significand_form
        GR[r1] = FR[f2].significand;
    if (fp_is_natval(FR[f2]))
        GR[r1].nat = 1;
    else
        GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault Disabled Floating-point Register fault

hint — Performance Hint

Format:

(*qp*) hint imm_{21}	pseudo-op	
(*qp*) hint.i imm_{21}	i_unit_form	I18
(*qp*) hint.b imm_{21}	b_unit_form	B9
(*qp*) hint.m imm_{21}	m_unit_form	M48
(*qp*) hint.f imm_{21}	f_unit_form	F16
(*qp*) hint.x imm_{62}	x_unit_form	X5

Description: Provides a performance hint to the processor about the program being executed. It has no effect on architectural machine state, and operates as a `nop` instruction except for its performance effects.

The immediate, imm_{21} or imm_{62}, specifies the hint. For the x_unit_form, the L slot of the bundle contains the upper 41 bits of imm_{62}.

This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Table 2-31. Hint Immediates

imm_{21} or imm_{62}	Mnemonic	Hint
0x0	@pause	Indicates to the processor that the currently executing stream is waiting, spinning, or performing low priority tasks. This hint can be used by the processor to allocate more resources or time to another executing stream on the same processor. For the case where the currently executing stream is spinning or otherwise waiting for a particular address in memory to change, an advanced load to that address should be done before executing a `hint @pause`; this hint can be used by the processor to resume normal allocation of resources or time to the currently executing stream at the point when some other stream stores to that address.
0x1	@priority	Indicates to the processor that the currently executing stream is performing a high priority task. This hint can be used by the processor to allocate more resources or time to this stream. Implementations will ensure that such increased allocation is only temporary, and that repeated use of this hint will not impair longer-term fairness of allocation.
0x02-0x3f		These values are available for future architected extensions and will execute as a `nop` on all current processors. Use of these values may cause unexpected performance issues on future processors and should not be used.
other		Implementation specific. Performs an implementation-specific hint action. Consult processor model-specific documentation for details.

Operation:

```
if (PR[qp]) {
    if (x_unit_form)
        hint = imm62;
    else // i_unit_form || b_unit_form || b_unit_form || f_unit_form
        hint = imm21;

    if (is_supported_hint(hint))
        execute_hint(hint);
}
```

Interruptions: None

invala — Invalidate ALAT

Format:

(*qp*) invala	complete_form	M24
(*qp*) invala.e r_1	gr_form, entry_form	M26
(*qp*) invala.e f_1	fr_form, entry_form	M27

Description: The selected entry or entries in the ALAT are invalidated.

In the complete_form, all ALAT entries are invalidated. In the entry_form, the ALAT is queried using the general register specifier r_1 (gr_form), or the floating-point register specifier f_1 (fr_form), and if any ALAT entry matches, it is invalidated.

Operation:

```
if (PR[qp]) {
    if (complete_form)
        alat_inval();
    else { // entry_form
        if (gr_form)
            alat_inval_single_entry(GENERAL, $r_1$);
        else // fr_form
            alat_inval_single_entry(FLOAT, $f_1$);
    }
}
```

Interruptions: None

itc — Insert Translation Cache

Format:	(qp) itc.i r_2	instruction_form	M41
	(qp) itc.d r_2	data_form	M41

Description: An entry is inserted into the instruction or data translation cache. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}. The processor determines which entry to replace based on an implementation-specific replacement algorithm.

The visibility of the `itc` instruction to externally generated purges (`ptc.g`, `ptc.ga`) must occur before subsequent memory operations. From a software perspective, this is similar to acquire semantics. Serialization is still required to observe the side-effects of a translation being present.

`itc` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The TLB is first purged of any overlapping entries as specified by Table 4-1 on page 2:52.

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

To ensure forward progress, software must ensure that PSR.ic remains 0 until `rfi`-ing to the instruction that requires the translation.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r2].nat)
        register_nat_consumption_fault(0);

    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!impl_check_mov_ifa() &&
            unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(ITC_TYPE);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    } else {                                          // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(DTC_TYPE);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    }
}
```

Interruptions:

Machine Check abort
Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault
Virtualization fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

itr — Insert Translation Register

Format: (*qp*) itr.i itr[r_3] = r_2 instruction_form M42
(*qp*) itr.d dtr[r_3] = r_2 data_form M42

Description: A translation is inserted into the instruction or data translation register specified by the contents of GR r_3. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}.

As described in Table 4-1, “Purge Behavior of TLB Inserts and Purges” on page 2:52, the TLB is first purged of any entries that overlap with the newly inserted translation. The translation previously contained in the TR slot specified by GR r_3 is not necessarily purged from the processor's TLBs and may remain as a TC entry. To ensure that the previous TR translation is purged, software must use explicit `ptr` instructions before inserting the new TR entry.

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

Operation:

```
if (PR[qp]) {
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);

    slot = GR[r3]{7:0};
    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    tmp_tr_type = instruction_form ? ITR_TYPE : DTR_TYPE;

    if (is_reserved_reg(tmp_tr_type, slot))
        reserved_register_field_fault();
    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!impl_check_mov_ifa() &&
            unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
    } else {                                        // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
    }
}
```

Interruptions: Machine Check abort
Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault
Virtualization fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

Notes: The processor may use invalid translation registers for translation cache entries. Performance can be improved on some processor models by ensuring translation registers are allocated beginning at translation register zero and continuing contiguously upwards.

ld — Load

Format:

(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3]	no_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], r_2	reg_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], imm_9	imm_base_update_form	M3
(*qp*) ld16.*ldhint* r_1, ar.csd = [r_3]	sixteen_byte_form, no_base_update_form	M2
(*qp*) ld16.acq.*ldhint* r_1, ar.csd = [r_3]	sixteen_byte_form, acquire_form, no_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3]	fill_form, no_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], r_2	fill_form, reg_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], imm_9	fill_form, imm_base_update_form	M3

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then zero extended and placed in GR r_1. The values of the *sz* completer are given in Table 2-32. The NaT bit corresponding to GR r_1 is cleared, except as described below for speculative loads. The *ldtype* completer specifies special load operations, which are described in Table 2-33.

For the sixteen_byte_form, two 8-byte values are loaded as a single, 16-byte memory read. The value at the lowest address is placed in GR r_1, and the value at the highest address is placed in the Compare and Store Data application register (AR[CSD]). The only load types supported for this sixteen_byte_form are *none* and *acq*.

For the fill_form, an 8-byte value is loaded, and a bit in the UNAT application register is copied into the target register NaT bit. This instruction is used for reloading a spilled register/NaT pair. See Section 4.4.4, “Control Speculation” on page 1:60 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised. Base register update is not supported for the `ld16` instruction.

Table 2-32. *sz* Completers

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

Table 2-33. Load Types

ldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes the target register’s NaT bit to be set. The NaT bit is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, the target register and NaT bit is cleared, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.

Table 2-33. Load Types (Continued)

ldtype Completer	Interpretation	Special Load Operation
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes the target register's NaT bit to be set, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load – no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.
c.clr.acq	Ordered check load – clear	This type behaves the same as the unordered clear form, except that the ALAT lookup (and resulting load, if no ALAT entry is found) is performed with acquire semantics.
acq	Ordered load	An ordered load is performed with acquire semantics.
bias	Biased load	A hint is provided to the implementation to acquire exclusive ownership of the accessed cache line.

For more details on ordered, biased, speculative, advanced and check loads see Section 4.4.4, "Control Speculation" on page 1:60 and Section 4.4.5, "Data Speculation" on page 1:63. For more details on ordered loads see Section 4.4.7, "Memory Access Ordering" on page 1:73. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details on biased loads. Details on memory attributes are described in Section 4.4, "Memory Attributes" on page 2:75.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details.

Table 2-34. Load Hints

ldhint Completer	Interpretation
none	Temporal locality, level 1

Table 2-34. Load Hints (Continued)

ldhint Completer	Interpretation
nt1	No temporal locality, level 1
nta	No temporal locality, all levels

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

For the base update forms, specifying the same register address in r_1 and r_3 will cause an Illegal Operation fault.

Hardware support for `ld16` instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ld16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen_byte_form, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See Section 3.1.11, “Processor Identification Registers” on page 1:34 for details.

Operation:

```
if (PR[qp]) {
    size = fill_form ? 8 : (sixteen_byte_form ? 16 : sz);

    speculative = (ldtype == 's' || ldtype == 'sa');
    advanced = (ldtype == 'a' || ldtype == 'sa');
    check_clear = (ldtype == 'c.clr' || ldtype == 'c.clr.acq');
    check_no_clear = (ldtype == 'c.nc');
    check = check_clear || check_no_clear;
    acquire = (acquire_form || ldtype == 'acq' || ldtype == 'c.clr.acq');
    otype = acquire ? ACQUIRE : UNORDERED;
    bias = (ldtype == 'bias') ? BIAS : 0 ;
    translate_address = 1;
    read_memory = 1;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced) itype |= ADVANCE ;
    if (size == 16) itype |= UNCACHE_OPT ;

    if (sixteen_byte_form && !instruction_implemented(LD16))
        illegal_operation_fault();
    if ((reg_base_update_form || imm_base_update_form) && ($r_1$ == $r_3$))
        illegal_operation_fault();
    check_target_register($r_1$);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register($r_3$);

    if (reg_base_update_form) {
        tmp_r2 = GR[$r_2$];
        tmp_r2nat = GR[$r_2$].nat;
    }

    if (!speculative && GR[$r_3$].nat)                // fault on NaT address
        register_nat_consumption_fault(itype);
    defer = speculative && (GR[$r_3$].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(GENERAL, $r_1$)) {
        translate_address = alat_translate_address_on_hit(ldtype, GENERAL,
$r_1$);
        read_memory = alat_read_memory_on_hit(ldtype, GENERAL, $r_1$);
    }
    if (!translate_address) {
        if (check_clear || advanced)                // remove any old alat entry
            alat_inval_single_entry(GENERAL, $r_1$);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[$r_3$], size, itype, PSR.cpl, &mattr,
                                  &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, otype,
                                  bias | ldhint, &defer);
            if (!defer && read_memory) {
                if (size == 16) {
                    mem_read_pair(&val, &val_ar, paddr, size, UM.be, mattr,
                                  otype, ldhint);
                }
                else {
                    val = mem_read(paddr, size, UM.be, mattr, otype,
                                   bias | ldhint);
                }
            }
        }
        if (check_clear || advanced)                    // remove any old ALAT entry
            alat_inval_single_entry(GENERAL, r1);
        if (defer) {
            if (speculative) {
                GR[r1] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                      bias | ldhint);
                GR[r1].nat = 1;
            } else {
                GR[r1] = 0;                             // ld.a to sequential memory
                GR[r1].nat = 0;
            }
        } else {                                        // execute load normally
            if (fill_form) {                            // fill NaT on ld8.fill
                bit_pos = GR[r3]{8:3};
                GR[r1] = val;
                GR[r1].nat = AR[UNAT]{bit_pos};
            } else {                                    // clear NaT on other types
                if (size == 16) {
                    GR[r1] = val;
                    AR[CSD] = val_ar;
                }
                else {
                    GR[r1] = zero_ext(val, size * 8);
                }
                GR[r1].nat = 0;
            }
            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                        // add entry to ALAT
                alat_write(ldtype, GENERAL, r1, paddr, size);
        }
    }

    if (imm_base_update_form) {                         // update base register
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + tmp_r2;
        GR[r3].nat = GR[r3].nat || tmp_r2nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], ldhint | bias, itype);
}
```

Interruptions: Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

ldf — Floating-point Load

Format:

(qp) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3]	no_base_update_form	M9
(qp) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3], r_2	reg_base_update_form	M7
(qp) ldf*fsz*.*fldtype*.*ldhint* f_1 = [r_3], imm_9	imm_base_update_form	M8
(qp) ldf8.*fldtype*.*ldhint* f_1 = [r_3]	integer_form, no_base_update_form	M9
(qp) ldf8.*fldtype*.*ldhint* f_1 = [r_3], r_2	integer_form, reg_base_update_form	M7
(qp) ldf8.*fldtype*.*ldhint* f_1 = [r_3], imm_9	integer_form, imm_base_update_form	M8
(qp) ldf.fill.*ldhint* f_1 = [r_3]	fill_form, no_base_update_form	M9
(qp) ldf.fill.*ldhint* f_1 = [r_3], r_2	fill_form, reg_base_update_form	M7
(qp) ldf.fill.*ldhint* f_1 = [r_3], imm_9	fill_form, imm_base_update_form	M8

Description: A value consisting of *fsz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then converted into the floating-point register format and placed in FR f_1. See Section 5.1, “Data Types and Formats” on page 1:85 for details on conversion to floating-point register format. The values of the *fsz* completer are given in Table 2-35. The *fldtype* completer specifies special load operations, which are described in Table 2-36.

For the integer_form, an 8-byte value is loaded and placed in the significand field of FR f_1 without conversion. The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

For the fill_form, a 16-byte value is loaded, and the appropriate fields are placed in FR f_1 without conversion. This instruction is used for reloading a spilled register. See Section 4.4.4, “Control Speculation” on page 1:60 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised.

Table 2-35. *fsz* Completers

fsz Completer	Bytes Accessed	Memory Format
s	4 bytes	Single precision
d	8 bytes	Double precision
e	10 bytes	Extended precision

Table 2-36. FP Load Types

fldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes NaTVal to be placed in the target register. The NaTVal value is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, no ALAT entry is added to the ALAT and the target register is set as follows: for the integer_form, the exponent is set to 0x1003E and the sign and significand are set to zero; for all other forms, the sign, exponent and significand are set to zero. The absence of an ALAT entry is later used to detect deferral or collision.

Table 2-36. FP Load Types (Continued)

fldtype Completer	Interpretation	Special Load Operation
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes NaTVal to be placed in the target register, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load – no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.

For more details on speculative, advanced and check loads see Section 4.4.4, “Control Speculation” on page 1:60 and Section 4.4.5, “Data Speculation” on page 1:63. Details on memory attributes are described in Section 4.4, “Memory Attributes” on page 2:75.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-34 on page 3:152. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69 for details.

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1.

Hardware support for `ldfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ldfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted. The fault is delivered only on the normal, advanced, and check load flavors. Control-speculative flavors of `ldfe` always defer the Unsupported Data Reference fault.

Operation:

```
if (PR[qp]) {
    size = (fill_form ? 16 : (integer_form ? 8 : fsz));
    speculative = (fldtype == ‘s’ || fldtype == ‘sa’);
    advanced = (fldtype == ‘a’ || fldtype == ‘sa’);
    check_clear = (fldtype == ‘c.clr’ );
    check_no_clear = (fldtype == ‘c.nc’);
    check = check_clear || check_no_clear;
    translate_address = 1;
    read_memory = 1;

    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;
    if (size == 10) itype |= UNCACHE_OPT;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)               // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {
        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
    }

    if (!translate_address) {
        if (check_clear || advanced)          // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                    &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, UNORDERED,
                                    ldhint, &defer);
            if (!defer && read_memory)
                val = mem_read(paddr, size, UM.be, mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)              // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                  // execute load normally
            FR[f1] = fp_mem_to_fr_format(val, size, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                  // add entry to ALAT
                alat_write(fldtype, FLOAT, f1, paddr, size);
        }
        }

        if (imm_base_update_form) {                    // update base register
            GR[r3] = GR[r3] + sign_ext(imm9, 9);
            GR[r3].nat = GR[r3].nat;
        } else if (reg_base_update_form) {
            GR[r3] = GR[r3] + GR[r2];
            GR[r3].nat = GR[r3].nat || GR[r2].nat;
        }

        if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
            mem_implicit_prefetch(GR[r3], ldhint, itype);

        fp_update_psr(f1);
    }
```

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

ldfp — Floating-point Load Pair

Format:

(qp) ldfps.*fldtype*.*ldhint* f_1, f_2 = [r_3]	single_form, no_base_update_form	M11
(qp) ldfps.*fldtype*.*ldhint* f_1, f_2 = [r_3], 8	single_form, base_update_form	M12
(qp) ldfpd.*fldtype*.*ldhint* f_1, f_2 = [r_3]	double_form, no_base_update_form	M11
(qp) ldfpd.*fldtype*.*ldhint* f_1, f_2 = [r_3], 16	double_form, base_update_form	M12
(qp) ldfp8.*fldtype*.*ldhint* f_1, f_2 = [r_3]	integer_form, no_base_update_form	M11
(qp) ldfp8.*fldtype*.*ldhint* f_1, f_2 = [r_3], 16	integer_form, base_update_form	M12

Description: Eight (single_form) or sixteen (double_form/integer_form) bytes are read from memory starting at the address specified by the value in GR r_3. The value read is treated as a contiguous pair of floating-point numbers for the single_form/double_form and as integer/Parallel FP data for the integer_form. Each number is converted into the floating-point register format. The value at the lowest address is placed in FR f_1, and the value at the highest address is placed in FR f_2. See Section 5.1, "Data Types and Formats" on page 1:85 for details on conversion to floating-point register format. The *fldtype* completer specifies special load operations, which are described in Table 2-36 on page 3:157.

For more details on speculative, advanced and check loads see Section 4.4.4, "Control Speculation" on page 1:60 and Section 4.4.5, "Data Speculation" on page 1:63.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred.

In the base_update_form, the value in GR r_3 is added to an implied immediate value (equal to double the data size) and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address.

The value of the *ldhint* modifier specifies the locality of the memory access. The mnemonic values of *ldhint* are given in Table 2-34 on page 3:152. A prefetch hint is implied in the base update form. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details.

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

The PSR.mfl and PSR.mfh bits are updated to reflect the modification of FR f_1 and FR f_2.

There is a restriction on the choice of target registers. Register specifiers f_1 and f_2 must specify one odd-numbered physical FR and one even-numbered physical FR. Specifying two odd or two even registers will cause an Illegal Operation fault to be raised. The restriction is on physical register numbers after register rotation. This means that if f_1 and f_2 both specify static registers or both specify rotating registers, then f_1 and f_2 must be odd/even or even/odd. If f_1 and f_2 specify one static and one rotating register, the restriction depends on CFM.rrb.fr. If CFM.rrb.fr is even, the restriction is the same; f_1 and f_2 must be odd/even or even/odd. If CFM.rrb.fr is odd, then f_1 and f_2 must be even/even or odd/odd. Specifying one static and one rotating register should only be done when CFM.rrb.fr will have a predictable value (such as 0).

Operation:

```
if (PR[qp]) {
    size = single_form ? 8 : 16;

    speculative = (fldtype == ‘s’ || fldtype == ‘sa’);
    advanced = (fldtype == ‘a’ || fldtype == ‘sa’);
    check_clear = (fldtype == ‘c.clr’);
    check_no_clear = (fldtype == ‘c.nc’);
    check = check_clear || check_no_clear;
    translate_address = 1;
    read_memory = 1;

    itype = READ;
    if (speculative) itype |= SPEC;
    if (advanced) itype |= ADVANCE;

    if (fp_reg_bank_conflict(f1, f2))
        illegal_operation_fault();

    if (base_update_form)
        check_target_register(r3);

    fp_check_target_register(f1);
    fp_check_target_register(f2);
    if (tmp_isrcode = fp_reg_disabled(f1, f2, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, itype);

    if (!speculative && GR[r3].nat)                 // fault on NaT address
        register_nat_consumption_fault(itype);

    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(FLOAT, f1)) {
        translate_address = alat_translate_address_on_hit(fldtype, FLOAT, f1);
        read_memory = alat_read_memory_on_hit(fldtype, FLOAT, f1);
    }

    if (!translate_address) {
        if (check_clear || advanced)            // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                     &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, UNORDERED,
                                     ldhint, &defer);
            if (!defer && read_memory)
                mem_read_pair(&f1_val, &f2_val, paddr, size, UM.be,
                                     mattr, UNORDERED, ldhint);
        }
        if (check_clear || advanced)                // remove any old ALAT entry
            alat_inval_single_entry(FLOAT, f1);
        if (speculative && defer) {
            FR[f1] = NATVAL;
            FR[f2] = NATVAL;
        } else if (advanced && !speculative && defer) {
            FR[f1] = (integer_form ? FP_INT_ZERO : FP_ZERO);
            FR[$f_2$] = (integer_form ? FP_INT_ZERO : FP_ZERO);
        } else {                                 // execute load normally
            FR[$f_1$] = fp_mem_to_fr_format(f1_val, size/2, integer_form);
            FR[$f_2$] = fp_mem_to_fr_format(f2_val, size/2, integer_form);

            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                 // add entry to ALAT
                alat_write(fldtype, FLOAT, $f_1$, paddr, size);
        }
    }

    if (base_update_form) {                      // update base register
        GR[$r_3$] = GR[$r_3$] + size;
        GR[$r_3$].nat = GR[$r_3$].nat;
        if (!GR[$r_3$].nat)
            mem_implicit_prefetch(GR[$r_3$], ldhint, itype);
    }

    fp_update_psr($f_1$);
    fp_update_psr($f_2$);
}
```

Interruptions: Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault

lfetch — Line Prefetch

Format:

(*qp*) lfetch.*lftype*.*lfhint* [r_3]	no_base_update_form	M18
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M20
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M22
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M18
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M20
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M22

Description: The line containing the address specified by the value in GR r_3 is moved to the highest level of the data memory hierarchy. The value of the *lfhint* modifier specifies the locality of the memory access; see Section 4.4, “Memory Access Instructions” on page 1:57 for details. The mnemonic values of *lfhint* are given in Table 2-38.

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. See Chapter 4, “Addressing and Protection” in Volume 2. Line size is implementation dependent but must be a power of two greater than or equal to 32 bytes. In the exclusive form, the cache line is allowed to be marked in an exclusive state. This qualifier is used when the program expects soon to modify a location in that line. If the memory attribute for the page containing the line is not cacheable, then no reference is made.

The completer, *lftype*, specifies whether or not the instruction raises faults normally associated with a regular load. Table 2-37 defines these two options.

Table 2-37. *lftype* Mnemonic Values

lftype Mnemonic	Interpretation
none	No faults are raised
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r_3 is incremented by either the sign-extended value in imm_9 (in the imm_base_update_form) or the value in GR r_2 (in the reg_base_update_form). In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set – no fault is raised.

In the reg_base_update_form and the imm_base_update_form, if the NaT bit corresponding to GR r_3 is clear, then the address specified by the value in GR r_3 after the post-increment acts as a hint to implicitly prefetch the indicated cache line. This implicit prefetch uses the locality hints specified by *lfhint*. The implicit prefetch does not affect program functionality, does not raise any faults, and may be ignored by the implementation.

In the no_base_update_form, the value in GR r_3 is not modified and no implicit prefetch hint is implied.

If the NaT bit corresponding to GR r_3 is set then the state of memory is not affected. In the reg_base_update_form and imm_base_update_form, the post increment of GR r_3 is performed and prefetch is hinted as described above.

`lfetch` instructions, like hardware prefetches, are not orderable operations, i.e., they have no order with respect to prior or subsequent memory operations.

Table 2-38. *lfhint* Mnemonic Values

lfhint Mnemonic	Interpretation
none	Temporal locality, level 1
nt1	No temporal locality, level 1
nt2	No temporal locality, level 2
nta	No temporal locality, all levels

A faulting `lfetch` to an unimplemented address results in an Unimplemented Data Address fault. A non-faulting `lfetch` to an unimplemented address does not take the fault and will not issue a prefetch request, but, if specified, will perform a register post-increment.

Both the non-faulting and the faulting forms of `lfetch` can be used speculatively. The purpose of raising faults on the faulting form is to allow the operating system to resolve problems with the address to the extent that it can do so relatively quickly. If problems with the address cannot be resolved quickly, the OS simply returns to the program, and forces the data prefetch to be skipped over.

Specifically, if a faulting `lfetch` takes any of the listed faults (other than Illegal Operation fault), the operating system must handle this fault to the extent that it can do so relatively quickly and invisibly to the interrupted program. If the fault cannot be handled quickly or cannot be handled invisibly (e.g., if handling the fault would involve terminating the program), the OS must return to the interrupted program, skipping over the data prefetch. This can easily be done by setting the IPSR.ed bit to 1 before executing an `rfi` to go back to the process, which will allow the `lfetch.fault` to perform its base register post-increment (if specified), but will suppress any prefetch request and hence any prefetch-related fault. Note that the OS can easily identify that a faulting `lfetch` was the cause of the fault by observing that ISR.na is 1, and ISR.code{3:0} is 4. The one exception to this is the Illegal Operation fault, which can be caused by an `lfetch.fault` if base register post-increment is specified, and the base register is outside of the current stack frame, or is GR0. Since this one fault is not related to the prefetch aspect of `lfetch.fault`, but rather to the base update portion, Illegal Operation faults on `lfetch.fault` should be handled the same as for any other instruction.

Operation:

```
if (PR[qp]) {
    itype = READ|NON_ACCESS;
    itype |= (lftype == ‘fault’) ? LFETCH_FAULT : LFETCH;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register($r_3$);

    if (lftype == ‘fault’) {            // faulting form
        if (GR[$r_3$].nat && !PSR.ed)      // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    excl_hint = (exclusive_form) ? EXCLUSIVE : 0;

    if (!GR[$r_3$].nat && !PSR.ed) {// faulting form already faulted if $r_3$ is nat
        paddr = tlb_translate(GR[$r_3$], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (imm_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
        GR[$r_3$].nat = GR[$r_3$].nat;
    } else if (reg_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + GR[$r_2$];
        GR[$r_3$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[$r_3$].nat)
        mem_implicit_prefetch(GR[$r_3$], lfhint | excl_hint, itype);
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault

loadrs — Load Register Stack

Format: loadrs M25

Description: This instruction ensures that a specified number of bytes (registers values and/or NaT collections) below the current BSP have been loaded from the backing store into the stacked general registers. The loaded registers are placed into the dirty partition of the register stack. All other stacked general registers are marked as invalid, without being saved to the backing store.

The number of bytes to be loaded is specified in a sub-field of the RSC application register (RSC.loadrs). Backing store addresses are always 8-byte aligned, and therefore the low order 3 bits of the `loadrs` field (RSC.loadrs{2:0}) are ignored. This instruction can be used to invalidate all stacked registers outside the current frame, by setting RSC.loadrs to zero.

This instruction will fault with an Illegal Operation fault under any of the following conditions:

- the RSE is not in enforced lazy mode (RSC.mode is non-zero).
- CFM.sof and RSC.loadrs are both non-zero.
- an attempt is made to load up more registers than are available in the physical stacked register file.

This instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0; otherwise, the results are undefined. This instruction cannot be predicated.

Operation:

```
if (AR[RSC].mode != 0)
    illegal_operation_fault();

if ((CFM.sof != 0) && (AR[RSC].loadrs != 0))
    illegal_operation_fault();

rse_ensure_regs_loaded(AR[RSC].loadrs);   // can raise faults listed below
AR[RNAT] = undefined();
```

Interruptions:

Illegal Operation fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Access Bit fault
Data Debug fault

mf — Memory Fence

Format:

(*qp*)	mf	ordering_form	M24
(*qp*)	mf.a	acceptance_form	M24

Description: This instruction forces ordering between prior and subsequent memory accesses. The ordering_form ensures all prior data memory accesses are made visible prior to any subsequent data memory accesses being made visible. It does not ensure prior data memory references have been accepted by the external platform, nor that prior data memory references are visible.

The acceptance_form prevents any subsequent data memory accesses by the processor from initiating transactions to the external platform until:

- all prior loads to sequential pages have returned data, and
- all prior stores to sequential pages have been accepted by the external platform.

The definition of “acceptance” is platform dependent. The acceptance_form is typically used to ensure the processor has “waited” until a memory-mapped I/O transaction has been “accepted” before initiating additional external transactions. The acceptance_form does not ensure ordering, or acceptance to memory areas other than sequential pages.

Operation:

```
if (PR[qp]){
    if (acceptance_form)
        acceptance_fence();
    else // ordering_form
        ordering_fence();
}
```

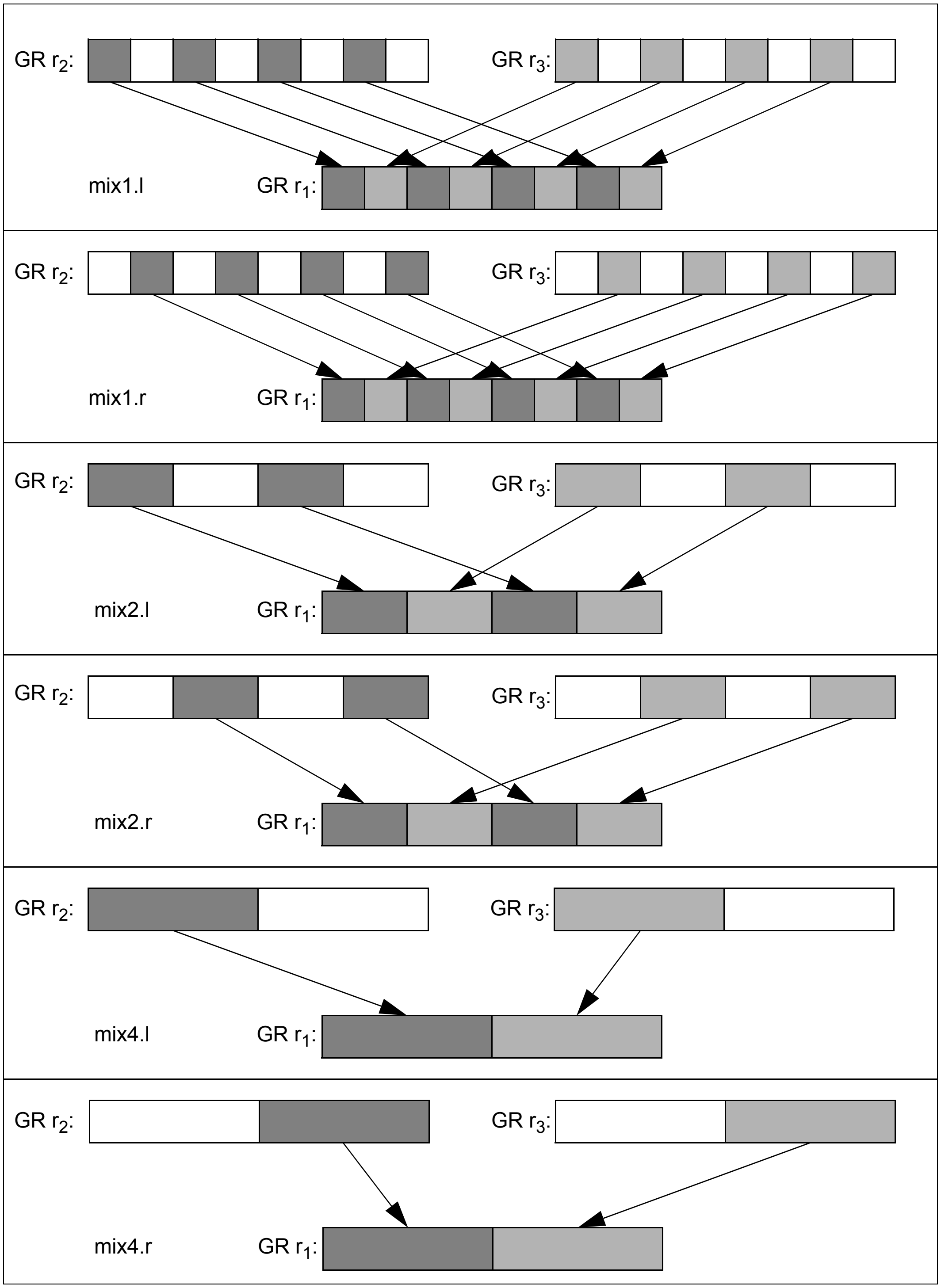
Interruptions: None

mix — Mix

Format:	(*qp*) mix1.l $r_1 = r_2, r_3$	one_byte_form, left_form	I2
	(*qp*) mix2.l $r_1 = r_2, r_3$	two_byte_form, left_form	I2
	(*qp*) mix4.l $r_1 = r_2, r_3$	four_byte_form, left_form	I2
	(*qp*) mix1.r $r_1 = r_2, r_3$	one_byte_form, right_form	I2
	(*qp*) mix2.r $r_1 = r_2, r_3$	two_byte_form, right_form	I2
	(*qp*) mix4.r $r_1 = r_2, r_3$	four_byte_form, right_form	I2

Description: The data elements of GR r_2 and r_3 are mixed as shown in Figure 2-25, and the result placed in GR r_1. The data elements in the source registers are grouped in pairs, and one element from each pair is selected for the result. In the left_form, the result is formed from the leftmost elements from each of the pairs. In the right_form, the result is formed from the rightmost elements. Elements are selected alternately from the two source registers.

Figure 2-25. Mix Examples



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

        if (left_form)
            GR[r1] = concatenate8(x[7], y[7], x[5], y[5],
                                  x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate8(x[6], y[6], x[4], y[4],
                                  x[2], y[2], x[0], y[0]);

    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

        if (left_form)
            GR[r1] = concatenate4(x[3], y[3], x[1], y[1]);
        else // right_form
            GR[r1] = concatenate4(x[2], y[2], x[0], y[0]);

    } else {                                          // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        if (left_form)
            GR[r1] = concatenate2(x[1], y[1]);
        else // right_form
            GR[r1] = concatenate2(x[0], y[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

mov — Move Application Register

Format:

(*qp*) mov $r_1 = ar_3$	pseudo-op	
(*qp*) mov $ar_3 = r_2$	pseudo-op	
(*qp*) mov $ar_3 = imm_8$	pseudo-op	
(*qp*) mov.i $r_1 = ar_3$	i_form, from_form	I28
(*qp*) mov.i $ar_3 = r_2$	i_form, register_form, to_form	I26
(*qp*) mov.i $ar_3 = imm_8$	i_form, immediate_form, to_form	I27
(*qp*) mov.m $r_1 = ar_3$	m_form, from_form	M31
(*qp*) mov.m $ar_3 = r_2$	m_form, register_form, to_form	M29
(*qp*) mov.m $ar_3 = imm_8$	m_form, immediate_form, to_form	M30

Description: The source operand is copied to the destination register.

In the from_form, the application register specified by ar_3 is copied into GR r_1 and the corresponding NaT bit is cleared.

In the to_form, the value in GR r_2 (in the register_form), or the sign-extended value in imm_8 (in the immediate_form), is placed in AR ar_3. In the register_form if the NaT bit corresponding to GR r_2 is set, then a Register NaT Consumption fault is raised.

Only a subset of the application registers can be accessed by each execution unit (M or I). Table 3-3 on page 1:28 indicates which application registers may be accessed from which execution unit type. An access to an application register from the wrong unit type causes an Illegal Operation fault.

This instruction has multiple forms with the pseudo operation eliminating the need for specifying the execution unit. Accesses of the ARs are always implicitly serialized. While implicitly serialized, read-after-write and write-after-write dependency violations must be avoided (e.g., setting CCV, followed by `cmpxchg` in the same instruction group, or simultaneous writes to the UNAT register by `ld.fill` and mov to UNAT).

Operation:

```
if (PR[qp]) {
    tmp_type = (i_form ? AR_I_TYPE : AR_M_TYPE);
    if (is_reserved_reg(tmp_type, ar3))
        illegal_operation_fault();

    if (from_form) {
        check_target_register(r1);
        if (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0))
            illegal_operation_fault();

        if ((ar3 == ITC || ar3 == RUC) && PSR.si && PSR.cpl != 0)
            privileged_register_fault();

        if ((ar3 == ITC || ar3 == RUC) && PSR.si && PSR.vm == 1)
            virtualization_fault();

        GR[r1] = (is_ignored_reg(ar3)) ? 0 : AR[ar3];
        GR[r1].nat = 0;
    } else {                                          // to_form
        tmp_val = (register_form) ? GR[r2] : sign_ext(imm8, 8);

        if (is_read_only_reg(AR_TYPE, ar3) ||
            (((ar3 == BSPSTORE) || (ar3 == RNAT)) && (AR[RSC].mode != 0)))
            illegal_operation_fault();

        if (register_form && GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(AR_TYPE, ar3, tmp_val))
            reserved_register_field_fault();

        if ((is_kernel_reg(ar3) || ar3 == ITC || ar3 == RUC) && (PSR.cpl != 0))
            privileged_register_fault();

        if ((ar3 == ITC || ar3 == RUC) && PSR.vm == 1)
            virtualization_fault();

        if (!is_ignored_reg(ar3)) {
            tmp_val = ignored_field_mask(AR_TYPE, ar3, tmp_val);
            // check for illegal promotion
            if (ar3 == RSC && tmp_val{3:2} u< PSR.cpl)
                tmp_val{3:2} = PSR.cpl;
            AR[ar3] = tmp_val;

            if (ar3 == BSPSTORE) {
                AR[BSP] = rse_update_internal_stack_pointers(tmp_val);
                AR[RNAT] = undefined();
            }
        }
    }
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Privileged Register fault
Virtualization fault

mov — Move Branch Register

Format:

(*qp*) mov $r_1 = b_2$	from_form	I22
(*qp*) mov $b_1 = r_2$	pseudo-op	
(*qp*) mov.*mwh*.*ih* $b_1 = r_2, tag_{13}$	to_form	I21
(*qp*) mov.ret.*mwh*.*ih* $b_1 = r_2, tag_{13}$	return_form, to_form	I21

Description: The source operand is copied to the destination register.

In the from_form, the branch register specified by b_2 is copied into GR r_1. The NaT bit corresponding to GR r_1 is cleared.

In the to_form, the value in GR r_2 is copied into BR b_1. If the NaT bit corresponding to GR r_2 is 1, then a Register NaT Consumption fault is taken.

A set of hints can also be provided when moving to a branch register. These hints are very similar to those provided on the `brp` instruction, and provide prediction information about a future branch which may use the value being moved into BR b_1. The return_form is used to provide the hint that this value will be used in a return-type branch.

The values for the *mwh* whether hint completer are given in Table 2-39. For a description of the *ih* hint completer see the Branch Prediction instruction and Table 2-13 on page 3:32.

Table 2-39. Move to BR Whether Hints

mwh Completer	Move to BR Whether Hint
none	Ignore all hints
sptk	Static Taken
dptk	Dynamic

A pseudo-op is provided for copying a general register into a branch register when there is no hint information to be specified. This is encoded with a value of 0 for tag_{13} and values corresponding to *none* for the hint completers.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register($r_1$);
        GR[$r_1$] = BR[$b_2$];
        GR[$r_1$].nat = 0;
    } else { // to_form
        tmp_tag = IP + sign_ext(($timm_9$ << 4), 13);
        if (GR[$r_2$].nat)
            register_nat_consumption_fault(0);
        BR[$b_1$] = GR[$r_2$];
        branch_predict(mwh, ih, return_form, GR[$r_2$], tmp_tag);
    }
}
```

Interruptions: Illegal Operation fault Register NaT Consumption fault

mov — Move Control Register

Format: (*qp*) mov r_1 = cr_3 from_form M33
(*qp*) mov cr_3 = r_2 to_form M32

Description: The source operand is copied to the destination register.

For the from_form, the control register specified by cr_3 is read and the value copied into GR r_1.

For the to_form, GR r_2 is read and the value copied into CR cr_3.

Control registers can only be accessed at the most privileged level, and when PSR.vm is 0. Reading or writing an interruption control register (CR16-CR27), when the PSR.ic bit is one, will result in an Illegal Operation fault.

Operation:

```
if (PR[qp]) {
    if (is_reserved_reg(CR_TYPE, cr3)
        || to_form && is_read_only_reg(CR_TYPE, cr3)
        || PSR.ic && is_interruption_cr(cr3))
    {
        illegal_operation_fault();
    }

    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == IVR)
            check_interrupt_request();

        if (cr3 == ITIR)
            GR[r1] = impl_itir_cwi_mask(CR[ITIR]);
        else
            GR[r1] = CR[cr3];

        GR[r1].nat = 0;
    } else {            // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(CR_TYPE, cr3, GR[r2]))
            reserved_register_field_fault();
        if ((cr3 == IFA) && impl_check_mov_ifa() &&
            unimplemented_virtual_address(GR[r2], PSR.vm))
            unimplemented_data_address_fault(0);
        if (PSR.vm == 1)
            virtualization_fault();
        if (cr3 == EOI)
            end_of_interrupt();

        tmp_val = ignored_field_mask(CR_TYPE, cr3, GR[r2]);
        CR[cr3] = tmp_val;
        if (cr3 == IIPA)
```

```
          last_IP = tmp_val;
    }
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault
Virtualization fault

Serialization: Reads of control registers reflect the results of all prior instruction groups and interruptions.

In general, writes to control registers do not immediately affect subsequent instructions. Software must issue a serialize operation before a dependent instruction uses a modified resource.

Control register writes are not implicitly synchronized with a corresponding control register read and requires data serialization.

mov — Move Floating-point Register

Format: (*qp*) mov $f_1 = f_3$ pseudo-op of: (*qp*) fmerge.s $f_1 = f_3, f_3$

Description: The value of FR f_3 is copied to FR f_1.

Operation: `See “fmerge — Floating-point Merge” on page 3:80.`

mov — Move General Register

Format: (*qp*) mov $r_1 = r_3$ pseudo-op of: (*qp*) adds $r_1 = 0, r_3$

Description: The value of GR r_3 is copied to GR r_1.

Operation: `See “add — Add” on page 3:14.`

mov — Move Immediate

Format: (qp) mov r_1 = imm_{22} pseudo-op of: (qp) addl r_1 = imm_{22}, r0

Description: The immediate value, imm_{22}, is sign extended to 64 bits and placed in GR r_1.

Operation: `See “add — Add” on page 3:14.`

mov — Move Indirect Register

Format: (*qp*) mov *r_1* = *ireg*[*r_3*] from_form M43

(*qp*) mov *ireg*[*r_3*] = *r_2* to_form M42

Description: The source operand is copied to the destination register.

For move from indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg* (see Table 2-40 below). The indexed register is read and its value is copied into GR r_1.

For move to indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg*. GR r_2 is read and its value copied into the indexed register.

Table 2-40. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
dbr	Data Breakpoint Register
ibr	Instruction Breakpoint Register
pkr	Protection Key Register
pmc	Performance Monitor Configuration Register
pmd	Performance Monitor Data Register
rr	Region Register

For all register files other than the region registers, bits {7:0} of GR r_3 are used as the index. For region registers, bits {63:61} are used. The remainder of the bits are ignored.

Instruction and data breakpoint, performance monitor configuration, protection key, and region registers can only be accessed at the most privileged level. Performance monitor data registers can only be written at the most privileged level.

The CPU identification registers can only be read. There is no to_form of this instruction.

For move to protection key register, the processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers. If any matching keys are found, duplicate protection keys are invalidated.

Apart from the PMC and PMD register files, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of PMC and PMD register files result in implementation dependent behavior but do not fault.

Modifying a region register or a protection key register which is being used to translate:

- the executing instruction stream when PSR.it == 1, or
- the data space for an eager RSE reference when PSR.rt == 1

is an undefined operation.

Operation:

```
if (PR[qp]) {
    if (ireg == RR_TYPE)
        tmp_index = GR[r3]{63:61};
    else // all other register types
        tmp_index = GR[r3]{7:0};
```

```
if (from_form) {
    check_target_register(r1);

    if (PSR.cpl != 0 && !(ireg == PMD_TYPE || ireg == CPUID_TYPE))
        privileged_operation_fault(0);

    if (GR[r3].nat)
        register_nat_consumption_fault(0);

    if (is_reserved_reg(ireg, tmp_index))
        reserved_register_field_fault();

    if (PSR.vm == 1 && ireg != PMD_TYPE)
        virtualization_fault();

    if (ireg == PMD_TYPE) {
        if ((PSR.cpl != 0) && ((PSR.sp == 1) ||
             (tmp_index > 3 &&
             tmp_index <= IMPL_MAXGENERIC_PMCPMD &&
             PMC[tmp_index].pm == 1)))
            GR[r1] = 0;
        else
            GR[r1] = pmd_read(tmp_index);
    } else
        switch (ireg) {
            case CPUID_TYPE:  GR[r1] = CPUID[tmp_index]; break;
            case DBR_TYPE:    GR[r1] = DBR[tmp_index]; break;
            case IBR_TYPE:    GR[r1] = IBR[tmp_index]; break;
            case PKR_TYPE:    GR[r1] = PKR[tmp_index]; break;
            case PMC_TYPE:    GR[r1] = pmc_read(tmp_index); break;
            case RR_TYPE:     GR[r1] = RR[tmp_index]; break;
        }
    GR[r1].nat = 0;
} else {         // to_form
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (GR[r2].nat || GR[r3].nat)
        register_nat_consumption_fault(0);

    if (is_reserved_reg(ireg, tmp_index)
        || ireg == CPUID_TYPE
        || is_reserved_field(ireg, tmp_index, GR[r2]))
        reserved_register_field_fault();
    if (PSR.vm == 1)
        virtualization_fault();
    if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing valid prot key
        if ((tmp_slot = tlb_search_pkr(GR[r2]{31:8})) != NOT_FOUND)
            PKR[tmp_slot].v = 0; // clear valid bit of matching key reg
    }
    tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);
    switch (ireg) {
        case DBR_TYPE:    DBR[tmp_index] = tmp_val; break;
        case IBR_TYPE:    IBR[tmp_index] = tmp_val; break;
        case PKR_TYPE:    PKR[tmp_index] = tmp_val; break;
        case PMC_TYPE:    pmc_write(tmp_index, tmp_val); break;
```

```
            case PMD_TYPE:    pmd_write(tmp_index, tmp_val); break;
            case RR_TYPE:     RR[tmp_index]= tmp_val; break;
        }
    }
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

Serialization: For move to data breakpoint registers, software must issue a data serialize operation before issuing a memory reference dependent on the modified register.

For move to instruction breakpoint registers, software must issue an instruction serialize operation before fetching an instruction dependent on the modified register.

For move to protection key, region, performance monitor configuration, and performance monitor data registers, software must issue an instruction or data serialize operation to ensure the changes are observed before issuing any dependent instruction.

To obtain improved accuracy, software can issue an instruction or data serialize operation before reading the performance monitors.

mov — Move Instruction Pointer

Format: (*qp*) mov r_1 = ip I25

Description: The Instruction Pointer (IP) for the bundle containing this instruction is copied into GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = IP;
    GR[r1].nat = 0;
}
```

Interruptions: Illegal Operation fault

mov — Move Predicates

Format:	(*qp*) mov r_1 = pr	from_form	I25
	(*qp*) mov pr = r_2, $mask_{17}$	to_form	I23
	(*qp*) mov pr.rot = imm_{44}	to_rotate_form	I24

Description: The source operand is copied to the destination register.

For moving the predicates to a GR, PR i is copied to bit position i within GR r_1.

For moving to the predicates, the source can either be a general register, or an immediate value. In the to_form, the source operand is GR r_2 and only those predicates specified by the immediate value $mask_{17}$ are written. The value $mask_{17}$ is encoded in the instruction in an imm_{16} field such that: $imm_{16} = mask_{17} >> 1$. Predicate register 0 is always one. The $mask_{17}$ value is sign extended. The most significant bit of $mask_{17}$, therefore, is the mask bit for all of the rotating predicates. If there is a deferred exception for GR r_2 (the NaT bit is 1), a Register NaT Consumption fault is taken.

In the to_rotate_form, only the 48 rotating predicates can be written. The source operand is taken from the imm_{44} operand (which is encoded in the instruction in an imm_{28} field, such that: $imm_{28} = imm_{44} >> 16$). The low 16-bits correspond to the static predicates. The immediate is sign extended to set the top 21 predicates. Bit position i in the source operand is copied to PR i.

This instruction operates as if the predicate rotation base in the Current Frame Marker (CFM.rrb.pr) were zero.

Operation:

```
if (PR[$qp$]) {
    if (from_form) {
        check_target_register($r_1$);
        GR[$r_1$] = 1;                              // PR[0] is always 1
        for (i = 1; i <= 63; i++) {
            GR[$r_1$]{i} = PR[pr_phys_to_virt(i)];
        }
        GR[$r_1$].nat = 0;
    } else if (to_form) {
        if (GR[$r_2$].nat)
            register_nat_consumption_fault(0);
        tmp_src = sign_ext($mask_{17}$, 17);
        for (i = 1; i <= 63; i++) {
            if (tmp_src{i})
                PR[pr_phys_to_virt(i)] = GR[$r_2$]{i};
        }
    } else {   // to_rotate_form
        tmp_src = sign_ext($imm_{44}$, 44);
        for (i = 16; i <= 63; i++) {
            PR[pr_phys_to_virt(i)] = tmp_src{i};
        }
    }
}
```

Interruptions: Illegal Operation fault   Register NaT Consumption fault

mov — Move Processor Status Register

Format:	(*qp*) mov r_1 = psr	from_form	M36
	(*qp*) mov psr.l = r_2	to_form	M35

Description: The source operand is copied to the destination register. See Section 3.3.2, “Processor Status Register (PSR)” on page 2:23.

For move from processor status register, PSR bits {36:35} and {31:0} are read, and copied into GR r_1. All other bits of the PSR read as zero.

For move to processor status register, GR r_2 is read, bits {31:0} copied into PSR{31:0} and bits {63:32} are ignored. Bits {31:0} of GR r_2 corresponding to reserved fields of the PSR must be 0 or a Reserved Register/Field fault will result. An implementation may also raise Reserved Register/Field fault if bits {63:32} in GR r_2 corresponding to reserved fields of the PSR are non-zero.

Moves to and from the PSR can only be performed at the most privileged level, and when PSR.vm is 0.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1) are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (from_form)
        check_target_register(r1);
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (from_form) {
        if (PSR.vm == 1)
            virtualization_fault();
        tmp_val = zero_ext(PSR{31:0}, 32);     // read lower 32 bits
        tmp_val |= PSR{36:35} << 35;           // read mc and it bits
        GR[r1] = tmp_val;                      // other bits read as zero
        GR[r1].nat = 0;
    } else {          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_MOVPART, GR[r2]))
            reserved_register_field_fault();

        if (PSR.vm == 1)
            virtualization_fault();

        PSR{31:0} = GR[r2]{31:0};
    }
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

Serialization: Software must issue an instruction or data serialize operation before issuing instructions dependent upon the altered PSR bits. Unlike with the `rsm` instruction, the PSR.i bit is not treated specially when cleared.

mov — Move User Mask

Format:	(*qp*) mov r_1 = psr.um	from_form	M36
	(*qp*) mov psr.um = r_2	to_form	M35

Description: The source operand is copied to the destination register.

For move from user mask, PSR{5:0} is read, zero-extend, and copied into GR r_1.

For move to user mask, PSR{5:0} is written by bits {5:0} of GR r_2. PSR.up can only be modified if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Writing a non-zero value into any other parts of the PSR results in a Reserved Register/Field fault.

Operation:

```
if (PR[qp]) {
    if (from_form) {
        check_target_register(r1);

        GR[r1] = zero_ext(PSR{5:0}, 6);
        GR[r1].nat = 0;
    } else {                                          // to_form
        if (GR[r2].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_field(PSR_TYPE, PSR_UM, GR[r2]))
            reserved_register_field_fault();

        PSR{1:0} = GR[r2]{1:0};

        if (PSR.sp == 0)            // unsecured perf monitor
            PSR{2} = GR[r2]{2};

        PSR{5:3} = GR[r2]{5:3};
    }
}
```

Interruptions: Illegal Operation fault; Reserved Register/Field fault; Register NaT Consumption fault

Serialization: All user mask modifications are observed by the next instruction group.

movl — Move Long Immediate

Format: (qp) movl r_1 = imm_{64} X2

Description: The immediate value imm_{64} is copied to GR r_1. The L slot of the bundle contains 41 bits of imm_{64}.

Operation:

```
if (PR[qp]) {
    check_target_register(r_1);

    GR[r_1] = imm_64;
    GR[r_1].nat = 0;
}
```

Interruptions: Illegal Operation fault

mpy4 — Unsigned Integer Multiply

Format: (*qp*) mpy4 $r_1 = r_2, r_3$ I2

Description: The lower 32 bits of each of the two source operands are treated as unsigned values and are multiplied, and the result is placed in GR r_1. The upper 32 bits of each of the source operands are ignored.

Operation:

```
if (PR[qp]) {
    if (!instruction_implemented(mpy4))
        illegal_operation_fault();
    check_target_register($r_1$);

    GR[$r_1$] = zero_ext(GR[$r_2$], 32) * zero_ext(GR[$r_3$], 32);
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

mpyshl4 — Unsigned Integer Shift Left and Multiply

Format: (*qp*) mpyshl4 $r_1 = r_2, r_3$ I2

Description: The upper 32 bits of GR r_2 and the lower 32 bits of GR r_3 are treated as unsigned values and are multiplied. The result of the multiplication is shifted left 32 bits, with the vacated bit positions filled with zeroes, and the result is placed in GR r_1. The lower 32 bits of GR r_2 and the upper 32 bits of GR r_3 are ignored.

This instruction can be used to perform a 64-bit integer multiply operation producing a 64-bit result ($r_c = r_a * r_b$):

```
mpy4        r1 = ra, rb;;  //partial product low 32 bits * low 32 bits
mpyshl4     r2 = ra, rb;;  //partial product high 32 bits * low 32 bits
mpyshl4     r3 = rb, ra    //partial product low 32 bits * high 32 bits
add         r1 = r1, r2;;  //partial sum
add         rc = r1, r3    //final sum
```

Operation:

```
if (PR[qp]) {
    if (!instruction_implemented(MPYSHL4))
        illegal_operation_fault();
    check_target_register(r1);

    GR[r1] = (zero_ext((GR[r2] >> 32), 32) * zero_ext(GR[r3], 32)) << 32;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

mux — Mux

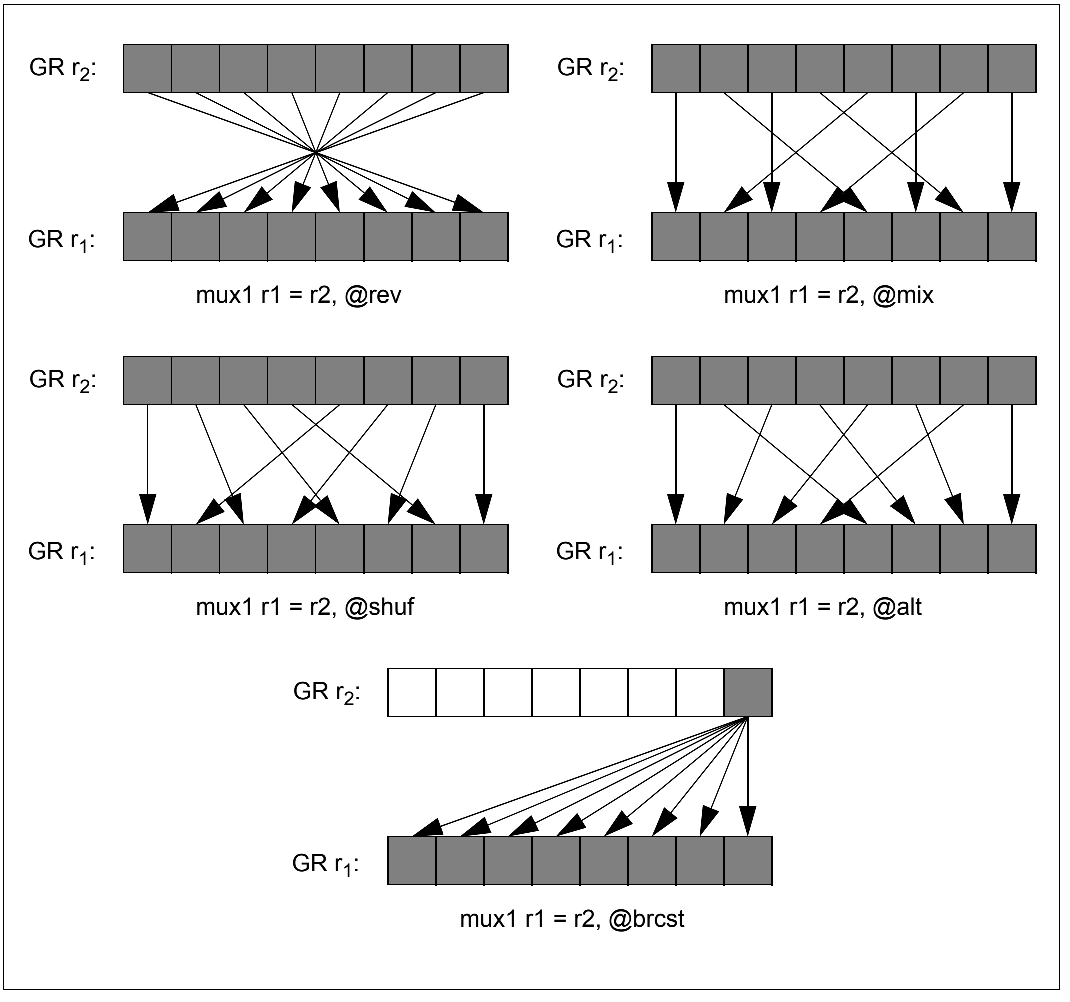
Format: (*qp*) mux1 $r_1 = r_2$, $mbtype_4$ one_byte_form I3
(*qp*) mux2 $r_1 = r_2$, $mhtype_8$ two_byte_form I4

Description: A permutation is performed on the packed elements in a single source register, GR r_2, and the result is placed in GR r_1. For 8-bit elements, only some of all possible permutations can be specified. The five possible permutations are given in Table 2-41 and shown in Figure 2-26.

Table 2-41. Mux Permutations for 8-bit Elements

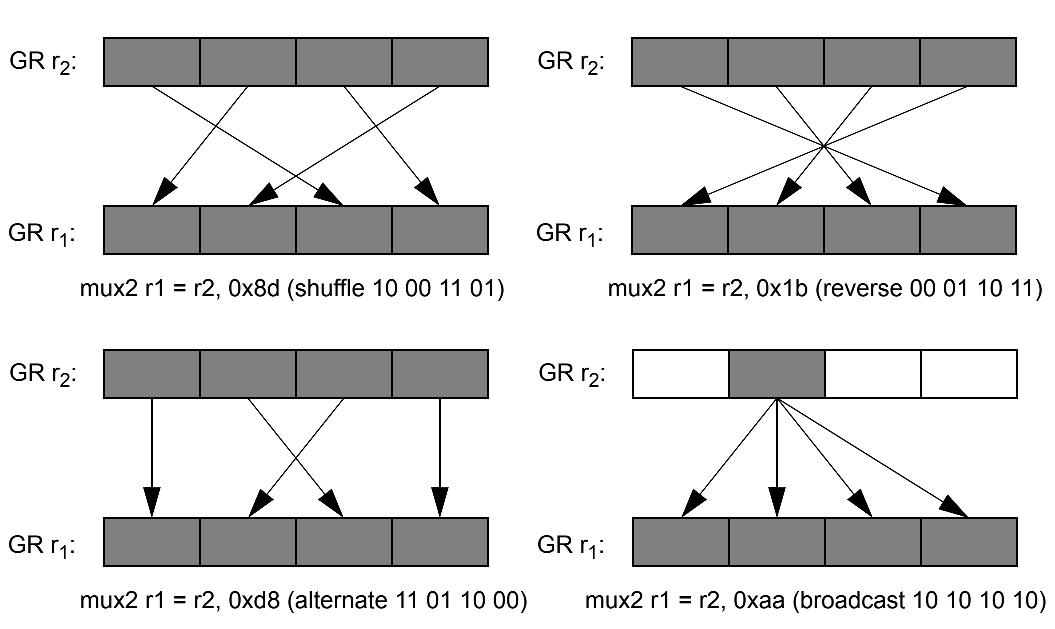
$mbtype_4$	Function
@rev	Reverse the order of the bytes
@mix	Perform a Mix operation on the two halves of GR r_2
@shuf	Perform a Shuffle operation on the two halves of GR r_2
@alt	Perform an Alternate operation on the two halves of GR r_2
@brcst	Perform a Broadcast operation on the least significand byte of GR r_2

Figure 2-26. Mux1 Operation (8-bit elements)



For 16-bit elements, all possible permutations, with and without repetitions can be specified. They are expressed with an 8-bit *mhtype*$_8$ field, which encodes the indices of the four 16-bit data elements. The indexed 16-bit elements of GR r_2 are copied to corresponding 16-bit positions in the target register GR r_1. The indices are encoded in little-endian order. (The 8 bits of *mhtype*$_8$[7:0] are grouped in pairs of bits and named *mhtype*$_8$[3], *mhtype*$_8$[2], *mhtype*$_8$[1], *mhtype*$_8$[0] in the Operation section).

Figure 2-27. Mux2 Examples (16-bit elements)


GR r_2:
GR r_1:
mux2 r1 = r2, 0x8d (shuffle 10 00 11 01)
GR r_2:
GR r_1:
mux2 r1 = r2, 0x1b (reverse 00 01 10 11)
GR r_2:
GR r_1:
mux2 r1 = r2, 0xd8 (alternate 11 01 10 00)
GR r_2:
GR r_1:
mux2 r1 = r2, 0xaa (broadcast 10 10 10 10)

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};
        x[1] = GR[r2]{15:8};
        x[2] = GR[r2]{23:16};
        x[3] = GR[r2]{31:24};
        x[4] = GR[r2]{39:32};
        x[5] = GR[r2]{47:40};
        x[6] = GR[r2]{55:48};
        x[7] = GR[r2]{63:56};

        switch (mbtype) {
            case '@rev':
                GR[r1] = concatenate8(x[0], x[1], x[2], x[3],
                                      x[4], x[5], x[6], x[7]);
                break;

            case '@mix':
                GR[r1] = concatenate8(x[7], x[3], x[5], x[1],
                                      x[6], x[2], x[4], x[0]);
                break;

            case '@shuf':
                GR[r1] = concatenate8(x[7], x[3], x[6], x[2],
                                      x[5], x[1], x[4], x[0]);
                break;

            case '@alt':
                GR[r1] = concatenate8(x[7], x[5], x[3], x[1],
                                      x[6], x[4], x[2], x[0]);
                break;

            case '@brcst':
                GR[r1] = concatenate8(x[0], x[0], x[0], x[0],
                                      x[0], x[0], x[0], x[0]);
                break;
        }
    } else {                                          // two_byte_form
        x[0] = GR[r2]{15:0};
        x[1] = GR[r2]{31:16};
        x[2] = GR[r2]{47:32};
        x[3] = GR[r2]{63:48};

        res[0] = x[mhtype8{1:0}];
        res[1] = x[mhtype8{3:2}];
        res[2] = x[mhtype8{5:4}];
        res[3] = x[mhtype8{7:6}];

        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat;
}
```

Interruptions: Illegal Operation fault

nop — No Operation

Format:

(*qp*) nop *imm*$_{21}$	pseudo-op	
(*qp*) nop.i *imm*$_{21}$	i_unit_form	I18
(*qp*) nop.b *imm*$_{21}$	b_unit_form	B9
(*qp*) nop.m *imm*$_{21}$	m_unit_form	M48
(*qp*) nop.f *imm*$_{21}$	f_unit_form	F16
(*qp*) nop.x *imm*$_{62}$	x_unit_form	X5

Description: No operation is done.

The immediate, *imm*$_{21}$ or *imm*$_{62}$, can be used by software as a marker in program code. It is ignored by hardware.

For the x_unit_form, the L slot of the bundle contains the upper 41 bits of *imm*$_{62}$.

A `nop.i` instruction may be encoded in an MLI-template bundle, in which case the L slot of the bundle is ignored.

This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Operation:

```
if (PR[qp]) {
    ; // no operation
}
```

Interruptions: None

or — Logical Or

Format:

(*qp*) or r_1 = r_2, r_3	register_form	A1
(*qp*) or r_1 = imm_8, r_3	imm8_form	A3

Description: The two source operands are logically ORed and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src | GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pack — Pack

Format:

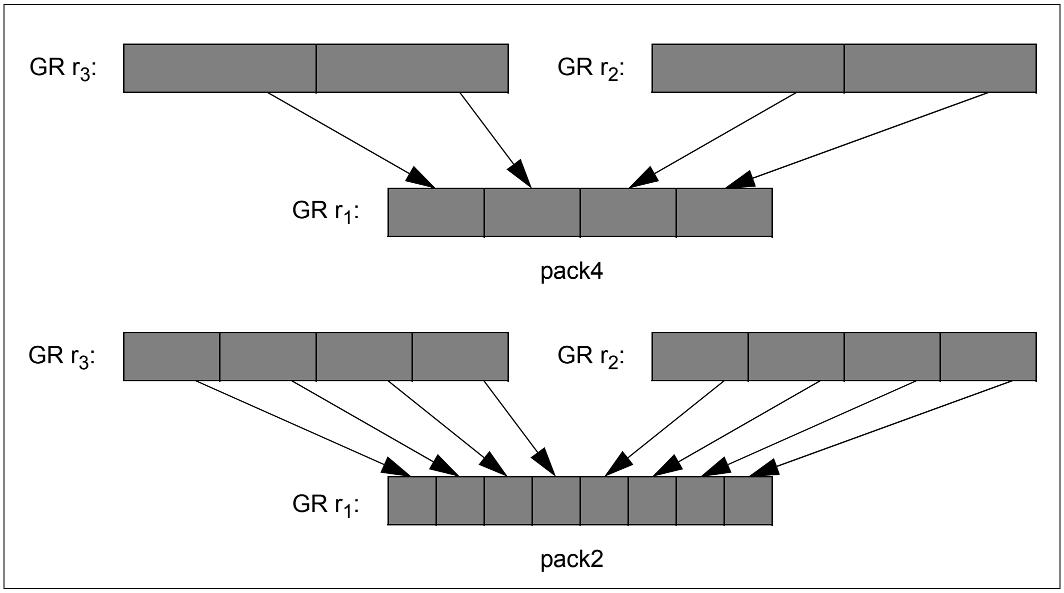
(qp) pack2.sss $r_1 = r_2, r_3$	two_byte_form, signed_saturation_form	I2
(qp) pack2.uss $r_1 = r_2, r_3$	two_byte_form, unsigned_saturation_form	I2
(qp) pack4.sss $r_1 = r_2, r_3$	four_byte_form, signed_saturation_form	I2

Description: 32-bit or 16-bit elements from GR r_2 and GR r_3 are converted into 16-bit or 8-bit elements respectively, and the results are placed GR r_1. The source elements are treated as signed values. If a source element cannot be represented in the result element, then saturation clipping is performed. The saturation can either be signed or unsigned. If an element is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-42.

Table 2-42. Pack Saturation Limits

Size	Source Element Width	Result Element Width	Saturation	Upper Limit	Lower Limit
2	16 bit	8 bit	signed	0x7f	0x80
2	16 bit	8 bit	unsigned	0xff	0x00
4	32 bit	16 bit	signed	0x7fff	0x8000

Figure 2-28. Pack Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (two_byte_form) {
        if (signed_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
        } else {                                    // unsigned_saturation_form
            max = 0xff;
            min = 0x00;
        }
        temp[0] = sign_ext(GR[r2]{15:0},  16);
        temp[1] = sign_ext(GR[r2]{31:16}, 16);
        temp[2] = sign_ext(GR[r2]{47:32}, 16);
        temp[3] = sign_ext(GR[r2]{63:48}, 16);
        temp[4] = sign_ext(GR[r3]{15:0},  16);
        temp[5] = sign_ext(GR[r3]{31:16}, 16);
        temp[6] = sign_ext(GR[r3]{47:32}, 16);
        temp[7] = sign_ext(GR[r3]{63:48}, 16);

        for (i = 0; i < 8; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);

    } else {                                        // four_byte_form
        max = sign_ext(0x7fff, 16);                 // signed_saturation_form
        min = sign_ext(0x8000, 16);
        temp[0] = sign_ext(GR[r2]{31:0},  32);
        temp[1] = sign_ext(GR[r2]{63:32}, 32);
        temp[2] = sign_ext(GR[r3]{31:0},  32);
        temp[3] = sign_ext(GR[r3]{63:32}, 32);

        for (i = 0; i < 4; i++) {
            if (temp[i] > max)
                temp[i] = max;

            if (temp[i] < min)
                temp[i] = min;
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

padd — Parallel Add

Format:	(*qp*) padd1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
	(*qp*) padd1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
	(*qp*) padd1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
	(*qp*) padd1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
	(*qp*) padd2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
	(*qp*) padd2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
	(*qp*) padd2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
	(*qp*) padd2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
	(*qp*) padd4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

Description: The sets of elements from the two source operands are added, and the results placed in GR r_1.

If a sum of two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-43. If the sum of two elements is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-44.

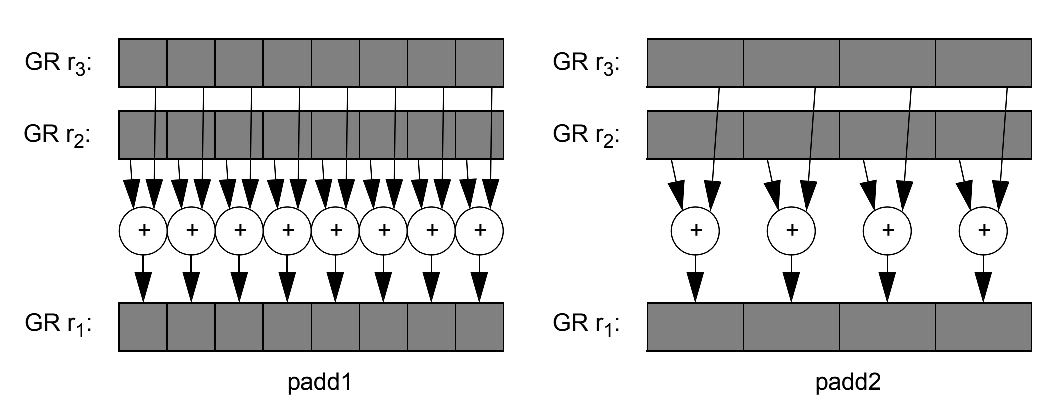
Table 2-43. Parallel Add Saturation Completers

Completer	Result r_1 treated as	Source r_2 treated as	Source r_3 treated as
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-44. Parallel Add Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-29. Parallel Add Examples



Operation:

```
if (PR[$qp$]) {
    check_target_register($r_1$);

    if (one_byte_form) {                                // one-byte elements
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};

        if (sss_saturation_form) {
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);

            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {
            max = 0xff;
            min = 0x00;

            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        } else {                                        // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[$r_1$] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                               temp[3], temp[2], temp[1], temp[0]);

    } else if (two_byte_form) {                         // 2-byte elements
        x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[r2]{47:32};      y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};      y[3] = GR[r3]{63:48};

        if (sss_saturation_form) {
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);

            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {
            max = 0xffff;
            min = 0x0000;

            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        } else {                                     // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;

                if (temp[i] < min)
                    temp[i] = min;
            }
        }
        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);

    } else {                                         // four-byte elements
        x[0] = GR[r2]{31:0};       y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};      y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                    // modulo_form
            temp[i] = zero_ext(x[i], 32) + zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pavg — Parallel Average

Format:	(*qp*) pavg1 $r_1 = r_2, r_3$	normal_form, one_byte_form	A9
	(*qp*) pavg1.raz $r_1 = r_2, r_3$	raz_form, one_byte_form	A9
	(*qp*) pavg2 $r_1 = r_2, r_3$	normal_form, two_byte_form	A9
	(*qp*) pavg2.raz $r_1 = r_2, r_3$	raz_form, two_byte_form	A9

Description: The unsigned data elements of GR r_2 are added to the unsigned data elements of GR r_3. The results of the add are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the carry bits of the sums. To prevent cumulative round-off errors, an averaging is performed. The unsigned results are placed in GR r_1.

The averaging operation works as follows. In the normal_form, the low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding sum is 1. In the raz_form, the average rounds away from zero by adding 1 to each of the sums.

Figure 2-30. Parallel Average Example

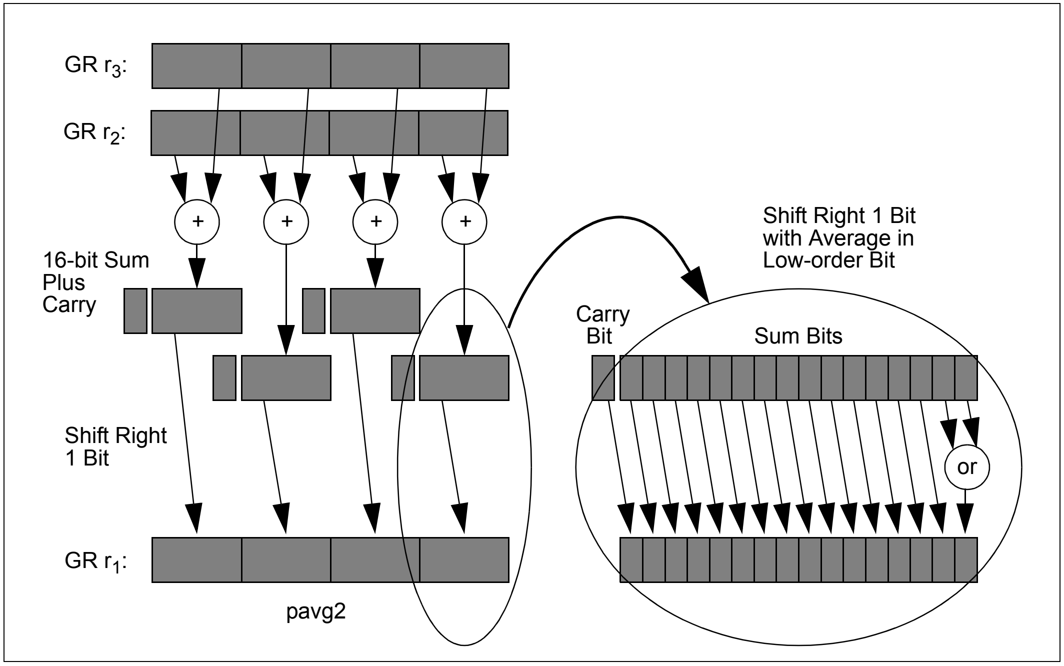
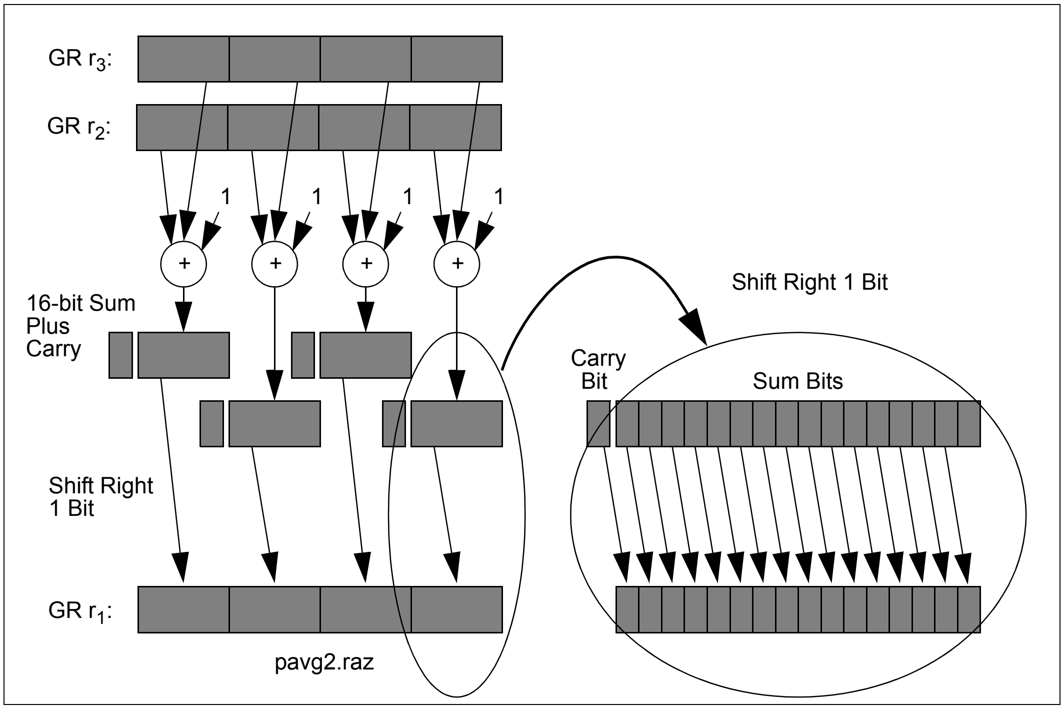


Figure 2-31. Parallel Average with Round Away from Zero Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {
        x[0] = GR[r2]{7:0};       y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};      y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};     y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};     y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};     y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};     y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};     y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};     y[7] = GR[r3]{63:56};

        if (raz_form) {
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                    // normal form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                              res[3], res[2], res[1], res[0]);

    } else {                                        // two_byte_form
        x[0] = GR[r2]{15:0};      y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};     y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};     y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};     y[3] = GR[r3]{63:48};

        if (raz_form) {
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16) + 1;
                res[i] = shift_right_unsigned(temp[i], 1);
            }
        } else {                                    // normal form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);
                res[i] = shift_right_unsigned(temp[i], 1) | (temp[i]{0});
            }
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

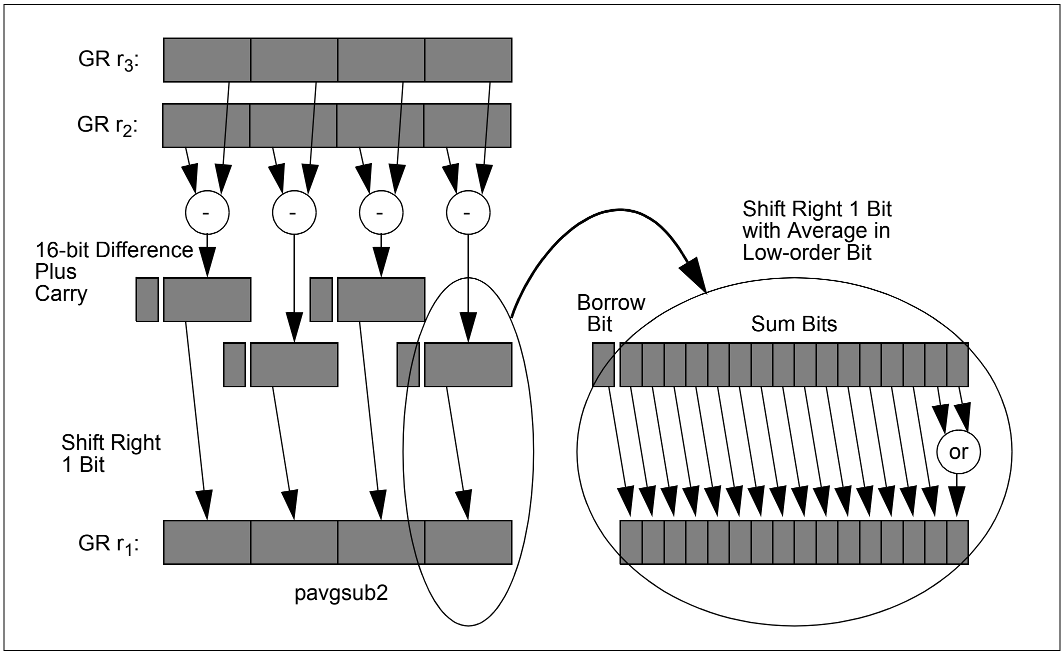
Interruptions: Illegal Operation fault

pavgsub — Parallel Average Subtract

Format: (*qp*) pavgsub1 $r_1 = r_2, r_3$ one_byte_form A9
(*qp*) pavgsub2 $r_1 = r_2, r_3$ two_byte_form A9

Description: The unsigned data elements of GR r_3 are subtracted from the unsigned data elements of GR r_2. The results of the subtraction are then each independently shifted to the right by one bit position. The high-order bits of each element are filled with the borrow bits of the subtraction (the complements of the ALU carries). To prevent cumulative round-off errors, an averaging is performed. The low-order bit of each result is set to 1 if at least one of the two least significant bits of the corresponding difference is 1. The signed results are placed in GR r_1.

Figure 2-32. Parallel Average Subtract Example



Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);

    if (one_byte_form) {
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};

        for (i = 0; i < 8; i++) {
            temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            res[i] = (temp[i]{8:0} u>> 1) | (temp[i]{0});
        }
        GR[$r_1$] = concatenate8(res[7], res[6], res[5], res[4],
                               res[3], res[2], res[1], res[0]);

    } else {                                          // two_byte_form
        x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};

        for (i = 0; i < 4; i++) {
            temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            res[i] = (temp[i]{16:0} u>> 1) | (temp[i]{0});
        }
        GR[$r_1$] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

pcmp — Parallel Compare

Format:

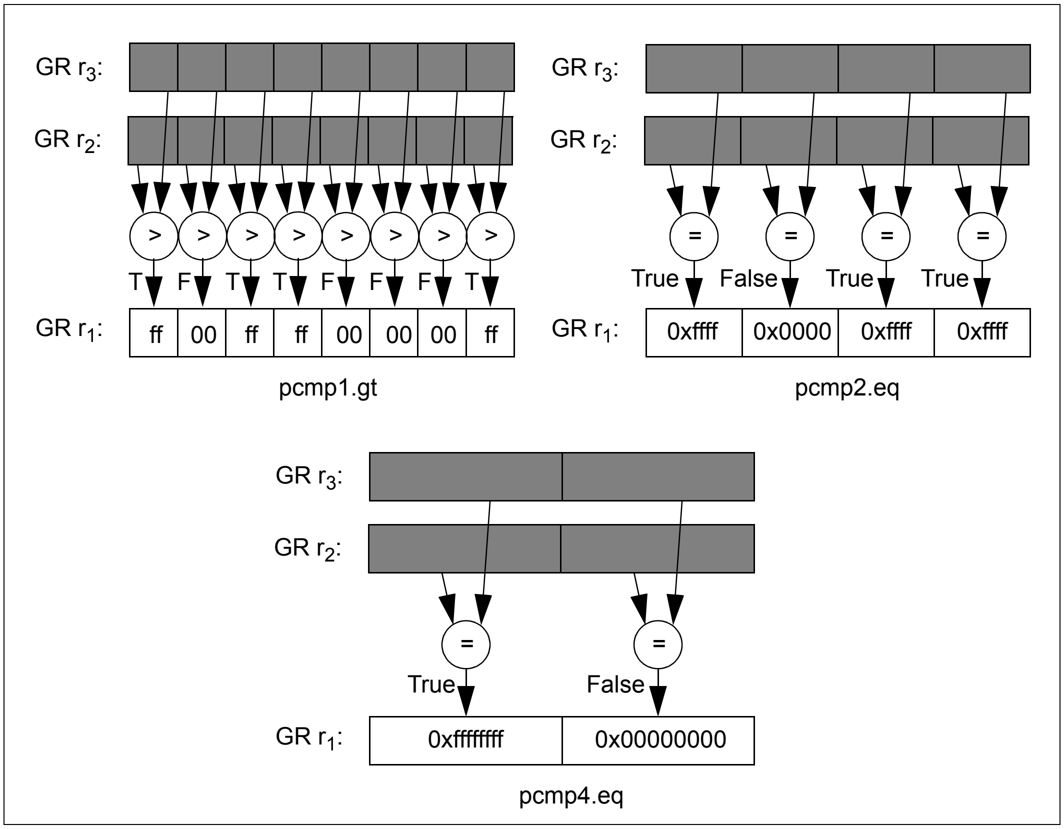
(qp) pcmp1.*prel* $r_1 = r_2, r_3$	one_byte_form	A9
(qp) pcmp2.*prel* $r_1 = r_2, r_3$	two_byte_form	A9
(qp) pcmp4.*prel* $r_1 = r_2, r_3$	four_byte_form	A9

Description: The two source operands are compared for one of the two relations shown in Table 2-45. If the comparison condition is true for corresponding data elements of GR r_2 and GR r_3, then the corresponding data element in GR r_1 is set to all ones. If the comparison condition is false, then the corresponding data element in GR r_1 is set to all zeros. For the '>' relation, both operands are interpreted as signed.

Table 2-45. Pcmp Relations

prel	Compare Relation (r_2 *prel* r_3)
eq	$r_2 == r_3$
gt	$r_2 > r_3$ (signed)

Figure 2-33. Parallel Compare Examples



Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);

    if (one_byte_form) {                              // one-byte elements
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};
        for (i = 0; i < 8; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 8),
                                         sign_ext(y[i], 8));

            if (tmp_rel)
                res[i] = 0xff;
            else
                res[i] = 0x00;
        }
        GR[$r_1$] = concatenate8(res[7], res[6], res[5], res[4],
                               res[3], res[2], res[1], res[0]);
    } else if (two_byte_form) {                       // two-byte elements
        x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};
        for (i = 0; i < 4; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 16),
                                         sign_ext(y[i], 16));

            if (tmp_rel)
                res[i] = 0xffff;
            else
                res[i] = 0x0000;
        }
        GR[$r_1$] = concatenate4(res[3], res[2], res[1], res[0]);
    } else {                                          // four-byte elements
        x[0] = GR[$r_2$]{31:0};     y[0] = GR[$r_3$]{31:0};
        x[1] = GR[$r_2$]{63:32};    y[1] = GR[$r_3$]{63:32};
        for (i = 0; i < 2; i++) {
            if (prel == 'eq')
                tmp_rel = x[i] == y[i];
            else // 'gt'
                tmp_rel = greater_signed(sign_ext(x[i], 32),
                                         sign_ext(y[i], 32));

            if (tmp_rel)
                res[i] = 0xffffffff;
                else
                    res[i] = 0x00000000;
            }
            GR[r1] = concatenate2(res[1], res[0]);
        }
        GR[r1].nat = GR[r2].nat || GR[r3].nat;
    }
```

Interruptions: Illegal Operation fault

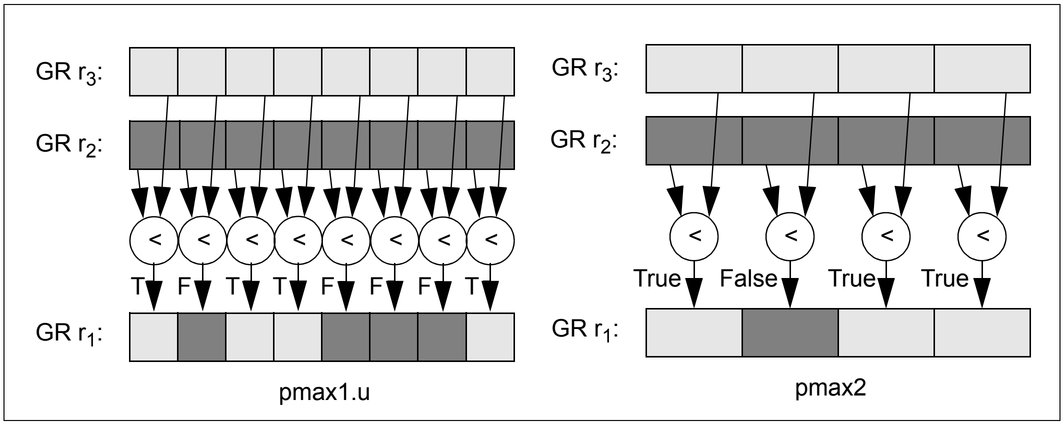
pmax — Parallel Maximum

Format: (*qp*) pmax1.u $r_1 = r_2, r_3$ one_byte_form I2

(*qp*) pmax2 $r_1 = r_2, r_3$ two_byte_form I2

Description: The maximum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the greater of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the greater of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-34. Parallel Maximum Examples



Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);

    if (one_byte_form) {                                // one-byte elements
        x[0] = GR[$r_2$]{7:0};       y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};      y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};     y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};     y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};     y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};     y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};     y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};     y[7] = GR[$r_3$]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? y[i] : x[i];
        }
        GR[$r_1$] = concatenate8(res[7], res[6], res[5], res[4],
                                res[3], res[2], res[1], res[0]);
    } else {                                            // two-byte elements
        x[0] = GR[$r_2$]{15:0};      y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};     y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};     y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};     y[3] = GR[$r_3$]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? y[i] : x[i];
        }
        GR[$r_1$] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

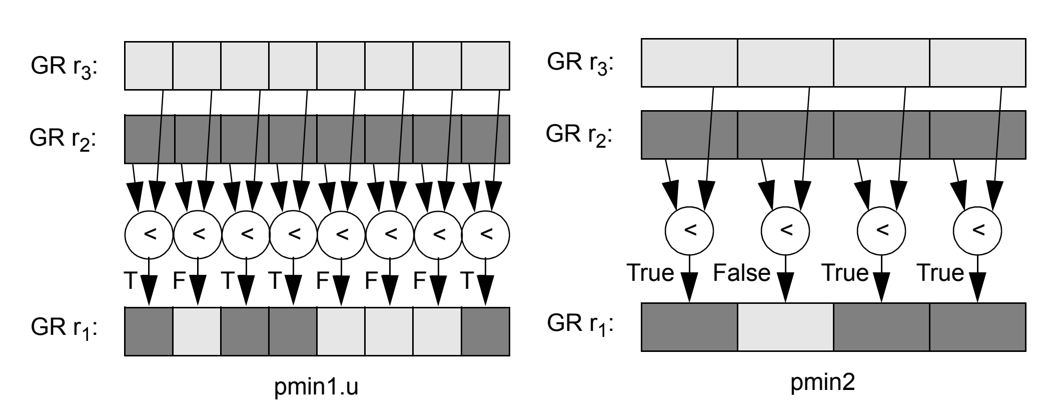
Interruptions: Illegal Operation fault

pmin — Parallel Minimum

Format: (*qp*) pmin1.u $r_1 = r_2, r_3$ one_byte_form I2
(*qp*) pmin2 $r_1 = r_2, r_3$ two_byte_form I2

Description: The minimum of the two source operands is placed in the result register. In the one_byte_form, each unsigned 8-bit element of GR r_2 is compared with the corresponding unsigned 8-bit element of GR r_3 and the smaller of the two is placed in the corresponding 8-bit element of GR r_1. In the two_byte_form, each signed 16-bit element of GR r_2 is compared with the corresponding signed 16-bit element of GR r_3 and the smaller of the two is placed in the corresponding 16-bit element of GR r_1.

Figure 2-35. Parallel Minimum Examples



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (one_byte_form) {                                   // one-byte elements
        x[0] = GR[r2]{7:0};       y[0] = GR[r3]{7:0};
        x[1] = GR[r2]{15:8};      y[1] = GR[r3]{15:8};
        x[2] = GR[r2]{23:16};     y[2] = GR[r3]{23:16};
        x[3] = GR[r2]{31:24};     y[3] = GR[r3]{31:24};
        x[4] = GR[r2]{39:32};     y[4] = GR[r3]{39:32};
        x[5] = GR[r2]{47:40};     y[5] = GR[r3]{47:40};
        x[6] = GR[r2]{55:48};     y[6] = GR[r3]{55:48};
        x[7] = GR[r2]{63:56};     y[7] = GR[r3]{63:56};
        for (i = 0; i < 8; i++) {
            res[i] = (zero_ext(x[i],8) < zero_ext(y[i],8)) ? x[i] : y[i];
        }
        GR[r1] = concatenate8(res[7], res[6], res[5], res[4],
                               res[3], res[2], res[1], res[0]);
    } else {                                               // two-byte elements
        x[0] = GR[r2]{15:0};      y[0] = GR[r3]{15:0};
        x[1] = GR[r2]{31:16};     y[1] = GR[r3]{31:16};
        x[2] = GR[r2]{47:32};     y[2] = GR[r3]{47:32};
        x[3] = GR[r2]{63:48};     y[3] = GR[r3]{63:48};
        for (i = 0; i < 4; i++) {
            res[i] = (sign_ext(x[i],16) < sign_ext(y[i],16)) ? x[i] : y[i];
        }
        GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    }
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

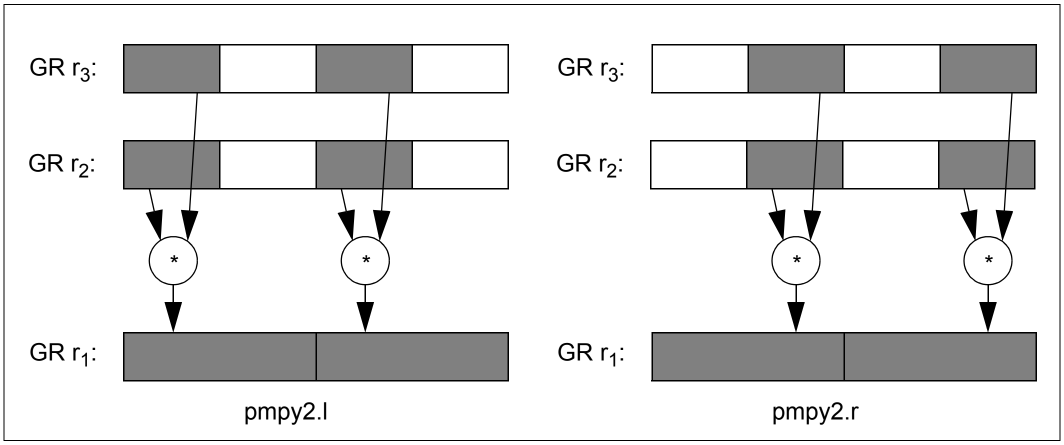
Interruptions: Illegal Operation fault

pmpy — Parallel Multiply

Format: (*qp*) pmpy2.r $r_1 = r_2, r_3$ right_form I2
(*qp*) pmpy2.l $r_1 = r_2, r_3$ left_form I2

Description: Two signed 16-bit data elements of GR r_2 are multiplied by the corresponding two signed 16-bit data elements of GR r_3 as shown in Figure 2-36. The two 32-bit results are placed in GR r_1.

Figure 2-36. Parallel Multiply Operation



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (right_form) {
        GR[r1]{31:0} = sign_ext(GR[r2]{15:0}, 16) *
                           sign_ext(GR[r3]{15:0}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{47:32}, 16) *
                           sign_ext(GR[r3]{47:32}, 16);
    } else {                                        // left_form
        GR[r1]{31:0} = sign_ext(GR[r2]{31:16}, 16) *
                           sign_ext(GR[r3]{31:16}, 16);
        GR[r1]{63:32} = sign_ext(GR[r2]{63:48}, 16) *
                           sign_ext(GR[r3]{63:48}, 16);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pmpyshr — Parallel Multiply and Shift Right

Format: (*qp*) pmpyshr2 $r_1 = r_2, r_3, count_2$ signed_form I1

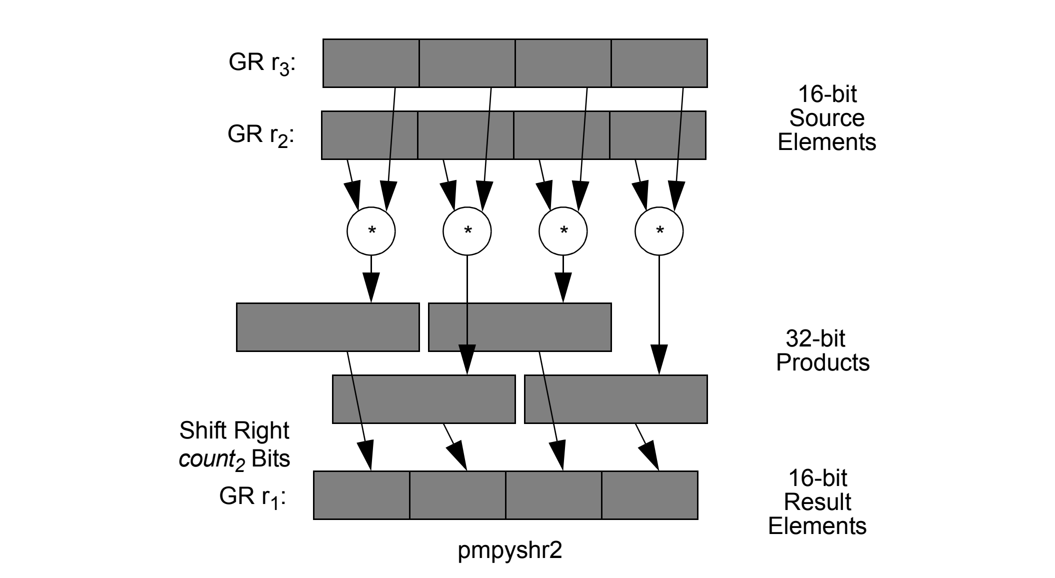
(*qp*) pmpyshr2.u $r_1 = r_2, r_3, count_2$ unsigned_form I1

Description: The four 16-bit data elements of GR r_2 are multiplied by the corresponding four 16-bit data elements of GR r_3 as shown in Figure 2-37. This multiplication can either be signed (pmpyshr2), or unsigned (pmpyshr2.u). Each product is then shifted to the right $count_2$ bits, and the least-significant 16-bits of each shifted product form 4 16-bit results, which are placed in GR r_1. A $count_2$ of 0 gives the 16 low bits of the results, a $count_2$ of 16 gives the 16 high bits of the results. The allowed values for $count_2$ are given in Table 2-46.

Table 2-46. Parallel Multiply and Shift Right Shift Options

$count_2$	Selected Bit Field from Each 32-bit Product
0	15:0
7	22:7
15	30:15
16	31:16

Figure 2-37. Parallel Multiply and Shift Right Operation



Operation:

```
if (PR[$qp$]) {
    check_target_register($r_1$);
    x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
    x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
    x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
    x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};
    for (i = 0; i < 4; i++) {
        if (unsigned_form)                          // unsigned multiplication
            temp[i] = zero_ext(x[i], 16) * zero_ext(y[i], 16);
        else                                        // signed multiplication
            temp[i] = sign_ext(x[i], 16) * sign_ext(y[i], 16);

        res[i] = temp[i]{($count_2$ + 15):$count_2$};
    }

    GR[$r_1$] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

popcnt — Population Count

Format: (*qp*) popcnt $r_1 = r_3$ I9

Description: The number of bits in GR r_3 having the value 1 is counted, and the resulting sum is placed in GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    res = 0;
    // Count up all the one bits
    for (i = 0; i < 64; i++) {
        res += GR[r3]{i};
    }

    GR[r1] = res;
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

probe — Probe Access

Format:

(qp) probe.r $r_1 = r_3, r_2$	regular_form, read_form, register_form	M38
(qp) probe.w $r_1 = r_3, r_2$	regular_form, write_form, register_form	M38
(qp) probe.r $r_1 = r_3, imm_2$	regular_form, read_form, immediate_form	M39
(qp) probe.w $r_1 = r_3, imm_2$	regular_form, write_form, immediate_form	M39
(qp) probe.r.fault r_3, imm_2	fault_form, read_form, immediate_form	M40
(qp) probe.w.fault r_3, imm_2	fault_form, write_form, immediate_form	M40
(qp) probe.rw.fault r_3, imm_2	fault_form, read_write_form, immediate_form	M40

Description: This instruction determines whether read or write access, with a specified privilege level, to a given virtual address is permitted. In the regular_form, GR r_1 is set to 1 if the specified access is allowed and to 0 otherwise. In the fault_form, if the specified access is allowed this instruction does nothing; if the specified access is not allowed, a fault is taken.

When PSR.dt is 1, the DTLB and the VHPT are queried for present translations to determine if access to the virtual address specified by GR r_3 bits {60:0} and the region register indexed by GR r_3 bits {63:61}, is permitted at the privilege level given by either GR r_2 bits{1:0} or imm_2. If PSR.pk is 1, protection key checks are also performed. The read or write form specifies whether the instruction checks for read or write access, or both.

When PSR.dt is 0, a regular_form `probe` uses its address operand as a virtual address to query the DTLB only, because the VHPT walker is disabled. If the probed address is found in the DTLB, the regular_form `probe` returns the appropriate value, if not an Alternate Data TLB fault is raised if psr.ic is 1 or a Data Nested TLB fault is raised if psr.ic is 0 or in-flight.

When PSR.dt is 0, a fault_form `probe` treats its address operand as a physical address, and takes no TLB related faults.

A regular_form `probe` to an unimplemented virtual address returns 0. A fault_form `probe` to an unimplemented virtual address (when PSR.dt is 1) or unimplemented physical address (when PSR.dt is 0) takes an Unimplemented Data Address fault.

If this instruction faults, then it will set the non-access bit in the ISR and set the ISR read or write bits depending on the completer. The faults generated by the different forms of the `probe` instruction are shown in Table 2-47 below:

Table 2-47. Faults for regular_form and fault_form Probe Instructions

Probe Form Type	Faults
regular_form	Register NaT Consumption fault Virtualization fault[a] Data Nested TLB fault Alternate Data TLB fault VHPT Data fault Data TLB fault Data Page Not Present fault Data NaT Page Consumption fault Data Key Miss fault
fault_form	Register NaT Consumption fault Unimplemented Data Address fault Virtualization fault[a] Data Nested TLB fault Alternate Data TLB fault VHPT Data fault Data TLB fault Data Page Not Present fault Data NaT Page Consumption fault Data Key Miss fault Data Key Permission fault Data Access Rights fault Data Dirty Bit fault Data Access Bit fault Data Debug fault

[a] This instruction may optionally raise Virtualization faults, see Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.

This instruction can only probe with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

When PSR.vm is 1, this instruction may optionally raise Virtualization faults, see Section 11.7.4.2.8, “Probe Instruction Virtualization” on page 2:344 for details.

Please refer to the ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** for usage information of the `probe` instruction.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS;
    itype |= (read_write_form) ? READ|WRITE : ((write_form) ? WRITE : READ);
    itype |= (fault_form) ? PROBE_FAULT : PROBE;
    itype |= (register_form) ? REGISTER_FORM : IMM_FORM;

    if (!fault_form)
        check_target_register(r1);

    if (GR[r3].nat || (register_form ? GR[r2].nat : 0))
        register_nat_consumption_fault(itype);

    tmp_pl = (register_form) ? GR[r2]{1:0} : imm2;
    if (tmp_pl < PSR.cpl)
        tmp_pl = PSR.cpl;

    if (fault_form) {
        tlb_translate(GR[r3], 1, itype, tmp_pl, &mattr, &defer);
    } else {               // regular_form
        if (impl_probe_intercept())
            check_probe_virtualization_fault(itype, tmp_pl);
        GR[r1] = tlb_grant_permission(GR[r3], itype, tmp_pl);
        GR[r1].nat = 0;
    }
}
```

Interruptions:

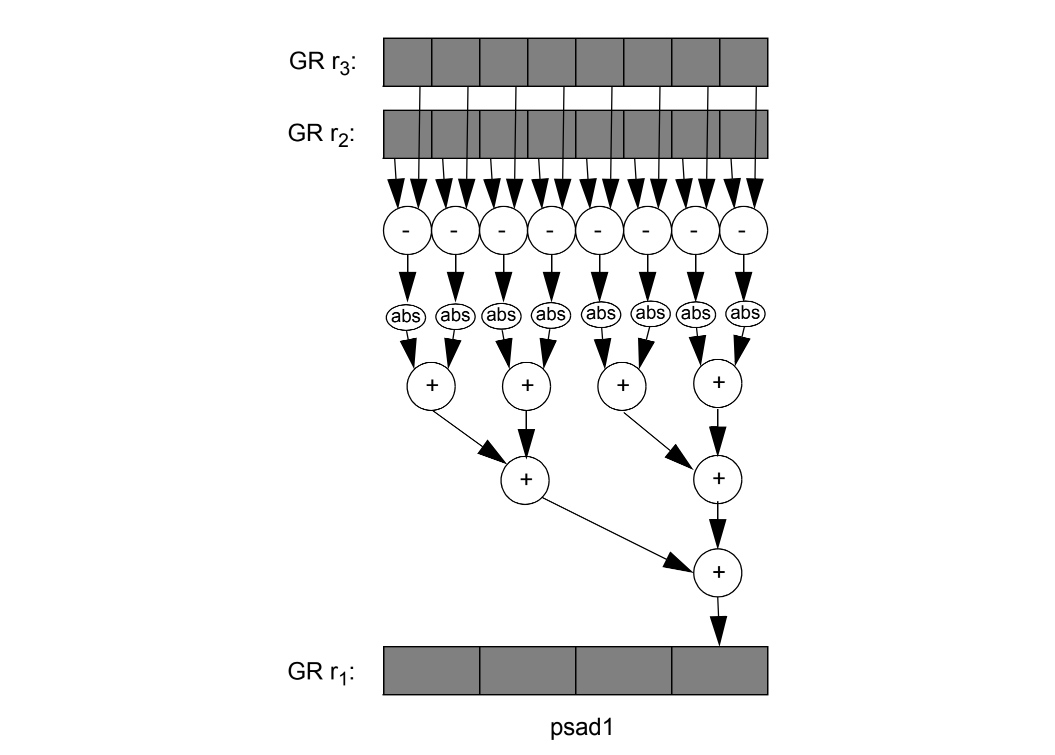
Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

psad — Parallel Sum of Absolute Difference

Format: (qp) psad1 $r_1 = r_2, r_3$ I2

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

Figure 2-38. Parallel Sum of Absolute Difference Example



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{7:0};      y[0] = GR[r3]{7:0};
    x[1] = GR[r2]{15:8};     y[1] = GR[r3]{15:8};
    x[2] = GR[r2]{23:16};    y[2] = GR[r3]{23:16};
    x[3] = GR[r2]{31:24};    y[3] = GR[r3]{31:24};
    x[4] = GR[r2]{39:32};    y[4] = GR[r3]{39:32};
    x[5] = GR[r2]{47:40};    y[5] = GR[r3]{47:40};
    x[6] = GR[r2]{55:48};    y[6] = GR[r3]{55:48};
    x[7] = GR[r2]{63:56};    y[7] = GR[r3]{63:56};

    GR[r1] = 0;
    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
        if (temp[i] < 0)
            temp[i] = -temp[i];
        GR[r1] += temp[i];
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

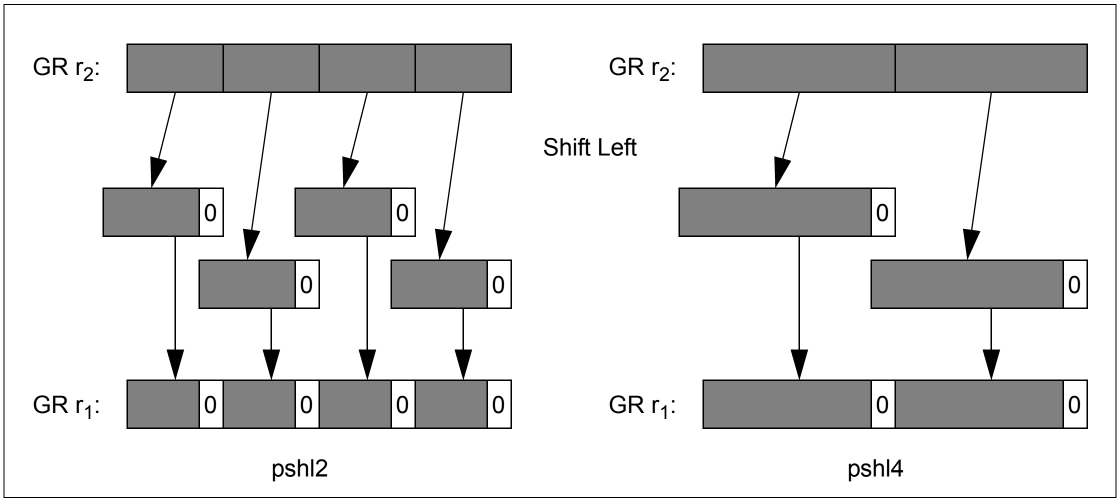
Interruptions: Illegal Operation fault

pshl — Parallel Shift Left

Format: (*qp*) pshl2 $r_1 = r_2, r_3$	two_byte_form, variable_form	I7
(*qp*) pshl2 $r_1 = r_2, count_5$	two_byte_form, fixed_form	I8
(*qp*) pshl4 $r_1 = r_2, r_3$	four_byte_form, variable_form	I7
(*qp*) pshl4 $r_1 = r_2, count_5$	four_byte_form, fixed_form	I8

Description: The data elements of GR r_2 are each independently shifted to the left by the scalar shift count in GR r_3, or in the immediate field $count_5$. The low-order bits of each element are filled with zeros. The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero results. The results are placed in GR r_1.

Figure 2-39. Parallel Shift Left Examples



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r3] : count5);
    tmp_nat = (variable_form ? GR[r3].nat : 0);

    if (two_byte_form) {                                // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        GR[r1]{15:0}  = GR[r2]{15:0}  << shift_count;
        GR[r1]{31:16} = GR[r2]{31:16} << shift_count;
        GR[r1]{47:32} = GR[r2]{47:32} << shift_count;
        GR[r1]{63:48} = GR[r2]{63:48} << shift_count;
    } else {                                            // four_byte_form
        if (shift_count u> 32)
            shift_count = 32;
        GR[r1]{31:0}  = GR[r2]{31:0}  << shift_count;
        GR[r1]{63:32} = GR[r2]{63:32} << shift_count;
    }

    GR[r1].nat = GR[r2].nat || tmp_nat;
}
```

Interruptions: Illegal Operation fault

pshladd — Parallel Shift Left and Add

Format: (*qp*) pshladd2 $r_1 = r_2, count_2, r_3$ A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the left by $count_2$ bits (shifting zeros into the low-order bits), and added to the four signed 16-bit data elements of GR r_3. Both the left shift and the add operations are saturating: if the result of either the shift or the add is not representable as a signed 16-bit value, the final result is saturated. The four signed 16-bit results are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = sign_ext(x[i], 16) << count2;

        if (temp[i] > max)
            res[i] = max;
        else if (temp[i] < min)
            res[i] = min;
        else {
            res[i] = temp[i] + sign_ext(y[i], 16);
            if (res[i] > max)
                res[i] = max;
            if (res[i] < min)
                res[i] = min;
        }
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

pshr — Parallel Shift Right

Format:	(*qp*) pshr2 $r_1 = r_3, r_2$	signed_form, two_byte_form, variable_form	I5
	(*qp*) pshr2 $r_1 = r_3, count_5$	signed_form, two_byte_form, fixed_form	I6
	(*qp*) pshr2.u $r_1 = r_3, r_2$	unsigned_form, two_byte_form, variable_form	I5
	(*qp*) pshr2.u $r_1 = r_3, count_5$	unsigned_form, two_byte_form, fixed_form	I6
	(*qp*) pshr4 $r_1 = r_3, r_2$	signed_form, four_byte_form, variable_form	I5
	(*qp*) pshr4 $r_1 = r_3, count_5$	signed_form, four_byte_form, fixed_form	I6
	(*qp*) pshr4.u $r_1 = r_3, r_2$	unsigned_form, four_byte_form, variable_form	I5
	(*qp*) pshr4.u $r_1 = r_3, count_5$	unsigned_form, four_byte_form, fixed_form	I6

Description: The data elements of GR r_3 are each independently shifted to the right by the scalar shift count in GR r_2, or in the immediate field $count_5$. The high-order bits of each element are filled with either the initial value of the sign bits of the data elements in GR r_3 (arithmetic shift) or zeros (logical shift). The shift count is interpreted as unsigned. Shift counts greater than 15 (for 16-bit quantities) or 31 (for 32-bit quantities) yield all zero or all one results depending on the initial values of the sign bits of the data elements in GR r_3 and whether a signed or unsigned shift is done. The results are placed in GR r_1.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    shift_count = (variable_form ? GR[r2] : count5);
    tmp_nat = (variable_form ? GR[r2].nat : 0);

    if (two_byte_form) {                             // two_byte_form
        if (shift_count u> 16)
            shift_count = 16;
        if (unsigned_form) {                         // unsigned shift
            GR[r1]{15:0}  = shift_right_unsigned(zero_ext(GR[r3]{15:0},  16),
                                                shift_count);
            GR[r1]{31:16} = shift_right_unsigned(zero_ext(GR[r3]{31:16}, 16),
                                                shift_count);
            GR[r1]{47:32} = shift_right_unsigned(zero_ext(GR[r3]{47:32}, 16),
                                                shift_count);
            GR[r1]{63:48} = shift_right_unsigned(zero_ext(GR[r3]{63:48}, 16),
                                                shift_count);
        } else {                                     // signed shift
            GR[r1]{15:0}  = shift_right_signed(sign_ext(GR[r3]{15:0},  16),
                                                shift_count);
            GR[r1]{31:16} = shift_right_signed(sign_ext(GR[r3]{31:16}, 16),
                                                shift_count);
            GR[r1]{47:32} = shift_right_signed(sign_ext(GR[r3]{47:32}, 16),
                                                shift_count);
            GR[r1]{63:48} = shift_right_signed(sign_ext(GR[r3]{63:48}, 16),
                                                shift_count);
        }
    } else {                                         // four_byte_form
        if (shift_count > 32)
            shift_count = 32;
        if (unsigned_form) {                         // unsigned shift
            GR[r1]{31:0}  = shift_right_unsigned(zero_ext(GR[r3]{31:0},  32),
                                                shift_count);
            GR[r1]{63:32} = shift_right_unsigned(zero_ext(GR[r3]{63:32}, 32),
                                                shift_count);
        } else {                                     // signed shift
            GR[r1]{31:0}  = shift_right_signed(sign_ext(GR[r3]{31:0},  32),
                                                shift_count);
            GR[r1]{63:32} = shift_right_signed(sign_ext(GR[r3]{63:32}, 32),
                                                shift_count);
        }
    }

    GR[r1].nat = GR[r3].nat || tmp_nat;
}
```

Interruptions: Illegal Operation fault

pshradd — Parallel Shift Right and Add

Format: (qp) pshradd2 $r_1 = r_2$, $count_2$, r_3 A10

Description: The four signed 16-bit data elements of GR r_2 are each independently shifted to the right by $count_2$ bits, and added to the four signed 16-bit data elements of GR r_3. The right shift operation fills the high-order bits of each element with the initial value of the sign bits of the data elements in GR r_2. The add operation is performed with signed saturation. The four signed 16-bit results of the add are placed in GR r_1. The first operand can be shifted by 1, 2 or 3 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    x[0] = GR[r2]{15:0};     y[0] = GR[r3]{15:0};
    x[1] = GR[r2]{31:16};    y[1] = GR[r3]{31:16};
    x[2] = GR[r2]{47:32};    y[2] = GR[r3]{47:32};
    x[3] = GR[r2]{63:48};    y[3] = GR[r3]{63:48};

    max = sign_ext(0x7fff, 16);
    min = sign_ext(0x8000, 16);

    for (i = 0; i < 4; i++) {
        temp[i] = shift_right_signed(sign_ext(x[i], 16), count2);

        res[i] = temp[i] + sign_ext(y[i], 16);
        if (res[i] > max)
            res[i] = max;
        if (res[i] < min)
            res[i] = min;
    }

    GR[r1] = concatenate4(res[3], res[2], res[1], res[0]);
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

psub — Parallel Subtract

Format:

(*qp*) psub1 $r_1 = r_2, r_3$	one_byte_form, modulo_form	A9
(*qp*) psub1.sss $r_1 = r_2, r_3$	one_byte_form, sss_saturation_form	A9
(*qp*) psub1.uus $r_1 = r_2, r_3$	one_byte_form, uus_saturation_form	A9
(*qp*) psub1.uuu $r_1 = r_2, r_3$	one_byte_form, uuu_saturation_form	A9
(*qp*) psub2 $r_1 = r_2, r_3$	two_byte_form, modulo_form	A9
(*qp*) psub2.sss $r_1 = r_2, r_3$	two_byte_form, sss_saturation_form	A9
(*qp*) psub2.uus $r_1 = r_2, r_3$	two_byte_form, uus_saturation_form	A9
(*qp*) psub2.uuu $r_1 = r_2, r_3$	two_byte_form, uuu_saturation_form	A9
(*qp*) psub4 $r_1 = r_2, r_3$	four_byte_form, modulo_form	A9

Description: The sets of elements from the two source operands are subtracted, and the results placed in GR r_1.

If the difference between two elements cannot be represented in the result element and a saturation completer is specified, then saturation clipping is performed. The saturation can either be signed or unsigned, as given in Table 2-48. If the difference of two elements is larger than the upper limit value, the result is the upper limit value. If it is smaller than the lower limit value, the result is the lower limit value. The saturation limits are given in Table 2-49.

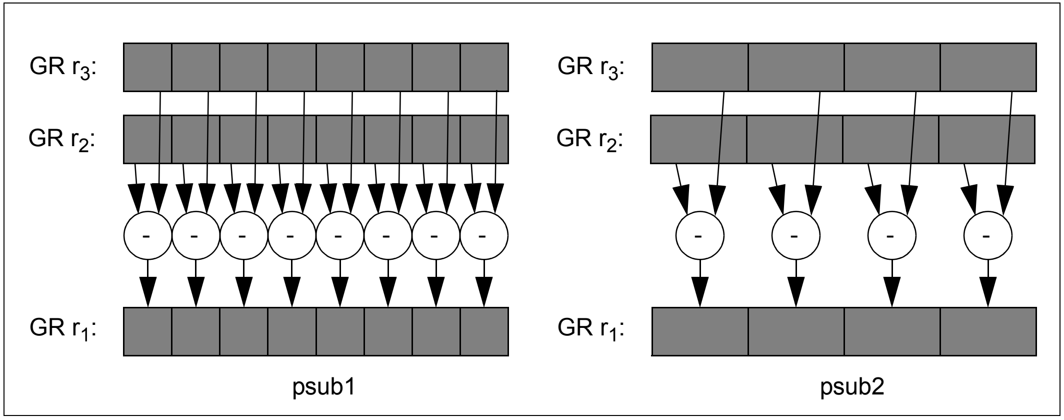
Table 2-48. Parallel Subtract Saturation Completers

Completer	Result r_1 treated as	Source r_2 treated as	Source r_3 treated as
sss	signed	signed	signed
uus	unsigned	unsigned	signed
uuu	unsigned	unsigned	unsigned

Table 2-49. Parallel Subtract Saturation Limits

Size	Element Width	Result r_1 Signed		Result r_1 Unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Figure 2-40. Parallel Subtract Examples



Operation:

```
if (PR[qp]) {
    check_target_register($r_1$);

    if (one_byte_form) {                                    // one-byte elements
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};

        if (sss_saturation_form) {                          // sss_saturation_form
            max = sign_ext(0x7f, 8);
            min = sign_ext(0x80, 8);
            for (i = 0; i < 8; i++) {
                temp[i] = sign_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uus_saturation_form) {                   // uus_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - sign_ext(y[i], 8);
            }
        } else if (uuu_saturation_form) {                   // uuu_saturation_form
            max = 0xff;
            min = 0x00;
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        } else {                                            // modulo_form
            for (i = 0; i < 8; i++) {
                temp[i] = zero_ext(x[i], 8) - zero_ext(y[i], 8);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 8; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[$r_1$] = concatenate8(temp[7], temp[6], temp[5], temp[4],
                              temp[3], temp[2], temp[1], temp[0]);
    } else if (two_byte_form) {                             // two-byte elements
        x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};

        if (sss_saturation_form) {                          // sss_saturation_form
```

```
            max = sign_ext(0x7fff, 16);
            min = sign_ext(0x8000, 16);
            for (i = 0; i < 4; i++) {
                temp[i] = sign_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uus_saturation_form) {               // uus_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - sign_ext(y[i], 16);
            }
        } else if (uuu_saturation_form) {               // uuu_saturation_form
            max = 0xffff;
            min = 0x0000;
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        } else {                                        // modulo_form
            for (i = 0; i < 4; i++) {
                temp[i] = zero_ext(x[i], 16) - zero_ext(y[i], 16);
            }
        }

        if (sss_saturation_form || uus_saturation_form ||
            uuu_saturation_form) {
            for (i = 0; i < 4; i++) {
                if (temp[i] > max)
                    temp[i] = max;
                if (temp[i] < min)
                    temp[i] = min;
            }
        }

        GR[r1] = concatenate4(temp[3], temp[2], temp[1], temp[0]);
    } else {                                            // four-byte elements
        x[0] = GR[r2]{31:0};     y[0] = GR[r3]{31:0};
        x[1] = GR[r2]{63:32};    y[1] = GR[r3]{63:32};

        for (i = 0; i < 2; i++) {                       // modulo_form
            temp[i] = zero_ext(x[i], 32) - zero_ext(y[i], 32);
        }

        GR[r1] = concatenate2(temp[1], temp[0]);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

ptc.e — Purge Translation Cache Entry

Format: (*qp*) ptc.e r_3 M47

Description: One or more translation entries are purged from the local processor's instruction and data translation cache. Translation Registers and the VHPT are not modified.

The number of translation cache entries purged is implementation specific. Some implementations may purge all levels of the translation cache hierarchy with one iteration of PTC.e, while other implementations may require several iterations to flush all levels, sets and associativities of both instruction and data translation caches. GR r_3 specifies an implementation-specific parameter associated with each iteration.

The following loop is defined to flush the entire translation cache for all processor models. Software can acquire parameters through a processor dependent layer that is accessed through a procedural interface. The selected region registers must remain unchanged during the loop.

```
disable_interrupts();
addr = base;
for (i = 0; i < count1; i++) {
    for (j = 0; j < count2; j++) {
            ptc.e(addr);
            addr += stride2;
        }
    addr += stride1;
}
enable_interrupts();
```

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat)
        register_nat_consumption_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();
    tlb_purge_translation_cache(GR[r3]);
}
```

Interruptions: Privileged Operation fault, Virtualization fault, Register NaT Consumption fault

Serialization: Software must issue a data serialization operation to ensure the purge is complete before issuing a data access or non-access reference dependent upon the purge. Software must issue instruction serialize operation before fetching an instruction dependent upon the purge.

ptc.g, ptc.ga — Purge Global Translation Cache

Format:	(*qp*) ptc.g r_3, r_2	global_form	M45
	(*qp*) ptc.ga r_3, r_2	global_alat_form	M45

Description: The instruction and data translation cache for each processor in the local TLB coherence domain are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. These entries are removed.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. See Section 4.1.1.7, "Page Sizes" on page 2:57 for details on supported page sizes for TLB purges.

Based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

`ptc.g` has release semantics and is guaranteed to be made visible after all previous data memory accesses are made visible. Serialization is still required to observe the side-effects of a translation being removed. If it is desired that the `ptc.g` become visible before any subsequent data memory accesses are made visible, a memory fence instruction (`mf`) should be executed immediately following the `ptc.g`.

`ptc.g` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The behavior of the `ptc.ga` instruction is similar to `ptc.g`. In addition to the behavior specified for `ptc.g` the `ptc.ga` instruction encodes an extra bit of information in the broadcast transaction. This information specifies the purge is due to a page remapping as opposed to a protection change or page tear down. The remote processors within the coherence domain will then take what ever additional action is necessary to make their ALAT consistent. Matching entries in the local ALAT are optionally invalidated; software must perform a local ALAT invalidation via the `invala` instruction on the processor issuing the `ptc.ga` to ensure the local ALAT is coherent.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Unless specifically supported by the processors and platform, only one global purge transaction may be issued at a time by all processors, the operation is undefined otherwise. Software is responsible for enforcing this restriction. Implementations may optionally support multiple concurrent global purge transactions. The firmware returns if implementations support this optional behavior. It also returns the maximum number of simultaneous outstanding purges allowed.

Propagation of `ptc.g` between multiple local TLB coherence domains is platform dependent, and must be handled by software. It is expected that the local TLB coherence domain covers at least the processors on the same local bus.

Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);

    if (global_alat_form) tmp_ptc_type = GLOBAL_ALAT_FORM;
    else tmp_ptc_type = GLOBAL_FORM;

    tlb_broadcast_purge(tmp_rid, tmp_va, tmp_size, tmp_ptc_type);
}
```

Interruptions:

Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault

Serialization: The broadcast purge TC is not synchronized with the instruction stream on a remote processor. Software cannot depend on any such synchronization with the instruction stream. Hardware on the remote machine cannot reload an instruction from memory or cache after acknowledging a broadcast purge TC without first retranslating the I-side access in the TLB. Hardware may continue to use a valid private copy of the instruction stream data (possibly in an I-buffer) obtained prior to acknowledging a broadcast purge TC to a page containing the i-stream data. Hardware must retranslate access to an instruction page upon an interruption or any explicit or implicit instruction serialization event (e.g., `srlz.i`, `rfi`).

Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a local data access, non-access reference, or local instruction fetch access dependent upon the purge.

ptc.l — Purge Local Translation Cache

Format: (*qp*) ptc.l r_3, r_2 M45

Description: The instruction and data translation cache of the local processor is searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. See Section 4.1.1.7, "Page Sizes" on page 2:57 for details on supported page sizes for TLB purges.

The processor ensures that all entries matching the purging parameters are removed. However, based on the processor model, the translation cache may be also purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system. This instruction ensures that all prior stores are made locally visible before the actual purge operation is performed.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);
    if (unimplemented_virtual_address(GR[r3], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    tmp_rid = RR[GR[r3]{63:61}].rid;
    tmp_va = GR[r3]{60:0};
    tmp_size = GR[r2]{7:2};
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);
    tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
    tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
}
```

Interruptions: Machine Check abort
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault

Serialization: Software must issue the appropriate data and/or instruction serialization operation to ensure the purge is completed before a data access, non-access reference, or instruction fetch access dependent upon the purge.

ptr — Purge Translation Register

Format:	(*qp*) ptr.d r_3, r_2	data_form	M45
	(*qp*) ptr.i r_3, r_2	instruction_form	M45

Description: In the data form of this instruction, the data translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the instruction translation registers are unaffected by the data form of the purge.

In the instruction form, the instruction translation registers and caches are searched for all entries whose virtual address and page size partially or completely overlap the specified purge virtual address and purge address range. All these entries are removed. Entries in the data translation registers are unaffected by the instruction form of the purge.

In addition, in both forms, the instruction and data translation cache may be purged of more translations than specified by the purge parameters up to and including removal of all entries within the translation cache.

The purge virtual address is specified by GR r_3 bits{60:0} and the purge region identifier is selected by GR r_3 bits {63:61}. GR r_2 specifies the address range of the purge as 1<<GR[r_2]{7:2} bytes in size. See Section 4.1.1.7, “Page Sizes” on page 2:57 for details on supported page sizes for TLB purges.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

This is a local operation, no purge broadcast to other processors occurs in a multiprocessor system.

As described in Section 4.1.1.2, “Translation Cache (TC)” on page 2:49, the processor may use the translation caches to cache virtual address mappings held by translation registers. The `ptr.i` and `ptr.d` instructions purge the processor’s translation registers as well as cached translation register copies that may be contained in the respective translation caches.

Operation:

```
if (PR[qp]) {
   if (PSR.cpl != 0)
      privileged_operation_fault(0);
   if (GR[r3].nat || GR[r2].nat)
      register_nat_consumption_fault(0);
   if (unimplemented_virtual_address(GR[r3], PSR.vm))
      unimplemented_data_address_fault(0);
   if (PSR.vm == 1)
      virtualization_fault();

   tmp_rid = RR[GR[r3]{63:61}].rid;
   tmp_va = GR[r3]{60:0};
   tmp_size = GR[r2]{7:2};
   tmp_va = align_to_size_boundary(tmp_va, tmp_size);

   if (data_form) {
      tlb_must_purge_dtr_entries(tmp_rid, tmp_va, tmp_size);
      tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
      tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
   } else {                                        // instruction_form
      tlb_must_purge_itr_entries(tmp_rid, tmp_va, tmp_size);
      tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
      tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
   }
}
```

Interruptions: Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault

Serialization: For the data form, software must issue a data serialization operation to ensure the purge is completed before issuing an instruction dependent upon the purge. For the instruction form, software must issue an instruction serialization operation to ensure the purge is completed before fetching an instruction dependent on that purge.

rfi — Return From Interruption

Format: rfi B8

Description: The machine context prior to an interruption is restored. PSR is restored from IPSR, IPSR is unmodified, and IP is restored from IIP. Execution continues at the bundle address loaded into the IP, and the instruction slot loaded into PSR.ri.

This instruction must be immediately followed by a stop; otherwise, operation is undefined. This instruction switches to the register bank specified by IPSR.bn. Instructions in the same instruction group that access GR16 to GR31 reference the previous register bank. Subsequent instruction groups reference the new register bank.

This instruction performs instruction serialization, which ensures:

- prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed.
- prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.
- prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache.
- subsequent instruction group fetches (including the target instruction group) are re-initiated after `rfi` completes.

The `rfi` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0. This instruction can not be predicated.

Execution of this instruction is undefined if PSR.ic or PSR.i are 1. Software must ensure that an interruption cannot occur that could modify IIP, IPSR, or IFS between when they are written and the subsequent `rfi`.

Execution of this instruction is undefined if IPSR.ic is 0 and the current register stack frame is incomplete.

This instruction does not take Lower Privilege Transfer, Taken Branch or Single Step traps.

If this instruction sets PSR.ri to 2 and the target is an MLX bundle, then an Illegal Operation fault will be taken on the target bundle.

If IPSR.is is 1, control is resumed in the IA-32 instruction set at the virtual linear address specified by IIP{31:0}. PSR.di does not inhibit instruction set transitions for this instruction. If PSR.dfh is 1 after `rfi` completes execution, a Disabled FP Register fault is raised on the target IA-32 instruction.

If IPSR.is is 1 and an Unimplemented Instruction Address trap is taken, IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

When entering the IA-32 instruction set, the size of the current stack frame is set to zero, and all stacked general registers are left in an undefined state. Software can not rely on the value of these registers across an instruction set transition. Software must ensure that BSPSTORE==BSP on entry to the IA-32 instruction set, otherwise undefined behavior may result.

If IPSR.is is 1, software must set other IPSR fields properly for IA-32 instruction set execution; otherwise processor operation is undefined. See Table 3-2, "Processor Status Register Fields" on page 2:24 for details.

Software must issue a `mf` instruction before this instruction if memory ordering is required between IA-32 processor-consistent and Itanium unordered memory references. The processor does not ensure Itanium-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instructions.

Software must ensure the code segment descriptor and selector are loaded before issuing this instruction. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA_32_Exception(GPFault) exception is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if IIP is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction.
EFLAG.rf and PSR.id are unmodified until the successful completion of the target IA-32 instruction. PSR.da, PSR.dd, PSR.ia and PSR.ed are cleared to zero before the target IA-32 instruction begins execution.

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT state across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored.

Operation:

```
if (!followed_by_stop())
    undefined_behavior();

unimplemented_address = 0;
if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (PSR.vm == 1)
    virtualization_fault();

taken_rfi = 1;

PSR = CR[IPSR];
if (CR[IPSR].is == 1) {         //resume IA-32 instruction set
    if (CR[IPSR].ic == 0 || CR[IPSR].dt == 0 ||
        CR[IPSR].mc == 1 || CR[IPSR].it == 0)
        undefined_behavior();
    tmp_IP = CR[IIP];
    if (!impl_uia_fault_supported() &&
        ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP))))
        unimplemented_address = 1;
                                //compute effective instruction pointer
    EIP{31:0} = CR[IIP]{31:0} - AR[CSD].Base;
                                //force zero-sized restored frame
    rse_restore_frame(0, 0, CFM.sof);
    CFM.sof = 0;
    CFM.sol = 0;
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
    //The register stack engine is disabled during IA-32
    //instruction set execution.
} else {                         //return to Itanium instruction set
    tmp_IP = CR[IIP] & ~0xf;
    slot = CR[IPSR].ri;
    if ((CR[IPSR].it && unimplemented_virtual_address(tmp_IP, IPSR.vm))
        || (!CR[IPSR].it && unimplemented_physical_address(tmp_IP)))
        unimplemented_address = 1;
    if (CR[IFS].v) {
        tmp_growth = -CFM.sof;
        alat_frame_update(-CR[IFS].ifm.sof, 0);
        rse_restore_frame(CR[IFS].ifm.sof, tmp_growth, CFM.sof);
        CFM = CR[IFS].ifm;
    }
    rse_enable_current_frame_load();
}
IP = tmp_IP;
instruction_serialize();
if (unimplemented_address)
    unimplemented_instruction_address_trap(0, tmp_IP);
```

Interruptions: Privileged Operation fault

Unimplemented Instruction Address trap

Virtualization fault

Additional Faults on IA-32 target instructions
IA_32_Exception(GPFault)
Disabled FP Reg Fault if PSR.dfh is 1

Serialization: An implicit instruction and data serialization operation is performed.

rsm — Reset System Mask

Format: (*qp*) rsm *imm*$_{24}$ M44

Description: The complement of the *imm*$_{24}$ operand is ANDed with the system mask (PSR{23:0}) and the result is placed in the system mask. See Section 3.3.2, "Processor Status Register (PSR)" on page 2:23.

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

When the current privilege level is zero (PSR.cpl is 0), an `rsm` instruction whose mask includes PSR.i may cause external interrupts to be disabled for an implementation-dependent number of instructions, even if the qualifying predicate for the `rsm` instruction is false. Architecturally, the extents of this external interrupt disabling "window" are defined as follows:

- External interrupts may be disabled for any instructions in the same instruction group as the `rsm`, including those that precede the `rsm` in sequential program order, regardless of the value of the qualifying predicate of the `rsm` instruction.
- If the qualifying predicate of the `rsm` is true, then external interrupts are disabled immediately following the `rsm` instruction.
- If the qualifying predicate of the `rsm` is false, then external interrupts may be disabled until the next data serialization operation that follows the `rsm` instruction.

The external interrupt disable window is guaranteed to be no larger than defined by the above criteria, but it may be smaller, depending on the processor implementation.

When the current privilege level is non-zero (PSR.cpl is not 0), an `rsm` instruction whose mask includes PSR.i may briefly disable external interrupts, regardless of the value of the qualifying predicate of the `rsm` instruction. However, processor implementations guarantee that non-privileged code cannot lock out external interrupts indefinitely (e.g., via an arbitrarily long sequence of `rsm` instructions with zero-valued qualifying predicates).

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (PSR.vm == 1)
        virtualization_fault();

    if (imm24{1})     PSR{1} = 0;)      // be
    if (imm24{2})     PSR{2} = 0;)      // up
    if (imm24{3})     PSR{3} = 0;)      // ac
    if (imm24{4})     PSR{4} = 0;)      // mfl
    if (imm24{5})     PSR{5} = 0;)      // mfh
    if (imm24{13})    PSR{13} = 0;)     // ic
    if (imm24{14})    PSR{14} = 0;)     // i
    if (imm24{15})    PSR{15} = 0;)     // pk
    if (imm24{17})    PSR{17} = 0;)     // dt
    if (imm24{18})    PSR{18} = 0;)     // dfl
    if (imm24{19})    PSR{19} = 0;)     // dfh
    if (imm24{20})    PSR{20} = 0;)     // sp
    if ($imm_{24}${21})    PSR{21} = 0;)     // pp
    if ($imm_{24}${22})    PSR{22} = 0;)     // di
    if ($imm_{24}${23})    PSR{23} = 0;)     // si
}
```

Interruptions: Privileged Operation fault
Virtualization fault
Reserved Register/Field fault

Privileged Operation fault	Virtualization fault
Reserved Register/Field fault	

Serialization: Software must use a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits – except the PSR.i bit. The PSR.i bit is implicitly serialized and the processor ensures that external interrupts are masked by the time the next instruction executes.

rum — Reset User Mask

Format: (*qp*) rum *imm*$_{24}$ M44

Description: The complement of the *imm*$_{24}$ operand is ANDed with the user mask (PSR{5:0}) and the result is placed in the user mask. See Section 3.3.2, “Processor Status Register (PSR)” on page 2:23.

PSR.up is only cleared if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})     PSR{1} = 0;)      // be
    if (imm24{2} && PSR.sp == 0)        //non-secure perf monitor
                      PSR{2} = 0;)      // up
    if (imm24{3})     PSR{3} = 0;)      // ac
    if (imm24{4})     PSR{4} = 0;)      // mfl
    if (imm24{5})     PSR{5} = 0;)      // mfh
}
```

Interruptions: Reserved Register/Field fault

Serialization: All user mask modifications are observed by the next instruction group.

setf — Set Floating-point Value, Exponent, or Significand

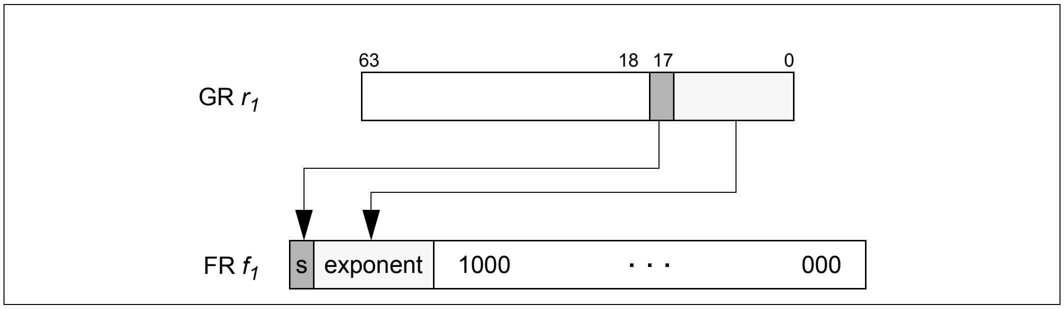
Format:

(*qp*) setf.s $f_1 = r_2$	single_form	M18
(*qp*) setf.d $f_1 = r_2$	double_form	M18
(*qp*) setf.exp $f_1 = r_2$	exponent_form	M18
(*qp*) setf.sig $f_1 = r_2$	significand_form	M18

Description: In the single and double forms, GR r_2 is treated as a single precision (in the single_form) or double precision (in the double_form) memory representation, converted into floating-point register format, and placed in FR f_1, as shown in Figure 5-4 and Figure 5-5 on page 1:93, respectively.

In the exponent_form, bits 16:0 of GR r_2 are copied to the exponent field of FR f_1 and bit 17 of GR r_2 is copied to the sign bit of FR f_1. The significand field of FR f_1 is set to one (0x800...000).

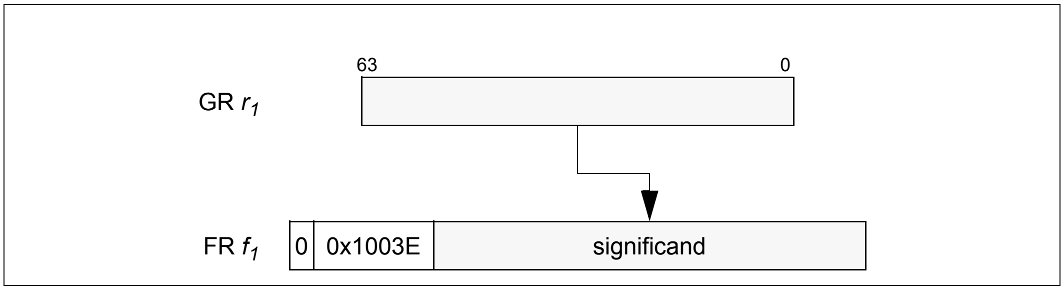
Figure 2-41. Function of setf.exp



In the significand_form, the value in GR r_2 is copied to the significand field of FR f_1.

The exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Figure 2-42. Function of setf.sig



For all forms, if the NaT bit corresponding to r_2 is equal to 1, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register(f1);
    if (tmp_isrcode = fp_reg_disabled(f1, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (!GR[r2].nat) {
        if (single_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 4, 0);
        else if (double_form)
            FR[f1] = fp_mem_to_fr_format(GR[r2], 8, 0);
        else if (significand_form) {
            FR[f1].significand = GR[r2];
            FR[f1].exponent = FP_INTEGER_EXP;
            FR[f1].sign = 0;
        } else {                                      // exponent_form
            FR[f1].significand = 0x8000000000000000;
            FR[f1].exp = GR[r2]{16:0};
            FR[f1].sign = GR[r2]{17};
        }
    } else
        FR[f1] = NATVAL;

    fp_update_psr(f1);
}
```

Interruptions: Illegal Operation fault   Disabled Floating-point Register fault

shl — Shift Left

Format: (*qp*) shl r_1= r_2, r_3 I7

(*qp*) shl r_1 = r_2, $count_6$ pseudo-op of: (*qp*) dep.z r_1 = r_2, $count_6$, 64-$count_6$

Description: The value in GR r_2 is shifted to the left, with the vacated bit positions filled with zeroes, and placed in GR r_1. The number of bit positions to shift is specified by the value in GR r_3 or by an immediate value $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_3 is greater than 63, then the result is all zeroes.

See "dep — Deposit" on page 3:51 for the immediate form.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    count = GR[r3];
    GR[r1] = (count > 63) ? 0: GR[r2] << count;

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

shladd — Shift Left and Add

Format: (*qp*) shladd $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then added to the second source operand and the result placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = (GR[r2] << count2) + GR[r3];
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

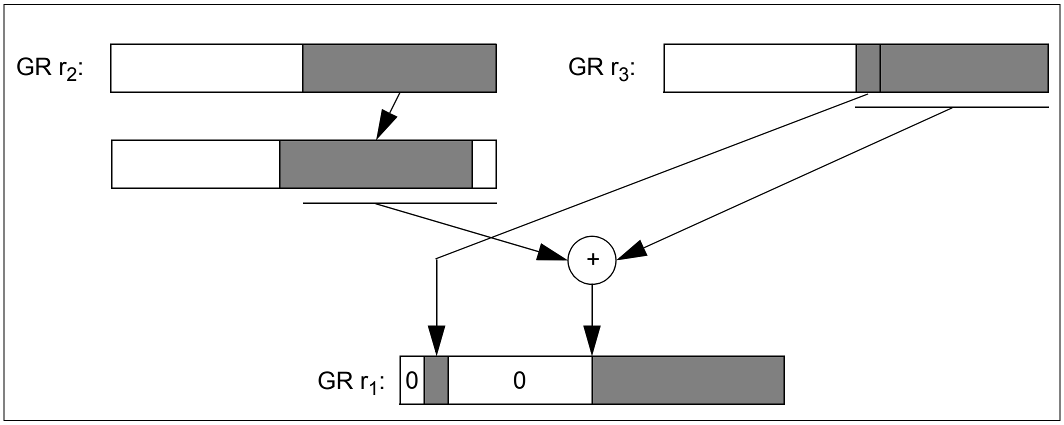
Interruptions: Illegal Operation fault

shladdp4 — Shift Left and Add Pointer

Format: (qp) shladdp4 $r_1 = r_2, count_2, r_3$ A2

Description: The first source operand is shifted to the left by $count_2$ bits and then is added to the second source operand. The upper 32 bits of the result are forced to zero, and then bits {31:30} of GR r_3 are copied to bits {62:61} of the result. This result is placed in GR r_1. The first operand can be shifted by 1, 2, 3, or 4 bits.

Figure 2-43. Shift Left and Add Pointer



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_res = (GR[r2] << count2) + GR[r3];
    tmp_res = zero_ext(tmp_res{31:0}, 32);
    tmp_res{62:61} = GR[r3]{31:30};
    GR[r1] = tmp_res;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

shr — Shift Right

Format:

(*qp*) shr $r_1 = r_3, r_2$	signed_form	I5
(*qp*) shr.u $r_1 = r_3, r_2$	unsigned_form	I5
(*qp*) shr $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr $r_1 = r_3, count_6$, 64-$count_6$	
(*qp*) shr.u $r_1 = r_3, count_6$	pseudo-op of: (*qp*) extr.u $r_1 = r_3, count_6$, 64-$count_6$	

Description: The value in GR r_3 is shifted to the right and placed in GR r_1. In the signed_form the vacated bit positions are filled with bit 63 of GR r_3; in the unsigned_form the vacated bit positions are filled with zeroes. The number of bit positions to shift is specified by the value in GR r_2 or by an immediate value $count_6$. The shift count is interpreted as an unsigned number. If the value in GR r_2 is greater than 63, then the result is all zeroes (for the unsigned_form, or if bit 63 of GR r_3 was 0) or all ones (for the signed_form if bit 63 of GR r_3 was 1).

If the .u completer is specified, the shift is unsigned (logical), otherwise it is signed (arithmetic).

See "extr — Extract" on page 3:54 for the immediate forms.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (signed_form) {
        count = (GR[r2] > 63) ? 63 : GR[r2];
        GR[r1] = shift_right_signed(GR[r3], count);
    } else {
        count = GR[r2];
        GR[r1] = (count > 63) ? 0 : shift_right_unsigned(GR[r3], count);
    }

    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

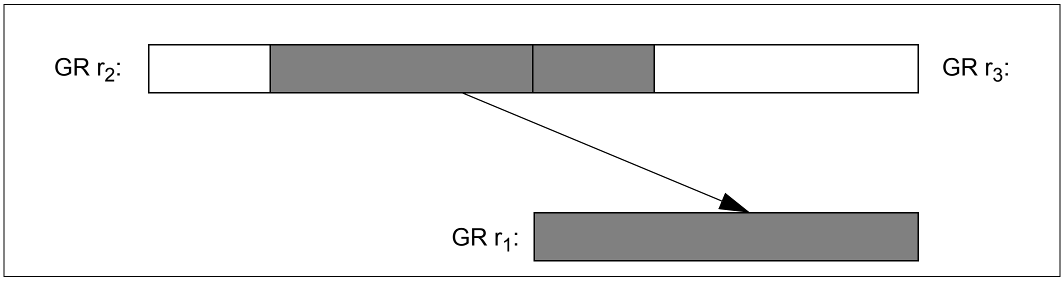
shrp — Shift Right Pair

Format: (qp) shrp $r_1 = r_2, r_3, count_6$ I10

Description: The two source operands, GR r_2 and GR r_3, are concatenated to form a 128-bit value and shifted to the right $count_6$ bits. The least-significant 64 bits of the result are placed in GR r_1.

The immediate value $count_6$ can be any number in the range 0 to 63.

Figure 2-44. Shift Right Pair



Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    temp1 = shift_right_unsigned(GR[r3], count6);
    temp2 = GR[r2] << (64 - count6);
    GR[r1] = zero_ext(temp1, 64 - count6) | temp2;
    GR[r1].nat = GR[r2].nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

srlz — Serialize

Format:	(*qp*) srlz.i	instruction_form	M24
	(*qp*) srlz.d	data_form	M24

Description: Instruction serialization (`srlz.i`) ensures:

- prior modifications to processor register resources that affect fetching of subsequent instruction groups are observed,
- prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed,
- prior memory synchronization (`sync.i`) operations have taken effect on the local processor instruction cache,
- subsequent instruction group fetches are re-initiated after `srlz.i` completes.

The `srlz.i` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized. Operations dependent on the serialization must be in an instruction group after the instruction group containing the `srlz.i`.

Data serialization (`srlz.d`) ensures:

- prior modifications to processor register resources that affect subsequent execution or data memory accesses are observed.

The `srlz.d` instruction must be in an instruction group after the instruction group containing the operation that is to be serialized. Operations dependent on the serialization must follow the `srlz.d`, but they can be in the same instruction group as the `srlz.d`.

A `srlz` cannot be used to stall processor data memory references until prior data memory references, or memory fences are visible or "accepted" by the external platform.

The following processor resources require a serialize to ensure side-effects are observed; CRs, PSR, DBRs, IBRs, PMDs, PMCs, RRs, PKRs, TRs and TCs (refer to Section 3.2, "Serialization" on page 2:17 for details).

Operation:

```
if (PR[qp]) {
    if (instruction_form)
        instruction_serialize();
    else       // data_form
        data_serialize();
}
```

Interruptions: None

ssm — Set System Mask

Format: (*qp*) ssm *imm*$_{24}$ M44

Description: The *imm*$_{24}$ operand is ORed with the system mask (PSR{23:0}) and the result is placed in the system mask. See Section 3.3.2, “Processor Status Register (PSR)” on page 2:23.

The PSR system mask can only be written at the most privileged level, and when PSR.vm is 0.

The contents of the interruption resources (that are overwritten when the PSR.ic bit is 1), are undefined if an interruption occurs between the enabling of the PSR.ic bit and a subsequent instruction serialize operation.

Operation:

```
if (PR[qp]) {
    if (PSR.cpl != 0)
        privileged_operation_fault(0);

    if (is_reserved_field(PSR_TYPE, PSR_SM, imm24))
        reserved_register_field_fault();

    if (PSR.vm == 1)
        virtualization_fault();

    if (imm24{1})     PSR{1} = 1;)      // be
    if (imm24{2})     PSR{2} = 1;)      // up
    if (imm24{3})     PSR{3} = 1;)      // ac
    if (imm24{4})     PSR{4} = 1;)      // mfl
    if (imm24{5})     PSR{5} = 1;)      // mfh
    if (imm24{13})    PSR{13} = 1;)     // ic
    if (imm24{14})    PSR{14} = 1;)     // i
    if (imm24{15})    PSR{15} = 1;)     // pk
    if (imm24{17})    PSR{17} = 1;)     // dt
    if (imm24{18})    PSR{18} = 1;)     // dfl
    if (imm24{19})    PSR{19} = 1;)     // dfh
    if (imm24{20})    PSR{20} = 1;)     // sp
    if (imm24{21})    PSR{21} = 1;)     // pp
    if (imm24{22})    PSR{22} = 1;)     // di
    if (imm24{23})    PSR{23} = 1;)     // si
}
```

Interruptions: Privileged Operation fault
Reserved Register/Field fault
Virtualization fault

Serialization: Software must issue a data serialize or instruction serialize operation before issuing instructions dependent upon the altered PSR bits from the `ssm` instruction. Unlike with the `rsm` instruction, setting the PSR.i bit is not treated specially. Refer to Section 3.2, “Serialization” on page 2:17 for a description of serialization.

st — Store

Format:			
	(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2	normal_form, no_base_update_form	M6
	(qp) st*sz*.*sttype*.*sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
	(qp) st16.*sttype*.*sthint* [r_3] = r_2, ar.csd	sixteen_byte_form, no_base_update_form	M6
	(qp) st8.spill.*sthint* [r_3] = r_2	spill_form, no_base_update_form	M6
	(qp) st8.spill.*sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

Description: A value consisting of the least significant *sz* bytes of the value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The values of the *sz* completer are given in Table 2-32 on page 3:151. The *sttype* completer specifies special store operations, which are described in Table 2-50. If the NaT bit corresponding to GR r_3 is 1, or in sixteen_byte_form or normal_form, if the NaT bit corresponding to GR r_2 is 1, a Register NaT Consumption fault is taken.

In the sixteen_byte_form, two 8-byte values are stored as a single, 16-byte atomic memory write. The value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The value in the Compare and Store Data application register (AR[CSD]) is written to memory starting at the address specified by the value in GR r_3 plus 8.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR r_2 is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair. See Section 4.4.4, "Control Speculation" on page 1:60 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where r_2 and r_3 specify the same register). Base register update is not supported for the `st16` instruction.

Table 2-50. Store Types

sttype Completer	Interpretation	Special Store Operation
none	Normal store	
rel	Ordered store	An ordered store is performed with release semantics.

For more details on ordered stores see Section 4.4.7, "Memory Access Ordering" on page 1:73.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-51. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69.

Hardware support for `st16` instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `st16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen_byte_form, Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See Section 3.1.11, “Processor Identification Registers” on page 1:34 for details.

Table 2-51. Store Hints

sthint Completer	Interpretation
none	Temporal locality, level 1
nta	Non-temporal locality, all levels

Operation:

```
if (PR[qp]) {
    size = spill_form ? 8 : (sixteen_byte_form ? 16 : sz);
    itype = WRITE;
    if (size == 16) itype |= UNCACHE_OPT;
    otype = (sttype == 'rel') ? RELEASE : UNORDERED;

    if (sixteen_byte_form && !instruction_implemented(ST16))
        illegal_operation_fault();
    if (imm_base_update_form)
        check_target_register(r3);
    if (GR[r3].nat || ((sixteen_byte_form || normal_form) && GR[r2].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                            &tmp_unused);
    if (spill_form && GR[r2].nat) {
        natd_gr_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    }
    else {
        if (sixteen_byte_form)
            mem_write16(GR[r2], AR[CSD], paddr, UM.be, mattr, otype, sthint);
        else
            mem_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    }

    if (spill_form) {
        bit_pos = GR[r3]{8:3};
        AR[UNAT]{bit_pos} = GR[r2].nat;
    }

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
        mem_implicit_prefetch(GR[r3], sthint, WRITE);
    }
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault

Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Unaligned Data Reference fault
Unsupported Data Reference fault

stf — Floating-point Store

Format:

(*qp*) stf*fsz*.*sthint* [r_3] = f_2	normal_form, no_base_update_form	M13
(*qp*) stf*fsz*.*sthint* [r_3] = f_2, imm_9	normal_form, imm_base_update_form	M10
(*qp*) stf8.*sthint* [r_3] = f_2	integer_form, no_base_update_form	M13
(*qp*) stf8.*sthint* [r_3] = f_2, imm_9	integer_form, imm_base_update_form	M10
(*qp*) stf.spill.*sthint* [r_3] = f_2	spill_form, no_base_update_form	M13
(*qp*) stf.spill.*sthint* [r_3] = f_2, imm_9	spill_form, imm_base_update_form	M10

Description: A value, consisting of *fsz* bytes, is generated from the value in FR f_2 and written to memory starting at the address specified by the value in GR r_3. In the normal_form, the value in FR f_2 is converted to the memory format and then stored. In the integer_form, the significand of FR f_2 is stored. The values of the *fsz* completer are given in Table 2-35 on page 3:157. In the normal_form or the integer_form, if the NaT bit corresponding to GR r_3 is 1 or if FR f_2 contains NaTVal, a Register NaT Consumption fault is taken. See Section 5.1, "Data Types and Formats" on page 1:85 for details on conversion from floating-point register format.

In the spill_form, a 16-byte value from FR f_2 is stored without conversion. This instruction is used for spilling a register. See Section 4.4.4, "Control Speculation" on page 1:60 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-51 on page 3:252. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69.

Hardware support for `stfe` (10-byte) instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `stfe` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

Operation:

```
if (PR[qp]) {
    if (imm_base_update_form)
        check_target_register($r_3$);
    if (tmp_isrcode = fp_reg_disabled($f_2$, 0, 0, 0))
        disabled_fp_register_fault(tmp_isrcode, WRITE);

    if (GR[$r_3$].nat || (!spill_form && (FR[$f_2$] == NATVAL)))
        register_nat_consumption_fault(WRITE);

    size = spill_form ? 16 : (integer_form ? 8 : fsz);
    itype = WRITE;
    if (size == 10) itype |= UNCACHE_OPT;

    paddr = tlb_translate(GR[$r_3$], size, itype, PSR.cpl, &mattr, &tmp_unused);
    val = fp_fr_to_mem_format(FR[$f_2$], size, integer_form);
    mem_write(val, paddr, size, UM.be, mattr, UNORDERED, sthint);

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[$r_3$] = GR[$r_3$] + sign_ext($imm_9$, 9);
        GR[$r_3$].nat = 0;
        mem_implicit_prefetch(GR[$r_3$], sthint, WRITE);
    }
}
```

Interruptions:

Illegal Operation fault
Disabled Floating-point Register fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

sub — Subtract

Format:

(*qp*) sub $r_1 = r_2, r_3$	register_form	A1
(*qp*) sub $r_1 = r_2, r_3$, 1	minus1_form, register_form	A1
(*qp*) sub $r_1 = imm_8, r_3$	imm8_form	A3

Description: The second source operand (and an optional constant 1) are subtracted from the first operand and the result placed in GR r_1. In the register form the first operand is GR r_2; in the immediate form the first operand is taken from the sign-extended imm_8 encoding field.

The minus1_form is available only in the register_form (although the equivalent effect can be achieved by adjusting the immediate).

Operation:

```
if (PR[$qp$]) {
    check_target_register($r_1$);

    tmp_src = (register_form ? GR[$r_2$] : sign_ext($imm_8$, 8));
    tmp_nat = (register_form ? GR[$r_2$].nat : 0);

    if (minus1_form)
        GR[$r_1$] = tmp_src - GR[$r_3$] - 1;
    else
        GR[$r_1$] = tmp_src - GR[$r_3$];

    GR[$r_1$].nat = tmp_nat || GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

sum — Set User Mask

Format: (*qp*) sum *imm*$_{24}$ M44

Description: The *imm*$_{24}$ operand is ORed with the user mask (PSR{5:0}) and the result is placed in the user mask. See Section 3.3.2, “Processor Status Register (PSR)” on page 2:23.

PSR.up can only be set if the secure performance monitor bit (PSR.sp) is zero. Otherwise PSR.up is not modified.

Operation:

```
if (PR[qp]) {
    if (is_reserved_field(PSR_TYPE, PSR_UM, imm24))
        reserved_register_field_fault();

    if (imm24{1})      PSR{1} = 1;)      // be
    if (imm24{2} && PSR.sp == 0)         //non-secure perf monitor
                       PSR{2} = 1;)      // up
    if (imm24{3})      PSR{3} = 1;)      // ac
    if (imm24{4})      PSR{4} = 1;)      // mfl
    if (imm24{5})      PSR{5} = 1;)      // mfh
}
```

Interruptions: Reserved Register/Field fault

Serialization: All user mask modifications are observed by the next instruction group.

sxt — Sign Extend

Format: (*qp*) sxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is sign extended from the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-52.

Table 2-52. *xsz* Mnemonic Values

xsz Mnemonic	Bit Position
1	7
2	15
4	31

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = sign_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

sync — Memory Synchronization

Format: (*qp*) sync.i M24

Description: `sync.i` ensures that when previously initiated Flush Cache (`fc`, `fc.i`) operations issued by the local processor become visible to local data memory references, prior Flush Cache operations are also observed by the local processor instruction fetch stream. `sync.i` also ensures that at the time previously initiated Flush Cache (`fc`, `fc.i`) operations are observed on a remote processor by data memory references they are also observed by instruction memory references on the remote processor. `sync.i` is ordered with respect to all cache flush operations as observed by another processor. A `sync.i` and a previous `fc` must be in separate instruction groups. If semantically required, the programmer must explicitly insert ordered data references (acquire, release or fence type) to appropriately constrain `sync.i` (and hence `fc` and `fc.i`) visibility to the data stream on other processors.

`sync.i` is used to maintain an ordering relationship between instruction and data caches on local and remote processors. An instruction serialize operation must be used to ensure synchronization initiated by `sync.i` on the local processor has been observed by a given point in program execution.

An example of self-modifying code (local processor):

```
    st  [L1] = data   //store into local instruction stream
    fc.i L1           //flush stale datum from instruction/data cache
    ;;                //require instruction boundary between fc.i and sync.i
    sync.i            //ensure local and remote data/inst caches
                      //are synchronized
    ;;
    srlz.i            //ensure sync has been observed by the local processor,
    ;;                //ensure subsequent instructions observe
                      //modified memory
L1: target            //instruction modified
```

Operation:

```
if (PR[qp]) {
    instruction_synchronize();
}
```

Interruptions: None

tak — Translation Access Key

Format: (*qp*) tak $r_1 = r_3$ M46

Description: The protection key for a given virtual address is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found, the protection key of the translation is placed in bits 31:8 of GR r_1. If a matching present translation is not found or if an unimplemented virtual address is specified by GR r_3, the value 1 is returned.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, the value 1 is returned.

A translation with the NaTPage attribute is not treated differently and returns its key field.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TAK;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    if (PSR.vm == 1)
        virtualization_fault();

    GR[r1] = tlb_access_key(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Virtualization fault

tbit — Test Bit

Format: (*qp*) tbit.*trel*.*ctype* p_1, p_2 = r_3, pos_6 I16

Description: The bit specified by the pos_6 immediate is selected from GR r_3. The selected bit forms a single bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:39.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-53. Test Bit Relations for Normal and unc tbits

trel	Test Relation	Pseudo-op of
nz	selected bit == 1	z $p_1 \leftrightarrow p_2$
z	selected bit == 0	

Table 2-54. Test Bit Relations for Parallel tbits

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if ($p_1$ == $p_2$)
        illegal_operation_fault();

    if (trel == ‘nz’)                                         // ‘nz’ - test for 1
        tmp_rel = GR[$r_3$]{$pos_6$};
    else                                                      // ‘z’ - test for 0
        tmp_rel = !GR[$r_3$]{$pos_6$};

    switch (ctype) {
        case ‘and’:                                           // and-type compare
            if (GR[$r_3$].nat || !tmp_rel) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            }
            break;
        case ‘or’:                                            // or-type compare
            if (!GR[$r_3$].nat && tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 1;
            }
            break;
        case ‘or.andcm’:                                      // or.andcm-type compare
            if (!GR[$r_3$].nat && tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 0;
            }
            break;
        case ‘unc’:                                           // unc-type compare
        default:                                              // normal compare
            if (GR[$r_3$].nat) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            } else {
                PR[$p_1$] = tmp_rel;
                PR[$p_2$] = !tmp_rel;
            }
            break;
    }
} else {
    if (ctype == ‘unc’) {
        if ($p_1$ == $p_2$)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

Interruptions: Illegal Operation fault

tf — Test Feature

Format: (*qp*) tf.*trel*.*ctype* p_1, p_2 = imm_5 I30

Description: The imm_5 value (in the range of 32-63) selects the feature bit defined in Table 2-57 to be tested from the features vector in CPUID[4]. See Section 3.1.11, "Processor Identification Registers" on page 1:34 for details on CPUID registers. The selected bit forms a single-bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:39.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-55. Test Feature Relations for Normal and unc tf

trel	Test Relation	Pseudo-op of	
nz	selected feature available	z	$p_1 \leftrightarrow p_2$
z	selected feature unavailable		

Table 2-56. Test Feature Relations for Parallel tf

trel	Test Relation
nz	selected feature available
z	selected feature unavailable

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set or the compare type is unc.

Table 2-57. Test Feature Features Assignment

imm_5	Feature Symbol	Feature
32	@clz	`clz` feature
33	@mpy	`mpy4`, `mpyshl4` feature
34 - 63	*none*	Not currently defined

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_rel = (psr.vm && pal_vp_env_enabled() && VAC.a_tf) ?
                vcpuid[4]{imm5} : cpuid[4]{imm5};

    if (trel == 'z')                                // 'z' - test for 0, not 1
        tmp_rel = !tmp_rel;

    switch (ctype) {
        case 'and':                                 // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                                  // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                            // or.andcm-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                                 // unc-type compare
        default:                                    // normal compare
            PR[p1] = tmp_rel;
            PR[p2] = !tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

thash — Translation Hashed Entry Address

Format: (*qp*) thash $r_1 = r_3$ M46

Description: A Virtual Hashed Page Table (VHPT) entry address is generated based on the specified virtual address and the result is placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

If `thash` is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

When the processor is configured to use the region-based short format VHPT (PTA.vf=0), the value returned by `thash` is defined by the architected short format hash function. See Section 4.1.5.3, "Region-based VHPT Short Format" on page 2:63.

When the processor is configured to use the long format VHPT (PTA.vf=1), `thash` performs an implementation-specific long format hash function on the virtual address to generate a hash index into the long format VHPT.

In the long format, a translation in the VHPT must be uniquely identified by its hash index generated by this instruction and the hash tag produced from the `ttag` instruction.

The hash function must use all implemented region bits and only virtual address bits {60:0} to determine the offset into the VHPT. Virtual address bits {63:61} are used only by the short format hash to determine the region of the VHPT.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

This instruction can only be executed when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_hash(tmp_vr, tmp_va, RR[tmp_vr].rid,
                               RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault   Virtualization fault

tnat — Test NaT

Format: (*qp*) tnat.*trel*.*ctype* p_1, p_2 = r_3 I17

Description: The NaT bit from GR r_3 forms a single bit result, either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations, p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:39.

The *trel* completer values .nz and .z indicate non-zero and zero sense of the test. For normal and unc types, only the .z value is directly implemented in hardware; the .nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-58. Test NaT Relations for Normal and unc tnats

trel	Test Relation	Pseudo-op of	
nz	selected bit == 1	z	$p_1 \leftrightarrow p_2$
z	selected bit == 0		

Table 2-59. Test NaT Relations for Parallel tnats

trel	Test Relation
nz	selected bit == 1
z	selected bit == 0

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set, or if the compare type is unc.

Operation:

```
if (PR[qp]) {
    if ($p_1$ == $p_2$)
        illegal_operation_fault();

    if (trel == 'nz')                              // 'nz' - test for 1
        tmp_rel = GR[$r_3$].nat;
    else                                           // 'z' - test for 0
        tmp_rel = !GR[$r_3$].nat;

    switch (ctype) {
        case 'and':                                // and-type compare
            if (!tmp_rel) {
                PR[$p_1$] = 0;
                PR[$p_2$] = 0;
            }
            break;
        case 'or':                                 // or-type compare
            if (tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 1;
            }
            break;
        case 'or.andcm':                           // or.andcm-type compare
            if (tmp_rel) {
                PR[$p_1$] = 1;
                PR[$p_2$] = 0;
            }
            break;
        case 'unc':                                // unc-type compare
        default:                                   // normal compare
            PR[$p_1$] = tmp_rel;
            PR[$p_2$] = !tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if ($p_1$ == $p_2$)
            illegal_operation_fault();
        PR[$p_1$] = 0;
        PR[$p_2$] = 0;
    }
}
```

Interruptions: Illegal Operation fault

tpa — Translate to Physical Address

Format: (*qp*) tpa $r_1 = r_3$ M46

Description: The physical address for the virtual address specified by GR r_3 is obtained and placed in GR r_1.

When PSR.dt is 1, the DTLB and the VHPT are searched for the virtual address specified by GR r_3 and the region register indexed by GR r_3 bits {63:61}. If a matching present translation is found the physical address of the translation is placed in GR r_1. If a matching present translation is not found the appropriate TLB fault is taken.

When PSR.dt is 0, only the DTLB is searched, because the VHPT walker is disabled. If no matching present translation is found in the DTLB, an Alternate Data TLB fault is raised if psr.ic is one or a Data Nested TLB fault is raised if psr.ic is zero.

If this instruction faults, then it will set the non-access bit in the ISR. The ISR read and write bits are not set.

This instruction can only be executed at the most privileged level, and when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    itype = NON_ACCESS|TPA;
    check_target_register(r1);

    if (PSR.cpl != 0)
        privileged_operation_fault(itype);

    if (GR[r3].nat)
        register_nat_consumption_fault(itype);

    GR[r1] = tlb_translate_nonaccess(GR[r3], itype);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Virtualization fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault

ttag — Translation Hashed Entry Tag

Format: (qp) ttag r_1 = r_3 M46

Description: A tag used for matching during searches of the long format Virtual Hashed Page Table (VHPT) is generated and placed in GR r_1. The virtual address is specified by GR r_3 and the region register selected by GR r_3 bits {63:61}.

If `ttag` is given a NaT input argument or an unimplemented virtual address as an input, the resulting target register value is undefined, and its NaT bit is set to one.

The tag generation function generates an implementation-specific long format VHPT tag. The tag generation function must use all implemented region bits and only virtual address bits {60:0}. PTA.vf is ignored by this instruction.

A translation in the long format VHPT must be uniquely identified by its hash index generated by the `thash` instruction and the tag produced from this instruction.

This instruction must be implemented on all processor models, even processor models that do not implement a VHPT walker.

This instruction can only be executed when PSR.vm is 0.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (PSR.vm == 1)
        virtualization_fault();

    if (GR[r3].nat || unimplemented_virtual_address(GR[r3], PSR.vm)) {
        GR[r1] = undefined();
        GR[r1].nat = 1;
    } else {
        tmp_vr = GR[r3]{63:61};
        tmp_va = GR[r3]{60:0};
        GR[r1] = tlb_vhpt_tag(tmp_va, RR[tmp_vr].rid, RR[tmp_vr].ps);
        GR[r1].nat = 0;
    }
}
```

Interruptions: Illegal Operation fault Virtualization fault

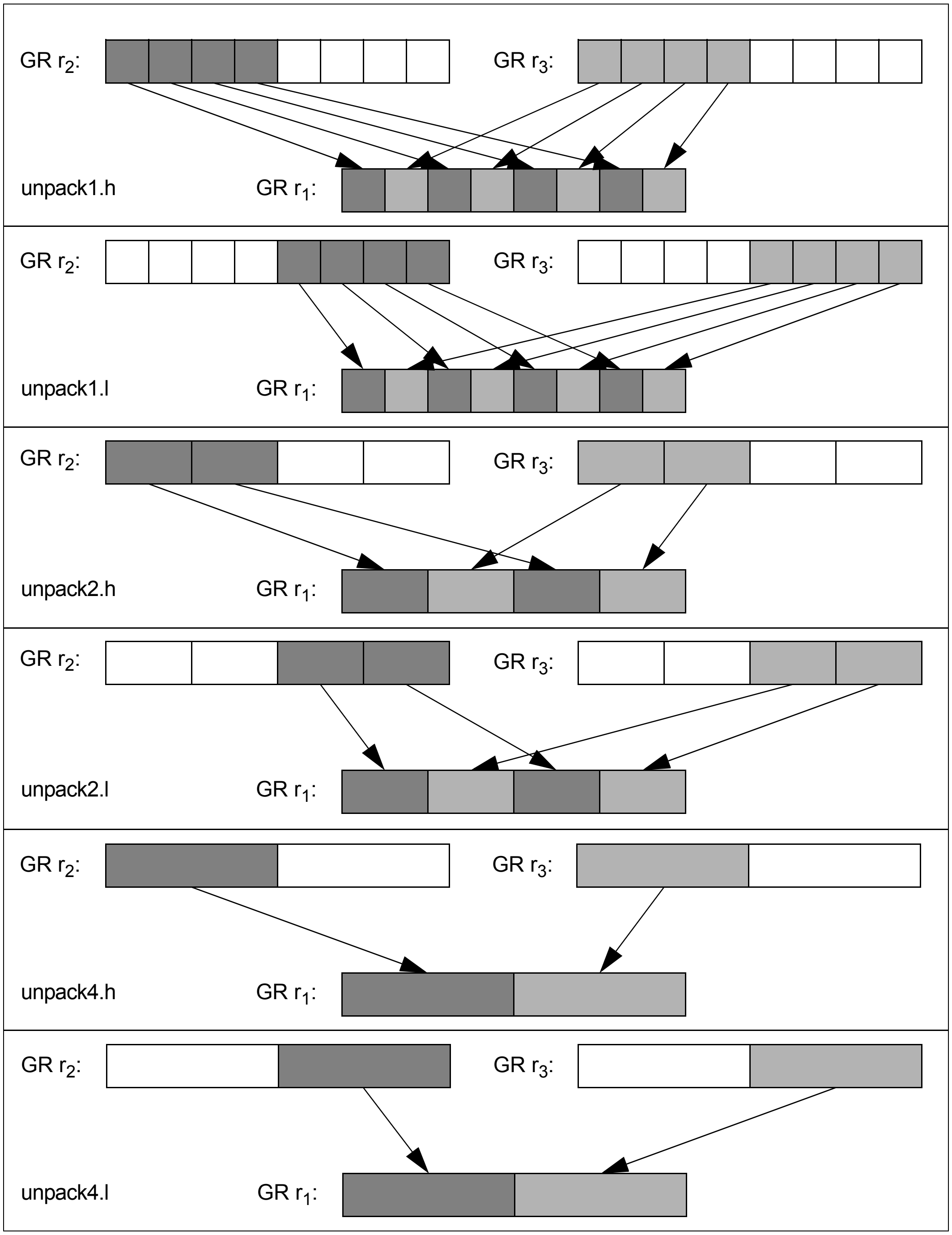
unpack — Unpack

Format:

(qp) unpack1.h $r_1 = r_2, r_3$	one_byte_form, high_form	I2
(qp) unpack2.h $r_1 = r_2, r_3$	two_byte_form, high_form	I2
(qp) unpack4.h $r_1 = r_2, r_3$	four_byte_form, high_form	I2
(qp) unpack1.l $r_1 = r_2, r_3$	one_byte_form, low_form	I2
(qp) unpack2.l $r_1 = r_2, r_3$	two_byte_form, low_form	I2
(qp) unpack4.l $r_1 = r_2, r_3$	four_byte_form, low_form	I2

Description: The data elements of GR r_2 and r_3 are unpacked, and the result placed in GR r_1. In the high_form, the most significant elements of each source register are selected, while in the low_form the least significant elements of each source register are selected. Elements are selected alternately from the source registers.

Figure 2-45. Unpack Operation



Operation:

```
if (PR[$qp$]) {
    check_target_register($r_1$);

    if (one_byte_form) {                             // one-byte elements
        x[0] = GR[$r_2$]{7:0};      y[0] = GR[$r_3$]{7:0};
        x[1] = GR[$r_2$]{15:8};     y[1] = GR[$r_3$]{15:8};
        x[2] = GR[$r_2$]{23:16};    y[2] = GR[$r_3$]{23:16};
        x[3] = GR[$r_2$]{31:24};    y[3] = GR[$r_3$]{31:24};
        x[4] = GR[$r_2$]{39:32};    y[4] = GR[$r_3$]{39:32};
        x[5] = GR[$r_2$]{47:40};    y[5] = GR[$r_3$]{47:40};
        x[6] = GR[$r_2$]{55:48};    y[6] = GR[$r_3$]{55:48};
        x[7] = GR[$r_2$]{63:56};    y[7] = GR[$r_3$]{63:56};

        if (high_form)
            GR[$r_1$] = concatenate8(   x[7], y[7], x[6], y[6],
                                       x[5], y[5], x[4], y[4]);
        else // low_form
            GR[$r_1$] = concatenate8(   x[3], y[3], x[2], y[2],
                                       x[1], y[1], x[0], y[0]);
    } else if (two_byte_form) {                      // two-byte elements
        x[0] = GR[$r_2$]{15:0};     y[0] = GR[$r_3$]{15:0};
        x[1] = GR[$r_2$]{31:16};    y[1] = GR[$r_3$]{31:16};
        x[2] = GR[$r_2$]{47:32};    y[2] = GR[$r_3$]{47:32};
        x[3] = GR[$r_2$]{63:48};    y[3] = GR[$r_3$]{63:48};

        if (high_form)
            GR[$r_1$] = concatenate4(x[3], y[3], x[2], y[2]);
        else // low_form
            GR[$r_1$] = concatenate4(x[1], y[1], x[0], y[0]);
    } else {                                         // four-byte elements
        x[0] = GR[$r_2$]{31:0};     y[0] = GR[$r_3$]{31:0};
        x[1] = GR[$r_2$]{63:32};    y[1] = GR[$r_3$]{63:32};

        if (high_form)
            GR[$r_1$] = concatenate2(x[1], y[1]);
        else // low_form
            GR[$r_1$] = concatenate2(x[0], y[0]);
    }
    GR[$r_1$].nat = GR[$r_2$].nat || GR[$r_3$].nat;
}
```

Interruptions: Illegal Operation fault

vmsw — Virtual Machine Switch

Format:

vmsw.0	zero_form	B8
vmsw.1	one_form	B8

Description: This instruction sets the PSR.vm bit to the specified value. This instruction can be used to implement transitions to/from virtual machine mode without the overhead of an interruption.

If instruction address translation is enabled and the page containing the `vmsw` instruction has access rights equal to 7, then the new value is written to the PSR.vm bit. In the zero_form, PSR.vm is set to 0, and in the one_form, PSR.vm is set to 1.

Instructions after the `vmsw` instruction in the same instruction group may be executed with the old or new value of PSR.vm. Instructions in subsequent instruction groups will be executed with PSR.vm equal to the new value.

If the above conditions are not met, this instruction takes a Virtualization fault.

This instruction can only be executed at the most privileged level. This instruction cannot be predicated.

Implementation of PSR.vm is optional. If it is not implemented, this instruction takes Illegal Operation fault. If it is implemented but either virtual machine features or the `vmsw` instruction are disabled, this instruction takes Virtualization fault when executed at the most privileged level.

Operation:

```
if (!implemented_vm())
    illegal_operation fault();

if (PSR.cpl != 0)
    privileged_operation_fault(0);

if (!(PSR.it == 1 && itlb_ar() == 7) || vm_disabled() || vmsw_disabled())
    virtualization_fault();

if (zero_form) {
    PSR.vm = 0;
}
else {
    PSR.vm = 1;
}
```

Interruptions: Illegal Operation fault
Privileged Operation fault
Virtualization fault

xchg — Exchange

Format: (qp) xchg*sz*.*ldhint* r_1 = [r_3], r_2 M16

Description: A value consisting of ***sz*** bytes is read from memory starting at the address specified by the value in GR r_3. The least significant *sz* bytes of the value in GR r_2 are written to memory starting at the address specified by the value in GR r_3. The value read from memory is then zero extended and placed in GR r_1 and the NaT bit corresponding to GR r_1 is cleared. The values of the ***sz*** completer are given in Table 2-60.

If the address specified by the value in GR r_3 is not naturally aligned to the size of the value being accessed in memory, an Unaligned Data Reference fault is taken independent of the state of the User Mask alignment checking bit, UM.ac (PSR.ac in the Processor Status Register).

Both read and write access privileges for the referenced page are required.

Table 2-60. Memory Exchange Size

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

The exchange is performed with acquire semantics, i.e., the memory read/write is made visible prior to all subsequent data memory accesses. See Section 4.4.7, “Sequentiality Attribute and Ordering” on page 2:82 for details on memory ordering.

The memory read and write are guaranteed to be atomic.

This instruction is only supported to cacheable pages with write-back write policy. Accesses to NaTPages cause a Data NaT Page Consumption fault. Accesses to pages with other memory attributes cause an Unsupported Data Reference fault.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34 on page 3:152. Locality hints do not affect program functionality and may be ignored by the implementation. See Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69 for details.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(SEMAPHORE);

    paddr = tlb_translate(GR[r3], sz, SEMAPHORE, PSR.cpl, &mattr,
                          &tmp_unused);

    if (!ma_supports_semaphores(mattr))
        unsupported_data_reference_fault(SEMAPHORE, GR[r3]);

    val = mem_xchg(GR[r2], paddr, sz, UM.be, mattr, ACQUIRE, ldhint);

    alat_inval_multiple_entries(paddr, sz);

    GR[r1] = zero_ext(val, sz * 8);
    GR[r1].nat = 0;
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
Unimplemented Data Address fault
Data Nested TLB fault
Alternate Data TLB fault
VHPT Data fault
Data TLB fault
Data Page Not Present fault
Data NaT Page Consumption fault
Data Key Miss fault
Data Key Permission fault
Data Access Rights fault
Data Dirty Bit fault
Data Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault

xma — Fixed-Point Multiply Add

Format:

(qp) xma.l $f_1 = f_3, f_4, f_2$	low_form	F2
(qp) xma.lu $f_1 = f_3, f_4, f_2$	pseudo-op of: (qp) xma.l $f_1 = f_3, f_4, f_2$	
(qp) xma.h $f_1 = f_3, f_4, f_2$	high_form	F2
(qp) xma.hu $f_1 = f_3, f_4, f_2$	high_unsigned_form	F2

Description: Two source operands (FR f_3 and FR f_4) are treated as either signed or unsigned integers and multiplied. The third source operand (FR f_2) is zero extended and added to the product. The upper or lower 64 bits of the resultant sum are selected and placed in FR f_1.

In the high_unsigned_form, the significand fields of FR f_3 and FR f_4 are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The significand field of FR f_2 is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In the high_form, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR f_2 is zero extended and added to the product. The most significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In the other forms, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The significand field of FR f_2 is zero extended and added to the product. The least significant 64-bits of the resultant sum are placed in the significand field of FR f_1.

In all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0).

Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

In all forms, if any of FR f_3 , FR f_4 , or FR f_2 is a NaTVal, FR f_1 is set to NaTVal instead of the computed result.

Operation:

```
if (PR[qp]) {
    fp_check_target_register($f_1$);
    if (tmp_isrcode = fp_reg_disabled($f_1$, $f_2$, $f_3$, $f_4$))
        disabled_fp_register_fault(tmp_isrcode, 0);

    if (fp_is_natval(FR[$f_2$]) || fp_is_natval(FR[$f_3$]) ||
        fp_is_natval(FR[$f_4$])) {
        FR[$f_1$] = NATVAL;
    } else {
        if (low_form || high_form)
            tmp_res_128 =
                fp_I64_x_I64_to_I128(FR[$f_3$].significand, FR[$f_4$].significand);
        else // high_unsigned_form
            tmp_res_128 =
                fp_U64_x_U64_to_U128(FR[$f_3$].significand, FR[$f_4$].significand);

        tmp_res_128 =
            fp_U128_add(tmp_res_128, fp_U64_to_U128(FR[$f_2$].significand));

        if (high_form || high_unsigned_form)
            FR[$f_1$].significand = tmp_res_128.hi;
        else // low_form
            FR[$f_1$].significand = tmp_res_128.lo;

        FR[$f_1$].exponent = FP_INTEGER_EXP;
        FR[$f_1$].sign = FP_SIGN_POSITIVE;
    }

    fp_update_psr($f_1$);
}
```

Interruptions: Disabled Floating-point Register fault

xmpy — Fixed-Point Multiply

Format:

(*qp*) xmpy.l $f_1 = f_3, f_4$ pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.lu $f_1 = f_3, f_4$ pseudo-op of: (*qp*) xma.l $f_1 = f_3, f_4$, f0
(*qp*) xmpy.h $f_1 = f_3, f_4$ pseudo-op of: (*qp*) xma.h $f_1 = f_3, f_4$, f0
(*qp*) xmpy.hu $f_1 = f_3, f_4$ pseudo-op of: (*qp*) xma.hu $f_1 = f_3, f_4$, f0

Description: Two source operands (FR f_3 and FR f_4) are treated as either signed or unsigned integers and multiplied. The upper or lower 64 bits of the resultant product are selected and placed in FR f_1.

In the high_unsigned_form, the significand fields of FR f_3 and FR f_4 are treated as unsigned integers and multiplied to produce a full 128-bit unsigned result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the high_form, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The most significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In the other forms, the significand fields of FR f_3 and FR f_4 are treated as signed integers and multiplied to produce a full 128-bit signed result. The least significant 64-bits of the resultant product are placed in the significand field of FR f_1.

In all forms, the exponent field of FR f_1 is set to the biased exponent for 2.0^{63} (0x1003E) and the sign field of FR f_1 is set to positive (0). Note: f1 as an operand is not an integer 1; it is just the register file format's 1.0 value.

Operation: See "xma — Fixed-Point Multiply Add" on page 3:276.

xor — Exclusive Or

Format:	(qp) xor $r_1 = r_2, r_3$	register_form	A1
	(qp) xor $r_1 = imm_8, r_3$	imm8_form	A3

Description: The two source operands are logically XORed and the result placed in GR r_1. In the register_form the first operand is GR r_2; in the imm8_form the first operand is taken from the imm_8 encoding field.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    tmp_src = (register_form ? GR[r2] : sign_ext(imm8, 8));
    tmp_nat = (register_form ? GR[r2].nat : 0);

    GR[r1] = tmp_src ^ GR[r3];
    GR[r1].nat = tmp_nat || GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

zxt — Zero Extend

Format: (*qp*) zxt*xsz* $r_1 = r_3$ I29

Description: The value in GR r_3 is zero extended above the bit position specified by *xsz* and the result is placed in GR r_1. The mnemonic values for *xsz* are given in Table 2-52 on page 3:258.

Operation:

```
if (PR[qp]) {
    check_target_register(r1);

    GR[r1] = zero_ext(GR[r3],xsz * 8);
    GR[r1].nat = GR[r3].nat;
}
```

Interruptions: Illegal Operation fault

§

Pseudo-Code Functions 3

This chapter contains a table of all pseudo-code functions used on the Itanium instruction pages.

Table 3-1. Pseudo-code Functions

Function	Operation
*xxx*_fault(parameters ...)	There are several fault functions. Each fault function accepts parameters specific to the fault, e.g., exception code values, virtual addresses, etc. If the fault is deferred for speculative load exceptions the fault function will return with a deferral indication. Otherwise, fault routines do not return and terminate the instruction sequence.
*xxx*_trap(parameters ...)	There are several trap functions. Each trap function accepts parameters specific to the trap, e.g., trap code values, virtual addresses, etc. Trap routines do not return.
acceptance_fence()	Ensures prior data memory references to uncached ordered-sequential memory pages are “accepted” before subsequent data memory references are performed by the processor.
alat_cmp(rtype, raddr)	Returns a one if the implementation finds an ALAT entry which matches the register type specified by `rtype` and the register address specified by `raddr`, else returns zero. This function is implementation specific. Note that an implementation may optionally choose to return zero (indicating no match) even if a matching entry exists in the ALAT. This provides implementation flexibility in designing fast ALAT lookup circuits.
alat_frame_update(delta_bof, delta_sof)	Notifies the ALAT of a change in the bottom of frame and/or size of frame. This allows management of the ALAT’s tag bits or other management functions it might need.
alat_inval()	Invalidate all entries in the ALAT.
alat_inval_multiple_entries(paddr, size)	The ALAT is queried using the physical memory address specified by `paddr` and the access size specified by `size`. All matching ALAT entries are invalidated. No value is returned.
alat_inval_single_entry(rtype, rega)	The ALAT is queried using the register type specified by `rtype` and the register address specified by `rega`. At most one matching ALAT entry is invalidated. No value is returned.
alat_read_memory_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should perform a memory access (requires prior address translation); returns a zero otherwise.
alat_translate_address_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should translate the source address and take associated faults; returns a zero otherwise.
alat_write(ldtype, rtype, raddr, paddr, size)	Allocates a new ALAT entry or updates an existing entry using the load type specified by `ldtype`, the register type specified by `rtype`, the register address specified by `raddr`, the physical memory address specified by `paddr`, and the access size specified by `size`. No value is returned. This function guarantees that at most only one ALAT entry exists for a given `raddr`. Based on the load type `ldtype`, if a `ld.c.nc`, `ldf.c.nc`, or `ldfp.c.nc` instruction's `raddr` matches an existing ALAT entry's register tag, but the instruction's `size` and/or `paddr` are different than that of the existing entry's, then this function may either preserve the existing entry, or invalidate it and write a new entry with the instruction's specified `size` and `paddr`.
align_to_size_boundary(vaddr, size)	Returns `vaddr` aligned to the boundary specified by `size`.
branch_predict(wh, ih, ret, target, tag)	Implementation-dependent routine which updates the processor’s branch prediction structures.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
check_branch_implemented(check_type)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether a failing check instruction causes a branch (TRUE), or a Speculative Operation fault (FALSE). The result may be different for different types of check instructions: CHKS_GENERAL, CHKS_FLOAT, CHKA_GENERAL, CHKA_FLOAT. In addition, the result may depend on other implementation-dependent parameters.
check_probe_virtualization_fault(type, cpl)	If implemented, this function may raise virtualization faults for specific probe instructions. Please refer to the instruction page for probe instruction for details.
check_target_register(r1)	If the `r1` argument specifies an out-of-frame stacked register (as defined by CFM) or `r1` specifies GR0, an Illegal Operation fault is delivered, and this function does not return.
check_target_register_sof(r1, newsof)	If the `r1` argument specifies an out-of-frame stacked register (as defined by the `newsof` argument) or `r1` specifies GR0, an Illegal Operation fault is delivered and this function does not return.
concatenate2(x1, x2)	Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.
concatenate4(x1, x2, x3, x4)	Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.
data_serialize()	Ensures all prior register updates with side-effects are observed before subsequent execution and data memory references are performed.
deliver_unmasked_pending_interrupt()	This implementation-specific function checks whether any unmasked external interrupts are pending, and if so, transfers control to the external interrupt vector.
execute_hint(hint)	Executes the hint specified by `hint`.
fadd(fp_dp, fr2)	Adds a floating-point register value to the infinitely precise product and return the infinitely precise sum, ready for rounding.
fcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcmp` instruction.
fcvt_fx_exception_fault_check(fr2, signed_form, trunc_form, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcvt.fx`, `fcvt.fxu`, `fcvt.fx.trunc` and `fcvt.fxu.trunc` instructions. It propagates NaNs.
fma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fma` instruction. It propagates NaNs and special IEEE results.
fminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `famax`, `famin`, `fmax`, and `fmin` instructions.
fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fms` and `fnma` instructions. It propagates NaNs and special IEEE results.
fmul(fr3, fr4)	Performs an infinitely precise multiply of two floating-point register values.
followed_by_stop()	Returns TRUE if the current instruction is followed by a stop; otherwise, returns FALSE.
fp_check_target_register(f1)	If the specified floating-point register identifier is 0 or 1, this function causes an illegal operation fault.
fp_decode_fault(tmp_fp_env)	Returns floating-point exception fault code values for ISR.code.
fp_decode_traps(tmp_fp_env)	Returns floating-point trap code values for ISR.code.
fp_equal(fr1, fr2)	IEEE standard equality relationship test.
fp_fr_to_mem_format(freg, size)	Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the `size` parameter.
fp_ieee_recip(num, den)	Returns the true quotient for special sets of operands, or an approximation to the reciprocal of the divisor to be used in the software divide algorithm.
fp_ieee_recip_sqrt(root)	Returns the true square root result for special operands, or an approximation to the reciprocal square root to be used in the software square root algorithm.
fp_is_nan(freg)	Returns true when floating register contains a NaN.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
fp_is_nan_or_inf(freg)	Returns true if the floating-point exception_fault_check functions returned a IEEE fault disabled default result or a propagated NaN.
fp_is_natval(freg)	Returns true when floating register contains a NaTVal
fp_is_normal(freg)	Returns true when floating register contains a normal number.
fp_is_pos_inf(freg)	Returns true when floating register contains a positive infinity.
fp_is_qnan(freg)	Returns true when floating register contains a quiet NaN.
fp_is_snan(freg)	Returns true when floating register contains a signalling NaN.
fp_is_unorm(freg)	Returns true when floating register contains an unnormalized number.
fp_is_unsupported(freg)	Returns true when floating register contains an unsupported format.
fp_less_than(fr1, fr2)	IEEE standard less-than relationship test.
fp_lesser_or_equal(fr1, fr2)	IEEE standard less-than or equal-to relationship test
fp_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
fp_normalize(fr1)	Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file
fp_raise_fault(tmp_fp_env)	Checks the local instruction state for any faulting conditions which require an interruption to be raised.
fp_raise_traps(tmp_fp_env)	Checks the local instruction state for any trapping conditions which require an interruption to be raised.
fp_reg_bank_conflict(f1, f2)	Returns true if the two specified FRs are in the same bank.
fp_reg_disabled(f1, f2, f3, f4)	Check for possible disabled floating-point register faults.
fp_reg_read(freg)	Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.
fp_unordered(fr1, fr2)	IEEE standard unordered relationship
fp_update_fpsr(sf, tmp_fp_env)	Copies a floating-point instruction's local state into the global FPSR.
fp_update_psr(dest_freg)	Conditionally sets PSR.mfl or PSR.mfh based on dest_freg.
fpcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcmp` instruction.
fpcvt_exception_fault_check(f2, signed_form, trunc_form, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcvt.fx`, `fpcvt.fxu`, `fpcvt.fx.trunc`, and `fpcvt.fxu.trunc` instructions. It propagates NaNs.
fpma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpma` instruction. It propagates NaNs and special IEEE results.
fpminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpmin`, `fpmax`, `fpamin` and `fpamax` instructions.
fpms_fpnma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpms` and `fpnma` instructions. It propagates NaNs and special IEEE results.
fprcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprcpa` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
fprsqrta_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprsqrta` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
frcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frcpa` instruction. It propagates NaNs and special IEEE results.
frsqrta_exception_fault_check(f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frsqrta` instruction. It propagates NaNs and special IEEE results
ignored_field_mask(regclass, reg, value)	Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
impl_check_mov_itir()	Implementation-specific function that returns TRUE if ITIR is checked for reserved fields and encodings on a mov to ITIR instruction.
impl_check_mov_psr_l(gr)	Implementation-specific function to check bits {63:32} of `gr` corresponding to reserved fields of the PSR for Reserved Register/Field fault.
impl_check_tlb_itir()	Implementation-specific function that returns TRUE if all fields of ITIR are checked for reserved encodings on a TLB insert instruction regardless of whether the translation is present.
impl_gitc_enable()	Implementation-specific function that indicates whether guest MOV-from-AR.ITC optimization is enabled.
impl_ia32_ar_reserved_ignored(ar3)	Implementation-specific function which indicates how the reserved and ignored fields in the specified IA-32 application register, `ar3`, behave. If it returns FALSE, the reserved and/or ignored bits in the specified application register can be written, and when read they return the value most-recently written. If it returns TRUE, attempts to write a non-zero value to a reserved field in the specified application register cause a Reserved Register/Field fault, and reads return 0; writing to an ignored field in the specified application register is ignored, and reads return the constant value defined for that field.
impl_iib()	Implementation-specific function which indicates whether Interruption Instruction Bundle registers (IIB0-1) are implemented.
impl_itir_cwi_mask()	Implementation-specific function that either returns the value passed to it or the value passed to it masked with zeros in bit positions {63:32} and/or {1:0}.
impl_ito()	Implementation-specific function which indicates whether Interval Timer Offset (ITO) register is implemented.
impl_probe_intercept()	Implementation-specific function indicates whether probe interceptions are supported.
impl_ruc()	Implementation-specific function which indicates whether Resource Utilization Counter (RUC) application register is implemented.
impl_uia_fault_supported()	Implementation-specific function that either returns TRUE if the processor reports unimplemented instruction addresses with an Unimplemented Instruction Address fault, and returns FALSE if the processor reports them with an Unimplemented Instruction Address trap.
implemented_vm()	Returns TRUE if the processor implements the PSR.vm bit (regardless of whether virtual machine features are enabled or disabled).
instruction_implemented(inst)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether `inst` is implemented.
instruction_serialize()	Ensures all prior register updates with side-effects are observed before subsequent instruction and data memory references are performed. Also ensures prior SYNC.i operations have been observed by the instruction cache.
instruction_synchronize()	Synchronizes the instruction and data stream for Flush Cache operations. This function ensures that when prior Flush Cache operations are observed by the local data cache they are observed by the local instruction cache, and when prior Flush Cache operations are observed by another processor's data cache they are observed within the same processor's instruction cache.
is_finite(freg)	Returns true when floating register contains a finite number.
is_ignored_reg(regnum)	Boolean function that returns true if `regnum` is an ignored application register, otherwise false.
is_inf(freg)	Returns true when floating register contains an infinite number.
is_interruption_cr(regnum)	Boolean function that returns true if `regnum` is one of the Interruption Control registers (see Section 3.3.5, "Interruption Control Registers" on page 2:36), otherwise false.
is_kernel_reg(ar_addr)	Returns a one if `ar_addr` is the address of a kernel register application register

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
is_read_only_reg(rtype, raddr)	Returns a one if the register addressed by `raddr` in the register bank of type `rtype` is a read only register.
is_reserved_field(regclass, arg2, arg3)	Returns true if the specified data would write a one in a reserved field.
is_reserved_reg(regclass, regnum)	Returns true if register `regnum` is reserved in the `regclass` register file.
is_supported_hint(hint)	Returns true if the implementation supports the specified `hint`. This function may depend on factors other than the `hint` value, such as which execution unit it is executed on or the slot number the instruction was encoded in.
itlb_ar()	Returns the page access rights from the ITLB for the page addressed by the current IP, or INVALID_AR if PSR.it is 0.
make_icache_coherent(paddr)	The cache line addressed by the physical address `paddr` is flushed in an implementation-specific manner that ensures that the instruction cache is coherent with the data caches.
mem_flush(paddr)	The line addressed by the physical address `paddr` is invalidated in all levels of the memory hierarchy above memory and written back to memory if it is inconsistent with memory.
mem_flush_pending_stores()	The processor is instructed to start draining pending stores in write coalescing and write buffers. This operation is a hint. There is no indication when prior stores have actually been drained.
mem_implicit_prefetch(vaddr, hint, type)	Moves the line addressed by `vaddr` to the location of the memory hierarchy specified by `hint`. This function is implementation dependent and can be ignored. The `type` allows the implementation to distinguish prefetches for different instruction types.
mem_promote(paddr, mtype, hint)	Moves the line addressed by `paddr` to the highest level of the memory hierarchy conditioned by the access hints specified by `hint`. Implementation dependent and can be ignored.
mem_read(paddr, size, border, mattr, otype, hint)	Returns the `size` bytes starting at the physical memory location specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE.
mem_read_pair(*low_value, *high_value, paddr, size, border, mattr, otype, hint)	Reads the `size` / 2 bytes of memory starting at the physical memory address specified by `paddr` into `low_value`, and the `size` / 2 bytes of memory starting at the physical memory address specified by (`paddr` + `size` / 2) into `high_value`, with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE. No value is returned.
mem_write(value, paddr, size, border, mattr, otype, hint)	Writes the least significant `size` bytes of `value` into memory starting at the physical memory address specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_write16(gr_value, ar_value, paddr, border, mattr, otype, hint)	Writes the 8 bytes of `gr_value` into memory starting at the physical memory address specified by `paddr`, and the 8 bytes of `ar_value` into memory starting at the physical memory address specified by (`paddr` + 8), with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. After the read, the least significant `size` bytes of data are written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and must be ACQUIRE.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by paddr. The read is conditioned by the locality hint specified by hint. The least significant size bytes of the sum of the value read from memory and add_val is then written to size bytes in memory starting at the physical address specified by paddr. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by byte_order. otype specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by paddr. The read is conditioned by the locality hint specified by hint. If the value read from memory is equal to cmp_val, then the least significant size bytes of data are written to size bytes in memory starting at the physical address specified by paddr. If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by byte_order. otype specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg16_cond(cmp_val, gr_data, ar_data, paddr, byte_order, mattr, otype, hint)	Returns 8 bytes from memory starting at the physical address specified by paddr. The read is conditioned by the locality hint specified by hint. If the value read from memory is equal to cmp_val, then the 8 bytes of gr_data are written to 8 bytes in memory starting at the physical address specified by (paddr & ~0x8), and the 8 bytes of ar_data are written to 8 bytes in memory starting at the physical address specified by ((paddr & ~0x8) + 8). If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by byte_order. The byte ordering only affects the ordering of bytes within each of the 8-byte values stored. otype specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
ordering_fence()	Ensures prior data memory references are made visible before future data memory references are made visible by the processor.
partially_implemented_ip()	Implementation-dependent routine which returns TRUE if the implementation, on an Unimplemented Instruction Address trap, writes IIP with the sign-extended virtual address or zero-extended physical address for what would have been the next value of IP. Returns FALSE if the implementation, on this trap, simply writes IIP with the full address which would have been the next value of IP.
pending_virtual_interrupt()	Check for unmasked pending virtual interrupt.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, phys_id of the predicate.
rotate_regs()	Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.
rse_enable_current_frame_load()	If the RSE load pointer (RSE.BSPLoad) is greater than AR[BSP], the RSE.CFLE bit is set to indicate that mandatory RSE loads are allowed to restore registers in the current frame (in no other case does the RSE spill or fill registers in the current frame). This function does not perform mandatory RSE loads. This procedure does not cause any interruptions.
rse_ensure_regs_loaded(number_of_bytes)	All registers and NaT collections between AR[BSP] and (AR[BSP]-number_of_bytes) which are not already in stacked registers are loaded into the register stack with mandatory RSE loads. If the number of registers to be loaded is greater than RSE.N_STACK_PHYS an Illegal Operation fault is raised. All registers starting with backing store address (AR[BSP] - 8) and decrementing down to and including backing store address (AR[BSP] - number_of_bytes) are made part of the dirty partition. With exception of the current frame, all other stacked registers are made part of the invalid partition. Note that number_of_bytes may be zero. The resulting sequence of RSE loads may be interrupted. Mandatory RSE loads may cause an interruption; see Table 6-6, "RSE Interruption Summary" on page 6-145.
rse_invalidate_non_current_regs()	All registers outside the current frame are invalidated.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
rse_load(type)	Restores a register or NaT collection from the backing store (load_address = RSE.BspLoad - 8). If load_address{8:3} is equal to 0x3f then a NaT collection is loaded into a NaT dispersal register. (dispersal register may not be the same as AR[RNAT].) If load_address{8:3} is not equal to 0x3f then the register RSE.LoadReg - 1 is loaded and the NaT bit for that register is set to dispersal_register{load_address{8:3}}. If the load is successful RSE.BspLoad is decremented by 8. If the load is successful and a register was loaded RSE.LoadReg is decremented by 1 (possibly wrapping in the stacked registers). The load moves a register from the invalid partition to the current frame if RSE.CFLE is 1, or to the clean partition if RSE.CFLE is 0. For mandatory RSE loads, type is MANDATORY. Mandatory RSE loads may cause interruptions. See Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_new_frame(current_frame_size, new_frame_size)	A new frame is defined without changing any register renaming. The new frame size is completely defined by the new_frame_size parameter (successive calls are not cumulative). If new_frame_size is larger than current_frame_size and the number of registers in the invalid and clean partitions is less than the size of frame growth then mandatory RSE stores are issued until enough registers are available. The resulting sequence of RSE stores may be interrupted. Mandatory RSE stores may cause interruptions; see Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_preserve_frame(preserved_frame_size)	The number of registers specified by preserved_frame_size are marked to be preserved by the RSE. Register renaming causes the preserved_frame_size registers after GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored.
rse_restore_frame(preserved_sol, growth, current_frame_size)	The first two parameters define how the current frame is about to be updated by a branch return or rfi: preserved_sol defines how many registers need to be restored below RSE.BOF; growth defines by how many registers the top of the current frame will grow (growth will generally be negative). The number of registers specified by preserved_sol are marked to be restored. Register renaming causes the preserved_sol registers before GR[32] to be renamed to GR[32]. AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored. If the number of dirty and clean registers is less than preserved_sol then mandatory RSE loads must be issued before the new current frame is considered valid. This function does not perform mandatory RSE loads. This function returns TRUE if the preserved frame grows beyond the invalid and clean regions into the dirty region. In this case the third argument, current_frame_size, is used to force the returned to frame to zero (see Section 6.5.5, “Bad PFS used by Branch Return” on page 2:143).
rse_store(type)	Saves a register or NaT collection to the backing store (store_address = AR[BSPSTORE]). If store_address{8:3} is equal to 0x3f then the NaT collection AR[RNAT] is stored. If store_address{8:3} is not equal to 0x3f then the register RSE.StoreReg is stored and the NaT bit from that register is deposited in AR[RNAT]{store_address{8:3}}. If the store is successful AR[BSPSTORE] is incremented by 8. If the store is successful and a register was stored RSE.StoreReg is incremented by 1 (possibly wrapping in the stacked registers). This store moves a register from the dirty partition to the clean partition. For mandatory RSE stores, type is MANDATORY. Mandatory RSE stores may cause interruptions. See Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_update_internal_stack_pointers(new_store_pointer)	Given a new value for AR[BSPSTORE] (new_store_pointer) this function computes the new value for AR[BSP]. This value is equal to new_store_pointer plus the number of dirty registers plus the number of intervening NaT collections. This means that the size of the dirty partition is the same before and after a write to AR[BSPSTORE]. All clean registers are moved to the invalid partition.
sign_ext(value, pos)	Returns a 64 bit number with bits pos-1 through 0 taken from value and bit pos-1 of value replicated in bit positions pos through 63. If pos is greater than or equal to 64, value is returned.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
spontaneous_deferral(paddr, size, border, mattr, otype, hint, *defer)	Implementation-dependent routine which optionally forces `*defer` to TRUE if all of the following are true: spontaneous deferral is enabled, spontaneous deferral is permitted by the programming model, and the processor determines it would be advantageous to defer the speculative load (e.g., based on a miss in some particular level of cache).
spontaneous_deferral_enabled()	Implementation-dependent routine which returns TRUE or FALSE, depending on whether spontaneous deferral of speculative loads is enabled or disabled in the processor.
tlb_access_key(vaddr, itype)	This function returns, in bits 31:8, the access key from the TLB for the entry corresponding to `vaddr` and `itype`; bits 63:32 and 7:0 return 0. If `vaddr` is an unimplemented virtual address, or a matching present translation is not found, the value 1 is returned.
tlb_broadcast_purge(rid, vaddr, size, type)	Sends a broadcast purge DTC and ITC transaction to other processors in the multiprocessor coherency domain, where the region identifier (`rid`), virtual address (`vaddr`) and page size (`size`) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type (`type`) specifies whether the ALAT on other processors should also be purged in conjunction with the TC.
tlb_enter_privileged_code()	This function determines the new privilege level for epc from the TLB entry for the page containing this instruction. If the page containing the epc instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the epc instruction.
tlb_grant_permission(vaddr, type, pl)	Returns a boolean indicating if read, write access is granted for the specified virtual memory address (`vaddr`) and privilege level (`pl`). The access type (`type`) specifies either read or write. The following faults are checked:: • Data Nested TLB fault • Alternate Data TLB fault • VHPT Data fault • Data TLB fault • Data Page Not Present fault • Data NaT Page Consumption fault • Data Key Miss fault If a fault is generated, this function does not return.
tlb_insert_data(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the DTLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true the entry is placed in the TR section, otherwise the TC section.
tlb_insert_inst(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the ITLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true, the entry is placed in the TR section, otherwise the TC section.
tlb_may_purge_dtc_entries(rid, vaddr, size)	May locally purge DTC entries that match the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
tlb_may_purge_itc_entries(rid, vaddr, size)	May locally purge ITC entries that match the specified virtual address (vaddr), region identifier (rid) and page size (size). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_dtc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTC entries matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing DTR translation, an implementation may generate a machine check abort.
tlb_must_purge_dtr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTR entries matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_itc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITC entry matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing ITR translation, an implementation may generate a machine check abort.
tlb_must_purge_itr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITR entry matching the specified region identifier (rid), virtual address (vaddr) and page size (size). vaddr{63:61} (VRN) is ignored in the purge, i.e all entries that match vaddr{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_purge_translation_cache(loop)	Removes 1 to N translations from the local processor's ITC and DTC. The number of entries removed is implementation specific. The parameter loop is used to generate an implementation-specific purge parameter.
tlb_replacement_algorithm(tlb)	Returns the next ITC or DTC slot number to replace. Replacement algorithms are implementation specific. tlb specifies to perform the algorithm on the ITC or DTC.
tlb_search_pkr(key)	Searches for a valid protection key register with a matching protection key. The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
tlb_translate(vaddr, size, type, cpl, *attr, *defer)	Returns the translated data physical address for the specified virtual memory address (vaddr) when translation enabled; otherwise, returns vaddr. size specifies the size of the access, type specifies the type of access (e.g., read, write, advance, spec). cpl specifies the privilege level for access checking purposes. *attr returns the mapped physical memory attribute. If any fault conditions are detected and deferred, tlb_translate returns with *defer set. If a fault is generated but the fault is not deferred, tlb_translate does not return. tlb_translate checks the following faults: • Unimplemented Data Address fault • Data Nested TLB fault • Alternate Data TLB fault • VHPT Data fault • Data TLB fault • Data Page Not Present fault • Data NaT Page Consumption fault • Data Key Miss fault • Data Key Permission fault • Data Access Rights fault • Data Dirty Bit fault • Data Access Bit fault • Data Debug fault • Unaligned Data Reference fault • Unsupported Data Reference fault
tlb_translate_nonaccess(vaddr, type)	Returns the translated data physical address for the specified virtual memory address (vaddr). type specifies the type of access (e.g., FC, TPA). If a fault is generated, tlb_translate_nonaccess does not return. The following faults are checked: • Unimplemented Data Address fault • Virtualization fault (tpa only) • Data Nested TLB fault • Alternate Data TLB fault • VHPT Data fault • Data TLB fault • Data Page Not Present fault • Data NaT Page Consumption fault • Data Access Rights fault (fc only)
tlb_vhpt_hash(vrn, vaddr61, rid, size)	Generates a VHPT entry address for the specified virtual region number (vrn) and 61-bit virtual offset (vaddr61), region identifier (rid) and page size (size). Tlb_vhpt_hash hashes vaddr, rid and size parameters to produce a hash index. The hash index is then masked based on PTA.size and concatenated with PTA.base to generate the VHPT entry address. The long format hash is implementation specific.
tlb_vhpt_tag(vaddr, rid, size)	Generates a VHPT tag identifier for the specified virtual address (vaddr), region identifier (rid) and page size (size). Tlb_vhpt_tag hashes the vaddr, rid and size parameters to produce translation identifier. The tag in conjunction with the hash index is used to uniquely identify translations in the VHPT. Tag generation is implementation specific. All processor models tag function must guarantee that bit 63 of the generated tag is zero (ti bit).
undefined()	Returns an undefined 64-bit value.
undefined_behavior()	Causes undefined processor behavior. Extent of undefined behavior is described in Section 3.5, "Undefined Behavior" on page 1:44.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model specific.
unimplemented_virtual_address(vaddr, vm)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. If vm is 1, one additional bit of virtual address is treated as unimplemented. This function is model specific.
vm_all_probes()	Returns TRUE if the processor is configured to virtualize all probe instructions when PSR.vm is 1. See Section 11.7.4.2.8, "Probe Instruction Virtualization" on page 2:344 for details.
vm_disabled()	Returns TRUE if the processor implements the PSR.vm bit and virtual machine features are disabled. See Section 3.4, "Processor Virtualization" on page 2:44 in SDM and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 in SDM for details.
vm_select_probes()	Returns TRUE if the processor is configured to virtualize selected probe instructions when PSR.vm is 1. See Section 11.7.4.2.8, "Probe Instruction Virtualization" on page 2:344 for details.
vmsw_disabled()	Returns TRUE if the processor implements the PSR.vm bit and the `vmsw` instruction is disabled. See Section 3.4, "Processor Virtualization" on page 2:44 in SDM and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:446 in SDM for details.
zero_ext(value, pos)	Returns a 64 bit unsigned number with bits `pos`-1 through 0 taken from `value` and zeroes in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.

§

Instruction Formats 4

Each Itanium instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 4-1 lists the instruction types and the execution unit type on which they are executed:

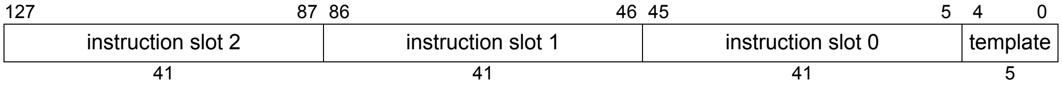
Table 4-1. Relationship between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit[a]

[a] L+X Major Opcodes 0 - 7 execute on an I-unit. L+X Major Opcodes 8 - F execute on a B-unit.

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 4-1.

Figure 4-1. Bundle Format



The template field specifies two properties: stops within the current bundle, and the mapping of instruction slots to execution unit types. Not all combinations of these two properties are allowed - Table 4-2 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle; listed within each column is the execution unit type controlled by that instruction slot for each encoding of the template field. A double line to the right of an instruction slot indicates that a stop occurs at that point within the current bundle. See "Instruction Encoding Overview" on page 1:38 for the definition of a stop. Within a bundle, execution order proceeds from slot 0 to slot 2. Unused template values (appearing as empty rows in Table 4-2) are reserved and cause an Illegal Operation fault.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

Table 4-2. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit ‖
02	M-unit	I-unit ‖	I-unit
03	M-unit	I-unit ‖	I-unit ‖
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a] ‖
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit ‖
0A	M-unit ‖	M-unit	I-unit
0B	M-unit ‖	M-unit	I-unit ‖
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit ‖
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit ‖
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit ‖
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit ‖
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit ‖
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit ‖
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit ‖
1E			
1F			

a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode break.i and nop.i in addition to any X-unit instruction.

4.1 Format Summary

All instructions in the instruction set are 41 bits in length. The leftmost 4 bits (40:37) of each instruction are the major opcode. Table 4-3 shows the major opcode assignments for each of the 5 instruction types — ALU (A), Integer (I), Memory (M), Floating-point (F), and Branch (B). Bundle template bits are used to distinguish among the 4 columns, so the same major op values can be reused in each column.

Unused major ops (appearing as blank entries in Table 4-3) behave in one of four ways:

- Ignored major ops (white entries in Table 4-3) execute as `nop` instructions.

- Reserved major ops (light gray in the gray scale version of Table 4-3, brown in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 major ops (dark gray in the gray scale version of Table 4-3, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit major ops (medium gray in the gray scale version of Table 4-3, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 major ops (purple) only in their RAW dependency behavior (see "RAW Dependency Table" on page 3:374).

Table 4-3. Major Opcode Assignments

Major Op (Bits 40:37)	Instruction Type				
	I/A	M/A	F	B	L+X
0	Misc 0	Sys/Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Sys/Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	Indirect Predict/Nop 2	2
3	3	3	3	3	3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	4
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	5
6	6	FP Ld/St +Reg/setf 6	6	6	movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	IP-relative Predict 7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	e 8	8
9	Add Imm$_{22}$ 9	Add Imm$_{22}$ 9	fma 9	e 9	9
A	A	A	fms A	e A	A
B	B	B	fms B	e B	B
C	Compare C	Compare C	fnma C	e C	Long Branch C
D	Compare D	Compare D	fnma D	e D	Long Call D
E	Compare E	Compare E	fselect/xma E	e E	E
F	F	F	F	e F	F

Table 4-4 on page 3:296 summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in Table 4-5 on page 3:298. A color version of this chapter is available for those heavily involved in working with the instruction encodings.

The instruction field names, used throughout this chapter, are described in Table 4-6 on page 3:298. The set of special notations (such as whether an instruction is privileged) are listed in Table 4-7 on page 3:299. These notations appear in the "Instruction" column of the opcode tables.

Most instruction containing immediates encode those immediates in more than one instruction field. For example, the 14-bit immediate in the Add Imm$_{14}$ instruction (format A4) is formed from the imm$_{7b}$, imm$_{6d}$, and s fields. Table 4-74 on page 3:368 shows how the immediates are formed from the instruction fields for each instruction which has an immediate.

Table 4-4. Instruction Format Summary

<table>
<tr><td></td><td></td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>ALU</td><td>A1</td><td colspan="4">8</td><td></td><td colspan="2">x_{2a}</td><td>v_e</td><td colspan="4">x_4</td><td colspan="2">x_{2b}</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Shift L and Add</td><td>A2</td><td colspan="4">8</td><td></td><td colspan="2">x_{2a}</td><td>v_e</td><td colspan="4">x_4</td><td colspan="2">ct_{2d}</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>ALU Imm_8</td><td>A3</td><td colspan="4">8</td><td>s</td><td colspan="2">x_{2a}</td><td>v_e</td><td colspan="4">x_4</td><td colspan="2">x_{2b}</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Add Imm_{14}</td><td>A4</td><td colspan="4">8</td><td>s</td><td colspan="2">x_{2a}</td><td>v_e</td><td colspan="6">imm_{6d}</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Add Imm_{22}</td><td>A5</td><td colspan="4">9</td><td>s</td><td colspan="9">imm_{9d}</td><td colspan="5">imm_{5c}</td><td colspan="2">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Compare</td><td>A6</td><td colspan="4">C - E</td><td>t_b</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>Compare to Zero</td><td>A7</td><td colspan="4">C - E</td><td>t_b</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">r_3</td><td colspan="7">0</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>Compare Imm_8</td><td>A8</td><td colspan="4">C - E</td><td>s</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>MM ALU</td><td>A9</td><td colspan="4">8</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td colspan="4">x_4</td><td colspan="2">x_{2b}</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Shift and Add</td><td>A10</td><td colspan="4">8</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td colspan="4">x_4</td><td colspan="2">ct_{2d}</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Multiply Shift</td><td>I1</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">ct_{2d}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Mpy/Mix/Pack</td><td>I2</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Mux1</td><td>I3</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td colspan="4"></td><td colspan="4">mbt_{4c}</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Mux2</td><td>I4</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td colspan="8">mht_{8c}</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Shift R Variable</td><td>I5</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Shift R Fixed</td><td>I6</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td></td><td colspan="5">$count_{5b}$</td><td></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Shift L Variable</td><td>I7</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>MM Shift L Fixed</td><td>I8</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td colspan="3"></td><td colspan="5">$ccount_{5c}$</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Bit Strings</td><td>I9</td><td colspan="4">7</td><td>z_a</td><td colspan="2">x_{2a}</td><td>z_b</td><td>v_e</td><td colspan="2">x_{2c}</td><td colspan="2">x_{2b}</td><td></td><td colspan="7">r_3</td><td colspan="7">0</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Shift Right Pair</td><td>I10</td><td colspan="4">5</td><td></td><td colspan="2">x_2</td><td>x</td><td colspan="6">$count_{6d}$</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Extract</td><td>I11</td><td colspan="4">5</td><td></td><td colspan="2">x_2</td><td>x</td><td colspan="6">len_{6d}</td><td colspan="7">r_3</td><td colspan="6">pos_{6b}</td><td>y</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Dep.Z</td><td>I12</td><td colspan="4">5</td><td></td><td colspan="2">x_2</td><td>x</td><td colspan="6">len_{6d}</td><td>y</td><td colspan="6">$cpos_{6c}$</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Dep.Z Imm_8</td><td>I13</td><td colspan="4">5</td><td>s</td><td colspan="2">x_2</td><td>x</td><td colspan="6">len_{6d}</td><td>y</td><td colspan="6">$cpos_{6c}$</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Deposit Imm_1</td><td>I14</td><td colspan="4">5</td><td>s</td><td colspan="2">x_2</td><td>x</td><td colspan="6">len_{6d}</td><td colspan="7">r_3</td><td colspan="6">$cpos_{6b}$</td><td></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Deposit</td><td>I15</td><td colspan="4">4</td><td colspan="6">$cpos_{6d}$</td><td colspan="4">len_{4d}</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Test Bit</td><td>I16</td><td colspan="4">5</td><td>t_b</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">r_3</td><td colspan="6">pos_{6b}</td><td>y</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>Test NaT</td><td>I17</td><td colspan="4">5</td><td>t_b</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">r_3</td><td>x</td><td colspan="5"></td><td>y</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>Nop/Hint</td><td>I18</td><td colspan="4">0</td><td>i</td><td colspan="3">x_3</td><td colspan="6">x_6</td><td>y</td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td></tr>
<tr><td>Break</td><td>I19</td><td colspan="4">0</td><td>i</td><td colspan="3">x_3</td><td colspan="6">x_6</td><td></td><td colspan="20">imm_{20a}</td><td colspan="6">qp</td></tr>
<tr><td>Int Spec Check</td><td>I20</td><td colspan="4">0</td><td>s</td><td colspan="3">x_3</td><td colspan="13">imm_{13c}</td><td colspan="7">r_2</td><td colspan="7">imm_{7a}</td><td colspan="6">qp</td></tr>
<tr><td>Move to BR</td><td>I21</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="9">$timm_{9c}$</td><td>ih</td><td>x</td><td colspan="2">wh</td><td colspan="7">r_2</td><td colspan="4"></td><td colspan="3">b_1</td><td colspan="6">qp</td></tr>
<tr><td>Move from BR</td><td>I22</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="11"></td><td colspan="3">b_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Move to Pred</td><td>I23</td><td colspan="4">0</td><td>s</td><td colspan="3">x_3</td><td></td><td colspan="8">$mask_{8c}$</td><td colspan="4"></td><td colspan="7">r_2</td><td colspan="7">$mask_{7a}$</td><td colspan="6">qp</td></tr>
<tr><td>Move to Pred Imm_{44}</td><td>I24</td><td colspan="4">0</td><td>s</td><td colspan="3">x_3</td><td colspan="27">imm_{27a}</td><td colspan="6">qp</td></tr>
<tr><td>Move from Pred/IP</td><td>I25</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="14"></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Move to AR</td><td>I26</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="7">ar_3</td><td colspan="7">r_2</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>Move to AR Imm_8</td><td>I27</td><td colspan="4">0</td><td>s</td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="7">ar_3</td><td colspan="7">imm_{7b}</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>Move from AR</td><td>I28</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="7">ar_3</td><td colspan="7"></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Sxt/Zxt/Czx</td><td>I29</td><td colspan="4">0</td><td></td><td colspan="3">x_3</td><td colspan="6">x_6</td><td colspan="7">r_3</td><td colspan="7"></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Test Feature</td><td>I30</td><td colspan="4">5</td><td>t_b</td><td colspan="2">x_2</td><td>t_a</td><td colspan="6">p_2</td><td colspan="7">0</td><td>x</td><td colspan="5">imm_{5b}</td><td>y</td><td>c</td><td colspan="6">p_1</td><td colspan="6">qp</td></tr>
<tr><td>Int Load</td><td>M1</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7"></td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Int Load +Reg</td><td>M2</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Int Load +Imm</td><td>M3</td><td colspan="4">5</td><td>s</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>i</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Int Store</td><td>M4</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>Int Store +Imm</td><td>M5</td><td colspan="4">5</td><td>s</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>i</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">imm_{7a}</td><td colspan="6">qp</td></tr>
<tr><td>FP Load</td><td>M6</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7"></td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>FP Load +Reg</td><td>M7</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>FP Load +Imm</td><td>M8</td><td colspan="4">7</td><td>s</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>i</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>FP Store</td><td>M9</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">f_2</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>FP Store +Imm</td><td>M10</td><td colspan="4">7</td><td>s</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>i</td><td colspan="7">r_3</td><td colspan="7">f_2</td><td colspan="7">imm_{7a}</td><td colspan="6">qp</td></tr>
<tr><td>FP Load Pair</td><td>M11</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>FP Load Pair +Imm</td><td>M12</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">f_2</td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>Line Prefetch</td><td>M13</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="14"></td><td colspan="6">qp</td></tr>
<tr><td>Line Prefetch +Reg</td><td>M14</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>Line Prefetch +Imm</td><td>M15</td><td colspan="4">7</td><td>s</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>i</td><td colspan="7">r_3</td><td colspan="7">imm_{7b}</td><td colspan="7"></td><td colspan="6">qp</td></tr>
<tr><td>(Cmp &) Exchg</td><td>M16</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="7">r_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Fetch & Add</td><td>M17</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2">hint</td><td>x</td><td colspan="7">r_3</td><td colspan="4"></td><td>s</td><td colspan="2">i_{2b}</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td>Set FR</td><td>M18</td><td colspan="4">6</td><td>m</td><td colspan="6">x_6</td><td colspan="2"></td><td>x</td><td colspan="7"></td><td colspan="7">r_2</td><td colspan="7">f_1</td><td colspan="6">qp</td></tr>
<tr><td>Get FR</td><td>M19</td><td colspan="4">4</td><td>m</td><td colspan="6">x_6</td><td colspan="2"></td><td>x</td><td colspan="7"></td><td colspan="7">f_2</td><td colspan="7">r_1</td><td colspan="6">qp</td></tr>
<tr><td></td><td></td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Table 4-4. Instruction Format Summary (Continued)

		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Int Spec Check	M20	1				s	x_3			imm$_{13c}$													r_2							imm$_{7a}$							qp						
FP Spec Check	M21	1				s	x_3			imm$_{13c}$													f_2							imm$_{7a}$							qp						
Int ALAT Check	M22	0				s	x_3			imm$_{20b}$																				r_1							qp						
FP ALAT Check	M23	0				s	x_3			imm$_{20b}$																				f_1							qp						
Sync/Srlz/ALAT	M24	0					x_3			x_2		x_4																									qp						
RSE Control	M25	0					x_3			x_2		x_4																									0						
Int ALAT Inval	M26	0					x_3			x_2		x_4																		r_1							qp						
FP ALAT Inval	M27	0					x_3			x_2		x_4																		f_1							qp						
Flush Cache	M28	1				x	x_3			x_6						r_3																					qp						
Move to AR	M29	1					x_3			x_6						ar_3							r_2														qp						
Move to AR Imm$_8$	M30	0				s	x_3			x_2		x_4				ar_3							imm$_{7b}$														qp						
Move from AR	M31	1					x_3			x_6						ar_3														r_1							qp						
Move to CR	M32	1					x_3			x_6						cr_3							r_2														qp						
Move from CR	M33	1					x_3			x_6						cr_3														r_1							qp						
Alloc	M34	1					x_3					sor				sol							sof							r_1							qp						
Move to PSR	M35	1					x_3			x_6													r_2														qp						
Move from PSR	M36	1					x_3			x_6																				r_1							qp						
Break	M37	0				i	x_3			x_2		x_4					imm$_{20a}$																				qp						
Probe	M38	1					x_3			x_6						r_3							r_2							r_1							qp						
Probe Imm$_2$	M39	1					x_3			x_6						r_3												i_{2b}		r_1							qp						
Probe Fault Imm$_2$	M40	1					x_3			x_6						r_3												i_{2b}									qp						
TC Insert	M41	1					x_3			x_6													r_2														qp						
Mv to Ind/TR Ins	M42	1					x_3			x_6						r_3							r_2														qp						
Mv from Ind	M43	1					x_3			x_6						r_3														r_1							qp						
Set/Reset Mask	M44	0				i	x_3			i_{2d}		x_4				imm$_{21a}$																					qp						
Translation Purge	M45	1					x_3			x_6						r_3							r_2														qp						
Translation Access	M46	1					x_3			x_6						r_3														r_1							qp						
TC Entry Purge	M47	1					x_3			x_6						r_3																					qp						
Nop/Hint	M48	0				i	x_3			x_2		x_4				y	imm$_{20a}$																				qp						
IP-Relative Branch	B1	4				s	d	wh		imm$_{20b}$																				p				btype			qp						
Counted Branch	B2	4				s	d	wh		imm$_{20b}$																				p				btype			0						
IP-Relative Call	B3	5				s	d	wh		imm$_{20b}$																				p				b_1			qp						
Indirect Branch	B4	0					d	wh		x_6																	b_2			p				btype			qp						
Indirect Call	B5	1					d	wh																			b_2			p				b_1			qp						
IP-Relative Predict	B6	7				s	ih	t_{2e}		imm$_{20b}$																				timm$_{7a}$								wh					
Indirect Predict	B7	2					ih	t_{2e}		x_6																	b_2			timm$_{7a}$								wh					
Misc	B8	0								x_6																											0						
Break/Nop/Hint	B9	0/2				i				x_6							imm$_{20a}$																				qp						
FP Arithmetic	F1	8 - D				x	sf		f_4							f_3							f_2							f_1							qp						
Fixed Multiply Add	F2	E				x	x_2		f_4							f_3							f_2							f_1							qp						
FP Select	F3	E				x			f_4							f_3							f_2							f_1							qp						
FP Compare	F4	4				r_b	sf		r_a	p_2						f_3							f_2							t_a	p_1						qp						
FP Class	F5	5						fc_2		p_2						fclass$_{7c}$							f_2							t_a	p_1						qp						
FP Recip Approx	F6	0 - 1				q	sf		x	p_2						f_3							f_2							f_1							qp						
FP Recip Sqrt App	F7	0 - 1				q	sf		x	p_2						f_3														f_1							qp						
FP Min/Max/Pcmp	F8	0 - 1					sf		x	x_6						f_3							f_2							f_1							qp						
FP Merge/Logical	F9	0 - 1							x	x_6						f_3							f_2							f_1							qp						
Convert FP to Fixed	F10	0 - 1					sf		x	x_6													f_2							f_1							qp						
Convert Fixed to FP	F11	0							x	x_6													f_2							f_1							qp						
FP Set Controls	F12	0					sf		x	x_6						omask$_{7c}$							amask$_{7b}$														qp						
FP Clear Flags	F13	0					sf		x	x_6																											qp						
FP Check Flags	F14	0				s	sf		x	x_6							imm$_{20a}$																				qp						
Break	F15	0				i			x	x_6							imm$_{20a}$																				qp						
Nop/Hint	F16	0				i			x	x_6						y	imm$_{20a}$																				qp						
Break	X1	0				i	x_3			x_6							imm$_{20a}$																				qp						imm$_{41}$
Move Imm$_{64}$	X2	6				i	imm$_{9d}$									imm$_{5c}$					i_c	v_c	imm$_{7b}$							r_1							qp						imm$_{41}$
Long Branch	X3	C				i	d	wh		imm$_{20b}$																				p				btype			qp						imm$_{39}$
Long Call	X4	D				i	d	wh		imm$_{20b}$																				p				b_1			qp						imm$_{39}$
Nop/Hint	X5	0				i	x_3			x_6						y	imm$_{20a}$																				qp						imm$_{41}$
		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Table 4-5. Instruction Field Color Key

Field & Color	
ALU Instruction	Opcode Extension
Integer Instruction	Opcode Hint Extension
Memory Instruction	Immediate
Branch Instruction	Indirect Source
Floating-point Instruction	Predicate Destination
Integer Source	Integer Destination
Memory Source	Memory Source & Destination
Shift Source	Shift Immediate
Special Register Source	Special Register Destination
Floating-point Source	Floating-point Destination
Branch Source	Branch Destination
Address Source	Branch Tag Immediate
Qualifying Predicate	Reserved Instruction
Ignored Field/Instruction	Reserved Inst if PR[qp] is 1
	Reserved B-type Inst if PR[qp] is 1

Table 4-6. Instruction Field Names

Field Name	Description
ar_3	application register source/target
b_1, b_2	branch register source/target
btype	branch type opcode extension
c	complement compare relation opcode extension
$ccount_{5c}$	multimedia shift left complemented shift count immediate
$count_{5b}$, $count_{6d}$	multimedia shift right/shift right pair shift count immediate
$cpos_x$	deposit complemented bit position immediate
cr_3	control register source/target
ct_{2d}	multimedia multiply shift/shift and add shift count immediate
d	branch cache deallocation hint opcode extension
f_n	floating-point register source/target
fc_2, $fclass_{7c}$	floating-point class immediate
hint	memory reference hint opcode extension
i, i_{2b}, $i_{2d,}$ imm_x	immediate of length 1, 2, or *x*
ih	branch importance hint opcode extension
len_{4d}, len_{6d}	extract/deposit length immediate
m	memory reference post-modify opcode extension
$mask_x$	predicate immediate mask
mbt_{4c}, mht_{8c}	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p_1, p_2	predicate register target
pos_{6b}	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r_n	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension

Table 4-6. Instruction Field Names (Continued)

Field Name	Description
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t_a, t_b	compare type opcode extension
t_{2e}, $timm_x$	branch predict tag immediate
v_x	reserved opcode extension field
wh	branch whether hint opcode extension
x, x_n	opcode extension of length 1 or *n*
y	extract/deposit/test bit/test NaT/hint opcode extension
z_a, z_b	multimedia operand size opcode extension

Table 4-7. Special Instruction Notations

Notation	Description
e	instruction ends an instruction group when taken, or for Reserved if PR[qp] is 1 (cyan) encodings and non-branch instructions with a qualifying predicate, when its PR[qp] is 1, or for Reserved (brown) encodings, unconditionally
f	instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0
i	instruction is allowed in the I slot of an MLI template
l	instruction must be the last in an instruction group
p	privileged instruction
t	instruction is only allowed in instruction slot 2

The remaining sections of this chapter present the detailed encodings of all instructions. The "A-Unit Instruction encodings" are presented first, followed by the "I-Unit Instruction Encodings" on page 3:310, "M-Unit Instruction Encodings" on page 3:323, "B-Unit Instruction Encodings" on page 3:349, "F-Unit Instruction Encodings" on page 3:356, and "X-Unit Instruction Encodings" on page 3:365.

Within each section, the instructions are grouped by function, and appear with their instruction format in the same order as in Table 4-4, "Instruction Format Summary" on page 3:296. The opcode extension fields are briefly described and tables present the opcode extension assignments. Unused instruction encodings (appearing as blank entries in the opcode extensions tables) behave in one of four ways:

- Ignored instructions (white color entries in the tables) execute as `nop` instructions.
- Reserved instructions (light gray color in the gray scale version of the tables, brown color in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 instructions (dark gray in the gray scale version of the tables, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit instructions (medium gray in the gray scale version of the tables, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 instructions (purple) only in their RAW dependency behavior (see "RAW Dependency Table" on page 3:374).

Some processors may implement the Reserved if PR[qp] is 1 (purple) and Reserved if PR[qp] is 1 B-unit (cyan) encodings in the L+X opcode space as Reserved (brown). These encodings appear in the L+X column of Table 4-3 on page 3:295, and in Table 4-69 on page 3:366, Table 4-70 on page 3:366, Table 4-71 on page 3:367, and Table 4-72 on page 3:367. On processors which implement these encodings as Reserved (brown), the operating system is required to provide an Illegal Operation fault handler which emulates them as Reserved if PR[qp] is 1 (cyan/purple) by decoding the reserved opcodes, checking the qualifying predicate, and returning to the next instruction if PR[qp] is 0.

Constant 0 fields in instructions must be 0 or undefined operation results. The undefined operation may include checking that the constant field is 0 and causing an Illegal Operation fault if it is not. If an instruction having a constant 0 field also has a qualifying predicate (qp field), the fault or other undefined operation must not occur if PR[qp] is 0. For constant 0 fields in instruction bits 5:0 (normally used for qp), the fault or other undefined operation may or may not depend on the PR addressed by those bits.

Ignored (white space) fields in instructions should be coded as 0. Although ignored in this revision of the architecture, future architecture revisions may define these fields as hint extensions. These hint extensions will be defined such that the 0 value in each field corresponds to the default hint. It is expected that assemblers will automatically set these fields to zero by default.

Unused opcode hint extension values (white color entries in Hint Completer tables) should not be used by software. Processors must perform the architected functional behavior of the instruction independent of the hint extension value (whether defined or unused), but different processor models may interpret unused opcode hint extension values in different ways, resulting in undesirable performance effects.

4.2 A-Unit Instruction Encodings

4.2.1 Integer ALU

All integer ALU instructions are encoded within major opcode 8 using a 2-bit opcode extension field in bits 35:34 (x_{2a}) and most have a second 2-bit opcode extension field in bits 28:27 (x_{2b}), a 4-bit opcode extension field in bits 32:29 (x_4), and a 1-bit reserved opcode extension field in bit 33 (v_e). Table 4-8 shows the 2-bit x_{2a} and 1-bit v_e assignments, Table 4-9 shows the integer ALU 4-bit+2-bit assignments, and Table 4-12 on page 3:306 shows the multimedia ALU 1-bit+2-bit assignments (which also share major opcode 8).

Table 4-8. Integer ALU 2-bit+1-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	
		0	1
8	0	Integer ALU 4-bit+2-bit Ext (Table 4-9)	🟪
	1	Multimedia ALU 1-bit+2-bit Ext (Table 4-12)	
	2	adds – imm_{14} A4	🟪
	3	addp4 – imm_{14} A4	🟪

Table 4-9. Integer ALU 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
				0	1	2	3
8	0	0	0	add A1	add +1 A1		
			1	sub -1 A1	sub A1		
			2	addp4 A1			
			3	and A1	andcm A1	or A1	xor A1
			4	shladd A2			
			5				
			6	shladdp4 A2			
			7				
			8				
			9		sub – imm_8 A3		
			A				
			B	and – imm_8 A3	andcm – imm_8 A3	or – imm_8 A3	xor – imm_8 A3
			C				
			D				
			E				
			F				

4.2.1.1 Integer ALU – Register-Register

A1

40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
8		x_{2a}	v_e	x_4	x_{2b}	r_3	r_2	r_1	qp
4	1	2	1	4	2	7	7	7	6

Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
add	$r_1 = r_2, r_3$	8	0	0	0	0
	$r_1 = r_2, r_3, 1$					1
sub	$r_1 = r_2, r_3$				1	1
	$r_1 = r_2, r_3, 1$					0
addp4	$r_1 = r_2, r_3$				2	0
and					3	0
andcm						1
or						2
xor						3

4.2.1.2 Shift Left and Add

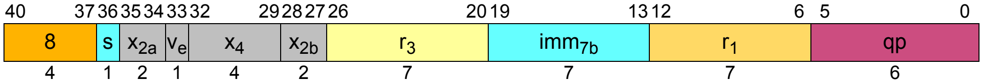
A2

40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
8		x_{2a}	v_e	x_4	ct_{2d}	r_3	r_2	r_1	qp
4	1	2	1	4	2	7	7	7	6

Instruction	Operands	Opcode	Extension		
			x_{2a}	v_e	x_4
shladd	$r_1 = r_2, count_2, r_3$	8	0	0	4
shladdp4					6

4.2.1.3 Integer ALU – Immediate$_8$-Register

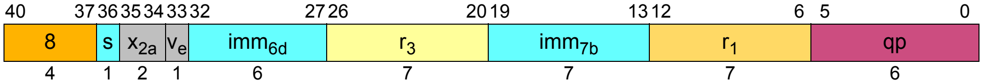
A3


40 37 36 35 34 33 32 29 28 27 26 20 19 13 12 6 5 0
8 | s | x_{2a} | v_e | x_4 | x_{2b} | r_3 | imm_{7b} | r_1 | qp
4 1 2 1 4 2 7 7 7 6

Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
sub	r_1 = imm_8, r_3	8	0	0	9	1
and					B	0
andcm						1
or						2
xor						3

4.2.1.4 Add Immediate$_{14}$

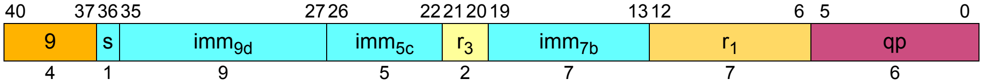
A4


40 37 36 35 34 33 32 27 26 20 19 13 12 6 5 0
8 | s | x_{2a} | v_e | imm_{6d} | r_3 | imm_{7b} | r_1 | qp
4 1 2 1 6 7 7 7 6

Instruction	Operands	Opcode	Extension	
			x_{2a}	v_e
adds	r_1 = imm_{14}, r_3	8	2	0
addp4			3	

4.2.1.5 Add Immediate$_{22}$

A5


40 37 36 35 27 26 22 21 20 19 13 12 6 5 0
9 | s | imm_{9d} | imm_{5c} | r_3 | imm_{7b} | r_1 | qp
4 1 9 5 2 7 7 6

Instruction	Operands	Opcode
addl	r_1 = imm_{22}, r_3	9

4.2.2 Integer Compare

The integer compare instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), and 12 (c), as shown in Table 4-10. The integer compare immediate instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (t_a) and 12 (c), as shown in Table 4-11.

Table 4-10. Integer Compare Opcode Extensions

x_2 Bits 35:34	t_b Bit 36	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.ltu A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.ltu.unc A6	cmp.eq.unc A6
		1	0	cmp.eq.and A6	cmp.eq.or A6	cmp.eq.or.andcm A6
			1	cmp.ne.and A6	cmp.ne.or A6	cmp.ne.or.andcm A6
	1	0	0	cmp.gt.and A7	cmp.gt.or A7	cmp.gt.or.andcm A7
			1	cmp.le.and A7	cmp.le.or A7	cmp.le.or.andcm A7
		1	0	cmp.ge.and A7	cmp.ge.or A7	cmp.ge.or.andcm A7
			1	cmp.lt.and A7	cmp.lt.or A7	cmp.lt.or.andcm A7
1	0	0	0	cmp4.lt A6	cmp4.ltu A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.ltu.unc A6	cmp4.eq.unc A6
		1	0	cmp4.eq.and A6	cmp4.eq.or A6	cmp4.eq.or.andcm A6
			1	cmp4.ne.and A6	cmp4.ne.or A6	cmp4.ne.or.andcm A6
	1	0	0	cmp4.gt.and A7	cmp4.gt.or A7	cmp4.gt.or.andcm A7
			1	cmp4.le.and A7	cmp4.le.or A7	cmp4.le.or.andcm A7
		1	0	cmp4.ge.and A7	cmp4.ge.or A7	cmp4.ge.or.andcm A7
			1	cmp4.lt.and A7	cmp4.lt.or A7	cmp4.lt.or.andcm A7

Table 4-11. Integer Compare Immediate Opcode Extensions

x_2 Bits 35:34	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm$_8$ A8	cmp.ltu – imm$_8$ A8	cmp.eq – imm$_8$ A8
		1	cmp.lt.unc – imm$_8$ A8	cmp.ltu.unc – imm$_8$ A8	cmp.eq.unc – imm$_8$ A8
	1	0	cmp.eq.and – imm$_8$ A8	cmp.eq.or – imm$_8$ A8	cmp.eq.or.andcm – imm$_8$ A8
		1	cmp.ne.and – imm$_8$ A8	cmp.ne.or – imm$_8$ A8	cmp.ne.or.andcm – imm$_8$ A8
3	0	0	cmp4.lt – imm$_8$ A8	cmp4.ltu – imm$_8$ A8	cmp4.eq – imm$_8$ A8
		1	cmp4.lt.unc – imm$_8$ A8	cmp4.ltu.unc – imm$_8$ A8	cmp4.eq.unc – imm$_8$ A8
	1	0	cmp4.eq.and – imm$_8$ A8	cmp4.eq.or – imm$_8$ A8	cmp4.eq.or.andcm – imm$_8$ A8
		1	cmp4.ne.and – imm$_8$ A8	cmp4.ne.or – imm$_8$ A8	cmp4.ne.or.andcm – imm$_8$ A8

4.2.2.1 Integer Compare – Register-Register

	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
A6	C - E	t_b	x_2	t_a	p_2	r_3	r_2	c	p_1	qp
	4	1	2	1	6	7	7	1	6	6

Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.lt	p_1, p_2 = r_2, r_3	C	0	0	0	0
cmp.ltu		D				
cmp.eq		E				
cmp.lt.unc		C				1
cmp.ltu.unc		D				
cmp.eq.unc		E				
cmp.eq.and		C			1	0
cmp.eq.or		D				
cmp.eq.or.andcm		E				
cmp.ne.and		C				1
cmp.ne.or		D				
cmp.ne.or.andcm		E				
cmp4.lt		C	1	0	0	0
cmp4.ltu		D				
cmp4.eq		E				
cmp4.lt.unc		C				1
cmp4.ltu.unc		D				
cmp4.eq.unc		E				
cmp4.eq.and		C			1	0
cmp4.eq.or		D				
cmp4.eq.or.andcm		E				
cmp4.ne.and		C				1
cmp4.ne.or		D				
cmp4.ne.or.andcm		E				

4.2.2.2 Integer Compare to Zero – Register

	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
A7	C - E	t_b	x_2	t_a	p_2	r_3	0	c	p_1	qp
	4	1	2	1	6	7	7	1	6	6

Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.gt.and	p_1, p_2 = r0, r_3	C	0	1	0	0
cmp.gt.or		D				
cmp.gt.or.andcm		E				
cmp.le.and		C				1
cmp.le.or		D				
cmp.le.or.andcm		E				
cmp.ge.and		C			1	0
cmp.ge.or		D				
cmp.ge.or.andcm		E				
cmp.lt.and		C				1
cmp.lt.or		D				
cmp.lt.or.andcm		E				
cmp4.gt.and		C	1		0	0
cmp4.gt.or		D				
cmp4.gt.or.andcm		E				
cmp4.le.and		C				1
cmp4.le.or		D				
cmp4.le.or.andcm		E				
cmp4.ge.and		C			1	0
cmp4.ge.or		D				
cmp4.ge.or.andcm		E				
cmp4.lt.and		C				1
cmp4.lt.or		D				
cmp4.lt.or.andcm		E				

4.2.2.3 Integer Compare – Immediate-Register

	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
A8	C - E	s	x_2	t_a	p_2	r_3	imm_{7b}	c	p_1	qp
	4	1	2	1	6	7	7	1	6	6

Instruction	Operands	Opcode	Extension		
			x_2	t_a	c
cmp.lt	$p_1, p_2 = imm_8, r_3$	C	2	0	0
cmp.ltu		D			
cmp.eq		E			
cmp.lt.unc		C			1
cmp.ltu.unc		D			
cmp.eq.unc		E			
cmp.eq.and		C		1	0
cmp.eq.or		D			
cmp.eq.or.andcm		E			
cmp.ne.and		C			1
cmp.ne.or		D			
cmp.ne.or.andcm		E			
cmp4.lt		C	3	0	0
cmp4.ltu		D			
cmp4.eq		E			
cmp4.lt.unc		C			1
cmp4.ltu.unc		D			
cmp4.eq.unc		E			
cmp4.eq.and		C		1	0
cmp4.eq.or		D			
cmp4.eq.or.andcm		E			
cmp4.ne.and		C			1
cmp4.ne.or		D			
cmp4.ne.or.andcm		E			

4.2.3 Multimedia

All multimedia ALU instructions are encoded within major opcode 8 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 2-bit opcode extension field in bits 35:34 (x_{2a}) as shown in Table 4-12. The multimedia ALU instructions also have a 4-bit opcode extension field in bits 32:29 (x_4), and a 2-bit opcode extension field in bits 28:27 (x_{2b}) as shown in Table 4-13 on page 3:307.

Table 4-12. Multimedia ALU 2-bit+1-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	
8	1	0	0	Multimedia ALU Size 1 (Table 4-13)
			1	Multimedia ALU Size 2 (Table 4-14)
		1	0	Multimedia ALU Size 4 (Table 4-15)
			1	

Table 4-13. Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	0	0	padd1 A9	padd1.sss A9	padd1.uuu A9	padd1.uus A9
				1	psub1 A9	psub1.sss A9	psub1.uuu A9	psub1.uus A9
				2			pavg1 A9	pavg1.raz A9
				3			pavgsub1 A9	
				4				
				5				
				6				
				7				
				8				
				9	pcmp1.eq A9	pcmp1.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-14. Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	1	0	padd2 A9	padd2.sss A9	padd2.uuu A9	padd2.uus A9
				1	psub2 A9	psub2.sss A9	psub2.uuu A9	psub2.uus A9
				2			pavg2 A9	pavg2.raz A9
				3			pavgsub2 A9	
				4	pshladd2 A10			
				5				
				6	pshradd2 A10			
				7				
				8				
				9	pcmp2.eq A9	pcmp2.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-15. Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

4.2.3.1 Multimedia ALU

	40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
A9	8	z_a	x_{2a}	z_b	x_4	x_{2b}	r_3	r_2	r_1	qp
	4	1	2	1	4	2	7	7	7	6

Instruction	Operands	Opcode	Extension				
			x_{2a}	z_a	z_b	x_4	x_{2b}
padd1	$r_1 = r_2, r_3$	8	1	0	0	0	0
padd2					1		
padd4				1	0		
padd1.sss				0	0		1
padd2.sss					1		
padd1.uuu				0	0		2
padd2.uuu					1		
padd1.uus				0	0		3
padd2.uus					1		
psub1				0	0	1	0
psub2					1		
psub4				1	0		
psub1.sss				0	0		1
psub2.sss					1		
psub1.uuu				0	0		2
psub2.uuu					1		
psub1.uus				0	0		3
psub2.uus					1		
pavg1				0	0	2	2
pavg2					1		
pavg1.raz				0	0		3
pavg2.raz					1		
pavgsub1				0	0	3	2
pavgsub2					1		
pcmp1.eq				0	0	9	0
pcmp2.eq					1		
pcmp4.eq				1	0		
pcmp1.gt				0	0		1
pcmp2.gt					1		
pcmp4.gt				1	0		

4.2.3.2 Multimedia Shift and Add

	40 37	36	35 34	33	32 29	28 27	26 20	19 13	12 6	5 0
A10	8	z_a	x_{2a}	z_b	x_4	ct_{2d}	r_3	r_2	r_1	qp
	4	1	2	1	4	2	7	7	7	6

Instruction	Operands	Opcode	Extension			
			x_{2a}	z_a	z_b	x_4
pshladd2	$r_1 = r_2, count_2, r_3$	8	1	0	1	4
pshradd2						6

4.3 I-Unit Instruction Encodings

4.3.1 Multimedia and Variable Shifts

All multimedia multiply/shift/max/min/mix/mux/pack/unpack and variable shift instructions are encoded within major opcode 7 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 1-bit reserved opcode extension in bit 32 (v_e) as shown in Table 4-16. They also have a 2-bit opcode extension field in bits 35:34 (x_{2a}) and a 2-bit field in bits 29:28 (x_{2b}) and most have a 2-bit field in bits 31:30 (x_{2c}) as shown in Table 4-17.

Table 4-16. Multimedia and Variable Shift 1-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	
			0	1
7	0	0	Multimedia Size 1 (Table 4-17)	
		1	Multimedia Size 2 (Table 4-18)	
	1	0	Multimedia Size 4 (Table 4-19)	
		1	Variable Shift (Table 4-20)	

Table 4-17. Multimedia Opcode 7 Size 1 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	0	0	0	0				
					1				
					2				
					3				
				1	0				
					1				
					2				
					3				
				2	0		unpack1.h I2	mix1.r I2	
					1	pmin1.u I2	pmax1.u I2		
					2		unpack1.l I2	mix1.l I2	
					3			psad1 I2	
				3	0				
					1				
					2			mux1 I3	
					3				

Table 4-18. Multimedia Opcode 7 Size 2 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	clz I9
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

Table 4-19. Multimedia Opcode 7 Size 4 2-bit Opcode Extensions

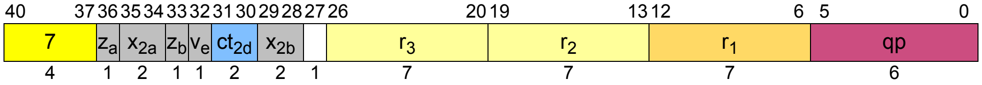
Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				mpy4 I2
					2	pshr4 – var I5			
					3				mpyshl4 I2
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

Table 4-20. Variable Shift Opcode 7 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u – var I5	shl – var I7		
					1				
					2	shr – var I5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

4.3.1.1 Multimedia Multiply and Shift

I1

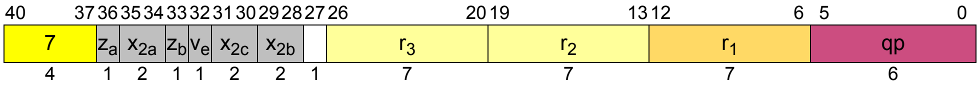


40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
7	z_a	x_{2a}	z_b	v_e	ct_{2d}	x_{2b}		r_3	r_2	r_1	qp
4	1	2	1	1	2	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension				
			z_a	z_b	v_e	x_{2a}	x_{2b}
pmpyshr2 pmpyshr2.u	$r_1 = r_2, r_3, count_2$	7	0	1	0	0	3 1

4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack

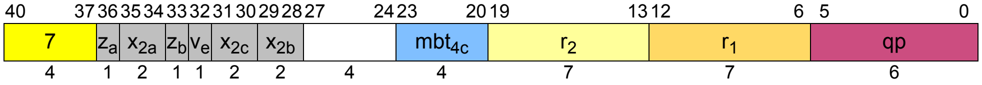
I2



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mpy4	$r_1 = r_2, r_3$	7	1	0	0	0	1	3
mpyshl4							3	
pmpy2.r			0	1		2	1	3
pmpy2.l							3	
mix1.r			0	0			0	2
mix2.r			0	1				
mix4.r			1	0				
mix1.l			0	0			2	
mix2.l			0	1				
mix4.l			1	0				
pack2.uss			0	1			0	0
pack2.sss			0	1			2	
pack4.sss			1	0				
unpack1.h			0	0			0	1
unpack2.h			0	1				
unpack4.h			1	0				
unpack1.l			0	0			2	
unpack2.l			0	1				
unpack4.l			1	0				
pmin1.u			0	0			1	0
pmax1.u								1
pmin2			0	1			3	0
pmax2								1
psad1			0	0			3	2

4.3.1.3 Multimedia Mux1

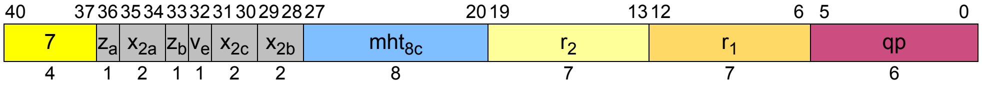
I3



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux1	$r_1 = r_2, mbtype_4$	7	0	0	0	3	2	2

4.3.1.4 Multimedia Mux2

I4



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux2	$r_1 = r_2, mhtype_8$	7	0	1	0	3	2	2

4.3.1.5 Shift Right – Variable

I5

40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	r_2	r_1	qp
4	1	2	1	1	2	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, r_2$	7	0	1	0	0	2	0
pshr4			1	0				
shr			1	1				
pshr2.u			0	1			0	
pshr4.u			1	0				
shr.u			1	1				

4.3.1.6 Multimedia Shift Right – Fixed

I6

40 37	36	35 34	33	32	31 30	29 28	27	26 20	19	18 14	13	12 6	5 0
7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3		$count_{5b}$		r_1	qp
4	1	2	1	1	2	2	1	7	1	5	1	7	6

Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, count_5$	7	0	1	0	1	3	0
pshr4			1	0				
pshr2.u			0	1			1	
pshr4.u			1	0				

4.3.1.7 Shift Left – Variable

I7

40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	r_2	r_1	qp
4	1	2	1	1	2	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, r_3$	7	0	1	0	0	0	1
pshl4			1	0				
shl			1	1				

4.3.1.8 Multimedia Shift Left – Fixed

I8

40 37	36	35 34	33	32	31 30	29 28	27 25	24 20	19 13	12 6	5 0
7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		$ccount_{5c}$	r_2	r_1	qp
4	1	2	1	1	2	2	3	5	7	7	6

Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, count_5$	7	0	1	0	3	1	1
pshl4			1	0				

4.3.1.9 Bit Strings

	40 37	36	35 34	33	32	31 30	29 28	27	26 20	19 13	12 6	5 0
I9	7	z_a	x_{2a}	z_b	v_e	x_{2c}	x_{2b}		r_3	0	r_1	qp
	4	1	2	1	1	2	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2
clz								3

4.3.2 Integer Shifts

The integer shift, test bit, and test NaT instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) and a 1-bit opcode extension field in bit 33 (x). The extract and test bit instructions also have a 1-bit opcode extension field in bit 13 (y). Table 4-21 shows the test bit, extract, and shift right pair assignments.

Table 4-21. Integer Shift/Test Bit/Test NaT 2-bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 13	
			0	1
5	0	0	Test Bit (Table 4-23)	Test NaT/Test Feature (Table 4-23)
	1		extr.u I11	extr I11
	2			
	3		shrp I10	

Most deposit instructions also have a 1-bit opcode extension field in bit 26 (y). Table 4-22 shows these assignments.

Table 4-22. Deposit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 26	
			0	1
5	0	1	Test Bit/Test NaT/Test Feature (Table 4-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

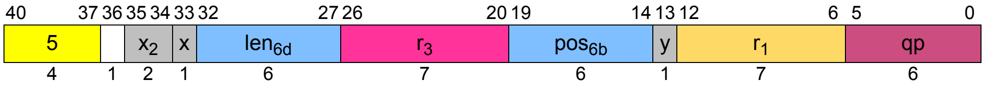
4.3.2.1 Shift Right Pair

	40 37	36	35 34	33	32 27	26 20	19 13	12 6	5 0
I10	5		x_2	x	$count_{6d}$	r_3	r_2	r_1	qp
	4	1	2	1	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

4.3.2.2 Extract

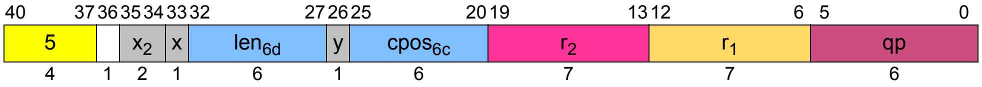
I11



Instruction	Operands	Opcode	Extension		
			x_2	x	y
extr.u	$r_1 = r_3, pos_6, len_6$	5	1	0	0
extr					1

4.3.2.3 Zero and Deposit

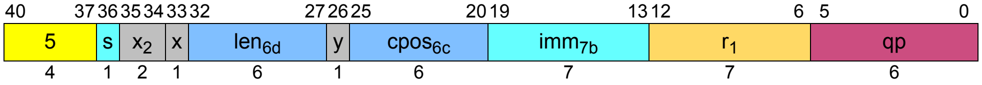
I12



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

4.3.2.4 Zero and Deposit Immediate$_8$

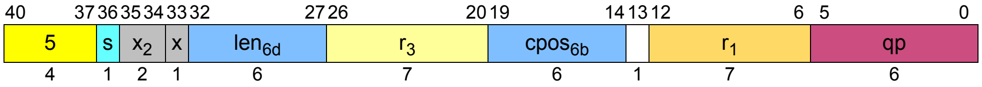
I13



Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

4.3.2.5 Deposit Immediate$_1$

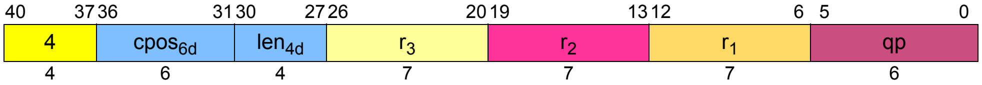
I14



Instruction	Operands	Opcode	Extension	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

4.3.2.6 Deposit

I15



Instruction	Operands	Opcode
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

4.3.3 Test Bit

All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) plus five 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), 12 (c), 13 (y) and 19 (x). Table 4-23 summarizes these assignments.

Table 4-23. Test Bit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	t_a Bit 33	t_b Bit 36	c Bit 12	y Bit 13	x Bit 19	
						0	1
5	0	0	0	0	0	tbit.z I16	
					1	tnat.z I17	tf.z I30
				1	0	tbit.z.unc I16	
					1	tnat.z.unc I17	tf.z.unc I30
			1	0	0	tbit.z.and I16	
					1	tnat.z.and I17	tf.z.and I30
				1	0	tbit.nz.and I16	
					1	tnat.nz.and I17	tf.nz.and I30
		1	0	0	0	tbit.z.or I16	
					1	tnat.z.or I17	tf.z.or I30
				1	0	tbit.nz.or I16	
					1	tnat.nz.or I17	tf.nz.or I30
			1	0	0	tbit.z.or.andcm I16	
					1	tnat.z.or.andcm I17	tf.z.or.andcm I30
				1	0	tbit.nz.or.andcm I16	
					1	tnat.nz.or.andcm I17	tf.nz.or.andcm I30

4.3.3.1 Test Bit

	40 37	36	35 34	33	32 27	26 20	19 14	13	12	11 6	5 0
I16	5	t_b	x_2	t_a	p_2	r_3	pos_{6b}	y	c	p_1	qp
	4	1	2	1	6	7	6	1	1	6	6

Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tbit.z	$p_1, p_2 = r_3, pos_6$	5	0	0	0	0	0
tbit.z.unc							1
tbit.z.and					1		0
tbit.nz.and							1
tbit.z.or				1	0		0
tbit.nz.or							1
tbit.z.or.andcm					1		0
tbit.nz.or.andcm							1

4.3.3.2 Test NaT

	40 37	36	35 34	33	32 27	26 20	19	18 14	13	12	11 6	5 0
I17	5	t_b	x_2	t_a	p_2	r_3	x		y	c	p_1	qp
	4	1	2	1	6	7	1	5	1	1	6	6

Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tnat.z	p_1, p_2 = r_3	5	0	0	0	1	0	0
tnat.z.unc								1
tnat.z.and					1			0
tnat.nz.and								1
tnat.z.or				1	0			0
tnat.nz.or								1
tnat.z.or.andcm					1			0
tnat.nz.or.andcm								1

4.3.4 Miscellaneous I-Unit Instructions

The miscellaneous I-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33. Some also have a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-24 shows the 3-bit assignments and Table 4-25 summarizes the 6-bit assignments.

Table 4-24. Misc I-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm_{44} I24
	3	mov to pr I23
	4	
	5	
	6	
	7	mov to b I21

Table 4-25. Misc I-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	1-bit Ext (Table 4-26)	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm_8 I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

4.3.4.1 Nop/Hint (I-Unit)

I-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-26.

Table 4-26. Misc I-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.i
			1	hint.i

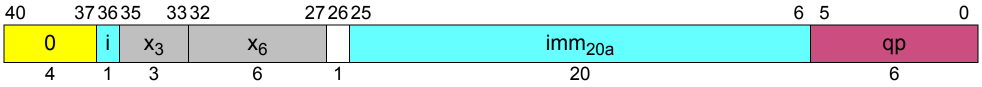
I18

40 37	36	35 33	32 27	26	25 6	5 0
0	i	x_3	x_6	y	imm_{20a}	qp
4	1	3	6	1	20	6

Instruction	Operands	Opcode	Extension		
			x_3	x_6	y
nop.i [i]	imm_{21}	0	0	01	0
hint.i					1

4.3.4.2 Break (I-Unit)

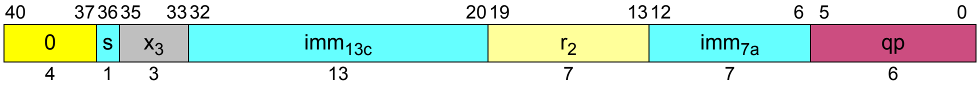
I19



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.i[i]	*imm_{21}*	0	0	00

4.3.4.3 Integer Speculation Check (I-Unit)

I20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.i	*r_2, $target_{25}$*	0	1

4.3.5 GR/BR Moves

The GR/BR move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 3:318 for a summary of the opcode extensions. The mov to BR instruction uses a 2-bit "whether" prediction hint field in bits 21:20 (wh) as shown in Table 4-27.

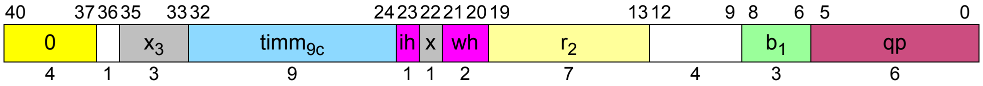
Table 4-27. Move to BR Whether Hint Completer

wh Bits 21:20	*mwh*
0	.sptk
1	*none*
2	.dptk
3	

The mov to BR instruction also uses a 1-bit opcode extension field (x) in bit 22 to distinguish the return form from the normal form, and a 1-bit hint extension in bit 23 (ih) (see Table 4-56 on page 3:354).

4.3.5.1 Move to BR

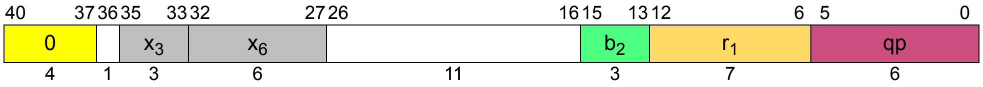
I21



Instruction	Operands	Opcode	Extension			
			x_3	x	ih	wh
mov.*mwh.ih*	*b_1 = r_2, tag_{13}*	0	7	0	See Table 4-56 on page 3:354	See Table 4-27 on page 3:320
mov.ret.*mwh.ih*				1		

4.3.5.2 Move from BR

I22



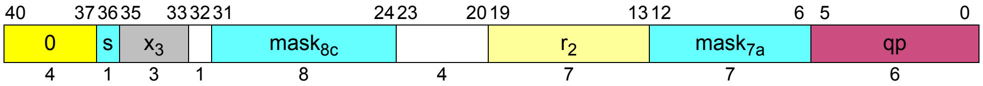
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

4.3.6 GR/Predicate/IP Moves

The GR/Predicate/IP move instructions are encoded in major opcode 0. See "Miscellaneous I-Unit Instructions" on page 3:318 for a summary of the opcode extensions.

4.3.6.1 Move to Predicates – Register

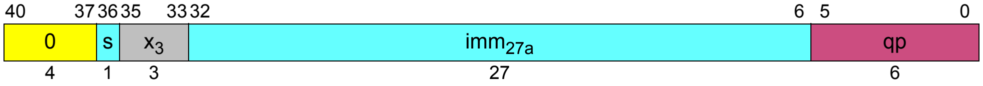
I23



Instruction	Operands	Opcode	Extension
			x_3
mov	pr = r_2, $mask_{17}$	0	3

4.3.6.2 Move to Predicates – Immediate$_{44}$

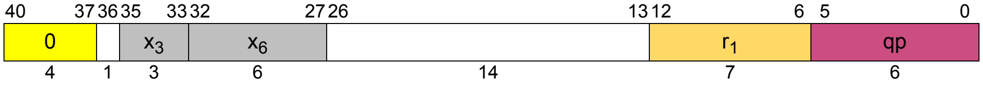
I24



Instruction	Operands	Opcode	Extension
			x_3
mov	pr.rot = imm_{44}	0	2

4.3.6.3 Move from Predicates/IP

I25



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = ip r_1 = pr	0	0	30 33

4.3.7 GR/AR Moves (I-Unit)

The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using system/memory management instructions on the M-unit. See "GR/AR Moves (M-Unit)" on page 3:342.) See "Miscellaneous I-Unit Instructions" on page 3:318 for a summary of the I-Unit GR/AR opcode extensions.

4.3.7.1 Move to AR – Register (I-Unit)

I26

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
0		x_3	x_6	ar_3	r_2		qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

4.3.7.2 Move to AR – Immediate$_8$ (I-Unit)

I27

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
0	s	x_3	x_6	ar_3	imm_{7b}		qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

4.3.7.3 Move from AR (I-Unit)

I28

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
0		x_3	x_6	ar_3		r_1	qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

4.3.8 Sign/Zero Extend/Compute Zero Index

I29

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
0		x_3	x_6	r_3		r_1	qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
zxt1	$r_1 = r_3$	0	0	10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

4.3.9 Test Feature

	40 37	36	35 34	33	32 27	26 20	19	18 14	13	12	11 6	5 0
I30	5	t_b	x_2	t_a	p_2	0	x	imm_{5b}	y	c	p_1	qp
	4	1	2	1	6	7	1	5	1	1	6	6

Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tf.z	p_1, p_2 = imm_5	5	0	0	0	1	1	0
tf.z.unc								1
tf.z.and					1			0
tf.nz.and								1
tf.z.or				1	0			0
tf.nz.or								1
tf.z.or.andcm					1			0
tf.nz.or.andcm								1

4.4 M-Unit Instruction Encodings

4.4.1 Loads and Stores

All load and store instructions are encoded within major opcodes 4, 5, 6, and 7 using a 6-bit opcode extension field in bits 35:30 (x_6). Instructions in major opcode 4 (integer load/store, semaphores, and get FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-28. Instructions in major opcode 6 (floating-point load/store, load pair, and set FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-29.

Table 4-28. Integer Load/Store/Semaphore/Get FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
4	0	0	Load/Store (Table 4-30)
	0	1	Semaphore/get FR (Table 4-33)
	1	0	Load +Reg (Table 4-31)
	1	1	

Table 4-29. Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
6	0	0	FP Load/Store (Table 4-34)
	0	1	FP Load Pair/set FR (Table 4-37)
	1	0	FP Load +Reg (Table 4-35)
	1	1	FP Load Pair +Imm (Table 4-38)

The integer load/store opcode extensions are summarized in Table 4-30 on page 3:324, Table 4-31 on page 3:324, and Table 4-32 on page 3:325, and the semaphore and get FR opcode extensions in Table 4-33 on page 3:325. The floating-point load/store opcode extensions are summarized in Table 4-34 on page 3:326, Table 4-35 on page 3:326, and Table 4-36 on page 3:327, the floating-point load pair and set FR opcode extensions in Table 4-37 on page 3:327 and Table 4-38 on page 3:328.

Table 4-30. Integer Load/Store Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C	st1 M6	st2 M6	st4 M6	st8 M6
			D	st1.rel M6	st2.rel M6	st4.rel M6	st8.rel M6
			E				st8.spill M6
			F				

Table 4-31. Integer Load +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	1	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C				
			D				
			E				
			F				

Table 4-32. Integer Load/Store +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
5	0	ld1 M3	ld2 M3	ld4 M3	ld8 M3
	1	ld1.s M3	ld2.s M3	ld4.s M3	ld8.s M3
	2	ld1.a M3	ld2.a M3	ld4.a M3	ld8.a M3
	3	ld1.sa M3	ld2.sa M3	ld4.sa M3	ld8.sa M3
	4	ld1.bias M3	ld2.bias M3	ld4.bias M3	ld8.bias M3
	5	ld1.acq M3	ld2.acq M3	ld4.acq M3	ld8.acq M3
	6				ld8.fill M3
	7				
	8	ld1.c.clr M3	ld2.c.clr M3	ld4.c.clr M3	ld8.c.clr M3
	9	ld1.c.nc M3	ld2.c.nc M3	ld4.c.nc M3	ld8.c.nc M3
	A	ld1.c.clr.acq M3	ld2.c.clr.acq M3	ld4.c.clr.acq M3	ld8.c.clr.acq M3
	B				
	C	st1 M5	st2 M5	st4 M5	st8 M5
	D	st1.rel M5	st2.rel M5	st4.rel M5	st8.rel M5
	E				st8.spill M5
	F				

Table 4-33. Semaphore/Get FR/16-Byte Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8	cmp8xchg16.acq M16			
			9	cmp8xchg16.rel M16			
			A	ld16 M2			
			B	ld16.acq M2			
			C	st16 M6			
			D	st16.rel M6			
			E				
			F				

Table 4-34. Floating-point Load/Store/Lfetch Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M9	ldf8 M9	ldfs M9	ldfd M9
			1	ldfe.s M9	ldf8.s M9	ldfs.s M9	ldfd.s M9
			2	ldfe.a M9	ldf8.a M9	ldfs.a M9	ldfd.a M9
			3	ldfe.sa M9	ldf8.sa M9	ldfs.sa M9	ldfd.sa M9
			4				
			5				
			6				ldf.fill M9
			7				
			8	ldfe.c.clr M9	ldf8.c.clr M9	ldfs.c.clr M9	ldfd.c.clr M9
			9	ldfe.c.nc M9	ldf8.c.nc M9	ldfs.c.nc M9	ldfd.c.nc M9
			A				
			B	lfetch M18	lfetch.excl M18	lfetch.fault M18	lfetch.fault.excl M18
			C	stfe M13	stf8 M13	stfs M13	stfd M13
			D				
			E				stf.spill M13
			F				

Table 4-35. Floating-point Load/Lfetch +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M20	lfetch.excl M20	lfetch.fault M20	lfetch.fault.excl M20
			C				
			D				
			E				
			F				

Table 4-36. Floating-point Load/Store/Lfetch +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M22	lfetch.excl M22	lfetch.fault M22	lfetch.fault.excl M22
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

Table 4-37. Floating-point Load Pair/Set FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	1	0		ldfp8 M11	ldfps M11	ldfpd M11
			1		ldfp8.s M11	ldfps.s M11	ldfpd.s M11
			2		ldfp8.a M11	ldfps.a M11	ldfpd.a M11
			3		ldfp8.sa M11	ldfps.sa M11	ldfpd.sa M11
			4				
			5				
			6				
			7	setf.sig M18	setf.exp M18	setf.s M18	setf.d M18
			8		ldfp8.c.clr M11	ldfps.c.clr M11	ldfpd.c.clr M11
			9		ldfp8.c.nc M11	ldfps.c.nc M11	ldfpd.c.nc M11
			A				
			B				
			C				
			D				
			E				
			F				

Table 4-38. Floating-point Load Pair +Imm Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

The load and store instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint). Table 4-39 and Table 4-40 summarize these assignments.

Table 4-39. Load Hint Completer

hint Bits 29:28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

Table 4-40. Store Hint Completer

hint Bits 29:28	*sthint*
0	*none*
1	
2	
3	.nta

4.4.1.1 Integer Load

M2

40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
4	m	x_6	hint	x	r_3		r_1	qp
4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	r_1 = [r_3]	4	0	0	00	See Table 4-39 on page 3:328
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	
ld16.*ldhint*	r_1, ar.csd = [r_3]		0	1	28	
ld16.acq.*ldhint*					2C	

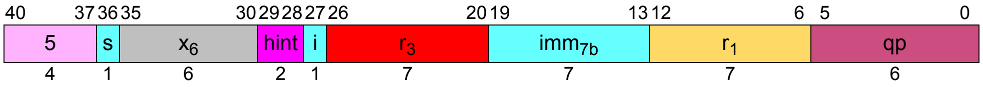
4.4.1.2 Integer Load – Increment by Register

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M2	4	m	x_6	hint	x	r_3	r_2	r_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	r_1 = [r_3], r_2	4	1	0	00	See Table 4-39 on page 3:328
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	

4.4.1.3 Integer Load – Increment by Immediate

M3



Instruction	Operands	Opcode	Extension	
			x_6	hint
ld1.*ldhint*	r_1 = [r_3], imm_9	5	00	See Table 4-39 on page 3:328
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			0A	
ld8.a.*ldhint*			0B	
ld1.sa.*ldhint*			0C	
ld2.sa.*ldhint*			0D	
ld4.sa.*ldhint*			0E	
ld8.sa.*ldhint*			0F	
ld1.bias.*ldhint*			10	
ld2.bias.*ldhint*			11	
ld4.bias.*ldhint*			12	
ld8.bias.*ldhint*			13	
ld1.acq.*ldhint*			14	
ld2.acq.*ldhint*			15	
ld4.acq.*ldhint*			16	
ld8.acq.*ldhint*			17	
ld8.fill.*ldhint*			1B	
ld1.c.clr.*ldhint*			20	
ld2.c.clr.*ldhint*			21	
ld4.c.clr.*ldhint*			22	
ld8.c.clr.*ldhint*			23	
ld1.c.nc.*ldhint*			24	
ld2.c.nc.*ldhint*			25	
ld4.c.nc.*ldhint*			26	
ld8.c.nc.*ldhint*			27	
ld1.c.clr.acq.*ldhint*			28	
ld2.c.clr.acq.*ldhint*			29	
ld4.c.clr.acq.*ldhint*			2A	
ld8.c.clr.acq.*ldhint*			2B	

4.4.1.4 Integer Store

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M6	4	m	x_6	hint	x	r_3	r_2		qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
st1.*sthint*	$[r_3] = r_2$	4	0	0	30	See Table 4-40 on page 3:328
st2.*sthint*					31	
st4.*sthint*					32	
st8.*sthint*					33	
st1.rel.*sthint*					34	
st2.rel.*sthint*					35	
st4.rel.*sthint*					36	
st8.rel.*sthint*					37	
st8.spill.*sthint*					3B	
st16.*sthint*	$[r_3] = r_2$, ar.csd		0	1	30	
st16.rel.*sthint*					34	

4.4.1.5 Integer Store – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M5	5	s	x_6	hint	i	r_3	r_2	imm_{7a}	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_6	hint
st1.*sthint*	$[r_3] = r_2$, imm_9	5	30	See Table 4-40 on page 3:328
st2.*sthint*			31	
st4.*sthint*			32	
st8.*sthint*			33	
st1.rel.*sthint*			34	
st2.rel.*sthint*			35	
st4.rel.*sthint*			36	
st8.rel.*sthint*			37	
st8.spill.*sthint*			3B	

4.4.1.6 Floating-point Load

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M9	6	m	x_6	hint	x	r_3		f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	f_1 = [r_3]	6	0	0	02	See Table 4-39 on page 3:328
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.7 Floating-point Load – Increment by Register

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M7	6	m	x_6	hint	x	r_3	r_2	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	$f_1 = [r_3], r_2$	6	1	0	02	See Table 4-39 on page 3:328
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.8 Floating-point Load – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M8	7	s	x_6	hint	i	r_3	imm_{7b}	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_6	hint
ldfs.*ldhint*	f_1 = [r_3], imm_9	7	02	See Table 4-39 on page 3:328
ldfd.*ldhint*			03	
ldf8.*ldhint*			01	
ldfe.*ldhint*			00	
ldfs.s.*ldhint*			06	
ldfd.s.*ldhint*			07	
ldf8.s.*ldhint*			05	
ldfe.s.*ldhint*			04	
ldfs.a.*ldhint*			0A	
ldfd.a.*ldhint*			0B	
ldf8.a.*ldhint*			09	
ldfe.a.*ldhint*			08	
ldfs.sa.*ldhint*			0E	
ldfd.sa.*ldhint*			0F	
ldf8.sa.*ldhint*			0D	
ldfe.sa.*ldhint*			0C	
ldf.fill.*ldhint*			1B	
ldfs.c.clr.*ldhint*			22	
ldfd.c.clr.*ldhint*			23	
ldf8.c.clr.*ldhint*			21	
ldfe.c.clr.*ldhint*			20	
ldfs.c.nc.*ldhint*			26	
ldfd.c.nc.*ldhint*			27	
ldf8.c.nc.*ldhint*			25	
ldfe.c.nc.*ldhint*			24	

4.4.1.9 Floating-point Store

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M13	6	m	x_6	hint	x	r_3	f_2		qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
stfs.*sthint*	[r_3] = f_2	6	0	0	32	See Table 4-40 on page 3:328
stfd.*sthint*					33	
stf8.*sthint*					31	
stfe.*sthint*					30	
stf.spill.*sthint*					3B	

4.4.1.10 Floating-point Store – Increment by Immediate

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M10	7	s	x_6	hint	i	r_3	f_2	imm_{7a}	qp
	4	1	6	2	1	7	7	7	6

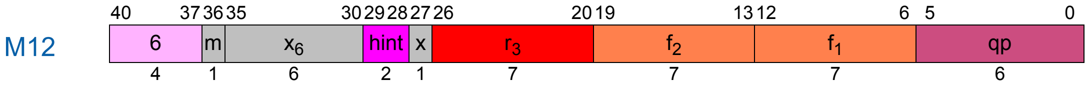
Instruction	Operands	Opcode	Extension	
			x_6	hint
stfs.*sthint*	[r_3] = f_2, imm_9	7	32	See Table 4-40 on page 3:328
stfd.*sthint*			33	
stf8.*sthint*			31	
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

4.4.1.11 Floating-point Load Pair

	40 37	36	35 30	29 28	27	26 20	19 13	12 6	5 0
M11	6	m	x_6	hint	x	r_3	f_2	f_1	qp
	4	1	6	2	1	7	7	7	6

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	f_1, f_2 = [r_3]	6	0	1	02	See Table 4-39 on page 3:328
ldfpd.*ldhint*					03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*					06	
ldfpd.s.*ldhint*					07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*					0A	
ldfpd.a.*ldhint*					0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*					0E	
ldfpd.sa.*ldhint*					0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*					22	
ldfpd.c.clr.*ldhint*					23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*					26	
ldfpd.c.nc.*ldhint*					27	
ldfp8.c.nc.*ldhint*					25	

4.4.1.12 Floating-point Load Pair – Increment by Immediate



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	f_1, f_2 = [r_3], 8	6	1	1	02	See Table 4-39 on page 3:328
ldfpd.*ldhint*	f_1, f_2 = [r_3], 16				03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*	f_1, f_2 = [r_3], 8				06	
ldfpd.s.*ldhint*	f_1, f_2 = [r_3], 16				07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*	f_1, f_2 = [r_3], 8				0A	
ldfpd.a.*ldhint*	f_1, f_2 = [r_3], 16				0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*	f_1, f_2 = [r_3], 8				0E	
ldfpd.sa.*ldhint*	f_1, f_2 = [r_3], 16				0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*	f_1, f_2 = [r_3], 8				22	
ldfpd.c.clr.*ldhint*	f_1, f_2 = [r_3], 16				23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*	f_1, f_2 = [r_3], 8				26	
ldfpd.c.nc.*ldhint*	f_1, f_2 = [r_3], 16				27	
ldfp8.c.nc.*ldhint*					25	

4.4.2 Line Prefetch

The line prefetch instructions are encoded in major opcodes 6 and 7 along with the floating-point load/store instructions. See "Loads and Stores" on page 3:323 for a summary of the opcode extensions.

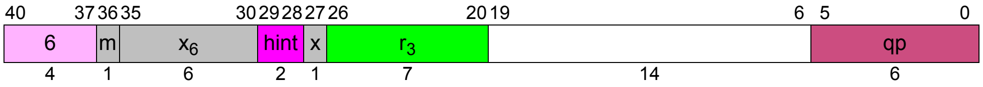
The line prefetch instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint) as shown in Table 4-44.

Table 4-41. Line Prefetch Hint Completer

hint Bits 29:28	*lfhint*
0	*none*
1	.nt1
2	.nt2
3	.nta

4.4.2.1 Line Prefetch

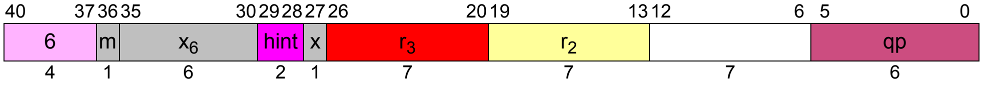
M13



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.excl.*lfhint*	[r_3]	6	0	0	2D	See Table 4-41 on page 3:337
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

4.4.2.2 Line Prefetch – Increment by Register

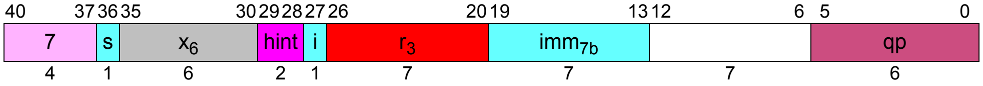
M14



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
lfetch.*lfhint*	[r_3], r_2	6	1	0	2C	See Table 4-41 on page 3:337
lfetch.excl.*lfhint*					2D	
lfetch.fault.*lfhint*					2E	
lfetch.fault.excl.*lfhint*					2F	

4.4.2.3 Line Prefetch – Increment by Immediate

M15



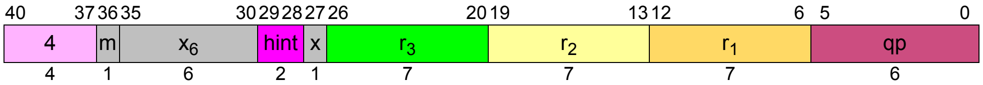
Instruction	Operands	Opcode	Extension	
			x_6	hint
lfetch.*lfhint*	[r_3], imm_9	7	2C	See Table 4-41 on page 3:337
lfetch.excl.*lfhint*			2D	
lfetch.fault.*lfhint*			2E	
lfetch.fault.excl.*lfhint*			2F	

4.4.3 Semaphores

The semaphore instructions are encoded in major opcode 4 along with the integer load/store instructions. See "Loads and Stores" on page 3:323 for a summary of the opcode extensions. These instructions have the same cache locality opcode hint extension field in bits 29:28 (hint) as load instructions. See Table 4-39, "Load Hint Completer" on page 3:328.

4.4.3.1 Exchange/Compare and Exchange

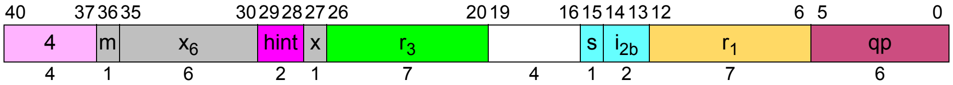
M16



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
cmpxchg1.acq.*ldhint*	r_1 = [r_3], r_2, ar.ccv	4	0	1	00	See Table 4-39 on page 3:328
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
cmp8xchg16.acq.*ldhint*	r_1 = [r_3], r_2, ar.csd, ar.ccv				20	
cmp8xchg16.rel.*ldhint*					24	
xchg1.*ldhint*	r_1 = [r_3], r_2				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

4.4.3.2 Fetch and Add – Immediate

M17



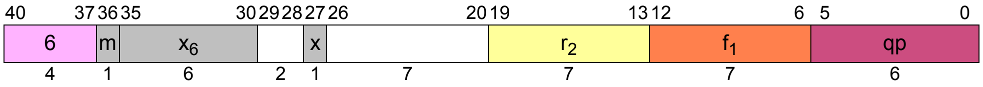
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
fetchadd4.acq.*ldhint*	r_1 = [r_3], *inc*$_3$	4	0	1	12	See Table 4-39 on page 3:328
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

4.4.4 Set/Get FR

The set FR instructions are encoded in major opcode 6 along with the floating-point load/store instructions. The get FR instructions are encoded in major opcode 4 along with the integer load/store instructions. See "Loads and Stores" on page 3:323 for a summary of the opcode extensions.

4.4.4.1 Set FR

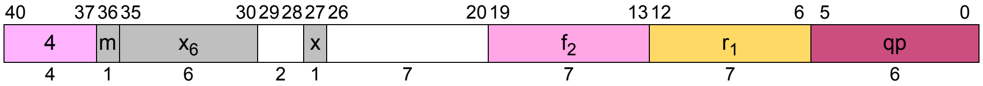
M18



Instruction	Operands	Opcode	Extension		
			m	x	x_6
setf.sig	$f_1 = r_2$	6	0	1	1C
setf.exp					1D
setf.s					1E
setf.d					1F

4.4.4.2 Get FR

M19



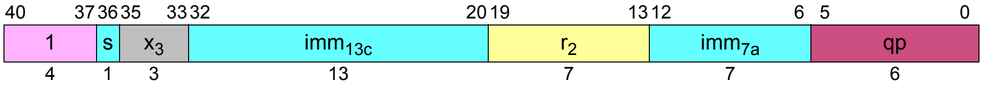
Instruction	Operands	Opcode	Extension		
			m	x	x_6
getf.sig	$r_1 = f_2$	4	0	1	1C
getf.exp					1D
getf.s					1E
getf.d					1F

4.4.5 Speculation and Advanced Load Checks

The speculation and advanced load check instructions are encoded in major opcodes 0 and 1 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.5.1 Integer Speculation Check (M-Unit)

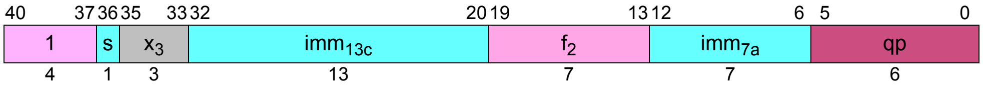
M20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.m	r_2, $target_{25}$	1	1

4.4.5.2 Floating-point Speculation Check

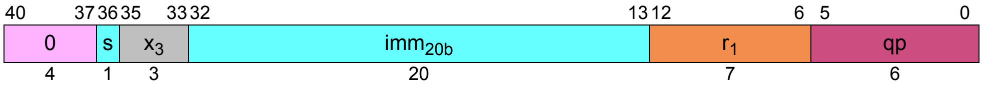
M21



Instruction	Operands	Opcode	Extension
			x_3
chk.s	f_2, $target_{25}$	1	3

4.4.5.3 Integer Advanced Load Check

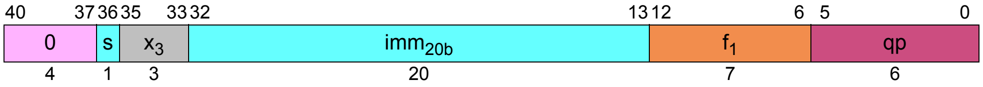
M22



Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	r_1, *target*$_{25}$	0	4
chk.a.clr			5

4.4.5.4 Floating-point Advanced Load Check

M23



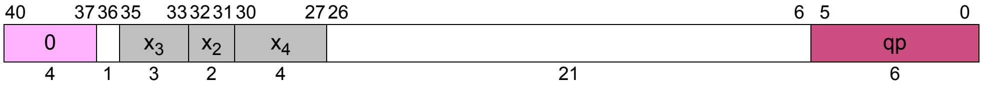
Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	f_1, *target*$_{25}$	0	6
chk.a.clr			7

4.4.6 Cache/Synchronization/RSE/ALAT

The cache/synchronization/RSE/ALAT instructions are encoded in major opcode 0 along with the memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.6.1 Sync/Fence/Serialize/ALAT Control

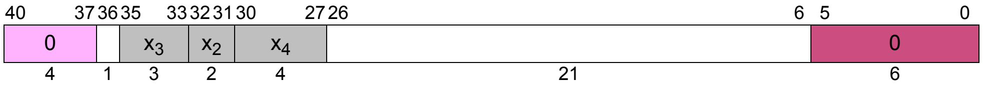
M24



Instruction	Opcode	Extension		
		x_3	x_4	x_2
invala	0	0	0	1
fwb			0	2
mf			2	
mf.a			3	
srlz.d			0	3
srlz.i			1	
sync.i			3	

4.4.6.2 RSE Control

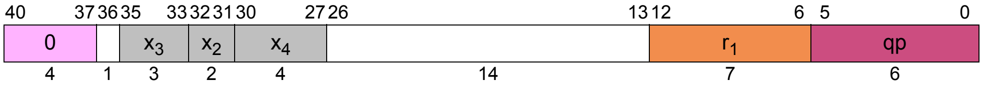
M25



Instruction	Opcode	Extension		
		x_3	x_4	x_2
flushrs ^{f}	0	0	C	0
loadrs ^{f}			A	

4.4.6.3 Integer ALAT Entry Invalidate

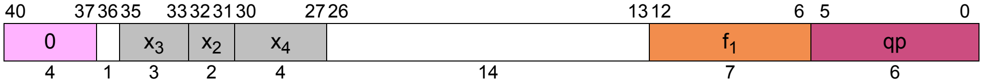
M26



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

4.4.6.4 Floating-point ALAT Entry Invalidate

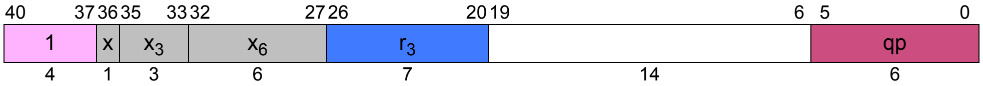
M27



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1

4.4.6.5 Flush Cache

M28



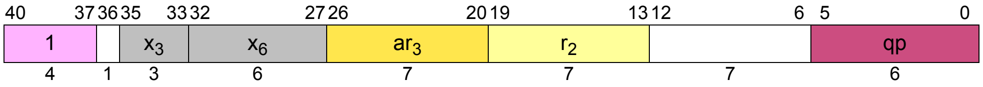
Instruction	Operands	Opcode	Extension		
			x_3	x_6	x
fc	r_3	1	0	30	0
fc.i					1

4.4.7 GR/AR Moves (M-Unit)

The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the system/memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See "GR/AR Moves (I-Unit)" on page 3:321.) See "System/Memory Management" on page 3:345 for a summary of the M-Unit GR/AR opcode extensions.

4.4.7.1 Move to AR – Register (M-Unit)

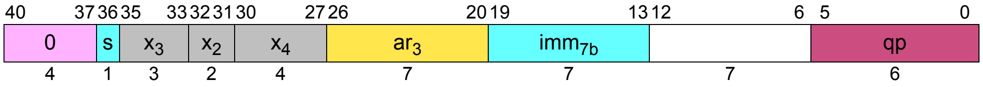
M29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

4.4.7.2 Move to AR – Immediate$_8$ (M-Unit)

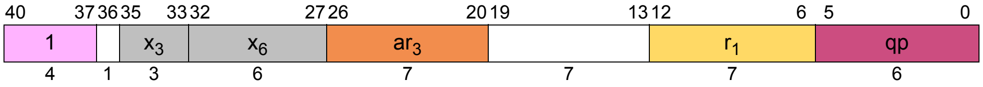
M30



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
mov.m	$ar_3 = imm_8$	0	0	8	2

4.4.7.3 Move from AR (M-Unit)

M31



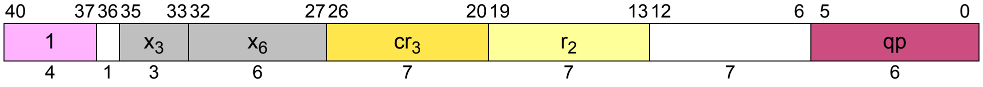
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

4.4.8 GR/CR Moves

The GR/CR move instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.8.1 Move to CR

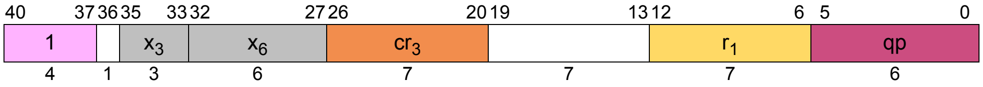
M32



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$cr_3 = r_2$	1	0	2C

4.4.8.2 Move from CR

M33



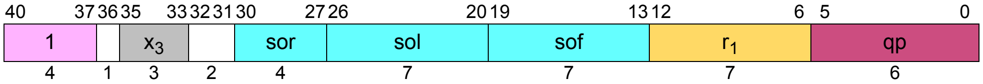
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	$r_1 = cr_3$	1	0	24

4.4.9 Miscellaneous M-Unit Instructions

The miscellaneous M-unit instructions are encoded in major opcode 0 along with the system/memory management instructions. See "System/Memory Management" on page 3:345 for a summary of the opcode extensions.

4.4.9.1 Allocate Register Stack Frame

M34

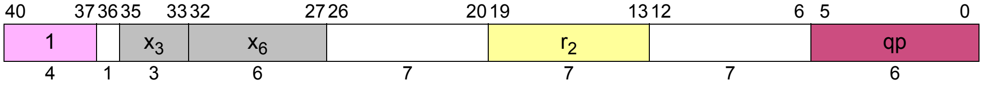


Instruction	Operands	Opcode	Extension
			x_3
alloc ^{f}	r_1 = ar.pfs, i, l, o, r	1	6

Note: The three immediates in the instruction encoding are formed from the operands as follows:
sof = $i + l + o$
sol = $i + l$
sor = r >> 3

4.4.9.2 Move to PSR

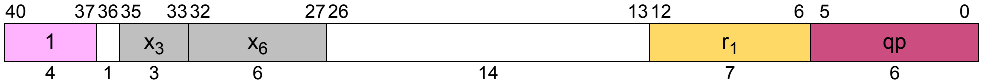
M35



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	psr.l = r_2	1	0	2D
mov	psr.um = r_2			29

4.4.9.3 Move from PSR

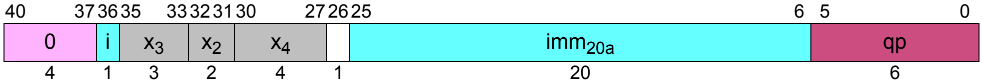
M36



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	r_1 = psr	1	0	25
mov	r_1 = psr.um			21

4.4.9.4 Break (M-Unit)

M37



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
break.m	imm_{21}	0	0	0	0

4.4.10 System/Memory Management

All system/memory management instructions are encoded within major opcodes 0 and 1 using a 3-bit opcode extension field (x_3) in bits 35:33. Some instructions also have a 4-bit opcode extension field (x_4) in bits 30:27, or a 6-bit opcode extension field (x_6) in bits 32:27. Most of the instructions having a 4-bit opcode extension field also have a 2-bit extension field (x_2) in bits 32:31. Table 4-42 shows the 3-bit assignments for opcode 0, Table 4-43 summarizes the 4-bit+2-bit assignments for opcode 0, Table 4-44 shows the 3-bit assignments for opcode 1, and Table 4-45 summarizes the 6-bit assignments for opcode 1.

Table 4-42. Opcode 0 System/Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	System/Memory Management 4-bit+2-bit Ext (Table 4-43)
	1	
	2	
	3	
	4	chk.a.nc – int M22
	5	chk.a.clr – int M22
	6	chk.a.nc – fp M23
	7	chk.a.clr – fp M23

Table 4-43. Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31			
			0	1	2	3
0	0	0	break.m M37	invala M24	fwb M24	srlz.d M24
		1	1-bit Ext (Table 4-46)			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6	ssm M44			
		7	rsm M44			
		8			mov.m to ar – imm_8 M30	
		9				
		A	loadrs M25			
		B				
		C	flushrs M25			
		D				
		E				
		F				

Table 4-44. Opcode 1 System/Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
1	0	System/Memory Management 6-bit Ext (Table 4-45)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

Table 4-45. Opcode 1 System/Memory Management 6-bit Opcode Extensions

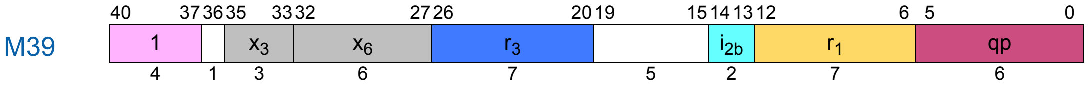
Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0	mov to rr M42	mov from rr M43		fc M28
		1	mov to dbr M42	mov from dbr M43	mov from psr.um M36	probe.rw.fault – imm_2 M40
		2	mov to ibr M42	mov from ibr M43	mov.m from ar M31	probe.r.fault – imm_2 M40
		3	mov to pkr M42	mov from pkr M43		probe.w.fault – imm_2 M40
		4	mov to pmc M42	mov from pmc M43	mov from cr M33	ptc.e M47
		5	mov to pmd M42	mov from pmd M43	mov from psr M36	
		6				
		7		mov from cpuid M43		
		8		probe.r – imm_2 M39		probe.r M38
		9	ptc.l M45	probe.w – imm_2 M39	mov to psr.um M35	probe.w M38
		A	ptc.g M45	thash M46	mov.m to ar M29	
		B	ptc.ga M45	ttag M46		
		C	ptr.d M45		mov to cr M32	
		D	ptr.i M45		mov to psr.l M35	
		E	itr.d M42	tpa M46	itc.d M41	
		F	itr.i M42	tak M46	itc.i M41	

4.4.10.1 Probe – Register

	40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
M38	1		x_3	x_6	r_3	r_2	r_1	qp
	4	1	3	6	7	7	7	6

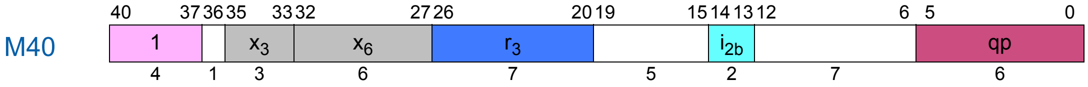
Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r	$r_1 = r_3, r_2$	1	0	38
probe.w				39

4.4.10.2 Probe – Immediate$_2$



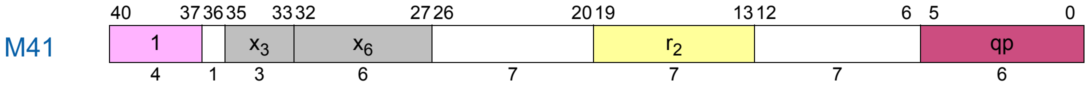
Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r	$r_1 = r_3, imm_2$	1	0	18
probe.w				19

4.4.10.3 Probe Fault – Immediate$_2$



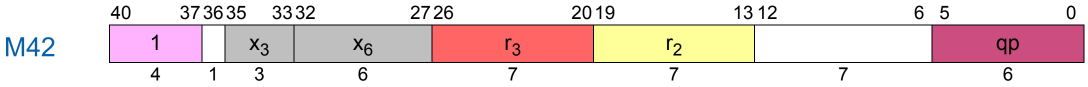
Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.rw.fault	r_3, imm_2	1	0	31
probe.r.fault				32
probe.w.fault				33

4.4.10.4 Translation Cache Insert



Instruction	Operands	Opcode	Extension	
			x_3	x_6
itc.d $^{l\ p}$	r_2	1	0	2E
itc.i $^{l\ p}$				2F

4.4.10.5 Move to Indirect Register/Translation Register Insert



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	rr[r_3] = r_2	1	0	00
	dbr[r_3] = r_2			01
	ibr[r_3] = r_2			02
	pkr[r_3] = r_2			03
	pmc[r_3] = r_2			04
	pmd[r_3] = r_2			05
itr.d ^{p}	dtr[r_3] = r_2			0E
itr.i ^{p}	itr[r_3] = r_2			0F

4.4.10.6 Move from Indirect Register

M43

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
1		x_3	x_6	r_3		r_1	qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov ^{p}	r_1 = rr[r_3]	1	0	10
	r_1 = dbr[r_3]			11
	r_1 = ibr[r_3]			12
	r_1 = pkr[r_3]			13
	r_1 = pmc[r_3]			14
mov	r_1 = pmd[r_3]			15
	r_1 = cpuid[r_3]			17

4.4.10.7 Set/Reset User/System Mask

M44

40 37	36	35 33	32 31	30 27	26 6	5 0
0	i	x_3	i_{2d}	x_4	imm_{21a}	qp
4	1	3	2	4	21	6

Instruction	Operands	Opcode	Extension	
			x_3	x_4
sum	*imm*$_{24}$	0	0	4
rum				5
ssm ^{p}				6
rsm ^{p}				7

4.4.10.8 Translation Purge

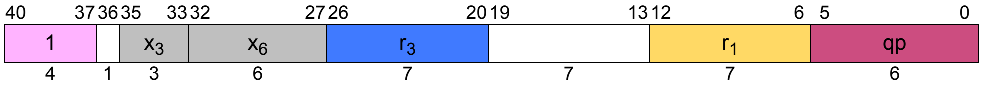
M45

40 37	36	35 33	32 27	26 20	19 13	12 6	5 0
1		x_3	x_6	r_3	r_2		qp
4	1	3	6	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.l ^{p}	r_3, r_2	1	0	09
ptc.g $^{l\ p}$				0A
ptc.ga $^{l\ p}$				0B
ptr.d ^{p}				0C
ptr.i ^{p}				0D

4.4.10.9 Translation Access

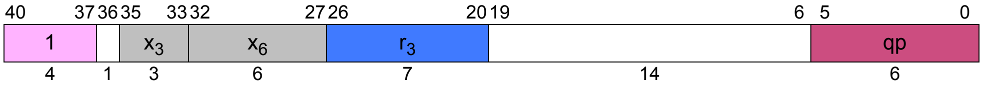
M46



Instruction	Operands	Opcode	Extension	
			x_3	x_6
thash ttag tpa ^{p} tak ^{p}	r_1 = r_3	1	0	1A 1B 1E 1F

4.4.10.10 Purge Translation Cache Entry

M47



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.e ^{p}	r_3	1	0	34

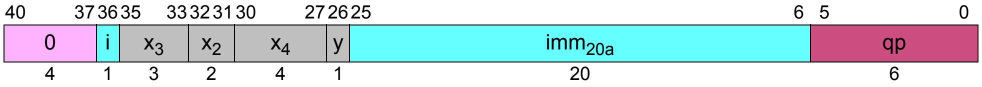
4.4.11 Nop/Hint (M-Unit)

M-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 2-bit opcode extension field in bits 32:31 (x_2), a 4-bit opcode extension field in bits 30:27 (x_4), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

Table 4-46. Misc M-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31	y Bit 26	
0	0	1	0	0	nop.m
				1	hint.m

M48



Instruction	Operands	Opcode	Extension			
			x_3	x_4	x_2	y
nop.m hint.m	imm_{21}	0	0	1	0	0 1

4.5 B-Unit Instruction Encodings

The branch-unit includes branch, predict, and miscellaneous instructions.

4.5.1 Branches

Opcode 0 is used for indirect branch, opcode 1 for indirect call, opcode 4 for IP-relative branch, and opcode 5 for IP-relative call.

The IP-relative branch instructions encoded within major opcode 4 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-47.

Table 4-47. IP-Relative Branch Types

Opcode Bits 40:37	btype Bits 8:6	
4	0	br.cond B1
	1	e
	2	br.wexit B1
	3	br.wtop B1
	4	e
	5	br.cloop B2
	6	br.cexit B2
	7	br.ctop B2

The indirect branch, indirect return, and miscellaneous branch-unit instructions are encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-48 summarizes these assignments.

Table 4-48. Indirect/Miscellaneous Branch Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
0	0	break.b B9	epc B8	Indirect Branch (Table 4-49)	e
	1		e	Indirect Return (Table 4-50)	e
	2	cover B8	e	e	e
	3	e	e	e	e
	4	clrrrb B8	e	e	e
	5	clrrrb.pr B8	e	e	e
	6	e	e	e	e
	7	e	e	e	e
	8	rfi B8	vmsw.0 B8	e	e
	9		vmsw.1 B8	e	e
	A	e	e	e	e
	B	e	e	e	e
	C	bsw.0 B8	e	e	e
	D	bsw.1 B8	e	e	e
	E	e	e	e	e
	F	e	e	e	e

The indirect branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-49.

Table 4-49. Indirect Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	20	0	br.cond B4
		1	br.ia B4
		2	e
		3	e
		4	e
		5	e
		6	e
		7	e

The indirect return branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-50.

Table 4-50. Indirect Return Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	21	0	e
		1	e
		2	e
		3	e
		4	br.ret B4
		5	e
		6	e
		7	e

All of the branch instructions have a 1-bit sequential prefetch opcode hint extension field, p, in bit 12. Table 4-51 summarizes these assignments.

Table 4-51. Sequential Prefetch Hint Completer

p Bit 12	*ph*
0	.few
1	.many

The IP-relative and indirect branch instructions all have a 2-bit branch prediction “whether” opcode hint extension field in bits 34:33 (wh) as shown in Table 4-52. Indirect call instructions have a 3-bit “whether” opcode hint extension field in bits 34:32 (wh) as shown in Table 4-53.

Table 4-52. Branch Whether Hint Completer

wh Bits 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

Table 4-53. Indirect Call Whether Hint Completer

wh Bits 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

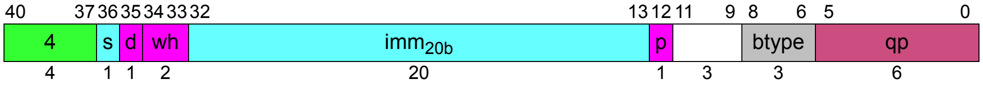
The branch instructions also have a 1-bit branch cache deallocation opcode hint extension field in bit 35 (d) as shown in Table 4-54.

Table 4-54. Branch Cache Deallocation Hint Completer

d Bit 35	*dh*
0	*none*
1	.clr

4.5.1.1 IP-Relative Branch

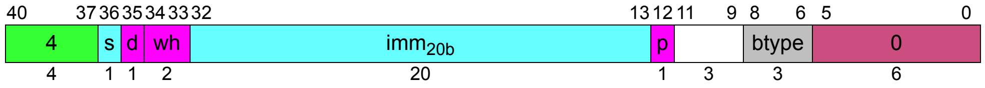
B1



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.*bwh.ph.dh* [e] br.wexit.*bwh.ph.dh* [e t] br.wtop.*bwh.ph.dh* [e t]	*target*$_{25}$	4	0 2 3	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.2 IP-Relative Counted Branch

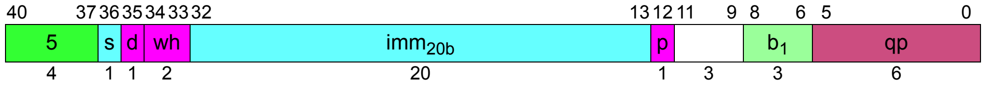
B2



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.*bwh.ph.dh* $^{e\ t}$ br.cexit.*bwh.ph.dh* $^{e\ t}$ br.ctop.*bwh.ph.dh* $^{e\ t}$	*target$_{25}$*	4	5 6 7	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.3 IP-Relative Call

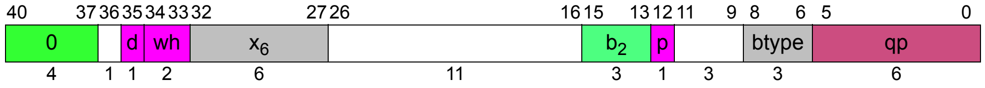
B3



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* ^{e}	b_1 = *target$_{25}$*	5	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.5.1.4 Indirect Branch

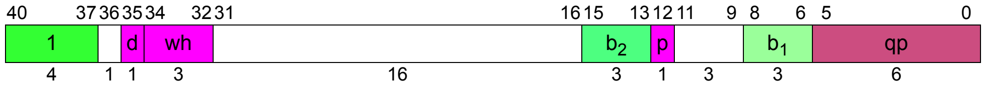
B4



Instruction	Operands	Opcode	Extension				
			x$_6$	btype	p	wh	d
br.cond.*bwh.ph.dh* ^{e}	b_2	0	20	0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352
br.ia.*bwh.ph.dh* ^{e}				1			
br.ret.*bwh.ph.dh* ^{e}			21	4			

4.5.1.5 Indirect Call

B5



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* ^{e}	b_1 = b_2	1	See Table 4-51 on page 3:351	See Table 4-53 on page 3:352	See Table 4-54 on page 3:352

4.5.2 Branch Predict/Nop/Hint

The branch predict, nop, and hint instructions are encoded in major opcodes 2 (Indirect Predict/Nop/Hint) and 7 (IP-relative Predict). The indirect predict, nop, and hint instructions in major opcode 2 use a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-55 summarizes these assignments.

Table 4-55. Indirect Predict/Nop/Hint Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9	brp B7		
	1	hint.b B9	brp.ret B7		
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

The branch predict instructions all have a 1-bit branch importance opcode hint extension field in bit 35 (ih). The mov to BR instruction (page 3:320) also has this hint in bit 23. Table 4-56 shows these assignments.

Table 4-56. Branch Importance Hint Completer

ih Bit 23 or Bit 35	*ih*
0	*none*
1	.imp

The IP-relative branch predict instructions have a 2-bit branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in Table 4-57. Note that the combination of the .loop or .exit whether hint completer with the *none* importance hint completer is undefined.

Table 4-57. IP-Relative Predict Whether Hint Completer

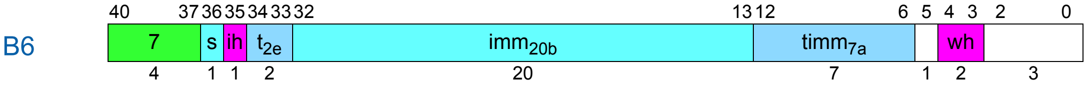
wh Bits 4:3	*ipwh*
0	.sptk
1	.loop
2	.dptk
3	.exit

The indirect branch predict instructions have a 2-bit branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in Table 4-58.

Table 4-58. Indirect Predict Whether Hint Completer

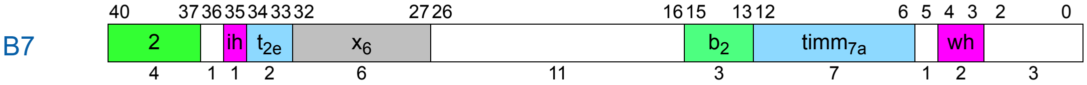
wh Bits 4:3	*indwh*
0	.sptk
1	
2	.dptk
3	

4.5.2.1 IP-Relative Predict



Instruction	Operands	Opcode	Extension	
			ih	wh
brp.*ipwh.ih*	*target*$_{25}$, *tag*$_{13}$	7	See Table 4-56 on page 3:354	See Table 4-57 on page 3:354

4.5.2.2 Indirect Predict

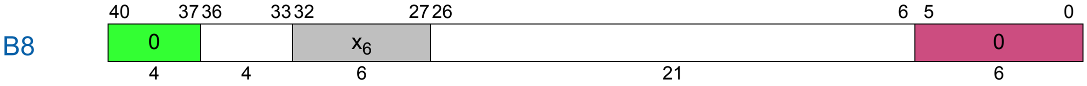


Instruction	Operands	Opcode	Extension		
			x_6	ih	wh
brp.*indwh.ih*	b_2, *tag*$_{13}$	2	10	See Table 4-56 on page 3:354	See Table 4-58 on page 3:355
brp.ret.*indwh.ih*			11		

4.5.3 Miscellaneous B-Unit Instructions

The miscellaneous branch-unit instructions include a number of instructions encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6) as described in Table 4-48 on page 3:350.

4.5.3.1 Miscellaneous (B-Unit)

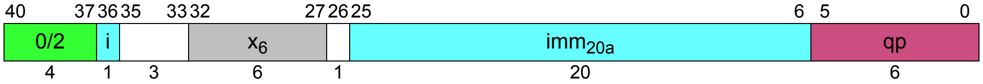


Instruction	Opcode	Extension
		x_6
cover ^{l}	0	02
clrrrb ^{l}		04
clrrrb.pr ^{l}		05
rfi $^{e\ l\ p}$		08
bsw.0 $^{l\ p}$		0C
bsw.1 $^{l\ p}$		0D
epc		10

Instruction	Opcode	Extension
		x_6
vmsw.0 ^{p}	0	18
vmsw.1 ^{p}		19

4.5.3.2 Break/Nop/Hint (B-Unit)

B9



Instruction	Operands	Opcode	Extension
			x_6
break.b ^{e}	imm_{21}	0	00
nop.b		2	
hint.b			01

4.6 F-Unit Instruction Encodings

The floating-point instructions are encoded in major opcodes 8 – E for floating-point and fixed-point arithmetic, opcode 4 for floating-point compare, opcode 5 for floating-point class, and opcodes 0 and 1 for miscellaneous floating-point instructions.

The miscellaneous and reciprocal approximation floating-point instructions are encoded within major opcodes 0 and 1 using a 1-bit opcode extension field (x) in bit 33 and either a second 1-bit extension field in bit 36 (q) or a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-59 shows the 1-bit x assignments, Table 4-62 shows the additional 1-bit q assignments for the reciprocal approximation instructions; Table 4-60 and Table 4-61 summarize the 6-bit x_6 assignments.

Table 4-59. Miscellaneous Floating-point 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext (Table 4-60)
	1	Reciprocal Approximation (Table 4-62)
1	0	6-bit Ext (Table 4-61)
	1	Reciprocal Approximation (Table 4-62)

Table 4-60. Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	1-bit Ext (Table 4-68)	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

Table 4-61. Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

Table 4-62. Reciprocal Approximation 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	q Bit 36	
0	1	0	frcpa F6
		1	frsqrta F7
1		0	fprcpa F6
		1	fprsqrta F7

Most floating-point instructions have a 2-bit opcode extension field in bits 35:34 (sf) which encodes the FPSR status field to be used. Table 4-63 summarizes these assignments.

Table 4-63. Floating-point Status Field Completer

sf Bits 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

4.6.1 Arithmetic

The floating-point arithmetic instructions are encoded within major opcodes 8 – D using a 1-bit opcode extension field (x) in bit 36 and a 2-bit opcode extension field (sf) in bits 35:34. The opcode and x assignments are shown in Table 4-64.

Table 4-64. Floating-point Arithmetic 1-bit Opcode Extensions

x Bit 36	Opcode Bits 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

The fixed-point arithmetic and parallel floating-point select instructions are encoded within major opcode E using a 1-bit opcode extension field (x) in bit 36. The fixed-point arithmetic instructions also have a 2-bit opcode extension field (x_2) in bits 35:34. These assignments are shown in Table 4-65.

Table 4-65. Fixed-point Multiply Add and Select Opcode Extensions

Opcode Bits 40:37	x Bit 36	x_2 Bits 35:34			
		0	1	2	3
E	0	fselect F3			
	1	xma.l F2		xma.hu F2	xma.h F2

4.6.1.1 Floating-point Multiply Add

F1

40 37	36	35 34	33 27	26 20	19 13	12 6	5 0
8 - D	x	sf	f_4	f_3	f_2	f_1	qp
4	1	2	7	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x	sf
fma.*sf*	$f_1 = f_3, f_4, f_2$	8	0	See Table 4-63 on page 3:358
fma.s.*sf*			1	
fma.d.*sf*		9	0	
fpma.*sf*			1	
fms.*sf*		A	0	
fms.s.*sf*			1	
fms.d.*sf*		B	0	
fpms.*sf*			1	
fnma.*sf*		C	0	
fnma.s.*sf*			1	
fnma.d.*sf*		D	0	
fpnma.*sf*			1	

4.6.1.2 Fixed-point Multiply Add

F2

40 37	36	35 34	33 27	26 20	19 13	12 6	5 0
E	x	x_2	f_4	f_3	f_2	f_1	qp
4	1	2	7	7	7	7	6

Instruction	Operands	Opcode	Extension	
			x	x_2
xma.l	$f_1 = f_3, f_4, f_2$	E	1	0
xma.h				3
xma.hu				2

4.6.2 Parallel Floating-point Select

F3

40 37	36	35 34	33 27	26 20	19 13	12 6	5 0
E	x		f_4	f_3	f_2	f_1	qp
4	1	2	7	7	7	7	6

Instruction	Operands	Opcode	Extension
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

4.6.3 Compare and Classify

The predicate setting floating-point compare instructions are encoded within major opcode 4 using three 1-bit opcode extension fields in bits 33 (r_a), 36 (r_b), and 12 (t_a), and a 2-bit opcode extension field (sf) in bits 35:34. The opcode, r_a, r_b, and t_a assignments are shown in Table 4-66. The sf assignments are shown in Table 4-63 on page 3:358.

The parallel floating-point compare instructions are described on page 3:362.

Table 4-66. Floating-point Compare Opcode Extensions

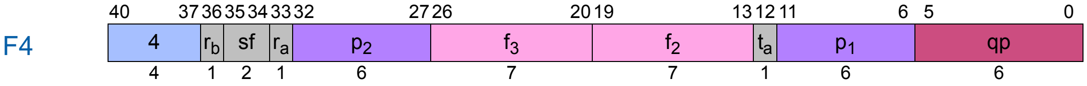
Opcode Bits 40:37	r_a Bit 33	r_b Bit 36	t_a Bit 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

The floating-point class instructions are encoded within major opcode 5 using a 1-bit opcode extension field in bit 12 (t_a) as shown in Table 4-67.

Table 4-67. Floating-point Class 1-bit Opcode Extensions

Opcode Bits 40:37	t_a Bit 12	
5	0	fclass.m F5
	1	fclass.m.unc F5

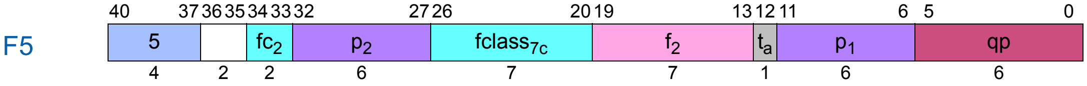
4.6.3.1 Floating-point Compare



	40 37	36	35 34	33	32 27	26 20	19 13	12	11 6	5 0
F4	4	r_b	sf	r_a	p_2	f_3	f_2	t_a	p_1	qp
	4	1	2	1	6	7	7	1	6	6

Instruction	Operands	Opcode	Extension			
			r_a	r_b	t_a	sf
fcmp.eq.*sf*	$p_1, p_2 = f_2, f_3$	4	0	0	0	See Table 4-63 on page 3:358
fcmp.lt.*sf*				1		
fcmp.le.*sf*			1	0		
fcmp.unord.*sf*				1		
fcmp.eq.unc.*sf*			0	0	1	
fcmp.lt.unc.*sf*				1		
fcmp.le.unc.*sf*			1	0		
fcmp.unord.unc.*sf*				1		

4.6.3.2 Floating-point Class



	40 37	36 35	34 33	32 27	26 20	19 13	12	11 6	5 0
F5	5		fc_2	p_2	$fclass_{7c}$	f_2	t_a	p_1	qp
	4	2	2	6	7	7	1	6	6

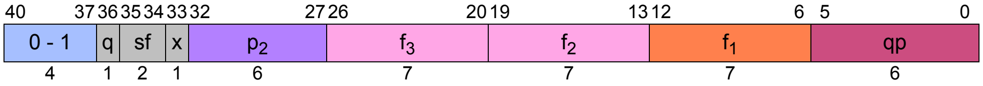
Instruction	Operands	Opcode	Extension
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

4.6.4 Approximation

4.6.4.1 Floating-point Reciprocal Approximation

There are two Reciprocal Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

F6

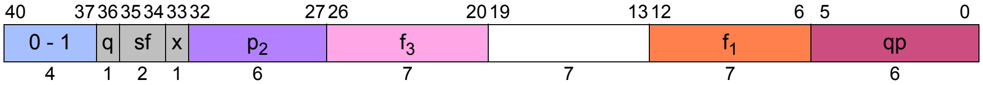


Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.*sf*	$f_1, p_2 = f_2, f_3$	0	1	0	See Table 4-63 on page 3:358
fprcpa.*sf*		1			

4.6.4.2 Floating-point Reciprocal Square Root Approximation

There are two Reciprocal Square Root Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

F7



Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.*sf*	$f_1, p_2 = f_3$	0	1	1	See Table 4-63 on page 3:358
fprsqrta.*sf*		1			

4.6.5 Minimum/Maximum and Parallel Compare

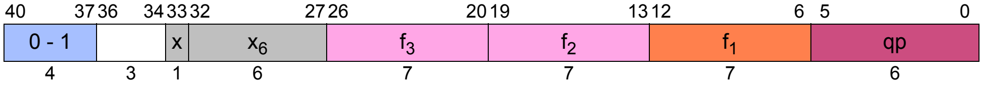
There are two groups of Minimum/Maximum instructions. The first group, in major op 0, encodes the full register variants. The second group, in major op 1, encodes the parallel variants. The parallel compare instructions are all encoded in major op 1.

	40 37	36	35 34	33	32 27	26 20	19 13	12 6	5 0
F8	0 - 1		sf	x	x_6	f_3	f_2	f_1	qp
	4	1	2	1	6	7	7	7	6

Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fmin.*sf*	$f_1 = f_2, f_3$	0	0	14	See Table 4-63 on page 3:358
fmax.*sf*				15	
famin.*sf*				16	
famax.*sf*				17	
fpmin.*sf*		1		14	
fpmax.*sf*				15	
fpamin.*sf*				16	
fpamax.*sf*				17	
fpcmp.eq.*sf*				30	
fpcmp.lt.*sf*				31	
fpcmp.le.*sf*				32	
fpcmp.unord.*sf*				33	
fpcmp.neq.*sf*				34	
fpcmp.nlt.*sf*				35	
fpcmp.nle.*sf*				36	
fpcmp.ord.*sf*				37	

4.6.6 Merge and Logical

F9

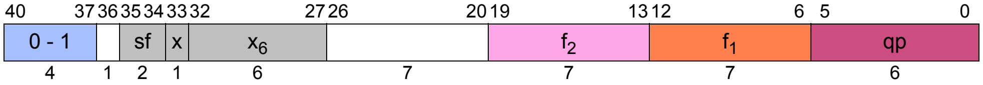


Instruction	Operands	Opcode	Extension	
			x	x_6
fmerge.s	$f_1 = f_2, f_3$	0	0	10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack				28
fswap				34
fswap.nl				35
fswap.nr				36
fand				2C
fandcm				2D
for				2E
fxor				2F
fpmerge.s		1		10
fpmerge.ns				11
fpmerge.se				12

4.6.7 Conversion

4.6.7.1 Convert Floating-point to Fixed-point

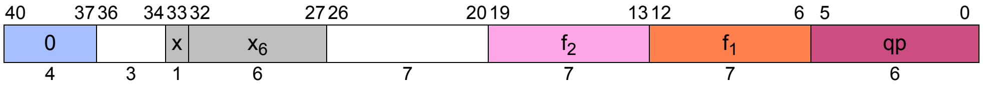
F10



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	See Table 4-63 on page 3:358
fcvt.fxu.*sf*				19	
fcvt.fx.trunc.*sf*				1A	
fcvt.fxu.trunc.*sf*				1B	
fpcvt.fx.*sf*		1		18	
fpcvt.fxu.*sf*				19	
fpcvt.fx.trunc.*sf*				1A	
fpcvt.fxu.trunc.*sf*				1B	

4.6.7.2 Convert Fixed-point to Floating-point

F11

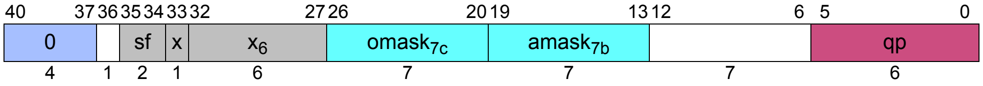


Instruction	Operands	Opcode	Extension	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

4.6.8 Status Field Manipulation

4.6.8.1 Floating-point Set Controls

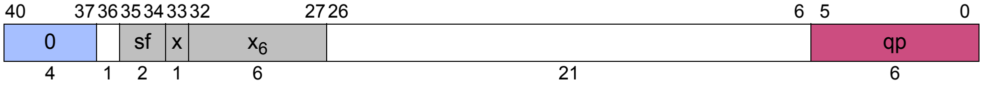
F12



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fsetc.*sf*	*amask*$_7$, *omask*$_7$	0	0	04	See Table 4-63 on page 3:358

4.6.8.2 Floating-point Clear Flags

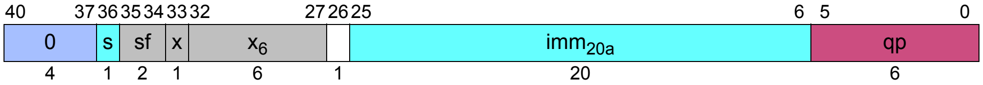
F13



Instruction	Opcode	Extension		
		x	x_6	sf
fclrf.*sf*	0	0	05	See Table 4-63 on page 3:358

4.6.8.3 Floating-point Check Flags

F14

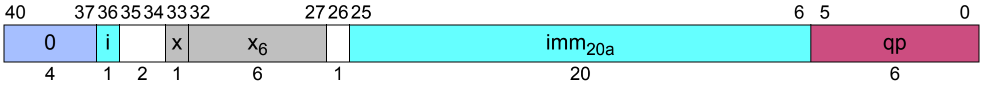


Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fchkf.*sf*	*target*$_{25}$	0	0	08	See Table 4-63 on page 3:358

4.6.9 Miscellaneous F-Unit Instructions

4.6.9.1 Break (F-Unit)

F15



Instruction	Operands	Opcode	Extension	
			x	x_6
break.f	*imm_{21}*	0	0	00

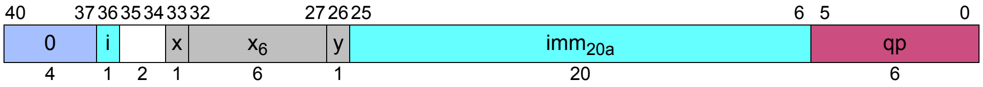
4.6.9.2 Nop/Hint (F-Unit)

F-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-46.

Table 4-68. Misc F-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit :33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.f
			1	hint.f

F16



Instruction	Operands	Opcode	Extension		
			x	x_6	y
nop.f hint.f	*imm_{21}*	0	0	01	0 1

4.7 X-Unit Instruction Encodings

The X-unit instructions occupy two instruction slots, L+X. The major opcode, opcode extensions and hints, qp, and small immediate fields occupy the X instruction slot. For movl, break.x, and nop.x, the imm_{41} field occupies the L instruction slot. For brl, the imm_{39} field and a 2-bit Ignored field occupy the L instruction slot.

4.7.1 Miscellaneous X-Unit Instructions

The miscellaneous X-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33 and a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-69 shows the 3-bit assignments and Table 4-70 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

Table 4-69. Misc X-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-70)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

Table 4-70. Misc X-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
0	0	0	break.x X1			
		1	1-bit Ext (Table 4-73)			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

4.7.1.1 Break (X-Unit)

	40 37	36	35 33	32 27	26	25 6	5 0	40 0
X1	0	i	x_3	x_6		imm_{20a}	qp	imm_{41}
	4	1	3	6	1	20	6	41

Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.x	imm_{62}	0	0	00

4.7.2 Move Long Immediate$_{64}$

The move long immediate instruction is encoded within major opcode 6 using a 1-bit reserved opcode extension in bit 20 (v_c) as shown in Table 4-71. This instruction is executed by an I-unit.

Table 4-71. Move Long 1-bit Opcode Extensions

Opcode Bits 40:37	v_c Bit 20	
6	0	movl X2
	1	

	40 37	36	35 27	26 22	21	20	19 13	12 6	5 0	40 0
X2	6	i	imm_{9d}	imm_{5c}	i_c	v_c	imm_{7b}	r_1	qp	imm_{41}
	4	1	9	5	1	1	7	7	6	41

Instruction	Operands	Opcode	Extension
			v_c
movl ^{i}	$r_1 = imm_{64}$	6	0

4.7.3 Long Branches

Long branches are executed by a B-unit. Opcode C is used for long branch and opcode D for long call.

The long branch instructions encoded within major opcode C use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-72.

Table 4-72. Long Branch Types

Opcode Bits 40:37	btype Bits 8:6	
C	0	brl.cond X3
	1	
	2	
	3	
	4	
	5	
	6	
	7	

The long branch instructions have the same opcode hint fields in bit 12 (p), bits 34:33 (wh), and bit 35 (d) as normal IP-relative branches. These are shown in Table 4-51 on page 3:351, Table 4-52 on page 3:352, and Table 4-54 on page 3:352.

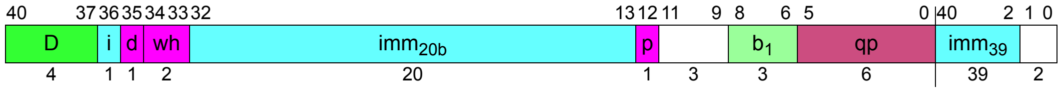
4.7.3.1 Long Branch

	40 37	36	35	34 33	32 13	12	11 9	8 6	5 0	40 2	1 0
X3	C	i	d	wh	imm_{20b}	p		btype	qp	imm_{39}	
	4	1	1	2	20	1	3	3	6	39	2

Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
brl.cond.*bwh.ph.dh* $^{e\ l}$	$target_{64}$	C	0	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

4.7.3.2 Long Call

X4



Instruction	Operands	Opcode	Extension		
			p	wh	d
brl.call.*bwh.ph.dh* [e] [l]	*b_1 = $target_{64}$*	D	See Table 4-51 on page 3:351	See Table 4-52 on page 3:352	See Table 4-54 on page 3:352

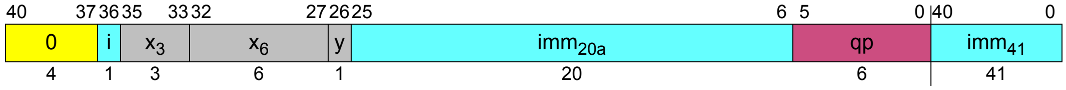
4.7.4 Nop/Hint (X-Unit)

X-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-73. These instructions are executed by an I-unit.

Table 4-73. Misc X-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.x
			1	hint.x

X5



Instruction	Operands	Opcode	Extension		
			x_3	x_6	y
nop.x	*imm_{62}*	0	0	01	0
hint.x					1

4.8 Immediate Formation

Table 4-74 shows, for each instruction format that has one or more immediates, how those immediates are formed. In each equation, the symbol to the left of the equals is the assembly language name for the immediate. The symbols to the right are the field names in the instruction encoding.

Table 4-74. Immediate Formation

Instruction Format	Immediate Formation
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	imm_8 = sign_ext(s << 7 | imm_{7b}, 8)
A4	imm_{14} = sign_ext(s << 13 | imm_{6d} << 7 | imm_{7b}, 14)
A5	imm_{22} = sign_ext(s << 21 | imm_{5c} << 16 | imm_{9d} << 7 | imm_{7b}, 22)
A10	$count_2$ = (ct_{2d} > 2) ? reservedQP[a] : ct_{2d} + 1
I1	$count_2$ = (ct_{2d} == 0) ? 0 : (ct_{2d} == 1) ? 7 : (ct_{2d} == 2) ? 15 : 16

Table 4-74. Immediate Formation (Continued)

Instruction Format	Immediate Formation
I3	$mbtype_4$ = (mbt_{4c} == 0) ? @brcst : (mbt_{4c} == 8) ? @mix : (mbt_{4c} == 9) ? @shuf : (mbt_{4c} == 0xA) ? @alt : (mbt_{4c} == 0xB) ? @rev : reservedQP[a]
I4	$mhtype_8$ = mht_{8c}
I6	$count_5$ = $count_{5b}$
I8	$count_5$ = 31 – $ccount_{5c}$
I10	$count_6$ = $count_{6d}$
I11	len_6 = len_{6d} + 1 pos_6 = pos_{6b}
I12	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6c}$
I13	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6c}$ imm_8 = sign_ext(s << 7 | imm_{7b}, 8)
I14	len_6 = len_{6d} + 1 pos_6 = 63 – $cpos_{6b}$ imm_1 = sign_ext(s, 1)
I15	len_4 = len_{4d} + 1 pos_6 = 63 – $cpos_{6d}$
I16	pos_6 = pos_{6b}
I18 I19 M37 M48	imm_{21} = i << 20 | imm_{20a}
I21	tag_{13} = IP + (sign_ext($timm_{9c}$, 9) << 4)
I23	$mask_{17}$ = sign_ext(s << 16 | $mask_{8c}$ << 8 | $mask_{7a}$ << 1, 17)
I24	imm_{44} = sign_ext(s << 43 | imm_{27a} << 16, 44)
I30	imm_5 = imm_{5b} + 32
M3 M8 M22	imm_9 = sign_ext(s << 8 | i << 7 | imm_{7b}, 9)
M5 M10	imm_9 = sign_ext(s << 8 | i << 7 | imm_{7a}, 9)
M17	inc_3 = sign_ext(((s) ? –1 : 1) * ((i_{2b} == 3) ? 1 : 1 << (4 – i_{2b})), 6)
I20 M20 M21	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{13c} << 7 | imm_{7a}, 21) << 4)
M22 M23	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)

M34	il = sol
	o = sof – sol
	r = sor << 3

M39 M40	imm_2 = i_{2b}
M44	imm_{24} = i << 23 | i_{2d} << 21 | imm_{21a}
B1 B2 B3	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4)
B6	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20b}, 21) << 4) tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B7	tag_{13} = IP + (sign_ext(t_{2e} << 7 | $timm_{7a}$, 9) << 4)
B9	imm_{21} = i << 20 | imm_{20a}
F5	$fclass_9$ = $fclass_{7c}$ << 2 | fc_2
F12	$amask_7$ = $amask_{7b}$ $omask_7$ = $omask_{7c}$
F14	$target_{25}$ = IP + (sign_ext(s << 20 | imm_{20a}, 21) << 4)
F15 F16	imm_{21} = i << 20 | imm_{20a}
X1 X5	imm_{62} = imm_{41} << 21 | i << 20 | imm_{20a}
X2	imm_{64} = i << 63 | imm_{41} << 22 | i_c << 21 | imm_{5c} << 16 | imm_{9d} << 7 | imm_{7b}
X3 X4	$target_{64}$ = IP + ((i << 59 | imm_{39} << 20 | imm_{20b}) << 4)

[a] This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.

§

Resource and Dependency Semantics 5

5.1 Reading and Writing Resources

An Itanium instruction is said to be a **reader** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or is one of the instructions that reads a resource even when its qualifying predicate is 0, and the execution of the instruction depends on that resource.

An Itanium instruction is said to be an **writer** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or writes the resource even when the qualifying predicate is 0, and the execution of the instruction writes that resource.

An Itanium instruction is said to be a reader or writer of a resource even if it only sometimes depends on that resource and it cannot be determined statically whether the resource will be read or written. For example, `cover` only writes CR[IFS] when PSR.ic is 0, but for purposes of dependency, it is treated as if it always writes the resource since this condition cannot be determined statically. On the other hand, `rsm` conditionally writes several bits in the PSR depending on a mask which is encoded as an immediate in the instruction. Since the PSR bits to be written can be determined by examining the encoded instruction, the instruction is treated as only writing those bits which have a corresponding mask bit set. All exceptions to these general rules are described in this appendix.

5.2 Dependencies and Serialization

A **RAW** (Read-After-Write) dependency is a sequence of two events where the first is a writer of a resource and the second is a reader of the same resource. Events may be instructions, interruptions, or other 'uses' of the resource such as instruction stream fetches and VHPT walks. Table 5-2 covers only dependencies based on instruction readers and writers.

A **WAW** (Write-After-Write) dependency is a sequence of two events where both events write the resource in question. Events may be instructions, interruptions, or other 'updates' of the resource. Table 5-3 covers only dependencies based on instruction writers.

A **WAR** (Write-After-Read) dependency is a sequence of two instructions, where the first is a reader of a resource and the second is a writer of the same resource. Such dependencies are always allowed except as indicated in Table 5-4 and only those related to instruction readers and writers are included.

A **RAR** (Read-After-Read) dependency is a sequence of two instructions where both are readers of the same resource. Such dependencies are always allowed.

RAW and WAW dependencies are generally not allowed without some type of serialization event (an implied, data, or instruction serialization after the first writing instruction. (See Section 3.2, “Serialization” on page 2:17 for details on serialization.) The tables and associated rules in this appendix provide a comprehensive list of readers and writers of resources and describe the serialization required for the dependency to be observed and possible outcomes if the required serialization is not met. Even when targeting code for machines which do not check for particular disallowed dependencies, such code sequences are considered architecturally undefined and may cause code to behave differently across processors, operating systems, or even separate executions of the code sequence during the same program run. In some cases, different serializations may yield different, but well-defined results.

The serialization of application level (non-privileged) resources is always implied. This means that if a writer of that resource and a subsequent read of that same resource are in different instruction groups, then the reader will see the value written. In addition, for dependencies on PRs and BRs, where the writer is a non-branch instruction and the reader is a branch instruction, the writer and reader may be in the same instruction group.

System resources generally require explicit serialization, i.e., the use of a `srlz.i` or `srlz.d` instruction, between the writing and the reading of that resource. Note that RAW accesses to CRs are not exceptional – they require explicit data or instruction serialization. However, in some cases (other than CRs) where pairs of instructions explicitly encode the same resource, serialization is implied.

There are cases where it is architecturally allowed to omit a serialization, and that the response from the CPU must be atomic (act as if either the old or the new state were fully in place). The tables in this appendix indicate dependency requirements under the assumption that the desired result is for the dependency to always be observed. In some such cases, the programmer may not care if the old or new state is used; such situations are allowed, but the value seen is not deterministic.

On the other hand, if an *impliedF* dependency is violated, then the program is incorrectly coded and the processor's behavior is undefined.

5.3 Resource and Dependency Table Format Notes

- The “Writers” and “Readers” columns of the dependency tables contain instruction class names and instruction mnemonic prefixes as given in the format section of each instruction page. To avoid ambiguity, instruction classes are shown in bold, while instruction mnemonic prefixes are in regular font. For instruction mnemonic prefixes, all instructions that exactly match the name specified or those that begin with the specified text and are followed by a ‘.’ and then followed by any other text will match.
- The dependency on a listed instruction is in effect no matter what values are encoded in the instruction or what dynamic values occur in operands, unless a superscript is present or one of the special case instruction rules in Section 5.3.1 applies. Instructions listed are still subject to rules regarding qualifying predicates.
- Instruction classes are groups of related instructions. Such names appear in boldface for clarity. The list of all instruction classes is contained in Table 5-5. Note that an instruction may appear in multiple instruction classes, instruction classes

may expand to contain other classes, and that when fully expanded, a set of classes (e.g., the readers of some resource) may contain the same instruction multiple times.

- The syntax '**x**\y' where **x** and **y** are both instruction classes, indicates an unnamed instruction class that includes all instructions in instruction class **x** but that are not in instruction class **y**. Similarly, the notation '**x\y\z**' means all instructions in instruction class **x**, but that are not in either instruction class **y** or instruction class **z**.
- Resources on separate rows of a table are independent resources. This means that there are no serialization requirements for an event which references one of them followed by an event which uses a different resource. In cases where resources are broken into subrows, dependencies only apply between instructions within a subrow. Instructions that do not appear in a subrow together have no dependencies (reader/writer or writer/writer dependencies) for the resource in question, although they may still have dependencies on some other resource.
- The dependencies listed for pairs of instructions on each resource are not unique – the same pair of instructions might also have a dependency on some other resource with a different semantics of dependency. In cases where there are multiple resource dependencies for the same pair of instructions, the most stringent semantics are assumed: *instr* overrides *data* which overrides *impliedF* which overrides *implied* which overrides *none*.
- Arrays of numbered resources are represented in a single row of a table using the % notation as a substitute for the number of the resource. In such cases, the semantics of the table are as if each numbered resource had its own row in that table and is thus an independent resource. The range of values that the % can take are given in the "Resource Name" column.
- An asterisk '*' in the "Resource Name" column indicates that this resource may not have a physical resource associated with it, but is added to enforce special dependencies.
- A pound sign '#' in the "Resource Name" column indicates that this resource is an array of resources that are indexed by a value in a GR. The number of individual elements in the array is described in the detailed description of each resource.
- The "Semantics of Dependency" column describes the outcome given various serialization and instruction group boundary conditions. The exact definition for each keyword is given in Table 5-1.

Table 5-1. Semantics of Dependency Codes

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
instr	Instruction Serialization (See "Instruction Serialization" on page 2:18).	Atomic: Any attempt to read a resource after one or more insufficiently serialized writes is either the value previously in the register (before any of the unserialized writes) or the value of one of any unserialized writes. Which value is returned is unpredictable and multiple insufficiently serialized reads may see different results. No fault will be caused by the insufficient serialization.
data	Data Serialization (See "Data Serialization" on page 2:18)	
implied	Instruction Group Break. Writer and reader must be in separate instruction groups. (See "Instruction Sequencing Considerations" on page 1:39).	

Table 5-1. Semantics of Dependency Codes (Continued)

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
impliedF	Instruction Group Break (same as above).	An undefined value is returned, or an Illegal Operation fault may be taken. If no fault is taken, the value returned is unpredictable, and may be unrelated to past writes, but will not be data which could not be accessed by the current process (e.g., if PSR.cpl != 0, the undefined value to return cannot be read from some control register).
stop	Stop. Writer and reader must be separated by a stop.	
none	None	N/A
specific	Implementation Specific	
SC	Special Case	Described elsewhere in book, see referenced section in the entry.

5.3.1 Special Case Instruction Rules

The following rules apply to the specified instructions when they appear in Table 5-2, Table 5-3, Table 5-4, or Table 5-5:

- An instruction always reads a given resource if its qualifying predicate is 1 and it appears in the "Reader" column of the table (except as noted). An instruction always writes a given resource if its qualifying predicate is 1 and it appears in the "Writer" column of the table (except as noted). An instruction never reads or writes the specified resource if its qualifying predicate is 0 (except as noted). These rules include branches and their qualifying predicate. Instructions in the **unpredicatable-instructions** class have no qualifying predicate and thus always read or write their resources (except as noted).
- An instruction of type **mov-from-PR** reads all PRs if its PR[qp] is true. If the PR[qp] is false, then only the PR[qp] is read.
- An instruction of type **mov-to-PR** writes only those PRs as indicated by the immediate mask encoded in the instruction.
- A `st8.spill` only writes AR[UNAT]{*X*} where *X* equals the value in bits 8:3 of the store's data address. A `ld8.fill` instruction only reads AR[UNAT]{*Y*} where *Y* equals the value in bits 8:3 of the load's data address.
- Instructions of type **mod-sched-brs** always read AR[EC] and the rotating register base registers in CFM, and always write AR[EC], the rotating register bases in CFM, and PR[63] even if they do not change their values or if their PR[qp] is false.
- Instructions of type **mod-sched-brs-counted** always read and write AR[LC], even if they do not change its value.
- For instructions of type **pr-or-writers** or **pr-and-writers**, if their completer is `or.andcm`, then only the first target predicate is an or-compare and the second target predicate is an and-compare. Similarly, if their completer is `and.orcm`, then only the second target predicate is an or-compare and the first target predicate is an and-compare.
- `rum` and `sum` only read PSR.sp when the bit corresponding to PSR.up (bit 2) is set in the immediate field of the instruction.

5.3.2 RAW Dependency Table

Table 5-2 architecturally defines the following information:

- A list of all architecturally-defined, independently-writable resources in the Itanium architecture. Each row represents an 'atomic' resource. Thus, for each row in the table, hardware will probably require a separate write-enable control signal.
- For each resource, a complete list of readers and writers.
- For each instruction, a complete list of all resources read and written. Such a list can be obtained by taking the union of all the rows in which each instruction appears.

Table 5-2. RAW Dependencies Organized by Resource

Resource Name	Writers	Readers	Semantics of Dependency
ALAT	chk.a.clr, **mem-readers-alat**, **mem-writers**, **invala-all**	**mem-readers-alat**, **mem-writers**, **chk-a**, invala.e	none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi	br.call, brl.call, br.ia, br.ret, cover, flushrs, loadrs, **mov-from-AR-BSP**, rfi	impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, **mov-from-AR-BSPSTORE**	impliedF
AR[CCV]	**mov-to-AR-CCV**	br.ia, **cmpxchg**, **mov-from-AR-CCV**	impliedF
AR[CFLG]	**mov-to-AR-CFLG**	br.ia, **mov-from-AR-CFLG**	impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**	br.ia, cmp8xchg16, **mov-from-AR-CSD**, st16	impliedF
AR[EC]	**mod-sched-brs**, br.ret, **mov-to-AR-EC**	br.call, brl.call, br.ia, **mod-sched-brs**, **mov-from-AR-EC**	impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**	br.ia, **mov-from-AR-EFLAG**	impliedF
AR[FCR]	**mov-to-AR-FCR**	br.ia, **mov-from-AR-FCR**	impliedF
AR[FDR]	**mov-to-AR-FDR**	br.ia, **mov-from-AR-FDR**	impliedF
AR[FIR]	**mov-to-AR-FIR**	br.ia, **mov-from-AR-FIR**	impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0	br.ia, **fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**, fsetc, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1	br.ia, **fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**, **mov-from-AR-FPSR**	
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2	br.ia, **fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**, **mov-from-AR-FPSR**	
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3	br.ia, **fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**, **mov-from-AR-FPSR**	
AR[FPSR].sf0.flags	**fp-arith-s0**, fclrf.s0, **fcmp-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	br.ia, fchkf, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, fclrf.s1, **fcmp-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	br.ia, fchkf.s1, **mov-from-AR-FPSR**	
AR[FPSR].sf2.flags	**fp-arith-s2**, fclrf.s2, **fcmp-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	br.ia, fchkf.s2, **mov-from-AR-FPSR**	
AR[FPSR].sf3.flags	**fp-arith-s3**, fclrf.s3, **fcmp-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	br.ia, fchkf.s3, **mov-from-AR-FPSR**	
AR[FPSR].traps	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FSR]	**mov-to-AR-FSR**	br.ia, **mov-from-AR-FSR**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
AR[ITC]	**mov-to-AR-ITC**	br.ia, **mov-from-AR-ITC**	impliedF
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]	br.ia, **mov-from-AR-K**[1]	impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**	br.ia, **mod-sched-brs-counted**, **mov-from-AR-LC**	impliedF
AR[PFS]	br.call, brl.call	alloc, br.ia, br.ret, epc, **mov-from-AR-PFS**	impliedF
	mov-to-AR-PFS	alloc, br.ia, epc, **mov-from-AR-PFS**	impliedF
		br.ret	none
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RNAT**	impliedF
AR[RSC]	**mov-to-AR-RSC**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RSC**, **mov-from-AR-BSPSTORE**, **mov-to-AR-RNAT**, **mov-from-AR-RNAT**, **mov-to-AR-BSPSTORE**	impliedF
AR[RUC]	**mov-to-AR-RUC**	br.ia, **mov-from-AR-RUC**	impliedF
AR[SSD]	**mov-to-AR-SSD**	br.ia, **mov-from-AR-SSD**	impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill	br.ia, ld8.fill, **mov-from-AR-UNAT**	impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	**none**	br.ia, **mov-from-AR-rv**[1]	none
AR%, % in 48-63, 112-127	**mov-to-AR-ig**[1]	br.ia, **mov-from-AR-ig**[1]	impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**indirect-brs**[1], **indirect-brp**[1], **mov-from-BR**[1]	impliedF
	mov-to-BR[1]	**indirect-brs**[1]	none
		indirect-brp[1], **mov-from-BR**[1]	impliedF
CFM	**mod-sched-brs**	**mod-sched-brs**	impliedF
		cover, alloc, rfi, loadrs, br.ret, br.call, brl.call	impliedF
		cfm-readers[2]	impliedF
	br.call, brl.call, br.ret, clrrrb, cover, rfi	**cfm-readers**	impliedF
	alloc	**cfm-readers**	none
CPUID#	**none**	**mov-from-IND-CPUID**[3]	specific
CR[CMCV]	**mov-to-CR-CMCV**	**mov-from-CR-CMCV**	data
CR[DCR]	**mov-to-CR-DCR**	**mov-from-CR-DCR**, **mem-readers-spec**	data

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CR[EOI]	**mov-to-CR-EOI**	**none**	SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 2:124
CR[IFA]	**mov-to-CR-IFA**	itc.i, itc.d, itr.i, itr.d	implied
		mov-from-CR-IFA	data
CR[IFS]	**mov-to-CR-IFS**	**mov-from-CR-IFS**	data
		rfi	implied
	cover	rfi, **mov-from-CR-IFS**	implied
CR[IHA]	**mov-to-CR-IHA**	**mov-from-CR-IHA**	data
CR[IIB%], % in 0 - 1	**mov-to-CR-IIB**	**mov-from-CR-IIB**	data
CR[IIM]	**mov-to-CR-IIM**	**mov-from-CR-IIM**	data
CR[IIP]	**mov-to-CR-IIP**	**mov-from-CR-IIP**	data
		rfi	implied
CR[IIPA]	**mov-to-CR-IIPA**	**mov-from-CR-IIPA**	data
CR[IPSR]	**mov-to-CR-IPSR**	**mov-from-CR-IPSR**	data
		rfi	implied
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**	**mov-from-CR-IRR**[1]	data
CR[ISR]	**mov-to-CR-ISR**	**mov-from-CR-ISR**	data
CR[ITIR]	**mov-to-CR-ITIR**	**mov-from-CR-ITIR**	data
		itc.i, itc.d, itr.i, itr.d	implied
CR[ITM]	**mov-to-CR-ITM**	**mov-from-CR-ITM**	data
CR[ITO]	**mov-to-CR-ITO**	**mov-from-AR-ITC**, **mov-from-CR-ITO**	data
CR[ITV]	**mov-to-CR-ITV**	**mov-from-CR-ITV**	data
CR[IVA]	**mov-to-CR-IVA**	**mov-from-CR-IVA**	instr
CR[IVR]	**none**	**mov-from-CR-IVR**	SC Section 5.8.3.2, “External Interrupt Vector Register (IVR – CR65)” on page 2:123
CR[LID]	**mov-to-CR-LID**	**mov-from-CR-LID**	SC Section 5.8.3.1, “Local ID (LID – CR64)” on page 2:122
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]	**mov-from-CR-LRR**[1]	data
CR[PMV]	**mov-to-CR-PMV**	**mov-from-CR-PMV**	data
CR[PTA]	**mov-to-CR-PTA**	**mov-from-CR-PTA**, **mem-readers**, **mem-writers**, **non-access**, thash	data

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CR[TPR]	**mov-to-CR-TPR**	**mov-from-CR-TPR**, **mov-from-CR-IVR**	data
		mov-to-PSR-l[17], ssm[17]	SC Section 5.8.3.3, “Task Priority Register (TPR – CR66)” on page 2:123
		rfi	implied
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**	**mov-from-CR-rv**[1]	none
DBR#	**mov-to-IND-DBR**[3]	**mov-from-IND-DBR**[3]	impliedF
		probe-all, **lfetch-all**, **mem-readers**, **mem-writers**	data
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	**mem-readers**, **mem-writers**, **non-access**	data
	itc.i, itc.d, itr.i, itr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
		itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF
DTR	itr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d, itr.d	impliedF
	ptr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d	none
		itr.d, itc.d	impliedF
FR%, % in 0 - 1	**none**	**fr-readers**[1]	none
FR%, % in 2 - 127	**fr-writers**[1]**\ldf-c**[1]**\ldfp-c**[1]	**fr-readers**[1]	impliedF
	ldf-c[1], **ldfp-c**[1]	**fr-readers**[1]	none
GR0	**none**	**gr-readers**[1]	none
GR%, % in 1 - 127	**ld-c**[1,13]	**gr-readers**[1]	none
	gr-writers[1]**\ld-c**[1,13]	**gr-readers**[1]	impliedF
IBR#	**mov-to-IND-IBR**[3]	**mov-from-IND-IBR**[3]	impliedF
InService*	**mov-to-CR-EOI**	**mov-from-CR-IVR**	data
	mov-from-CR-IVR	**mov-from-CR-IVR**	impliedF
	mov-to-CR-EOI	**mov-to-CR-EOI**	impliedF
IP	**all**	**all**	none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	epc, vmsw	instr
		itc.i, itc.d, itr.i, itr.d	impliedF
		ptr.i, ptr.d, ptc.e, ptc.g, ptc.ga, ptc.l	none
	itc.i, itc.d, itr.i, itr.d	epc, vmsw	instr
		itc.d, itc.i, itr.d, itr.i, ptr.d, ptr.i, ptc.g, ptc.ga, ptc.l	impliedF
ITC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
ITR	itr.i	itr.i, itc.i, ptc.g, ptc.ga, ptc.l, ptr.i	impliedF
		epc, vmsw	instr
	ptr.i	itc.i, itr.i	impliedF
		ptc.g, ptc.ga, ptc.l, ptr.i	none
		epc, vmsw	instr
memory	**mem-writers**	**mem-readers**	none
PKR#	**mov-to-IND-PKR**[3]	**mem-readers**, **mem-writers**, **mov-from-IND-PKR**[4], **probe-all**	data
		mov-to-IND-PKR[4]	none
		mov-from-IND-PKR[3]	impliedF
		mov-to-IND-PKR[3]	impliedF
PMC#	**mov-to-IND-PMC**[3]	**mov-from-IND-PMC**[3]	impliedF
		mov-from-IND-PMD[3]	SC Section 7.2.1, "Generic Performance Counter Registers" for PMC[0].fr on page 2:156
PMD#	**mov-to-IND-PMD**[3]	**mov-from-IND-PMD**[3]	impliedF
PR0	**pr-writers**[1]	**pr-readers-br**[1], **pr-readers-nobr-nomovpr**[1], **mov-from-PR**[12], **mov-to-PR**[12]	none
PR%, % in 1 - 15	**pr-writers**[1], **mov-to-PR-allreg**[7]	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7]	**pr-readers-br**[1]	none
PR%, % in 16 - 62	**pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none
PR63	**mod-sched-brs**, **pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	impliedF
	pr-writers-fp[1], **mod-sched-brs**	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.bn	bsw, rfi	**gr-readers**[10], **gr-writers**[10]	impliedF
PSR.cpl	epc, br.ret	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	implied
	rfi	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	impliedF
PSR.da	rfi	**mem-readers**, **lfetch-all**, **mem-writers**, **probe-fault**	impliedF
PSR.db	**mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-fault**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-fault**	impliedF
PSR.dd	rfi	**lfetch-all**, **mem-readers**, **probe-fault**, **mem-writers**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-readers**[8], **fr-writers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-readers**[8], **fr-writers**[8], **mov-from-PSR**	impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-writers**[8], **fr-readers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-writers**[8], **fr-readers**[8], **mov-from-PSR**	impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	br.ia	data
		mov-from-PSR	impliedF
	rfi	br.ia, **mov-from-PSR**	impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**, **non-access**	data
		mov-from-PSR	impliedF
	rfi	**mem-readers**, **mem-writers**, **non-access**, **mov-from-PSR**	impliedF
PSR.ed	rfi	**lfetch-all**, **mem-readers-spec**	impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ia	rfi	**all**	none
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	data
	rfi	**mov-from-PSR**, cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	impliedF
PSR.id	rfi	**all**	none
PSR.is	br.ia, rfi	**none**	none
PSR.it	rfi	**branches**, **mov-from-PSR**, chk, epc, fchkf, vmsw	impliedF
PSR.lp	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		br.ret	data
	rfi	**mov-from-PSR**, br.ret	impliedF
PSR.mc	rfi	**mov-from-PSR**	impliedF
PSR.mfh	**fr-writers**[9], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.mfl	**fr-writers**[9], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-all**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-all**	impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ri	rfi	**all**	none
PSR.rt	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		alloc, flushrs, loadrs	data
	rfi	**mov-from-PSR**, alloc, flushrs, loadrs	impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-AR-ITC, **mov-from-AR-RUC**	data
	rfi	**mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-PSR**	impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-IND-PMD, **mov-to-PSR-um**, rum, sum	data
	rfi	**mov-from-IND-PMD**, **mov-from-PSR**, **mov-to-PSR-um**, rum, sum	impliedF
PSR.ss	rfi	**all**	impliedF
PSR.tb	**mov-to-PSR-l**	**branches**, chk, fchkf	data
		mov-from-PSR	impliedF
	rfi	**branches**, chk, fchkf, **mov-from-PSR**	impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.vm	vmsw	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	implied
	rfi	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	impliedF
RR#	**mov-to-IND-RR**[6]	**mem-readers**, **mem-writers**, itc.i, itc.d, itr.i, itr.d, **non-access**, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, thash, ttag	data
		mov-from-IND-RR[6]	impliedF
RSE	**rse-writers**[14]	**rse-readers**[14]	impliedF

5.3.3 WAW Dependency Table

General rules specific to the WAW table:

- All resources require at most an instruction group break to provide sequential behavior.
- Some resources require no instruction group break to provide sequential behavior.
- There are a few special cases that are described in greater detail elsewhere in the manual and are indicated with an SC (special case) result.
- Each sub-row of writers represents a group of instructions that when taken in pairs in any combination has the dependency result indicated. If the column is split in sub-columns, then the dependency semantics apply to any pair of instructions where one is chosen from left sub-column and one is chosen from the right sub-column.

Table 5-3. WAW Dependencies Organized by Resource

Resource Name	Writers		Semantics of Dependency
ALAT	**mem-readers-alat**, **mem-writers**, chk.a.clr, **invala-all**		none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi		impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**		impliedF
AR[CCV]	**mov-to-AR-CCV**		impliedF
AR[CFLG]	**mov-to-AR-CFLG**		impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**		impliedF
AR[EC]	br.ret, **mod-sched-brs**, **mov-to-AR-EC**		impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**		impliedF
AR[FCR]	**mov-to-AR-FCR**		impliedF
AR[FDR]	**mov-to-AR-FDR**		impliedF
AR[FIR]	**mov-to-AR-FIR**		impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0		impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1		impliedF
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2		impliedF
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3		impliedF
AR[FPSR].sf0.flags	**fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**		none
	fclrf.s0, **fcmp-s0**, **fp-arith-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	fclrf.s0, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**		none
	fclrf.s1, **fcmp-s1**, **fp-arith-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	fclrf.s1, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf2.flags	**fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**		none
	fclrf.s2, **fcmp-s2**, **fp-arith-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	fclrf.s2, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf3.flags	**fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**		none
	fclrf.s3, **fcmp-s3**, **fp-arith-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	fclrf.s3, **mov-to-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**		impliedF
AR[FPSR].traps	**mov-to-AR-FPSR**		impliedF
AR[FSR]	**mov-to-AR-FSR**		impliedF
AR[ITC]	**mov-to-AR-ITC**		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]		impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**		impliedF
AR[PFS]	br.call, brl.call		none
	br.call, brl.call	**mov-to-AR-PFS**	impliedF
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**		impliedF
AR[RSC]	**mov-to-AR-RSC**		impliedF
AR[RUC]	**mov-to-AR-RUC**		impliedF
AR[SSD]	**mov-to-AR-SSD**		impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill		impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	**none**		none
AR%, % in 48 - 63, 112-127	**mov-to-AR-ig**[1]		impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**mov-to-BR**[1]	impliedF
	mov-to-BR[1]		impliedF
	br.call[1], brl.call[1]		none
CFM	**mod-sched-brs**, br.call, brl.call, br.ret, alloc, clrrrb, cover, rfi		impliedF
CPUID#	**none**		none
CR[CMCV]	**mov-to-CR-CMCV**		impliedF
CR[DCR]	**mov-to-CR-DCR**		impliedF
CR[EOI]	**mov-to-CR-EOI**		SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 2:124
CR[IFA]	**mov-to-CR-IFA**		impliedF
CR[IFS]	**mov-to-CR-IFS**, cover		impliedF
CR[IHA]	**mov-to-CR-IHA**		impliedF
CR[IIB%], % in 0 - 1	**mov-to-CR-IIB**		impliedF
CR[IIM]	**mov-to-CR-IIM**		impliedF
CR[IIP]	**mov-to-CR-IIP**		impliedF
CR[IIPA]	**mov-to-CR-IIPA**		impliedF
CR[IPSR]	**mov-to-CR-IPSR**		impliedF
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**		impliedF
CR[ISR]	**mov-to-CR-ISR**		impliedF
CR[ITIR]	**mov-to-CR-ITIR**		impliedF
CR[ITM]	**mov-to-CR-ITM**		impliedF
CR[ITO]	**mov-to-CR-ITO**		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
CR[ITV]	**mov-to-CR-ITV**		impliedF
CR[IVA]	**mov-to-CR-IVA**		impliedF
CR[IVR]	**none**		SC
CR[LID]	**mov-to-CR-LID**		SC
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]		impliedF
CR[PMV]	**mov-to-CR-PMV**		impliedF
CR[PTA]	**mov-to-CR-PTA**		impliedF
CR[TPR]	**mov-to-CR-TPR**		impliedF
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**		none
DBR#	**mov-to-IND-DBR**[3]		impliedF
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga		impliedF
DTR	itr.d		impliedF
	itr.d	ptr.d	impliedF
	ptr.d		none
FR%, % in 0 - 1	**none**		none
FR%, % in 2 - 127	**fr-writers**[1], **ldf-c**[1], **ldfp-c**[1]		impliedF
GR0	**none**		none
GR%, % in 1 - 127	**ld-c**[1], **gr-writers**[1]		impliedF
IBR#	**mov-to-IND-IBR**[3]		impliedF
InService*	**mov-to-CR-EOI**, **mov-from-CR-IVR**		SC
IP	**all**		none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
ITR	itr.i	itr.i, ptr.i	impliedF
	ptr.i		none
memory	**mem-writers**		none
PKR#	**mov-to-IND-PKR**[3]	**mov-to-IND-PKR**[4]	none
	mov-to-IND-PKR[3]		impliedF
PMC#	**mov-to-IND-PMC**[3]		impliedF
PMD#	**mov-to-IND-PMD**[3]		impliedF
PR0	**pr-writers**[1]		none

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
PR%, % in 1 - 15	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7]	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7]	impliedF
PR%, % in 16 - 62	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF
PR63	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	mod-sched-brs, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**mod-sched-brs**, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.bn	bsw, rfi		impliedF
PSR.cpl	epc, br.ret, rfi		impliedF
PSR.da	rfi		impliedF
PSR.db	**mov-to-PSR-l**, rfi		impliedF
PSR.dd	rfi		impliedF
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ed	rfi		impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ia	rfi		impliedF
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.id	rfi		impliedF
PSR.is	br.ia, rfi		impliedF
PSR.it	rfi		impliedF
PSR.lp	**mov-to-PSR-l**, rfi		impliedF
PSR.mc	rfi		impliedF

Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
PSR.mfh	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.mfl	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ri	rfi		impliedF
PSR.rt	**mov-to-PSR-l**, rfi		impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ss	rfi		impliedF
PSR.tb	**mov-to-PSR-l**, rfi		impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.vm	rfi, vmsw		impliedF
RR#	**mov-to-IND-RR**[6]		impliedF
RSE	**rse-writers**[14]		impliedF

5.3.4 WAR Dependency Table

A general rule specific to the WAR table:

1. WAR dependencies are always allowed within instruction groups except for the entry in Table 5-4 below. The readers and subsequent writers specified must be separated by a stop in order to have defined behavior.

Table 5-4. WAR Dependencies Organized by Resource

Resource Name	Readers	Writers	Semantics of Dependency
PR63	**pr-readers-br**[1]	**mod-sched-brs**	stop

5.3.5 Listing of Rules Referenced in Dependency Tables

The following rules restrict the specific instances in which some of the instructions in the tables cause a dependency and must be applied where referenced to correctly interpret those entries. Rules only apply to the instance of the instruction class, or instruction mnemonic prefix where the rule is referenced as a superscript. If the rule is referenced in Table 5-5 where instruction classes are defined, then it applies to all instances of the instruction class.

Rule 1. These instructions only write a register when that register’s number is explicitly encoded as a target of the instruction and is only read when it is encoded as a source of the instruction (or encoded as its PR[qp]).

Rule 2. These instructions only read CFM when they access a rotating GR, FR, or PR. **mov-to-PR** and **mov-from-PR** only access CFM when their qualifying predicate is in the rotating region.

Rule 3. These instructions use a general register value to determine the specific indirect register accessed. These instructions only access the register resource specified by the value in bits {7:0} of the dynamic value of the index register.

Rule 4. These instructions only read the given resource when bits {7:0} of the indirect index register value *does not* match the register number of the resource.

Rule 5. All rules are implementation specific.

Rule 6. There is a dependency only when both the index specified by the reader and the index specified by the writer have the same value in bits {63:61}.

Rule 7. These instructions access the specified resource only when the corresponding mask bit is set.

Rule 8. PSR.dfh is only read when these instructions reference FR32-127. PSR.dfl is only read when these instructions reference FR2-31.

Rule 9. PSR.mfl is only written when these instructions write FR2-31. PSR.mfh is only written when these instructions write FR32-127.

Rule 10.The PSR.bn bit is only accessed when one of GR16-31 is specified in the instruction.

Rule 11.The target predicates are written independently of PR[qp], but source registers are only read if PR[qp] is true.

Rule 12.This instruction only reads the specified predicate register when that register is the PR[qp].

Rule 13.This reference to ld-c only applies to the GR whose value is loaded with data returned from memory, not the post-incremented address register. Thus, a stop is still required between a post-incrementing ld-c and a consumer that reads the post-incremented GR.

Rule 14.The RSE resource includes implementation-specific internal state. At least one (and possibly more) of these resources are read by each instruction listed in the **rse-readers** class. At least one (and possibly more) of these resources are written by each instruction listed in the **rse-writers** class. To determine exactly which instructions read or write each individual resource, see the corresponding instruction pages.

Rule 15.This class represents all instructions marked as Reserved if PR[qp] is 1 B-type instructions as described in "Format Summary" on page 3:294.

Rule 16.This class represents all instructions marked as Reserved if PR[qp] is 1 instructions as described in "Format Summary" on page 3:294.

Rule 17.CR[TPR] has a RAW dependency only between **mov-to-CR-TPR** and **mov-to-PSR-l** or ssm instructions that set PSR.i, PSR.pp or PSR.up.

5.4 Support Tables

Table 5-5. Instruction Classes

Class	Events/Instructions
all	**predicatable-instructions**, **unpredicatable-instructions**
branches	**indirect-brs**, **ip-rel-brs**
cfm-readers	**fr-readers**, **fr-writers**, **gr-readers**, **gr-writers**, **mod-sched-brs**, **predicatable-instructions**, **pr-writers**, alloc, br.call, brl.call, br.ret, cover, loadrs, rfi, **chk-a**, invala.e
chk-a	chk.a.clr, chk.a.nc
cmpxchg	cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8, cmp8xchg16
czx	czx1, czx2
fcmp-s0	fcmp[Field(sf)==s0]
fcmp-s1	fcmp[Field(sf)==s1]
fcmp-s2	fcmp[Field(sf)==s2]
fcmp-s3	fcmp[Field(sf)==s3]
fetchadd	fetchadd4, fetchadd8
fp-arith	fadd, famax, famin, fcvt.fx, fcvt.fxu, fcvt.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmpy, fnorm, fpamax, fpamin, fpcvt.fx, fpcvt.fxu, fpma, fpmax, fpmin, fpmpy, fpms, fpnma, fpnmpy, fprcpa, fprsqrta, frcpa, frsqrta, fsub
fp-arith-s0	**fp-arith**[Field(sf)==s0]
fp-arith-s1	**fp-arith**[Field(sf)==s1]
fp-arith-s2	**fp-arith**[Field(sf)==s2]
fp-arith-s3	**fp-arith**[Field(sf)==s3]
fp-non-arith	fabs, fand, fandcm, fclass, fcvt.xf, fmerge, fmix, fneg, fnegabs, for, fpabs, fpmerge, fpack, fpneg, fpnegabs, fselect, fswap, fsxt, fxor, xma, xmpy
fpcmp-s0	fpcmp[Field(sf)==s0]
fpcmp-s1	fpcmp[Field(sf)==s1]
fpcmp-s2	fpcmp[Field(sf)==s2]
fpcmp-s3	fpcmp[Field(sf)==s3]
fr-readers	**fp-arith**, **fp-non-arith**, **mem-writers-fp**, **pr-writers-fp**, chk.s[Format in {M21}], getf
fr-writers	**fp-arith**, **fp-non-arith**\fclass, **mem-readers-fp**, setf
gr-readers	**gr-readers-writers**, **mem-readers**, **mem-writers**, chk.s, cmp, cmp4, fc, itc.i, itc.d, itr.i, itr.d, **mov-to-AR-gr**, **mov-to-BR**, **mov-to-CR**, **mov-to-IND**, **mov-from-IND**, **mov-to-PR-allreg**, **mov-to-PSR-l**, **mov-to-PSR-um**, **probe-all**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, setf, tbit, tnat
gr-readers-writers	**mov-from-IND**, add, addl, addp4, adds, and, andcm, clz, **czx**, dep\dep[Format in {I13}], extr, **mem-readers-int**, **ld-all-postinc**, **lfetch-postinc**, **mix**, **mux**, or, **pack**, **padd**, **pavg**, **pavgsub**, **pcmp**, **pmax**, **pmin**, **pmpy**, **pmpyshr**, popcnt, **probe-regular**, **psad**, **pshl**, **pshladd**, **pshr**, **pshradd**, **psub**, shl, shladd, shladdp4, shr, shrp, **st-postinc**, sub, **sxt**, tak, thash, tpa, ttag, **unpack**, xor, **zxt**
gr-writers	alloc, dep, getf, **gr-readers-writers**, **mem-readers-int**, **mov-from-AR**, **mov-from-BR**, **mov-from-CR**, **mov-from-PR**, **mov-from-PSR**, **mov-from-PSR-um**, **mov-ip**, movl
indirect-brp	brp[Format in {B7}]
indirect-brs	br.call[Format in {B5}], br.cond[Format in {B4}], br.ia, br.ret
invala-all	invala[Format in {M24}], invala.e
ip-rel-brs	**mod-sched-brs**, br.call[Format in {B3}], brl.call, brl.cond, br.cond[Format in {B1}], br.cloop
ld	ld1, ld2, ld4, ld8, ld8.fill, ld16
ld-a	ld1.a, ld2.a, ld4.a, ld8.a

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
ld-all-postinc	**ld**[Format in {M2 M3}], **ldfp**[Format in {M12}], **ldf**[Format in {M7 M8}]
ld-c	**ld-c-nc**, **ld-c-clr**
ld-c-clr	ld1.c.clr, ld2.c.clr, ld4.c.clr, ld8.c.clr, **ld-c-clr-acq**
ld-c-clr-acq	ld1.c.clr.acq, ld2.c.clr.acq, ld4.c.clr.acq, ld8.c.clr.acq
ld-c-nc	ld1.c.nc, ld2.c.nc, ld4.c.nc, ld8.c.nc
ld-s	ld1.s, ld2.s, ld4.s, ld8.s
ld-sa	ld1.sa, ld2.sa, ld4.sa, ld8.sa
ldf	ldfs, ldfd, ldfe, ldf8, ldf.fill
ldf-a	ldfs.a, ldfd.a, ldfe.a, ldf8.a
ldf-c	**ldf-c-nc**, **ldf-c-clr**
ldf-c-clr	ldfs.c.clr, ldfd.c.clr, ldfe.c.clr, ldf8.c.clr
ldf-c-nc	ldfs.c.nc, ldfd.c.nc, ldfe.c.nc, ldf8.c.nc
ldf-s	ldfs.s, ldfd.s, ldfe.s, ldf8.s
ldf-sa	ldfs.sa, ldfd.sa, ldfe.sa, ldf8.sa
ldfp	ldfps, ldfpd, ldfp8
ldfp-a	ldfps.a, ldfpd.a, ldfp8.a
ldfp-c	**ldfp-c-nc**, **ldfp-c-clr**
ldfp-c-clr	ldfps.c.clr, ldfpd.c.clr, ldfp8.c.clr
ldfp-c-nc	ldfps.c.nc, ldfpd.c.nc, ldfp8.c.nc
ldfp-s	ldfps.s, ldfpd.s, ldfp8.s
ldfp-sa	ldfps.sa, ldfpd.sa, ldfp8.sa
lfetch-all	lfetch
lfetch-fault	lfetch[Field(lftype)==fault]
lfetch-nofault	lfetch[Field(lftype)==]
lfetch-postinc	lfetch[Format in {M20 M22}]
mem-readers	**mem-readers-fp**, **mem-readers-int**
mem-readers-alat	**ld-a**, **ldf-a**, **ldfp-a**, **ld-sa**, **ldf-sa**, **ldfp-sa**, **ld-c**, **ldf-c**, **ldfp-c**
mem-readers-fp	**ldf**, **ldfp**
mem-readers-int	**cmpxchg**, **fetchadd**, **xchg**, **ld**
mem-readers-spec	**ld-s**, **ld-sa**, **ldf-s**, **ldf-sa**, **ldfp-s**, **ldfp-sa**
mem-writers	**mem-writers-fp**, **mem-writers-int**
mem-writers-fp	**stf**
mem-writers-int	**cmpxchg**, **fetchadd**, **xchg**, **st**
mix	mix1, mix2, mix4
mod-sched-brs	br.cexit, br.ctop, br.wexit, br.wtop
mod-sched-brs-counted	br.cexit, br.cloop, br.ctop
mov-from-AR	**mov-from-AR-M**, **mov-from-AR-I**, **mov-from-AR-IM**
mov-from-AR-BSP	**mov-from-AR-M**[Field(ar3) == BSP]
mov-from-AR-BSPSTORE	**mov-from-AR-M**[Field(ar3) == BSPSTORE]
mov-from-AR-CCV	**mov-from-AR-M**[Field(ar3) == CCV]
mov-from-AR-CFLG	**mov-from-AR-M**[Field(ar3) == CFLG]
mov-from-AR-CSD	**mov-from-AR-M**[Field(ar3) == CSD]
mov-from-AR-EC	**mov-from-AR-I**[Field(ar3) == EC]
mov-from-AR-EFLAG	**mov-from-AR-M**[Field(ar3) == EFLAG]
mov-from-AR-FCR	**mov-from-AR-M**[Field(ar3) == FCR]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-from-AR-FDR	**mov-from-AR-M**[Field(ar3) == FDR]
mov-from-AR-FIR	**mov-from-AR-M**[Field(ar3) == FIR]
mov-from-AR-FPSR	**mov-from-AR-M**[Field(ar3) == FPSR]
mov-from-AR-FSR	**mov-from-AR-M**[Field(ar3) == FSR]
mov-from-AR-I	mov_ar[Format in {I28}]
mov-from-AR-ig	**mov-from-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-from-AR-IM	mov_ar[Format in {I28 M31}]
mov-from-AR-ITC	**mov-from-AR-M**[Field(ar3) == ITC]
mov-from-AR-K	**mov-from-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-from-AR-LC	**mov-from-AR-I**[Field(ar3) == LC]
mov-from-AR-M	mov_ar[Format in {M31}]
mov-from-AR-PFS	**mov-from-AR-I**[Field(ar3) == PFS]
mov-from-AR-RNAT	**mov-from-AR-M**[Field(ar3) == RNAT]
mov-from-AR-RSC	**mov-from-AR-M**[Field(ar3) == RSC]
mov-from-AR-RUC	**mov-from-AR-M**[Field(ar3) == RUC]
mov-from-AR-rv	**none**
mov-from-AR-SSD	**mov-from-AR-M**[Field(ar3) == SSD]
mov-from-AR-UNAT	**mov-from-AR-M**[Field(ar3) == UNAT]
mov-from-BR	mov_br[Format in {I22}]
mov-from-CR	mov_cr[Format in {M33}]
mov-from-CR-CMCV	**mov-from-CR**[Field(cr3) == CMCV]
mov-from-CR-DCR	**mov-from-CR**[Field(cr3) == DCR]
mov-from-CR-EOI	**mov-from-CR**[Field(cr3) == EOI]
mov-from-CR-IFA	**mov-from-CR**[Field(cr3) == IFA]
mov-from-CR-IFS	**mov-from-CR**[Field(cr3) == IFS]
mov-from-CR-IHA	**mov-from-CR**[Field(cr3) == IHA]
mov-from-CR-IIB	**mov-from-CR**[Field(cr3) in {IIB0 IIB1}]
mov-from-CR-IIM	**mov-from-CR**[Field(cr3) == IIM]
mov-from-CR-IIP	**mov-from-CR**[Field(cr3) == IIP]
mov-from-CR-IIPA	**mov-from-CR**[Field(cr3) == IIPA]
mov-from-CR-IPSR	**mov-from-CR**[Field(cr3) == IPSR]
mov-from-CR-IRR	**mov-from-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-from-CR-ISR	**mov-from-CR**[Field(cr3) == ISR]
mov-from-CR-ITIR	**mov-from-CR**[Field(cr3) == ITIR]
mov-from-CR-ITM	**mov-from-CR**[Field(cr3) == ITM]
mov-from-CR-ITO	**mov-from-CR**[Field(cr3) == ITO]
mov-from-CR-ITV	**mov-from-CR**[Field(cr3) == ITV]
mov-from-CR-IVA	**mov-from-CR**[Field(cr3) == IVA]
mov-from-CR-IVR	**mov-from-CR**[Field(cr3) == IVR]
mov-from-CR-LID	**mov-from-CR**[Field(cr3) == LID]
mov-from-CR-LRR	**mov-from-CR**[Field(cr3) in {LRR0 LRR1}]
mov-from-CR-PMV	**mov-from-CR**[Field(cr3) == PMV]
mov-from-CR-PTA	**mov-from-CR**[Field(cr3) == PTA]
mov-from-CR-rv	**none**
mov-from-CR-TPR	**mov-from-CR**[Field(cr3) == TPR]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-from-IND	mov_indirect[Format in {M43}]
mov-from-IND-CPUID	**mov-from-IND**[Field(ireg) == cpuid]
mov-from-IND-DBR	**mov-from-IND**[Field(ireg) == dbr]
mov-from-IND-IBR	**mov-from-IND**[Field(ireg) == ibr]
mov-from-IND-PKR	**mov-from-IND**[Field(ireg) == pkr]
mov-from-IND-PMC	**mov-from-IND**[Field(ireg) == pmc]
mov-from-IND-PMD	**mov-from-IND**[Field(ireg) == pmd]
mov-from-IND-priv	**mov-from-IND**[Field(ireg) in {dbr ibr pkr pmc rr}]
mov-from-IND-RR	**mov-from-IND**[Field(ireg) == rr]
mov-from-interruption-CR	**mov-from-CR-ITIR**, **mov-from-CR-IFS**, **mov-from-CR-IIB**, **mov-from-CR-IIM**, **mov-from-CR-IIP**, **mov-from-CR-IPSR**, **mov-from-CR-ISR**, **mov-from-CR-IFA**, **mov-from-CR-IHA**, **mov-from-CR-IIPA**
mov-from-PR	mov_pr[Format in {I25}]
mov-from-PSR	mov_psr[Format in {M36}]
mov-from-PSR-um	mov_um[Format in {M36}]
mov-ip	mov_ip[Format in {I25}]
mov-to-AR	**mov-to-AR-M**, **mov-to-AR-I**
mov-to-AR-BSP	**mov-to-AR-M**[Field(ar3) == BSP]
mov-to-AR-BSPSTORE	**mov-to-AR-M**[Field(ar3) == BSPSTORE]
mov-to-AR-CCV	**mov-to-AR-M**[Field(ar3) == CCV]
mov-to-AR-CFLG	**mov-to-AR-M**[Field(ar3) == CFLG]
mov-to-AR-CSD	**mov-to-AR-M**[Field(ar3) == CSD]
mov-to-AR-EC	**mov-to-AR-I**[Field(ar3) == EC]
mov-to-AR-EFLAG	**mov-to-AR-M**[Field(ar3) == EFLAG]
mov-to-AR-FCR	**mov-to-AR-M**[Field(ar3) == FCR]
mov-to-AR-FDR	**mov-to-AR-M**[Field(ar3) == FDR]
mov-to-AR-FIR	**mov-to-AR-M**[Field(ar3) == FIR]
mov-to-AR-FPSR	**mov-to-AR-M**[Field(ar3) == FPSR]
mov-to-AR-FSR	**mov-to-AR-M**[Field(ar3) == FSR]
mov-to-AR-gr	**mov-to-AR-M**[Format in {M29}], **mov-to-AR-I**[Format in {I26}]
mov-to-AR-I	mov_ar[Format in {I26 I27}]
mov-to-AR-ig	**mov-to-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-to-AR-IM	mov_ar[Format in {I26 I27 M29 M30}]
mov-to-AR-ITC	**mov-to-AR-M**[Field(ar3) == ITC]
mov-to-AR-K	**mov-to-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-to-AR-LC	**mov-to-AR-I**[Field(ar3) == LC]
mov-to-AR-M	mov_ar[Format in {M29 M30}]
mov-to-AR-PFS	**mov-to-AR-I**[Field(ar3) == PFS]
mov-to-AR-RNAT	**mov-to-AR-M**[Field(ar3) == RNAT]
mov-to-AR-RSC	**mov-to-AR-M**[Field(ar3) == RSC]
mov-to-AR-RUC	**mov-to-AR-M**[Field(ar3) == RUC]
mov-to-AR-SSD	**mov-to-AR-M**[Field(ar3) == SSD]
mov-to-AR-UNAT	**mov-to-AR-M**[Field(ar3) == UNAT]
mov-to-BR	mov_br[Format in {I21}]
mov-to-CR	mov_cr[Format in {M32}]
mov-to-CR-CMCV	**mov-to-CR**[Field(cr3) == CMCV]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-to-CR-DCR	**mov-to-CR**[Field(cr3) == DCR]
mov-to-CR-EOI	**mov-to-CR**[Field(cr3) == EOI]
mov-to-CR-IFA	**mov-to-CR**[Field(cr3) == IFA]
mov-to-CR-IFS	**mov-to-CR**[Field(cr3) == IFS]
mov-to-CR-IHA	**mov-to-CR**[Field(cr3) == IHA]
mov-to-CR-IIB	**mov-to-CR**[Field(cr3) in {IIB0 IIB1}]
mov-to-CR-IIM	**mov-to-CR**[Field(cr3) == IIM]
mov-to-CR-IIP	**mov-to-CR**[Field(cr3) == IIP]
mov-to-CR-IIPA	**mov-to-CR**[Field(cr3) == IIPA]
mov-to-CR-IPSR	**mov-to-CR**[Field(cr3) == IPSR]
mov-to-CR-IRR	**mov-to-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-to-CR-ISR	**mov-to-CR**[Field(cr3) == ISR]
mov-to-CR-ITIR	**mov-to-CR**[Field(cr3) == ITIR]
mov-to-CR-ITM	**mov-to-CR**[Field(cr3) == ITM]
mov-to-CR-ITO	**mov-to-CR**[Field(cr3) == ITO]
mov-to-CR-ITV	**mov-to-CR**[Field(cr3) == ITV]
mov-to-CR-IVA	**mov-to-CR**[Field(cr3) == IVA]
mov-to-CR-IVR	**mov-to-CR**[Field(cr3) == IVR]
mov-to-CR-LID	**mov-to-CR**[Field(cr3) == LID]
mov-to-CR-LRR	**mov-to-CR**[Field(cr3) in {LRR0 LRR1}]
mov-to-CR-PMV	**mov-to-CR**[Field(cr3) == PMV]
mov-to-CR-PTA	**mov-to-CR**[Field(cr3) == PTA]
mov-to-CR-TPR	**mov-to-CR**[Field(cr3) == TPR]
mov-to-IND	mov_indirect[Format in {M42}]
mov-to-IND-CPUID	**mov-to-IND**[Field(ireg) == cpuid]
mov-to-IND-DBR	**mov-to-IND**[Field(ireg) == dbr]
mov-to-IND-IBR	**mov-to-IND**[Field(ireg) == ibr]
mov-to-IND-PKR	**mov-to-IND**[Field(ireg) == pkr]
mov-to-IND-PMC	**mov-to-IND**[Field(ireg) == pmc]
mov-to-IND-PMD	**mov-to-IND**[Field(ireg) == pmd]
mov-to-IND-priv	**mov-to-IND**
mov-to-IND-RR	**mov-to-IND**[Field(ireg) == rr]
mov-to-interruption-CR	**mov-to-CR-ITIR**, **mov-to-CR-IFS**, **mov-to-CR-IIB**, **mov-to-CR-IIM**, **mov-to-CR-IIP**, **mov-to-CR-IPSR**, **mov-to-CR-ISR**, **mov-to-CR-IFA**, **mov-to-CR-IHA**, **mov-to-CR-IIPA**
mov-to-PR	**mov-to-PR-allreg**, **mov-to-PR-rotreg**
mov-to-PR-allreg	mov_pr[Format in {I23}]
mov-to-PR-rotreg	mov_pr[Format in {I24}]
mov-to-PSR-l	mov_psr[Format in {M35}]
mov-to-PSR-um	mov_um[Format in {M35}]
mux	mux1, mux2
non-access	fc, lfetch, **probe-all**, tpa, tak
none	-
pack	pack2, pack4
padd	padd1, padd2, padd4
pavg	pavg1, pavg2

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
pavgsub	pavgsub1, pavgsub2
pcmp	pcmp1, pcmp2, pcmp4
pmax	pmax1, pmax2
pmin	pmin1, pmin2
pmpy	pmpy2
pmpyshr	pmpyshr2
pr-and-writers	**pr-gen-writers-int**[Field(ctype) in {and andcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-gen-writers-fp	fclass, fcmp
pr-gen-writers-int	cmp, cmp4, tbit, tf, tnat
pr-norm-writers-fp	**pr-gen-writers-fp**[Field(ctype)==]
pr-norm-writers-int	**pr-gen-writers-int**[Field(ctype)==]
pr-or-writers	**pr-gen-writers-int**[Field(ctype) in {or orcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-readers-br	br.call, br.cond, brl.call, brl.cond, br.ret, br.wexit, br.wtop, break.b, hint.b, nop.b, **ReservedBQP**
pr-readers-nobr-nomovpr	add, addl, addp4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, **chk-a**, cmp, cmp4, **cmpxchg**, clz, **czx**, dep, extr, **fp-arith**, **fp-non-arith**, fc, fchkf, fclrf, fcmp, **fetchadd**, fpcmp, fsetc, fwb, getf, hint.f, hint.i, hint.m, hint.x, **invala-all**, itc.i, itc.d, itr.i, itr.d, **ld**, **ldf**, **ldfp**, **lfetch-all**, mf, **mix**, **mov-from-AR-M**, **mov-from-AR-IM**, **mov-from-AR-I**, **mov-to-AR-M**, **mov-to-AR-I**, **mov-to-AR-IM**, **mov-to-BR**, **mov-from-BR**, **mov-to-CR**, **mov-from-CR**, **mov-to-IND**, **mov-from-IND**, **mov-ip**, **mov-to-PSR-l**, **mov-to-PSR-um**, **mov-from-PSR**, **mov-from-PSR-um**, movl, **mux**, nop.f, nop.i, nop.m, nop.x, or, **pack**, **padd**, **pavg**, **pavgsub**, **pcmp**, **pmax**, **pmin**, **pmpy**, **pmpyshr**, popcnt, **probe-all**, **psad**, **pshl**, **pshladd**, **pshr**, **pshradd**, **psub**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.d, ptr.i, **ReservedQP**, rsm, setf, shl, shladd, shladdp4, shr, shrp, srlz.i, srlz.d, ssm, **st**, **stf**, sub, sum, **sxt**, sync, tak, tbit, tf, thash, tnat, tpa, ttag, **unpack**, **xchg**, xma, xmpy, xor, **zxt**
pr-unc-writers-fp	**pr-gen-writers-fp**[Field(ctype)==unc][11], fprcpa[11], fprsqrta[11], frcpa[11], frsqrta[11]
pr-unc-writers-int	**pr-gen-writers-int**[Field(ctype)==unc][11]
pr-writers	**pr-writers-int**, **pr-writers-fp**
pr-writers-fp	**pr-norm-writers-fp**, **pr-unc-writers-fp**
pr-writers-int	**pr-norm-writers-int**, **pr-unc-writers-int**, **pr-and-writers**, **pr-or-writers**
predicatable-instructions	**mov-from-PR**, **mov-to-PR**, **pr-readers-br**, **pr-readers-nobr-nomovpr**
priv-ops	**mov-to-IND-priv**, bsw, itc.i, itc.d, itr.i, itr.d, **mov-to-CR**, **mov-from-CR**, **mov-to-PSR-l**, **mov-from-PSR**, **mov-from-IND-priv**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, rfi, rsm, ssm, tak, tpa, vmsw
probe-all	**probe-fault**, **probe-regular**
probe-fault	probe[Format in {M40}]
probe-regular	probe[Format in {M38 M39}]
psad	psad1
pshl	pshl2, pshl4
pshladd	pshladd2
pshr	pshr2, pshr4
pshradd	pshradd2
psub	psub1, psub2, psub4
ReservedBQP	-[15]
ReservedQP	-[16]

Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
rse-readers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-from-AR-BSP**, **mov-from-AR-BSPSTORE**, **mov-to-AR-BSPSTORE**, **mov-from-AR-RNAT**, **mov-to-AR-RNAT**, rfi
rse-writers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-to-AR-BSPSTORE**, rfi
st	st1, st2, st4, st8, st8.spill, st16
st-postinc	**stf**[Format in {M10}], **st**[Format in {M5}]
stf	stfs, stfd, stfe, stf8, stf.spill
sxt	sxt1, sxt2, sxt4
sys-mask-writers-partial	rsm, ssm
unpack	unpack1, unpack2, unpack4
unpredicatable-instructions	alloc, br.cloop, br.ctop, br.cexit, br.ia, brp, bsw, clrrrb, cover, epc, flushrs, loadrs, rfi, vmsw
user-mask-writers-partial	rum, sum
xchg	xchg1, xchg2, xchg4, xchg8
zxt	zxt1, zxt2, zxt4

§

Intel® Itanium® Architecture

Software Developer's Manual

Revision 2.3

Volume 4: IA-32 Instruction Set

Intel® Itanium® Architecture Software Developer's Manual

Volume 4: IA-32 Instruction Set Reference

Revision 2.3

May 2010

Document Number: 323208

Contents

Figures

Tables

§

About this Manual 1

The Intel® Itanium® architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The Itanium architecture features a revolutionary 64-bit instruction set architecture (ISA) which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the Itanium architecture is IA-32 instruction set compatibility.

The *Intel® Itanium® Architecture Software Developer's Manual* provides a comprehensive description of the programming environment, resources, and instruction set visible to both the application and system programmer. In addition, it also describes how programmers can take advantage of the features of the Itanium architecture to help them optimize code.

1.1 Overview of Volume 1: Application Architecture

This volume defines the Itanium application architecture, including application level resources, programming environment, and the IA-32 application interface. This volume also describes optimization techniques used to generate high performance software.

1.1.1 Part 1: Application Architecture Guide

Chapter 1, "About this Manual" provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer's Manual*.

Chapter 2, "Introduction to the Intel® Itanium® Architecture" provides an overview of the architecture.

Chapter 3, "Execution Environment" describes the Itanium register set used by applications and the memory organization models.

Chapter 4, "Application Programming Model" gives an overview of the behavior of Itanium application instructions (grouped into related functions).

Chapter 5, "Floating-point Programming Model" describes the Itanium floating-point architecture (including integer multiply).

Chapter 6, "IA-32 Application Execution Model in an Intel® Itanium® System Environment" describes the operation of IA-32 instructions within the Itanium System Environment from the perspective of an application programmer.

1.1.2 Part 2: Optimization Guide for the Intel® Itanium® Architecture

Chapter 1, "About the Optimization Guide" gives an overview of the optimization guide.

Chapter 2, “Introduction to Programming for the Intel® Itanium® Architecture” provides an overview of the application programming environment for the Itanium architecture.

Chapter 3, “Memory Reference” discusses features and optimizations related to control and data speculation.

Chapter 4, “Predication, Control Flow, and Instruction Stream” describes optimization features related to predication, control flow, and branch hints.

Chapter 5, “Software Pipelining and Loop Support” provides a detailed discussion on optimizing loops through use of software pipelining.

Chapter 6, “Floating-point Applications” discusses current performance limitations in floating-point applications and features that address these limitations.

1.2 Overview of Volume 2: System Architecture

This volume defines the Itanium system architecture, including system level resources and programming state, interrupt model, and processor firmware interface. This volume also provides a useful system programmer's guide for writing high performance system software.

1.2.1 Part 1: System Architecture Guide

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Intel® Itanium® System Environment” introduces the environment designed to support execution of Itanium architecture-based operating systems running IA-32 or Itanium architecture-based applications.

Chapter 3, “System State and Programming Model” describes the Itanium architectural state which is visible only to an operating system.

Chapter 4, “Addressing and Protection” defines the resources available to the operating system for virtual to physical address translation, virtual aliasing, physical addressing, and memory ordering.

Chapter 5, “Interruptions” describes all interruptions that can be generated by a processor based on the Itanium architecture.

Chapter 6, “Register Stack Engine” describes the architectural mechanism which automatically saves and restores the stacked subset (GR32 – GR 127) of the general register file.

Chapter 7, “Debugging and Performance Monitoring” is an overview of the performance monitoring and debugging resources that are available in the Itanium architecture.

Chapter 8, “Interruption Vector Descriptions” lists all interruption vectors.

Chapter 9, "IA-32 Interruption Vector Descriptions" lists IA-32 exceptions, interrupts and intercepts that can occur during IA-32 instruction set execution in the Itanium System Environment.

Chapter 10, "Itanium® Architecture-based Operating System Interaction Model with IA-32 Applications" defines the operation of IA-32 instructions within the Itanium System Environment from the perspective of an Itanium architecture-based operating system.

Chapter 11, "Processor Abstraction Layer" describes the firmware layer which abstracts processor implementation-dependent features.

1.2.2 Part 2: System Programmer's Guide

Chapter 1, "About the System Programmer's Guide" gives an introduction to the second section of the system architecture guide.

Chapter 2, "MP Coherence and Synchronization" describes multiprocessing synchronization primitives and the Itanium memory ordering model.

Chapter 3, "Interruptions and Serialization" describes how the processor serializes execution around interruptions and what state is preserved and made available to low-level system code when interruptions are taken.

Chapter 4, "Context Management" describes how operating systems need to preserve Itanium register contents and state. This chapter also describes system architecture mechanisms that allow an operating system to reduce the number of registers that need to be spilled/filled on interruptions, system calls, and context switches.

Chapter 5, "Memory Management" introduces various memory management strategies.

Chapter 6, "Runtime Support for Control and Data Speculation" describes the operating system support that is required for control and data speculation.

Chapter 7, "Instruction Emulation and Other Fault Handlers" describes a variety of instruction emulation handlers that Itanium architecture-based operating systems are expected to support.

Chapter 8, "Floating-point System Software" discusses how processors based on the Itanium architecture handle floating-point numeric exceptions and how the software stack provides complete IEEE-754 compliance.

Chapter 9, "IA-32 Application Support" describes the support an Itanium architecture-based operating system needs to provide to host IA-32 applications.

Chapter 10, "External Interrupt Architecture" describes the external interrupt architecture with a focus on how external asynchronous interrupt handling can be controlled by software.

Chapter 11, "I/O Architecture" describes the I/O architecture with a focus on platform issues and support for the existing IA-32 I/O port space.

Chapter 12, “Performance Monitoring Support” describes the performance monitor architecture with a focus on what kind of support is needed from Itanium architecture-based operating systems.

Chapter 13, “Firmware Overview” introduces the firmware model, and how various firmware layers (PAL, SAL, UEFI, ACPI) work together to enable processor and system initialization, and operating system boot.

1.2.3 Appendices

Appendix A, “Code Examples” provides OS boot flow sample code.

1.3 Overview of Volume 3: Intel® Itanium® Instruction Set Reference

This volume is a comprehensive reference to the Itanium instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Instruction Reference” provides a detailed description of all Itanium instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “Pseudo-Code Functions” provides a table of pseudo-code functions which are used to define the behavior of the Itanium instructions.

Chapter 4, “Instruction Formats” describes the encoding and instruction format instructions.

Chapter 5, “Resource and Dependency Semantics” summarizes the dependency rules that are applicable when generating code for processors based on the Itanium architecture.

1.4 Overview of Volume 4: IA-32 Instruction Set Reference

This volume is a comprehensive reference to the IA-32 instruction set, including instruction format/encoding.

Chapter 1, “About this Manual” provides an overview of all volumes in the *Intel® Itanium® Architecture Software Developer’s Manual*.

Chapter 2, “Base IA-32 Instruction Reference” provides a detailed description of all base IA-32 instructions, organized in alphabetical order by assembly language mnemonic.

Chapter 3, “IA-32 Intel® MMX™ Technology Instruction Reference” provides a detailed description of all IA-32 Intel® MMX™ technology instructions designed to increase performance of multimedia intensive applications. Organized in alphabetical order by assembly language mnemonic.

Chapter 4, “IA-32 SSE Instruction Reference” provides a detailed description of all IA-32 SSE instructions designed to increase performance of multimedia intensive applications, and is organized in alphabetical order by assembly language mnemonic.

1.5 Terminology

The following definitions are for terms related to the Itanium architecture and will be used throughout this document:

Instruction Set Architecture (ISA) – Defines application and system level resources. These resources include instructions and registers.

Itanium Architecture – The new ISA with 64-bit instruction capabilities, new performance- enhancing features, and support for the IA-32 instruction set.

IA-32 Architecture – The 32-bit and 16-bit Intel architecture as described in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

Itanium System Environment – The operating system environment that supports the execution of both IA-32 and Itanium architecture-based code.

IA-32 System Environment – The operating system privileged environment and resources as defined by the *Intel Architecture Software Developer's Manual*. Resources include virtual paging, control registers, debugging, performance monitoring, machine checks, and the set of privileged instructions.

Itanium® Architecture-based Firmware – The Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL).

Processor Abstraction Layer (PAL) – The firmware layer which abstracts processor features that are implementation dependent.

System Abstraction Layer (SAL) – The firmware layer which abstracts system features that are implementation dependent.

1.6 Related Documents

The following documents can be downloaded at the Intel's Developer Site at http://developer.intel.com:

- ***Dual-Core Update to the Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization***– Document number 308065 provides model-specific information about the dual-core Itanium processors.
- ***Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*** – This document (Document number 251110) describes

model-specific architectural features incorporated into the Intel® Itanium® 2 processor, the second processor based on the Itanium architecture.

- ***Intel® Itanium® Processor Reference Manual for Software Development*** – This document (Document number 245320) describes model-specific architectural features incorporated into the Intel® Itanium® processor, the first processor based on the Itanium architecture.
- ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** – This set of manuals describes the Intel 32-bit architecture. They are available from the Intel Literature Department by calling 1-800-548-4725 and requesting Document Numbers 243190, 243191and 243192.
- ***Intel® Itanium® Software Conventions and Runtime Architecture Guide*** – This document (Document number 245358) defines general information necessary to compile, link, and execute a program on an Itanium architecture-based operating system.
- ***Intel® Itanium® Processor Family System Abstraction Layer Specification*** – This document (Document number 245359) specifies requirements to develop platform firmware for Itanium architecture-based systems.

The following document can be downloaded at the Unified EFI Forum website at http://www.uefi.org:

- ***Unified Extensible Firmware Interface Specification*** – This document defines a new model for the interface between operating systems and platform firmware.

1.7 Revision History

Date of Revision	Revision Number	Description
March 2010	2.3	Added information about illegal virtualization optimization combinations and IIPA requirements. Added Resource Utilization Counter and PAL_VP_INFO. PAL_VP_INIT and VPD.vpr changes. New PAL_VPS_RESUME_HANDLER parameter to indicate RSE Current Frame Load Enable setting at the target instruction. PAL_VP_INIT_ENV implementation-specific configuration option. Minimum Virtual address increased to 54 bits. New PAL_MC_ERROR_INFO health indicator. New PAL_MC_ERROR_INJECT implementation-specific bit fields. MOV-to_SR.L reserved field checking. Added virtual machine disable. Added variable frequency mode additions to ACPI P-state description. Removed *pal_proc_vector* argument from PAL_VP_SAVE and PAL_VP_RESTORE. Added PAL_PROC_SET_FEATURES data speculation disable. Added Interruption Instruction Bundle registers. Min-state save area size change. PAL_MC_DYNAMIC_STATE changes. PAL_PROC_SET_FEATURES data poisoning promotion changes. ACPI P-state clarifications. Synchronization requirements for virtualization opcode optimization. New priority hint and multi-threading hint recommendations.

Date of Revision	Revision Number	Description
August 2005	2.2	Allow register fields in CR.LID register to be read-only and CR.LID checking on interruption messages by processors optional. See Vol 2, Part I, Ch 5 “Interruptions” and Section 11.2.2 PALE_RESET Exit State for details. Relaxed reserved and ignored fields checkings in IA-32 application registers in Vol 1 Ch 6 and Vol 2, Part I, Ch 10. Introduced visibility constraints between stores and local purges to ensure TLB consistency for UP VHPT update and local purge scenarios. See Vol 2, Part I, Ch 4 and description of `ptc.l` instruction in Vol 3 for details. Architecture extensions for processor Power/Performance states (P-states). See Vol 2 PAL Chapter for details. Introduced Unimplemented Instruction Address fault. Relaxed ordering constraints for VHPT walks. See Vol 2, Part I, Ch 4 and 5 for details. Architecture extensions for processor virtualization. All instructions which must be last in an instruction group results in undefined behavior when this rule is violated. Added architectural sequence that guarantees increasing ITC and PMD values on successive reads. Addition of PAL_BRAND_INFO, PAL_GET_HW_POLICY, PAL_MC_ERROR_INJECT, PAL_MEMORY_BUFFER, PAL_SET_HW_POLICY and PAL_SHUTDOWN procedures. Allows IPI-redirection feature to be optional. Undefined behavior for 1-byte accesses to the non-architected regions in the IPI block. Modified insertion behavior for TR overlaps. See Vol 2, Part I, Ch 4 for details. “Bus parking” feature is now optional for PAL_BUS_GET_FEATURES. Introduced low-power synchronization primitive using `hint` instruction. FR32-127 is now preserved in PAL calling convention. New return value from PAL_VM_SUMMARY procedure to indicate the number of multiple concurrent outstanding TLB purges. Performance Monitor Data (PMD) registers are no longer sign-extended. New memory attribute transition sequence for memory on-line delete. See Vol 2, Part I, Ch 4 for details. Added 'shared error' (se) bit to the Processor State Parameter (PSP) in PAL_MC_ERROR_INFO procedure. Clarified PMU interrupts as edge-triggered. Modified ‘proc_number’ parameter in PAL_LOGICAL_TO_PHYSICAL procedure. Modified pal_copy_info alignment requirements. New bit in PAL_PROC_GET_FEATURES for variable P-state performance. Clarified descriptions for check_target_register and check_target_register_sof. Various fixes in dependency tables in Vol 3 Ch 5. Clarified effect of sending IPIs to non-existent processor in Vol 2, Part I, Ch 5. Clarified instruction serialization requirements for interruptions in Vol 2, Part II, Ch 3. Updated performance monitor context switch routine in Vol 2, Part I, Ch 7.

Date of Revision	Revision Number	Description
August 2002	2.1	Added Predicate Behavior of `alloc` Instruction Clarification (Section 4.1.2, Part I, Volume 1; Section 2.2, Part I, Volume 3). Added New `fc.i` Instruction (Section 4.4.6.1, and 4.4.6.2, Part I, Volume 1; Section 4.3.3, 4.4.1, 4.4.5, 4.4.6, 4.4.7, 5.5.2, and 7.1.2, Part I, Volume 2; Section 2.5, 2.5.1, 2.5.2, 2.5.3, and 4.5.2.1, Part II, Volume 2; Section 2.2, 3, 4.1, 4.4.6.5, and 4.4.10.10, Part I, Volume 3). Added Interval Time Counter (ITC) Fault Clarification (Section 3.3.2, Part I, Volume 2). Added Interruption Control Registers Clarification (Section 3.3.5, Part I, Volume 2). Added Spontaneous NaT Generation on Speculative Load (`ld.s`) (Section 5.5.5 and 11.9, Part I, Volume 2; Section 2.2 and 3, Part I, Volume 3). Added Performance Counter Standardization (Sections 7.2.3 and 11.6, Part I, Volume 2). Added Freeze Bit Functionality in Context Switching and Interrupt Generation Clarification (Sections 7.2.1, 7.2.2, 7.2.4.1, and 7.2.4.2, Part I, Volume 2) Added IA_32_Exception (Debug) IIPA Description Change (Section 9.2, Part I, Volume 2). Added capability for Allowing Multiple PAL_A_SPEC and PAL_B Entries in the Firmware Interface Table (Section 11.1.6, Part I, Volume 2). Added BR1 to Min-state Save Area (Sections 11.3.2.3 and 11.3.3, Part I, Volume 2). Added Fault Handling Semantics for `lfetch.fault` Instruction (Section 2.2, Part I, Volume 3).
December 2001	2.0	Volume 1: Faults in ld.c that hits ALAT clarification (Section 4.4.5.3.1). IA-32 related changes (Section 6.2.5.4, Section 6.2.3, Section 6.2.4, Section 6.2.5.3). Load instructions change (Section 4.4.1).

Date of Revision	Revision Number	Description
		Volume 2: Class pr-writers-int clarification (Table A-5). PAL_MC_DRAIN clarification (Section 4.4.6.1). VHPT walk and forward progress change (Section 4.1.1.2). IA-32 IBR/DBR match clarification (Section 7.1.1). ISR figure changes (pp. 8-5, 8-26, 8-33 and 8-36). PAL_CACHE_FLUSH return argument change – added new status return argument (Section 11.8.3). PAL self-test Control and PAL_A procedure requirement change – added new arguments, figures, requirements (Section 11.2). PAL_CACHE_FLUSH clarifications (Chapter 11). Non-speculative reference clarification (Section 4.4.6). RID and Preferred Page Size usage clarification (Section 4.1). VHPT read atomicity clarification (Section 4.1). IIP and WC flush clarification (Section 4.4.5). Revised RSE and PMC typographical errors (Section 6.4). Revised DV table (Section A.4). Memory attribute transitions – added new requirements (Section 4.4). MCA for WC/UC aliasing change (Section 4.4.1). Bus lock deprecation – changed behavior of DCR ‘lc’ bit (Section 3.3.4.1, Section 10.6.8, Section 11.8.3). PAL_PROC_GET/SET_FEATURES changes – extend calls to allow implementation-specific feature control (Section 11.8.3). Split PAL_A architecture changes (Section 11.1.6). Simple barrier synchronization clarification (Section 13.4.2). Limited speculation clarification – added hardware-generated speculative references (Section 4.4.6). PAL memory accesses and restrictions clarification (Section 11.9). PSP validity on INITs from PAL_MC_ERROR_INFO clarification (Section 11.8.3). Speculation attributes clarification (Section 4.4.6). PAL_A FIT entry, PAL_VM_TR_READ, PSP, PAL_VERSION clarifications (Sections 11.8.3 and 11.3.2.1). TLB searching clarifications (Section 4.1). IA-32 related changes (Section 10.3, Section 10.3.2, Section 10.3.2, Section 10.3.3.1, Section 10.10.1). IPSR.ri and ISR.ei changes (Table 3-2, Section 3.3.5.1, Section 3.3.5.2, Section 5.5, Section 8.3, and Section 2.2).
		Volume 3: IA-32 CPUID clarification (p. 5-71). Revised figures for extract, deposit, and alloc instructions (Section 2.2). RCPPS, RCPSS, RSQRTPS, and RSQRTSS clarification (Section 7.12). IA-32 related changes (Section 5.3). tak, tpa change (Section 2.2).
July 2000	1.1	Volume 1: Processor Serial Number feature removed (Chapter 3). Clarification on exceptions to instruction dependency (Section 3.4.3).

Date of Revision	Revision Number	Description
		Volume 2: Clarifications regarding “reserved” fields in ITIR (Chapter 3). Instruction and Data translation must be enabled for executing IA-32 instructions (Chapters 3,4 and 10). FCR/FDR mappings, and clarification to the value of PSR.ri after an RFI (Chapters 3 and 4). Clarification regarding ordering data dependency. Out-of-order IPI delivery is now allowed (Chapters 4 and 5). Content of EFLAG field changed in IIM (p. 9-24). PAL_CHECK and PAL_INIT calls – exit state changes (Chapter 11). PAL_CHECK processor state parameter changes (Chapter 11). PAL_BUS_GET/SET_FEATURES calls – added two new bits (Chapter 11). PAL_MC_ERROR_INFO call – Changes made to enhance and simplify the call to provide more information regarding machine check (Chapter 11). PAL_ENTER_IA_32_Env call changes – entry parameter represents the entry order; SAL needs to initialize all the IA-32 registers properly before making this call (Chapter 11). PAL_CACHE_FLUSH – added a new cache_type argument (Chapter 11). PAL_SHUTDOWN – removed from list of PAL calls (Chapter 11). Clarified memory ordering changes (Chapter 13). Clarification in dependence violation table (Appendix A).
		Volume 3: fmix instruction page figures corrected (Chapter 2). Clarification of “reserved” fields in ITIR (Chapters 2 and 3). Modified conditions for alloc/loadrs/flushrs instruction placement in bundle/ instruction group (Chapters 2 and 4). IA-32 JMPE instruction page typo fix (p. 5-238). Processor Serial Number feature removed (Chapter 5).
January 2000	1.0	Initial release of document.

§

Base IA-32 Instruction Reference 2

This section lists all IA-32 instructions and their behavior in the Itanium System Environment and IA-32 System Environments on an processor based on the Itanium architecture. Unless noted otherwise all IA-32 and MMX technology and SSE instructions operate as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

This volume describes the complete IA-32 Architecture instruction set, including the integer, floating-point, MMX technology and SSE technology, and system instructions. The instruction descriptions are arranged in alphabetical order. For each instruction, the forms are given for each operand combination, including the opcode, operands required, and a description. Also given for each instruction are a description of the instruction and its operands, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of the exceptions that can be generated.

For all IA-32 the following relationships hold:

- **Writes** – Writes of any IA-32 general purpose, floating-point or SSE, MMX technology registers by IA-32 instructions are reflected in the Itanium registers defined to hold that IA-32 state when IA-32 instruction set completes execution.
- **Reads** – Reads of any IA-32 general purpose, floating-point or SSE, MMX technology registers by IA-32 instructions see the state of the Itanium registers defined to hold the IA-32 state after entering the IA-32 instruction set.
- **State mappings** – IA-32 numeric instructions are controlled by and reflect their status in FCW, FSW, FTW, FCS, FIP, FOP, FDS and FEA. On exit from the IA-32 instruction set, Itanium numeric status and control resources defined to hold IA-32 state reflect the results of all IA-32 prior numeric instructions in FCR, FSR, FIR and FDR. Itanium numeric status and control resources defined to hold IA-32 state are honored by IA-32 numeric instructions when entering the IA-32 instruction set.

2.1 Additional Intel® Itanium® Faults

The following fault behavior is defined for all IA-32 instructions in the Itanium System Environment:

- **IA-32 Faults** – All IA-32 faults are performed as defined in the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***, unless otherwise noted. IA-32 faults are delivered on the IA_32_Exception interruption vector.
- **IA-32 GPFault** – Null segments are signified by the segment descriptor register's P-bit being set to zero. IA-32 memory references through DSD, ESD, FSD, and GSD with the P-bit set to zero result in an IA-32 GPFault.
- **Itanium Low FP Reg Fault** – If PSR.dfl is 1, execution of any IA-32 MMX technology, SSE or floating-point instructions results in a Disabled FP Register fault (regardless of whether FR2-31 is referenced).
- **Itanium High FP Reg Fault** – If PSR.dfh is 1, execution of the first target IA-32 instruction following an `br.ia` or `rfi` results in a Disabled FP Register fault (regardless of whether FR32-127 is referenced).

- **Itanium Instruction Mem Faults** – The following additional Itanium memory faults can be generated on each virtual page referenced when fetching IA-32 or MMX technology or SSE instructions for execution:
    - Alternative instruction TLB fault
    - VHPT instruction fault
    - Instruction TLB fault
    - Instruction Page Not Present fault
    - Instruction NaT Page Consumption Abort
    - Instruction Key Miss fault
    - Instruction Key Permission fault
    - Instruction Access Rights fault
    - Instruction Access Bit fault
- **Itanium Data Mem Faults** – The following additional Itanium memory faults can be generated on each virtual page touched when reading or writing memory operands from the IA-32 instruction set including MMX technology and SSE instructions:
    - Nested TLB fault
    - Alternative data TLB fault
    - VHPT data fault
    - Data TLB fault
    - Data Page Not Present fault
    - Data NaT Page Consumption Abort
    - Data Key Miss fault
    - Data Key Permission fault
    - Data Access Rights fault
    - Data Dirty bit fault
    - Data Access bit fault

2.2 Interpreting the IA-32 Instruction Reference Pages

This section describes the information contained in the various sections of the instruction reference pages that make up the majority of this chapter. It also explains the notational conventions and abbreviations used in these sections.

2.2.1 IA-32 Instruction Format

The following is an example of the format used for each Intel architecture instruction description in this chapter.

2.2.1.0.0.1 CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

2.2.1.1 Opcode Column

The “Opcode” column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **/digit** – A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r** – Indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.
- **cb, cw, cd, cp** – A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id** – A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.
- **+rb, +rw, +rd** – A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The register codes are given in Table 2-1.
- **+i** – A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 2-1. Register Encodings Associated with the +rb, +rw, and +rd Nomenclature

	rb			rw			rd	
AL	=	0	AX	=	0	EAX	=	0
CL	=	1	CX	=	1	ECX	=	1
DL	=	2	DX	=	2	EDX	=	2
BL	=	3	BX	=	3	EBX	=	3
	rb			rw			rd	
AH	=	4	SP	=	4	ESP	=	4
CH	=	5	BP	=	5	EBP	=	5
DH	=	6	SI	=	6	ESI	=	6
BH	=	7	DI	=	7	EDI	=	7

2.2.1.2 Instruction Column

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** – A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16 and rel32** – A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.

- **ptr16:16 and ptr16:32** – A far pointer, typically in a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** – One of the byte general-purpose registers AL, CL, DL, BL, AH, CH, DH, or BH.
- **r16** – One of the word general-purpose registers AX, CX, DX, BX, SP, BP, SI, or DI.
- **r32** – One of the doubleword general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
- **imm8** – An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** – An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** – An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and -2,147,483,648 inclusive.
- **r/m8** – A byte operand that is either the contents of a byte general-purpose register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
- **r/m16** – A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, BX, CX, DX, SP, BP, SI, and DI. The contents of memory are found at the address provided by the effective address computation.
- **r/m32** – A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The contents of memory are found at the address provided by the effective address computation.
- **m** – A 16- or 32-bit operand in memory.
- **m8** – A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions and the XLAT instruction.
- **m16** – A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** – A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** – A memory quadword operand in memory. This nomenclature is used only with the CMPXCHG8B instruction.
- **m16:16, m16:32** – A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32** – A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All

memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers.

- **moffs8, moffs16, moffs32** – A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** – A segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.
- **m32real, m64real, m80real** – A single-, double-, and extended-real (respectively) floating-point operand in memory.
- **m16int, m32int, m64int** – A word-, short-, and long-integer (respectively) floating-point operand in memory.
- **ST or ST(0)** – The top element of the FPU register stack.
- **ST(i)** – The i^{th} element from the top of the FPU register stack. (i = 0 through 7).
- **mm** – An MMX technology register. The 64-bit MMX technology registers are: MM0 through MM7.
- **mm/m32** – The low order 32 bits of an MMX technology register or a 32-bit memory operand. The 64-bit MMX technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** – An MMX technology register or a 64-bit memory operand. The 64-bit MMX technology registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

2.2.1.3 Description Column

The "Description" column following the "Instruction" column briefly explains the various forms of the instruction. The following "Description" and "Operation" sections contain more details of the instruction's operation.

2.2.1.4 Description

The "Description" section describes the purpose of the instructions and the required operands. It also discusses the effect of the instruction on flags.

2.2.2 Operation

The "Operation" section contains an algorithmic description (written in pseudo-code) of the instruction. The pseudo-code uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as IF, THEN, ELSE, and FI for an if statement, DO and OD for a do statement, or CASE... OF and ESAC for a case statement.

- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that an offset is read from the SI register if the current address-size attribute is 16 or is read from the ESI register if the address-size attribute is 32.
- Brackets are also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.
- A ← B; indicates that the value of B is assigned to A.
- The symbols =, ≠, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "<< COUNT" and ">> COUNT" indicates that the destination operand should be shifted left or right, respectively, by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** – The OperandSize identifier represents the operand-size attribute of the instruction, which is either 16 or 32 bits. The AddressSize identifier represents the address-size attribute, which is either 16 or 32 bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the CMPS instruction used.

```
IF instruction = CMPSW
    THEN OperandSize ← 16;
    ELSE
        IF instruction = CMPSD
            THEN OperandSize ← 32;
        FI;
FI;
```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 1*, for general guidelines on how these attributes are determined.
- **StackAddrSize** – Represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits (see "Address-Size Attribute for Stack" in Chapter 4 of the *Intel Architecture Software Developer's Manual, Volume 1*).
- **SRC** – Represents the source operand.
- **DEST** – Represents the destination operand.

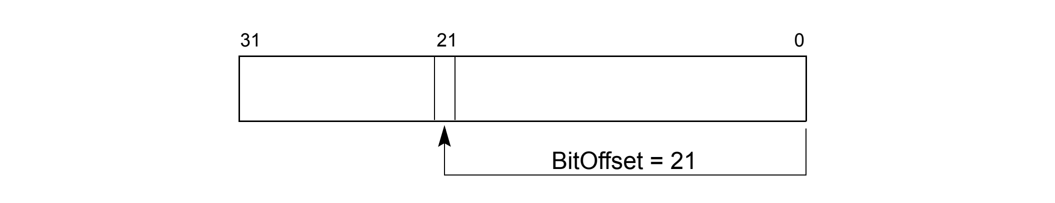
The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** – Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of -10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)** – Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value -10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** – Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than -128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** – Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than -32768, it is represented by the saturated value
  -32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** – Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** – Represents the result of an operation as a signed 8-bit value. If the result is less than -128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** – Represents the result of an operation as a signed 16-bit value. If the result is less than -32768, it is represented by the saturated value -32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** – Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** – Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** – Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** – Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** – Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction.
- **Pop()** – Removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute.
- **PopRegisterStack** – Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** – Performs a task switch.
- **Bit(BitBase, BitOffset)** – Returns the value of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to

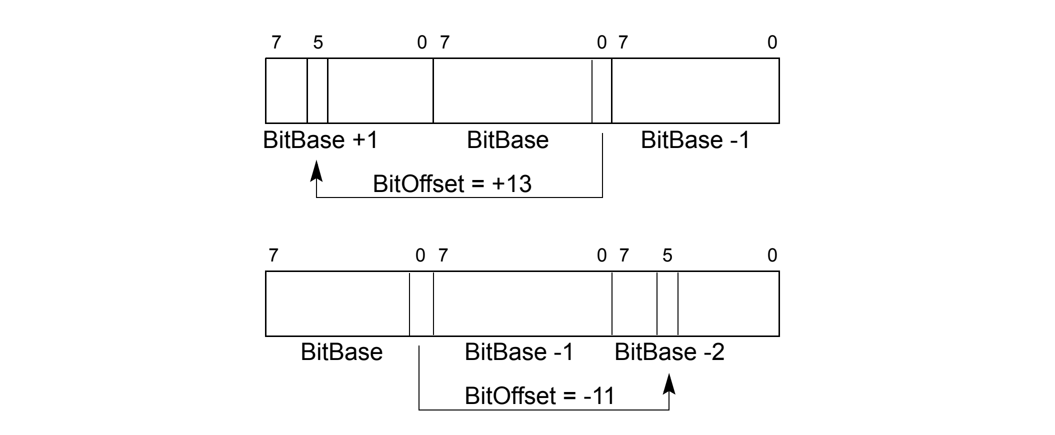
high-order within registers and within memory bytes. If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, the function Bit[EAX, 21] is illustrated in Figure 2-2.

Figure 2-2. Bit Offset for BIT[EAX,21]



If BitBase is a memory address, BitOffset can range from -2 GBits to 2 GBits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number. This operation is illustrated in Figure 2-3.

Figure 2-3. Memory Bit Indexing



2.2.3 Flags Affected

The "Flags Affected" section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, *EFLAGS Cross-Reference*, in the *Intel Architecture Software Developer's Manual, Volume 1*). Non-conventional assignments are described in the "Operation" section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

2.2.4 FPU Flags Affected

The floating-point instructions have an "FPU Flags Affected" section that describes how each instruction can affect the four condition code flags of the FPU status word.

2.2.5 Protected Mode Exceptions

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 2-2 associates each two-letter mnemonic with the corresponding interrupt vector number and exception name. See Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

2.2.6 Real-address Mode Exceptions

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode.

Table 2-2. Exception Mnemonics, Names, and Vector Numbers

Vector No.	Mnemonic	Name	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	UD2 instruction or reserved opcode.[a]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
16	#MF	Floating-point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[b]
18	#MC	Machine Check	Model dependent.[c]

a. The UD2 instruction was introduced in the Pentium® Pro processor.
b. This exception was introduced in the Intel® 486 processor.
c. This exception was introduced in the Pentium processor and enhanced in the Pentium Pro processor.

2.2.7 Virtual-8086 Mode Exceptions

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode.

2.2.8 Floating-point Exceptions

The "Floating-point Exceptions" section lists additional exceptions that can occur when a floating-point instruction is executed in any mode. All of these exception conditions result in a floating-point error exception (#MF, vector number 16) being generated. Table 2-3 associates each one- or two-letter mnemonic with the corresponding exception name. See "Floating-Point Exception Conditions" in Chapter 7 of the *Intel Architecture Software Developer's Manual, Volume 1*, for a detailed description of these exceptions.

Table 2-3. Floating-point Exception Mnemonics and Names

Vector No.	Mnemonic	Name	Source
16	 #IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - FPU stack overflow or underflow - Invalid FPU arithmetic operation
16	#Z	Floating-point divide-by-zero	FPU divide-by-zero
16	#D	Floating-point denormalized operation	Attempting to operate on a denormal number
16	#O	Floating-point numeric overflow	FPU numeric overflow
16	#U	Floating-point numeric underflow	FPU numeric underflow
16	#P	Floating-point inexact result (precision)	Inexact result (precision)

2.3 IA-32 Base Instruction Reference

The remainder of this chapter provides detailed descriptions of each of the Intel architecture instructions.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Description
37	AAA	ASCII adjust AL after addition

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register is incremented by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are cleared to 0.

Operation

```
IF ((AL AND FH) > 9) OR (AF = 1)
  THEN
      AL ← (AL + 6);
      AH ← AH + 1;
      AF ← 1;
      CF ← 1;
  ELSE
      AF ← 0;
      CF ← 0;
FI;
AL ←  AL AND FH;
```

Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Description
D5 0A	AAD	ASCII adjust AX before division

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AL register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to (AL + (10 * AH)), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit number in registers AH and AL.

Operation

```
tempAL ← AL;
tempAH ← AH;
AL ← (tempAL + (tempAH ∗ imm8)) AND FFH;
AH ← 0
```

The immediate value (*imm8*) is taken from the second byte of the instruction, which under normal assembly is 0AH (10 decimal). However, this immediate value can be changed to produce a different result.

Flags Affected

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Description
D4 0A	AAM	ASCII adjust AX after multiply

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked BCD result.

Operation

```
tempAL ← AL;
AH ← tempAL / imm8;
AL ← tempAL MOD imm8;
```

The immediate value (*imm8*) is taken from the second byte of the instruction, which under normal assembly is 0AH (10 decimal). However, this immediate value can be changed to produce a different result.

Flags Affected

The SF, ZF, and PF flags are set according to the result. The OF, AF, and CF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Description
3F	AAS	ASCII adjust AL after subtraction

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register is decremented by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0.

Operation

```
IF ((AL AND FH) > 9) OR (AF = 1)
THEN
   AL ← AL - 6;
   AH ← AH - 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI;
AL ← AL AND FH;
```

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

ADC—Add with Carry

Opcode	Instruction	Description
14 *ib*	ADC AL,*imm8*	Add with carry *imm8* to AL
15 *iw*	ADC AX,*imm16*	Add with carry *imm16* to AX
15 *id*	ADC EAX,*imm32*	Add with carry *imm32* to EAX
80 /2 *ib*	ADC *r/m8,imm8*	Add with carry *imm8* to *r/m8*
81 /2 *iw*	ADC *r/m16,imm16*	Add with carry *imm16* to *r/m16*
81 /2 *id*	ADC *r/m32,imm32*	Add with CF *imm32* to *r/m32*
83 /2 *ib*	ADC *r/m16,imm8*	Add with CF sign-extended *imm8* to *r/m16*
83 /2 *ib*	ADC *r/m32,imm8*	Add with CF sign-extended *imm8* into *r/m32*
10 */r*	ADC *r/m8,r8*	Add with carry byte register to *r/m8*
11 */r*	ADC *r/m16,r16*	Add with carry *r16* to *r/m16*
11 */r*	ADC *r/m32,r32*	Add with CF *r32* to *r/m32*
12 */r*	ADC *r8,r/m8*	Add with carry *r/m8* to byte register
13 */r*	ADC *r16,r/m16*	Add with carry *r/m16* to *r16*
13 */r*	ADC *r32,r/m32*	Add with CF *r/m32* to *r32*

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

Operation

```
DEST ← DEST + SRC + CF;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ADC—Add with Carry (Continued)

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

ADD—Add

Opcode	Instruction	Description
04 *ib*	ADD AL,*imm8*	Add *imm8* to AL
05 *iw*	ADD AX,*imm16*	Add *imm16* to AX
05 *id*	ADD EAX,*imm32*	Add *imm32* to EAX
80 /0 *ib*	ADD *r/m8,imm8*	Add *imm8* to *r/m8*
81 /0 *iw*	ADD *r/m16,imm16*	Add *imm16* to *r/m16*
81 /0 *id*	ADD *r/m32,imm32*	Add *imm32* to *r/m32*
83 /0 *ib*	ADD *r/m16,imm8*	Add sign-extended *imm8* to *r/m16*
83 /0 *ib*	ADD *r/m32,imm8*	Add sign-extended *imm8* to *r/m32*
00 /*r*	ADD *r/m8,r8*	Add *r8* to *r/m8*
01 /*r*	ADD *r/m16,r16*	Add *r16* to *r/m16*
01 /*r*	ADD *r/m32,r32*	Add r32 to *r/m32*
02 /*r*	ADD *r8,r/m8*	Add *r/m8* to *r8*
03 /*r*	ADD *r16,r/m16*	Add *r/m16* to *r16*
03 /*r*	ADD *r32,r/m32*	Add *r/m32* to *r32*

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Operation

```
DEST ← DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ADD—Add (Continued)

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0)If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

AND—Logical AND

Opcode	Instruction	Description
24 *ib*	AND AL,*imm8*	AL AND *imm8*
25 *iw*	AND AX,*imm16*	AX AND i*mm16*
25 *id*	AND EAX,*imm32*	EAX AND *imm32*
80 /4 *ib*	AND *r/m8,imm8*	*r/m8* AND *imm8*
81 /4 *iw*	AND *r/m16,imm16*	*r/m16* AND *imm16*
81 /4 *id*	AND *r/m32,imm32*	*r/m32* AND *imm32*
83 /4 *ib*	AND *r/m16,imm8*	*r/m16* AND *imm8*
83 /4 *ib*	AND *r/m32,imm8*	*r/m32* AND *imm8*
20 */r*	AND *r/m8,r8*	*r/m8* AND *r8*
21 */r*	AND *r/m16,r16*	*r/m16* AND *r16*
21 */r*	AND *r/m32,r32*	*r/m32* AND *r32*
22 */r*	AND *r8,r/m8*	*r8* AND *r/m8*
23 */r*	AND *r16,r/m16*	*r16* AND *r/m16*
23 */r*	AND *r32,r/m32*	*r32* AND *r/m32*

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

```
DEST ← DEST AND SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Description
63 /*r*	ARPL *r/m16,r16*	Adjust RPL of *r/m16* to not less than RPL of *r16*

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program. (The segment selector for the application program's code segment can be read from the procedure stack following a procedure call.)

See the *Intel Architecture Software Developer's Manual, Volume 3* for more information about the use of this instruction.

Operation

```
IF DEST(RPL) < SRC(RPL)
THEN
   ZF ← 1;
   DEST(RPL) ← SRC(RPL);
ELSE
   ZF ← 0;
FI;
```

Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, is cleared to 0.

ARPL—Adjust RPL Field of Segment Selector (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#UD The ARPL instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD The ARPL instruction is not recognized in virtual 8086 mode.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Description
62 /r	BOUND *r16,m16&16*	Check if *r16* (array index) is within bounds specified by *m16&16*
62 /r	BOUND *r32,m32&32*	Check if *r32* (array index) is within bounds specified by *m16&16*

Description

Determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that points to a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. (When a this exception is generated, the saved return instruction pointer points to the BOUND instruction.)

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

Operation

```
IF (ArrayIndex < LowerBound OR ArrayIndex > (UppderBound + OperandSize/8]))
   (* Below lower bound or above upper bound *)
   THEN
         #BR;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BOUND—Check Array Index Against Bounds (Continued)

Protected Mode Exceptions

#BR If the bounds test fails.

#UD If second operand is not a memory location.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#BR If the bounds test fails.

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#BR If the bounds test fails.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

BSF—Bit Scan Forward

Opcode	Instruction	Description
0F BC	BSF *r16,r/m16*	Bit scan forward on *r/m16*
0F BC	BSF *r32,r/m32*	Bit scan forward on *r/m32*

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Operation

```
IF SRC = 0
  THEN
        ZF ← 1;
        DEST is undefined;
  ELSE
        ZF ← 0;
        temp ← 0;
  WHILE Bit(SRC, temp) = 0
  DO
        temp ← temp + 1;
        DEST ← temp;
  OD;
FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BSF—Bit Scan Forward (Continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

BSR—Bit Scan Reverse

Opcode	Instruction	Description
0F BD	BSR *r16,r/m16*	Bit scan reverse on *r/m16*
0F BD	BSR *r32,r/m32*	Bit scan reverse on *r/m32*

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Operation

```
IF SRC = 0
  THEN
      ZF ← 1;
      DEST is undefined;
  ELSE
      ZF ← 0;
      temp ← OperandSize - 1;
  WHILE Bit(SRC, temp) = 0
  DO
      temp ← temp – 1;
      DEST ← temp;
  OD;
FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BSR—Bit Scan Reverse (Continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

BSWAP—Byte Swap

Opcode	Instruction	Description
0F C8+*rd*	BSWAP *r32*	Reverses the byte order of a 32-bit register.

Description

Reverses the byte order of a 32-bit (destination) register: bits 0 through 7 are swapped with bits 24 through 31, and bits 8 through 15 are swapped with bits 16 through 23. This instruction is provided for converting little-endian values to big-endian format and vice versa.

To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

Operation

```
TEMP ← DEST
DEST(7..0) ← TEMP(31..24)
DEST(15..8) ← TEMP(23..16)
DEST(23..16) ← TEMP(15..8)
DEST(31..24) ← TEMP(7..0)
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

Intel Architecture Compatibility Information

The BSWAP instruction is not supported on Intel architecture processors earlier than the Intel486™ processor family. For compatibility with this instruction, include functionally-equivalent code for execution on Intel processors earlier than the Intel486 processor family.

BT—Bit Test

Opcode	Instruction	Description
0F A3	BT *r/m16,r16*	Store selected bit in CF flag
0F A3	BT *r/m32,r32*	Store selected bit in CF flag
0F BA /4 *ib*	BT *r/m16,imm8*	Store selected bit in CF flag
0F BA /4 *ib*	BT *r/m32,imm8*	Store selected bit in CF flag

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively. If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

```
Effective Address + (4 ∗ (BitOffset DIV 32))
```

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

```
Effective Address + (2 ∗ (BitOffset DIV 16))
```

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

Operation

```
CF ← Bit(BitBase, BitOffset)
```

Flags Affected

The CF flag contains the value of the selected bit. The OF, SF, ZF, AF, and PF flags are undefined.

BT—Bit Test (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

BTC—Bit Test and Complement

Opcode	Instruction	Description
0F BB	BTC *r/m16,r16*	Store selected bit in CF flag and complement
0F BB	BTC *r/m32,r32*	Store selected bit in CF flag and complement
0F BA /7 *ib*	BTC *r/m16,imm8*	Store selected bit in CF flag and complement
0F BA /7 *ib*	BTC *r/m32,imm8*	Store selected bit in CF flag and complement

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively. If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 4:40 for more information on this addressing mechanism.

Operation

```
CF ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The OF, SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTC—Bit Test and Complement (Continued)

Protected Mode Exceptions

#GP(0) If the destination operand points to a non-writable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

BTR—Bit Test and Reset

Opcode	Instruction	Description
0F B3	BTR *r/m16,r16*	Store selected bit in CF flag and clear
0F B3	BTR *r/m32,r32*	Store selected bit in CF flag and clear
0F BA /6 *ib*	BTR *r/m16,imm8*	Store selected bit in CF flag and clear
0F BA /6 *ib*	BTR *r/m32,imm8*	Store selected bit in CF flag and clear

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively. If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 4:40 for more information on this addressing mechanism.

Operation

```
CF ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← 0;
```

Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The OF, SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTR—Bit Test and Reset (Continued)

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

BTS—Bit Test and Set

Opcode	Instruction	Description
0F AB	BTS *r/m16,r16*	Store selected bit in CF flag and set
0F AB	BTS *r/m32,r32*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m16,imm8*	Store selected bit in CF flag and set
0F BA /5 *ib*	BTS *r/m32,imm8*	Store selected bit in CF flag and set

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively. If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The offset operand then selects a bit position within the range -2^{31} to $2^{31} - 1$ for a register offset and 0 to 31 for an immediate offset.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" on page 4:40 for more information on this addressing mechanism.

Operation

```
CF ← Bit(BitBase, BitOffset)
Bit(BitBase, BitOffset) ← 1;
```

Flags Affected

The CF flag contains the value of the selected bit before it is set. The OF, SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

BTS—Bit Test and Set (Continued)

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

CALL—Call Procedure

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, *r/m16* indirect
FF /2	CALL *r/m32*	Call near, *r/m32* indirect
9A *cd*	CALL *ptr16:16*	Call far, to full pointer given
9A *cp*	CALL *ptr16:32*	Call far, to full pointer given
FF /3	CALL *m16:16*	Call far, address at *r/m16*
FF /3	CALL *m16:32*	Call far, address at *r/m32*

Description

Saves procedure linking information on the procedure stack and jumps to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call – A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call – A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call – A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure. **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**
- Task switch – A call to a procedure located in a different task. **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See Chapter 6 in the *Intel Architecture Software Developer's Manual, Volume 3* for information on task switching with the CALL instruction.

When executing a near call, the processor pushes the value of the EIP register (which contains the address of the instruction following the CALL instruction) onto the procedure stack (for use later as a return-instruction pointer. The processor then jumps to the address specified with the target operand for the called procedure. The target operand specifies either an absolute address in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the call). An absolute address is specified directly in a register or indirectly in a memory location (*r/m16* or *r/m32* target-operand form). (When accessing an absolute address indirectly using the stack pointer (ESP) as a base register, the base value used is the value of the ESP before the instruction executes.) A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value, which is added to the instruction pointer.

CALL—Call Procedure (Continued)

When executing a near call, the operand-size attribute determines the size of the target operand (16 or 32 bits) for absolute addresses. Absolute addresses are loaded directly into the EIP register. When a relative offset is specified, it is added to the value of the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. The CS register is not changed on near calls.

When executing a far call, the processor pushes the current value of both the CS and EIP registers onto the procedure stack for use as a return-instruction pointer. The processor then performs a far jump to the code segment and address specified with the target operand for the called procedure. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Any far call from a 32-bit code segment to a 16-bit code segment should be made from the first 64 Kbytes of the 32-bit code segment, because the operand-size attribute of the instruction is set to 16, allowing only a 16-bit return address offset to be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack.

When the processor is operating in protected mode, a far call can also be used to access a code segment at a different privilege level or to switch tasks. Here, the processor uses the segment selector part of the far address to access the segment descriptor for the segment being jumped to. Depending on the value of the type and access rights information in the segment selector, the CALL instruction can perform:

- A far call to the same privilege level (described in the previous paragraph).
- An far call to a different privilege level. **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**
- A task switch. **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**

When executing an inter-privilege-level far call, the code segment for the procedure being called is accessed through a call gate. The segment selector specified by the target operand identifies the call gate. In executing a call through a call gate where a change of privilege level occurs, the processor switches to the stack for the privilege level of the called procedure, pushes the current values of the CS and EIP registers and the SS and ESP values for the old stack onto the new stack, then performs a far jump to the new code segment. The new code segment is specified in the call gate descriptor; the new stack segment is specified in the TSS for the currently running task. The jump to the new code segment occurs after the stack switch. On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, a set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.)

Finally, the processor jumps to the address of the procedure being called within the new code segment. The procedure address is the offset specified by the target operand. Here again, the target operand can specify the far address of the call gate and procedure either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

CALL—Call Procedure (Continued)

Executing a task switch with the CALL instruction, is similar to executing a call through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to and the address of the procedure being called in the task. The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The CALL instruction can also specify the segment selector of the TSS directly. See the *Intel Architecture Software Developer's Manual, Volume 3* the for detailed information on the mechanics of a task switch.

Operation

```
IF near call
  THEN IF near relative call
      IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
      THEN IF OperandSize = 32
          THEN
              IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
              Push(EIP);
              EIP ← EIP + DEST; (* DEST is rel32 *)
          ELSE (* OperandSize = 16 *)
              IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
              Push(IP);
              EIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
      FI;
  FI;
  ELSE (* near absolute call *)
      IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
      IF OperandSize = 32
          THEN
              IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
              Push(EIP);
              EIP ← DEST; (* DEST is r/m32 *)
          ELSE (* OperandSize = 16 *)
              IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
              Push(IP);
              EIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
      FI;
  FI:
  IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;
IF far call AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real address or virtual 8086 mode *)
  THEN
      IF OperandSize = 32
          THEN
              IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
              IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
              Push(CS); (* padded with 16 high-order bits *)
              Push(EIP);
              CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
              EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
          ELSE (* OperandSize = 16 *)
              IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
              IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
              Push(CS);
```

CALL—Call Procedure (Continued)

```
                Push(IP);
                CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
        IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;

IF far call AND (PE = 1 AND VM = 0) (* Protected mode, not virtual 8086 mode *)
  THEN
        IF segment selector in target operand null THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code selector);
        FI;
        Read type and access rights of selected segment descriptor;
        IF segment type is not a conforming or nonconforming code segment, call gate,
            task gate, or TSS THEN #GP(segment selector); FI;
        Depending on type and access rights
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
  IF DPL > CPL THEN #GP(new code segment selector); FI;
  IF not present THEN #NP(selector); FI;
  IF OperandSize = 32
        THEN
            IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset);
        ELSE (* OperandSize = 16 *)
            IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS);
            Push(IP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset) AND 0000FFFFH; (* clear upper 16 bits *)
  FI;
  IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
END;

NONCONFORMING-CODE-SEGMENT:
  IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(new code segment selector); FI;
```

CALL—Call Procedure (Continued)

```
    IF stack not large enough for return address THEN #SS(0); FI;
    tempEIP ← DEST(offset)
    IF OperandSize=16
        THEN
            tempEIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
    FI;
    IF tempEIP outside code segment limit THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE (* OperandSize = 16 *)
            Push(CS);
            Push(IP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
    FI;
    IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
END;

CALL-GATE:
    IF call gate DPL < CPL or RPL THEN #GP(call gate selector); FI;
    IF not present THEN #NP(call gate selector); FI;
    IF Itanium System Environment THEN IA-32_Intercept(Gate,CALL);
    IF call gate code-segment selector is null THEN #GP(0); FI;
    IF call gate code-segment selector index is outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    OR code-segment segment descriptor DPL > CPL
        THEN #GP(code segment selector); FI;
    IF code segment not present THEN #NP(new code segment selector); FI;
    IF code segment is non-conforming AND DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← new code segment (DPL ∗ 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            newSS ← TSSstackAddress + 4;
            newESP ← stack address;
        ELSE (* TSS is 16-bit *)
```

CALL—Call Procedure (Continued)

```
                TSSstackAddress ← new code segment (DPL ∗ 4) + 2
                IF (TSSstackAddress + 4) > TSS limit
                    THEN #TS(current TSS selector); FI;
                newESP ← TSSstackAddress;
                newSS ← TSSstackAddress + 2;
        FI;
        IF stack segment selector is null THEN #TS(stack segment selector); FI;
        IF stack segment selector index is not within its descriptor table limits
            THEN #TS(SS selector); FI
        Read code segment descriptor;
        IF stack segment selector's RPL ≠ DPL of code segment
            OR stack segment DPL ≠ DPL of code segment
            OR stack segment is not a writable data segment
                THEN #TS(SS selector); FI
        IF stack segment not present THEN #SS(SS selector); FI;
        IF CallGateSize = 32
            THEN
                IF stack does not have room for parameters plus 16 bytes
                    THEN #SS(SS selector); FI;
                IF CallGate(InstructionPointer) not within code segment limit THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:EIP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure’s stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
            ELSE (* CallGateSize = 16 *)
                IF stack does not have room for parameters plus 8 bytes
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not within code segment limit
                    THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:IP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure’s stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
        FI;
        CPL ← CodeSegment(DPL)
        CS(RPL) ← CPL
    END;

    SAME-PRIVILEGE:
        IF CallGateSize = 32
            THEN
                IF stack does not have room for 8 bytes
                    THEN #SS(0); FI;
```

CALL—Call Procedure (Continued)

```
            IF EIP not within code segment limit then #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* return address to calling procedure *)
        ELSE (* CallGateSize = 16 *)
            IF stack does not have room for parameters plus 4 bytes
                THEN #SS(0); FI;
            IF IP not within code segment limit THEN #GP(0); FI;
            CS:IP ← CallGate(CS:instruction pointer)
            (* segment descriptor information also loaded *)
            Push(oldCS:oldIP); (* return address to calling procedure *)
    FI;
    CS(RPL) ← CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector);
    FI;
    IF task gate not present
        THEN #NP(task gate selector);
    FI;
    IF Itanium System Environment THEN IA-32_Intercept(Gate,CALL);
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS descriptor in GDT;

    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    ORTSS segment selector local/global bit is set to local
    OR TSS descriptor indicates TSS not available
        THEN #GP(TSS selector);
    FI;
    IF TSS is not present
        THEN #NP(TSS selector);
    FI;
    IF Itanium System Environment THEN IA-32_Intercept(Gate,CALL);
    SWITCH-TASKS (with nesting) to TSS
    IF EIP not within code segment limit
```

CALL—Call Procedure (Continued)

```
            THEN #GP(0);
    FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Additional Itanium System Environment Exceptions

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Gate Intercept for CALLs through CALL Gates, Task Gates and Task Segments

IA_32_Exception Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0) If target offset in destination operand is beyond the new code segment limit.

If the segment selector in the destination operand is null.

If the code segment selector in the gate is null.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#GP(selector) If code segment or gate or TSS selector index is outside descriptor table limits.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.

If the segment selector from a call gate is beyond the descriptor table limits.

If the DPL for a code-segment obtained from a call gate is greater than the CPL.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

CALL—Call Procedure (Continued)

#SS(0) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.

If a memory operand effective address is outside the SS segment limit.

#SS(selector) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.

If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.

If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.

#NP(selector) If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.

#TS(selector) If the new stack segment selector and ESP are beyond the end of the TSS.

If the new stack segment selector is null.

If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.

If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.

If the new stack segment is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the target offset is beyond the code segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the target offset is beyond the code segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Description
98	CBW	AX ← sign-extend of AL
98	CWDE	EAX ← sign-extend of AX

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

The CBW and CWDE mnemonics reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16 and the CWDE instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CBW is used and to 32 when CWDE is used. Others may treat these mnemonics as synonyms (CBW/CWDE) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

The CWDE instruction is different from the CWD (convert word to double) instruction. The CWD instruction uses the DX:AX register pair as a destination operand; whereas, the CWDE instruction uses the EAX register as a destination.

Operation

```
IF OperandSize = 16 (* instruction = CBW *)
   THEN AX ← SignExtend(AL);
   ELSE (* OperandSize = 32, instruction = CWDE *)
         EAX ← SignExtend(AX);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

CDQ—Convert Double to Quad

See entry for CWD/CDQ — Convert Word to Double/Convert Double to Quad.

CLC—Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear CF flag

Description

Clears the CF flag in the EFLAGS register.

Operation

```
CF ← 0;
```

Flags Affected

The CF flag is cleared to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

CLD—Clear Direction Flag

Opcode	Instruction	Description
FC	CLD	Clear DF flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

Operation

```
DF ← 0;
```

Flags Affected

The DF flag is cleared to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

CLI—Clear Interrupt Flag

Opcode	Instruction	Description
FA	CLI	Clear interrupt flag; interrupts disabled when interrupt flag cleared

Description

Clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts. **In the Itanium System Environment, external interrupts are enabled for IA-32 instructions if PSR.i and (~CFLG.if or EFLAG.if) is 1 and for Itanium instructions if PSR.i is 1.**

The following decision table indicates the action of the CLI instruction (bottom of the table) depending on the processor's mode of operating and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1	1
VM =	X	0	X	0	1
CPL	X	≤ IOPL	X	> IOPL	X
IOPL	X	X	= 3	X	< 3
IF ← 0	Y	Y	Y	N	N
#GP(0)	N	N	N	Y	Y

Notes:
XDon't care.
NAction in column 1 not taken.
YAction in column 1 taken.

Operation

```
OLD_IF <- IF;
IF PE = 0 (* Executing in real-address mode *)
   THEN
         IF ← 0;
   ELSE
         IF VM = 0   (* Executing in protected mode *)
               THEN
                     IF CR4.PVI = 1
                           THEN
                                 IF CPL = 3
                                 THEN
                                       IF IOPL<3
                                       THEN VIF <- 0;
                                       ELSE IF <- 0;
                                       FI;
                                 ELSE (*CPL < 3*)
                                       IF IOPL < CPL
                                       THEN #GP(0);
                                       ELSE IF <- 0;
```

CLI—Clear Interrupt Flag (Continued)

```
                          FI;
                      FI;
                  ELSE (*CR4.PVI==0 *)
                      IF IOPL < CPL
                      THEN #GP(0);
                      ELSE IF <- 0;
                      FI;
              FI;
          ELSE  (* Executing in Virtual-8086 mode  *)
              IF IOPL = 3
                  THEN
                      IF  ← 0;
                  ELSE
                      IF CR4.VME= 0
                      THEN #GP(0);
                      ELSE VIF <- 0;
                      FI;
              FI;
      FI;
FI;
IF Itanium System Environment AND CFLG.ii AND IF != OLD_IF
  THEN IA-32_Intercept(System_Flag,CLI);
```

Flags Affected

The IF is cleared to 0 if the CPL is equal to or less than the IOPL; otherwise, the it is not affected. The other flags in the EFLAGS register are unaffected.

Additional Itanium System Environment Exceptions

IA-32_Intercept System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state.

Protected Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Description
0F 06	CLTS	Clears TS flag in CR0

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the *Intel Architecture Software Developer's Manual, Volume 3* for more information about this flag.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,CLTS);
CR0(TS) ← 0;
```

Flags Affected

The TS flag in CR0 register is cleared.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept fault.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than 0.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If the CPL is greater than 0.

CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement CF flag

Description

Complements the CF flag in the EFLAGS register.

Operation

```
CF ← NOT CF;
```

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

CMOV*cc*—Conditional Move

Opcode	Instruction	Description
0F 47 *cw/cd*	CMOVA *r16, r/m16*	Move if above (CF=0 and ZF=0)
0F 47 *cw/cd*	CMOVA *r32, r/m32*	Move if above (CF=0 and ZF=0)
0F 43 *cw/cd*	CMOVAE *r16, r/m16*	Move if above or equal (CF=0)
0F 43 *cw/cd*	CMOVAE *r32, r/m32*	Move if above or equal (CF=0)
0F 42 *cw/cd*	CMOVB *r16, r/m16*	Move if below (CF=1)
0F 42 *cw/cd*	CMOVB *r32, r/m32*	Move if below (CF=1)
0F 46 *cw/cd*	CMOVBE *r16, r/m16*	Move if below or equal (CF=1 or ZF=1)
0F 46 *cw/cd*	CMOVBE *r32, r/m32*	Move if below or equal (CF=1 or ZF=1)
0F 42 *cw/cd*	CMOVC *r16, r/m16*	Move if carry (CF=1)
0F 42 *cw/cd*	CMOVC *r32, r/m32*	Move if carry (CF=1)
0F 44 *cw/cd*	CMOVE *r16, r/m16*	Move if equal (ZF=1)
0F 44 *cw/cd*	CMOVE *r32, r/m32*	Move if equal (ZF=1)
0F 4F *cw/cd*	CMOVG *r16, r/m16*	Move if greater (ZF=0 and SF=OF)
0F 4F *cw/cd*	CMOVG *r32, r/m32*	Move if greater (ZF=0 and SF=OF)
0F 4D *cw/cd*	CMOVGE *r16, r/m16*	Move if greater or equal (SF=OF)
0F 4D *cw/cd*	CMOVGE *r32, r/m32*	Move if greater or equal (SF=OF)
0F 4C *cw/cd*	CMOVL *r16, r/m16*	Move if less (SF<>OF)
0F 4C *cw/cd*	CMOVL *r32, r/m32*	Move if less (SF<>OF)
0F 4E *cw/cd*	CMOVLE *r16, r/m16*	Move if less or equal (ZF=1 or SF<>OF)
0F 4E *cw/cd*	CMOVLE *r32, r/m32*	Move if less or equal (ZF=1 or SF<>OF)
0F 46 *cw/cd*	CMOVNA *r16, r/m16*	Move if not above (CF=1 or ZF=1)
0F 46 *cw/cd*	CMOVNA *r32, r/m32*	Move if not above (CF=1 or ZF=1)
0F 42 *cw/cd*	CMOVNAE *r16, r/m16*	Move if not above or equal (CF=1)
0F 42 *cw/cd*	CMOVNAE *r32, r/m32*	Move if not above or equal (CF=1)
0F 43 *cw/cd*	CMOVNB *r16, r/m16*	Move if not below (CF=0)
0F 43 *cw/cd*	CMOVNB *r32, r/m32*	Move if not below (CF=0)
0F 47 *cw/cd*	CMOVNBE *r16, r/m16*	Move if not below or equal (CF=0 and ZF=0)
0F 47 *cw/cd*	CMOVNBE *r32, r/m32*	Move if not below or equal (CF=0 and ZF=0)
0F 43 *cw/cd*	CMOVNC *r16, r/m16*	Move if not carry (CF=0)
0F 43 *cw/cd*	CMOVNC *r32, r/m32*	Move if not carry (CF=0)
0F 45 *cw/cd*	CMOVNE *r16, r/m16*	Move if not equal (ZF=0)
0F 45 *cw/cd*	CMOVNE *r32, r/m32*	Move if not equal (ZF=0)
0F 4E *cw/cd*	CMOVNG *r16, r/m16*	Move if not greater (ZF=1 or SF<>OF)
0F 4E *cw/cd*	CMOVNG *r32, r/m32*	Move if not greater (ZF=1 or SF<>OF)
0F 4C *cw/cd*	CMOVNGE *r16, r/m16*	Move if not greater or equal (SF<>OF)
0F 4C *cw/cd*	CMOVNGE *r32, r/m32*	Move if not greater or equal (SF<>OF)
0F 4D *cw/cd*	CMOVNL *r16, r/m16*	Move if not less (SF=OF)
0F 4D *cw/cd*	CMOVNL *r32, r/m32*	Move if not less (SF=OF)
0F 4F *cw/cd*	CMOVNLE *r16, r/m16*	Move if not less or equal (ZF=0 and SF=OF)
0F 4F *cw/cd*	CMOVNLE *r32, r/m32*	Move if not less or equal (ZF=0 and SF=OF)

CMOV*cc*—Conditional Move (Continued)

Opcode	Instruction	Description
0F 41 *cw/cd*	CMOVNO *r16, r/m16*	Move if not overflow (OF=0)
0F 41 *cw/cd*	CMOVNO *r32, r/m32*	Move if not overflow (OF=0)
0F 4B *cw/cd*	CMOVNP *r16, r/m16*	Move if not parity (PF=0)
0F 4B *cw/cd*	CMOVNP *r32, r/m32*	Move if not parity (PF=0)
0F 49 *cw/cd*	CMOVNS *r16, r/m16*	Move if not sign (SF=0)
0F 49 *cw/cd*	CMOVNS *r32, r/m32*	Move if not sign (SF=0)
0F 45 *cw/cd*	CMOVNZ *r16, r/m16*	Move if not zero (ZF=0)
0F 45 *cw/cd*	CMOVNZ *r32, r/m32*	Move if not zero (ZF=0)
0F 40 *cw/cd*	CMOVO *r16, r/m16*	Move if overflow (OF=0)
0F 40 *cw/cd*	CMOVO *r32, r/m32*	Move if overflow (OF=0)
0F 4A *cw/cd*	CMOVP *r16, r/m16*	Move if parity (PF=1)
0F 4A *cw/cd*	CMOVP *r32, r/m32*	Move if parity (PF=1)
0F 4A *cw/cd*	CMOVPE *r16, r/m16*	Move if parity even (PF=1)
0F 4A *cw/cd*	CMOVPE *r32, r/m32*	Move if parity even (PF=1)
0F 4B *cw/cd*	CMOVPO *r16, r/m16*	Move if parity odd (PF=0)
0F 4B *cw/cd*	CMOVPO *r32, r/m32*	Move if parity odd (PF=0)
0F 48 *cw/cd*	CMOVS *r16, r/m16*	Move if sign (SF=1)
0F 48 *cw/cd*	CMOVS *r32, r/m32*	Move if sign (SF=1)
0F 44 *cw/cd*	CMOVZ *r16, r/m16*	Move if zero (ZF=1)
0F 44 *cw/cd*	CMOVZ *r32, r/m32*	Move if zero (ZF=1)

Description

The CMOV*cc* instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV*cc* instruction.

If the condition is false for the memory form, some processor implementations will initiate the load (and discard the loaded data), possible memory faults can be generated. Other processor models will not initiate the load and not generate any faults if the condition is false.

These instructions can move a 16- or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The conditions for each CMOV*cc* mnemonic is given in the description column of the above table. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

CMOV*cc*—Conditional Move (Continued)

The CMOV*cc* instructions are new for the Pentium Pro processor family; however, they may not be supported by all the processors in the family. Software can determine if the CMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" on page 4:78).

Operation

```
temp ← DEST
IF condition TRUE
   THEN
         DEST ← SRC
   ELSE
         DEST ← temp
FI;
```

Flags Affected

None.

If the condition is false for the memory form, some processor implementations will initiate the load (and discard the loaded data), possible memory faults can be generated. Other processor models will not initiate the load and not generate any faults if the condition is false.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

CMOV*cc*—Conditional Move (Continued)

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

CMP—Compare Two Operands

Opcode	Instruction	Description
3C *ib*	CMP AL, *imm8*	Compare *imm8* with AL
3D *iw*	CMP AX, *imm16*	Compare *imm16* with AX
3D *id*	CMP EAX, *imm32*	Compare *imm32* with EAX
80 /7 *ib*	CMP *r/m8, imm8*	Compare *imm8* with *r/m8*
81 /7 *iw*	CMP *r/m16, imm16*	Compare *imm16* with *r/m16*
81 /7 *id*	CMP *r/m32,imm32*	Compare *imm32* with *r/m32*
83 /7 *ib*	CMP *r/m16,imm8*	Compare *imm8* with *r/m16*
83 /7 *ib*	CMP *r/m32,imm8*	Compare *imm8* with *r/m32*
38 */r*	CMP *r/m8,r8*	Compare *r8* with *r/m8*
39 */r*	CMP *r/m16,r16*	Compare *r16* with *r/m16*
39 */r*	CMP *r/m32,r32*	Compare *r32* with *r/m32*
3A */r*	CMP *r8,r/m8*	Compare *r/m8* with *r8*
3B */r*	CMP *r16,r/m16*	Compare *r/m16* with *r16*
3B */r*	CMP *r32,r/m32*	Compare *r/m32* with *r32*

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction.

Operation

```
temp ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

CMP—Compare Two Operands (Continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands

Opcode	Instruction	Description
A6	CMPS DS:(E)SI, ES:(E)DI	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPS DS:SI, ES:DI	Compares byte at address DS:SI with byte at address ES:DI and sets the status flags accordingly
A7	CMPS DS:ESI, ES:EDI	Compares byte at address DS:ESI with byte at address ES:EDI and sets the status flags accordingly
A6	CMPSB	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPSW	Compares byte at address DS:SI with byte at address ES:DI and sets the status flags accordingly
A7	CMPSD	Compares byte at address DS:ESI with byte at address ES:EDI and sets the status flags accordingly

Description

Compares the byte, word, or double word specified with the first source operand with the byte, word, or double word specified with the second source operand and sets the status flags in the EFLAGS register according to the results. The first source operand specifies the memory location at the address DS:ESI and the second source operand specifies the memory location at address ES:EDI. (When the operand-size attribute is 16, the SI and DI register are used as the source-index and destination-index registers, respectively.) The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

The CMPSB, CMPSW, and CMPSD mnemonics are synonyms of the byte, word, and doubleword versions of the CMPS instructions. They are simpler to use, but provide no type or segment checking. (For the CMPS instruction, “DS:ESI” and “ES:EDI” must be explicitly specified in the instruction.)

After the comparison, the ESI and EDI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI and EDI register are incremented; if the DF flag is 1, the ESI and EDI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The CMPS, CMPSB, CMPSW, and CMPSD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made.

CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands (Continued)

Operation

```
temp ←SRC1 – SRC2;
SetStatusFlags(temp);
IF (byte comparison)
   THEN IF DF = 0
         THEN (E)DI ← 1; (E)SI ← 1;
         ELSE (E)DI ← -1; (E)SI ← -1;
   FI;
   ELSE IF (word comparison)
         THEN IF DF = 0
               THEN DI ← 2; (E)SI ← 2;
               ELSE DI ← -2; (E)SI ← -2;
         FI;
         ELSE (* doubleword comparison *)
               THEN IF DF = 0
                     THEN EDI ← 4; (E)SI ← 4;
                     ELSE EDI ← -4; (E)SI ← -4;
               FI;
   FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands (Continued)

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

CMPXCHG—Compare and Exchange

Opcode	Instruction	Description
0F B0/*r*	CMPXCHG *r/m8,r8*	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
0F B1/*r*	CMPXCHG *r/m16,r16*	Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AL
0F B1/*r*	CMPXCHG *r/m32,r32*	Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into AL

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Operation

```
(* accumulator = AL, AX, or EAX, depending on whether *)
(* a byte, word, or doubleword comparison is being performed*)

IF Itanium System Environment AND External_Atomic_Lock_Required AND DCR.lc
  THEN IA-32_Intercept(LOCK,CMPXCHG);
IF accumulator = DEST
  THEN
      ZF ← 1
      DEST ← SRC
  ELSE
      ZF ← 0
      accumulator ← DEST
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

CMPXCHG—Compare and Exchange (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Lock Intercept – If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

CMPXCHG8B—Compare and Exchange 8 Bytes

Opcode	Instruction	Description
0F C7 /1 m64	CMPXCHG8B *m64*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX.

Description

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor’s bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Operation

```
IF Itanium System Environment AND External_Atomic_Lock_Required AND DCR.lc
  THEN IA-32_Intercept(LOCK,CMPXCHG);
IF (EDX:EAX = DEST)
  ZF ← 1
  DEST ← ECX:EBX
ELSE
  ZF ← 0
  EDX:EAX ← DEST
FI;
```

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA-32_Intercept	Lock Intercept – If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction

CMPXCHG8B—Compare and Exchange 8 Bytes (Continued)

Protected Mode Exceptions

#UD If the destination operand is not a memory location.

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Pentium processors.

CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers, according to the input value entered initially in the EAX register.

Description

Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The information returned is selected by entering a value in the EAX register before the instruction is executed. Table 2-4 shows the information returned, depending on the initial value loaded into the EAX register.

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction.

The information returned with the CPUID instruction is divided into two groups: basic information and extended function information. Basic information is returned by entering an input value starting at 0 in the EAX register; extended function information is returned by entering an input value starting at 80000000H. When the input value in the EAX register is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning basic information. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX return value for basic information. When the input value in the EAX register is 80000000H, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning extended function information. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX return value for extended function information.

The CPUID instruction can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

Table 2-4. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0	EAX	Maximum CPUID Input Value
	EBX	756E6547H “Genu” (G in BL)
	ECX	6C65746EH “ntel” (n in CL)
	EDX	49656E69H “inel” (i in DL)
1H	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Bits 7-0: Brand Index[a]
		Bits 15-8: CLFLUSH line size (Value * 8 = cache line size in bytes)
		Bits 23-16: Number of logical processors per physical processor
		Bits 31-24: Local APIC ID[b]
	ECX	Reserved
	EDX	Feature Information (see Table 2-5)
2H	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information

Table 2-4. Information Returned by CPUID Instruction (Continued)

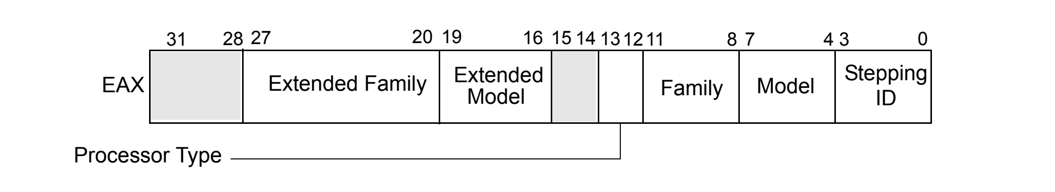
Initial EAX Value	Information Provided about the Processor	
	Extended Function CPUID Information	
8000000H	EAX	Maximum Input Value for Extended Function CPUID Information
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
8000001H	EAX	Extended Processor Signature and Extended Feature Bits. (Currently reserved.)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
8000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
8000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued

a. This field is not supported for processors based on Itanium architecture, zero (unsupported encoding) is returned.

b. This field is invalid for processors based on Itanium architecture, reserved value is returned.

When the input value is 1, the processor returns version information in the EAX register (see Figure 2-4). The version information consists of an Intel architecture family identifier, a model identifier, a stepping ID, and a processor type.

Figure 2-4. Version Information in Registers EAX



If the values in the family and/or model fields reach or exceed FH, the CPUID instruction will generate two additional fields in the EAX register: the extended family field and the extended model field. Here, a value of FH in either the model field or the family field indicates that the extended model or family field, respectively, is valid. Family and model numbers beyond FH range from 0FH to FFH, with the least significant hexadecimal digit always FH.

See AP-485, *Intel® Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on identifying Intel architecture processors.

CPUID—CPU Identification (Continued)

When the input value in EAX is 1, three unrelated pieces of information are returned to the EBX register:

- Brand index (low byte of EBX) – this number provides an entry into a brand string table that contains brand strings for IA-32 processors. Please refer to AP-485, *Intel® Processor Identification and the CPUID Instruction* (Order Number 241618) for information on brand indices.

  Note: The Brand index field is not supported for processors based on Itanium architecture, zero (unsupported encoding) is returned.

- CLFLUSH instruction cache line size (second byte of EBX) – this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field is valid only when the CLFSH feature flag is set.
- Local APIC ID (high byte of EBX) – this number is the 8-bit ID that is assigned to the local APIC on the processor during power up.

  Note: The local APIC ID field is invalid for processors based on the Itanium architecture, reserved value is returned. Software should check the feature flags to make sure they are not running on processors based on the Itanium architecture before interpreting the return value in this field.

When the EAX register contains a value of 1, the CPUID instruction (in addition to loading the processor signature in the EAX register) loads the EDX register with the feature flags. The feature flags (when a Flag = 1) indicate what features the processor supports. Table 2-5 lists the currently defined feature flag values.

A feature flag set to 1 indicates the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags.

Table 2-5. Feature Flags Returned in EDX Register

Bit	Mnemonic	Description
0	FPU	**Floating Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4Mbyte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.

Table 2-5. Feature Flags Returned in EDX Register (Continued)

Bit	Mnemonic	Description
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2 Mbyte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model-specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default – some processors permit the APIC to be relocated).
10	Reserved	Reserved.
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**PTE Global Bit.** The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported.
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
17	PSE-36	**32-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	NX	Execute Disable Bit.
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities.

Table 2-5. Feature Flags Returned in EDX Register (Continued)

Bit	Mnemonic	Description
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Hyper-Threading Technology.** The processor implements Hyper-Threading technology.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Processor based on the Intel Itanium architecture	The processor is based on the Intel Itanium architecture and is capable of executing the Intel Itanium instruction set. IA-32 application level software MUST also check with the running operating system to see if the system can also support Itanium architecture-based code before switching to the Intel Itanium instruction set.
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

When the input value is 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers. The encoding of these registers is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors.

Please see the processor-specific supplement for further information on how to decode the return values for the processors internal caches and TLBs.

CPUID performs instruction serialization and a memory fence operation.

CPUID—CPU Identification (Continued)

Operation

```
CASE (EAX) OF
  EAX = 0H:
      EAX ← Highest input value understood by CPUID;
      EBX ← Vendor identification string;
      EDX ← Vendor identification string;
      ECX ← Vendor identification string;
  BREAK;
  EAX = 1H:
      EAX[3:0] ← Stepping ID;
      EAX[7:4] ← Model;
      EAX[11:8] ← Family;
      EAX[13:12] ← Processor Type;
      EAX[15:14] ← Reserved;
      EAX[19:16] ← Extended Model;
      EAX[27:20] ← Extended Family;
      EAX[31:28] ← Reserved;
      EBX[7:0] ← Brand Index; (* Always zero for processors based on Itanium architecture *)
      EBX[15:8] ← CLFLUSH Line Size;
      EBX[16:23] ← Number of logical processors per physical processor;
      EBX[31:24] ← Initial APIC ID; (* Reserved for processors based on Itanium architecture *)
      ECX ← Reserved;
      EDX ← Feature flags;
  BREAK;
  EAX = 2H:
      EAX ← Cache and TLB information;
      EBX ← Cache and TLB information;
      ECX ← Cache and TLB information;
      EDX ← Cache and TLB information;
  BREAK;
  EAX = 80000000H:
      EAX ← Highest extended function input value understood by CPUID;
      EBX ← Reserved;
      ECX ← Reserved;
      EDX ← Reserved;
  BREAK;
  EAX = 80000001H:
      EAX ← Extended Processor Signature and Feature Bits; (* Currently Reserved *)
      EBX ← Reserved;
      ECX ← Reserved;
      EDX ← Reserved;
  BREAK;
  EAX = 80000002H:
      EAX ← Processor Name;
      EBX ← Processor Name;
      ECX ← Processor Name;
      EDX ← Processor Name;
  BREAK;
  EAX = 80000003H:
      EAX ← Processor Name;
      EBX ← Processor Name;
      ECX ← Processor Name;
      EDX ← Processor Name;
```

CPUID—CPU Identification (Continued)

```
BREAK;
  EAX = 80000004H:
        EAX ← Processor Name;
        EBX ← Processor Name;
        ECX ← Processor Name;
        EDX ← Processor Name;
  BREAK;
  DEFAULT: (* EAX > highest value recognized by CPUID *)
        EAX ← Reserved, Undefined;
        EBX ← Reserved, Undefined;
        ECX ← Reserved, Undefined;
        EDX ← Reserved, Undefined;
  BREAK;
ESAC;

memory_fence();
instruction_serialize();
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults   NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

Intel Architecture Compatibility

The CPUID instruction is not supported in early models of the Intel486 processor or in any Intel architecture processor earlier than the Intel486 processor. The ID flag in the EFLAGS register can be used to determine if this instruction is supported. If a procedure is able to set or clear this flag, the CPUID is supported by the processor running the procedure.

CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Description
99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

Operation

```
IF OperandSize = 16 (* CWD instruction *)
  THEN DX ← SignExtend(AX);
  ELSE (* OperandSize = 32, CDQ instruction *)
      EDX ← SignExtend(EAX);
FI;
```

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Flags Affected

None.

Exceptions (All Operating Modes)

None.

CWDE—Convert Word to Doubleword

See entry for CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword.

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Description
27	DAA	Decimal adjust AL after addition

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

Operation

```
IF (((AL AND 0FH) > 9) or AF = 1)
   THEN
         AL ← AL + 6;
         CF ← CF OR CarryFromLastAddition; (* CF OR carry from AL ← AL + 6 *)
         AF ← 1;
   ELSE
         AF ← 0;
FI;
IF ((AL AND F0H) > 90H) or CF = 1)
   THEN
         AL ←  AL + 60H;
         CF  ← 1;
   ELSE
         CF ← 0;
FI;
```

Example

```
ADD AL, BL       Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
                 After: AL=AEH BL=35H EFLAGS(0SZAPC)=110000
DAA              Before: AL=79H BL=35H EFLAGS(OSZAPC)=110000
                 After: AL=AEH BL=35H EFLAGS(0SZAPC)=X00111
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see “Operation” above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults   NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Description
2F	DAS	Decimal adjust AL after subtraction

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
  THEN
      AL ← AL – 6;
      CF ← CF OR BorrowFromLastSubtraction; (* CF OR borrow from AL ← AL – 6 *)
      AF ← 1;
  ELSE AF ← 0;
FI;
IF ((AL > 9FH) or CF = 1)
  THEN
      AL ← AL – 60H;
      CF ← 1;
  ELSE CF ← 0;
FI;
```

Example

```
SUB AL, BL          Before: AL=35H BL=47H EFLAGS(OSZAPC)=XXXXXX
                    After: AL=EEH BL=47H EFLAGS(0SZAPC)=010111
DAA                 Before: AL=EEH BL=47H EFLAGS(OSZAPC)=010111
                    After: AL=88H BL=47H EFLAGS(0SZAPC)=X10111
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see “Operation” above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

DEC—Decrement by 1

Opcode	Instruction	Description
FE /1	DEC *r/m8*	Decrement *r/m8* by 1
FF /1	DEC *r/m16*	Decrement *r/m16* by 1
FF /1	DEC *r/m32*	Decrement *r/m32* by 1
48+rw	DEC *r16*	Decrement *r16* by 1
48+rd	DEC *r32*	Decrement *r32* by 1

Description

Subtracts 1 from the operand, while preserving the state of the CF flag. The source operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a SUB instruction with an immediate operand of 1 to perform a decrement operation that does updates the CF flag.)

Operation

```
DEST ← DEST - 1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

DEC—Decrement by 1 (Continued)

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV *r/m8*	Unsigned divide AX by *r/m8*; AL ← Quotient, AH ← Remainder
F7 /6	DIV *r/m16*	Unsigned divide DX:AX by *r/m16*; AX ← Quotient, DX ← Remainder

F7 /6	DIV *r/m32*	Unsigned divide EDX:EAX by *r/m32* doubleword;
		EAX ← Quotient, EDX ← Remainder

Description

Divides (unsigned) the value in the AL, AX, or EAX register (dividend) by the source operand (divisor) and stores the result in the AX, DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size, as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

Operation

```
IF SRC = 0
   THEN #DE; (* divide error *)
FI;
IF OpernadSize = 8 (* word/byte operation *)
   THEN
         temp ← AX / SRC;
         IF temp > FFH
               THEN #DE; (* divide error *) ;
               ELSE
                     AL ← temp;
                     AH ← AX MOD SRC;
         FI;
   ELSE
         IF OpernadSize = 16 (* doubleword/word operation *)
               THEN
                     temp ← DX:AX / SRC;
                     IF temp > FFFFH
                           THEN #DE; (* divide error *) ;
                           ELSE
                                 AX ← temp;
                                 DX ← DX:AX MOD SRC;
                     FI;
```

DIV—Unsigned Divide (Continued)

```
          ELSE (* quadword/doubleword operation *)
              temp ← EDX:EAX / SRC;
              IF temp > FFFFFFFFH
                  THEN #DE; (* divide error *) ;
                  ELSE
                      EAX ← temp;
                      EDX ← EDX:EAX MOD SRC;
              FI;
      FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#DE If the source operand (divisor) is 0

If the quotient is too large for the designated register.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#DE If the source operand (divisor) is 0.

If the quotient is too large for the designated register.

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

DIV—Unsigned Divide (Continued)

Virtual 8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Description
C8 *iw* 00	ENTER *imm16*,0	Create a stack frame for a procedure
C8 *iw* 01	ENTER *imm16*,1	Create a nested stack frame for a procedure
C8 *iw* ib	ENTER *imm16,imm8*	Create a nested stack frame for a procedure

Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits) or EBP (32 bits) register specifies the current frame pointer and whether SP (16 bits) or ESP (32 bits) specifies the stack pointer.

The ENTER and companion LEAVE instructions are provided to support block structured languages. They do not provide a jump or call to another procedure; they merely set up a new stack frame for an already called procedure. An ENTER instruction is commonly followed by a CALL, JMP, or J*cc* instruction to transfer program control to the procedure being called.

If the nesting level is 0, the processor pushes the frame pointer from the EBP register onto the stack, copies the current stack pointer from the ESP register into the EBP register, and loads the ESP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack.

Operation

```
NestingLevel ← NestingLevel MOD 32
IF StackSize = 32
  THEN
      Push(EBP) ;
      FrameTemp ← ESP;
  ELSE (* StackSize = 16*)
      Push(BP);
      FrameTemp ← SP;
FI;
IF NestingLevel = 0
  THEN GOTO CONTINUE;
FI;
IF (NestingLevel > 0)
  FOR i ← 1 TO (NestingLevel – 1)
      DO
          IF OperandSize = 32
              THEN
```

ENTER—Make Stack Frame for Procedure Parameters (Continued)

```
                  IF StackSize = 32
                      EBP ← EBP – 4;
                      Push([EBP]); (* doubleword push *)
                  ELSE (* StackSize = 16*)
                      BP ← BP – 4;
                      Push([BP]); (* doubleword push *)
                  FI;
              ELSE (* OperandSize = 16 *)
                  IF StackSize = 32
                      THEN
                          EBP ← EBP – 2;
                          Push([EBP]); (* word push *)
                      ELSE (* StackSize = 16*)
                          BP ← BP – 2;
                          Push([BP]); (* word push *)
                  FI;
          FI;
  OD;
  IF OperandSize = 32
      THEN
          Push(FrameTemp); (* doubleword push *)
      ELSE (* OperandSize = 16 *)
          Push(FrameTemp); (* word push *)
  FI;
  GOTO CONTINUE;
FI;
CONTINUE:
IF StackSize = 32
  THEN
      EBP ← FrameTemp
      ESP ← EBP – Size;
  ELSE (* StackSize = 16*)
      BP ← FrameTemp
      SP ← BP – Size;
FI;
END;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

ENTER—Make Stack Frame for Procedure Parameters (Continued)

Protected Mode Exceptions

#SS(0) If the new value of the SP or ESP register is outside the stack segment limit.

#PF(fault-code) If a page fault occurs.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

None.

F2XM1—Compute 2^x-1

Opcode	Instruction	Description
D9 F0	F2XM1	Replace ST(0) with ($2^{ST(0)}$ - 1)

Description

Calculates the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range -1.0 to +1.0. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs:

ST(0) SRC	ST(0) DEST
-1.0 to −0	−0.5 to −0
−0	−0
+0	+0
+0 to +1.0	+0 to 1.0

Values other than 2 can be exponentiated using the following formula:

$x^y = 2^{(y * \log_2 x)}$

Operation

$ST(0) \leftarrow (2^{ST(0)} - 1);$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Result is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

F2XM1—Compute 2^x-1 (Continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FABS—Absolute Value

Opcode	Instruction	Description
D9 E1	FABS	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

ST(0) SRC	ST(0) DEST
–•	+∞
–F	+F
–0	+0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

Note:
Fmeans finite-real number.

Operation

```
ST(0) ← |ST(0)|
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FADD/FADDP/FIADD—Add

Opcode	Instruction	Description
D8 /0	FADD *m32 real*	Add *m32real* to ST(0) and store result in ST(0)
DC /0	FADD *m64real*	Add *m64real* to ST(0) and store result in ST(0)
D8 C0+i	FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0)
DC C0+i	FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(*i*)
DE C0+i	FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(*i*), and pop the register stack
DE C1	FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack
DA /0	FIADD *m32int*	Add *m32int* to ST(0) and store result in ST(0)
DE /0	FIADD *m16int*	Add *m16int* to ST(0) and store result in ST(0)

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a real or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(*i*) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to extended-real format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated.

FADD/FADDP/FIADD—Add (Continued)

.

		DEST						
		-•	–F	–0	+0	+F	+∞	NaN
	-∞	-∞	-∞	-∞	-∞	-∞	*	NaN
	–F or –I	-∞	–F	SRC	SRC	±F or ±0	+∞	NaN
SRC	–0	-∞	DEST	–0	±0	DEST	+∞	NaN
	+0	-∞	DEST	±0	+0	DEST	+∞	NaN
	+For +I	-∞	±F or ±0	SRC	SRC	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Lmeans integer.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIADD
   THEN
         DEST ← DEST + ConvertExtendedReal(SRC);
   ELSE (* source operand is real number *)
         DEST ← DEST + SRC;
FI;
IF instruction = FADDP
   THEN
         PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

FADD/FADDP/FIADD—Add (Continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of unlike sign.
#D	Result is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	Description
DF /4	FBLD *m80 dec*	Convert BCD value to real and push onto the FPU stack.

Description

Converts the BCD source operand into extended-real format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

Operation

```
TOP ← TOP – 1;
ST(0) ← ExtendedReal(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack overflow occurred.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FBLD—Load Binary Coded Decimal (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	Description
DF /6	FBSTP m80bcd	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

ST(0)	DEST
–•	*
–F < –1	–D
–1 < –F < –0	**
–0	–0
+0	+0
+0 < +F < +1	**
+F > +1	+D
+∞	*
NaN	*

Notes:
Fmeans finite-real number.
Dmeans packed-BCD number.
*indicates floating-point invalid-operation (#IA) exception.
**±0 or ±1, depending on the rounding mode.

If the source value is too large for the destination format and the invalid-operation exception is not masked, an invalid-operation exception is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

If the source value is a quiet NaN, an invalid-operation exception is generated. Quiet NaNs do not normally cause this exception to be generated.

Operation

```
DEST ← BCD(ST(0));
PopRegisterStack;
```

FBSTP—Store BCD Integer and Pop (Continued)

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact exception (#P) is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is empty; contains a NaN, ±∞, or unsupported format; or contains value that exceeds 18 BCD digits in length.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a segment register is being loaded with a segment selector that points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

FBSTP—Store BCD Integer and Pop (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FCHS—Change Sign

Opcode	Instruction	Description
D9 E0	FCHS	Complements sign of ST(0)

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice-versa. The following table shows the results obtained when creating the absolute value of various classes of numbers.

ST(0) SRC	ST(0) DEST
–•	+∞
–F	+F
–0	+0
+0	–0
+F	–F
+∞	–•
NaN	NaN

Note:
Fmeans finite-real number.

Operation

```
SignBit(ST(0)) ← NOT (SignBit(ST(0)))
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FCLEX/FNCLEX—Clear Exceptions

Opcode	Instruction	Description
9B DB E2	FCLEX	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

Operation

```
FPUStatusWord[0..7] ← 0;
FPUStatusWord[15] ← 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set. /

FCMOV*cc*—Floating-point Conditional Move

Opcode	Instruction	Description
DA C0+i	FCMOVB ST(0), ST(*i*)	Move if below (CF=1)
DA C8+i	FCMOVE ST(0), ST(*i*)	Move if equal (ZF=1)
DA D0+i	FCMOVBE ST(0), ST(*i*)	Move if below or equal (CF=1 or ZF=1)
DA D8+i	FCMOVU ST(0), ST(*i*)	Move if unordered (PF=1)
DB C0+i	FCMOVNB ST(0), ST(*i*)	Move if not below (CF=0)
DB C8+i	FCMOVNE ST(0), ST(*i*)	Move if not equal (ZF=0)
DB D0+i	FCMOVNBE ST(0), ST(*i*)	Move if not below or equal (CF=0 and ZF=0)
DB D8+i	FCMOVNU ST(0), ST(*i*)	Move if not unordered (PF=0)

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The source operand is always in the ST(*i*) register and the destination operand is always ST(0).

The FCMOV*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor in the Pentium Pro processor family may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" on page 4:78). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

Operation

```
IF condition TRUE
   ST(0) ← ST(i)
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.

Integer Flags Affected

None.

FCMOV*cc*—Floating-point Conditional Move (Continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FCOM/FCOMP/FCOMPP—Compare Real

Opcode	Instruction	Description
D8 /2	FCOM *m32real*	Compare ST(0) with *m32real*.
DC /2	FCOM *m64real*	Compare ST(0) with *m64real*.
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP *m32real*	Compare ST(0) with *m32real* and pop register stack.
DC /3	FCOMP *m64real*	Compare ST(0) with m64real and pop register stack.
D8 D8+i	FCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that -0.0 = +0.0.

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered[a]	1	1	1

[a] Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared. If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to "unordered." If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

FCOM/FCOMP/FCOMPP—Compare Real (Continued)

Operation

```
CASE (relation of operands) OF
   ST > SRC:          C3, C2, C0 ← 000;
   ST < SRC:          C3, C2, C0 ← 001;
   ST = SRC:          C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC = NaN or unsupported format
   THEN
         #IA
         IF FPUControlWord.IM = 1
               THEN
                     C3, C2, C0 ← 111;
         FI;
FI;
IF instruction = FCOMP
   THEN
         PopRegisterStack;
FI;
IF instruction = FCOMPP
   THEN
         PopRegisterStack;
         PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	See table on previous page.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats. Register is marked empty.
#D	One or both operands are denormal values.

FCOM/FCOMP/FCOMPP—Compare Real (Continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS

Opcode	Instruction	Description
DB F0+i	FCOMI ST, ST(*i*)	Compare ST(0) with ST(*i*) and set status flags accordingly
DF F0+i	FCOMIP ST, ST(*i*)	Compare ST(0) with ST(*i*), set status flags accordingly, and pop register stack
DB E8+i	FUCOMI ST, ST(*i*)	Compare ST(0) with ST(*i*), check for ordered values, and set status flags accordingly
DF E8+i	FUCOMIP ST, ST(*i*)	Compare ST(0) with ST(*i*), check for ordered values, set status flags accordingly, and pop register stack

Description

Compares the contents of register ST(0) and ST(*i*) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that -0.0 = +0.0.

Comparison Results	ZF	PF	CF
ST0 > ST(*i*)	0	0	0
ST0 < ST(*i*)	0	0	1
ST0 = ST(*i*)	1	0	0
Unordered[a]	1	1	1

[a] Flags not set if unmasked invalid-arithmetic- operand (#IA) exception is generated.

The FCOMI/FCOMIP instructions perform the same operation as the FUCOMI/FUCOMIP instructions. The only difference is how they handle QNaN operands. The FCOMI/FCOMIP instructions set the status flags to “unordered” and generate an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value (SNaN or QNaN) or is in an unsupported format.

The FUCOMI/FUCOMIP instructions perform the same operation as the FCOMI/FCOMIP instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

If invalid-operation exception is unmasked, the status flags are not set if the invalid-arithmetic-operand exception is generated.

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS (Continued)

Operation

```
CASE (relation of operands) OF
   ST(0) > ST(i):      ZF, PF, CF ← 000;
   ST(0) < ST(i):      ZF, PF, CF ← 001;
   ST(0) = ST(i):      ZF, PF, CF ← 100;
ESAC;
IF instruction is FCOMI or FCOMIP
   THEN
         IF ST(0) or ST(i) = NaN or unsupported format
               THEN
                     #IA
                     IF FPUControlWord.IM = 1
                           THEN
                                 ZF, PF, CF ← 111;
                     FI;
         FI;
FI;
IF instruction is FUCOMI or FUCOMIP
   THEN
         IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
               THEN
                     ZF, PF, CF ← 111;
               ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
                      #IA;
                     IF FPUControlWord.IM = 1
                           THEN
                                 ZF, PF, CF ← 111;
                     FI;
         FI;
FI;
IF instruction is FCOMIP or FUCOMIP
   THEN
         PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	Not affected.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Real and Set EFLAGS (Continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	(FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats. (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set./

FCOS—Cosine

Opcode	Instruction	Description
D9 FF	FCOS	Replace ST(0) with its cosine

Description

Calculates the cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers, assuming that neither overflow nor underflow occurs.

ST(0) SRC	ST(0) DEST
−∞	*
−F	−1 to +1
−0	+1
+0	+1
+F	−1 to +1
+∞	*
NaN	NaN

Notes:
Fmeans finite-real number.
* indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π.

Operation

```
IF |ST(0)| < $2^{63}$
THEN
   C2 ← 0;
   ST(0) ← cosine(ST(0));
ELSE (*source operand is out-of-range *)
   C2 ← 1;
FI;
```

FCOS—Cosine (Continued)

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.

Undefined if C2 is 1.

C2 Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.

C0, C3 Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value, ∞, or unsupported format.

#D Result is a denormal value.

#U Result is too small for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	Description
D9 F6	FDECSTP	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). The contents of the FPU data registers and tag register are not affected.

Operation

```
IF TOP = 0
   THEN TOP ← 7;
   ELSE TOP ← TOP - 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0; otherwise, cleared to 0. The C0, C2, and C3 flags are undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	Description
D8 /6	FDIV *m32real*	Divide ST(0) by *m32real* and store result in ST(0)
DC /6	FDIV *m64real*	Divide ST(0) by *m64real* and store result in ST(0)
D8 F0+i	FDIV ST(0), ST(*i*)	Divide ST(0) by ST(*i*) and store result in ST(0)
DC F8+i	FDIV ST(i), ST(0)	Divide ST(*i*) by ST(0) and store result in ST(*i*)
DE F8+i	FDIVP ST(*i*), ST(0)	Divide ST(*i*) by ST(0), store result in ST(*i*), and pop the register stack
DE F9	FDIVP	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack
DA /6	FIDIV *m32int*	Divide ST(0) by *m32int* and store result in ST(0)
DE /6	FIDIV *m16int*	Divide ST(0) by *m64int* and store result in ST(0)

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a real or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(*i*) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to extended-real format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide by zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

FDIV/FDIVP/FDIVP—Divide (Continued)

		DEST						
		-•	–F	–0	+0	+F	+∞	NaN
	-∞	*	+0	+0	–0	–0	*	NaN
	–F	+∞	+F	+0	–0	–F	-•	NaN
	–I	+∞	+F	+0	–0	–F	-•	NaN
SRC	–0	+∞	**	*	*	**	-•	NaN
	+0	-•	**	*	*	**	+∞	NaN
	+I	-•	–F	–0	+0	+F	+∞	NaN
	+F	-•	–F	–0	+0	+F	+∞	NaN
	+∞	*	–0	–0	+0	+0	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Imeans integer.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.
**indicates floating-point zero-divide (#Z) exception.

Operation

```
IF SRC = 0
   THEN
         #Z
   ELSE
         IF instruction is FIDIV
               THEN
                     DEST ← DEST / ConvertExtendedReal(SRC);
               ELSE (* source operand is real number *)
                     DEST ← DEST / SRC;
         FI;
FI;
IF instruction = FDIVP
   THEN
         PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

FDIV/FDIVP/FIDIV—Divide (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

±∞ / ±∞; ±0 / ±0

#D Result is a denormal value.

#Z DEST / ±0, where DEST is not equal to ±0.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	Description
D8 /7	FDIVR *m32real*	Divide *m32real* by ST(0) and store result in ST(0)
DC /7	FDIVR *m64real*	Divide *m64real* by ST(0) and store result in ST(0)
D8 F8+i	FDIVR ST(0), ST(i)	Divide ST(*i*) by ST(0) and store result in ST(0)
DC F0+i	FDIVR ST(i), ST(0)	Divide ST(0) by ST(*i*) and store result in ST(*i*)
DE F0+i	FDIVRP ST(i), ST(0)	Divide ST(0) by ST(*i*), store result in ST(*i*), and pop the register stack
DE F1	FDIVRP	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack
DA /7	FIDIVR *m32int*	Divide *m32int* by ST(0) and store result in ST(0)
DE /7	FIDIVR *m16int*	Divide *m64int* by ST(0) and store result in ST(0)

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a real or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(*i*) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to extended-real format before performing the division.

If an unmasked divide by zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

FDIVR/FDIVRP/FIDIVR—Reverse Divide (Continued)

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
	−∞	*	+∞	+∞	-•	−∞	*	NaN
SRC	−F	+0	+F	**	**	-F	−0	NaN
	−I	+0	+F	**	**	-F	−0	NaN
	−0	+0	+0	*	*	−0	−0	NaN
	+0	−0	−0	*	*	+0	+0	NaN
	+I	−0	-F	**	**	+F	+∞	NaN
	+F	−0	-F	**	**	+F	+∞	NaN
	+∞	*	−∞	−∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Imeans integer.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.
**indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0.

Operation

```
IF DEST = 0
   THEN
         #Z
   ELSE
         IF instruction is FIDIVR
               THEN
                     DEST ← ConvertExtendedReal(SRC) / DEST;
               ELSE (* source operand is real number *)
                     DEST ← SRC / DEST;
         FI;
FI;
IF instruction = FDIVRP
   THEN
         PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

FDIVR/FDIVRP/FIDIVR—Reverse Divide (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

±∞ / ±∞; ±0 / ±0

#D Result is a denormal value.

#Z SRC / ±0, where SRC is not equal to ±0.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FFREE—Free Floating-point Register

Opcode	Instruction	Description
DD C0+i	FFREE ST(*i*)	Sets tag for ST(*i*) to empty

Description

Sets the tag in the FPU tag register associated with register ST(*i*) to empty (11B). The contents of ST(*i*) and the FPU stack-top pointer (TOP) are not affected.

Operation

```
TAG(i) ← 11B;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FICOM/FICOMP—Compare Integer

Opcode	Instruction	Description
DE /2	FICOM *m16int*	Compare ST(0) with *m16int*
DA /2	FICOM *m32int*	Compare ST(0) with *m32int*
DE /3	FICOMP *m16int*	Compare ST(0) with *m16int* and pop stack register
DA /3	FICOMP *m32int*	Compare ST(0) with *m32int* and pop stack register

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to extended-real format before the comparison is made.

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared. If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that -0.0 = +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

Operation

```
CASE (relation of operands) OF
   ST(0) > SRC:      C3, C2, C0 ← 000;
   ST(0) < SRC:      C3, C2, C0 ← 001;
   ST(0) = SRC:      C3, C2, C0 ← 100;
   Unordered:        C3, C2, C0 ← 111;
ESAC;
IF instruction = FICOMP
   THEN
         PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set to 0.
C0, C2, C3	See table on previous page.

FICOM/FICOMP—Compare Integer (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FIILD—Load Integer

Opcode	Instruction	Description
DF /0	FILD *m16int*	Push *m16int* onto the FPU register stack.
DB /0	FILD *m32int*	Push *m32int* onto the FPU register stack.
DF /5	FILD *m64int*	Push *m64int* onto the FPU register stack.

Description

Converts the signed-integer source operand into extended-real format and pushes the value onto the FPU register stack. The source operand can be a word, short, or long integer value. It is loaded without rounding errors. The sign of the source operand is preserved.

Operation

```
TOP ← TOP – 1;
ST(0) ← ExtendedReal(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; cleared to 0 otherwise.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FILD—Load Integer (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	Description
D9 F7	FINCSTP	Increment the TOP field in the FPU status register

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

Operation

```
IF TOP = 7
   THEN TOP ← 0;
   ELSE TOP ← TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0; otherwise, generates an #IS fault. The C0, C2, and C3 flags are undefined.

Floating-point Exceptions

#IS

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FINIT/FNINIT—Initialize Floating-point Unit

Opcode	Instruction	Description
9B DB E3	FINIT	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT	Initialize FPU without checking for pending unmasked floating-point exceptions.

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

Operation

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

C0, C1, C2, C3 cleared to 0.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FIST/FISTP—Store Integer

Opcode	Instruction	Description
DF /2	FIST *m16int*	Store ST(0) in *m16int*
DB /2	FIST *m32int*	Store ST(0) in *m32int*
DF /3	FISTP *m16int*	Store ST(0) in *m16int* and pop register stack
DB /3	FISTP *m32int*	Store ST(0) in *m32int* and pop register stack
DF /7	FISTP *m64int*	Store ST(0) in *m64int* and pop register stack

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word- or short-integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction can also stores values in long-integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

ST(0)	DEST
−∞	*
−F < −1	−I
−1 < −F < −0	**
−0	0
+0	0
+0 < +F < +1	**
+F > +1	+I
+∞	*
NaN	*

Notes:
Fmeans finite-real number.
Imeans integer.
*indicates floating-point invalid-operation (#IA) exception.
**±0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the value being stored is too large for the destination format, is an ∞, is a NaN, or is in an unsupported format and if the invalid-arithmetic-operand exception (#IA) is unmasked, an invalid-operation exception is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in the destination operand.

FIST/FISTP—Store Integer (Continued)

Operation

```
DEST ← Integer(ST(0));
IF instruction = FISTP
  THEN
      PopRegisterStack;
FI;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction of if the inexact exception (#P) is generated: 0 = not roundup; 1 = roundup.

Cleared to 0 otherwise.

C0, C2, C3 Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is too large for the destination format

Source operand is a NaN value or unsupported format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FIST/FISTP—Store Integer (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FLD—Load Real

Opcode	Instruction	Description
D9 /0	FLD *m32real*	Push *m32real* onto the FPU register stack.
DD /0	FLD *m64real*	Push *m64real* onto the FPU register stack.
DB /5	FLD *m80real*	Push *m80real* onto the FPU register stack.
D9 C0+i	FLD ST(i)	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. If the source operand is in single- or double-real format, it is automatically converted to the extended-real format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(*i*)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

Operation

```
IF SRC is ST(i)
   THEN
         temp ← ST(i)
TOP ← TOP – 1;
FI;
IF SRC is memory-operand
   THEN
         ST(0) ← ExtendedReal(SRC);
   ELSE (* SRC is ST(i) *)
         ST(0) ← temp;
FI;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

#IS	Stack overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value. Does not occur if the source operand is in extended-real format.

FLD—Load Real (Continued)

FLD—Load Real (Continued)

Protected Mode Exceptions

#GP(0) If destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode	Instruction	Description
D9 E8	FLD1	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Push π onto the FPU register stack.
D9 EC	FLDLG2	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Push +0.0 onto the FPU register stack.

Description

Push one of seven commonly-used constants (in extended-real format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to external-real format. The inexact-result exception (#P) is not generated as a result of the rounding.

Operation

```
TOP ← TOP – 1;
ST(0) ← CONSTANT;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Floating-point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant (Continued)

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

Intel Architecture Compatibility Information

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel287 math coprocessors.

FLDCW—Load Control Word

Opcode	Instruction	Description
D9 /5	FLDCW m2byte	Load FPU control word from *m2byte.*

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions. To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

Operation

```
FPUControlWord ← SRC;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next waiting floating-point instruction.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FLDCW—Load Control Word (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FLDENV—Load FPU Environment

Opcode	Instruction	Description
D9 /4	FLDENV *m14/28byte*	Load FPU environment from *m14byte* or *m28byte*.

Description

Loads the complete FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory.This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions. To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

Operation

```
FPUControlWord ← SRC(FPUControlWord);
FPUStatusWord ← SRC(FPUStatusWord);
FPUTagWord ← SRC(FPUTagWord);
FPUDataPointer ← SRC(FPUDataPointer);
FPUInstructionPointer ← SRC(FPUInstructionPointer);
FPULastInstructionOpcode ← SRC(FPULastInstructionOpcode);
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Floating-point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next waiting floating-point instruction.

FLDENV—Load FPU Environment (Continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	Description
D8 /1	FMUL *m32real*	Multiply ST(0) by *m32real* and store result in ST(0)
DC /1	FMUL *m64real*	Multiply ST(0) by *m64real* and store result in ST(0)
D8 C8+i	FMUL ST(0), ST(i)	Multiply ST(0) by ST(i) and store result in ST(0)
DC C8+i	FMUL ST(*i*), ST(0)	Multiply ST(*i*) by ST(0) and store result in ST(*i*)
DE C8+i	FMULP ST(*i*), ST(0)	Multiply ST(*i*) by ST(0), store result in ST(*i*), and pop the register stack
DE C9	FMULP	Multiply ST(0) by ST(1), store result in ST(0), and pop the register stack
DA /1	FIMUL *m32int*	Multiply *m32int* by ST(0) and store result in ST(0)
DE /1	FIMUL *m16int*	Multiply *m16int* by ST(0) and store result in ST(0)

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

The no-operand version of the instruction multiplies the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version multiplies the contents of a memory location (either a real or an integer value) by the contents of the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(*i*) register or vice versa.

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to extended-real format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

FMUL/FMULP/FIMUL—Multiply (Continued)

		DEST						
		–∞	–F	–0	+0	+F	+∞	NaN
SRC	–∞	+∞	+∞	*	*	–∞	–∞	NaN
	–F	+∞	+F	+0	–0	–F	–∞	NaN
	–I	+∞	+F	+0	–0	–F	–∞	NaN
	–0	*	+0	+0	–0	–0	*	NaN
	+0	*	–0	–0	+0	+0	*	NaN
	+I	–∞	–F	–0	+0	+F	+∞	NaN
	+F	–∞	–F	–0	+0	+F	+∞	NaN
	+∞	–∞	–∞	*	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Imeans Integer.
*indicates invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIMUL
   THEN
         DEST ← DEST ∗ ConvertExtendedReal(SRC);
   ELSE (* source operand is real number *)
         DEST ← DEST ∗ SRC;
FI;
IF instruction = FMULP
   THEN
         PopRegisterStack
FI;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.

One operand is ±0 and the other is ±∞.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

FMUL/FMULP/FIMUL—Multiply (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FNOP—No Operation

Opcode	Instruction	Description
D9 D0	FNOP	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register.

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FPATAN—Partial Arctangent

Opcode	Instruction	Description
D9 F3	FPATAN	Replace ST(1) with arctan(ST(1)/ST(0)) and pop the register stack

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than +π.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 2-6. FPATAN Zeros and NaNs

		ST(0)						
		-•	–F	–0	+0	+F	+∞	NaN
ST(1)	-•	–3π/4	–π/2	–π/2	–π/2	–π/2	–π/4	NaN
	–F	-p	–π to –π/2	–π/2	–π/2	–π/2 to –0	-0	NaN
	–0	-p	-p	-p	–0	–0	–0	NaN
	+0	+π	+π	+π	+0	+0	+0	NaN
	+F	+π	+π to +π/2	+π/2	+π/2	+π/2 to +0	+0	NaN
	+∞	+3π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Note:
Fmeans finite-real number.

There is no restriction on the range of source operands that FPATAN can accept.

Operation

```
ST(1) ← arctan(ST(1) / ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPATAN—Partial Arctangent (Continued)

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

Intel Architecture Compatibility Information

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \leq |ST(1)| < |ST(0)| < +\infty$

FPREM—Partial Remainder

Opcode	Instruction	Description
D9 F8	FPREM	Replace ST(0) with the remainder obtained on dividing ST(0) by ST(1)

Description

Computes the remainder obtained on dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or *modulus*), and stores the result in ST(0). The remainder represents the following value:

Remainder = ST(0) − (N ∗ ST(1))

Here, N is an integer value that is obtained by truncating the real-number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 2-7. FPREM Zeros and NaNs

					ST(1)			
		-•	–F	–0	+0	+F	+∞	NaN
ST(0)	-•	*	*	*	*	*	*	NaN
	–F	ST(0)	–F or –0	**	**	–F or –0	ST(0)	NaN
	–0	–0	–0	*	*	–0	–0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.
**indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std. 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

FPREM—Partial Remainder (Continued)

The FPREM instruction gets its name "partial remainder" because of the way it computes the remainder. This instructions arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the *partial remainder*. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared.

Note: While executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

Operation

```
D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
  THEN
      Q ← Integer(TruncateTowardZero(ST(0) / ST(1)));
      ST(0) ← ST(0) - (ST(1) ∗ Q);
      C2 ← 0;
      C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
  ELSE
      C2 ← 1;
      N ← an implementation-dependent number between 32 and 63;
      QQ ← Integer(TruncateTowardZero((ST(0)  / ST(1)) / $2^{(D - N)}$));
      ST(0) ← ST(0) - (ST(1) ∗ QQ ∗ $2^{(D - N)}$);
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPREM—Partial Remainder (Continued)

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FPREM1—Partial Remainder

Opcode	Instruction	Description
D9 F5	FPREM1	Replace ST(0) with the IEEE remainder obtained on dividing ST(0) by ST(1)

Description

Computes the IEEE remainder obtained on dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or *modulus*), and stores the result in ST(0). The remainder represents the following value:

Remainder = ST(0) − (N ∗ ST(1))

Here, N is an integer value that is obtained by rounding the real-number quotient of [ST(0) / ST(1)] toward the nearest integer value. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 2-8. FPREM1 Zeros and NaNs

		ST(1)						
		-•	–F	–0	+0	+F	+∞	NaN
ST(0)	-•	*	*	*	*	*	*	NaN
	–F	ST(0)	–F or –0	**	**	–F or –0	ST(0)	NaN
	–0	–0	–0	*	*	–0	–0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.
**indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Std 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” below).

FPREM1—Partial Remainder (Continued)

Like the FPREM instruction, the FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the *partial remainder*. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared.

Note: While executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

Operation

```
D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
   THEN
         Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
         ST(0) ← ST(0) - (ST(1) ∗ Q);
         C2 ← 0;
         C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
   ELSE
         C2 ← 1;
         N ← an implementation-dependent number between 32 and 63;
         QQ ← Integer(TruncateTowardZero((ST(0)  / ST(1)) / $2^{(D – N)}$));
         ST(0) ← ST(0) - (ST(1) ∗ QQ ∗ $2^{(D – N)}$);
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FPREM1—Partial Remainder (Continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FPTAN—Partial Tangent

Opcode	Instruction	Clocks	Description
D9 F2	FPTAN	17-173	Replace ST(0) with its tangent and push 1 onto the FPU stack.

Description

Computes the tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

ST(0) SRC	ST(0) DEST
–∞	*
–F	–F to +F
–0	–0
+0	+0
+F	–F to +F
+∞	*
NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

Operation

```
IF ST(0) < $2^{63}$
THEN
    C2 ← 0;
    ST(0) ← tan(ST(0));
    TOP ← TOP – 1;
    ST(0) ← 1.0;
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FRNDINT—Round to Integer

Opcode	Instruction	Description
D9 FC	FRNDINT	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

Operation

```
ST(0) ← RoundToIntegralValue(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FRSTOR—Restore FPU State

Opcode	Instruction	Description
DD /4	FRSTOR *m94/108byte*	Load FPU state from *m94byte* or *m108byte*.

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

Operation

```
FPUControlWord ← SRC(FPUControlWord);
FPUStatusWord ← SRC(FPUStatusWord);
FPUTagWord ← SRC(FPUTagWord);
FPUDataPointer ← SRC(FPUDataPointer);
FPUInstructionPointer ← SRC(FPUInstructionPointer);
FPULastInstructionOpcode ← SRC(FPULastInstructionOpcode);
ST(0) ← SRC(ST(0));
ST(1) ← SRC(ST(1));
ST(2) ← SRC(ST(2));
ST(3) ← SRC(ST(3));
ST(4) ← SRC(ST(4));
ST(5) ← SRC(ST(5));
ST(6) ← SRC(ST(6));
ST(7) ← SRC(ST(7));
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

FRSTOR—Restore FPU State (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSAVE/FNSAVE—Store FPU State

Opcode	Instruction	Description
9B DD /6	FSAVE *m94/108byte*	Store FPU state to *m94byte* or *m108byte* after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE *m94/108byte*	Store FPU environment to *m94byte* or *m108byte* without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-point Unit" on page 4:133).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure.

Operation

```
(* Save FPU State and Registers *)
DEST(FPUControlWord) ← FPUControlWord;
DEST(FPUStatusWord) ← FPUStatusWord;
DEST(FPUTagWord) ← FPUTagWord;
DEST(FPUDataPointer) ← FPUDataPointer;
DEST(FPUInstructionPointer) ← FPUInstructionPointer;
DEST(FPULastInstructionOpcode) ← FPULastInstructionOpcode;
DEST(ST(0)) ← ST(0);
DEST(ST(1)) ← ST(1);
DEST(ST(2)) ← ST(2);
DEST(ST(3)) ← ST(3);
DEST(ST(4)) ← ST(4);
DEST(ST(5)) ← ST(5);
DEST(ST(6)) ← ST(6);
DEST(ST(7)) ← ST(7);
(* Initialize FPU *)
FPUControlWord ← 037FH;
```

FSAVE/FNSAVE—Store FPU State (Continued)

```
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If destination is located in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility Information

For Intel math coprocessors and FPUs prior to the Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps insure that the storage operation has been completed.

FSCALE—Scale

Opcode	Instruction	Description
D9 FD	FSCALE	Scale ST(0) by ST(1).

Description

Multiplies the destination operand by 2 to the power of the source operand and stores the result in the destination operand. This instruction provides rapid multiplication or division by integral powers of 2. The destination operand is a real value that is located in register ST(0). The source operand is the nearest integer value that is smaller than the value in the ST(1) register (that is, the value in register ST(1) is truncate toward 0 to its nearest integer value to form the source operand). The actual scaling operation is performed by adding the source operand (integer value) to the exponent of the value in register ST(0). The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

		ST(1)		
		–N	0	+N
	–∞	–∞	–∞	–∞
ST(0)	–F	–F	–F	–F
	–0	–0	–0	–0
	+0	+0	+0	+0
	+F	+F	+F	+F
	+∞	+∞	+∞	+∞
	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Nmeans integer.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction returns the recreated value to the FPU register where it originally resided.

FSCALE—Scale (Continued)

Operation

```
ST(0) ← ST(0) ∗ $2^{ST(1)}$;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FSIN—Sine

Opcode	Instruction	Description
D9 FE	FSIN	Replace ST(0) with its sine.

Description

Calculates the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

SRC (ST(0))	DEST (ST(0))
−∞	*
−F	−1 to +1
−0	−0
+0	+0
+F	−1 to +1
+∞	*
NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π.

Operation

```
IF ST(0) < $2^{63}$
THEN
   C2 ← 0;
   ST(0) ← sin(ST(0));
ELSE (* source operand out of range *)
   C2 ← 1;
FI:
```

FSIN—Sine (Continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FSINCOS—Sine and Cosine

Opcode	Instruction	Description
D9 FB	FSINCOS	Compute the sine and cosine of ST(0); replace ST(0) with the sine, and push the cosine onto the register stack.

Description

Computes both the sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

SRC	DEST	
ST(0))	ST(0) Cosine	ST(1) Sine
–∞	*	*
–F	–1 to +1	–1 to +1
–0	+1	–0
+0	+1	+0
+F	–1 to +1	–1 to +1
+∞	*	*
NaN	NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π or by using the FPREM instruction with a divisor of 2π.

Operation

```
IF ST(0) < $2^{63}$
THEN
   C2 ← 0;
   TEMP ← cosine(ST(0));
   ST(0) ← sine(ST(0));
   TOP ← TOP – 1;
   ST(0) ← TEMP;
ELSE (* source operand out of range *)
   C2 ← 1;
FI:
```

FSINCOS—Sine and Cosine (Continued)

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range -2^{63} to $+2^{63}$; otherwise, cleared to 0.
C0, C3	Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FSQRT—Square Root

Opcode	Instruction	Description
D9 FA	FSQRT	Calculates square root of ST(0) and stores the result in ST(0)

Description

Calculates the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

SRC (ST(0))	DEST (ST(0))
−∞	*
−F	*
−0	−0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
ST(0) ← SquareRoot(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for −0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

FSQRT—Square Root (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FST/FSTP—Store Real

Opcode	Instruction	Description
D9 /2	FST *m32real*	Copy ST(0) to *m32real*
DD /2	FST *m64real*	Copy ST(0) to *m64real*
DD D0+i	FST ST(*i*)	Copy ST(0) to ST(i)
D9 /3	FSTP *m32real*	Copy ST(0) to *m32real* and pop register stack
DD /3	FSTP *m64real*	Copy ST(0) to *m64real* and pop register stack
DB /7	FSTP *m80real*	Copy ST(0) to *m80real* and pop register stack
DD D8+i	FSTP ST(*i*)	Copy ST(0) to ST(*i*) and pop register stack

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU registers stack. When storing the value in memory, the value is converted to single- or double-real format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also stores values in memory in extended-real format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single- or double-real, the significand of the value being stored is rounded to the width of the destination (according to rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value’s identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

Operation

```
DEST ← ST(0);
IF instruction = FSTP
   THEN
         PopRegisterStack;
FI;
```

FST/FSTP—Store Real (Continued)

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value or unsupported format.

#U Result is too small for the destination format.

#O Result is too large for the destination format.

#P Value cannot be represented exactly in destination format.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

FST/FSTP—Store Real (Continued)

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FSTCW/FNSTCW—Store Control Word

Opcode	Instruction	Description
9B D9 /7	FSTCW *m2byte*	Store FPU control word to *m2byte* after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW *m2byte*	Store FPU control word to *m2byte* without checking for pending unmasked floating-point exceptions.

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

Operation

```
DEST ← FPUControlWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FSTCW/FNSTCW—Store Control Word (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FSTENV/FNSTENV—Store FPU Environment

Opcode	Instruction	Description
9B D9 /6	FSTENV *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV *m14/28byte*	Store FPU environment to *m14byte* or *m28byte* without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. See the ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** for the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the size of the current address attribute (16-bit or 32-bit). (In virtual-8086 mode, the real mode layouts are used.)

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not.The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the procedure stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

Operation

```
DEST(FPUControlWord) ← FPUControlWord;
DEST(FPUStatusWord) ← FPUStatusWord;
DEST(FPUTagWord) ← FPUTagWord;
DEST(FPUDataPointer) ← FPUDataPointer;
DEST(FPUInstructionPointer) ← FPUInstructionPointer;
DEST(FPULastInstructionOpcode) ← FPULastInstructionOpcode;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-point Exceptions

None.

FSTENV/FNSTENV—Store FPU Environment (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSTSW/FNSTSW—Store Status Word

Opcode	Instruction	Description
9B DD /7	FSTSW *m2byte*	Store FPU status word at *m2byte* after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW *m2byte*	Store FPU status word at *m2byte* without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW AX	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

Description

Stores the current value of the FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

Operation

```
DEST ← FPUStatusWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

FSTSW/FNSTSW—Store Status Word (Continued)

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	Description
D8 /4	FSUB *m32real*	Subtract *m32real* from ST(0) and store result in ST(0)
DC /4	FSUB *m64real*	Subtract *m64real* from ST(0) and store result in ST(0)
D8 E0+i	FSUB ST(0), ST(*i*)	Subtract ST(*i*) from ST(0) and store result in ST(0)
DC E8+i	FSUB ST(*i*), ST(0)	Subtract ST(0) from ST(*i*) and store result in ST(*i*)
DE E8+i	FSUBP ST(*i*), ST(0)	Subtract ST(0) from ST(*i*), store result in ST(*i*), and pop register stack
DE E9	FSUBP	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack
DA /4	FISUB *m32int*	Subtract *m32int* from ST(0) and store result in ST(0)
DE /4	FISUB *m16int*	Subtract *m16int* from ST(0) and store result in ST(0)

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a real or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(*i*) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to extended-real format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

FSUB/FSUBP/FISUB—Subtract (Continued)

Table 2-9. FSUB Zeros and NaNs

		SRC						
		–∞	–F or –I	–0	+0	+F or +I	+∞	NaN
DEST	–∞	*	–∞	–∞	–∞	–∞	–∞	NaN
	–F	+∞	±F or ±0	DEST	DEST	–F	–∞	NaN
	–0	+∞	–SRC	±0	–0	–SRC	–∞	NaN
	+0	+∞	–SRC	+0	±0	–SRC	–∞	NaN
	+F	+∞	+F	DEST	DEST	±F or ±0	–∞	NaN
	+∞	+∞	+∞	+∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Imeans integer.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUB
   THEN
         DEST ← DEST – ConvertExtendedReal(SRC);
   ELSE (* source operand is real number *)
         DEST ← DEST – SRC;
FI;
IF instruction = FSUBP
   THEN
         PopRegisterStack
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

FSUB/FSUBP/FISUB—Subtract (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM EM or TS in CR0 is set.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	Description
D8 /5	FSUBR *m32real*	Subtract ST(0) from *m32real* and store result in ST(0)
DC /5	FSUBR *m64real*	Subtract ST(0) from *m64real* and store result in ST(0)
D8 E8+i	FSUBR ST(0), ST(i)	Subtract ST(0) from ST(*i*) and store result in ST(0)
DC E0+i	FSUBR ST(i), ST(0)	Subtract ST(*i*) from ST(0) and store result in ST(*i*)
DE E0+i	FSUBRP ST(i), ST(0)	Subtract ST(0) from ST(*i*), store result in ST(*i*), and pop register stack
DE E1	FSUBRP	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack
DA /5	FISUBR *m32int*	Subtract ST(0) from *m32int* and store result in ST(0)
DE /5	FISUBR *m16int*	Subtract ST(0) from *m16int* and store result in ST(0)

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a real or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(*i*) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to extended-real format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

FSUBR/FSUBRP/FISUBR—Reverse Subtract (Continued)

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 2-10. FSUBR Zeros and NaNs

		SRC						
		–∞	–F	–0	+0	+F	+∞	NaN
	–∞	*	+∞	+∞	+∞	+∞	+∞	NaN
DEST	–F or –I	–∞	±F or ±0	–DEST	–DEST	+F	+∞	NaN
	–0	–∞	SRC	±0	+0	SRC	+∞	NaN
	+0	–∞	SRC	–0	±0	SRC	+∞	NaN
	+F or +I	–∞	–F	–DEST	–DEST	±F or ±0	+∞	NaN
	+∞	–∞	–∞	–∞	–∞	–∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
Imeans integer.
*indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FISUBR
  THEN
        DEST ← ConvertExtendedReal(SRC) – DEST;
  ELSE (* source operand is real number *)
        DEST ← SRC – DEST;
FI;
IF instruction = FSUBRP
  THEN
        PopRegisterStack
FI;
```

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

FSUBR/FSUBRP/FISUBR—Reverse Subtract (Continued)

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

FTST—TEST

Opcode	Instruction	Description
D9 E4	FTST	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0)	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine" on page 4:193). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to "unordered.")

The sign of zero is ignored, so that -0.0 = +0.0.

Operation

```
CASE (relation of operands) OF
   Not comparable:  C3, C2, C0 ← 111;
   ST(0) > 0.0:     C3, C2, C0 ← 000;
   ST(0) < 0.0:     C3, C2, C0 ← 001;
   ST(0) = 0.0:     C3, C2, C0 ← 100;
ESAC;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	See above table.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
#D	One or both operands are denormal values.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

FTST—TEST (Continued)

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Real

Opcode	Instruction	Description
DD E0+i	FUCOM ST(i)	Compare ST(0) with ST(i)
DD E1	FUCOM	Compare ST(0) with ST(1)
DD E8+i	FUCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack
DD E9	FUCOMP	Compare ST(0) with ST(1) and pop register stack
DA E9	FUCOMPP	Compare ST(0) with ST(1) and pop register stack twice

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that -0.0 = +0.0.

Comparison Results	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered[a]	1	1	1

[a] Flags not set if unmasked invalid-arithmetic- operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine" on page 4:193). The FUCOM instructions perform the same operation as the FCOM instructions. The only difference is that the FUCOM instruction raises the invalid-arithmetic-operand exception (#IA) only when either or both operands is an SNaN or is in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM instruction raises an invalid-operation exception when either or both of the operands is a NaN value of any kind or is in an unsupported format.

As with the FCOM instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instructions pop the register stack following the comparison operation and the FUCOMPP instructions pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

Operation

```
CASE (relation of operands) OF
   ST > SRC:          C3, C2, C0 ← 000;
   ST < SRC:          C3, C2, C0 ← 001;
   ST = SRC:          C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
```

```
 THEN
     C3, C2, C0 ← 111;
 ELSE (* ST(0) or SRC is SNaN or unsupported format *)
      #IA;
     IF FPUControlWord.IM = 1
         THEN
             C3, C2, C0 ← 111;
     FI;
FI;
IF instruction = FUCOMP
 THEN
     PopRegisterStack;
FI;
IF instruction = FUCOMPP
 THEN
     PopRegisterStack;
     PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	See table on previous page.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D	One or both operands are denormal values.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FWAIT—Wait

See entry for WAIT.

FXAM—Examine

Opcode	Instruction	Description
D9 E5	FXAM	Classify value or number in ST(0)

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

.

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

Operation

```
C1 ← sign bit of ST; (* 0 for positive, 1 for negative *)
CASE (class of value or number in ST(0)) OF
   Unsupported:C3, C2, C0 ← 000;
   NaN:         C3, C2, C0 ← 001;
   Normal:      C3, C2, C0 ← 010;
   Infinity:    C3, C2, C0 ← 011;
   Zero:        C3, C2, C0 ← 100;
   Empty:       C3, C2, C0 ← 101;
   Denormal:    C3, C2, C0 ← 110;
ESAC;
```

FPU Flags Affected

C1	Sign of value in ST(0).
C0, C2, C3	See table above.

Floating-point Exceptions

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FXAM—Examine (Continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FXCH—Exchange Register Contents

Opcode	Instruction	Description
D9 C8+i	FXCH ST(i)	Exchange the contents of ST(0) and ST(*i*)
D9 C9	FXCH	Exchange the contents of ST(0) and ST(1)

Description

Exchanges the contents of registers ST(0) and ST(*i*). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

Operation

```
IF number-of-operands is 1
  THEN
      temp ← ST(0);
      ST(0) ← SRC;
      SRC ← temp;
  ELSE
      temp ← ST(0);
      ST(0) ← ST(1);
      ST(1) ← temp;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, cleared to 0.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack underflow occurred.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

FXCH—Exchange Register Contents (Continued)

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FXTRACT—Extract Exponent and Significand

Opcode	Instruction	Description
D9 F4	FXTRACT	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a real number. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a real number. (The operation performed by this instruction is a superset of the IEEE-recommended logb(*x*) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in extended-real format to decimal representations (e.g. for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of -∞ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP← TOP – 1;
ST(0) ← TEMP;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

Floating-point Exceptions

#IS	Stack underflow occurred. Stack overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is ±0.
#D	Source operand is a denormal value.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

FXTRACT—Extract Exponent and Significand (Continued)

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FYL2X—Compute y × $\log_2 x$

Opcode	Instruction	Description
D9 F1	FYL2X	Replace ST(1) with (ST(1) ∗ $\log_2$ST(0)) and pop the register stack

Description

Calculates (ST(1) ∗ $\log_2$ (ST(0))), stores the result in resister ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 2-11. FYL2X Zeros and NaNs

		ST(0)						
		−∞	−F	+0	+0	+F	+∞	NaN
	−∞	*	*	+∞	+∞	+∞	−∞	NaN
ST(1)	−F	*	*	**	**	±F	−∞	NaN
	−0	*	*	*	*	+0	*	NaN
	+0	*	*	*	*	+0	*	NaN
	+F	*	*	**	**	±F	+∞	NaN
	+∞	*	*	−∞	−∞	−∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-operation (#IA) exception.
**indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$\log_b x = (\log_2 b)^{-1} * \log_2 x$

Operation

```
ST(1) ← ST(1) ∗ log2ST(0);
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

FYL2X—Compute y × $\log_2 x$ (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format.
	Source operand in register ST(0) is a negative finite value (not –0).
#Z	Source operand in register ST(0) is ±0.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM	EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	EM or TS in CR0 is set.

FYL2XP1—Compute y ∗ $\log_2(x + 1)$

Opcode	Instruction	Description
D9 F9	FYL2XP1	Replace ST(1) with ST(1) ∗ $\log_2$(ST(0) + 1.0) and pop the register stack

Description

Calculates the log epsilon (ST(1) ∗ $\log_2$(ST(0) + 1.0)), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1-\sqrt{2}/2))to(1-\sqrt{2}/2)$$

The source operand in ST(1) can range from −∞ to +∞. If either of the source operands is outside its acceptable range, the result is undefined and no exception is generated.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur:

Table 2-12. FYL2XP1 Zeros and NaNs

		ST(0)						
		−∞	$-(1-(\sqrt{2}/2))$ to −0	−0	+0	+0 to $+(1-(\sqrt{2}/2))$	+∞	NaN
ST(1)	−∞	*	+∞	*	*	−∞	−∞	NaN
	−F	*	+F	+0	−0	−F	−∞	NaN
	−0	*	+0	+0	−0	−0	*	NaN
	+0	*	−0	−0	+0	+0	*	NaN
	+F	*	−F	−0	+0	+F	+∞	NaN
	+∞	*	−∞	*	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
Fmeans finite-real number.
*indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. When the epsilon value (ε) is small, more significant digits can be retained by using the FYL2XP1 instruction than by using (ε+1) as an argument to the FYL2X instruction. The (ε+1) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} = \log_n 2$$

Operation

```
ST(1) ← ST(1) ∗ $\log_2$(ST(0) + 1.0);
PopRegisterStack;
```

FYL2XP1—Compute y $*$ $\log_2$(x +1) (Continued)

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.

C0, C2, C3 Undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Floating-point Exceptions

#IS Stack underflow occurred.

#IA Either operand is an SNaN value or unsupported format.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM EM or TS in CR0 is set.

Real Address Mode Exceptions

#NM EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM EM or TS in CR0 is set.

HLT—Halt

Opcode	Instruction	Description
F4	HLT	Halt

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual 8086 mode, the privilege level of a program or procedure must to 0 to execute the HLT instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,HALT);
Enter Halt state;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If the current privilege level is not 0.

IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV *r/m8*	Signed divide AX (where AH must contain sign-extension of AL) by *r/m* byte. (Results: AL=Quotient, AH=Remainder)
F7 /7	IDIV *r/m16*	Signed divide DX:AX (where DX must contain sign-extension of AX) by *r/m* word. (Results: AX=Quotient, DX=Remainder)
F7 /7	IDIV *r/m32*	Signed divide EDX:EAX (where EDX must contain sign-extension of EAX) by *r/m* doubleword. (Results: EAX=Quotient, EDX=Remainder)

Description

Divides (signed) the value in the AL, AX, or EAX register by the source operand and stores the result in the AX, DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size, as shown in the following table:

Table 2-13. IDIV Operands

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF flag.

Operation

```
IF SRC = 0
   THEN #DE; (* divide error *)
FI;
IF OpernadSize = 8 (* word/byte operation *)
   THEN
         temp ← AX / SRC; (* signed division *)
         IF (temp > 7FH) OR (temp < 80H)
         (* if a positive result is greater than 7FH or a negative result is less than 80H *)
               THEN #DE; (* divide error *) ;
               ELSE
                     AL ← temp;
                     AH ← AX SignedModulus SRC;
         FI;
   ELSE
         IF OpernadSize = 16 (* doubleword/word operation *)
               THEN
```

IDIV—Signed Divide (Continued)

```
            temp ← DX:AX / SRC; (* signed division *)
            IF (temp > 7FFFH) OR (temp < 8000H)
            (* if a positive result is greater than 7FFFH *)
            (* or a negative result is less than 8000H *)
                THEN #DE; (* divide error *) ;
                ELSE
                    AX ← temp;
                    DX ← DX:AX SignedModulus SRC;
            FI;
        ELSE (* quadword/doubleword operation *)
            temp ← EDX:EAX / SRC; (* signed division *)
            IF (temp > 7FFFFFFFH) OR (temp < 80000000H)
            (* if a positive result is greater than 7FFFFFFFH *)
            (* or a negative result is less than 80000000H *)
                THEN #DE; (* divide error *) ;
                ELSE
                    EAX ← temp;
                    EDX ← EDXE:AX SignedModulus SRC;
            FI;
    FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#DE If the source operand (divisor) is 0.

The signed result (quotient) is too large for the destination.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

IDIV—Signed Divide (Continued)

Real Address Mode Exceptions

#DE If the source operand (divisor) is 0.

The signed result (quotient) is too large for the destination.

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#DE If the source operand (divisor) is 0.

The signed result (quotient) is too large for the destination.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL *r/m8*	AX← AL ∗ *r/m* byte
F7 /5	IMUL *r/m16*	DX:AX ← AX ∗ *r/m* word
F7 /5	IMUL *r/m32*	EDX:EAX ← EAX ∗ *r/m* doubleword
0F AF */r*	IMUL *r16,r/m16*	word register ← word register ∗ *r/m* word
0F AF */r*	IMUL *r32,r/m32*	doubleword register ← doubleword register ∗ *r/m* doubleword
6B */r ib*	IMUL *r16,r/m16,imm8*	word register ← *r/m16* ∗ sign-extended immediate byte
6B */r ib*	IMUL *r32,r/m32,imm8*	doubleword register ← *r/m32* ∗ sign-extended immediate byte
6B */r ib*	IMUL *r16,imm8*	word register ← word register ∗ sign-extended immediate byte
6B */r ib*	IMUL *r32,imm8*	doubleword register ← doubleword register ∗ sign-extended immediate byte
69 */r iw*	IMUL *r16,r/ m16,imm16*	word register ← *r/m16* ∗ immediate word
69 */r id*	IMUL *r32,r/ m32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword
69 */r iw*	IMUL *r16,imm16*	word register ← *r/m16* ∗ immediate word
69 */r id*	IMUL *r32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form.** This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.
- **Two-operand form.** With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form.** This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

IMUL—Signed Multiply (Continued)

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands = 1)
   THEN IF (OperandSize = 8)
         THEN
               AX ← AL ∗ SRC  (* signed multiplication *)
               IF ((AH = 00H) OR (AH = FFH))
                     THEN CF = 0; OF = 0;
                     ELSE CF = 1; OF = 1;
               FI;
         ELSE IF OperandSize = 16
               THEN
                     DX:AX ← AX ∗ SRC  (* signed multiplication *)
                     IF ((DX = 0000H) OR (DX = FFFFH))
                           THEN CF = 0; OF = 0;
                           ELSE CF = 1; OF = 1;
                     FI;
               ELSE (* OperandSize = 32 *)
                     EDX:EAX ← EAX ∗ SRC  (* signed multiplication *)
                     IF ((EDX = 00000000H) OR (EDX = FFFFFFFFH))
                           THEN CF = 0; OF = 0;
                           ELSE CF = 1; OF = 1;
                     FI;
         FI;
   ELSE IF (NumberOfOperands = 2)
         THEN
               temp ← DEST ∗ SRC    (* signed multiplication; temp is double DEST size*)
               DEST ← DEST ∗ SRC  (* signed multiplication *)
               IF temp ≠ DEST
                     THEN CF = 1; OF = 1;
                     ELSE CF = 0; OF = 0;
               FI;

         ELSE (* NumberOfOperands = 3 *)
               DEST ← SRC1 ∗ SRC2   (* signed multiplication *)
               temp ← SRC1 ∗ SRC2     (* signed multiplication; temp is double SRC1 size *)
               IF temp ≠ DEST
                     THEN CF = 1; OF = 1;
                     ELSE CF = 0; OF = 0;
               FI;
   FI;
FI;
```

IMUL—Signed Multiply (Continued)

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

IN—Input from Port

Opcode	Instruction	Description
E4 *ib*	IN AL,*imm8*	Input byte from *imm8* I/O port address into AL
E5 *ib*	IN AX,*imm8*	Input byte from *imm8* I/O port address into AX
E5 *ib*	IN EAX,*imm8*	Input byte from *imm8* I/O port address into EAX
EC	IN AL,DX	Input byte from I/O port in DX into AL
ED	IN AX,DX	Input word from I/O port in DX into AX
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the Itanium System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize the TLB in the Itanium architecture to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing IN instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
  THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
      IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
          THEN #GP(0);
      FI;
```

IN—Input from Port (Continued)

```
    ELSE ( * Real-address mode or protected mode with CPL ≤ IOPL *)
    (* or virtual-8086 mode with all I/O permission bits for I/O port cleared *)
FI;

IF (Itanium_System_Environment THEN
    SRC_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
    SRC_PA = translate(SRC_VA);
    DEST ← [SRC_PA]; (* Reads from I/O port *)
FI;

memory_fence();
DEST <-SRC;
memory-fence();
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA_32_Exception Debug traps for data breakpoints and single step

IA_32_Exception Alignment faults

#GP(0) Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **when CFLG.io is 1.**

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

Adds 1 to the operand, while preserving the state of the CF flag. The source operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform a increment operation that does updates the CF flag.)

Operation

```
DEST ← DEST - 1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the operand is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

INC—Increment by 1 (Continued)

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Description
6C	INS ES:(E)DI, DX	Input byte from port DX into ES:(E)DI
6D	INS ES:DI, DX	Input word from port DX into ES:DI
6D	INS ES:EDI, DX	Input doubleword from port DX into ES:EDI
6C	INSB	Input byte from port DX into ES:(E)DI
6D	INSW	Input word from port DX into ES:DI
6D	INSD	Input doubleword from port DX into ES:EDI

Description

Copies the data from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand must be the DX register, allowing I/O port addresses from 0 to 65,535 to be accessed. When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

The destination operand is a memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the destination-index register.) The ES segment cannot be overridden with a segment override prefix.

The INSB, INSW, and INSD mnemonics are synonyms of the byte, word, and doubleword versions of the INS instructions. (For the INS instruction, "ES:EDI" must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the Itanium System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize the TLBs in the Itanium architecture to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

INS/INSB/INSW/INSD—Input from Port to String (Continued)

If the referenced I/O port is mapped to an unimplemented virtual address (via the IOBase register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing INS instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
   THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
         IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
               THEN #GP(0);
         FI;
   ELSE ( * I/O operation is allowed *)
FI;
IF (Itanium_System_Environment) THEN
   SRC_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
   SRC_PA = translate(SRC_VA);
   DEST ← [SRC_PA]; (* Reads from I/O port *)
FI;

memory_fence();
DEST <- SRC;
memory_fence();
         IF (byte transfer)
               THEN IF DF = 0
                     THEN (E)DI ← 1;
                     ELSE (E)DI ← -1;
               FI;
               ELSE IF (word transfer)
                     THEN IF DF = 0
                           THEN DI ← 2;
                           ELSE DI ← -2;
                     FI;
                     ELSE (* doubleword transfer *)
                           THEN IF DF = 0
                                 THEN EDI ← 4;
                                 ELSE EDI ← -4;
                           FI;
               FI;
         FI;
FI;
```

Flags Affected

None.

INS/INSB/INSW/INSD—Input from Port to String (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA_32_Exception Debug traps for data breakpoints and single step

IA_32_Exception Alignment faults

#GP(0) Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.

If the destination is located in a nonwritable segment.

If an illegal memory operand effective address in the ES segments is given.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

INT*n*/INTO/INT3—Call to Interrupt Procedure

Opcode	Instruction	Description
CC	INT3	Interrupt 3—trap to debugger
CD *ib*	INT *imm8*	Interrupt vector numbered by immediate byte
CE	INTO	Interrupt 4—if overflow flag is 1

Description

The INT*n* instruction generates a call to the interrupt or exception handler specified with the destination operand. The destination operand specifies an interrupt vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. The first 32 interrupt vectors are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT*n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1.

The INT3 instruction is a special mnemonic for calling the debug exception handler. The action of the INT3 instruction (opcode CC) is slightly different from the operation of the INT 3 instruction (opcode CC03), as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

The action of the INT*n* instruction (including the INTO and INT3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT*n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which points to an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to procedure in the selected segment.

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

Table 2-14. INT Cases

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

Notes:
– Don't Care
Y Yes, Action Taken
BlankAction Not Taken

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT*n* instruction. If the IOPL is less than 3, the processor generates a general protection exception (#GP); if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to three and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT*n* and INTO instructions, but also to external interrupts and exceptions.

```
IF Itanium System EnvironmentTHEN
  IF INT3 Form THEN IA_32_Exception(3);
  IF INTO Form THEN IA_32_Exception(4);
  IF INT Form THEN IA-32_Interrupt(N);
FI;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
/*IN the Itanium System Environment all of the following operations are intercepted*/
IF PE=0
   THEN
         GOTO REAL-ADDRESS-MODE;
   ELSE (* PE=1 *)
         GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE:
   IF ((DEST ∗ 4) + 3) is not within IDT limit THEN #GP; FI;
   IF stack not large enough for a 6-byte return information THEN #SS; FI;
   Push (EFLAGS[15:0]);
   IF ← 0; (* Clear interrupt flag *)
   TF ← 0; (* Clear trap flag *)
   AC ← 0; (*Clear AC flag*)
   Push(CS);
   Push(IP);
   (* No error codes are pushed *)
   CS ← IDT(Descriptor (vector ∗ 4), selector));
   EIP ← IDT(Descriptor (vector ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
   IF ((DEST ∗ 8) + 7) is not within IDT limits
         OR selected IDT descriptor is not an interrupt-, trap-, or task-gate type
               THEN #GP((DEST ∗ 8) + 2 + EXT);
               (* EXT is bit 0 in error code *)
   FI;
   IF software interrupt (* generated by INTn, INT3, or INTO *)
         THEN
               IF gate descriptor DPL < CPL
                     THEN #GP((vector number ∗ 8) + 2 );
                     (* PE=1, DPL<CPL, software interrupt *)
               FI;
   FI;
   IF gate not present THEN #NP((vector number ∗ 8) + 2 + EXT); FI;
   IF task gate (* specified in the selected interrupt table descriptor *)
         THEN GOTO TASK-GATE;
         ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE=1, trap/interrupt gate *)
   FI;
END;

TASK-GATE: (* PE=1, task gate *)
   Read segment selector in task gate (IDT descriptor);
         IF local/global bit is set to local
               OR index not within GDT limits
                     THEN #GP(TSS selector);
         FI;
         Access TSS descriptor in GDT;
         IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
               THEN #GP(TSS selector);
         FI;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
IF TSS not present
          THEN #NP(TSS selector);
      FI;
  SWITCH-TASKS (with nesting) to TSS;
  IF interrupt caused by fault with error code
      THEN
          IF stack limit does not allow push of two bytes
              THEN #SS(0);
          FI;
          Push(error code);
  FI;
  IF EIP not within code segment limit
      THEN #GP(0);
  FI;
END;
TRAP-OR-INTERRUPT-GATE
  Read segment selector for trap or interrupt gate (IDT descriptor);
  IF segment selector for code segment is null
      THEN #GP(0H + EXT); (* null selector with EXT flag set *)
  FI;
  IF segment selector is not within its descriptor table limits
      THEN #GP(selector + EXT);
  FI;
  Read trap or interrupt handler descriptor;
  IF descriptor does not indicate a code segment
      OR code segment descriptor DPL > CPL
          THEN #GP(selector + EXT);
  FI;
  IF trap or interrupt gate segment is not present,
      THEN #NP(selector + EXT);
  FI;
  IF code segment is non-conforming AND DPL < CPL
      THEN IF VM=0
          THEN
              GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
              (* PE=1, interrupt or trap gate, nonconforming *)
              (* code segment, DPL<CPL, VM=0 *)
          ELSE (* VM=1 *)
              IF code segment DPL ≠ 0 THEN #GP(new code segment selector); FI;
              GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE;
              (* PE=1, interrupt or trap gate, DPL<CPL, VM=1 *)
      FI;
      ELSE (* PE=1, interrupt or trap gate, DPL ≥ CPL *)
          IF VM=1 THEN #GP(new code segment selector); FI;
          IF code segment is conforming OR code segment DPL = CPL
              THEN
                  GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
              ELSE
                  #GP(CodeSegmentSelector + EXT);
                  (* PE=1, interrupt or trap gate, nonconforming *)
                  (* code segment, DPL>CPL *)
          FI;
```

```
    FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT
    (* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
    (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← new code segment (DPL ∗ 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← new code segment (DPL ∗ 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
    IF segment selector is null THEN #TS(EXT); FI;
    IF segment selector index is not within its descriptor table limits
        OR segment selector's RPL ≠ DPL of code segment,
            THEN #TS(SS selector + EXT);
    FI;
Read segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present THEN #SS(SS selector+EXT); FI;
    IF 32-bit gate
        THEN
            IF new stack does not have room for 24 bytes (error code pushed)
                OR 20 bytes (no error code pushed)
                    THEN #SS(segment selector + EXT);
            FI;
        ELSE (* 16-bit gate *)
            IF new stack does not have room for 12 bytes (error code pushed)
                OR 10 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT);
            FI;
    FI;
    IF instruction pointer is not within code segment limits THEN #GP(0); FI;
    SS:ESP ← TSS(SS:ESP) (* segment descriptor information also loaded *)
    IF 32-bit gate
        THEN
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
        ELSE (* 16-bit gate *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
    FI;
    IF 32-bit gate
        THEN
            Push(far pointer to old stack); (* old SS and ESP, 3 words padded to 4 *);
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
          Push(EFLAGS);
          Push(far pointer to return instruction); (* old CS and EIP, 3 words padded to 4*);
          Push(ErrorCode); (* if needed, 4 bytes *)
      ELSE(* 16-bit gate *)
          Push(far pointer to old stack); (* old SS and SP, 2 words *);
          Push(EFLAGS);
          Push(far pointer to return instruction); (* old CS and IP, 2 words *);
          Push(ErrorCode); (* if needed, 2 bytes *)
  FI;
  CPL ← CodeSegmentDescriptor(DPL);
  CS(RPL) ← CPL;
  IF interrupt gate
      THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
  TF ← 0;
  VM ← 0;
  RF ← 0;
  NT ← 0;
I END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
  (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
  IF current TSS is 32-bit TSS
      THEN
          TSSstackAddress ← new code segment (DPL ∗ 8) + 4
          IF (TSSstackAddress + 7) > TSS limit
              THEN #TS(current TSS selector); FI;
          NewSS ← TSSstackAddress + 4;
          NewESP ← stack address;
      ELSE (* TSS is 16-bit *)
          TSSstackAddress ← new code segment (DPL ∗ 4) + 2
          IF (TSSstackAddress + 4) > TSS limit
              THEN #TS(current TSS selector); FI;
          NewESP ← TSSstackAddress;
          NewSS ← TSSstackAddress + 2;
  FI;
      IF segment selector is null THEN #TS(EXT); FI;
      IF segment selector index is not within its descriptor table limits
          OR segment selector's RPL ≠ DPL of code segment,
              THEN #TS(SS selector + EXT);
      FI;
  Access segment descriptor for stack segment in GDT or LDT;
  IF stack segment DPL ≠ DPL of code segment,
      OR stack segment does not indicate writable data segment,
          THEN #TS(SS selector + EXT);
  FI;
  IF stack segment not present THEN #SS(SS selector+EXT); FI;
  IF 32-bit gate
      THEN
          IF new stack does not have room for 40 bytes (error code pushed)
              OR 36 bytes (no error code pushed);
                  THEN #SS(segment selector + EXT);
          FI;
      ELSE (* 16-bit gate *)
          IF new stack does not have room for 20 bytes (error code pushed)
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
              OR 18 bytes (no error code pushed);
                  THEN #SS(segment selector + EXT);
          FI;
  FI;
  IF instruction pointer is not within code segment limits THEN #GP(0); FI;

  IF CR4.VME = 0
      THEN
          IF IOPL=3
              THEN
                  IF Gate DPL = 3
                      THEN    (*CPL=3, VM=1, IOPL=3, VME=0, gate DPL=3)
                          IF Target CPL != 0
                              THEN #GP(0);
                              ELSE Goto VM86_INTERURPT_TO_PRIV0;
                          FI;
                      ELSE (*Gate DPL < 3*)
                          #GP(0);
                  FI;
              ELSE (*IOPL < 3*)
                  #GP(0);
          FI;
      ELSE (*VME = 1*)
          (*Check whether interrupt is directed for INT n instruction only,
          (*executes virtual 8086 interupt, protected mode interrupt or faults*)
          Ptr <- [TSS + 66];            (*Fetch IO permission bitmpa pointer*)
          IF BIT[Ptr-32,N] = 0          (*software redirection bitmap is 32 bytes below IO
Permission*)
          THEN (*Interrupt redirected*)
              Goto      VM86_INTERRUPT_TO_VM86;
          ELSE
              IF IOPL = 3
                  THEN
                      IF Gate DPL = 3
                          THEN
                              IF Target CPL != 0
                              THEN #GP(0);
                              ELSE Goto VM86_INTERRUPT_TO_PRIV0;
                              FI;
                          ELSE #GP(0);
                      FI;
                  ELSE (*IOPL < 3*)
                      #GP(0);
              FI;
          FI;
  FI;
END;

VM86_INTERRUPT_TO_PRIV0:
  tempEFLAGS ← EFLAGS;
  VM ← 0;
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
  TF ← 0;
  RF ← 0;
  IF service through interrupt gate THEN IF ← 0; FI;
  TempSS ← SS;
  TempESP ← ESP;
  SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
  (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates *)
  (* Segment selector pushes in 32-bit mode are padded to two words *)
  Push(GS);
  Push(FS);
  Push(DS);
  Push(ES);
  Push(TempSS);
  Push(TempESP);
  Push(TempEFlags);
  Push(CS);
  Push(EIP);
  GS ← 0; (*segment registers nullified, invalid in protected mode *)
  FS ← 0;
  DS ← 0;
  ES ← 0;
  CS ← Gate(CS);
  IF OperandSize=32
      THEN
          EIP ← Gate(instruction pointer);
      ELSE (* OperandSize is 16 *)
          EIP ← Gate(instruction pointer) AND 0000FFFFH;
  FI;
  (* Starts execution of new routine in Protected Mode *)
END;

VM86_INTERRUPT_TO_VM86:
  IF IOPL = 3
      THEN
          push(FLAGS OR 3000H);           (*Push FLAGS w/ IOPL bits as 11B or IOPL 3*)
          push(CS);
          push(IP);
          CS <- [N*4 + 2];                (*N is vector num, read from interrupt table*)
          IP <- [N*4];
          FLAGS <- FLAGS AND 7CD5H;       (*Clear TF and IF in EFLAGS like 8086*)
      ELSE
          TempFlags <- FLAGS OR 3000H;    (*Set IOPL to 11B or IOPL 3*)
          TempFlags.IF <- EFLAGS.VIF;
          push(TempFlags);
          push(CS);
          push(IP);
          CS <- [N*4 + 2];                (*N is vector num, read from interrupt table*)
          IP <- [N*4];
          FLAGS <- FLAGS AND 77ED5H;      (*Clear VIF and TF and IF in EFLAGS like 8086*)
  FI;
END;

INTRA-PRIVILEGE-LEVEL-INTERRUPT:
```

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

```
    (* PE=1, DPL = CPL or conforming segment *)
    IF 32-bit gate
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
                OR 12 bytes (no error code pushed); THEN #SS(0);
            FI;
        ELSE (* 16-bit gate *)
            IF current stack does not have room for 8 bytes (error code pushed)
                OR 6 bytes (no error code pushed); THEN #SS(0);
            FI;
    IF instruction pointer not within code segment limit THEN #GP(0); FI;
    IF 32-bit gate
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
        ELSE (* 16-bit gate *)
            Push (FLAGS);
            Push (far pointer to return location); (* 2 words *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
    FI;
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN
            IF ← 0; FI;
            TF ← 0;
            NT ← 0;
            VM ← 0;
            RF ← 0;
    FI;
END;
```

Flags Affected

The EFLAGS register is pushed onto stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see “Operation” section.)

Additional Itanium System Environment Exceptions

IA_32_Exception	If INT3 or INTO form, vector numbers are 3 and 4 respectively.
IA-32_Interrupt	If INT n form, vector number is N.

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

Protected Mode Exceptions

#GP(0) If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.

#GP(selector) If the segment selector in the interrupt-, trap-, or task gate is null.

If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.

If the interrupt vector is outside the IDT limits.

If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.

If an interrupt is generated by the INT*n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.

If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the stack segment.

#NP(selector) If code segment, interrupt-, trap-, or task gate, or TSS is not present.

#TS(selector) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.

If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.

If the stack segment selector in the TSS is null.

If the stack segment for the TSS is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the interrupt vector is outside the IDT limits.

#SS If stack limit violation on push.

If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment when a stack switch occurs.

INT*n*/INTO/INT3—Call to Interrupt Procedure (Continued)

Virtual 8086 Mode Exceptions

#GP(0) (For INT*n* instruction) If the IOPL is less than 3 and the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.

If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.

#GP(selector) If the segment selector in the interrupt-, trap-, or task gate is null.

If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.

If the interrupt vector is outside the IDT limits.

If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.

If an interrupt is generated by the INT*n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.

If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.

If the segment selector for a TSS has its local/global bit set for local.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.

#NP(selector) If code segment, interrupt-, trap-, or task gate, or TSS is not present.

#TS(selector) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.

If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.

If the stack segment selector in the TSS is null.

If the stack segment for the TSS is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#BP If the INT3 instruction is executed.

#OF If the INTO instruction is executed and the OF flag is set.

INVD—Invalidate Internal Caches

Opcode	Instruction	Description
0F 08	INVD	Flush internal caches; initiate flushing of external caches.

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also implementation-dependent; its function may be implemented differently on future Intel architecture processors.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,INVD);
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	The INVD instruction cannot be executed at the virtual 8086 mode.

INVD—Invalidate Internal Caches (Continued)

Intel Architecture Compatibility

This instruction is not supported on Intel architecture processors earlier than the Intel486 processor.

INVLPG—Invalidate TLB Entry

Opcode	Instruction	Description
0F 01/7	INVLPG m	Invalidate TLB Entry for page that contains *m*

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also implementation-dependent; its function may be implemented differently on future Intel architecture processors.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,INVLPG);
Flush(RelevantTLBEntries);
Continue (* Continue execution);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual 8086 mode.

Intel Architecture Compatibility

This instruction is not supported on Intel architecture processors earlier than the Intel486 processor.

IRET/IRETD—Interrupt Return

Opcode	Instruction	Description
CF	IRET	Interrupt return (16-bit operand size)
CF	IRETD	Interrupt return (32-bit operand size)

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt or, a software-generated interrupt, or returns from a nested task. IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Real Mode.
- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.

Return from nested task (task switch)

All forms of IRET result in an IA-32_Intercept(Inst,IRET) in the Itanium System Environment.

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

IRET/IRETD—Interrupt Return (Continued)

If the NT flag is set, the IRET instruction performs a return from a nested task (switches from the called task back to the calling task) or reverses the operation of an interrupt or exception that caused a task switch. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is reentered later, the code that follows the IRET instruction is executed.

IRET performs an instruction serialization and a memory fence operation.

Operation

```
IF(Itanium System Environment)
      THEN IA-32_Intercept(Inst,IRET);
IF PE = 0
   THEN
      GOTO REAL-ADDRESS-MODE:;
   ELSE
      GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE;
   IF OperandSize = 32
      THEN
         IF top 12 bytes of stack not within stack limits THEN #SS; FI;
         IF instruction pointer not within code segment limits THEN #GP(0); FI;
         EIP ← Pop();
         CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
         tempEFLAGS ← Pop();
         EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
      ELSE (* OperandSize = 16 *)
         IF top 6 bytes of stack are not within stack limits THEN #SS; FI;
         IF instruction pointer not within code segment limits THEN #GP(0); FI;
         EIP ← Pop();
         EIP ← EIP AND 0000FFFFH;
         CS ← Pop(); (* 16-bit pop *)
         EFLAGS[15:0] ← Pop();
   FI;
   END;

PROTECTED-MODE:
   IF VM = 1 (* Virtual-8086 mode: PE=1, VM=1 *)
      THEN
         GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE=1, VM=1 *)
   FI;
   IF NT = 1
      THEN
         GOTO TASK-RETURN;( *PE=1, VM=0, NT=1 *)
   FI;
   IF OperandSize=32
      THEN
         IF top 12 bytes of stack not within stack limits
```

```
              THEN #SS(0)
          FI;
          tempEIP ← Pop();
          tempCS ← Pop();
          tempEFLAGS ← Pop();
      ELSE (* OperandSize = 16 *)
          IF top 6 bytes of stack are not within stack limits
              THEN #SS(0);
          FI;
          tempEIP ← Pop();
          tempCS ← Pop();
          tempEFLAGS ← Pop();
          tempEIP ← tempEIP AND FFFFH;
          tempEFLAGS ← tempEFLAGS AND FFFFH;
  FI;
  IF tempEFLAGS(VM) = 1 AND CPL=0
      THEN
          GOTO RETURN-TO-VIRTUAL-8086-MODE;
          (* PE=1, VM=1 in EFLAGS image *)
      ELSE
          GOTO PROTECTED-MODE-RETURN;
          (* PE=1, VM=0 in EFLAGS image *)
  FI;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
  IF CR4.VME = 0
  THEN
      IF IOPL=3 (* Virtual mode: PE=1, VM=1, IOPL=3 *)
          THEN
              IF OperandSize = 32
              THEN
                  IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                  IF instruction pointer not within code segment limits THEN #GP(0); FI;
                  EIP ← Pop();
                  CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                  EFLAGS ← Pop();
                  (*VM,IOPL,VIP,and VIF EFLAGS bits are not modified by pop *)
              ELSE (* OperandSize = 16 *)
                  IF top 6 bytes of stack are not within stack limits THEN #SS(0); FI;
                  IF instruction pointer not within code segment limits THEN #GP(0); FI;
                  EIP ← Pop();
                  EIP ← EIP AND 0000FFFFH;
                  CS ← Pop(); (* 16-bit pop *)
                  EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS is not modified by pop *)
              FI;
          ELSE #GP(0); (* trap to virtual-8086 monitor: PE=1, VM=1, IOPL<3 *)
      FI;
  ELSE (*VME is 1*)
      IF IOPL = 3
          THEN
              IF OperandSize = 32
```

IRET/IRETD—Interrupt Return (Continued)

```
              THEN
                  EIP ← Pop();
                  CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                  TempEFlags ← Pop();
                  FLAGS = (EFLAGS AND 1B3000H) OR (TempEFlags AND 244FD7H)
                  (*VM,IOPL,RF,VIP,and VIF EFLAGS bits are not modified by pop *)
              ELSE (* OperandSize = 16 *)
                  EIP ← Pop();
                  EIP ← EIP AND 0000FFFFH;
                  CS ← Pop(); (* 16-bit pop *)
                  TempFlags <- Pop();
                  FLAGS = (FLAGS AND 3000H) OR (TempFLags AND 4FD5H)
                  (*IOPL unmodified*)
              FI;
      ELSE (*IOPL < 3*)
          IF OperandSize = 16
              THEN
                  IF ((STACK.TF !-0) OR (EFLAGS.VIP=1 AND STACK.IF=1))
                      THEN #GP(0);
                      ELSE
                          IP <- Pop();        (*Word Pops*)
                          CS <- Pop(0);
                          TempFlags <- Pop();
                          (*FLAGS IOPL, IF and TF are not modified*)
                          FLAGS = (FLAGS AND 3302H) OR (TempFlags AND 4CD5H)
                          EFLAGS.VIF <- TempFlags.IF;
                  FI;
              ELSE (*OperandSize = 32 *)
                  #GP(0);
          FI;
  FI;

END;

RETURN-TO-VIRTUAL-8086-MODE:
(* Interrupted procedure was in virtual-8086 mode: PE=1, VM=1 in flags image *)
  IF top 24 bytes of stack are not within stack segment limits
      THEN #SS(0);
  FI;
  IF instruction pointer not within code segment limits
      THEN #GP(0);
  FI;
  CS ← tempCS;
  EIP ← tempEIP;
  EFLAGS ← tempEFLAGS
  TempESP ← Pop();
  TempSS ← Pop();
  ES ← Pop(); (* pop 2 words; throw away high-order word *)
  DS ← Pop(); (* pop 2 words; throw away high-order word *)
  FS ← Pop(); (* pop 2 words; throw away high-order word *)
  GS ← Pop(); (* pop 2 words; throw away high-order word *)
  SS:ESP ← TempSS:TempESP;
```

IRET/IRETD—Interrupt Return (Continued)

```
    (* Resume execution in Virtual 8086 mode *)
END;

TASK-RETURN: (* PE=1, VM=1, NT=1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS for task specified in link field of current TSS;
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
        THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within code segment limit
        THEN #GP(0);
    FI;
END;

PROTECTED-MODE-RETURN: (* PE=1, VM=0 in flags image *)
    IF return code segment selector is null THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
        THEN GP(selector; FI;
    Read segment descriptor pointed to by the return code segment selector
    IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
        AND return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
    IF return code segment descriptor is not present THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
    FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE=1, VM=0 in flags image, RPL=CPL *)
    IF EIP is not within code segment limits THEN #GP(0); FI;
    EIP ← tempEIP;
    CS ← tempCS; (* segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize=32
        THEN
            EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
    IF CPL ≤ IOPL
        THEN
            EFLAGS(IF) ← tempEFLAGS;
    FI;
```

IRET/IRETD—Interrupt Return (Continued)

```
  IF CPL = 0
      THEN
          EFLAGS(IOPL) ← tempEFLAGS;
          IF OperandSize=32
              THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
          FI;
  FI;
END;

RETURN-TO-OUTER-PRIVILGE-LEVEL:

  IF OperandSize=32
      THEN
          IF top 8 bytes on stack are not within limits THEN #SS(0); FI;
      ELSE (* OperandSize=16 *)
          IF top 4 bytes on stack are not within limits THEN #SS(0); FI;
  FI;
  Read return segment selector;
  IF stack segment selector is null THEN #GP(0); FI;
  IF return stack segment selector index is not within its descriptor table limits
          THEN #GP(SSselector); FI;
  Read segment descriptor pointed to by return segment selector;
  IF stack segment selector RPL ≠ RPL of the return code segment selector
      IF stack segment selector RPL ≠ RPL of the return code segment selector
      OR the stack segment descriptor does not indicate a a writable data segment;
      OR stack segment DPL ≠ RPL of the return code segment selector
              THEN #GP(SS selector);
      FI;
      IF stack segment is not present THEN #NP(SS selector); FI;
  IF tempEIP is not within code segment limit THEN #GP(0); FI;
  EIP ← tempEIP;
  CS ← tempCS;
  EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
  IF OperandSize=32
      THEN
          EFLAGS(RF, AC, ID) ← tempEFLAGS;
  FI;
  IF CPO ≤ IOPL
      THEN
          EFLAGS(IF) ← tempEFLAGS;
  FI;
  IF CPL = 0
      THEN
          EFLAGS(IOPL) ← tempEFLAGS;
          IF OperandSize=32
              THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
          FI;
  FI;
  CPL ← RPL of the return code segment selector;
  FOR each of segment register (ES, FS, GS, and DS)
      DO;
          IF segment register points to data or non-conforming code segment
```

IRET/IRETD—Interrupt Return (Continued)

```
            AND CPL > segment descriptor DPL (* stored in hidden part of segment register *)
                THEN (* segment register invalid *)
                    SegmentSelector ← 0; (* null segment selector *)
            FI;
        OD;
END:
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Instruction Intercept Trap for ALL forms of IRET.

Protected Mode Exceptions

#GP(0) If the return code or stack segment selector is null.

If the return instruction pointer is not within the return code segment limit.

#GP(selector) If a segment selector index is outside its descriptor table limits.

If the return code segment selector RPL is greater than the CPL.

If the DPL of a conforming-code segment is greater than the return code segment selector RPL.

If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.

If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

If the stack segment is not a writable data segment.

If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

If the segment descriptor for a code segment does not indicate it is a code segment.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If the top bytes of stack are not within stack limits.

#NP(selector) If the return code or stack segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

IRET/IRETD—Interrupt Return (Continued)

Real Address Mode Exceptions

#GP If the return instruction pointer is not within the return code segment limit.

#SS If the top bytes of stack are not within stack limits.

Virtual 8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit.

IF IOPL not equal to 3

#PF(fault-code) If a page fault occurs.

#SS(0) If the top bytes of stack are not within stack limits.

#AC(0) If an unaligned memory reference occurs and alignment checking is enabled.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	Jump short if zero (ZF = 1)
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)
0F 8F *cw/cd*	JG *rel16/32*	Jump near if greater (ZF=0 and SF=OF)

Jcc—Jump if Condition Is Met (Continued)

Opcode	Instruction	Description
0F 8D *cw/cd*	JGE *rel16/32*	Jump near if greater or equal (SF=OF)
0F 8C *cw/cd*	JL *rel16/32*	Jump near if less (SF<>OF)
0F 8E *cw/cd*	JLE *rel16/32*	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 *cw/cd*	JNA *rel16/32*	Jump near if not above (CF=1 or ZF=1)
0F 82 *cw/cd*	JNAE *rel16/32*	Jump near if not above or equal (CF=1)
0F 83 *cw/cd*	JNB *rel16/32*	Jump near if not below (CF=0)
0F 87 *cw/cd*	JNBE *rel16/32*	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 *cw/cd*	JNC *rel16/32*	Jump near if not carry (CF=0)
0F 85 *cw/cd*	JNE *rel16/32*	Jump near if not equal (ZF=0)
0F 8E *cw/cd*	JNG *rel16/32*	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C *cw/cd*	JNGE *rel16/32*	Jump near if not greater or equal (SF<>OF)
0F 8D *cw/cd*	JNL *rel16/32*	Jump near if not less (SF=OF)
0F 8F *cw/cd*	JNLE *rel16/32*	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 *cw/cd*	JNO *rel16/32*	Jump near if not overflow (OF=0)
0F 8B *cw/cd*	JNP *rel16/32*	Jump near if not parity (PF=0)
0F 89 *cw/cd*	JNS *rel16/32*	Jump near if not sign (SF=0)
0F 85 *cw/cd*	JNZ *rel16/32*	Jump near if not zero (ZF=0)
0F 80 *cw/cd*	JO *rel16/32*	Jump near if overflow (OF=1)
0F 8A *cw/cd*	JP *rel16/32*	Jump near if parity (PF=1)
0F 8A *cw/cd*	JPE *rel16/32*	Jump near if parity even (PF=1)
0F 8B *cw/cd*	JPO *rel16/32*	Jump near if parity odd (PF=0)
0F 88 *cw/cd*	JS *rel16/32*	Jump near if sign (SF=1)
0F 84 *cw/cd*	JZ *rel16/32*	Jump near if 0 (ZF=1)

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each J*cc* mnemonic are given in the "Description" column of the above table. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Jcc—Jump if Condition Is Met (Continued)

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JECXZ and JCXZ instructions differs from the other J*cc* instructions because they do not check the status flags. Instead they check the contents of the ECX and CX registers, respectively, for 0. These instructions are useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE). They prevent entering the loop when the ECX or CX register is equal to 0, which would cause the loop to execute 2^{32} or 64K times, respectively, instead of zero times.

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF condition
   THEN
       EIP ← EIP + SignExtend(DEST);
       IF OperandSize = 16
           THEN
               EIP ← EIP AND 0000FFFFH;
       FI;
   IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA_32_Exception	Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.

Jcc—Jump if Condition Is Met (Continued)

Real Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

JMP—Jump

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump near, relative address
E9 *cw*	JMP *rel16*	Jump near, relative address
E9 *cd*	JMP *rel32*	Jump near, relative address
FF /4	JMP *r/m16*	Jump near, indirect address
FF /4	JMP *r/m32*	Jump near, indirect address
EA *cd*	JMP *ptr16:16*	Jump far, absolute address
EA *cp*	JMP *ptr16:32*	Jump far, absolute address
FF /5	JMP *m16:16*	Jump far, indirect address
FF /5	JMP *m16:32*	Jump far, indirect address

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

- Near jump – A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far jump – A jump to an instruction located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Task switch – A jump to an instruction located in a different task. (This is a form of a far jump.) **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**

A task switch can only be executed in protected mode (see Chapter 6 in the *Intel Architecture Software Developer's Manual, Volume 3* for information on task switching with the JMP instruction).

When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute address (that is an offset from the base of the code segment) or a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). An absolute address is specified directly in a register or indirectly in a memory location (*r/m16* or *r/m32* operand form). A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the value in the EIP register (that is, to the instruction following the JMP instruction). The operand-size attribute determines the size of the target operand (16 or 32 bits) for absolute addresses. Absolute addresses are loaded directly into the EIP register. When a relative offset is specified, it is added to the value of the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. The CS register is not changed on near jumps.

JMP—Jump (Continued)

When executing a far jump, the processor jumps to the code segment and address specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

When the processor is operating in protected mode, a far jump can also be used to access a code segment through a call gate or to switch tasks. Here, the processor uses the segment selector part of the far address to access the segment descriptor for the segment being jumped to. Depending on the value of the type and access rights information in the segment selector, the JMP instruction can perform:

- A far jump to a conforming or non-conforming code segment (same mechanism as the far jump described in the previous paragraph, except that the processor checks the access rights of the code segment being jumped to).
- An far jump through a call gate.
- A task switch. **Results in an IA-32_Intercept(Gate) in Itanium System Environment.**

The JMP instruction cannot be used to perform inter-privilege level jumps.

When executing an far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate and instruction either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction, is similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to. (The offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code, data, and stack segments and the instruction pointer to the target instruction. One form of the JMP instruction allows the jump to be made directly to a TSS, without going through a task gate. See Chapter 13 in *Intel Architecture Software Developer's Manual, Volume 3* the for detailed information on the mechanics of a task switch.

All branches are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

JMP—Jump (Continued)

Operation

```
IF near jump
   THEN IF near relative jump
         THEN
               tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
         ELSE (* near absolute jump *)
               tempEIP ← DEST;
   FI;
   IF tempEIP is beyond code segment limit THEN #GP(0); FI;
   IF OperandSize = 32
         THEN
               EIP ← tempEIP;
         ELSE (* OperandSize=16 *)
               EIP ← tempEIP AND 0000FFFFH;
   FI;
   IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI:

IF far jump AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real address or virtual 8086 mode *)
   THEN
         tempEIP ← DEST(offset); (* DEST is ptr16:32 or [m16:32] *)
         IF tempEIP is beyond code segment limit THEN #GP(0); FI;
         CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
         IF OperandSize = 32
               THEN
                     EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
               ELSE (* OperandSize = 16 *)
                     EIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
         FI;
         IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;
IF far call AND (PE = 1 AND VM = 0) (* Protected mode, not virtual 8086 mode *)
   THEN
         IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
               OR segment selector in target operand null
               THEN #GP(0);
         FI;
         IF segment selector index not within descriptor table limits
               THEN #GP(new selector);
         FI;
         Read type and access rights of segment descriptor;
         IF segment type is not a conforming or nonconforming code segment, call gate,
               task gate, or TSS THEN #GP(segment selector); FI;
         Depending on type and access rights
               GO TO CONFORMING-CODE-SEGMENT;
               GO TO NONCONFORMING-CODE-SEGMENT;
               GO TO CALL-GATE;
               GO TO TASK-GATE;
               GO TO TASK-STATE-SEGMENT;
   ELSE
         #GP(segment selector);
FI;
```

JMP—Jump (Continued)

```
CONFORMING-CODE-SEGMENT:
   IF DPL > CPL THEN #GP(segment selector); FI;
   IF segment not present THEN #NP(segment selector); FI;
   tempEIP ← DEST(offset);
   IF OperandSize=16
         THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
   IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
END;

NONCONFORMING-CODE-SEGMENT:
   IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(code segment selector); FI;
   IF segment not present THEN #NP(segment selector); FI;
   IF instruction pointer outside code segment limit THEN #GP(0); FI;
   tempEIP ← DEST(offset);
   IF OperandSize=16
         THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
   IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
END;

CALL-GATE:

   IF call gate DPL < CPL
         OR call gate DPL < call gate segment-selector RPL
               THEN #GP(call gate selector); FI;
   IF call gate not present THEN #NP(call gate selector); FI;
   IF Itanium System Environment THEN IA-32_Intercept(Gate,JMP);
   IF call gate code-segment selector is null THEN #GP(0); FI;
   IF call gate code-segment selector index is outside descriptor table limits
         THEN #GP(code segment selector); FI;
   Read code segment descriptor;
   IF code-segment segment descriptor does not indicate a code segment
         OR code-segment segment descriptor is conforming and DPL > CPL
         OR code-segment segment descriptor is non-conforming and DPL ≠ CPL
               THEN #GP(code segment selector); FI;
   IF code segment is not present THEN #NP(code-segment selector); FI;
   IF instruction pointer is not within code-segment limit THEN #GP(0); FI;
   tempEIP ← DEST(offset);
   IF GateSize=16
         THEN tempEIP ← tempEIP AND 0000FFFFH;
   FI;
   IF tempEIP not in code segment limit THEN #GP(0); FI;
   CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
   CS(RPL) ← CPL
   EIP ← tempEIP;
```

JMP—Jump (Continued)

```
END;

TASK-GATE:
  IF task gate DPL < CPL
      OR task gate DPL < task gate segment-selector RPL
          THEN #GP(task gate selector); FI;
  IF task gate not present THEN #NP(gate selector); FI;
  IF Itanium System Environment THEN IA-32_Intercept(Gate,JMP);
  Read the TSS segment selector in the task-gate descriptor;
  IF TSS segment selector local/global bit is set to local
      OR index not within GDT limits
      OR TSS descriptor specifies that the TSS is busy
          THEN #GP(TSS selector); FI;
  IF TSS not present THEN #NP(TSS selector); FI;
  SWITCH-TASKS to TSS;
  IF EIP not within code segment limit THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
  IF TSS DPL < CPL
      OR TSS DPL < TSS segment-selector RPL
      OR TSS descriptor indicates TSS not available
          THEN #GP(TSS selector); FI;
  IF TSS is not present THEN #NP(TSS selector); FI;
  IF Itanium System Environment THENIA-32_Intercept(Gate,JMP);
  SWITCH-TASKS to TSS
  IF EIP not within code segment limit THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Additional Itanium System Environment Exceptions

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Gate Intercept for JMP through CALL Gates, Task Gates and Task Segments

IA_32_Exception Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0) If offset in target operand, call gate, or TSS is beyond the code segment limits.

If the segment selector in the destination operand, call gate, task gate, or TSS is null.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

JMP—Jump (Continued)

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#GP(selector) If segment selector index is outside descriptor table limits.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL

(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

If the segment descriptor for selector in a call gate does not indicate it is a code segment.

If the segment descriptor for the segment selector in a task gate does not indicate available TSS.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP (selector) If the code segment being accessed is not present.

If call gate, task gate, or TSS not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If the target operand is beyond the code segment limits.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)

JMPE—Jump to Intel® Itanium® Instruction Set

Opcode	Instruction	Description
0F 00 /6	JMPE *r/m16*	Jump to Intel Itanium instruction set, indirect address specified by r/m16
0F 00 /6	JMPE *r/m32*	Jump to Intel Itanium instruction set, indirect address specified by r/m32
0F B8	JMPE *disp16*	Jump to Intel Itanium instruction set, absolute address specified by addr16
0F B8	JMPE *disp32*	Jump to Intel Itanium instruction set, absolute address specified by addr32

Description

This instruction is available only on processors based on the Itanium architecture in the Itanium System Environment. Otherwise, execution of this instruction at privilege levels 1, 2, and 3 results in an Illegal Opcode fault, and at privilege level 0, termination of the IA-32 System Environment on a processor based on the Itanium architecture.

JMPE switches the processor to the Itanium instruction set and starts execution at the specified target address There are two forms; an indirect form, r/m*r16/32,* and an unsigned absolute form, *disp16/32.* Both 16 and 32-bit formats are supported.

The absolute form computes the 16-byte aligned 64-bit virtual target address in the Itanium instruction set by adding the unsigned 16 or 32-bit displacement to the current CS base (*IP{31:0} = disp16/32 + CSD.base)*. The indirect form specifies the virtual target address by the contents of a register or memory location (*IP{31:0} = [r/m16/32] + CSD.base)*. Target addresses are constrained to the lower 4G-bytes of the 64-bit virtual address space within virtual region 0.

GR[1] is loaded with the next sequential instruction address following JMPE.

If PSR.di is 1, the instruction is nullified and a Disabled Instruction Set Transition fault is generated. If Itanium branch debugging is enabled, an IA_32_Exception(Debug) trap is taken after JMPE completes execution.

JMPE can be performed at any privilege level and does not change the privilege level of the processor.

JMPE performs a FWAIT operation, any pending IA-32 unmasked floating-point exceptions are reported as faults on the JMPE instruction.

JMPE does not perform a memory fence or serialization operation.

Successful execution of JMPE clears EFLAG.rf and PSR.id to zero.

If the register stack engine is enabled for eager execution, the register stack engine may immediately start loading registers when the processor enters the Itanium instruction set.

JMPE—Jump to Intel® Itanium® Instruction Set (Continued)

Operation

```
IF(NOT Itanium System Environment) {
        IF (PSR.cpl==0) Terminate_IA-32_System_Env();
        ELSE IA_32_Exception(IllegalOpcode);
} ELSE IF(PSR.di==1) {
        Disabled_Instruction_Set_Transition_Fault();
} ELSE IF(pending_numeric_exceptions()) {
        IA_32_exception(FPError);
} ELSE {
        IF(absolute_form) {                    //compute virtual target
            IP{31:0} = disp16/32 + AR[CSD].base;//disp is 16/32-bit unsigned value
        } ELSE IF(indirect_form) {
            IP{31:0} = [r/m16/32] + AR[CSD].base;
        }
        PSR.is = 0;                            //set Itanium Instruction Set bit
        IP{3:0}= 0;                            //Force 16-byte alignment
        IP{63:32} = 0;                         //zero extend from 32-bits to 64-bits
        GR[1]{31:0} = EIP + AR[CSD].base;      //next sequential instruction address
        GR[1]{63:32} = 0;
        PSR.id = EFLAG.rf = 0;

        IF (PSR.tb)                            //taken branch trap
            IA_32_Exception(Debug);
}
```

Flags Affected

None (other than EFLAG.rf)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Fault.
Disabled ISA	Disabled Instruction Set Transition Fault, if PSR.di is 1
IA_32_Exception	Floating-point Error, if any floating-point exceptions are pending
IA_32_Exception	Taken Branch trap, if PSR.tb is 1.

IA-32 System Environment Exceptions (All Operating Modes)

#UD	JMPE raises an invalid opcode exception at privilege levels 1, 2 and 3. Privilege level 0 results in termination of the IA-32 System Environment on a processor based on the Itanium architecture.

LAHF—Load Status Flags into AH Register

Opcode	Instruction	Description
9F	LAHF	Load: AH = EFLAGS(SF:ZF:0:AF:0:PF:1:CF)

Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the "Operation" below.

Operation

```
AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
```

Flags Affected

None (that is, the state of the flags in the EFLAGS register are not affected).

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

LAR—Load Access Rights Byte

Opcode	Instruction	Description
0F 02 /*r*	LAR *r16,r/m16*	*r16* ← *r/m16* masked by FF00H
0F 02 /*r*	LAR *r32,r/m32*	*r32* ← *r/m32* masked by 00FxFF00H

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can preform additional checks on the access rights information.

When the operand size is 32 bits, the access rights for a segment descriptor comprise the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FXFF00H before it is loaded into the destination operand. When the operand size is 16 bits, the access rights comprise the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode.

LAR—Load Access Rights Byte (Continued)

Table 2-15. LAR Descriptor Validity

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	Yes
5	16-bit/32-bit task gate	Yes
6	16-bit trap gate	No
7	16-bit interrupt gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	Yes
D	Reserved	No
E	32-bit trap gate	No
F	32-bit interrupt gate	No

Operation

```
IF SRC(Offset) > descriptor table limit THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
  AND (CPL > DPL) OR (RPL > DPL)
  OR Segment type is not valid for instruction
      THEN
          ZF ← 0
      ELSE
          IF OperandSize = 32
              THEN
                  DEST ← [SRC] AND 00FxFF00H;
              ELSE (*OperandSize = 16*)
                  DEST ← [SRC] AND FF00H;
          FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

LAR—Load Access Rights Byte (Continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

Real Address Mode Exceptions

#UD	The LAR instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual 8086 mode.

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Description
C5 /r	LDS *r16,m16:16*	Load DS:*r16* with far pointer from memory
C5 /r	LDS *r32,m16:32*	Load DS:*r32* with far pointer from memory
0F B2 /r	LSS *r16,m16:16*	Load SS:*r16* with far pointer from memory
0F B2 /r	LSS *r32,m16:32*	Load SS:*r32* with far pointer from memory
C4 /r	LES *r16,m16:16*	Load ES:*r16* with far pointer from memory
C4 /r	LES *r32,m16:32*	Load ES:*r32* with far pointer from memory
0F B4 /r	LFS *r16,m16:16*	Load FS:*r16* with far pointer from memory
0F B4 /r	LFS *r32,m16:32*	Load FS:*r32* with far pointer from memory
0F B5 /r	LGS *r16,m16:16*	Load GS:*r16* with far pointer from memory
0F B5 /r	LGS *r32,m16:32*	Load GS:*r32* with far pointer from memory

Description

Load a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register implied with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a null selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

Operation

```
IF ProtectedMode
  THEN IF SS is loaded
      THEN IF SegementSelector = null
          THEN #GP(0);
      FI;
      ELSE IF Segment selector index is not within descriptor table limits
      OR Segment selector RPL ≠ CPL
      OR Access rights indicate nonwritable data segment
      OR DPL ≠ CPL
          THEN #GP(selector);
      FI;
      ELSE IF Segment marked not present
          THEN #SS(selector);
      FI;
      SS ← SegmentSelector(SRC);
```

LDS/LES/LFS/LGS/LSS—Load Far Pointer (Continued)

```
        SS ← SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS, or GS is loaded with non-null segment selector
        THEN IF Segment selector index is not within descriptor table limits
        OR Access rights indicate segment neither data nor readable code segment
        OR (Segment is data or nonconforming-code segment
            AND both RPL and CPL > DPL)
            THEN #GP(selector);
        FI;
        ELSE IF Segment marked not present
            THEN #NP(selector);
        FI;
        SegmentRegister ← SegmentSelector(SRC) AND RPL;
        SegmentRegister ← SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS or GS is loaded with a null selector:
        SegmentRegister ← NullSelector;
        SegmentRegister(DescriptorValidBit) ← 0; (*hidden flag; not accessible by software*)
    FI;
FI;
IF (Real-Address or Virtual 8086 Mode)
    THEN
        SS ← SegmentSelector(SRC);
FI;
DEST ← Offset(SRC);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a null selector is loaded into the SS register. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.

LDS/LES/LFS/LGS/LSS—Load Far Pointer (Continued)

If the DS, ES, FS, or GS register is being loaded with a non-null segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#SS(selector) If the SS register is being loaded and the segment is marked not present.

#NP(selector) If DS, ES, FS, or GS register is being loaded with a non-null segment selector and the segment is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If source operand is not a memory location.

Virtual 8086 Mode Exceptions

#UD If source operand is not a memory location.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

LEA—Load Effective Address

Opcode	Instruction	Description
8D /*r*	LEA *r16,m*	Store effective address for *m* in register *r16*
8D /*r*	LEA *r32,m*	Store effective address for *m* in register *r32*

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 2-16. LEA Address and Operand Sizes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

Operation

```
IF OperandSize = 16 AND AddressSize = 16
   THEN
        DEST ← EffectiveAddress(SRC); (* 16-bit address *)
   ELSE IF OperandSize = 16 AND AddressSize = 32
        THEN
             temp ← EffectiveAddress(SRC); (* 32-bit address *)
             DEST ← temp[0..15]; (* 16-bit address *)
   ELSE IF OperandSize = 32 AND AddressSize = 16
        THEN
             temp ← EffectiveAddress(SRC); (* 16-bit address *)
             DEST ← ZeroExtend(temp); (* 32-bit address *)
   ELSE IF OperandSize = 32 AND AddressSize = 32
        THEN
             DEST ← EffectiveAddress(SRC); (* 32-bit address *)
   FI;
FI;
```

LEA—Load Effective Address (Continued)

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.

Protected Mode Exceptions

#UD	If source operand is not a memory location.

Real Address Mode Exceptions

#UD	If source operand is not a memory location.

Virtual 8086 Mode Exceptions

#UD	If source operand is not a memory location.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Description
C9	LEAVE	Set SP to BP, then pop BP
C9	LEAVE	Set ESP to EBP, then pop EBP

Description

Executes a return from a procedure or group of nested procedures established by an earlier ENTER instruction. The instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), releasing the stack space used by a procedure for its local variables. The old frame pointer (the frame pointer for the calling procedure that issued the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure and remove any arguments pushed onto the stack by the procedure being returned from.

Operation

```
IF StackAddressSize = 32
  THEN
      ESP ← EBP;
  ELSE (* StackAddressSize = 16*)
      SP ← BP;
FI;
IF OperandSize = 32
  THEN
      EBP ← Pop();
  ELSE (* OperandSize = 16*)
      BP ← Pop();
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.

LEAVE—High Level Procedure Exit (Continued)

Real Address Mode Exceptions

#GP If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.

Virtual 8086 Mode Exceptions

#GP(0) If the EBP register points to a location outside of the effective address space from 0 to 0FFFFH.

LES—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LFS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /2	LGDT *m16&32*	Load *m* into GDTR
0F 01 /3	LIDT *m16&32*	Load *m* into IDTR

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand is a pointer to 6 bytes of data in memory that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,LGDT/LIDT);
IF instruction is LIDT
  THEN
          IF OperandSize = 16
          THEN
              IDTR(Limit) ← SRC[0:15];
              IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
          ELSE (* 32-bit Operand Size *)
              IDTR(Limit) ← SRC[0:15];
              IDTR(Base) ← SRC[16:47];
      FI;
  ELSE (* instruction is LGDT *)
      IF OperandSize = 16
          THEN
              GDTR(Limit) ← SRC[0:15];
              GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
          ELSE (* 32-bit Operand Size *)
              GDTR(Limit) ← SRC[0:15];
              GDTR(Base) ← SRC[16:47];
      FI;
FI;
```

Flags Affected

None.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register (Continued)

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept for LIDT and LGDT

Protected Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

#UD	If source operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

LGS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /2	LLDT *r/m16*	Load segment selector *r/m16* into LDTR

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses to segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If the source operand is 0, the LDTR is marked invalid and all references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. Also, this instruction can only be executed in protected mode.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,LLDT);

IF SRC(Offset) > descriptor table limit THEN #GP(segment selector); FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ LDT THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
LDTR(SegmentSelector) ← SRC;
LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.

LLDT—Load Local Descriptor Table Register (Continued)

#GP(selector) If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.

Segment selector is beyond GDT limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP(selector) If the LDT descriptor is not present.

#PF(fault-code) If a page fault occurs.

Real Address Mode Exceptions

#UD The LLDT instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD The LLDT instruction is recognized in virtual 8086 mode.

LIDT—Load Interrupt Descriptor Table Register

See entry for LGDT/LIDT—Load Global Descriptor Table Register/Load Interrupt Descriptor Table Register.

LMSW—Load Machine Status Word

Opcode	Instruction	Description
0F 01 /6	LMSW *r/m16*	Loads *r/m16* in machine status word of CR0

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. The PE flag in the CR0 register is a sticky bit. Once set to 1, the LMSW instruction cannot be used clear this flag and force a switch back to real address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual 8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on processors more recent than the Intel 286 should use the MOV (control registers) instruction to load the machine status word.

This instruction is a serializing instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,LMSW);
CR0[0:3] ← SRC[0:3];
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

LMSW—Load Machine Status Word (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If the current privilege level is not 0.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Operation

```
IF Itanium System Environment AND External_Bus_Lock_Required AND DCR.lc
  THEN IA-32_Intercept(LOCK);
AssertLOCK#(DurationOfAccompaningInstruction)
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Lock Intercept – If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, the instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of the instruction.

Protected Mode Exceptions

#UD If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

LOCK—Assert LOCK# Signal Prefix (Continued)

Real Address Mode Exceptions

#UD If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

Virtual 8086 Mode Exceptions

#UD If the LOCK prefix is used with an instruction not listed in the "Description" section above. Other exceptions can be generated by the instruction that the LOCK prefix is being applied to.

LODS/LODSB/LODSW/LODSD—Load String Operand

Opcode	Instruction	Description
AC	LODS DS:(E)SI	Load byte at address DS:(E)SI into AL
AD	LODS DS:SI	Load word at address DS:SI into AX
AD	LODS DS:ESI	Load doubleword at address DS:ESI into EAX
AC	LODSB	Load byte at address DS:(E)SI into AL
AD	LODSW	Load word at address DS:SI into AX
AD	LODSD	Load doubleword at address DS:ESI into EAX

Description

Load a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location at the address DS:ESI. (When the operand-size attribute is 16, the SI register is used as the source-index register.) The DS segment may be overridden with a segment override prefix.

The LODSB, LODSW, and LODSD mnemonics are synonyms of the byte, word, and doubleword versions of the LODS instructions. (For the LODS instruction, "DS:ESI" must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the memory location into the AL, AX, or EAX register, the ESI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI register is incremented; if the DF flag is 1, the ESI register is decremented.) The ESI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct, because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" on page 4:337 for a description of the REP prefix.

Operation

```
IF (byte load)
  THEN
      AL ← SRC; (* byte load *)
          THEN IF DF = 0
              THEN (E)SI ← 1;
              ELSE (E)SI ← -1;
          FI;
  ELSE IF (word load)
      THEN
          AX ← SRC; (* word load *)
              THEN IF DF = 0
                  THEN SI ← 2;
                  ELSE SI ← -2;
              FI;
      ELSE (* doubleword transfer *)
          EAX ← SRC; (* doubleword load *)
```

LODS/LODSB/LODSW/LODSD—Load String Operand (Continued)

```
              THEN IF DF = 0
                  THEN ESI ← 4;
                  ELSE ESI ← -4;
              FI;
  FI;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Description
E2 *cb*	LOOP *rel8*	Decrement count; jump short if count ≠ 0
E1 *cb*	LOOPE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	Decrement count; jump short if count ≠ 0 and ZF=0

Description

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of -128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

All branches are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

Operation

```
IF AddressSize = 32
  THEN
      Count is ECX;
  ELSE (* AddressSize = 16 *)
      Count is CX;
FI;
Count ← Count - 1;

IF instruction is not LOOP
  THEN
      IF (instruction = LOOPE) OR (instruction = LOOPZ)
          THEN
             IF (ZF =1) AND (Count ≠ 0)
                 THEN BranchCond ← 1;
                 ELSE BranchCond ← 0;
             FI;
      FI;
```

LOOP/LOOP*cc*—Loop According to ECX Counter (Continued)

```
      IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
          THEN
              IF (ZF =0 ) AND (Count ≠ 0)
                  THEN BranchCond ← 1;
                  ELSE BranchCond ← 0;
              FI;
      FI;
  ELSE (* instruction = LOOP *)
      IF (Count ≠ 0)
          THEN BranchCond ← 1;
          ELSE BranchCond ← 0;
      FI;
FI;
IF BranchCond = 1
  THEN
       EIP ← EIP + SignExtend(DEST);
      IF OperandSize = 16
          THEN
              EIP ← EIP AND 0000FFFFH;
      FI;
      IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
  ELSE
      Terminate loop and continue program execution at EIP;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

IA_32_Exception Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0) If the offset jumped to is beyond the limits of the code segment.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

None.

LSL—Load Segment Limit

Opcode	Instruction	Description
0F 03 /*r*	LSL *r16,r/m16*	Load: *r16* ← segment limit, selector *r/m16*
0F 03 /*r*	LSL *r32,r/m32*	Load: *r32* ← segment limit, selector *r/m32*)

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit "raw" limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not null.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

LSL—Load Segment Limit (Continued)

Type	Name	Valid
0	Reserved	No
1	Available 16-bit TSS	Yes
2	LDT	Yes
3	Busy 16-bit TSS	Yes
4	16-bit call gate	No
5	16-bit/32-bit task gate	No
6	16-bit trap gate	No
7	16-bit interrupt gate	No
8	Reserved	No
9	Available 32-bit TSS	Yes
A	Reserved	No
B	Busy 32-bit TSS	Yes
C	32-bit call gate	No
D	Reserved	No
E	32-bit trap gate	No
F	32-bit interrupt gate	No

Operation

```
IF SRC(Offset) > descriptor table limit
   THEN ZF ← 0; FI;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
   AND (CPL > DPL) OR (RPL > DPL)
   OR Segment type is not valid for instruction
         THEN
               ZF ← 0
         ELSE
               temp ← SegmentLimit([SRC]);
               IF (G = 1)
                     THEN
                           temp ← ShiftLeft(12, temp) OR 00000FFFH;
               FI;
               IF OperandSize = 32
                     THEN
                           DEST ← temp;
                     ELSE (*OperandSize = 16*)
                           DEST ← temp AND FFFFH;
               FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is cleared to 0.

LSL—Load Segment Limit (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#UD The LSL instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD The LSL instruction is not recognized in virtual 8086 mode.

LSS—Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

LTR—Load Task Register

Opcode	Instruction	Description
0F 00 /3	LTR *r/m16*	Load *r/m16* into TR

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses to segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,LTR);
IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
  THEN #GP(segment selector);
FI;
Reat segment descriptor;
IF segment descriptor is not for an available TSS THEN #GP(segment selector); FI;
IF segment descriptor is not present THEN #NP(segment selector);
TSSsegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)
TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

LTR—Load Task Register (Continued)

#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real Address Mode Exceptions

#UD	The LTR instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD	The LTR instruction is not recognized in virtual 8086 mode.

MOV—Move

Opcode	Instruction	Description
88 /*r*	MOV *r/m8,r8*	Move *r8* to *r/m8*
89 /*r*	MOV *r/m16,r16*	Move *r16* to *r/m16*
89 /*r*	MOV *r/m32,r32*	Move *r32* to *r/m32*
8A /*r*	MOV *r8,r/m8*	Move *r/m8* to *r8*
8B /*r*	MOV *r16,r/m16*	Move *r/m16* to *r16*
8B /*r*	MOV *r32,r/m32*	Move *r/m32* to *r32*
8C /*r*	MOV *r/m16,Sreg***	Move segment register to *r/m16*
8E /*r*	MOV *Sreg,r/m16*	Move *r/m16* to segment register
A0	MOV AL,*moffs8**	Move byte at (*seg:offset*) to AL
A1	MOV AX,*moffs16**	Move word at (*seg:offset*) to AX
A1	MOV EAX,*moffs32**	Move doubleword at (*seg:offset*) to EAX
A2	MOV *moffs8**,AL	Move AL to (*seg:offset*)
A3	MOV *moffs16**,AX	Move AX to (*seg:offset*)
A3	MOV *moffs32**,EAX	Move EAX to (*seg:offset*)
B0+ *rb*	MOV *r8,imm8*	Move *imm8* to *r8*
B8+ *rw*	MOV *r16,imm16*	Move *imm16* to *r16*
B8+ *rd*	MOV *r32,imm32*	Move *imm32* to *r32*
C6 /*0*	MOV *r/m8,imm8*	Move *imm8* to *r/m8*
C7 /*0*	MOV *r/m16,imm16*	Move *imm16* to *r/m16*
C7 /*0*	MOV *r/m32,imm32*	Move *imm32* to *r/m32*

Notes:
*The *moffs8*, *moffs16*, and *moffs32* operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.
**In 32-bit mode, the assembler may require the use of the 16-bit operand size prefix (a byte with the value 66H preceding the instruction).

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the RET instruction.

MOV—Move (Continued)

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A null segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all external interrupts and traps until after the execution of the next instruction in the IA-32 System Environment. For the Itanium System Environment, MOV to SS results in a IA-32_Intercept(SystemFlag) trap after the instruction completes. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, *stack-pointer value*) before an interrupt occurs. The LSS instruction offers a more efficient method of loading the SS and ESP registers.

When moving data in 32-bit mode between a segment register and a 32-bit general-purpose register, the Pentium Pro processor does not require the use of a 16-bit operand size prefix; however, some assemblers do require this prefix. The processor assumes that the sixteen least-significant bits of the general-purpose register are the destination or source operand. When moving a value from a segment selector to a 32-bit register, the processor fills the two high-order bytes of the register with zeros.

Operation

```
DEST ← SRC;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
  THEN
      IF segment selector is null
          THEN #GP(0);
      FI;
      IF segment selector index is outside descriptor table limits
          OR segment selector's RPL ≠ CPL
          OR segment is not a writable data segment
          OR DPL ≠ CPL
              THEN #GP(selector);
      FI;
      IF segment not marked present
          THEN #SS(selector);
  ELSE
```

MOV—Move (Continued)

```
            SS ← segment selector;
            SS ← segment descriptor;
    FI;
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
    IF segment selector index is outside descriptor table limits
            OR segment is not a data or readable code segment
            OR ((segment is a data or nonconforming code segment)
                    AND (both RPL and CPL > DPL))
                            THEN #GP(selector);
            IF segment not marked present
                    THEN #NP(selector);
    ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor;
    FI;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
    THEN
            SegmentRegister ← null segment selector;
            SegmentRegister ← null segment descriptor;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	System Flag Intercept trap for Move to SS
Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.

MOV—Move (Continued)

If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

#NP If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If attempt is made to load the CS register.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If attempt is made to load the CS register.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If attempt is made to load the CS register.

MOV—Move to/from Control Registers

Opcode	Instruction	Description
0F 22 /r	MOV CR0,*r32*	Move *r32* to CR0
0F 22 /r	MOV CR2,*r32*	Move *r32* to CR2
0F 22 /r	MOV CR3,*r32*	Move *r32* to CR3
0F 22 /r	MOV CR4,*r32*	Move *r32* to CR4
0F 20 /r	MOV *r32*,CR0	Move CR0 to *r32*
0F 20 /r	MOV *r32*,CR2	Move CR2 to *r32*
0F 20 /r	MOV *r32*,CR3	Move CR3 to *r32*
0F 20 /r	MOV *r32*,CR4	Move CR4 to *r32*

Description

Moves the contents of a control register (CR0, CR2, CR3, or CR4) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See the *Intel Architecture Software Developer's Manual, Volume 3* for a detailed description of the flags and fields in the control registers.)

When loading a control register, a program should not attempt to change any of the reserved bits; that is, always set reserved bits to the value previously read.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are always 11B. The *r/m* field specifies the general-purpose register loaded or read.

These instructions have the following side effects:

- When writing to control register CR3, all non-global TLB entries are flushed (see the *Intel Architecture Software Developer's Manual, Volume 3*.
- When modifying any of the paging flags in the control registers (PE and PG in register CR0 and PGE, PSE, and PAE in register CR4), all TLB entries are flushed, including global entries. This operation is implementation specific for the Pentium Pro processor. Software should not depend on this functionality in future Intel architecture processors.
- If the PG flag is set to 1 and control register CR4 is written to set the PAE flag to 1 (to enable the physical address extension mode), the pointers (PDPTRs) in the page-directory pointers table will be loaded into the processor (into internal, non-architectural registers).
- If the PAE flag is set to 1 and the PG flag set to 1, writing to control register CR3 will cause the PDPTRs to be reloaded into the processor.
- If the PAE flag is set to 1 and control register CR0 is written to set the PG flag, the PDPTRs are reloaded into the processor.

Operation

```
IF Itanium System Environment AND Move To CR Form THEN IA-32_Intercept(INST,MOVCR);
DEST ← SRC;
```

MOV—Move to/from Control Registers (Continued)

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Additional Itanium System Environment Exceptions

IA-32_Intercept Move To CR#, Mandatory Instruction Intercept.

Move From CR#, read the virtualized control register values, CR0{15:6} return zeros.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

If an attempt is made to write a 1 to any reserved bit in CR4.

If an attempt is made to write reserved bits in the page-directory pointers table (used in the extended physical addressing mode) when the PAE flag in control register CR4 and the PG flag in control register CR0 are set to 1.

Real Address Mode Exceptions

#GP If an attempt is made to write a 1 to any reserved bit in CR4.

Virtual 8086 Mode Exceptions

#GP(0) These instructions cannot be executed in virtual 8086 mode.

MOV—Move to/from Debug Registers

Opcode	Instruction	Description
0F 21/*r*	MOV *r32,* DR0-DR3	Move debug registers to *r32*
0F 21/*r*	MOV *r32,* DR4-DR5	Move debug registers to *r32*
0F 21/*r*	MOV *r32,* DR6-DR7	Move debug registers to *r32*
0F 23 /*r*	MOV DR0-DR3, *r32*	Move *r32* to debug registers
0F 23 /*r*	MOV DR4-DR5, *r32*	Move *r32* to debug registers
0F 23 /*r*	MOV DR6-DR7,*r32*	Move *r32* to debug registers

Description

Moves the contents of two or more debug registers (DR0 through DR3, DR4 and DR5, or DR6 and DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits, regardless of the operand-size attribute. (See the *Intel Architecture Software Developer's Manual, Volume 3* for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386™ and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE set in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are always 11. The *r/m* field specifies the general-purpose register loaded or read.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,MOVDR);
IF ((DE = 1)  and (SRC or DEST = DR4 or DR5))
THEN
   #UD;
ELSE
   DEST ← SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.

MOV—Move to/from Debug Registers (Continued)

#DB If any debug register is accessed while the GD flag in debug register DR7 is set.

Real Address Mode Exceptions

#UD If the DE (debug extensions) bit of CR4 is set and a MOV instruction is executed involving DR4 or DR5.

#DB If any debug register is accessed while the GD flag in debug register DR7 is set.

Virtual 8086 Mode Exceptions

#GP(0) The debug registers cannot be loaded or read when in virtual 8086 mode.

MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String

Opcode	Instruction	Description
A4	MOVS ES:(E)DI, DS:(E)SI	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVS ES:DI,DS:SI	Move word at address DS:SI to address ES:DI
A5	MOVS ES:EDI, DS:ESI	Move doubleword at address DS:ESI to address ES:EDI
A4	MOVSB	Move byte at address DS:(E)SI to address ES:(E)DI
A5	MOVSW	Move word at address DS:SI to address ES:DI
A5	MOVSD	Move doubleword at address DS:ESI to address ES:EDI

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). The source operand specifies the memory location at the address DS:ESI and the destination operand specifies the memory location at address ES:EDI. (When the operand-size attribute is 16, the SI and DI register are used as the source-index and destination-index registers, respectively.) The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

The MOVSB, MOVSW, and MOVSD mnemonics are synonyms of the byte, word, and doubleword versions of the MOVS instructions. They are simpler to use, but provide no type or segment checking. (For the MOVS instruction, "DS:ESI" and "ES:EDI" must be explicitly specified in the instruction.)

After the transfer, the ESI and EDI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI and EDI register are incremented; if the DF flag is 1, the ESI and EDI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPNZ—Repeat Following String Operation" on "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" on page 4:337) for block moves of ECX bytes, words, or doublewords.

Operation

```
DEST ←SRC;
IF (byte move)
   THEN IF DF = 0
         THEN (E)DI ← 1;
         ELSE (E)DI ← -1;
   FI;
   ELSE IF (word move)
         THEN IF DF = 0
               THEN DI ← 2;
               ELSE DI ← -2;
```

MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String (Continued)

```
            FI;
            ELSE (* doubleword move*)
                THEN IF DF = 0
                    THEN EDI ← 4;
                    ELSE EDI ← -4;
                FI;
        FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

MOVSX—Move with Sign-Extension

Opcode	Instruction	Description
0F BE /*r*	MOVSX *r16,r/m8*	Move byte to word with sign-extension
0F BE /*r*	MOVSX *r32,r/m8*	Move byte to doubleword, sign-extension
0F BF /*r*	MOVSX *r32,r/m16*	Move word to doubleword, sign-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Operation

```
DEST ← SignExtend(SRC);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Description
0F B6 /*r*	MOVZX *r16,r/m8*	Move byte to word with zero-extension
0F B6 /*r*	MOVZX *r32,r/m8*	Move byte to doubleword, zero-extension
0F B7 /*r*	MOVZX *r32,r/m16*	Move word to doubleword, zero-extension

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute.

Operation

```
DEST ← ZeroExtend(SRC);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

MOVZX—Move with Zero-Extend (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

MUL—Unsigned Multiplication of AL, AX, or EAX

Opcode	Instruction	Description
F6 /4	MUL *r/m8*	Unsigned multiply (AX ← AL ∗ *r/m8*)
F7 /4	MUL *r/m16*	Unsigned multiply (DX:AX ← AX ∗ *r/m16*)
F7 /4	MUL *r/m32*	Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*)

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in the following table.

:

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

The AH, DX, or EDX registers (depending on the operand size) contain the high-order bits of the product. If the contents of one of these registers are 0, the CF and OF flags are cleared; otherwise, the flags are set.

Operation

```
IF byte operation
  THEN
      AX ← AL ∗ SRC
  ELSE (* word or doubleword operation *)
      IF OperandSize = 16
          THEN
              DX:AX ← AX ∗ SRC
          ELSE (* OperandSize = 32 *)
              EDX:EAX ← EAX ∗ SRC
      FI;
FI;
```

Flags Affected

The OF and CF flags are cleared to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MUL—Unsigned Multiplication of AL, AX, or EAX (Continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG *r/m8*	Two's complement negate *r/m8*
F7 /3	NEG *r/m16*	Two's complement negate *r/m16*
F7 /3	NEG *r/m32*	Two's complement negate *r/m32*

Description

Replaces the value of operand (the destination operand) with its two's complement. The destination operand is located in a general-purpose register or a memory location.

Operation

```
IF DEST = 0
   THEN CF ← 0
   ELSE CF ← 1;
FI;
DEST ←  - (DEST)
```

Flags Affected

The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

NEG—Two's Complement Negation (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

NOP—No Operation

Opcode	Instruction	Description
90	NOP	No operation

Description

Performs no operation. This instruction is a one-byte instruction that takes up space in the instruction stream but does not affect the machine context, except the EIP register.

The NOP instruction performs no operation, no registers are accessed and no faults are generated.

Flags Affected

None.

Exceptions (All Operating Modes)

None.

NOT—One's Complement Negation

Opcode	Instruction	Description
F6 /2	NOT *r/m8*	Reverse each bit of *r/m8*
F7 /2	NOT *r/m16*	Reverse each bit of *r/m16*
F7 /2	NOT *r/m32*	Reverse each bit of *r/m32*

Description

Performs a bitwise NOT operation (1's complement) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

Operation

```
DEST ← NOT DEST;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

NOT—One's Complement Negation (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

OR—Logical Inclusive OR

Opcode	Instruction	Description
0C *ib*	OR AL,*imm8*	AL OR *imm8*
0D *iw*	OR AX,*imm16*	AX OR *imm16*
0D *id*	OR EAX,*imm32*	EAXOR *imm32*
80 /1 *ib*	OR *r/m8,imm8*	*r/m8* OR *imm8*
81 /1 *iw*	OR *r/m16,imm16*	*r/m16* OR *imm16*
81 /1 *id*	OR *r/m32,imm32*	*r/m32* OR *imm32*
83 /1 *ib*	OR *r/m16,imm8*	*r/m16* OR *imm8*
83 /1 *ib*	OR *r/m32,imm8*	*r/m32* OR *imm8*
08 */r*	OR *r/m8,r8*	*r/m8* OR *r8*
09 */r*	OR *r/m16,r16*	*r/m16* OR *r16*
09 */r*	OR *r/m32,r32*	*r/m32* OR *r32*
0A */r*	OR *r8,r/m8*	*r8* OR *r/m8*
0B */r*	OR *r16,r/m16*	*r16* OR *r/m16*
0B */r*	OR *r32,r/m32*	*r32* OR *r/m32*

Description

Performs a bitwise OR operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

```
DEST ← DEST OR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

OR—Logical Inclusive OR (Continued)

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

OUT—Output to Port

Opcode	Instruction	Description
E6 *ib*	OUT *imm8*, AL	Output byte AL to *imm8* I/O port address
E7 *ib*	OUT *imm8*, AX	Output word AX to *imm8* I/O port address
E7 *ib*	OUT *imm8*, EAX	Output doubleword EAX to *imm8* I/O port address
EE	OUT DX, AL	Output byte AL to I/O port address in DX
EF	OUT DX, AX	Output word AX to I/O port address in DX
EF	OUT DX, EAX	Output doubleword EAX to I/O port address in DX

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

In the Itanium System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize TLBs in the Itanium architecture to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing OUT instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
  THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
      IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
          THEN #GP(0);
      FI;
  ELSE ( * Real-address mode or protected mode with CPL ≤ IOPL *)
```

OUT—Output to Port (Continued)

```
    (* or virtual-8086 mode with all I/O permission bits for I/O port cleared *)
FI;
IF (Itanium_System_Environment) THEN
    DEST_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
    DEST_PA = translate(DEST_VA);
    [DEST_PA] ← SRC; (* Writes to selected I/O port *)
FI;

memory_fence();
[DEST_PA] ← SRC; (* Writes to selected I/O port *)
memory_fence();
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault
IA_32_Exception	Debug traps for data breakpoints and single step
IA_32_Exception	Alignment faults
#GP(0)	Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode	Instruction	Description
6E	OUTS DX, DS:(E)SI	Output byte at address DS:(E)SI to I/O port in DX
6F	OUTS DX, DS:SI	Output word at address DS:SI to I/O port in DX
6F	OUTS DX, DS:ESI	Output doubleword at address DS:ESI to I/O port in DX
6E	OUTSB	Output byte at address DS:(E)SI to I/O port in DX
6F	OUTSW	Output word at address DS:SI to I/O port in DX
6F	OUTSD	Output doubleword at address DS:ESI to I/O port in DX

Description

Copies data from the second operand (source operand) to the I/O port specified with the first operand (destination operand). The source operand is a memory location at the address DS:ESI. (When the operand-size attribute is 16, the SI register is used as the source-index register.) The DS register may be overridden with a segment override prefix.

The destination operand must be the DX register, allowing I/O port addresses from 0 to 65,535 to be accessed. When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

The OUTSB, OUTSW and OUTSD mnemonics are synonyms of the byte, word, and doubleword versions of the OUTS instructions. (For the OUTS instruction, "DS:ESI" must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the memory location to the I/O port, the ESI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the ESI register is incremented; if the DF flag is 1, the EDI register is decremented.) The ESI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" on page 4:337 for a description of the REP prefix.

After an OUTS, OUTSB, OUTSW, or OUTSD instruction is executed, the processor waits for the acknowledgment of the OUT transaction before beginning to execute the next instruction. Note that the next instruction may be prefetched, even if the OUT transaction has not completed.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space.

I/O transactions are performed after all prior data memory operations. No subsequent data memory operations can pass an I/O transaction.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (Continued)

In the Itanium System Environment, I/O port references are mapped into the 64-bit virtual address pointed to by the IOBase register, with four ports per 4K-byte virtual page. Operating systems can utilize TLBs in the Itanium architecture to grant or deny permission to any four I/O ports. The I/O port space can be mapped into any arbitrary 64-bit physical memory location by operating system code. If CFLG.io is 1 and CPL>IOPL, the TSS is consulted for I/O permission. If CFLG.io is 0 or CPL<=IOPL, permission is granted regardless of the state of the TSS I/O permission bitmap (the bitmap is not referenced).

If the referenced I/O port is mapped to an unimplemented virtual address (via the I/O Base register) or if data translations are disabled (PSR.dt is 0) a GPFault is generated on the referencing OUTS instruction.

Operation

```
IF ((PE = 1) AND ((VM = 1) OR (CPL > IOPL)))
  THEN (* Protected mode or virtual-8086 mode with CPL > IOPL *)
      IF (CFLG.io AND Any I/O Permission Bit for I/O port being accessed = 1)
          THEN #GP(0);
      FI;
  ELSE ( * I/O operation is allowed *)
FI;

IF (Itanium_System_Environment) THEN
  DEST_VA = IOBase | (Port{15:2}<<12) | Port{11:0};
  DEST_PA = translate(DEST_VA);
  [DEST_PA] ← SRC; (* Writes to selected I/O port *)
FI;
memory_fence();
[DEST_PA] ← SRC; (* Writes to selected I/O port *)
memory_fence();

IF (byte operation)
      THEN IF DF = 0
              THEN (E)DI ← 1;
              ELSE (E)DI ← -1;
          FI;
          ELSE IF (word operation)
              THEN IF DF = 0
                  THEN DI ← 2;
                  ELSE DI ← -2;
              FI;
              ELSE (* doubleword operation *)
                  THEN IF DF = 0
                      THEN EDI ← 4;
                      ELSE EDI ← -4;
                  FI;
          FI;
      FI;
FI;
```

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port (Continued)

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA_32_Exception Debug traps for data breakpoints and single step

IA_32_Exception Alignment faults

#GP(0) Referenced Port is to an unimplemented virtual address or PSR.dt is zero.

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1 **and when CFLG.io is 1**.

If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the limit of the ES segment.

If the ES register contains a null segment selector.

If an illegal memory operand effective address in the ES segments is given.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP *m16*	Pop top of stack into *m16*; increment stack pointer
8F /0	POP *m32*	Pop top of stack into *m32*; increment stack pointer
58+ *rw*	POP *r16*	Pop top of stack into *r16*; increment stack pointer
58+ *rd*	POP *r32*	Pop top of stack into *r32*; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

Description

Loads the value from the top of the procedure stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The current address-size attribute for the stack segment and the operand-size attribute determine the amount the stack pointer is incremented (see the "Operation" below). For example, if 32-bit addressing and operands are being used, the ESP register (stack pointer) is incremented by 4 and, if 16-bit addressing and operands are being used, the SP register (stack pointer for 16-bit addressing) is incremented by 2. The B flag in the stack segment's segment descriptor determines the stack's address-size attribute.

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register, use the RET instruction.

A POP SS instruction inhibits all external interrupts, including the NMI interrupt, and traps until after execution of the next instruction. **in the IA-32 System Environment. For the Itanium System Environment, POP SS results in an IA-32_Intercept(SystemFlag) trap after the instruction completes.**This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, *stack-pointer value*) before an interrupt occurs. The LSS instruction offers a more efficient method of loading the SS and ESP registers.

POP—Pop a Value from the Stack (Continued)

This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instructions computes the effective address of the operand after it increments the ESP register.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

Operation

```
IF StackAddrSize = 32
  THEN
      IF OperandSize = 32
          THEN
              DEST ← SS:ESP; (* copy a doubleword *)
              ESP ← ESP + 4;
          ELSE (* OperandSize = 16*)
              DEST ← SS:ESP; (* copy a word *)
          ESP ← ESP + 2;
      FI;
  ELSE (* StackAddrSize = 16* )
      IF OperandSize = 16
          THEN
              DEST ← SS:SP; (* copy a word *)
              SP ← SP + 2;
          ELSE (* OperandSize = 32 *)
              DEST ← SS:SP; (* copy a doubleword *)
              SP ← SP + 4;
      FI;
FI;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
  THEN
      IF segment selector is null
          THEN #GP(0);
      FI;
      IF segment selector index is outside descriptor table limits
          OR segment selector's RPL ≠ CPL
          OR segment is not a writable data segment
          OR DPL ≠ CPL
              THEN #GP(selector);
      FI;
      IF segment not marked present
          THEN #SS(selector);
  ELSE
      SS ← segment selector;
      SS ← segment descriptor;
```

POP—Pop a Value from the Stack (Continued)

```
    FI;
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
    IF segment selector index is outside descriptor table limits
        OR segment is not a data or readable code segment
        OR ((segment is a data or nonconforming code segment)
            AND (both RPL and CPL > DPL))
                THEN #GP(selector);
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
    THEN
        SegmentRegister ← null segment selector;
        SegmentRegister ← null segment descriptor;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	System Flag Intercept trap for POP SS
Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector. If the destination operand is in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits. If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL. If the SS register is being loaded and the segment pointed to is a nonwritable data segment.

POP—Pop a Value from the Stack (Continued)

If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.

If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#SS(0) If the current top of stack is not within the stack segment.

If a memory operand effective address is outside the SS segment limit.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

#NP If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference is made while alignment checking is enabled.

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Description
61	POPA	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

Description

Pops doublewords (POPAD) or words (POPA) from the procedure stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the current operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded (see the “Operation” below).

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used. Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

Operation

```
IF OperandSize = 32 (* instruction = POPAD *)
THEN
   EDI ← Pop();
   ESI ← Pop();
   EBP ← Pop();
   increment ESP by 4 (* skip next 4 bytes of stack *)
   EBX ← Pop();
   EDX ← Pop();
   ECX ← Pop();
   EAX ← Pop();
ELSE (* OperandSize = 16, instruction = POPA *)
   DI ← Pop();
   SI ← Pop();
   BP ← Pop();
   increment ESP by 2 (* skip next 2 bytes of stack *)
   BX ← Pop();
   DX ← Pop();
   CX ← Pop();
   AX ← Pop();
FI;
```

Flags Affected

None.

POPA/POPAD—Pop All General-Purpose Registers (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0) If the starting or ending stack address is not within the stack segment.

#PF(fault-code) If a page fault occurs.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

POPF/POPFD—Pop Stack into EFLAGS Register

Opcode	Instruction	Description
9D	POPF	Pop top of stack into EFLAGS
9D	POPFD	Pop top of stack into EFLAGS

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register. (These instructions reverse the operation of the PUSHF/PUSHFD instructions.)

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

The effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the non-reserved flags in the EFLAGS register except the VIP and VIF flags can be modified. The VIP and VIF flags are cleared.

When operating in protected mode, but with a privilege level greater an 0, all the flags can be modified except the IOPL field and the VIP and VIF flags. Here, the IOPL flags are masked and the VIP and VIF flags are cleared.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are masked. If the IOPL is less than 3, the POPF/POPFD instructions cause a general protection exception (#GP).

The IOPL is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the IOPL. (Real-address mode is equivalent to privilege level 0.) If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

Operation

```
OLD_IF <- IF; OLD_AC <- AC; OLD_TF <- TF;
IF CR0.PE = 0 (*Real Mode *)
  THEN
      IF OperandSize = 32;
          THEN
              EFLAGS ← Pop();
              (* All non-reserved flags except VM, RF, VIP and VIF can be modified; *)
              ELSE (* OperandSize = 16 *)
              EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
      FI;
  ELSE (*In Protected Mode *)
```

POPF/POPFD—Pop Stack into EFLAGS Register (Continued)

```
IF VM=0 (* Not in Virtual-8086 Mode *)
    THEN
        IF CPL=0
            THEN
                IF OperandSize = 32;
                    THEN
                        EFLAGS ← Pop();
                        (* All non-reserved flags except VM, RF, VIP and VIF can be *)
                        (* modified; *)
                    ELSE (* OperandSize = 16 *)
                    EFLAGS[15:0] ← Pop(); (* All non-reserved flags can be modified; *)
                FI;
            ELSE (* CPL > 0 *)
                IF OperandSize = 32;
                    THEN
                        EFLAGS ← Pop()
                        (* All non-reserved bits except IOPL, RF, VM, VIP, and VIF can *)
                        (* be modified; *)
                        (* IOPL is masked *)
                    ELSE (* OperandSize = 16 *)
                        EFLAGS[15:0] ← Pop();
                        (* All non-reserved bits except IOPL can be modified; IOPL is
masked *)
                FI;
        FI;
    ELSE  (* In Virtual-8086 Mode *)
        IF IOPL=3
        THEN
            IF OperandSize=32
                THEN
                    EFLAGS ← Pop()
                    (* All non-reserved bits except VM, RF, IOPL, VIP, and VIF *)
                    (* can be modified; VM, RF, IOPL, VIP, and VIF are masked*)
                ELSE
                    EFLAGS[15:0] ← Pop()
                    (* All non-reserved bits except IOPL can be modified; IOPL is *)
                        (* masked *)
            FI;
        ELSE (* IOPL < 3 *)
            IF CR4.VME = 0
                THEN #GP(0);
                ELSE
                    IF ((OperandSize = 32) OR (STACK.TF = 1) OR (EFLAGS.VIP = 1
                        AND STACK.IF = 1)
                        THEN #GP(0);
                        ELSE
                            TempFlags <- pop();
                            FLAGS <- TempFlags; (*IF and IOPL bits are unchanged*)
                            EFLAGS.VIF <- TempFlags.IF;
                        FI;
                FI;
            FI;
```

POPF/POPFD—Pop Stack into EFLAGS Register (Continued)

```
        FI;
FI;

IF(Itanium System Environment AND (AC, TF != OLD_AC, OLD_TF)
    THEN IA-32_Intercept(System_Flag,POPF);
IF Itanium System Environment AND CFLG.ii AND IF != OLD_IF
    THEN IA-32_Intercept(System_Flag,POPF);
```

Flags Affected

All flags except the reserved bits.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state or if the AC, RF or TF changes state.

Protected Mode Exceptions

#SS(0) If the top of stack is not within the stack segment.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the I/O privilege level is less than 3.

If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.

#SS(0) If a memory operand effective address is outside the SS segment limit.

PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	Description
FF /6	PUSH *r/m16*	Push *r/m16*
FF /6	PUSH *r/m32*	Push *r/m32*
50+*rw*	PUSH *r16*	Push *r16*
50+*rd*	PUSH *r32*	Push *r32*
6A	PUSH *imm8*	Push *imm8*
68	PUSH *imm16*	Push *imm16*
68	PUSH *imm32*	Push *imm32*
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS

Description

Decrements the stack pointer and then stores the source operand on the top of the procedure stack. The current address-size attribute for the stack segment and the operand-size attribute determine the amount the stack pointer is decremented (see the "Operation" below). For example, if 32-bit addressing and operands are being used, the ESP register (stack pointer) is decremented by 4 and, if 16-bit addressing and operands are being used, the SP register (stack pointer for 16-bit addressing) is decremented by 2. Pushing 16-bit operands when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF StackAddrSize = 32
THEN
  IF OperandSize = 32
      THEN
          ESP ← ESP – 4;
          SS:ESP ← SRC; (* push doubleword *)
      ELSE (* OperandSize = 16*)
          ESP ← ESP – 2;
          SS:ESP ← SRC; (* push word *)
  FI;
ELSE (* StackAddrSize = 16*)
```

PUSH—Push Word or Doubleword Onto the Stack (Continued)

```
    IF OperandSize = 16
        THEN
            SP ← SP – 2;
             SS:SP ← SRC; (* push word *)
        ELSE (* OperandSize = 32*)
            SP ← SP – 4;
            SS:SP ← SRC; (* push doubleword *)
    FI;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

If the new value of the SP or ESP register is outside the stack segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

PUSH—Push Word or Doubleword Onto the Stack (Continued)

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

For Intel architecture processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Description
60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Description

Push the contents of the general-purpose registers onto the procedure stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). (These instructions perform the reverse operation of the POPA/POPAD instructions.) The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF OperandSize = 32 (* PUSHAD instruction *)
  THEN
      Temp ← (ESP);
      Push(EAX);
      Push(ECX);
      Push(EDX);
      Push(EBX);
      Push(Temp);
      Push(EBP);
      Push(ESI);
      Push(EDI);
  ELSE (* OperandSize = 16, PUSHA instruction *)
      Temp ← (SP);
      Push(AX);
      Push(CX);
      Push(DX);
      Push(BX);
      Push(Temp);
      Push(BP);
      Push(SI);
      Push(DI);
FI;
```

PUSHA/PUSHAD—Push All General-Purpose Registers (Continued)

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0) If the starting or ending stack address is outside the stack segment limit.

#PF(fault-code) If a page fault occurs.

Real Address Mode Exceptions

#GP If the ESP or SP register contains 7, 9, 11, 13, or 15.

Virtual 8086 Mode Exceptions

#GP(0) If the ESP or SP register contains 7, 9, 11, 13, or 15.

#PF(fault-code) If a page fault occurs.

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack

Opcode	Instruction	Description
9C	PUSHF	Push EFLAGS
9C	PUSHFD	Push EFLAGS

Description

Decrement the stack pointer by 4 (if the current operand-size attribute is 32) and push the entire contents of the EFLAGS register onto the procedure stack or decrement the stack pointer by 2 (if the operand-size attribute is 16) push the lower 16 bits of the EFLAGS register onto the stack. (These instructions reverse the operation of the POPF/POPFD instructions.)

When copying the entire EFLAGS register to the stack, bits 16 and 17, called the VM and RF flags, are not copied. Instead, the values for these flags are cleared in the EFLAGS image stored on the stack.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

When the I/O privilege level (IOPL) is less than 3 in virtual-8086 mode, the PUSHF/PUSHFD instructions causes a general protection exception (#GP). The IOPL is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the IOPL. (Real-address mode is equivalent to privilege level 0.) If a PUSHF/PUSHFD instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when the PUSHA/PUSHAD instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
  THEN
      IF OperandSize = 32
          THEN
              push(EFLAGS AND 00FCFFFFH);
              (* VM and RF EFLAG bits are cleared in image stored on the stack*)
          ELSE
              push(EFLAGS); (* Lower 16 bits only *)
      FI;
  ELSE (* In Virtual-8086 Mode *)
      IF IOPL=3
          THEN
              IF OperandSize = 32
```

PUSHF/PUSHFD—Push EFLAGS Register onto the Stack (Continued)

```
                THEN push(EFLAGS AND 0FCFFFFH);
                (* VM and RF EFLAGS bits are cleared in image stored on the stack*)
                ELSE push(EFLAGS); (* Lower 16 bits only *)
            FI;
        ELSE (*IOPL < 3*)
            IF OperandSize =32 OR CR$.VME=0
                THEN #GP(0); (* Trap to virtual-8086 monitor *)
                ELSE
                    TempFlags <- FLAGS OR 3000H; (*Set IOPL bits to 11B or IOPL 3 *)
                    TempFlags.IF <- EFLAGS.VIF;
                    push(TempFlags);
            FI;
    FI;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#SS(0) If the new value of the ESP register is outside the stack segment boundary.

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0) If the I/O privilege level is less than 3.

RCL/RCR/ROL/ROR-—Rotate

Opcode	Instruction	Description
D0 /2	RCL *r/m8*,1	Rotate 9 bits (CF,*r/m8*) left once
D2 /2	RCL *r/m8*,CL	Rotate 9 bits (CF,*r/m8*) left CL times
C0 /2 *ib*	RCL *r/m8,imm8*	Rotate 9 bits (CF,*r/m8*) left *imm8* times
D1 /2	RCL *r/m16*,1	Rotate 17 bits (CF,*r/m16*) left once
D3 /2	RCL *r/m16*,CL	Rotate 17 bits (CF,*r/m16*) left CL times
C1 /2 *ib*	RCL *r/m16,imm8*	Rotate 17 bits (CF,*r/m16*) left *imm8* times
D1 /2	RCL *r/m32*,1	Rotate 33 bits (CF,*r/m32*) left once
D3 /2	RCL *r/m32*,CL	Rotate 33 bits (CF,*r/m32*) left CL times
C1 /2 *ib*	RCL *r/m32,imm8*	Rotate 33 bits (CF,*r/m32*) left *imm8* times
D0 /3	RCR *r/m8*,1	Rotate 9 bits (CF,*r/m8*) right once
D2 /3	RCR *r/m8*,CL	Rotate 9 bits (CF,*r/m8*) right CL times
C0 /3 *ib*	RCR *r/m8,imm8*	Rotate 9 bits (CF,*r/m8*) right *imm8* times
D1 /3	RCR *r/m16*,1	Rotate 17 bits (CF,*r/m16*) right once
D3 /3	RCR *r/m16*,CL	Rotate 17 bits (CF,*r/m16*) right CL times
C1 /3 *ib*	RCR *r/m16,imm8*	Rotate 17 bits (CF,*r/m16*) right *imm8* times
D1 /3	RCR *r/m32*,1	Rotate 33 bits (CF,*r/m32*) right once
D3 /3	RCR *r/m32*,CL	Rotate 33 bits (CF,*r/m32*) right CL times
C1 /3 *ib*	RCR *r/m32,imm8*	Rotate 33 bits (CF,*r/m32*) right *imm8* times
D0 /0	ROL *r/m8*,1	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*,CL	Rotate 8 bits *r/m8* left CL times
C0 /0 *ib*	ROL *r/m8,imm8*	Rotate 8 bits *r/m8* left *imm8* times
D1 /0	ROL *r/m16*,1	Rotate 16 bits *r/m16* left once
D3 /0	ROL *r/m16*,CL	Rotate 16 bits *r/m16* left CL times
C1 /0 *ib*	ROL *r/m16,imm8*	Rotate 16 bits *r/m16* left *imm8* times
D1 /0	ROL *r/m32*,1	Rotate 32 bits *r/m32* left once
D3 /0	ROL *r/m32*,CL	Rotate 32 bits *r/m32* left CL times
C1 /0 *ib*	ROL *r/m32,imm8*	Rotate 32 bits *r/m32* left *imm8* times
D0 /1	ROR *r/m8*,1	Rotate 8 bits *r/m8* right once
D2 /1	ROR *r/m8*,CL	Rotate 8 bits *r/m8* right CL times
C0 /1 *ib*	ROR *r/m8,imm8*	Rotate 8 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m16*,1	Rotate 16 bits *r/m16* right once
D3 /1	ROR *r/m16*,CL	Rotate 16 bits *r/m16* right CL times
C1 /1 *ib*	ROR *r/m16,imm8*	Rotate 16 bits *r/m16* right *imm8* times
D1 /1	ROR *r/m32*,1	Rotate 32 bits *r/m32* right once
D3 /1	ROR *r/m32*,CL	Rotate 32 bits *r/m32* right CL times
C1 /1 *ib*	ROR *r/m32,imm8*	Rotate 32 bits *r/m32* right *imm8* times

RCL/RCR/ROL/ROR—Rotate (Continued)

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases. For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

Operation

```
SIZE ← OperandSize
CASE (determine count) OF
  SIZE = 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
  SIZE = 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
  SIZE = 32:    tempCOUNT ← COUNT AND 1FH;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
      tempCF ← MSB(DEST);
      DEST ← (DEST ∗ 2) + tempCF;
      tempCOUNT ← tempCOUNT - 1;
  OD;
ELIHW;
CF ← tempCF;
IF COUNT = 1
  THEN OF ← MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
(* ROR instruction operation *)
WHILE (tempCOUNT ≠ 0)
  DO
      tempCF ← LSB(SRC);
```

```
            DEST ← (DEST / 2) + (tempCF ∗ $2^{SIZE}$);
            tempCOUNT ← tempCOUNT - 1;
    OD;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
    ELSE OF is undefined;
FI;
(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
            tempCF ← MSB(DEST);
            DEST ← (DEST ∗ 2) + tempCF;
            tempCOUNT ← tempCOUNT - 1;
    OD;
ELIHW;
CF ← tempCF;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* RCR instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
            tempCF ← LSB(SRC);
            DEST ← (DEST / 2) + (tempCF * $2^{SIZE}$);
            tempCOUNT ← tempCOUNT - 1;
    OD;
IF COUNT = 1
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see "Description" above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

RCL/RCR/ROL/ROR—Rotate (Continued)

Protected Mode Exceptions

#GP(0) If the source operand is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

The 8086 does not mask the rotation count. All Intel architecture processors from the Intel386™ processor on do mask the rotation count in all operating modes.

RDMSR—Read from Model Specific Register

Opcode	Instruction	Description
0F 32	RDMSR	Load MSR specified by ECX into EDX:EAX

Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If less than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

See model-specific instructions for all the MSRs that can be written to with this instruction and their addresses

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,RDMSR);
EDX:EAX ← MSR[ECX];
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the value in ECX specifies a reserved or unimplemented MSR address.

Real Address Mode Exceptions

#GP	If the current privilege level is not 0 If the value in ECX specifies a reserved or unimplemented MSR address.

RDMSR—Read from Model Specific Register (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	The RDMSR instruction is not recognized in virtual 8086 mode.

Intel Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the Intel architecture with the Pentium processor. Execution of this instruction by an Intel architecture processor earlier than the Pentium processor results in an invalid opcode exception #UD.

RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	Description
0F 33	RDPMC	Read performance-monitoring counter specified by ECX into EDX:EAX

Description

Loads the contents of the N-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order N-32 bits of the counter and the EAX register is loaded with the low-order 32 bits.

The RDPMC instruction allows application code running at a privilege level of 1, 2, or 3 to read the performance-monitoring counters if the PCE flag in the CR4 register is set for IA-32 System Environment operation or in the Itanium System Environment if the performance counters have been configured as user level counters. This instruction is provided to allow performance monitoring by application code without incurring the overhead of a call to an operating-system procedure.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads.

The RDPMC instruction does not serialize instruction execution. That is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must use a serializing instruction (such as the CPUID instruction) before and/or after the execution of the RDPCM instruction.

The RDPMC instruction can execute in 16-bit addressing mode or virtual 8086 mode; however, the full contents of the ECX register are used to determine the counter to access and a full N-bit result is returned (the low-order 32 bits in the EAX register and the high-order N-32 bits in the EDX register).

Operation

```
IF (ECX != Implemented Counters) THEN #GP(0)
IF (Itanium System Environment)
THEN
  SECURED = PSR.sp || CR4.pce==0;
  IF ((PSR.cpl ==0) || (PSR.cpl!=0 && ~PMC[ECX].pm && ~SECURED)))
      THEN
          EDX:EAX ← PMD[ECX+4];
      ELSE
          #GP(0)
  FI;
ELSE
  IF ((CR4.PCE = 1 OR ((CR4.PCE = 0 ) AND (CPL=0)))
      THEN
          EDX:EAX ← PMD[ECX+4];
      ELSE (* CR4.PCE is 0 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
```

RDPMC—Read Performance-Monitoring Counters (Continued)

```
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
#GP(0)	If the current privilege level is not 0 and the selected PMD register's PM bit is 1, or if PSR.sp is 1.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear /*In IA-32 System Environment*/. If the value in the ECX register does not match an implemented performance counter.

Real Address Mode Exceptions

#GP	If the PCE flag in the CR4 register is clear. /*In the IA-32 System Environment*/ If the value in the ECX register does not match an implemented performance counter.

Virtual 8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear. /*In the IA-32 System Environment*/ If the value in the ECX register does not match an implemented performance counter.

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX

Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

In the IA-32 System Environment, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. The time-stamp counter can also be read with the RDMSR instruction.

In the Itanium System Environment, PSR.si and CR4.TSD restricts the use of the RDTSC instruction. When PSR.si is clear and CR4.TSD is clear, the RDTSC instruction can be executed at any privilege level; when PSR.si is set or CR4.TSD is set, the instruction can only be executed at privilege level 0.

The RDTSC instruction is not serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the Intel architecture in the Pentium processor.

Operation

```
IF (IA-32 System Environement)
  IF (CR4.TSD = 0) OR ((CR4.TSD = 1) AND (CPL=0))
      THEN
          EDX:EAX ← TimeStampCounter;
      ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
ELSE /*Itanium System Environment*/
  SECURED = PSR.si || CR4.TSD;
  IF (!SECURED) OR (SECURED AND (CPL=0))
      THEN
          EDX:EAX ← TimeStampCounter;
      ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
          #GP(0)
  FI;
FI;
```

Flags Affected

None.

RDTSC—Read Time-Stamp Counter (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
#GP(0)	If PSR.si is 1 or CR4.TSD is 1 and the CPL is greater than 0.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0. /*For the IA-32 System Environment only*/

Real Address Mode Exceptions

#GP	If the TSD flag in register CR4 is set. /*For the IA-32 System Environment only*/

Virtual 8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set. /*For the IA-32 System Environment only*/

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

F3 6C	REP INS *r/m8*, DX	Input ECX bytes from port DX into ES:[EDI]
F3 6D	REP INS *r/m16*,DX	Input ECX words from port DX into ES:[EDI]
F3 6D	REP INS *r/m32*,DX	Input ECX doublewords from port DX into ES:[EDI]
F3 A4	REP MOVS *m8,m8*	Move ECX bytes from DS:[ESI] to ES:[EDI]
F3 A5	REP MOVS *m16,m16*	Move ECX words from DS:[ESI] to ES:[EDI]
F3 A5	REP MOVS *m32,m32*	Move ECX doublewords from DS:[ESI] to ES:[EDI]
F3 6E	REP OUTS DX,*r/m8*	Output ECX bytes from DS:[ESI] to port DX
F3 6F	REP OUTS DX,*r/m16*	Output ECX words from DS:[ESI] to port DX
F3 6F	REP OUTS DX,*r/m32*	Output ECX doublewords from DS:[ESI] to port DX
F3 AC	REP LODS AL	Load ECX bytes from DS:[ESI] to AL
F3 AD	REP LODS AX	Load ECX words from DS:[ESI] to AX
F3 AD	REP LODS EAX	Load ECX doublewords from DS:[ESI] to EAX
F3 AA	REP STOS *m8*	Fill ECX bytes at ES:[EDI] with AL
F3 AB	REP STOS *m16*	Fill ECX words at ES:[EDI] with AX
F3 AB	REP STOS *m32*	Fill ECX doublewords at ES:[EDI] with EAX
F3 A6	REPE CMPS *m8,m8*	Find nonmatching bytes in ES:[EDI] and DS:[ESI]
F3 A7	REPE CMPS *m16,m16*	Find nonmatching words in ES:[EDI] and DS:[ESI]
F3 A7	REPE CMPS *m32,m32*	Find nonmatching doublewords in ES:[EDI] and DS:[ESI]
F3 AE	REPE SCAS *m8*	Find non-AL byte starting at ES:[EDI]
F3 AF	REPE SCAS *m16*	Find non-AX word starting at ES:[EDI]
F3 AF	REPE SCAS *m32*	Find non-EAX doubleword starting at ES:[EDI]
F2 A6	REPNE CMPS *m8,m8*	Find matching bytes in ES:[EDI] and DS:[ESI]
F2 A7	REPNE CMPS *m16,m16*	Find matching words in ES:[EDI] and DS:[ESI]
F2 A7	REPNE CMPS *m32,m32*	Find matching doublewords in ES:[EDI] and DS:[ESI]
F2 AE	REPNE SCAS *m8*	Find AL, starting at ES:[EDI]
F2 AF	REPNE SCAS *m16*	Find AX, starting at ES:[EDI]
F2 AF	REPNE SCAS *m32*	Find EAX, starting at ES:[EDI]

Description

Repeats a string instruction the number of times specified in the count register (ECX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

All of these repeat prefixes cause the associated instruction to be repeated until the count in register ECX is decremented to 0 (see the following table). The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the ECX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

Table 2-17. Repeat Conditions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	None
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a page fault occurs during CMPS or SCAS instructions that are prefixed with REPNE, the EFLAGS value may NOT be restored to the state prior to the execution of the instruction. Since SCAS and CMPS do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

Operation

```
IF AddressSize = 16
  THEN
        use CX for CountReg;
  ELSE (* AddressSize = 32 *)
        use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
  DO
        service pending interrupts (if any);
        execute associated string instruction;
        CountReg ← CountReg - 1;
```

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix (Continued)

```
        IF CountReg = 0
            THEN exit WHILE loop
        FI;
        IF (repeat prefix is REPZ or REPE) AND (ZF=0)
        OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
            THEN exit WHILE loop
        FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 *iw*	RET *imm16*	Near return to calling procedure and pop *imm16* bytes from stack
CA *iw*	RET *imm16*	Far return to calling procedure and pop *imm16* bytes from stack

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed.

The RET instruction can be used to execute three different types of returns:

- Near return – A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return – A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return – A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the procedure stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the procedure stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

RET—Return from Procedure (Continued)

Operation

```
(* Near return *)
IF instruction = near return
  THEN;
    IF OperandSize = 32
      THEN
        IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
        EIP ← Pop();
      ELSE (* OperandSize = 16 *)
        IF top 6 bytes of stack not within stack limits
          THEN #SS(0)
        FI;
        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits THEN #GP(0); FI;
        EIP ← tempEIP;
    FI;
  IF instruction has immediate operand
    THEN IF StackAddressSize=32
      THEN
        ESP ← ESP + SRC;
      ELSE (* StackAddressSize=16 *)
        SP ← SP + SRC;
    FI;
  FI;
  IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) OR (PE = 1 AND VM = 1)) AND instruction = far return
  THEN;
    IF OperandSize = 32
      THEN
        IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
      ELSE (* OperandSize = 16 *)
        IF top 6 bytes of stack not within stack limits THEN #SS(0); FI;
        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits THEN #GP(0); FI;
        EIP ← tempEIP;
        CS ← Pop(); (* 16-bit pop *)
    FI;
  IF instruction has immediate operand THEN SP ← SP + (SRC AND FFFFH); FI;
  IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);
FI;

(* Protected mode, not virtual 8086 mode *)
IF (PE = 1 AND VM = 0) AND instruction = far RET
  THEN
    IF OperandSize = 32
      THEN
```

RET—Return from Procedure (Continued)

```
              IF second doubleword on stack is not within stack limits THEN #SS(0); FI;
          ELSE (* OperandSize = 16 *)
              IF second word on stack is not within stack limits THEN #SS(0); FI;
      FI;
  IF return code segment selector is null THEN GP(0); FI;
  IF return code segment selector addrsses descriptor beyond diescriptor table limit
      THEN GP(selector; FI;
  Obtain descriptor to which return code segment selector points from descriptor table
  IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
  if return code segment selector RPL < CPL THEN #GP(selector); FI;
  IF return code segment descriptor is condorming
      AND return code segment DPL > return code segment selector RPL
          THEN #GP(selector); FI;
  IF return code segment descriptor is not present THEN #NP(selector); FI:
  IF return code segment selector RPL > CPL
      THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
      ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
  FI;
END;FI;

RETURN-SAME-PRIVILEGE-LEVEL:
  IF the return instruction pointer is not within ther return code segment limit
      THEN #GP(0);
  FI;
  IF OperandSize=32
      THEN
          EIP ← Pop();
          CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
          ESP ← ESP + SRC;
      ELSE (* OperandSize=16 *)
          EIP ← Pop();
          EIP ← EIP AND 0000FFFFH;
          CS ← Pop(); (* 16-bit pop *)
          ESP ← ESP + SRC;
  FI;
  IF Itanium System Environment AND PSR.tb THEN IA_32_Exception(Debug);


RETURN-OUTER-PRIVILEGE-LEVEL:
  IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize=32)
      OR top (8 + SRC) bytes of stack are not within stack limits (OperandSize=16)
          THEN #SS(0); FI;
  FI;
  Read return segment selector;
  IF stack segment selector is null THEN #GP(0); FI;
  IF return stack segment selector index is not within its descriptor table limits
          THEN #GP(selector); FI;
  Read segment descriptor pointed to by return segment selector;
  IF stack segment selector RPL ≠ RPL of the return code segment selector
      OR stack segment is not a writable data segment
      OR stack segment descriptor DPL ≠ RPL of the return code segment selector
          THEN #GP(selector); FI;
```

RET—Return from Procedure (Continued)

```
        IF stack segment not present THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit THEN #GP(0); FI:
 CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize=32
        THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                 (* segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                ESP ← ESP + SRC;
                tempESP ← Pop();
                tempSS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                 (* segment descriptor information also loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        ELSE (* OperandSize=16 *)
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                ESP ← ESP + SRC;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
                 (* segment descriptor information also loaded *)
                ESP ← tempESP;
                SS ← tempSS;
FI;
FOR each of segment register (ES, FS, GS, and DS)
        DO;
                IF segment register points to data or non-conforming code segment
                AND CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
                        THEN (* segment register invalid *)
                                SegmentSelector/Descriptor ← 0; (* null segment selector *)
                FI;
        OD;
For each of ES, FS, GS, and DS
DO
        IF segment descriptor indicates the segment is not a data or
                        readable code segment
                OR if the segment is a data or non-conforming code segment and the segment
                        descriptor's DPL < CPL or RPL of code segment's segment selector
                        THEN
                                segment selector register ← null selector;
OD;
```

Flags Affected

None.

RET—Return from Procedure (Continued)

Additional Itanium System Environment Exceptions

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA_32_Exception Taken Branch Debug Exception if PSR.tb is 1

Protected Mode Exceptions

#GP(0) If the return code or stack segment selector null.

If the return instruction pointer is not within the return code segment limit

#GP(selector) If the RPL of the return code segment selector is less then the CPL.

If the return code or stack segment selector index is not within its descriptor table limits.

If the return code segment descriptor does not indicate a code segment.

If the return code segment is non-conforming and the segment selector’s DPL is not equal to the RPL of the code segment’s segment selector

If the return code segment is conforming and the segment selector’s DPL greater than the RPL of the code segment’s segment selector

If the stack segment is not a writable data segment.

If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

#SS(0) If the top bytes of stack are not within stack limits.

If the return stack segment is not present.

#NP(selector) If the return code segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#GP If the return instruction pointer is not within the return code segment limit

#SS If the top bytes of stack are not within stack limits.

Virtual 8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit

#SS(0) If the top bytes of stack are not within stack limits.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

ROL/ROR—Rotate

See entry for RCL/RCR/ROL/ROR.

RSM—Resume from System Management Mode

Opcode	Instruction	Description
0F AA	RSM	Resume operation of interrupted program

Description

Returns program control from system management mode (SMM) to the application program or operating system procedure that was interrupted when the processor received an SSM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486 only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 9 in the *Intel Architecture Software Developer's Manual, Volume 3* for more information about SMM and the behavior of the RSM instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,RSM);

ReturnFromSSM;
ProcessorState ← Restore(SSMDump);
```

Flags Affected

All.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

Real Address Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

Virtual 8086 Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM.

SAHF—Store AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	2	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS registers are set as shown in the "Operation" below

Operation

```
EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are set to 1, 0, and 0, respectively.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

SAL/SAR/SHL/SHR—Shift Instructions

Opcode	Instruction	Description
D0 /4	SAL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SAL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SAL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SAL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SAL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SAL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SAL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SAL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SAL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /7	SAR *r/m8*,1	Signed divide* *r/m8* by 2, once
D2 /7	SAR *r/m8*,CL	Signed divide* *r/m8* by 2, CL times
C0 /7 *ib*	SAR *r/m8,imm8*	Signed divide* *r/m8* by 2, *imm8* times
D1 /7	SAR *r/m16*,1	Signed divide* *r/m16* by 2, once
D3 /7	SAR *r/m16*,CL	Signed divide* *r/m16* by 2, CL times
C1 /7 *ib*	SAR *r/m16,imm8*	Signed divide* *r/m16* by 2, *imm8* times
D1 /7	SAR *r/m32*,1	Signed divide* *r/m32* by 2, once
D3 /7	SAR *r/m32*,CL	Signed divide* *r/m32* by 2, CL times
C1 /7 *ib*	SAR *r/m32,imm8*	Signed divide* *r/m32* by 2, *imm8* times
D0 /4	SHL *r/m8*,1	Multiply *r/m8* by 2, once
D2 /4	SHL *r/m8*,CL	Multiply *r/m8* by 2, CL times
C0 /4 *ib*	SHL *r/m8,imm8*	Multiply *r/m8* by 2, *imm8* times
D1 /4	SHL *r/m16*,1	Multiply *r/m16* by 2, once
D3 /4	SHL *r/m16*,CL	Multiply *r/m16* by 2, CL times
C1 /4 *ib*	SHL *r/m16,imm8*	Multiply *r/m16* by 2, *imm8* times
D1 /4	SHL *r/m32*,1	Multiply *r/m32* by 2, once
D3 /4	SHL *r/m32*,CL	Multiply *r/m32* by 2, CL times
C1 /4 *ib*	SHL *r/m32,imm8*	Multiply *r/m32* by 2, *imm8* times
D0 /5	SHR *r/m8*,1	Unsigned divide *r/m8* by 2, once
D2 /5	SHR *r/m8*,CL	Unsigned divide *r/m8* by 2, CL times
C0 /5 *ib*	SHR *r/m8,imm8*	Unsigned divide *r/m8* by 2, *imm8* times
D1 /5	SHR *r/m16*,1	Unsigned divide *r/m16* by 2, once
D3 /5	SHR *r/m16*,CL	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16,imm8*	Unsigned divide *r/m16* by 2, *imm8* times
D1 /5	SHR *r/m32*,1	Unsigned divide *r/m32* by 2, once
D3 /5	SHR *r/m32*,CL	Unsigned divide *r/m32* by 2, CL times
C1 /5 *ib*	SHR *r/m32,imm8*	Unsigned divide *r/m32* by 2, *imm8* times

Note:
*Not the same form of division as IDIV; rounding is toward negative infinity.

Description

Shift the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to from 0 to 31. A special opcode encoding is provide for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared.

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit; the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position’s shifted value with the sign of the unshifted value.

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

Operation

```
tempCOUNT ← COUNT;
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
```

SAL/SAR/SHL/SHR—Shift Instructions (Continued)

```
  IF instruction is SAL or SHL
      THEN
          CF ← MSB(DEST);
      ELSE (* instruction is SAR or SHR *)
          CF ← LSB(DEST);
  FI;
  IF instruction is SAL or SHL
      THEN
          DEST ← DEST ∗ 2;
      ELSE
          IF instruction is SAR
              THEN
                  DEST ← DEST / 2 (*Signed divide, rounding toward negative infinity*);
              ELSE (* instruction is SHR *)
                  DEST ← DEST / 2 ; (* Unsigned divide *);
          FI;
  FI;
  temp ← temp - 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT = 1
  THEN
      IF instruction is SAL or SHL
          THEN
              OF ← MSB(DEST) XOR CF;
          ELSE
              IF instruction is SAR
                  THEN
                      OF ← 0;
                  ELSE (* instruction is SHR *)
                      OF ← MSB(tempDEST);
              FI;
      FI;
  ELSE
      OF ← undefined;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions count is greater than or equal to the size of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected.

SAL/SAR/SHL/SHR—Shift Instructions (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

The 8086 does not mask the shift count. All Intel architecture processors from the Intel386 processor on do mask the rotation count in all operating modes.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Description
1C *ib*	SBB AL,*imm8*	Subtract with borrow *imm8* from AL
1D *iw*	SBB AX,*imm16*	Subtract with borrow *imm16* from AX
1D *id*	SBB EAX,*imm32*	Subtract with borrow *imm32* from EAX
80 /3 *ib*	SBB *r/m8,imm8*	Subtract with borrow *imm8* from *r/m8*
81 /3 *iw*	SBB *r/m16,imm16*	Subtract with borrow *imm16* from *r/m16*
81 /3 *id*	SBB *r/m32,imm32*	Subtract with borrow *imm32* from *r/m32*
83 /3 *ib*	SBB *r/m16,imm8*	Subtract with borrow sign-extended *imm8* from *r/m16*
83 /3 *ib*	SBB *r/m32,imm8*	Subtract with borrow sign-extended *imm8* from *r/m32*
18 */r*	SBB *r/m8,r8*	Subtract with borrow *r8* from *r/m8*
19 */r*	SBB *r/m16,r16*	Subtract with borrow *r16* from *r/m16*
19 */r*	SBB *r/m32,r32*	Subtract with borrow *r32* from *r/m32*
1A */r*	SBB *r8,r/m8*	Subtract with borrow *r/m8* from *r8*
1B */r*	SBB *r16,r/m16*	Subtract with borrow *r/m16* from *r16*
1B */r*	SBB *r32,r/m32*	Subtract with borrow *r/m32* from *r32*

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

Operation

```
DEST ← DEST - (SRC + CF);
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults   NaT Register Consumption Abort.

SBB—Integer Subtraction with Borrow (Continued)

Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SCAS/SCASB/SCASW/SCASD—Scan String Data

Opcode	Instruction	Description
AE	SCAS ES:(E)DI	Compare AL with byte at ES:(E)DI and set status flags
AF	SCAS ES:DI	Compare AX with word at ES:DI and set status flags
AF	SCAS ES:EDI	Compare EAX with doubleword at ES:EDI and set status flags
AE	SCASB	Compare AL with byte at ES:(E)DI and set status flags
AF	SCASW	Compare AX with word at ES:DI and set status flags
AF	SCASD	Compare EAX with doubleword at ES:EDI and set status flags

Description

Compares the byte, word, or double word specified with the source operand with the value in the AL, AX, or EAX register, respectively, and sets the status flags in the EFLAGS register according to the results. The source operand specifies the memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the source-index register.) The ES segment cannot be overridden with a segment override prefix.

The SCASB, SCASW, and SCASD mnemonics are synonyms of the byte, word, and doubleword versions of the SCAS instructions. They are simpler to use, but provide no type or segment checking. (For the SCAS instruction, "ES:EDI" must be explicitly specified in the instruction.)

After the comparison, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The SCAS, SCASB, SCASW, and SCASD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" on page 4:337 for a description of the REP prefix.

Operation

```
IF (byte cmparison)
  THEN
      temp ← AL – SRC;
      SetStatusFlags(temp);
          THEN IF DF = 0
              THEN (E)DI ← 1;
              ELSE (E)DI ← -1;
          FI;
  ELSE IF (word comparison)
      THEN
          temp ← AX – SRC;
          SetStatusFlags(temp)
              THEN IF DF = 0
```

SCAS/SCASB/SCASW/SCASD—Scan String Data (Continued)

```
                THEN DI ← 2;
                ELSE DI ← -2;
            FI;
    ELSE (* doubleword comparison *)
        temp ← EAX – SRC;
        SetStatusFlags(temp)
            THEN IF DF = 0
                THEN EDI ← 4;
                ELSE EDI ← -4;
            FI;
  FI;
FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a null segment selector. If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SETcc—Set Byte on Condition

Opcode	Instruction	Description
0F 97	SETA *r/m8*	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE *r/m8*	Set byte if above or equal (CF=0)
0F 92	SETB *r/m8*	Set byte if below (CF=1)
0F 96	SETBE *r/m8*	Set byte if below or equal (CF=1 or (ZF=1)
0F 92	SETC *r/m8*	Set if carry (CF=1)
0F 94	SETE *r/m8*	Set byte if equal (ZF=1)
0F 9F	SETG *r/m8*	Set byte if greater (ZF=0 and SF=OF)
0F 9D	SETGE *r/m8*	Set byte if greater or equal (SF=OF)
0F 9C	SETL *r/m8*	Set byte if less (SF<>OF)
0F 9E	SETLE *r/m8*	Set byte if less or equal (ZF=1 or SF<>OF)
0F 96	SETNA *r/m8*	Set byte if not above (CF=1 or ZF=1)
0F 92	SETNAE *r/m8*	Set byte if not above or equal (CF=1)
0F 93	SETNB *r/m8*	Set byte if not below (CF=0)
0F 97	SETNBE *r/m8*	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC *r/m8*	Set byte if not carry (CF=0)
0F 95	SETNE *r/m8*	Set byte if not equal (ZF=0)
0F 9E	SETNG *r/m8*	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE *r/m8*	Set if not greater or equal (SF<>OF)
0F 9D	SETNL *r/m8*	Set byte if not less (SF=OF)
0F 9F	SETNLE *r/m8*	Set byte if not less or equal (ZF=0 and SF=OF)
0F 91	SETNO *r/m8*	Set byte if not overflow (OF=0)
0F 9B	SETNP *r/m8*	Set byte if not parity (PF=0)
0F 99	SETNS *r/m8*	Set byte if not sign (SF=0)
0F 95	SETNZ *r/m8*	Set byte if not zero (ZF=0)
0F 90	SETO *r/m8*	Set byte if overflow (OF=1)
0F 9A	SETP *r/m8*	Set byte if parity (PF=1)
0F 9A	SETPE *r/m8*	Set byte if parity even (PF=1)
0F 9B	SETPO *r/m8*	Set byte if parity odd (PF=0)
0F 98	SETS *r/m8*	Set byte if sign (SF=1)
0F 94	SETZ *r/m8*	Set byte if zero (ZF=1)

Description

Set the destination operand to the value 0 or 1, depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

SETcc—Set Byte on Condition (Continued)

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, the SETG (set byte if greater) and SETNLE (set if not less or equal) both have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible.

Some languages represent a logical one as an integer with all bits set. This representation can be arrived at by choosing the mutually exclusive condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

Operation

```
IF condition
   THEN DEST ← 1
   ELSE DEST ← 0;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

SETcc—Set Byte on Condition (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /0	SGDT *m*	Store GDTR to *m*
0F 01 /1	SIDT *m*	Store IDTR to *m*

Description

Stores the contents of the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR) in the destination operand. The destination operand is a pointer to 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the lower 2 bytes of the memory location and the 32-bit base address is stored in the upper 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the lower 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte is filled with 0s.

The SGDT and SIDT instructions are useful only in operating-system software; however, they can be used in application programs.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,SGDT/SIDT);
IF instruction is IDTR
  THEN
      IF OperandSize = 16
          THEN
              DEST[0:15] ← IDTR(Limit);
              DEST[16:39] ← IDTR(Base); (* 24 bits of base address loaded; *)
              DEST[40:47] ← 0;
          ELSE (* 32-bit Operand Size *)
              DEST[0:15] ← IDTR(Limit);
              DEST[16:47] ← IDTR(Base); (* full 32-bit base address loaded *)
      FI;
  ELSE (* instruction is SGDT *)
      IF OperandSize = 16
          THEN
              DEST[0:15] ← GDTR(Limit);
              DEST[16:39] ← GDTR(Base); (* 24 bits of base address loaded; *)
              DEST[40:47] ← 0;
          ELSE (* 32-bit Operand Size *)
              DEST[0:15] ← GDTR(Limit);
              DEST[16:47] ← GDTR(Base); (* full 32-bit base address loaded *)
      FI;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Instruction Intercept for SIDT and SGDT.

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register (Continued)

Protected Mode Exceptions

#UD If the destination operand is a register.

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real Address Mode Exceptions

#UD If the destination operand is a register.

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#UD If the destination operand is a register.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

Intel Architecture Compatibility

The 16-bit forms of the SGDT and SIDT instructions are compatible with the Intel 286 processor, if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium Pro processor fills these bits with 0s.

SHL/SHR—Shift Instructions

See entry for SAL/SAR/SHL/SHR.

SHLD—Double Precision Shift Left

Opcode	Instruction	Description
0F A4	SHLD *r/m16,r16,imm8*	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right
0F A5	SHLD *r/m16,r16*,CL	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right
0F A4	SHLD *r/m32,r32,imm8*	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right
0F A5	SHLD *r/m32,r32*,CL	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right

Description

Shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHLD instruction is useful for multi-precision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
  THEN
      no operation
  ELSE
      IF COUNT ≥ SIZE
          THEN (* Bad parameters *)
              DEST is undefined;
              CF, OF, SF, ZF, AF, PF are undefined;
          ELSE (* Perform the shift *)
              CF ← BIT[DEST, SIZE - COUNT];
              (* Last bit shifted out on exit *)
              FOR i ← SIZE - 1 DOWNTO COUNT
              DO
                  Bit(DEST, i) ← Bit(DEST, i - COUNT);
              OD;
              FOR i ← COUNT - 1 DOWNTO 0
```

SHLD—Double Precision Shift Left (Continued)

```
            DO
                BIT[DEST, i] ← BIT[SRC, i - COUNT + SIZE];
            OD;
      FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

SHRD—Double Precision Shift Right

Opcode	Instruction	Description
0F AC	SHRD *r/m16,r16,imm8*	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left
0F AD	SHRD *r/m16,r16*,CL	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left
0F AC	SHRD *r/m32,r32,imm8*	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left
0F AD	SHRD *r/m32,r32*,CL	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left

Description

Shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
  THEN
      no operation
  ELSE
      IF COUNT ≥ SIZE
          THEN (* Bad parameters *)
              DEST is undefined;
              CF, OF, SF, ZF, AF, PF are undefined;
          ELSE (* Perform the shift *)
              CF ← BIT[DEST, COUNT - 1]; (* last bit shifted out on exit *)
              FOR i ← 0 TO SIZE - 1 - COUNT
                  DO
                      BIT[DEST, i] ← BIT[DEST, i - COUNT];
                  OD;
              FOR i ← SIZE - COUNT TO SIZE - 1
                  DO
                      BIT[DEST,i] ← BIT[inBits,i+COUNT - SIZE];
                  OD;
      FI;
FI;
```

SHRD—Double Precision Shift Right (Continued)

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

SIDT—Store Interrupt Descriptor Table Register

See entry for SGDT/SIDT.

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Description
0F 00 /0	SLDT *r/m16*	Stores segment selector from LDTR in *r/m16*
0F 00 /0	SLDT *r/m32*	Store segment selector from LDTR in low-order 16 bits of *r/m32*; high-order 16 bits are undefined

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the LDT.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared to 0s. With the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

The SLDT instruction is only useful in operating-system software; however, it can be used in application programs. Also, this instruction can only be executed in protected mode.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,SLDT);
DEST ← LDTR(SegmentSelector);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	SLDT results in an IA-32 Intercept

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

SLDT—Store Local Descriptor Table Register (Continued)

Real Address Mode Exceptions

#UD	The SLDT instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD	The SLDT instruction is not recognized in virtual 8086 mode.

SMSW—Store Machine Status Word

Opcode	Instruction	Description
0F 01 /4	SMSW *r32/m16*	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a 16-bit general-purpose register or a memory location.

When the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the upper 16 bits of the register are undefined. With the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

The SMSW instruction is only useful in operating-system software; however, it is not a privileged instruction and can be used in application programs.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on processors more recent than the Intel 286 should use the MOV (control registers) instruction to load the machine status word.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,SMSW);
DEST ← CR0[15:0]; (* MachineStatusWord *);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

SMSW—Store Machine Status Word (Continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

STC—Set Carry Flag

Opcode	Instruction	Description
F9	STC	Set CF flag

Description

Sets the CF flag in the EFLAGS register.

Operation

```
CF ← 1;
```

Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

STD—Set Direction Flag

Opcode	Instruction	Description
FD	STD	Set DF flag

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI).

Operation

```
DF ← 1;
```

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Operation

```
DF ← 1;
```

Exceptions (All Operating Modes)

None.

STI—Set Interrupt Flag

Opcode	Instruction	Description
FB	STI	Set interrupt flag; interrupts enabled at the end of the next instruction

Description

Sets the interrupt flag (IF) in the EFLAGS register. **In the IA-32 System Environment,** after the IF flag is set, the processor begins responding to external maskable interrupts after the next instruction is executed. If the STI instruction is followed by a CLI instruction (which clears the IF flag) the effect of the STI instruction is negated. **In the Itanium System Environment, the processor will immediately respond do interrupts after STI, unless execution of STI results in a trap or intercept. External interrupts are enabled for IA-32 instructions if PSR.i and (~CFLG.if or EFLAG.if).**

The IF flag and the STI and CLI instruction have no affect on the generation of exceptions and NMI interrupts.

The following decision table indicates the action of the STI instruction (bottom of the table) depending on the processor's mode of operating and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1
VM =	X	0	0	1
CPL	X	≤ IOPL	> IOPL	=3
IOPL	X	X	X	=3
IF ← 1	Y	Y	N	Y
#GP(0)	N	N	Y	N

Notes:
XDon't care.
NAction in Column 1 not taken.
YAction in Column 1 taken.

Operation

```
OLD_IF <- IF;
IF PE=0  (* Executing in real-address mode *)
   THEN
         IF ← 1;  (* Set Interrupt Flag *)
   ELSE  (* Executing in protected mode or virtual-8086 mode *)
         IF VM=0  (* Executing in protected mode*)
               THEN
                     IF CR4.PVI = 0
                           THEN
                                 IF CPL <= IOPL
                                 THEN IF <- 1
                                 ELSE #GP(0);
                                 FI;
                           ELSE (*PVI is 1 *)
```

STI—Set Interrupt Flag (Continued)

```
                      IF CPL = 3
                      THENSTI—Set Interrupt Flag (Continued)

                          IF IOPL < 3
                          THEN
                              IF VIP = 0
                              THEN VIF <- 1;
                              ELSE #GP(0);
                              FI;
                          ELSE (*IOPL = 3 *)
                              IF <- 1;
                          FI;
                      ELSE (*CPL < 3*)
                              IF IOPL < CPL THEN #GP(0); FI;
                              IF IOPL>=CPL OR IOPL=3 THEN IF <-1; FI;
                          FI;
                  FI;
              ELSE (*Executing in Virtual-8086 Mode*)
                  IF IOPL = 3
                      THEN IF <- 1;
                  ELSE
                      IF CR4.VME = 0
                      THEN #GP(0);
                      ELSE
                          IF VIP = 1     (*virtual interrupt is pending*)
                          THEN #GP(0);
                          ELSE VIF <- 1;
                          FI;
                      FI;
                  FI;
          FI;
  FI;
FI;

IF Itanium System Environment AND CFLG.ii AND IF != OLD_IF
  THEN IA-32_Intercept(System_Flag,STI);
```

Flags Affected

The IF flag is set to 1.

Additional Itanium System Environment Exceptions

IA-32_Intercept System Flag Intercept Trap if CFLG.ii is 1 and the IF flag changes state.

Protected Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

STI—Set Interrupt Flag (Continued)

Real Address Mode Exceptions

None.

Virtual 8086 Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

STOS/STOSB/STOSW/STOSD—Store String Data

Opcode	Instruction	Description
AA	STOS ES:(E)DI	Store AL at address ES:(E)DI
AB	STOS ES:DI	Store AX at address ES:DI
AB	STOS ES:EDI	Store EAX at address ES:EDI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:DI
AB	STOSD	Store EAX at address ES:EDI

Description

Stores a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location at the address ES:EDI. (When the operand-size attribute is 16, the DI register is used as the source-index register.) The ES segment cannot be overridden with a segment override prefix.

The STOSB, STOSW, and STOSD mnemonics are synonyms of the byte, word, and doubleword versions of the STOS instructions. They are simpler to use, but provide no type or segment checking. (For the STOS instruction, "ES:EDI" must be explicitly specified in the instruction.)

After the byte, word, or doubleword is transfer from the AL, AX, or EAX register to the memory location, the EDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the EDI register is incremented; if the DF flag is 1, the EDI register is decremented.) The EDI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The STOS, STOSB, STOSW, and STOSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct, because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix" on page 4:337 for a description of the REP prefix.

Operation

```
IF (byte store)
  THEN
      DEST ← AL;
          THEN IF DF = 0
              THEN (E)DI ← 1;
              ELSE (E)DI ← -1;
          FI;
  ELSE IF (word store)
      THEN
          DEST ← AX;
              THEN IF DF = 0
                  THEN DI ← 2;
                  ELSE DI ← -2;
              FI;
      ELSE (* doubleword store *)
```

STOS/STOSB/STOSW/STOSD—Store String Data (Continued)

```
                DEST ← EAX;
                    THEN IF DF = 0
                        THEN EDI ← 4;
                        ELSE EDI ← -4;
                    FI;
  FI;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment. If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a null segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

STR—Store Task Register

Opcode	Instruction	Description
0F 00 /1	STR *r/m16*	Stores segment selector from TR in *r/m16*

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared to 0s. With the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,STR);
DEST ← TR(SegmentSelector);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a nonwritable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#UD	The STR instruction is not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD	The STR instruction is not recognized in virtual 8086 mode.

SUB—Integer Subtraction

Opcode	Instruction	Description
2C *ib*	SUB AL,*imm8*	Subtract *imm8* from AL
2D *iw*	SUB AX,*imm16*	Subtract *imm16* from AX
2D *id*	SUB EAX,*imm32*	Subtract *imm32* from EAX
80 /5 *ib*	SUB *r/m8,imm8*	Subtract *imm8* from *r/m8*
81 /5 *iw*	SUB *r/m16,imm16*	Subtract *imm16* from *r/m16*
81 /5 *id*	SUB *r/m32,imm32*	Subtract *imm32* from *r/m32*
83 /5 *ib*	SUB *r/m16,imm8*	Subtract sign-extended *imm8* from *r/m16*
83 /5 *ib*	SUB *r/m32,imm8*	Subtract sign-extended *imm8* from *r/m32*
28 /*r*	SUB *r/m8,r8*	Subtract *r8* from *r/m8*
29 /*r*	SUB *r/m16,r16*	Subtract *r16* from *r/m16*
29 /*r*	SUB *r/m32,r32*	Subtract *r32* from *r/m32*
2A /*r*	SUB *r8,r/m8*	Subtract *r/m8* from *r8*
2B /*r*	SUB *r16,r/m16*	Subtract *r/m16* from *r16*
2B /*r*	SUB *r32,r/m32*	Subtract *r/m32* from *r32*

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Operation

```
DEST ← DEST - SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

SUB—Integer Subtraction (Continued)

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

TEST—Logical Compare

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8,imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16,imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32,imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8,r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16,r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32,r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
    THEN ZF ← 0;
    ELSE ZF ← 1;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(*AF is Undefined*)
```

Flags Affected

The OF and CF flags are cleared to 0. The SF, ZF, and PF flags are set according to the result (see "Operation" above). The state of the AF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

TEST—Logical Compare (Continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

UD2—Undefined Instruction

Opcode	Instruction	Description
0F 0B	UD2	Raise invalid opcode exception

Description

Generates an invalid opcode. This instruction is provided for software testing to explicitly generate an invalid opcode. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction is the same as the NOP instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,0F0B);
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Exceptions (All Operating Modes)

#UD	Instruction is guaranteed to raise an invalid opcode exception in all operating modes).

VERR, VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Description
0F 00 /4	VERR *r/m16*	Set ZF=1 if segment specified with *r/m16* can be read
0F 00 /5	VERW *r/m16*	Set ZF=1 if segment specified with *r/m16* can be written

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not null.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable; the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment’s DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as if the segment were loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) were performed. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Operation

```
IF SRC(Offset) > (GDTR(Limit) OR (LDTR(Limit))
      THEN
          ZF ← 0
Read segment descriptor;
IF SegmentDescriptor(DescriptorType) = 0 (* system segment *)
  OR (SegmentDescriptor(Type) ≠ conforming code segment)
  AND (CPL > DPL) OR (RPL > DPL)
      THEN
          ZF ← 0
      ELSE
          IF ((Instruction = VERR) AND (segment = readable))
              OR ((Instruction = VERW) AND (segment = writable))
              THEN
                  ZF ← 1;
          FI;
FI;
```

VERR, VERW—Verify a Segment for Reading or Writing (Continued)

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is cleared to 0.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#UD The VERR and VERW instructions are not recognized in real address mode.

Virtual 8086 Mode Exceptions

#UD The VERR and VERW instructions are not recognized in virtual 8086 mode.

WAIT/FWAIT—Wait

Opcode	Instruction	Description
9B	WAIT	Check pending unmasked floating-point exceptions.
9B	FWAIT	Check pending unmasked floating-point exceptions.

Description

Causes the processor to check for and handle pending unmasked floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for the WAIT).

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction insures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results.

Operation

```
CheckPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-point Exceptions

None.

Protected Mode Exceptions

#NM	MP and TS in CR0 is set.

Real Address Mode Exceptions

#NM	MP and TS in CR0 is set.

Virtual 8086 Mode Exceptions

#NM	MP and TS in CR0 is set.

WBINVD—Write-Back and Invalidate Cache

Opcode	Instruction	Description
0F 09	WBINVD	Write-back and flush Internal caches; initiate writing-back and flushing of external caches.

Description

Writes back all modified cache lines in the processor's internal cache to main memory, invalidates (flushes) the internal caches, and issues a special-function bus cycle that directs external caches to also write back modified data.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction.

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,WBINVD);
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real Address Mode Exceptions

None.

WBINVD—Write-Back and Invalidate Cache (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	The WBINVD instruction cannot be executed at the virtual 8086 mode.

Intel Architecture Compatibility

The WDINVD instruction implementation-dependent; its function may be implemented differently on future Intel architecture processors. The instruction is not supported on Intel architecture processors earlier than the Intel486 processor.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Description
0F 30	WRMSR	Write the value in EDX:EAX to MSR specified by ECX

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The high-order 32 bits are copied from EDX and the low-order 32 bits are copied from EAX. Always set undefined or reserved bits in an MSR to the values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated, including the global entries see the *Intel Architecture Software Developer's Manual, Volume 3*).

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. See model-specific instructions for all the MSRs that can be written to with this instruction and their addresses.

The WRMSR instruction is a serializing instruction.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

Operation

```
IF Itanium System Environment THEN IA-32_Intercept(INST,WRMSR);
MSR[ECX] ← EDX:EAX;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

IA-32_Intercept	Mandatory Instruction Intercept.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the value in ECX specifies a reserved or unimplemented MSR address.

Real Address Mode Exceptions

#GP	If the current privilege level is not 0 If the value in ECX specifies a reserved or unimplemented MSR address.

WRMSR—Write to Model Specific Register (Continued)

Virtual 8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual 8086 mode.

Intel Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the Intel architecture with the Pentium processor. Execution of this instruction by an Intel architecture processor earlier than the Pentium processor results in an invalid opcode exception #UD.

XADD—Exchange and Add

Opcode	Instruction	Description
0F C0/r	XADD *r/m8,r8*	Exchange *r8* and *r/m8*; load sum into *r/m8*.
0F C1/r	XADD *r/m16,r16*	Exchange *r16* and *r/m16*; load sum into *r/m16*.
0F C1/r	XADD *r/m32,r32*	Exchange *r32* and *r/m32*; load sum into *r/m32*.

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

This instruction can be used with a LOCK prefix.

Operation

```
IF Itanium System Environment AND External_Bus_Lock_Required AND DCR.lc
  THEN IA-32_Intercept(LOCK,XADD);
TEMP ← SRC + DEST
SRC ← DEST
DEST ← TEMP
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result stored in the destination operand.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

IA-32_Intercept Lock Intercept – If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

Protected Mode Exceptions

#GP(0) If the destination is located in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

XADD—Exchange and Add (Continued)

Real Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Intel Architecture Compatibility

Intel architecture processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Description
90+*rw*	XCHG AX,*r16*	Exchange *r16* with AX
90+*rw*	XCHG *r16*,AX	Exchange *r16* with AX
90+*rd*	XCHG EAX,*r32*	Exchange *r32* with EAX
90+*rd*	XCHG *r32*,EAX	Exchange *r32* with EAX
86 /*r*	XCHG *r/m8,r8*	Exchange byte register with EA byte
86 /*r*	XCHG *r8,r/m8*	Exchange byte register with EA byte
87 /*r*	XCHG *r/m16,r16*	Exchange *r16* with EA word
87 /*r*	XCHG *r16,r/m16*	Exchange *r16* with EA word
87 /*r*	XCHG *r/m32,r32*	Exchange *r32* with EA doubleword
87 /*r*	XCHG *r32,r/m32*	Exchange *r32* with EA doubleword

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. When the operands are two registers, one of the registers must be the EAX or AX register. If a memory operand is referenced, the LOCK# signal is automatically asserted for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See Chapter 5, *Processor Management and Initialization*, in the *Intel Architecture Software Developer's Manual, Volume 3* for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

Operation

```
IF Itanium System Environment AND External_Atomic_Lock_Required AND DCR.lc
  THEN IA-32_Intercept(LOCK,XCHG);
TEMP ← DEST
DEST ← SRC
SRC ← TEMP
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

XCHG—Exchange Register/Memory with Register (Continued)

IA-32_Intercept Lock Intercept – If an external atomic bus lock is required to complete this operation and DCR.lc is 1, no atomic transaction occurs, this instruction is faulted and an IA-32_Intercept(Lock) fault is generated. The software lock handler is responsible for the emulation of this instruction.

Protected Mode Exceptions

#GP(0) If either operand is in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Description
D7	XLAT *m8*	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL]

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from the DS:EBX registers (or the DS:BX registers when the address-size attribute of 16 bits.) The XLAT instruction allows a different segment register to be specified with a segment override. When assembled, the XLAT and XLATB instructions produce the same machine code.

Operation

```
IF AddressSize = 16
   THEN
         AL ← (DS:BX + ZeroExtend(AL))
   ELSE (* AddressSize = 32 *)
         AL ← (DS:EBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

XLAT/XLATB—Table Look-up Translation (Continued)

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

XOR—Logical Exclusive OR

Opcode	Instruction	Description
34 *ib*	XOR AL,*imm8*	AL XOR *imm8*
35 *iw*	XOR AX,*imm16*	AX XOR *imm16*
35 *id*	XOR EAX,*imm32*	EAX XOR *imm32*
80 /6 *ib*	XOR *r/m8,imm8*	*r/m8* XOR *imm8*
81 /6 *iw*	XOR *r/m16,imm16*	*r/m16* XOR *imm16*
81 /6 *id*	XOR *r/m32,imm32*	*r/m32* XOR *imm32*
83 /6 *ib*	XOR *r/m16,imm8*	*r/m16* XOR *imm8*
83 /6 *ib*	XOR *r/m32,imm8*	*r/m32* XOR *imm8*
30 */r*	XOR *r/m8,r8*	*r/m8* XOR *r8*
31 */r*	XOR *r/m16,r16*	*r/m16* XOR *r16*
31 */r*	XOR *r/m32,r32*	*r/m32* XOR *r32*
32 */r*	XOR *r8,r/m8*	*r8* XOR *r/m8*
33 */r*	XOR *r16,r/m16*	*r8* XOR *r/m8*
33 */r*	XOR *r32,r/m32*	*r8* XOR *r/m8*

Description

Performs a bitwise exclusive-OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.

Operation

```
DEST ← DEST XOR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

XOR—Logical Exclusive OR (Continued)

Protected Mode Exceptions

#GP(0) If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual 8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

§

IA-32 Intel® MMX™ Technology Instruction Reference 3

This section lists the IA-32 MMX technology instructions designed to increase performance of multimedia intensive applications.

EMMS—Empty MMX State

Opcode	Instruction	Description
0F 77	EMMS	Set the FP tag word to empty.

Description

Sets the values of all the tags in the FPU tag word to empty (all ones). This operation marks the MMX technology registers as available, so they can subsequently be used by floating-point instructions. (See Figure 7-11 in the *Intel Architecture Software Developer's Manual, Volume 1*, for the format of the FPU tag word.) All other MMX technology instructions (other than the EMMS instruction) set all the tags in FPU tag word to valid (all zeros).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology routines and before calling other procedures or subroutines that may execute floating-point instructions. If a floating-point instruction loads one of the registers in the FPU register stack before the FPU tag word has been reset by the EMMS instruction, a floating-point stack overflow can occur that will result in a floating-point exception or incorrect result.

Operation

```
FPUTagWord ← FFFFH;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1.

Protected Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Real-Address Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

MOVD—Move 32 Bits

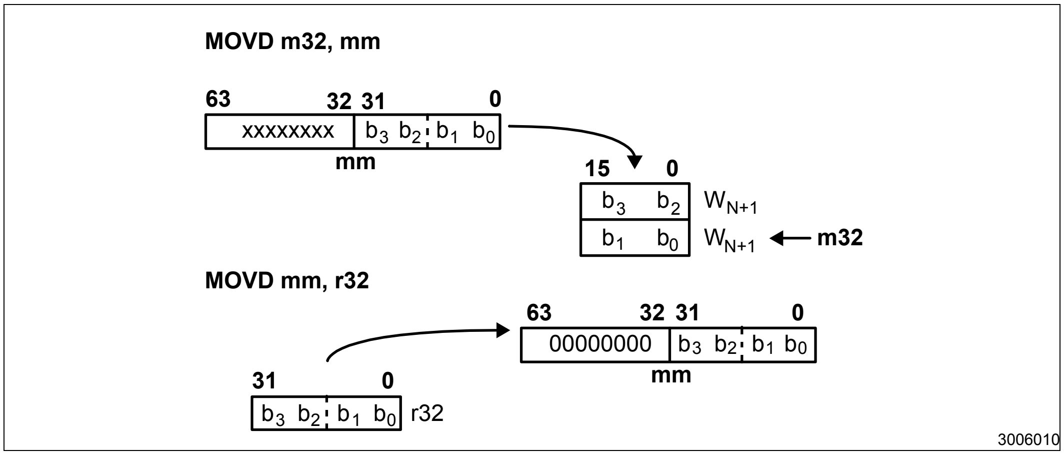
Opcode	Instruction	Description
0F 6E /r	MOVD *mm, r/m32*	Move doubleword from *r/m32* to *mm*.
0F 7E /r	MOVD *r/m32, mm*	Move doubleword from *mm* to *r/m32*.

Description

Copies doubleword from the source operand (second operand) to the destination operand (first operand). Source and destination operands can be MMX technology registers, memory locations, or 32-bit general-purpose registers; however, data cannot be transferred from an MMX technology register to an MMX technology register, from one memory location to another memory location, or from one general-purpose register to another general-purpose register.

When the destination operand is an MMX technology register, the 32-bit source value is written to the low-order 32 bits of the 64-bit MMX technology register and zero-extended to 64 bits (see Figure 3-1). When the source operand is an MMX technology register, the low-order 32 bits of the MMX technology register are written to the 32-bit general-purpose register or 32-bit memory location selected with the destination operand.

Figure 3-1. Operation of the MOVD Instruction



Operation

```
IF DEST is MMX register
   THEN
      DEST ← ZeroExtend(SRC);
   ELSE (* SRC is MMX register *)
      DEST ← LowOrderDoubleword(SRC);
```

MOVD—Move 32 Bits (continued)

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If the destination operand is in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

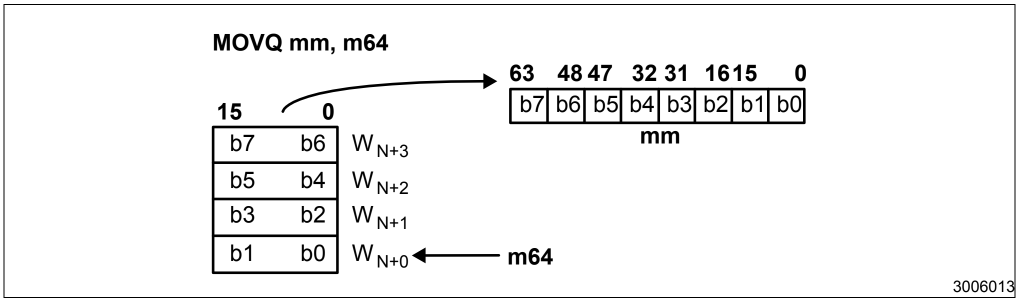
MOVQ—Move 64 Bits

Opcode	Instruction	Description
0F 6F /r	MOVQ *mm, mm/m64*	Move quadword from *mm/m64* to *mm*.
0F 7F /r	MOVQ *mm/m64, mm*	Move quadword from *mm* to *mm/m64*.

Description

Copies quadword from the source operand (second operand) to the destination operand (first operand). (See Figure 3-2.) A source or destination operand can be either an MMX technology register or a memory location; however, data cannot be transferred from one memory location to another memory location. Data can be transferred from one MMX technology register to another MMX technology register.

Figure 3-2. Operation of the MOVQ Instruction



Operation

```
DEST ← SRC;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MOVQ—Move 64 Bits (continued)

Protected Mode Exceptions

#GP(0) If the destination operand is in a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode	Instruction	Description
0F 63 /r	PACKSSWB *mm, mm/m64*	Packs and saturate pack 4 signed words from *mm* and 4 signed words from *mm/m64* into 8 signed bytes in *mm*.
0F 6B /r	PACKSSDW *mm, mm/m64*	Pack and saturate 2 signed doublewords from *mm* and 2 signed doublewords from *mm/m64* into 4 signed words in *mm*.

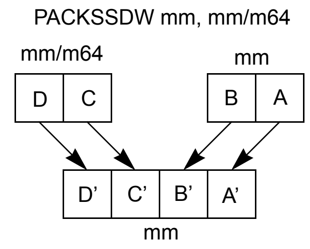
Description

Packs and saturates signed words into bytes (PACKSSWB) or signed doublewords into words (PACKSSDW). The PACKSSWB instruction packs 4 signed words from the destination operand (first operand) and 4 signed words from the source operand (second operand) into 8 signed bytes in the destination operand. If the signed value of a word is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is stored into the destination.

The PACKSSDW instruction packs 2 signed doublewords from the destination operand (first operand) and 2 signed doublewords from the source operand (second operand) into 4 signed words in the destination operand (see Figure 3-3). If the signed value of a doubleword is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is stored into the destination.

The destination operand for either the PACKSSWB or PACKSSDW instruction must be an MMX technology register; the source operand may be either an MMX technology register or a quadword memory location.

Figure 3-3. Operation of the PACKSSDW Instruction



Operation

```
IF instruction is PACKSSWB
  THEN
      DEST(7..0) ← SaturateSignedWordToSignedByte DEST(15..0);
      DEST(15..8) ← SaturateSignedWordToSignedByte DEST(31..16);
      DEST(23..16) ← SaturateSignedWordToSignedByte DEST(47..32);
      DEST(31..24) ← SaturateSignedWordToSignedByte DEST(63..48);
      DEST(39..32) ← SaturateSignedWordToSignedByte SRC(15..0);
      DEST(47..40) ← SaturateSignedWordToSignedByte SRC(31..16);
      DEST(55..48) ← SaturateSignedWordToSignedByte SRC(47..32);
      DEST(63..56) ← SaturateSignedWordToSignedByte SRC(63..48);
```

PACKSSWB/PACKSSDW—Pack with Signed Saturation (continued)

```
  ELSE (* instruction is PACKSSDW *)
      DEST(15..0) ← SaturateSignedDoublewordToSignedWord DEST(31..0);
      DEST(31..16) ← SaturateSignedDoublewordToSignedWord DEST(63..32);
      DEST(47..32) ← SaturateSignedDoublewordToSignedWord SRC(31..0);
      DEST(63..48) ← SaturateSignedDoublewordToSignedWord SRC(63..32);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

PACKSSWB/PACKSSDW—Pack with Signed Saturation (continued)

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PACKUSWB—Pack with Unsigned Saturation

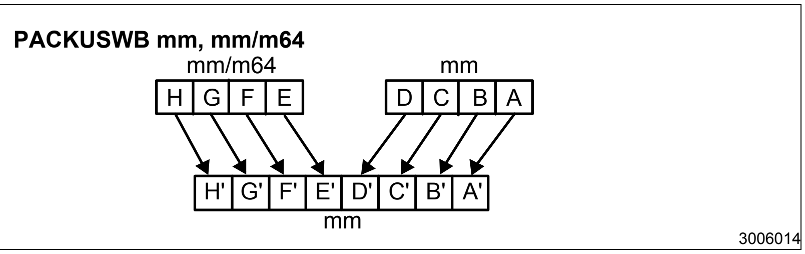
Opcode	Instruction	Description
0F 67 /r	PACKUSWB *mm, mm/m64*	Pack and saturate 4 signed words from *mm* and 4 signed words from *mm/m64* into 8 unsigned bytes in *mm*.

Description

Packs and saturates 4 signed words from the destination operand (first operand) and 4 signed words from the source operand (second operand) into 8 unsigned bytes in the destination operand (see Figure 3-4). If the signed value of a word is beyond the range of an unsigned byte (that is, greater than FFH or less than 00H), the saturated byte value of FFH or 00H, respectively, is stored into the destination.

The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a quadword memory location.

Figure 3-4. Operation of the PACKUSWB Instruction



Operation

```
DEST(7..0) ← SaturateSignedWordToUnsignedByte DEST(15..0);
DEST(15..8) ← SaturateSignedWordToUnsignedByte DEST(31..16);
DEST(23..16) ← SaturateSignedWordToUnsignedByte DEST(47..32);
DEST(31..24) ← SaturateSignedWordToUnsignedByte DEST(63..48);
DEST(39..32) ← SaturateSignedWordToUnsignedByte SRC(15..0);
DEST(47..40) ← SaturateSignedWordToUnsignedByte SRC(31..16);
DEST(55..48) ← SaturateSignedWordToUnsignedByte SRC(47..32);
DEST(63..56) ← SaturateSignedWordToUnsignedByte SRC(63..48);
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PACKUSWB—Pack with Unsigned Saturation (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PADDB/PADDW/PADDD—Packed Add

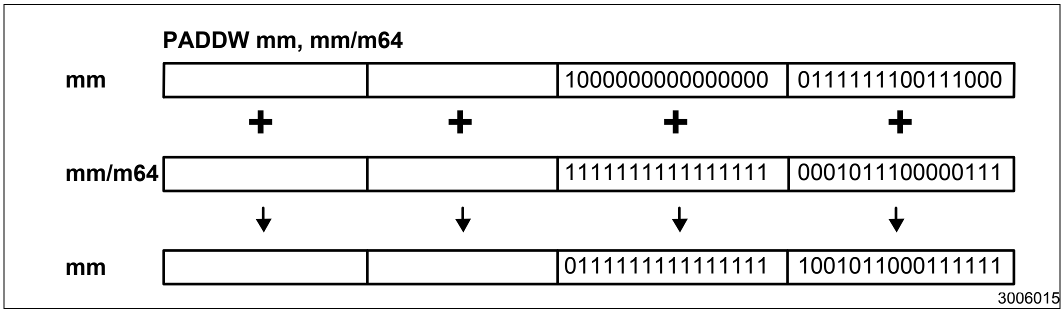
Opcode	Instruction	Description
0F FC /r	PADDB *mm, mm/m64*	Add packed bytes from *mm/m64* to packed bytes in *mm*.
0F FD /r	PADDW *mm, mm/m64*	Add packed words from *mm/m64* to packed words in *mm*.
0F FE /r	PADDD *mm, mm/m64*	Add packed doublewords from *mm/m64* to packed doublewords in *mm*.

Description

Adds the individual data elements (bytes, words, or doublewords) of the source operand (second operand) to the individual data elements of the destination operand (first operand). (See Figure 3-5.) If the result of an individual addition exceeds the range for the specified data type (overflows), the result is wrapped around, meaning that the result is truncated so that only the lower (least significant) bits of the result are returned (that is, the carry is ignored).

The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-5. Operation of the PADDW Instruction



The PADDB instruction adds the bytes of the source operand to the bytes of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 8 bits, the lower 8 bits of the result are written to the destination operand and therefore the result wraps around.

The PADDW instruction adds the words of the source operand to the words of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 16 bits, the lower 16 bits of the result are written to the destination operand and therefore the result wraps around.

The PADDD instruction adds the doublewords of the source operand to the doublewords of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 32 bits, the lower 32 bits of the result are written to the destination operand and therefore the result wraps around.

PADDB/PADDW/PADDD—Packed Add (continued)

Note that like the integer ADD instruction, the PADDB, PADDW, and PADDD instructions can operate on either unsigned or signed (two's complement notation) packed integers. Unlike the integer instructions, none of the MMX technology instructions affect the EFLAGS register. With MMX technology instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values or else use the "with saturation" MMX technology instructions.

Operation

```
IF instruction is PADDB
  THEN
      DEST(7..0) ← DEST(7..0) + SRC(7..0);
      DEST(15..8) ← DEST(15..8) + SRC(15..8);
      DEST(23..16) ← DEST(23..16)+ SRC(23..16);
      DEST(31..24) ← DEST(31..24) + SRC(31..24);
      DEST(39..32) ← DEST(39..32) + SRC(39..32);
      DEST(47..40) ← DEST(47..40)+ SRC(47..40);
      DEST(55..48) ← DEST(55..48) + SRC(55..48);
      DEST(63..56) ← DEST(63..56) + SRC(63..56);
ELSEIF instruction is PADDW
  THEN
      DEST(15..0) ← DEST(15..0) + SRC(15..0);
      DEST(31..16) ← DEST(31..16) + SRC(31..16);
      DEST(47..32) ← DEST(47..32) + SRC(47..32);
      DEST(63..48) ← DEST(63..48) + SRC(63..48);
  ELSE (* instruction is PADDD *)
      DEST(31..0) ← DEST(31..0) + SRC(31..0);
      DEST(63..32) ← DEST(63..32) + SRC(63..32);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PADDB/PADDW/PADDD—Packed Add (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

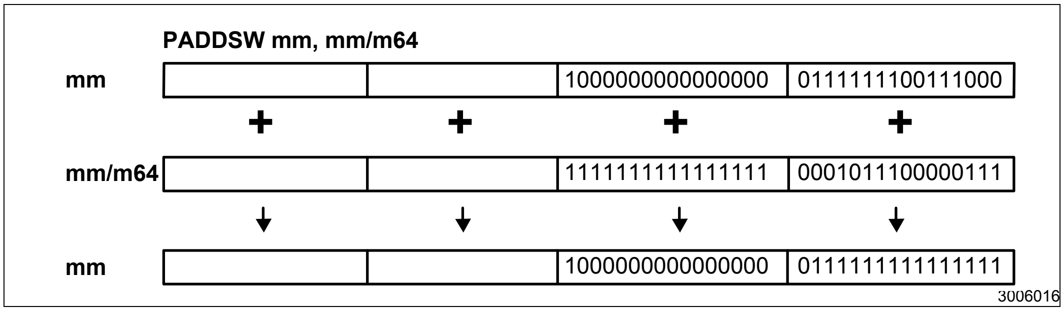
PADDSB/PADDSW—Packed Add with Saturation

Opcode	Instruction	Description
0F EC /r	PADDSB *mm, mm/m64*	Add signed packed bytes from *mm/m64* to signed packed bytes in *mm* and saturate.
0F ED /r	PADDSW *mm, mm/m64*	Add signed packed words from *mm/m64* to signed packed words in *mm* and saturate.

Description

Adds the individual signed data elements (bytes or words) of the source operand (second operand) to the individual signed data elements of the destination operand (first operand). (See Figure 3-6.) If the result of an individual addition exceeds the range for the specified data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-6. Operation of the PADDSW Instruction



The PADDSB instruction adds the signed bytes of the source operand to the signed bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is written to the destination operand.

The PADDSW instruction adds the signed words of the source operand to the signed words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is written to the destination operand.

Operation

```
IF instruction is PADDSB
  THEN
      DEST(7..0) ← SaturateToSignedByte(DEST(7..0) + SRC (7..0)) ;
      DEST(15..8) ← SaturateToSignedByte(DEST(15..8) + SRC(15..8) );
      DEST(23..16) ← SaturateToSignedByte(DEST(23..16)+ SRC(23..16) );
      DEST(31..24) ← SaturateToSignedByte(DEST(31..24) + SRC(31..24) );
      DEST(39..32) ← SaturateToSignedByte(DEST(39..32) + SRC(39..32) );
      DEST(47..40) ← SaturateToSignedByte(DEST(47..40)+ SRC(47..40) );
      DEST(55..48) ← SaturateToSignedByte(DEST(55..48) + SRC(55..48) );
      DEST(63..56) ← SaturateToSignedByte(DEST(63..56) + SRC(63..56) );
```

PADDSB/PADDSW—Packed Add with Saturation (continued)

```
    ELSE { (* instruction is PADDSW *)
        DEST(15..0) ← SaturateToSignedWord(DEST(15..0) + SRC(15..0) );
        DEST(31..16) ← SaturateToSignedWord(DEST(31..16) + SRC(31..16) );
        DEST(47..32) ← SaturateToSignedWord(DEST(47..32) + SRC(47..32) );
        DEST(63..48) ← SaturateToSignedWord(DEST(63..48) + SRC(63..48) );
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

PADDSB/PADDSW—Packed Add with Saturation (continued)

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

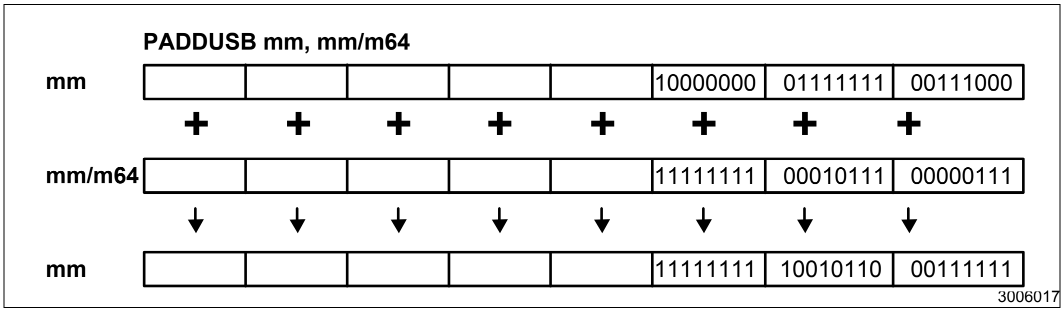
PADDUSB/PADDUSW—Packed Add Unsigned with Saturation

Opcode	Instruction	Description
0F DC /r	PADDUSB *mm, mm/m64*	Add unsigned packed bytes from *mm/m64* to unsigned packed bytes in *mm* and saturate.
0F DD /r	PADDUSW *mm, mm/m64*	Add unsigned packed words from *mm/m64* to unsigned packed words in *mm* and saturate.

Description

Adds the individual unsigned data elements (bytes or words) of the packed source operand (second operand) to the individual unsigned data elements of the packed destination operand (first operand). (See Figure 3-7.) If the result of an individual addition exceeds the range for the specified unsigned data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-7. Operation of the PADDUSB Instruction



The PADDUSB instruction adds the unsigned bytes of the source operand to the unsigned bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned byte (that is, greater than FFH), the saturated unsigned byte value of FFH is written to the destination operand.

The PADDUSW instruction adds the unsigned words of the source operand to the unsigned words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of an unsigned word (that is, greater than FFFFH), the saturated unsigned word value of FFFFH is written to the destination operand.

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation (continued)

Operation

```
IF instruction is PADDUSB
   THEN
         DEST(7..0) ← SaturateToUnsignedByte(DEST(7..0) + SRC (7..0) );
         DEST(15..8) ← SaturateToUnsignedByte(DEST(15..8) + SRC(15..8) );
         DEST(23..16) ← SaturateToUnsignedByte(DEST(23..16)+ SRC(23..16) );
         DEST(31..24) ← SaturateToUnsignedByte(DEST(31..24) + SRC(31..24) );
         DEST(39..32) ← SaturateToUnsignedByte(DEST(39..32) + SRC(39..32) );
         DEST(47..40) ← SaturateToUnsignedByte(DEST(47..40)+ SRC(47..40) );
         DEST(55..48) ← SaturateToUnsignedByte(DEST(55..48) + SRC(55..48) );
         DEST(63..56) ← SaturateToUnsignedByte(DEST(63..56) + SRC(63..56) );
   ELSE { (* instruction is PADDUSW *)
         DEST(15..0) ← SaturateToUnsignedWord(DEST(15..0) + SRC(15..0) );
         DEST(31..16) ← SaturateToUnsignedWord(DEST(31..16) + SRC(31..16) );
         DEST(47..32) ← SaturateToUnsignedWord(DEST(47..32) + SRC(47..32) );
         DEST(63..48) ← SaturateToUnsignedWord(DEST(63..48) + SRC(63..48) );
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PADDUSB/PADDUSW—Packed Add Unsigned with Saturation (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

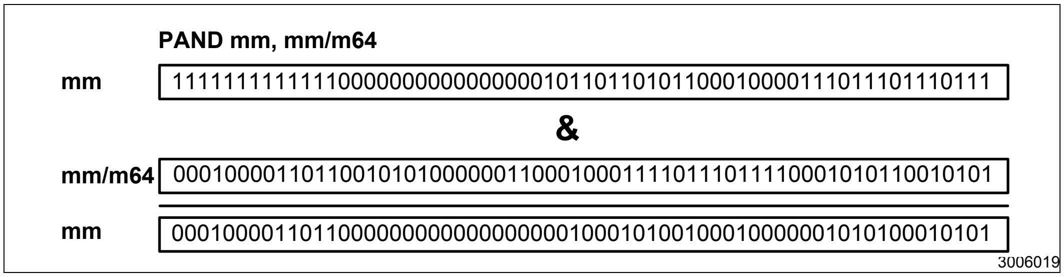
PAND—Logical AND

Opcode	Instruction	Description
0F DB /r	PAND *mm, mm/m64*	AND quadword from *mm/m64* to quadword in mm.

Description

Performs a bitwise logical AND operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 3-8). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result of the PAND instruction is set to 1 if the corresponding bits of the operands are both 1; otherwise it is made zero

Figure 3-8. Operation of the PAND Instruction



Operation

DEST ← DEST AND SRC;

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PAND—Logical AND (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PANDN—Logical AND NOT

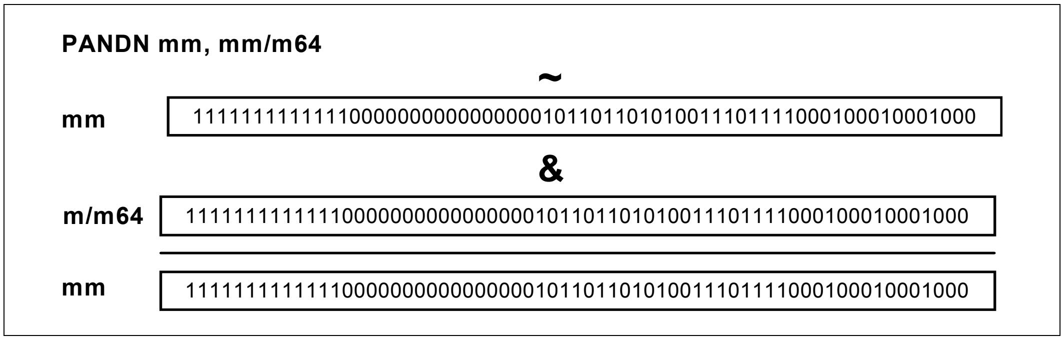
Opcode	Instruction	Description
0F DF /r	PANDN *mm, mm/m64*	AND quadword from *mm/m64* to NOT quadword in *mm*.

Description

Performs a bitwise logical NOT on the quadword destination operand (first operand). Then, the instruction performs a bitwise logical AND operation on the inverted destination operand and the quadword source operand (second operand). (See Figure 3-9.) Each bit of the result of the AND operation is set to one if the corresponding bits of the source and inverted destination bits are one; otherwise it is set to zero. The result is stored in the destination operand location.

The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register.

Figure 3-9. Operation of the PANDN Instruction



Operation

DEST ← (NOT DEST) AND SRC;

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PANDN—Logical AND NOT (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

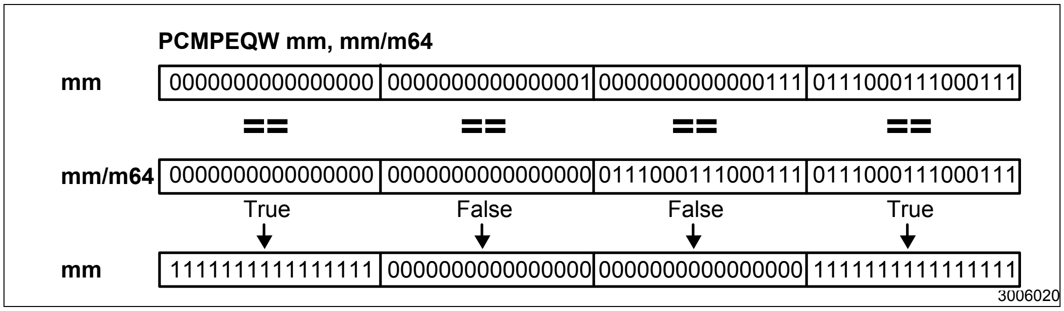
PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal

Opcode	Instruction	Description
0F 74 /r	PCMPEQB *mm, mm/m64*	Compare packed bytes in *mm/m64* with packed bytes in *mm* for equality.
0F 75 /r	PCMPEQW *mm, mm/m64*	Compare packed words in *mm/m64* with packed words in *mm* for equality.
0F 76 /r	PCMPEQD *mm, mm/m64*	Compare packed doublewords in *mm/m64* with packed doublewords in *mm* for equality.

Description

Compares the individual data elements (bytes, words, or doublewords) in the destination operand (first operand) to the corresponding data elements in the source operand (second operand). (See Figure 3-10.) If a pair of data elements are equal, the corresponding data element in the destination operand is set to all ones; otherwise, it is set to all zeros. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 3-10. Operation of the PCMPEQW Instruction



The PCMPEQB instruction compares the bytes in the destination operand to the corresponding bytes in the source operand, with the bytes in the destination operand being set according to the results.

The PCMPEQW instruction compares the words in the destination operand to the corresponding words in the source operand, with the words in the destination operand being set according to the results.

The PCMPEQD instruction compares the doublewords in the destination operand to the corresponding doublewords in the source operand, with the doublewords in the destination operand being set according to the results.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal (continued)

Operation

```
IF instruction is PCMPEQB
  THEN
      IF DEST(7..0) = SRC(7..0)
          THEN DEST(7  0) ← FFH;
          ELSE DEST(7..0) ← 0;
      * Continue comparison of second through seventh bytes in DEST and SRC *
      IF DEST(63..56) = SRC(63..56)
          THEN DEST(63..56) ← FFH;
          ELSE DEST(63..56) ← 0;
ELSE IF instruction is PCMPEQW
  THEN
      IF DEST(15..0) = SRC(15..0)
          THEN DEST(15..0) ← FFFFH;
          ELSE DEST(15..0) ← 0;
      * Continue comparison of second and third words in DEST and SRC *
      IF DEST(63..48) = SRC(63..48)
          THEN DEST(63..48) ← FFFFH;
          ELSE DEST(63..48) ← 0;
  ELSE (* instruction is PCMPEQD *)
      IF DEST(31..0) = SRC(31..0)
          THEN DEST(31..0) ← FFFFFFFFH;
          ELSE DEST(31..0) ← 0;
      IF DEST(63..32) = SRC(63..32)
          THEN DEST(63..32) ← FFFFFFFFH;
          ELSE DEST(63..32) ← 0;
FI;
```

Flags Affected

None:

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

PCMPEQB/PCMPEQW/PCMPEQD—Packed Compare for Equal (continued)

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

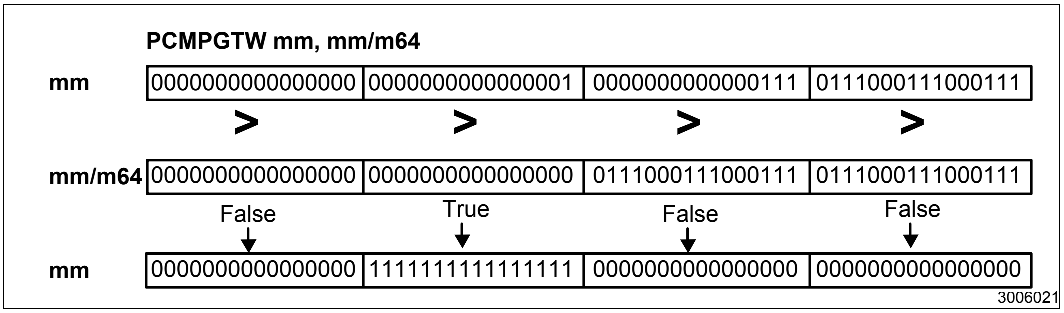
PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than

Opcode	Instruction	Description
0F 64 /r	PCMPGTB *mm, mm/m64*	Compare packed bytes in *mm* with packed bytes in *mm/m64* for greater value.
0F 65 /r	PCMPGTW *mm, mm/m64*	Compare packed words in *mm* with packed words in *mm/m64* for greater value.
0F 66 /r	PCMPGTD *mm, mm/m64*	Compare packed doublewords in *mm* with packed doublewords in *mm/m64* for greater value.

Description

Compare the individual signed data elements (bytes, words, or doublewords) in the destination operand (first operand) to the corresponding signed data elements in the source operand (second operand). (See Figure 3-11.) If a data element in the destination operand is greater than its corresponding data element in the source operand, the data element in the destination operand is set to all ones; otherwise, it is set to all zeros. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 3-11. Operation of the PCMPGTW Instruction



The PCMPGTB instruction compares the signed bytes in the destination operand to the corresponding signed bytes in the source operand, with the bytes in the destination operand being set according to the results.

The PCMPGTW instruction compares the signed words in the destination operand to the corresponding signed words in the source operand, with the words in the destination operand being set according to the results.

The PCMPGTD instruction compares the signed doublewords in the destination operand to the corresponding signed doublewords in the source operand, with the doublewords in the destination operand being set according to the results.

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than
(continued)

Operation

```
IF instruction is PCMPGTB
  THEN
      IF DEST(7..0) > SRC(7..0)
          THEN DEST(7  0) ← FFH;
          ELSE DEST(7..0) ← 0;
      * Continue comparison of second through seventh bytes in DEST and SRC *
      IF DEST(63..56) > SRC(63..56)
          THEN DEST(63..56) ← FFH;
          ELSE DEST(63..56) ← 0;
ELSE IF instruction is PCMPGTW
  THEN
      IF DEST(15..0) > SRC(15..0)
          THEN DEST(15..0) ← FFFFH;
          ELSE DEST(15..0) ←0;
      * Continue comparison of second and third bytes in DEST and SRC *
      IF DEST(63..48) > SRC(63..48)
          THEN DEST(63..48) ← FFFFH;
          ELSE DEST(63..48) ← 0;
  ELSE { (* instruction is PCMPGTD *)
      IF DEST(31..0) > SRC(31..0)
          THEN DEST(31..0) ← FFFFFFFFH;
          ELSE DEST(31..0) ← 0;
      IF DEST(63..32) > SRC(63..32)
          THEN DEST(63..32) ← FFFFFFFFH;
          ELSE DEST(63..32) ← 0;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PCMPGTB/PCMPGTW/PCMPGTD—Packed Compare for Greater Than (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PMADDWD—Packed Multiply and Add

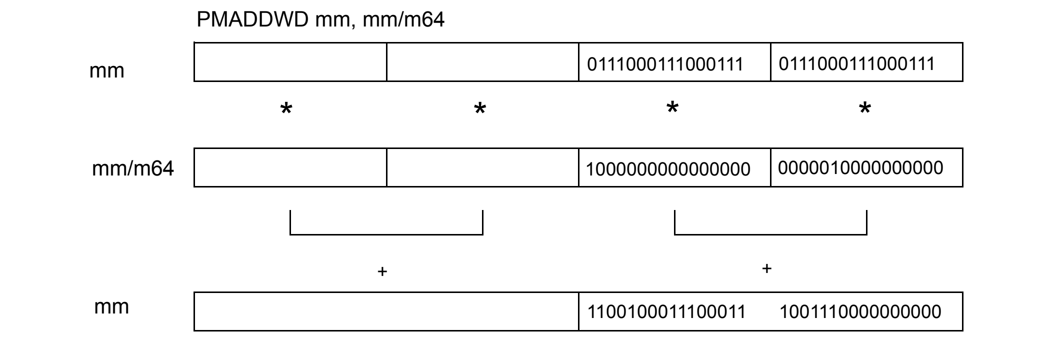
Opcode	Instruction	Description
0F F5 /r	PMADDWD *mm, mm/m64*	Multiply the packed words in *mm* by the packed words in *mm/m64*. Add the 32-bit pairs of results and store in *mm* as doubleword

Description

Multiplies the individual signed words of the destination operand by the corresponding signed words of the source operand, producing four signed, doubleword results (see Figure 3-12). The two doubleword results from the multiplication of the high-order words are added together and stored in the upper doubleword of the destination operand; the two doubleword results from the multiplication of the low-order words are added together and stored in the lower doubleword of the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

The PMADDWD instruction wraps around to 80000000H only when all four words of both the source and destination operands are 8000H.

Figure 3-12. Operation of the PMADDWD Instruction



Operation

```
DEST(31..0) ← (DEST(15..0) ∗ SRC(15..0)) + (DEST(31..16) ∗ SRC(31..16));
DEST(63..32) ← (DEST(47..32) ∗ SRC(47..32)) + (DEST(63..48) ∗ SRC(63..48));
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMADDWD—Packed Multiply and Add (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

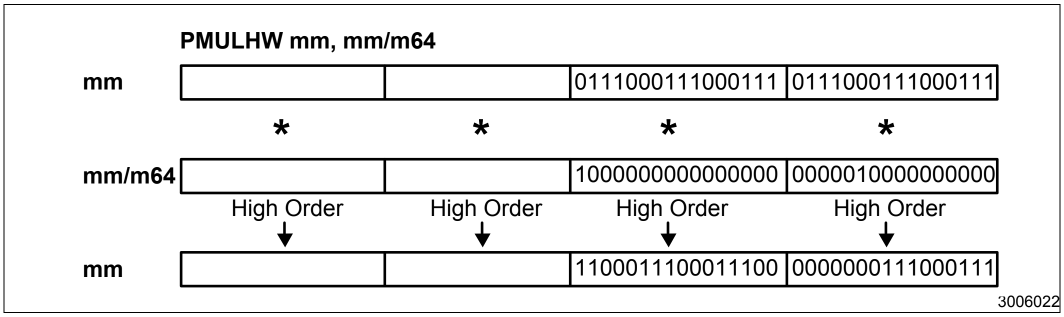
PMULHW—Packed Multiply High

Opcode	Instruction	Description
0F E5 /r	PMULHW *mm, mm/m64*	Multiply the signed packed words in *mm* by the signed packed words in *mm/m64*, then store the high-order word of each doubleword result in *mm*.

Description

Multiplies the four signed words of the source operand (second operand) by the four signed words of the destination operand (first operand), producing four signed, doubleword, intermediate results (see Figure 3-13). The high-order word of each intermediate result is then written to its corresponding word location in the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 3-13. Operation of the PMULHW Instruction



Operation

```
DEST(15..0) ← HighOrderWord(DEST(15..0) ∗ SRC(15..0));
DEST(31..16) ← HighOrderWord(DEST(31..16) ∗ SRC(31..16));
DEST(47..32) ← HighOrderWord(DEST(47..32) ∗ SRC(47..32));
DEST(63..48) ← HighOrderWord(DEST(63..48) ∗ SRC(63..48));
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMULHW—Packed Multiply High (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

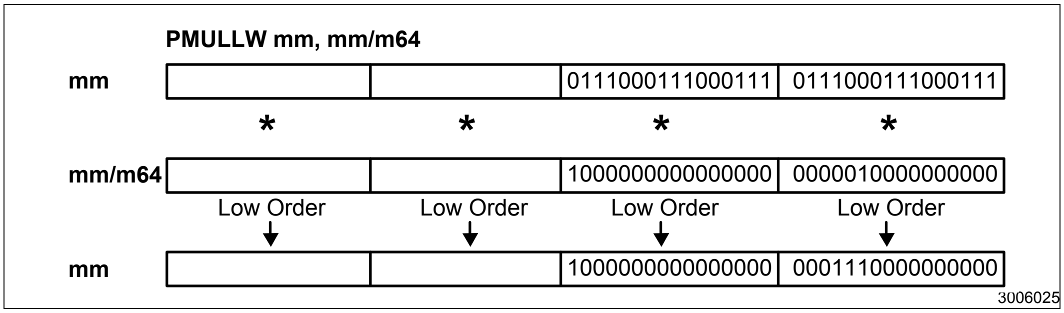
PMULLW—Packed Multiply Low

Opcode	Instruction	Description
0F D5 /r	PMULLW *mm, mm/m64*	Multiply the packed words in *mm* with the packed words in *mm/m64*, then store the low-order word of each doubleword result in *mm*.

Description

Multiplies the four signed or unsigned words of the source operand (second operand) with the four signed or unsigned words of the destination operand (first operand), producing four doubleword, intermediate results (see Figure 3-14). The low-order word of each intermediate result is then written to its corresponding word location in the destination operand. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location.

Figure 3-14. Operation of the PMULLW Instruction



Operation

```
DEST(15..0) ← LowOrderWord(DEST(15..0) ∗ SRC(15..0));
DEST(31..16) ← LowOrderWord(DEST(31..16) ∗ SRC(31..16));
DEST(47..32) ← LowOrderWord(DEST(47..32) ∗ SRC(47..32));
DEST(63..48) ← LowOrderWord(DEST(63..48) ∗ SRC(63..48));
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PMULLW—Packed Multiply Low (continued)

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

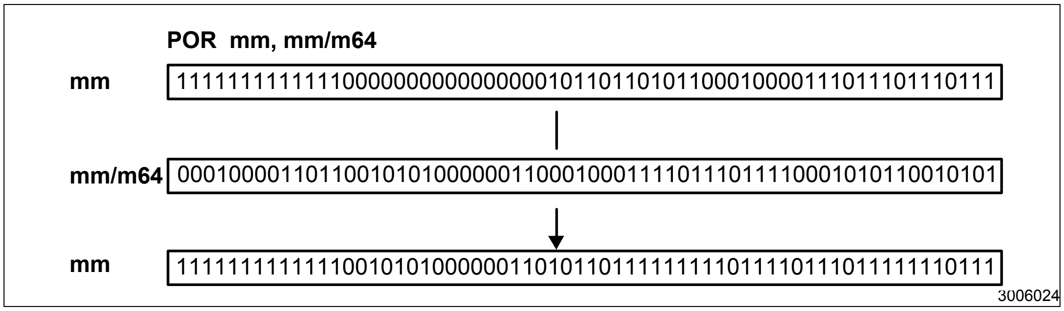
POR—Bitwise Logical OR

Opcode	Instruction	Description
0F EB /r	POR *mm, mm/m64*	OR quadword from mm/m64 to quadword in mm.

Description

Performs a bitwise logical OR operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 3-15). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result is made 0 if the corresponding bits of both operands are 0; otherwise the bit is set to 1.

Figure 3-15. Operation of the POR Instruction.



Operation

DEST ← DEST OR SRC;

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

POR—Bitwise Logical OR (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical

Opcode	Instruction	Description
0F F1 /r	PSLLW *mm, mm/m64*	Shift words in *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 71 /6, ib	PSLLW *mm, imm8*	Shift words in *mm* left by *imm8*, while shifting in zeros.
0F F2 /r	PSLLD *mm, mm/m64*	Shift doublewords in *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 72 /6 ib	PSLLD *mm, imm8*	Shift doublewords in *mm* by *imm8*, while shifting in zeros.
0F F3 /r	PSLLQ *mm, mm/m64*	Shift *mm* left by amount specified in *mm/m64*, while shifting in zeros.
0F 73 /6 ib	PSLLQ *mm, imm8*	Shift *mm* left by Imm8, while shifting in zeros.

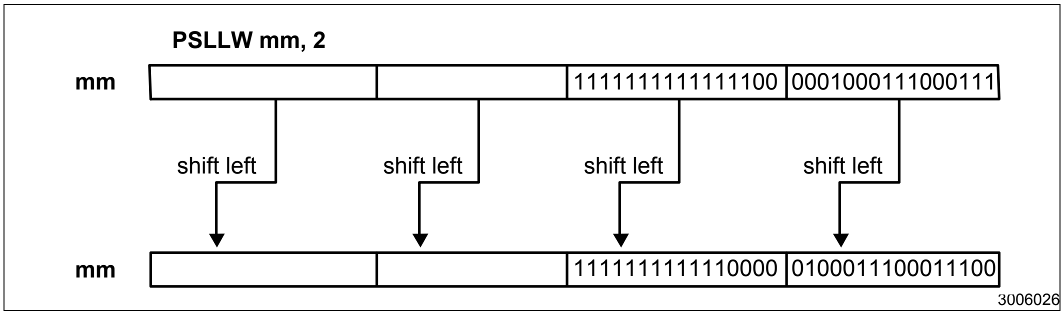
Description

Shifts the bits in the data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the unsigned count operand (second operand). (See Figure 3-16.) The result of the shift operation is written to the destination operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to zero). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all zeros.

The destination operand must be an MMX technology register; the count operand can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSLLW instruction shifts each of the four words of the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the two doublewords of the destination operand; and the PSLLQ instruction shifts the 64-bit quadword in the destination operand. As the individual data elements are shifted left, the empty low-order bit positions are filled with zeros.

Figure 3-16. Operation of the PSLLW Instruction



PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical (continued)

Operation

```
IF instruction is PSLLW
   THEN
         DEST(15..0) ← DEST(15..0) << COUNT;
         DEST(31..16) ← DEST(31..16) << COUNT;
         DEST(47..32) ← DEST(47..32) << COUNT;
         DEST(63..48) ← DEST(63..48) << COUNT;
   ELSE IF instruction is PSLLD
         THEN {
               DEST(31..0) ← DEST(31..0) << COUNT;
               DEST(63..32) ← DEST(63..32) << COUNT;
         ELSE  (* instruction is PSLLQ *)
               DEST ← DEST << COUNT;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

PSLLW/PSLLD/PSLLQ—Packed Shift Left Logical (continued)

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PSRAW/PSRAD—Packed Shift Right Arithmetic

Opcode	Instruction	Description
0F E1 /r	PSRAW *mm, mm/m64*	Shift words in *mm* right by amount specified in *mm/m64* while shifting in sign bits.
0F 71 /4 ib	PSRAW *mm, imm8*	Shift words in *mm* right by *imm8* while shifting in sign bits
0F E2 /r	PSRAD *mm, mm/m64*	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in sign bits.
0F 72 /4 ib	PSRAD *mm, imm8*	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.

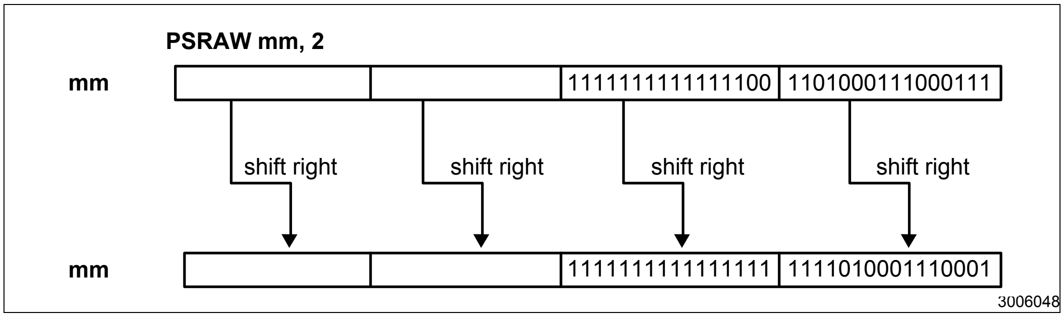
Description

Shifts the bits in the data elements (words or doublewords) in the destination operand (first operand) to the right by the amount of bits specified in the unsigned count operand (second operand). (See Figure 3-17.) The result of the shift operation is written to the destination operand. The empty high-order bits of each element are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element.

The destination operand must be an MMX technology register; the count operand (source operand) can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSRAW instruction shifts each of the four words in the destination operand to the right by the number of bits specified in the count operand; the PSRAD instruction shifts each of the two doublewords in the destination operand. As the individual data elements are shifted right, the empty high-order bit positions are filled with the sign value.

Figure 3-17. Operation of the PSRAW Instruction



PSRAW/PSRAD—Packed Shift Right Arithmetic (continued)

Operation

```
IF instruction is PSRAW
  THEN
      DEST(15..0) ← SignExtend (DEST(15..0) >> COUNT);
      DEST(31..16) ← SignExtend (DEST(31..16) >> COUNT);
      DEST(47..32) ← SignExtend (DEST(47..32) >> COUNT);
      DEST(63..48) ← SignExtend (DEST(63..48) >> COUNT);
  ELSE { (*instruction is PSRAD *)
      DEST(31..0) ← SignExtend (DEST(31..0) >> COUNT);
      DEST(63..32) ← SignExtend (DEST(63..32) >> COUNT);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

PSRAW/PSRAD—Packed Shift Right Arithmetic (continued)

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical

Opcode	Instruction	Description
0F D1 /r	PSRLW *mm, mm/m64*	Shift words in *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 71 /2 ib	PSRLW *mm, imm8*	Shift words in *mm* right by *imm8*.
0F D2 /r	PSRLD *mm, mm/m64*	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 72 /2 ib	PSRLD *mm, imm8*	Shift doublewords in *mm* right by *imm8*.
0F D3 /r	PSRLQ *mm, mm/m64*	Shift *mm* right by amount specified in *mm/m64* while shifting in zeros.
0F 73 /2 ib	PSRLQ *mm, imm8*	Shift *mm* right by *imm8* while shifting in zeros.

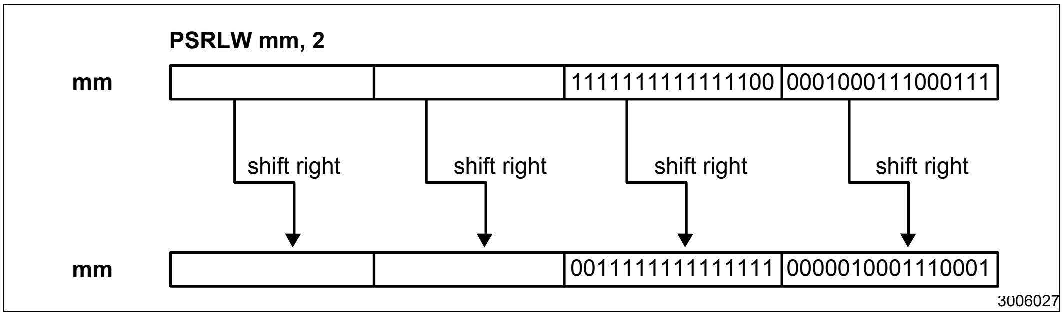
Description

Shifts the bits in the data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the unsigned count operand (second operand). (See Figure 3-18.) The result of the shift operation is written to the destination operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to zero). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all zeros.

The destination operand must be an MMX technology register; the count operand can be either an MMX technology register, a 64-bit memory location, or an 8-bit immediate.

The PSRLW instruction shifts each of the four words of the destination operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the two doublewords of the destination operand; and the PSRLQ instruction shifts the 64-bit quadword in the destination operand. As the individual data elements are shifted right, the empty high-order bit positions are filled with zeros.

Figure 3-18. Operation of the PSRLW Instruction



PSRLW/PSRLD/PSRLQ—Packed Shift Right Logical (continued)

Operation

```
IF instruction is PSRLW
   THEN {
         DEST(15..0) ← DEST(15..0) >> COUNT;
         DEST(31..16) ← DEST(31..16) >> COUNT;
         DEST(47..32) ← DEST(47..32) >> COUNT;
         DEST(63..48) ← DEST(63..48) >> COUNT;
ELSE IF instruction is PSRLD
   THEN {
         DEST(31..0) ← DEST(31..0) >> COUNT;
         DEST(63..32) ← DEST(63..32) >> COUNT;
   ELSE  (* instruction is PSRLQ *)
         DEST ← DEST >> COUNT;
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

PSUBB/PSUBW/PSUBD—Packed Subtract

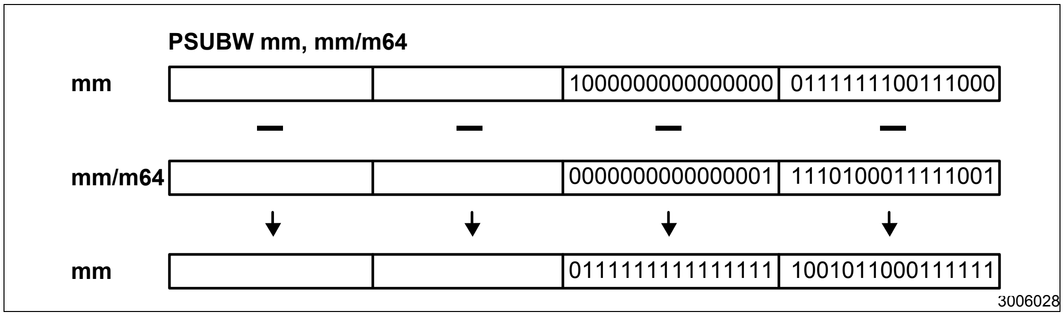
Opcode	Instruction	Description
0F F8 /r	PSUBB *mm, mm/m64*	Subtract packed bytes in *mm/m64* from packed bytes in *mm*.
0F F9 /r	PSUBW *mm, mm/m64*	Subtract packed words in*mm/m64* from packed words in *mm*.
0F FA /r	PSUBD *mm, mm/m64*	Subtract packed doublewords in *mm/m64* from packed doublewords in *mm*.

Description

Subtracts the individual data elements (bytes, words, or doublewords) of the source operand (second operand) from the individual data elements of the destination operand (first operand). (See Figure 3-19.) If the result of a subtraction exceeds the range for the specified data type (overflows), the result is wrapped around, meaning that the result is truncated so that only the lower (least significant) bits of the result are returned (that is, the carry is ignored).

The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-19. Operation of the PSUBW Instruction



The PSUBB instruction subtracts the bytes of the source operand from the bytes of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 8 bits, the lower 8 bits of the result are written to the destination operand and therefore the result wraps around.

The PSUBW instruction subtracts the words of the source operand from the words of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 16 bits, the lower 16 bits of the result are written to the destination operand and therefore the result wraps around.

The PSUBD instruction subtracts the doublewords of the source operand from the doublewords of the destination operand and stores the results to the destination operand. When an individual result is too large to be represented in 32 bits, the lower 32 bits of the result are written to the destination operand and therefore the result wraps around.

PSUBB/PSUBW/PSUBD—Packed Subtract (continued)

Note that like the integer SUB instruction, the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers. Unlike the integer instructions, none of the MMX technology instructions affect the EFLAGS register. With MMX technology instructions, there are no carry or overflow flags to indicate when overflow has occurred, so the software must control the range of values or else use the "with saturation" MMX technology instructions.

Operation

```
IF instruction is PSUBB
  THEN
      DEST(7..0) ← DEST(7..0) - SRC(7..0);
      DEST(15..8) ← DEST(15..8) - SRC(15..8);
      DEST(23..16) ← DEST(23..16) - SRC(23..16);
      DEST(31..24) ← DEST(31..24) - SRC(31..24);
      DEST(39..32) ← DEST(39..32) - SRC(39..32);
      DEST(47..40) ← DEST(47..40) - SRC(47..40);
      DEST(55..48) ← DEST(55..48) - SRC(55..48);
      DEST(63..56) ← DEST(63..56) - SRC(63..56);
ELSEIF instruction is PSUBW
  THEN
      DEST(15..0) ← DEST(15..0) - SRC(15..0);
      DEST(31..16) ← DEST(31..16) - SRC(31..16);
      DEST(47..32) ← DEST(47..32) - SRC(47..32);
      DEST(63..48) ← DEST(63..48) - SRC(63..48);
  ELSE { (* instruction is PSUBD *)
      DEST(31..0) ← DEST(31..0) - SRC(31..0);
      DEST(63..32) ← DEST(63..32) - SRC(63..32);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PSUBB/PSUBW/PSUBD—Packed Subtract (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

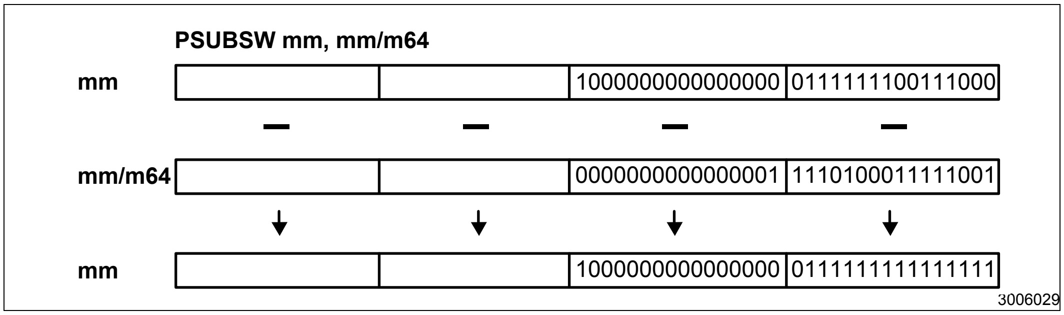
PSUBSB/PSUBSW—Packed Subtract with Saturation

Opcode	Instruction	Description
0F E8 /r	PSUBSB *mm, mm/m64*	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate.
0F E9 /r	PSUBSW *mm, mm/m64*	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate.

Description

Subtracts the individual signed data elements (bytes or words) of the source operand (second operand) from the individual signed data elements of the destination operand (first operand). (See Figure 3-20.) If the result of a subtraction exceeds the range for the specified data type, the result is saturated. The destination operand must be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-20. Operation of the PSUBSW Instruction



The PSUBSB instruction subtracts the signed bytes of the source operand from the signed bytes of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed byte (that is, greater than 7FH or less than 80H), the saturated byte value of 7FH or 80H, respectively, is written to the destination operand.

The PSUBSW instruction subtracts the signed words of the source operand from the signed words of the destination operand and stores the results to the destination operand. When an individual result is beyond the range of a signed word (that is, greater than 7FFFH or less than 8000H), the saturated word value of 7FFFH or 8000H, respectively, is written to the destination operand.

PSUBSB/PSUBSW—Packed Subtract with Saturation (continued)

Operation

```
IF instruction is PSUBSB
   THEN
         DEST(7..0) ← SaturateToSignedByte(DEST(7..0) - SRC (7..0));
         DEST(15..8) ← SaturateToSignedByte(DEST(15..8) - SRC(15..8));
         DEST(23..16) ← SaturateToSignedByte(DEST(23..16) - SRC(23..16));
         DEST(31..24) ← SaturateToSignedByte(DEST(31..24) - SRC(31..24));
         DEST(39..32) ← SaturateToSignedByte(DEST(39..32) - SRC(39..32));
         DEST(47..40) ← SaturateToSignedByte(DEST(47..40) - SRC(47..40));
         DEST(55..48) ← SaturateToSignedByte(DEST(55..48) - SRC(55..48));
         DEST(63..56) ← SaturateToSignedByte(DEST(63..56) - SRC(63..56))
   ELSE (* instruction is PSUBSW *)
         DEST(15..0) ← SaturateToSignedWord(DEST(15..0) - SRC(15..0));
         DEST(31..16) ← SaturateToSignedWord(DEST(31..16) - SRC(31..16));
         DEST(47..32) ← SaturateToSignedWord(DEST(47..32) - SRC(47..32));
         DEST(63..48) ← SaturateToSignedWord(DEST(63..48) - SRC(63..48));
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PSUBSB/PSUBSW—Packed Subtract with Saturation (continued)

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

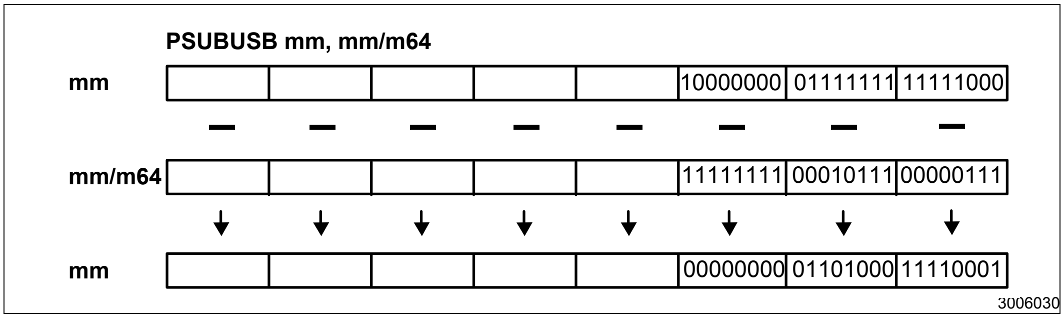
PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation

Opcode	Instruction	Description
0F D8 /r	PSUBUSB *mm, mm/m64*	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate.
0F D9 /r	PSUBUSW *mm, mm/m64*	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate.

Description

Subtracts the individual unsigned data elements (bytes or words) of the source operand (second operand) from the individual unsigned data elements of the destination operand (first operand). (See Figure 3-21.) If the result of an individual subtraction exceeds the range for the specified unsigned data type, the result is saturated. The destination operand musts be an MMX technology register; the source operand can be either an MMX technology register or a quadword memory location.

Figure 3-21. Operation of the PSUBUSB Instruction



The PSUBUSB instruction subtracts the unsigned bytes of the source operand from the unsigned bytes of the destination operand and stores the results to the destination operand. When an individual result is less than zero (a negative value), the saturated unsigned byte value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts the unsigned words of the source operand from the unsigned words of the destination operand and stores the results to the destination operand. When an individual result is less than zero (a negative value), the saturated unsigned word value of 0000H is written to the destination operand.

PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation (continued)

Operation

```
IF instruction is PSUBUSB
  THEN
      DEST(7..0) ← SaturateToUnsignedByte (DEST(7..0 - SRC (7..0) );
      DEST(15..8) ← SaturateToUnsignedByte ( DEST(15..8) - SRC(15..8) );
      DEST(23..16) ← SaturateToUnsignedByte (DEST(23..16) - SRC(23..16) );
      DEST(31..24) ← SaturateToUnsignedByte (DEST(31..24) - SRC(31..24) );
      DEST(39..32) ← SaturateToUnsignedByte (DEST(39..32) - SRC(39..32) );
      DEST(47..40) ← SaturateToUnsignedByte (DEST(47..40) - SRC(47..40) );
      DEST(55..48) ← SaturateToUnsignedByte (DEST(55..48) - SRC(55..48) );
      DEST(63..56) ← SaturateToUnsignedByte (DEST(63..56) - SRC(63..56) );
  ELSE { (* instruction is PSUBUSW *)
      DEST(15..0) ← SaturateToUnsignedWord (DEST(15..0) - SRC(15..0) );
      DEST(31..16) ← SaturateToUnsignedWord (DEST(31..16) - SRC(31..16) );
      DEST(47..32) ← SaturateToUnsignedWord (DEST(47..32) - SRC(47..32) );
      DEST(63..48) ← SaturateToUnsignedWord (DEST(63..48) - SRC(63..48) );
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PSUBUSB/PSUBUSW—Packed Subtract Unsigned with Saturation (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

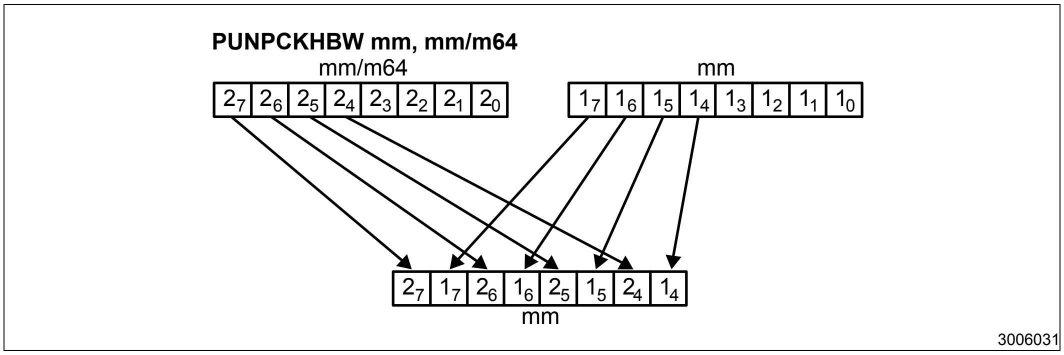
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data

Opcode	Instruction	Description
0F 68 /r	PUNPCKHBW *mm, mm/m64*	Interleave high-order bytes from *mm* and *mm/m64* into *mm*.
0F 69 /r	PUNPCKHWD *mm, mm/m64*	Interleave high-order words from *mm* and *mm/m64* into *mm*.
0F 6A /r	PUNPCKHDQ *mm, mm/m64*	Interleave high-order doublewords from *mm* and *mm/m64* into *mm*.

Description

Unpacks and interleaves the high-order data elements (bytes, words, or doublewords) of the destination operand (first operand) and source operand (second operand) into the destination operand (see Figure 3-22). The low-order data elements are ignored. The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a 64-bit memory location. When the source data comes from a memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits.

Figure 3-22. High-order Unpacking and Interleaving of Bytes with the PUNPCKHBW Instruction



The PUNPCKHBW instruction interleaves the four high-order bytes of the source operand and the four high-order bytes of the destination operand and writes them to the destination operand.

The PUNPCKHWD instruction interleaves the two high-order words of the source operand and the two high-order words of the destination operand and writes them to the destination operand.

The PUNPCKHDQ instruction interleaves the high-order doubleword of the source operand and the high-order doubleword of the destination operand and writes them to the destination operand.

If the source operand is all zeros, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. With the PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned words), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doublewords).

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data (continued)

Operation

```
IF instruction is PUNPCKHBW
   THEN
         DEST(7..0) ← DEST(39..32);
         DEST(15..8) ← SRC(39..32);
         DEST(23..16) ← DEST(47..40);
         DEST(31..24) ← SRC(47..40);
         DEST(39..32) ← DEST(55..48);
         DEST(47..40) ← SRC(55..48);
         DEST(55..48) ← DEST(63..56);
         DEST(63..56) ← SRC(63..56);
ELSE IF instruction is PUNPCKHW
   THEN
         DEST(15..0) ← DEST(47..32);
         DEST(31..16) ← SRC(47..32);
         DEST(47..32) ← DEST(63..48);
         DEST(63..48) ← SRC(63..48);
   ELSE (* instruction is PUNPCKHDQ *)
         DEST(31..0) ← DEST(63..32)
         DEST(63..32) ← SRC(63..32);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem FaultsVHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ—Unpack High Packed Data (continued)

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

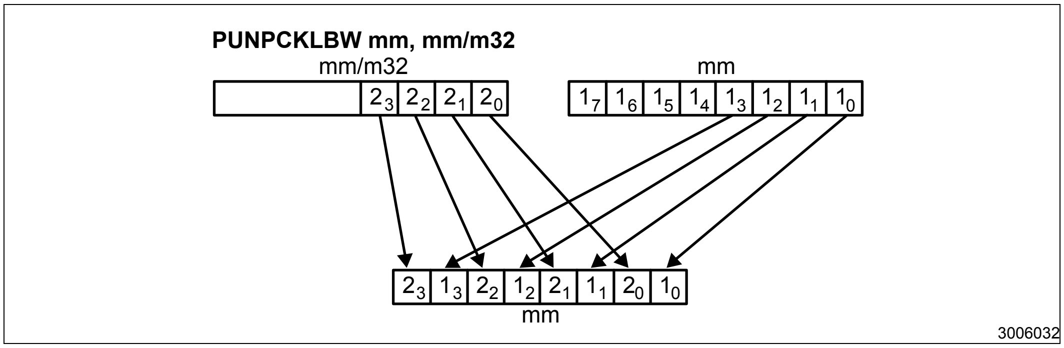
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data

Opcode	Instruction	Description
0F 60 /r	PUNPCKLBW *mm, mm/m32*	Interleave low-order bytes from *mm* and *mm/m64* into *mm*.
0F 61 /r	PUNPCKLWD *mm, mm/m32*	Interleave low-order words from *mm* and *mm/m64* into *mm*.
0F 62 /r	PUNPCKLDQ *mm, mm/m32*	Interleave low-order doublewords from *mm* and *mm/m64* into *mm*.

Description

Unpacks and interleaves the low-order data elements (bytes, words, or doublewords) of the destination and source operands into the destination operand (see Figure 3-23). The destination operand must be an MMX technology register; the source operand may be either an MMX technology register or a memory location. When source data comes from an MMX technology register, the upper 32 bits of the register are ignored. When the source data comes from a memory, only 32-bits are accessed from memory.

Figure 3-23. Low-order Unpacking and Interleaving of Bytes with the PUNPCKLBW Instruction



The PUNPCKLBW instruction interleaves the four low-order bytes of the source operand and the four low-order bytes of the destination operand and writes them to the destination operand.

The PUNPCKLWD instruction interleaves the two low-order words of the source operand and the two low-order words of the destination operand and writes them to the destination operand.

The PUNPCKLDQ instruction interleaves the low-order doubleword of the source operand and the low-order doubleword of the destination operand and writes them to the destination operand.

If the source operand is all zeros, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. With the PUNPCKLBW instruction the low-order bytes are zero extended (that is, unpacked into unsigned words), and with the PUNPCKLWD instruction, the low-order words are zero extended (unpacked into unsigned doublewords).

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data (continued)

Operation

```
IF instruction is PUNPCKLBW
  THEN
      DEST(63..56) ← SRC(31..24);
      DEST(55..48) ← DEST(31..24);
      DEST(47..40) ← SRC(23..16);
      DEST(39..32) ← DEST(23..16);
      DEST(31..24) ← SRC(15..8);
      DEST(23..16) ← DEST(15..8);
      DEST(15..8) ← SRC(7..0);
      DEST(7..0) ← DEST(7..0);
ELSE IF instruction is PUNPCKLWD
  THEN
      DEST(63..48) ← SRC(31..16);
      DEST(47..32) ← DEST(31..16);
      DEST(31..16) ← SRC(15..0);
      DEST(15..0) ← DEST(15..0);
  ELSE (* instruction is PUNPCKLDQ *)
      DEST(63..32) ← SRC(31..0);
      DEST(31..0) ← DEST(31..0);
FI;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.
Itanium Mem Faults	VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
#NM	If TS in CR0 is set.
#MF	If there is a pending FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ—Unpack Low Packed Data (continued)

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

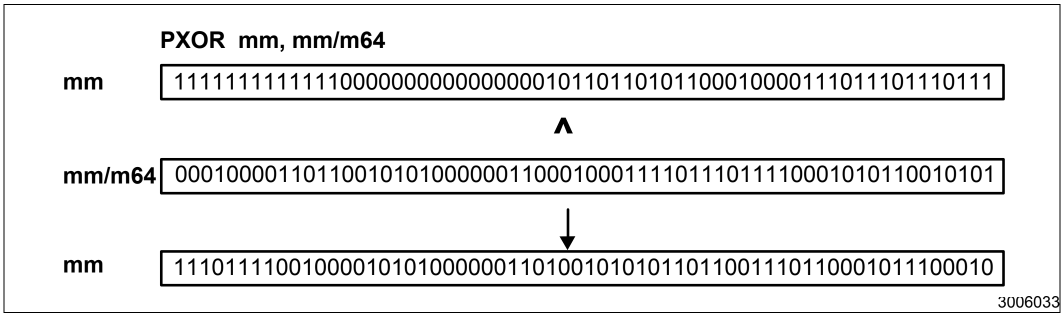
PXOR—Logical Exclusive OR

Opcode	Instruction	Description
0F EF /r	PXOR *mm, mm/m64*	XOR quadword from *mm/m64* to quadword in *mm*.

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the quadword source (second) and destination (first) operands and stores the result in the destination operand location (see Figure 3-24). The source operand can be an MMX technology register or a quadword memory location; the destination operand must be an MMX technology register. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

Figure 3-24. Operation of the PXOR Instruction



Operation

```
DEST ← DEST XOR SRC;
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Abort.

Itanium Mem Faults VHPT Data Fault, Nested TLB Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

PXOR—Logical Exclusive OR (continued)

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

Virtual-8086 Mode Exceptions

#GP If any part of the operand lies outside of the effective address space from 0 to FFFFH.

#UD If EM in CR0 is set.

#NM If TS in CR0 is set.

#MF If there is a pending FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

§

IA-32 SSE Instruction Reference 4

4.1 IA-32 SSE Instructions

This section lists the IA-32 SSE instructions designed to increase performance of IA-32 3D and floating-point intensive applications. For details on SSE please refer to the ***Intel® 64 and IA-32 Architectures Software Developer's Manual***.

4.2 About the Intel® SSE Architecture

The Intel SSE architecture accelerates performance of 3D graphics applications over the current P6 generation of the Pentium Pro, Pentium II and Pentium III processors. The programming model is similar to the MMX technology model except that instructions now operate on new packed floating-point data types which contain four single-precision floating-point numbers.

The Intel SSE architecture introduces new general purpose floating-point instructions, which operate on a new set of eight 128-bit SSE registers. This gives the programmer the ability to develop algorithms that can finely mix packed single-precision floating-point and integer using both SSE and MMX technology instructions respectively. In addition to these instructions, the Intel SSE architecture also provides new instructions to control cacheability of all MMX technology data types. These include ability to stream data into and from the processor while minimizing pollution of the caches and the ability to prefetch data before it is actually used. The main focus of packed floating-point instructions is the acceleration of 3D geometry. The new definition also contains additional SIMD Integer instructions to accelerate 3D rendering and video encoding and decoding. Together with the cacheability control instruction, this combination enables the development of new algorithms that can significantly accelerate 3D graphics.

The new SSE state requires OS support for saving and restoring the new state during a context switch. A new set of extended FSAVE/FRSTOR instructions will permit saving/restoring new and existing state for applications and OS. To make use of these new instructions, an application must verify that the processor supports the Intel SSE architecture and the operating system supports this new extension. If both the extension and support is enabled, then the software application can use the new features.

The SSE instruction set is fully compatible with all software written for Intel architecture microprocessors. All existing software continues to run correctly, without modification, on microprocessors that incorporate the Intel SSE architecture, as well as in the presence of existing and new applications that incorporate this technology.

4.3 Single Instruction Multiple Data

The Intel SSE architecture uses the Single Instruction Multiple Data (SIMD) technique. This technique speeds up software performance by processing multiple data elements in parallel, using a single instruction. The Intel SSE architecture supports operations on packed single-precision floating-point data types, and the additional SIMD Integer instructions support operations on packed quadrate data types (byte, word, or double-word). This approach was chosen because most 3D graphics and DSP applications have the following characteristics:

- Inherently parallel
- Wide dynamic range, hence floating-point based
- Regular and re-occurring memory access patterns
- Localized re-occurring operations performed on the data
- Data independent control flow

The Intel SSE architecture is 100% compatible with the IEEE Standard 754 for Binary Floating-point Arithmetic. The SSE instructions are accessible from all IA execution modes: Protected mode, Real address mode, and Virtual 8086 mode.New Features

The Intel SSE architecture provides the following new features, while maintaining backward compatibility with all existing Intel architecture microprocessors, IA applications and operating systems.

- New data type
- Eight SSE registers
- Enhanced instruction set

The Intel SSE architecture can enhance the performance of applications that use these features.

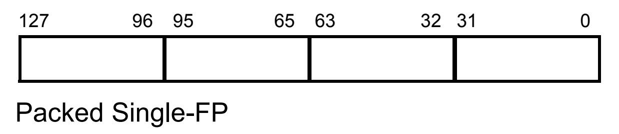
4.4 New Data Types

The principal data type of the Intel SSE architecture is a packed single-precision floating-point operand, specifically:

- Four 32-bit single-precision (SP) floating-point numbers (Figure 4-1).

The SIMD Integer instructions will operate on the packed byte, word or doubleword data types. The prefetch instruction works on typeless data of size 32 bytes or greater.

Figure 4-1. Packed Single-FP Data Type



4.5 SSE Registers

The Intel SSE architecture provides eight 128-bit general purpose registers, each of which can be directly addressed. These registers are new state, and require support from the operating system to use them.

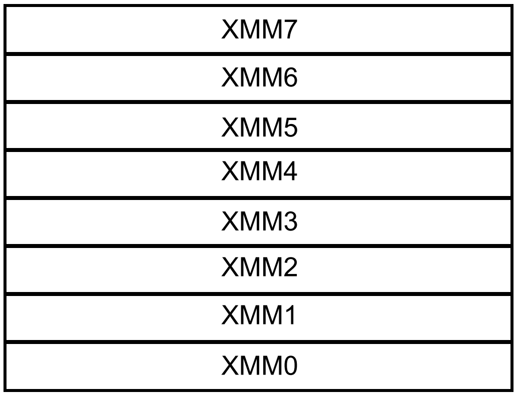
The SSE registers can hold packed 128-bit data. The SSE instructions access the SSE registers directly using the registers names XMM0 to XMM7 (Figure 4-2).

SSE registers can be used to perform calculation on data. They cannot be used to address memory; addressing is accomplished by using the integer registers and existing IA addressing modes.

The contents of SSE registers are cleared upon reset.

There is a new control/status register MXCSR which is used to mask/unmask numerical exception handling, to set rounding modes, to set flush-to-zero mode, and to view status flags.

Figure 4-2. SSE Register Set



4.6 Extended Instruction Set

The Intel SSE architecture supplies a rich set of instructions that operate on either all or the least significant pairs of packed data operands, in parallel. The packed instructions operate on a pair of operands as shown in Figure 4-3 while scalar instructions always operate on the least significant pair of the two operands as shown in Figure 4-4; for scalar operations, the three upper components from the first operand are passed through to the destination. In general, the address of a memory operand has to be aligned on a 16-byte boundary for all instructions, except for unaligned loads and stores.

Figure 4-3. Packed Operation

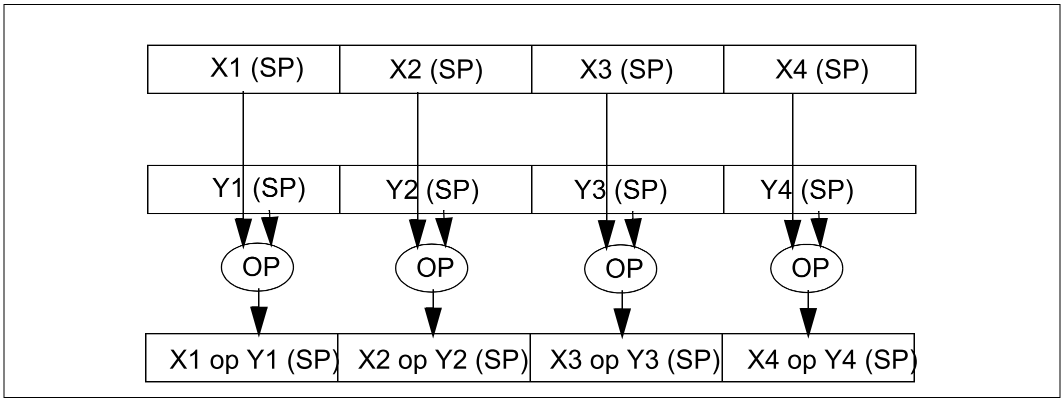
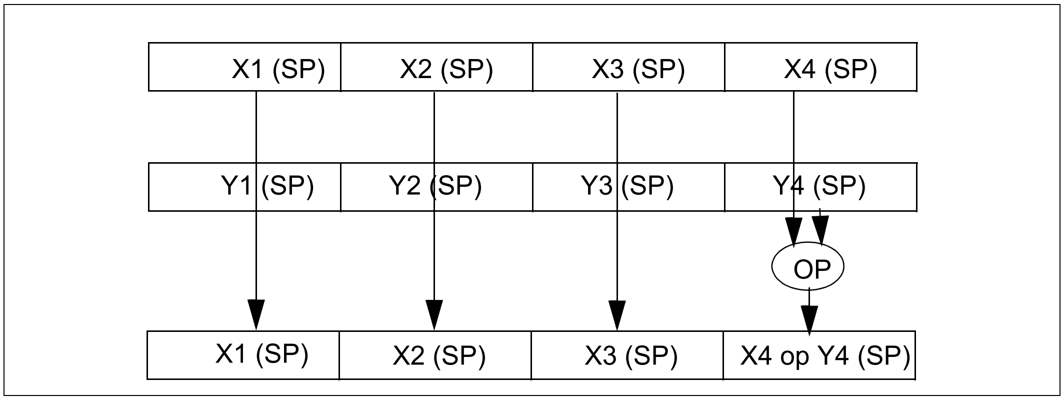


Figure 4-4. Scalar Operation



4.6.1 Instruction Group Review

4.6.1.1 Arithmetic Instructions

Packed/Scalar Addition and Subtraction

The ADDPS (Add packed single-precision floating-point) and SUBPS (Subtract packed single-precision floating-point) instructions add or subtract four pairs of packed single-precision floating-point operands.

The ADDSS (Add scalar single-precision floating-point) and SUBSS (Subtract scalar single-precision floating-point) instructions add or subtract the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

Packed/Scalar Multiplication and Division

The MULPS (Multiply packed single-precision floating-point) instruction multiplies four pairs of packed single-precision floating-point operands.

The MULSS (Multiply scalar single-precision floating-point) instruction multiplies the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

The DIVPS (Divide packed single-precision floating-point) instruction divides four pairs of packed single-precision floating-point operands.

The DIVSS (Divide scalar single-precision floating-point) instruction divides the least significant pair of packed single-precision floating-point operands; the upper three fields are passed through from the source operand.

Packed/Scalar Square Root

The SQRTPS (Square root packed single-precision floating-point) instruction returns the square root of the packed four single-precision floating-point numbers from the source to a destination register.

The SQRTSS (Square root scalar single-precision floating-point) instruction returns the square root of the least significant component of the packed single-precision floating-point numbers from source to a destination register; the upper three fields are passed through from the source operand.

Packed Maximum/Minimum

The MAXPS (Maximum packed single-precision floating-point) instruction returns the maximum of each pair of packed single-precision floating-point numbers into the destination register.

The MAXSS (Maximum scalar single-precision floating-point) instructions returns the maximum of the least significant pair of packed single-precision floating-point numbers into the destination register; the upper three fields are passed through from the source operand, to the destination register.

The MINPS (Minimum packed single-precision floating-point) instruction returns the minimum of each pair of packed single-precision floating-point numbers into the destination register.

The MINSS (Minimum scalar single-precision floating-point) instruction returns the minimum of the least significant pair of packed single-precision floating-point numbers into the destination register; the upper three fields are passed through from the source operand, to the destination register

4.6.1.2 Logical Instructions

The ANDPS (Bit-wise packed logical AND for single-precision floating-point) instruction returns a bitwise AND between the two operands.

The ANDNPS (Bit-wise packed logical AND NOT for single-precision floating-point) instruction returns a bitwise AND NOT between the two operands.

The ORPS (Bit-wise packed logical OR for single-precision floating-point) instruction returns a bitwise OR between the two operands.

The XORPS (Bit-wise packed logical XOR for single-precision floating-point) instruction returns a bitwise XOR between the two operands.

4.6.1.3 Compare Instructions

The CMPPS (Compare packed single-precision floating-point) instruction compares four pairs of packed single-precision floating-point numbers using the immediate operand as a predicate, returning per SP field an all "1" 32-bit mask or an all "0" 32-bit mask as a result. The instruction supports a full set of 12 conditions: equal, less than, less than equal, greater than, greater than or equal, unordered, not equal, not less than, not less than or equal, not greater than, not greater than or equal, ordered.

The CMPSS (Compare scalar single-precision floating-point) instruction compares the least significant pairs of packed single-precision floating-point numbers using the immediate operand as a predicate (same as CMPPS), returning per SP field an all "1" 32-bit mask or an all "0" 32-bit mask as a result.

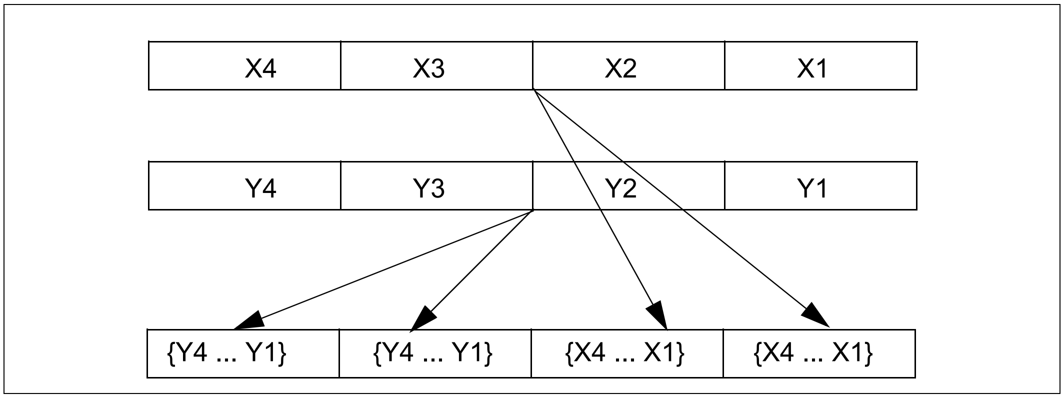
The COMISS (Compare scalar single-precision floating-point ordered and set EFLAGS) instruction compares the least significant pairs of packed single-precision floating-point numbers and sets the ZF,PF,CF bits in the EFLAGS register (the OF, SF and AF bits are cleared).

The UCOMISS (Unordered compare scalar single-precision floating-point ordered and set EFLAGS) instruction compares the least significant pairs of packed single-precision floating-point numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above (the OF, SF and AF bits are cleared).

4.6.1.4 Shuffle Instructions

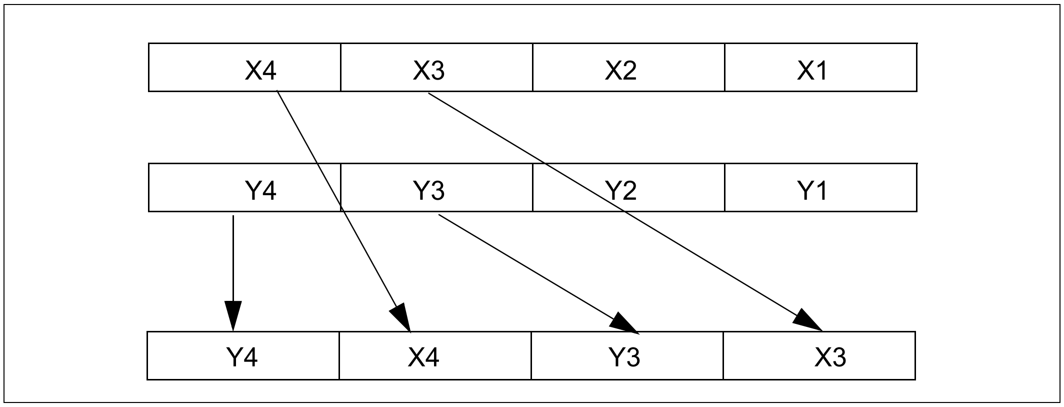
The SHUFPS (Shuffle packed single-precision floating-point) instruction is able to shuffle any of the packed four single-precision floating-point numbers from one source operand to the lower two destination fields; the upper two destination fields are generated from a shuffle of any of the four SP FP numbers from the second source operand (Figure 4-5). By using the same register for both sources, SHUFPS can return any combination of the four SP FP numbers from this register.

Figure 4-5. Packed Shuffle Operation



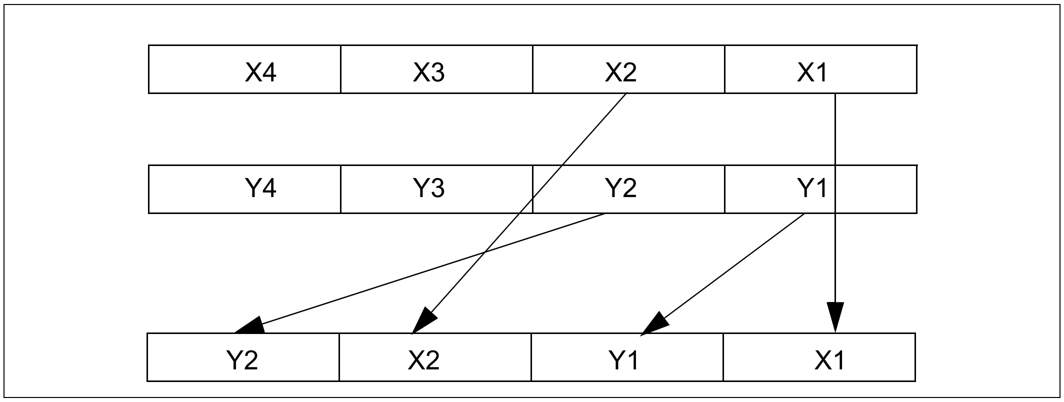
The UNPCKHPS (Unpacked high packed single-precision floating-point) instruction performs an interleaved unpack of the high-order data elements of first and second packed single-precision floating-point operands. It ignores the lower half part of the sources (Figure 4-6). When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the high order 64 bits are utilized by the instruction.

Figure 4-6. Unpack High Operation



The UNPCKLPS (Unpacked low packed single-precision floating-point) instruction performs an interleaved unpack of the low-order data elements of first and second packed single-precision floating-point operands. It ignores the higher half part of the sources (Figure 4-7). When unpacking from a memory operand, the full 128-bit operand is accessed from memory but only the low order 64 bits are utilized by the instruction.

Figure 4-7. Unpack Low Operation



4.6.1.5 Conversion Instructions

These instructions support packed and scalar conversions between 128-bit SSE registers and either 64-bit integer MMX technology registers or 32-bit integer IA-32 registers. The packed versions behave identically to original MMX technology instructions, in the presence of x87-FP instructions, including:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX technology instructions write ones (1's) to the exponent part of the corresponding x87-FP register.
- Use of EMMS for transition from MMX technology to x87-FP.

The CVTPI2PS (Convert packed 32-bit integer to packed single-precision floating-point) instruction converts two 32-bit signed integers in a MMX technology register to the two least significant single-precision floating-point numbers; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The upper two significant numbers in the destination register are retained.

The CVTSI2SS (Convert scalar 32-bit integer to scalar single-precision floating-point) instruction converts a 32-bit signed integer in a MMX technology register to the least significant single-precision floating-point number; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The upper three significant numbers in the destination register are retained.

The CVTPS2PI (Convert packed single-precision floating-point to packed 32-bit integer) instruction converts the two least significant single-precision floating-point numbers to two 32-bit signed integers in a MMX technology register; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned. The CVTTPS2PI (Convert truncate packed single-precision floating-point to packed 32-bit integer) instruction is similar to CVTPS2PI except if the conversion is inexact, in which case the truncated result is returned.

The CVTSS2SI (Convert scalar single-precision floating-point to a 32-bit integer) instruction converts the least significant single-precision floating-point number to a 32-bit signed integer in an Intel architecture 32-bit integer register; when the conversion is inexact, the rounded value according to the rounding mode in MXCSR is returned.The CVTTSS2SI (Convert truncate scalar single-precision floating-point to scalar 32-bit integer) instruction is similar to CVTSS2SI except if the conversion is inexact, the truncated result is returned.

4.6.1.6 Data Movement Instructions

The MOVAPS (Move aligned packed single-precision floating-point) instruction transfers 128-bits of packed data from memory to SSE registers and vice versa, or between SSE registers. The memory address is aligned to 16-byte boundary; if not then a general protection exception will occur.

The MOVUPS (Move unaligned packed single-precision floating-point) instruction transfers 128-bits of packed data from memory to SSE registers and vice versa, or between SSE registers. No assumption is made for alignment.

The MOVHPS (Move aligned high packed single-precision floating-point) instruction transfers 64-bits of packed data from memory to the upper two fields of a SSE register and vice versa. The lower field is left unchanged.

The MOVLPS (Move aligned low packed single-precision floating-point) instruction transfers 64-bits of packed data from memory to the lower two fields of a SSE register and vice versa. The upper field is left unchanged.

The MOVMSKPS (Move mask packed single-precision floating-point) instruction transfers the most significant bit of each of the four packed single-precision floating-point number to an IA integer register. This 4-bit value can then be used as a condition to perform branching.

The MOVSS (Move scalar single-precision floating-point) instruction transfers a single 32-bit floating-point number from memory to a SSE register or vice versa, and between registers.

4.6.1.7 State Management Instructions

The LDMXCSR (Load SSE Control and Status Register) instruction loads the SSE control and status register from memory. STMXCSR (Store SSE Control and Status Register) instruction stores the SSE control and status word to memory.

The FXSAVE instruction saves FP and MMX technology state and SSE state to memory. Unlike FSAVE, FXSAVE does not clear the x87-FP state. FXRSTOR loads FP and MMX technology state and SSE state from memory.

4.6.1.8 Additional SIMD Integer Instructions

Similar to the conversions instructions discussed in Section 4.6.1.5, "Conversion Instructions" on page 4:469, these SIMD Integer instructions also behave identically to original MMX technology instructions, in the presence of x87-FP instructions.

The PAVGB/PAVGW (Average unsigned source sub-operands, without incurring a loss in precision) instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register. The results of the add are then each independently right shifted right by one bit position. The high order bits of each element are filled with the carry bits of the sums. To prevent cumulative round-off errors, an averaging is performed. The low order bit of each final shifted result is set to 1 if at least one of the two least significant bits of the intermediate unshifted shifted sum is 1.

The PEXTRW (Extract 16-bit word from MMX technology register) instruction moves the word in a MMX technology register selected by the two least significant bits of the immediate operand to the lower half of a 32-bit integer register; the upper word in the integer register is cleared.

The PINSRW (Insert 16-bit word into MMX technology register) instruction moves the lower word in a 32-bit integer register or 16-bit word from memory into one of the four word locations in a MMX technology register, selected by the two least significant bits of the immediate operand.

The PMAXUB/PMAXSW (Maximum of packed unsigned integer bytes or signed integer words) instruction returns the maximum of each pair of packed elements into the destination register.

The PMINUB/PMINSW (Minimum of packed unsigned integer bytes or signed integer words) instructions returns the minimum of each pair of packed data elements into the destination register.

The PMOVMSKB (Move Byte Mask from MMX technology register) instruction returns an 8-bit mask formed of the most significant bits of each byte of its source operand in a MMX technology register to an IA integer register.

The PMULHUW (Unsigned high packed integer word multiply in MMX technology register) instruction performs an unsigned multiply on each word field of the two source MMX technology registers, returning the high word of each result to a MMX technology register.

The PSADBW (Sum of absolute differences) instruction computes the absolute difference for each pair of sub-operand byte sources and then accumulates the 8 differences into a single 16-bit result.

The PSHUFW (Shuffle packed integer word in MMX technology register) instruction performs a full shuffle of any source word field to any result word field, using an 8-bit immediate operand.

4.6.1.9 Cacheability Control Instructions

Data referenced by a programmer can have temporal (data will be used again) or spatial (data will be in adjacent locations, e.g. same cache line) locality. Some multimedia data types, such as the display list in a 3D graphics application, are referenced once and not reused in the immediate future. We will refer to this data type as non-temporal data. Thus the programmer does not want the application's cached code and data to be overwritten by this non-temporal data. The cacheability control instructions enable the programmer to control caching so that non-temporal accesses will minimize cache pollution.

In addition, the execution engine needs to be fed such that it does not become stalled waiting for data. SSE instructions allow the programmer to prefetch data long before it's final use. These instructions are not architectural since they do not update any architectural state, and are specific to each implementation. The programmer may have to tune his application for each implementation to take advantage of these instructions. These instructions merely provide a hint to the hardware, and they will not generate exceptions or faults. Excessive use of prefetch instructions may be throttled by the processor.

The following four instructions provide hints to the cache hierarchy which enables the data to be prefetched to different levels of the cache hierarchy and avoid polluting cache with non-temporal data.

The MASKMOVQ (Non-temporal byte mask store of packed integer in a MMX technology register) instruction stores data from a MMX technology register to the location specified by the EDI register. The most significant bit in each byte of the second MMX technology mask register is used to selectively write the data of the first register on a per-byte basis. The instruction is implicitly weakly-ordered, with all of the characteristics of the WC memory type; successive non-temporal stores may not write memory in program-order, do not write-allocate (i.e. the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store), write combine/collapse, and minimize cache pollution.

The MOVNTQ (Non-temporal store of packed integer in a MMX technology register) instruction stores data from a MMX technology register to memory. The instruction is implicitly weakly-ordered, does not write-allocate and minimizes cache pollution.

The MOVNTPS (Non-temporal store of packed single-precision floating-point) instruction stores data from a SSE register to memory. The memory address must be aligned to a 16-byte boundary; if it is not aligned, a general protection exception will occur. The instruction is implicitly weakly-ordered, does not write-allocate and minimizes cache pollution.

The main difference between a non-temporal store and a regular cacheable store is in the write-allocation policy. The memory type of the region being written to can override the non-temporal hint, leading to the following considerations:

- If the programmer specifies a non-temporal store to uncacheable memory, then the store behaves like an uncacheable store; the non-temporal hint is ignored and the memory type for the region is retained. Uncacheable as referred to here means that the region being written to has been mapped with either a UC or WP memory type. If the memory region has been mapped as WB, WT or WC, the non-temporal store will implement weakly-ordered (WC) semantic behavior.
- If the programmer specifies a non-temporal store to cacheable memory, two cases may result:
    - If the data is present in the cache hierarchy, the instruction will ensure consistency. A given processor may choose different ways to implement this; some examples include: updating data in-place in the cache hierarchy while preserving the memory type semantics assigned to that region, or evicting the data from the caches and writing the new non-temporal data to memory (with WC semantics).
    - If the data is not present in the cache hierarchy, and the destination region is mapped as WB, WT or WC, the transaction will be weakly ordered, and is subject to all WC memory semantics. The non-temporal store will not write allocate. Different implementations may choose to collapse and combine these stores.
- In general, WC semantics require software to ensure coherence, with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and a fencing operation (see SFENCE, below) must be performed for producer-consumer usage models. Fencing ensures that all system agents have global visibility of the stored data; for instance, failure to fence may result in a written cache line staying within a processor, and the line would not be visible to other agents. For processors which implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. Otherwise if mapped as WB or WT, there is the potential for speculative processor reads to bring the data into the caches; in this case, non-temporal stores would then update in place, and data would not be flushed from the processor by a subsequent fencing operation.
- The memory type visible on the bus in the presence of memory type aliasing is implementation specific. As one possible example, the memory type written to the bus may reflect the memory type for the first store to this line, as seen in program order; other alternatives are possible. This behavior should be considered reserved, and dependency on the behavior of any particular implementation risks future incompatibility.

The PREFETCH (Load 32 or greater number of bytes) instructions load either non-temporal data or temporal data in the specified cache level. This access and the cache level are specified as a hint. The prefetch instructions do not affect functional behavior of the program and will be implementation specific.

The SFENCE (Store Fence) instruction guarantees that every store instruction that precedes the store fence instruction in program order is globally visible before any store instruction which follows the fence. The SFENCE instruction provides an efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

4.7 IEEE Compliance

SSE floating-point computation is IEEE-754 compliant except when the control word is set to flush to zero mode. IEEE-754 compliance includes support for single-precision signed infinities, QNaNs, SNaNs, integer indefinite, signed zeros, denormals, masked and unmasked exceptions. single-precision floating-point values are represented identically both internally and in memory, and are of the following form:

Sign	Exponent	Significand
31	30...23	22...0

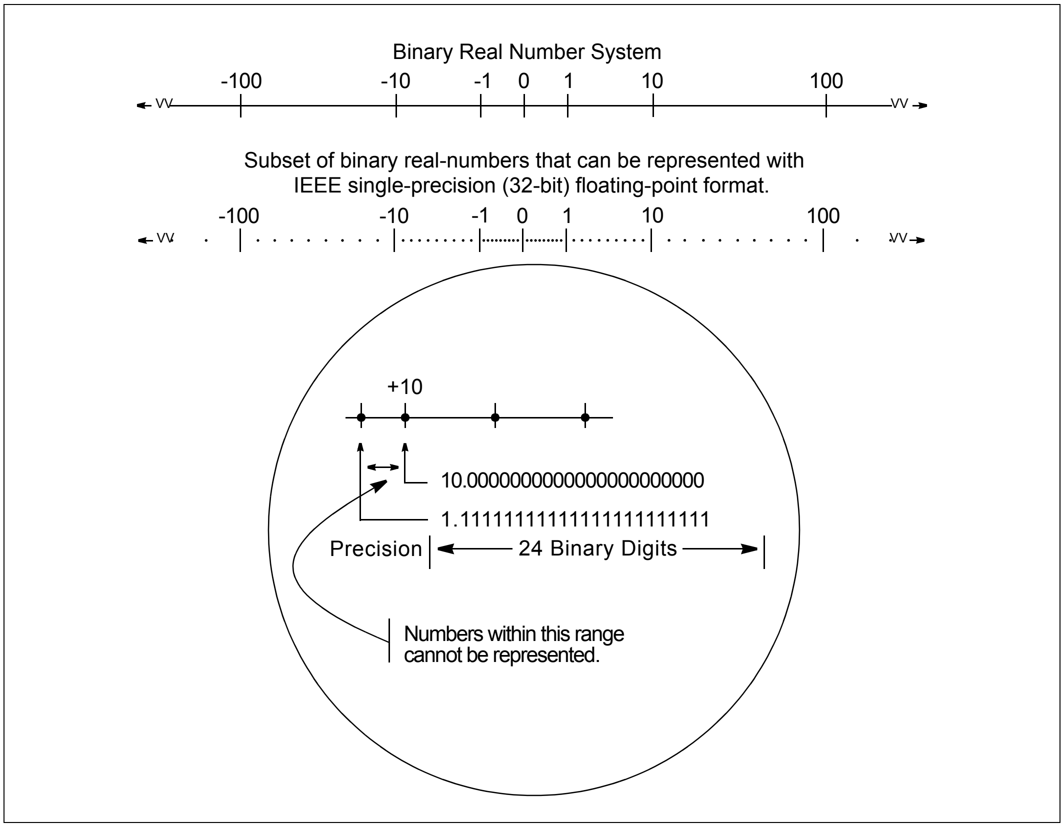
This is a change from x87 floating-point which internally represents all numbers in 80-bit extended format. This change implies that x87-FP libraries re-written to use SSE instructions may not produce results that are identical to the those of the x87-FP implementation.Real Numbers and Floating-point Formats.

This section describes how real numbers are represented in floating-point format in the processor. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE standards may wish to skip this section.

4.7.1 Real Number System

As shown in Figure 4-8, the real-number system comprises the continuum of real numbers from minus infinity (–∞) to plus infinity (+∞).

Figure 4-8. Binary Real Number System


Binary Real Number System
-100
-10
-1
0
1
10
100
Subset of binary real-numbers that can be represented with
IEEE single-precision (32-bit) floating-point format.
-100
-10
-1
0
1
10
100
+10
10.0000000000000000000000
1.11111111111111111111111
Precision
24 Binary Digits
Numbers within this range
cannot be represented.

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 4-1, the subset of real numbers that a particular processor supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the processor uses to represent real numbers.

4.7.1.1 Floating-point Format

To increase the speed and efficiency of real-number computations, computers typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 4-9 shows the binary floating-point format that SSE data uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The J-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.

Figure 4-9. Binary Floating-point Format

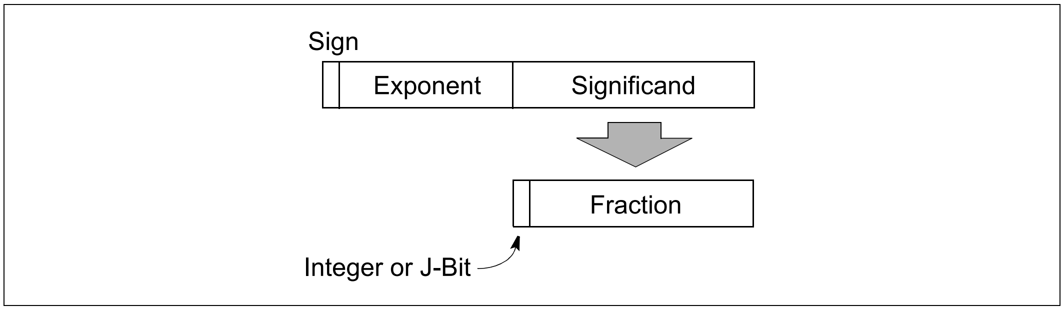


Table 4-1 shows how the real number 178.125 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the format that the processor uses. In this format, the binary real number is normalized and the exponent is biased.

Table 4-1. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
Single Format (Normalized)	Sign	Biased Exponent	Significand
	0	10000110	01100100010000000000000 1 (Implied)

4.7.1.2 Normalized Numbers

In most cases, the processor represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

4.7.1.3 Biased Exponent

The processor represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

4.7.1.4 Real Number and Non-Number Encodings

A variety of real numbers and special values can be encoded in the processor's floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros
- Denormalized finite numbers
- Normalized finite numbers
- Signed infinities
- NaNs
- Indefinite numbers

(The term NaN stands for "Not a Number.")

Figure 4-10 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term "S" indicates the sign bit, "E" the biased exponent, and "F" the fraction. (The exponent values are given in decimal.)

The processor can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

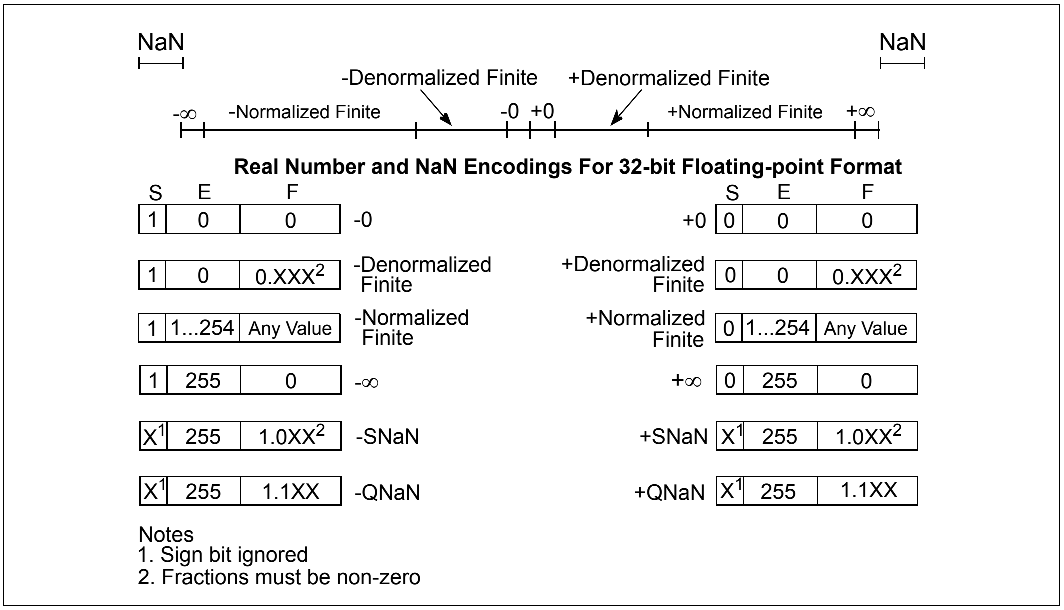
4.7.1.5 Signed Zeros

Zero can be represented as a +0 or a –0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

4.7.1.6 Normalized and Denormalized Finite Numbers

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the format shown in Figure 4-10, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

Figure 4-10. Real Numbers and NaNs



When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called *denormalized* (or *tiny*) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, a processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an *underflow* condition.

A denormalized number is computed through a technique called gradual underflow. Table 4-2 gives an example of gradual underflow in the denormalization process. Here the single-real format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 4-2. Denormalization Process

Operation	Sign	Exponent[a]	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00

Table 4-2. Denormalization Process

Operation	Sign	Exponent[a]	Significand
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

a. Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The processor deals with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

4.7.1.7 Signed Infinities

The two infinities, +∞ and –∞, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero significand (fraction and integer bit) and the maximum biased exponent allowed in the specified format (for example, 255_{10} for the single-real format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, -∞ is less than any finite number and +∞ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

4.7.1.8 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 4-10, the encoding space for NaNs in the processor floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two classes of NaN: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions, as well as detailed information on how the processor handles NaNs, are discussed in Section 4.7.2, “Operating on NaNs”.

4.7.1.9 Indefinite

In response to a masked invalid-operation floating-point exceptions, the indefinite value QNAN is produced. The integer indefinite, which can be produced during conversion from single-precision floating-point to 32-bit integer, is defined to be 80000000H.

4.7.2 Operating on NaNs

As was described in Section 4.7.1.8, “NaNs” on page 4:479, the Intel SSE architecture supports two types of NaNs: SNaNs and QNaNs. An SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an ∞.) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

As a general rule, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating-point invalid-operation exception to be signaled. SNaNs are typically used to trap or invoke an exception handler.

The invalid operation exception has a flag and a mask bit associated with it in MXCSR. The mask bit determines how the an SNaN value is handled. If the invalid operation mask bit is set, the SNaN is converted to a QNaN by setting the most-significant fraction bit of the value to 1. The result is then stored in the destination operand and the invalid operation flag is set. If the invalid operation mask is clear, an invalid operation fault is signaled and no result is stored in the destination operand.

When a real operation or exception delivers a QNaN result, the value of the result depends on the source operands, as shown in Table 4-3. The exceptions to the behavior described in Table 4-3 are the MINPS and MAXPS instructions. If only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 4-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPS/MAXPS allows NaN data to be screened out of the bounds-checking portion of an algorithm. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

In general Src1 and Src2 relate to an SSE instruction as follows:

```
ADDPS Src1, Src2/m128
```

Except for the rules given at the beginning of this section for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.

Table 4-3. Results of Operations with NAN Operands

Source Operands	NaN Result (invalid operation exception is masked)
An SNaN and a QNaN.	Src1 NaN (converted to QNaN if Src1 is an SNaN).
Two SNaNs.	Src1 NaN (converted to QNaN)
Two QNaNs.	Src1 QNaN
An SNaN and a real value.	The SNaN converted into a QNaN.
A QNaN and a real value.	The QNaN source operand.
An SNaN/QNaN value (for instructions which take only one operand i.e. RCPPS, RCPSS, RSQRTPS, RSQRTSS)	The SNaN converted into a QNaN/the source QNaN.
Neither source operand is a NaN and a floating-point invalid-operation exception is signaled.	The default QNaN *real indefinite*.

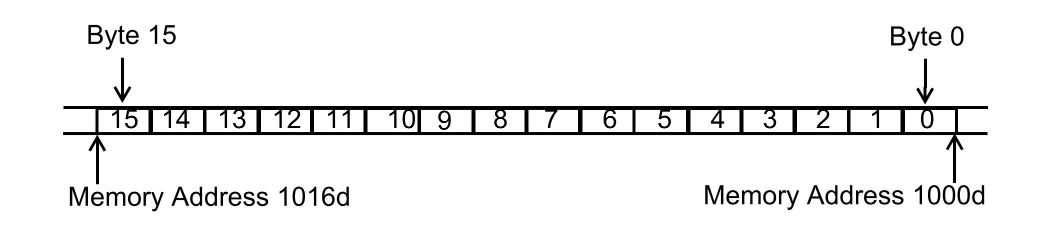
4.8 Data Formats

4.8.1 Memory Data Formats

The Intel SSE architecture introduces a new packed 128-bit data type which consists of 4 single-precision floating-point numbers. The 128 bits are numbered 0 through 127. Bit 0 is the least significant bit (LSB), and bit 127 is the most significant bit (MSB).

Bytes in the new data type format have consecutive memory addresses. The ordering is always little endian, that is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

Figure 4-11. Four Packed FP Data in Memory (at address 1000H)



4.8.2 SSE Register Data Formats

Values in SSE registers have the same format as a 128-bit quantity in memory. They have two data access modes: 128-bit access mode and 32-bit access mode. The data type corresponds directly to the single-precision format in the IEEE standard. Table 4-4 gives the precision and range of this data type. Only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. The exponent of the single-precision data type is encoded in biased format. The biasing constant is 127 for the single-precision format.

Table 4-4. Precision and Range of SSE Datatype

Data Type	Length	Precision (Bits)	Approximate Normalized Range	
			Binary	Decimal
Single-precision	32	24	2^{-126} to 2^{127}	1.18×10^{-38} to 3.40×10^{38}

Table 4-5 shows the encodings for all the classes of real numbers (that is, zero, denormalized-finite, normalized-finite, and ∞) and NaNs for the single-real data-type. It also gives the format for the real indefinite value, which is a QNaN encoding that is generated by several SSE instructions in response to a masked floating-point invalid-operation exception.

Table 4-5. Real Number and NaN Encodings

Class		Sign	Biased Exponent	Significand	
				Integer[1]	Fraction
Positive	+∞	0	11..11	1	00..00
	+Normals	0 . . 0	11..10 . . 00..01	1 . . 1	11..11 . . 00..00
	+Denormals	0 . . 0	00..00 . . 00..00	0 . . 0	11.11 . . 00..01
	+Zero	0	00..00	0	00..00
Negative	−Zero	1	00..00	0	00..00
	−Denormals	1 . . 1	00..00 . . 00..00	0 . . 0	00..01 . . 11..11
	−Normals	1 . . 1	00..01 . . 11..10	1 . . 1	00..00 . . 11..11
	-∞	1	11..11	1	00..00
NaNs	SNaN	X	11..11	1	0X..XX[2]
	QNaN	X	11..11	1	1X..XX
	Real Indefinite (QNaN)	1	11..11	1	10..00
	Single		⟵ 8 Bits ⟶		⟵ 23 Bits ⟶

When storing real values in memory, single-real values are stored in 4 consecutive bytes in memory. The 128-bit access mode is used for 128-bit memory accesses, 128-bit transfers between SSE registers, and all logical, unpack and arithmetic instructions.The 32-bit access mode is used for 32-bit memory access, 32-bit transfers between SSE registers, and all arithmetic instructions.

There are sixty-eight new instructions in SSE instruction set. This chapter describes the packed and scalar floating-point instructions in alphabetical order, with a full description of each instruction. The last two sections of this chapter describe the SIMD Integer instructions and the cacheability control instructions.

4.9 Instruction Formats

The nature of the Intel SSE architecture allows the use of existing instruction formats. Instructions use the ModR/M format and are preceded by the 0F prefix byte. In general, operations are not duplicated to provide two directions (i.e. separate load and store variants).

4.10 Instruction Prefixes

The SSE instructions use prefixes as specified in Table 4-6, Table 4-7, and Table 4-8. The effect of multiple prefixes (more than one prefix from a group) is unpredictable and may vary from processor to processor.

Applying a prefix, in a manner not defined in this document, is considered reserved behavior. For example, Table 4-6 shows general behavior for most SSE instructions; however, the application of a prefix (Repeat, Repeat NE, Operand Size) is reserved for the following instructions:

ANDPS, ANDNPS, COMISS, FXRSTOR, FXSAVE, ORPS, LDMXCSR, MOVAPS, MOVHPS, MOVLPS, MOVMSKPS, MOVNTPS, MOVUPS, SHUFPS, STMXCSR, UCOMISS, UNPCKHPS, UNPCKLPS, XORPS.

Table 4-6. SSE Instruction Behavior with Prefixes

Prefix Type	Effect on SSE Instructions
Address Size Prefix (67H)	Affects SSE instructions with memory operand Ignored by SSE instructions without memory operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects SSE instructions with mem.operand Ignored by SSE instructions without mem operand
Repeat Prefix (F3H)	Affects SSE instructions
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates invalid opcode exception.

Table 4-7. SIMD Integer Instructions – Behavior with Prefixes

Prefix Type	Effect on Intel® MMX™ Technology Instructions
Address Size Prefix (67H)	Affects Intel MMX technology instructions with mem. operand Ignored by Intel MMX technology instructions without mem. operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects Intel MMX technology instructions with mem. operand Ignored by Intel MMX technology instructions without mem operand
Repeat Prefix (F3H)	Reserved and may result in unpredictable behavior.
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates invalid opcode exception.

Table 4-8. Cacheability Control Instruction Behavior with Prefixes

Prefix Type	Effect on SSE Instructions
Address Size Prefix (67H)	Affects cacheability control instruction with a mem. operand Ignored by cacheability control instruction w/o a mem. operand.
Operand Size (66H)	Reserved and may result in unpredictable behavior.

Table 4-8. Cacheability Control Instruction Behavior with Prefixes

Prefix Type	Effect on SSE Instructions
Segment Override (2EH,36H,3EH,26H,64H,65H)	Affects cacheability control instructions with mem. operand Ignored by cacheability control instruction without mem operand
Repeat Prefix(F3H)	Reserved and may result in unpredictable behavior.
Repeat NE Prefix(F2H)	Reserved and may result in unpredictable behavior.
Lock Prefix (0F0H)	Generates an invalid opcode exception for all cacheability instructions.

4.11 Reserved Behavior and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as *reserved*. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only reserved, but unpredictable. In general, reserved behavior may also be applied in other areas. Software should follow these guidelines in dealing with reserved behavior:

- Do not depend on the states of any reserved fields when testing the values of registers which contain such bits. Mask out the reserved fields before testing.
- Do not depend on the states of any reserved fields when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved fields.
- When loading a register, always load the reserved fields with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

Note: Avoid any software dependency upon the reserved state/behavior. Depending upon reserved behavior will make the software dependent upon the unspecified manner in which the processor handles this behavior and risks incompatibility with future processors.

4.12 Notations

Besides opcodes, two kinds of notations are found which both describe information found in the ModR/M byte:

1. **/digit:** (digit between 0 and 7) indicates that the instruction uses only the r/m (register and memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
2. **/r**: indicates that the ModR/M byte of an instruction contains both a register operand and an r/m operand.

In addition, the following abbreviations are used:

- **r32**: Intel architecture 32-bit integer register.
- **xmm/m128**:Indicates a 128-bit multimedia register or a 128-bit memory location.
- **xmm/m64**: Indicates a 128-bit multimedia register or a 64-bit memory location.
- **xmm/m32:** Indicates a 128-bit multimedia register or a 32-bit memory location.
- **mm/m64**: Indicates a 64-bit multimedia register or a 64-bit memory location.

- **imm8**: Indicates an immediate 8-bit operand.
- **ib**: Indicates that an immediate byte operand follows the opcode, ModR/M byte or
  scaled-indexing byte.

When there is ambiguity, xmm1 indicates the first source operand and xmm2 the second source operand.

Table 4-9 describes the naming conventions used in the SSE instruction mnemonics.

Table 4-9. Key to SSE Naming Convention

Mnemonic	Description
PI	Packed integer qword (e.g. mm0)
PS	Packed single FP (e.g. xmm0)
SI	Scalar integer (e.g. eax)
SS	Scalar single-FP (e.g. low 32 bits of xmm0)

ADDPS: Packed Single-FP Add

Opcode	Instruction	Description
0F,58,/r	ADDPS xmm1, xmm2/m128	Add packed SP FP numbers from XMM2/Mem to XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0]   + xmm2/m128[31-0];

xmm1[63-32]  = xmm1[63-32]  + xmm2/m128[63-32];

xmm1[95-64]  = xmm1[95-64]  + xmm2/m128[95-64];

xmm1[127-96] = xmm1[127-96] + xmm2/m128[127-96];
```

Description: The ADDPS instruction adds the packed SP FP numbers of both their operands.

Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0)

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ADDSS: Scalar Single-FP Add

Opcode	Instruction	Description
F3,0F,58, /r	ADDSS xmm1, xmm2/m32	Add the lower SP FP number from XMM2/Mem to XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0] + xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The ADDSS instruction adds the lower SP FP numbers of both their operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ANDNPS: Bit-wise Logical And Not for Single-FP

Opcode	Instruction	Description
0F,55,/r	ANDNPS xmm1, xmm2/m128	Invert the 128 bits in XMM1and then AND the result with 128 bits from XMM2/Mem.

Operation: `xmm1[127-0] = ~(xmm1[127-0]) & xmm2/m128[127-0];`

Description: The ANDNPS instructions returns a bit-wise logical AND between the complement of XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ANDNPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ANDNPS risks incompatibility with future processors.

ANDPS: Bit-wise Logical And for Single-FP

Opcode	Instruction	Description
0F,54,/r	ANDPS xmm1, xmm2/m128	Logical AND of 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm1[127-0] &= xmm2/m128[127-0];`

Description: The ANDPS instruction returns a bit-wise logical AND between XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ANDPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ANDPS risks incompatibility with future processors.

CMPPS: Packed Single-FP Compare

Opcode	Instruction	Description
0F,C2,/r,ib	CMPPS xmm1, xmm2/m128, imm8	Compare packed SP FP numbers from XMM2/Mem to packed SP FP numbers in XMM1 register using imm8 as predicate.

Operation:

```
switch (imm8) {
    case eq:       op = eq;
    case lt:       op = lt;
    case le:       op = le;
    case unord:    op = unord;
    case neq:      op = neq;
    case nlt:      op = nlt;
    case nle:      op = nle;
    case ord:      op = ord;
    default:       Reserved;
    }

cmp0 = op(xmm1[31-0],xmm2/m128[31-0]);
cmp1 = op(xmm1[63-32],xmm2/m128[63-32]);
cmp2 = op(xmm1[95-64],xmm2/m128[95-64]);
cmp3 = op(xmm1[127-96],xmm2/m128[127-96]);

xmm1[31-0]   = (cmp0) ? 0xffffffff : 0x00000000;
xmm1[63-32]   = (cmp1) ? 0xffffffff : 0x00000000;
xmm1[95-64]   = (cmp2) ? 0xffffffff : 0x00000000;
xmm1[127-96]   = (cmp3) ? 0xffffffff : 0x00000000;
```

Description: For each individual pairs of SP FP numbers, the CMPPS instruction returns an all “1” 32-bit mask or an all “0” 32-bit mask, using the comparison predicate specified by imm8; note that a subsequent computational instruction which uses this mask as an input operand will not generate a fault, since a mask of all “0’s” corresponds to a FP value of +0.0 and a mask of all “1’s” corresponds to a FP value of -qNaN. Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading “Emulation.” The following table shows the different comparison types:

CMPPS: Packed Single-FP Compare (Continued)

Predicate	Description[a]	Relation	Emulation	imm8 Encoding	Result if NaN Operand	QNaN Operand Signals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

[a] The greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid if sNaN operand, invalid if qNaN and predicate as listed in above table, denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

CMPPS: Packed Single-FP Compare (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Compilers and assemblers should implement the following 2-operand pseudo-ops in addition to the 3-operand CMPPS instruction:

Pseudo-Op	Implementation
CMPEQPS xmm1, xmm2	CMPPS xmm1,xmm2, 0
CMPLTPS xmm1, xmm2	CMPPS xmm1,xmm2, 1
CMPLEPS xmm1, xmm2	CMPPS xmm1,xmm2, 2
CMPUNORDPS xmm1, xmm2	CMPPS xmm1,xmm2, 3
CMPNEQPS xmm1, xmm2	CMPPS xmm1,xmm2, 4
CMPNLTPS xmm1, xmm2	CMPPS xmm1,xmm2, 5
CMPNLEPS xmm1, xmm2	CMPPS xmm1,xmm2, 6
CMPORDPS xmm1, xmm2	CMPPS xmm1,xmm2, 7

The greater-than relations not implemented in hardware require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Bits 7-4 of the immediate field are reserved. Different processors may handle them differently. Usage of these bits risks incompatibility with future processors.

CMPSS: Scalar Single-FP Compare

Opcode	Instruction	Description
F3,0F,C2,/r,ib	CMPSS xmm1, xmm2/m32, imm8	Compare lowest SP FP number from XMM2/Mem to lowest SP FP number in XMM1 register using imm8 as predicate.

Operation:

```
switch (imm8) {
   case eq:      op = eq;
   case lt:      op = lt;
   case le:      op = le;
   case unord:   op = unord;
   case neq:     op = neq;
   case nlt:     op = nlt;
   case nle:     op = nle;
   case ord:     op = ord;
   default:      Reserved;
   }

cmp0 = op(xmm1[31-0],xmm2/m32[31-0]);

xmm1[31-0]   = (cmp0) ? 0xffffffff : 0x00000000;
xmm1[63-32]   = xmm1[63-32];
xmm1[95-64]   = xmm1[95-64];
xmm1[127-96]   = xmm1[127-96];
```

Description: For the lowest pair of SP FP numbers, the CMPSS instruction returns an all "1" 32-bit mask or an all "0" 32-bit mask, using the comparison predicate specified by imm8; the values for the upper three pairs of SP FP numbers are not compared. Note that a subsequent computational instruction which uses this mask as an input operand will not generate a fault, since a mask of all "0's" corresponds to a FP value of +0.0 and a mask of all "1's" corresponds to a FP value of -qNaN. Some of the comparisons can be achieved only through software emulation. For these comparisons the programmer must swap the operands, copying registers when necessary to protect the data that will now be in the destination, and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in under the heading "Emulation." The following table shows the different comparison types:

CMPSS: Scalar Single-FP Compare (Continued)

Predicate	Description[a]	Relation	Emulation	imm8 Encoding	Result if NaN Operand	qNaN OperandSignals Invalid
eq	equal	xmm1 == xmm2		000B	False	No
lt	less-than	xmm1 < xmm2		001B	False	Yes
le	less-than-or-equal	xmm1 <= xmm2		010B	False	Yes
	greater than	xmm1 > xmm2	swap, protect, lt		False	Yes
	greater-than-or-equal	xmm1 >= xmm2	swap protect, le		False	Yes
unord	unordered	xmm1 ? xmm2		011B	True	No
neq	not-equal	!(xmm1 == xmm2)		100B	True	No
nlt	not-less-than	!(xmm1 < xmm2)		101B	True	Yes
nle	not-less-than-or-equal	!(xmm1 <= xmm2)		110B	True	Yes
	not-greater-than	!(xmm1 > xmm2)	swap, protect, nlt		True	Yes
	not-greater-than-or-equal	!(xmm1 >= xmm2)	swap, protect, nle		True	Yes
ord	ordered	!(xmm1 ? xmm2)		111B	False	No

[a] The greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations are not directly implemented in hardware.

FP Exceptions: None.

Numeric Exceptions: Invalid if sNaN operand, invalid if qNaN and predicate as listed in above table, denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

CMPSS: Scalar Single-FP Compare (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Compilers and assemblers should implement the following 2-operand pseudo-ops in addition to the 3-operand CMPSS instruction:

Pseudo-Op	Implementation
CMPEQSS xmm1, xmm2	CMPSS xmm1,xmm2, 0
CMPLTSS xmm1, xmm2	CMPSS xmm1,xmm2, 1
CMPLESS xmm1, xmm2	CMPSS xmm1,xmm2, 2
CMPUNORDSS xmm1, xmm2	CMPSS xmm1,xmm2, 3
CMPNEQSS xmm1, xmm2	CMPSS xmm1,xmm2, 4
CMPNLTSS xmm1, xmm2	CMPSS xmm1,xmm2, 5
CMPNLESS xmm1, xmm2	CMPSS xmm1,xmm2, 6
CMPORDSS xmm1, xmm2	CMPSS xmm1,xmm2, 7

The greater-than relations not implemented in hardware require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Bits 7-4 of the immediate field are reserved. Different processors may handle them differently. Usage of these bits risks incompatibility with future processors.

COMISS: Scalar Ordered Single-FP Compare and set EFLAGS

Opcode	Instruction	Description
0F,2F,/r	COMISS xmm1, xmm2/m32	Compare lower SP FP number in XMM1 register with lower SP FP number in XMM2/Mem and set the status flags accordingly

Operation:

```
switch (xmm1[31-0] <> xmm2/m32[31-0]) {

   OF,SF,AF = 000;

   case UNORDERED:     ZF,PF,CF = 111;

   case GREATER_THAN:  ZF,PF,CF = 000;

   case LESS_THAN:     ZF,PF,CF = 001;

   case EQUAL:         ZF,PF,CF = 100;

}
```

Description: The COMISS instructions compare two SP FP numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above. Although the data type is packed single-FP, only the lower SP numbers are compared. In addition, the OF, SF and AF bits in the EFLAGS register are zeroed out. The unordered predicate is returned if either source operand is a NaN (qNaN or sNaN).

FP Exceptions: None.

Numeric Exceptions: Invalid (if SNaN or QNaN operands), Denormal. Integer EFLAGS values will not be updated in the presence of unmasked numeric exceptions.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

COMISS: Scalar Ordered Single-FP Compare and set EFLAGS (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: COMISS differs from UCOMISS in that it signals an invalid numeric exception when a source operand is either a qNaN or sNaN; UCOMISS signals invalid only if a source operand is an sNaN.

The usage of Repeat (F2H, F3H) and Operand-Size (66H) prefixes with COMISS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with COMISS risks incompatibility with future processors.

CVTPI2PS: Packed Signed INT32 to Packed Single-FP Conversion

Opcode	Instruction	Description
0F,2A,/r	CVTPI2PS xmm, mm/m64	Convert two 32-bit signed integers from MM/Mem to two SP FP.

Operation:

```
xmm[31-0]    = (float) (mm/m64[31-0]);

xmm[63-32]   = (float) (mm/m64[63-32]);

xmm[95-64]   = xmm[95-64];

xmm[127-96] = xmm[127-96];
```

Description: The CVTPI2PS instruction converts signed 32-bit integers to SP FP numbers; when the conversion is inexact, rounding is done according to MXCSR.

FP Exceptions: None.

Numeric Exceptions: Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTPI2PS: Packed Signed INT32 to Packed Single-FP Conversion (Continued)

Comments: This instruction behaves identically to original MMX technology instructions, in the presence of x87-FP instructions:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX technology instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

However, the use of a memory source operand with this instruction will not result in the above transition from x87-FP to MMX technology.

Prioritization for fault and assist behavior for CVTPI2PS is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #SS or #GP, for limit violation
4. #PF, page fault
5. SSE numeric fault (i.e. precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. SSE numeric fault (i.e. precision)

CVTPS2PI: Packed Single-FP to Packed INT32 Conversion

Opcode	Instruction	Description
0F,2D,/r	CVTPS2PI mm, xmm/m64	Convert lower 2 SP FP from XMM/Mem to 2 32-bit signed integers in MM using rounding specified by MXCSR.

Operation:

```
mm[31-0]   = (int) (xmm/m64[31-0]);

mm[63-32]  = (int) (xmm/m64[63-32]);
```

Description: The CVTPS2PI instruction converts the lower 2 SP FP numbers in xmm/m64 to signed 32-bit integers in mm; when the conversion is inexact, the value rounded according to the MXCSR is returned. If the converted result(s) is/are larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: This instruction behaves identically to original MMX technology instructions, in the presence of x87-FP instructions, including:

CVTPS2PI: Packed Single-FP to Packed INT32 Conversion (Continued)

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX technology instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

Prioritization for fault and assist behavior for CVTPS2PI is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. #SS or #GP, for limit violation
6. #PF, page fault
7. SSE numeric fault (i.e. invalid, precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. SSE numeric fault (i.e. precision)

CVTSI2SS: Scalar signed INT32 to Single-FP Conversion

Opcode	Instruction	Description
F3,0F,2A,/r	CVTSI2SS xmm, r/m32	Convert one 32-bit signed integer from Integer Reg/Mem to one SP FP.

Operation:

```
xmm[31-0]   = (float) (r/m32);

xmm[63-32]  = xmm[63-32];

xmm[95-64]  = xmm[95-64];

xmm[127-96] = xmm[127-96];
```

Description: The CVTSI2SS instruction converts a signed 32-bit integer from memory or from a 32-bit integer register to a SP FP number; when the conversion is inexact, rounding is done according to the MXCSR.

FP Exceptions: None.

Numeric Exceptions: Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTSS2SI: Scalar Single-FP to Signed INT32 Conversion

Opcode	Instruction	Description
F3,0F,2D,/r	CVTSS2SI r32, xmm/m32	Convert one SP FP from XMM/Mem to one 32 bit signed integer using rounding mode specified by MXCSR, and move the result to an integer register.

Operation: `r32 = (int) (xmm/m32[31-0]);`

Description: The CVTSS2SI instruction converts a SP FP number to a signed 32-bit integer and returns it in the 32-bit integer register; when the conversion is inexact, the rounded value according to the MXCSR is returned. If the converted result is larger than the maximum signed 32 bit integer, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT = 0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTTPS2PI: Packed Single-FP to Packed INT32 Conversion (truncate)

Opcode	Instruction	Description
0F,2C,/r	CVTTPS2PI mm, xmm/m64	Convert lower 2 SP FP from XMM/Mem to 2 32-bit signed integers in MM using truncate.

Operation:

```
mm[31-0]   = (int) (xmm/m64[31-0]);

mm[63-32]  = (int) (xmm/m64[63-32]);
```

Description: The CVTTPS2PI instruction converts the lower 2 SP FP numbers in xmm/m64 to 2 32-bit signed integers in mm; if the conversion is inexact, the truncated result is returned. If the converted result(s) is/are larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

CVTTPS2PI: Packed Single-FP to Packed INT32 Conversion (truncate) (Continued)

Comments: This instruction behaves identically to original MMX technology instructions, in the presence of x87-FP instructions, including:

- Transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).
- MMX technology instructions write ones (1's) to the exponent part of the corresponding x87-FP register.

Prioritization for fault and assist behavior for CVTTPS2PI is as follows:

Memory source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. #SS or #GP, for limit violation
6. #PF, page fault
7. SSE numeric fault (i.e. invalid, precision)

Register source

1. Invalid opcode (CR0.EM=1)
2. DNA (CR0.TS=1)
3. #MF, pending x87-FP fault signalled
4. After returning from #MF, x87-FP->MMX technology transition
5. SSE numeric fault (i.e. precision)

CVTTSS2SI: Scalar Single-FP to signed INT32 Conversion (truncate)

Opcode	Instruction	Description
F3,0F,2C,/r	CVTTSS2SI r32, xmm/m32	Convert lowest SP FP from XMM/Mem to one 32 bit signed integer using truncate, and move the result to an integer register.

Operation: `r32 = (int) (xmm/m32[31-0]);`

Description: The CVTTSS2SI instruction converts a SP FP number to a signed 32-bit integer and returns it in the 32-bit integer register; if the conversion is inexact, the truncated result is returned. If the converted result is larger than the maximum signed 32 bit value, the Integer Indefinite value (0x80000000) will be returned.

FP Exceptions: None.

Numeric Exceptions: Invalid, Precision.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

DIVPS: Packed Single-FP Divide

Opcode	Instruction	Description
0F,5E,/r	DIVPS xmm1, xmm2/m128	Divide packed SP FP numbers in XMM1 by XMM2/Mem

Operation:

```
xmm1[31-0]   = xmm1[31-0]   / (xmm2/m128[31-0]);

xmm1[63-32]  = xmm1[63-32]  / (xmm2/m128[63-32]);

xmm1[95-64]  = xmm1[95-64]  / (xmm2/m128[95-64]);

xmm1[127-96] = xmm1[127-96] / (xmm2/m128[127-96]);
```

Description: The DIVPS instruction divides the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Divide by Zero, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

DIVSS: Scalar Single-FP Divide

Opcode	Instruction	Description
F3,0F,5E,/r	DIVSS xmm1, xmm2/m32	Divide lower SP FP numbers in XMM1 by XMM2/Mem

Operation:

```
xmm1[31-0]   = xmm1[31-0] / (xmm2/m32[31-0]);

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The DIVSS instructions divide the lowest SP FP numbers of both operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Divide by Zero, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

FXRSTOR: Restore FP and Intel® MMX™ Technology State and SSE State

Opcode	Instruction	Description
0F,AE,/1	FXRSTOR m512byte	Load FP/Intel MMX technology and SSE state from m512byte.

Operation:

```
FP and MMX technology state and SSE state = m512byte;
```

Description: The FXRSTOR instruction reloads the FP and MMX technology state and SSE state (environment and registers) from the memory area defined by m512byte. This data should have been written by a previous FXSAVE.

The FP and MMX technology and SSE environment and registers consist of the following data structure (little-endian byte order as arranged in memory, with byte offset into row described by right column):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsrvd		CS		IP				FOP		FTW		FSW		FCW		0
Reserved				MXCSR				Rsrvd		DS		DP				16
Reserved						ST0/MM0										32
Reserved						ST1/MM1										48
Reserved						ST2/MM2										64
Reserved						ST3/MM3										80
Reserved						ST4/MM4										96
Reserved						ST5/MM5										112
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
XMM0																160
XMM1																176
XMM2																192
XMM3																208
XMM4																224
XMM5																240
XMM6																256
XMM7																272
Reserved																288
Reserved																304
Reserved																320
Reserved																336
Reserved																352
Reserved																368
Reserved																384
Reserved																400
Reserved																416
Reserved																432
Reserved																448

15 14	13 12	11 10 9 8	7 6	5 4	3 2	1 0	
Rsrvd	CS	IP	FOP	FTW	FSW	FCW	0
Reserved							464
Reserved							480
Reserved							496

Three fields in the floating-point save area contain reserved bits that are not indicated in the table:

- FOP: The lower 11-bits contain the opcode, upper 5-bits are reserved.
- IP & DP:32-bit mode: 32-bit IP-offset.
- 16-bit mode: lower 16-bits are IP-offset and upper 16-bits are reserved.

If the MXCSR state contains an unmasked exception with corresponding status flag also set, loading it will not result in a floating-point error condition being asserted; only the next occurrence of this unmasked exception will result in the error condition being asserted.

Some bits of MXCSR (bits 31-16 and bit 6) are defined as reserved and cleared; attempting to write a non-zero value to these bits will result in a general protection exception.

FXRSTOR does not flush pending x87-FP exceptions, unlike FRSTOR. To check and raise exceptions when loading a new operating environment, use FWAIT after FXRSTOR.

The SSE fields in the save image (XMM0-XMM7 and MXCSR) may not be loaded into the processor if the CR4.OSFXSR bit is not set. This CR4 bit must be set in order to enable execution of SSE instructions.

FP Exceptions: If #AC exception detection is disabled, a general protection exception is signalled if the address is not aligned on 16-byte boundary. Note that if #AC is enabled (and CPL is 3), signalling of #AC is not guaranteed and may vary with implementation; in all implementations where #AC is not signalled, a general protection fault will instead be signalled. In addition, the width of the alignment check when #AC is enabled may also vary with implementation; for instance, for a given implementation #AC might be signalled for a 2-byte misalignment, whereas #GP might be signalled for all other misalignments (4/8/16-byte). Invalid opcode exception if instruction is preceded by a LOCK override prefix. General protection fault if reserved bits of MXCSR are loaded with non-zero values

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set.

FXRSTOR: Restore FP and Intel® MMX™ Technology State and SSE State (Continued)

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Notes: State saved with FXSAVE and restored with FRSTOR (and vice versa) will result in incorrect restoration of state in the processor. The address size prefix will have the usual effect on address calculation but will have no effect on the format of the FXRSTOR image.

The use of Repeat (F2H, F3H) and Operand Size (66H) prefixes with FXRSTOR is reserved. Different processor implementations may handle this prefix differently. Use of this prefix with FXRSTOR risks incompatibility with future processors.

FXSAVE: Store FP and Intel® MMX™ Technology State and SSE State

Opcode	Instruction	Description
0F,AE,/0	FXSAVE m512byte	Store FP and Intel MMX technology state and SSE state to m512byte.

Operation:

```
m512byte = FP and MMX technology state and SSE state;
```

Description: The FXSAVE instruction writes the current FP and MMX technology state and SSE state (environment and registers) to the specified destination defined by m512byte. It does this without checking for pending unmasked floating-point exceptions, similar to the operation of FNSAVE. Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FP and MMX technology state and SSE state in the processor after the state has been saved. This instruction has been optimized to maximize floating-point save performance. The save data structure is as follows (little-endian byte order as arranged in memory, with byte offset into row described by right column):

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td colspan="2">Rsrvd</td><td colspan="2">CS</td><td colspan="4">IP</td><td colspan="2">FOP</td><td colspan="2">FTW</td><td colspan="2">FSW</td><td colspan="2">FCW</td><td>0</td></tr>
<tr><td colspan="4">Reserved</td><td colspan="4">MXCSR</td><td colspan="2">Rsrvd</td><td colspan="2">DS</td><td colspan="4">DP</td><td>16</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST0/MM0</td><td>32</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST1/MM1</td><td>48</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST2/MM2</td><td>64</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST3/MM3</td><td>80</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST4/MM4</td><td>96</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST5/MM5</td><td>112</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST6/MM6</td><td>128</td></tr>
<tr><td colspan="6">Reserved</td><td colspan="10">ST7/MM7</td><td>144</td></tr>
<tr><td colspan="16">XMM0</td><td>160</td></tr>
<tr><td colspan="16">XMM1</td><td>176</td></tr>
<tr><td colspan="16">XMM2</td><td>192</td></tr>
<tr><td colspan="16">XMM3</td><td>208</td></tr>
<tr><td colspan="16">XMM4</td><td>224</td></tr>
<tr><td colspan="16">XMM5</td><td>240</td></tr>
<tr><td colspan="16">XMM6</td><td>256</td></tr>
<tr><td colspan="16">XMM7</td><td>272</td></tr>
<tr><td colspan="16">Reserved</td><td>288</td></tr>
<tr><td colspan="16">Reserved</td><td>304</td></tr>
<tr><td colspan="16">Reserved</td><td>320</td></tr>
<tr><td colspan="16">Reserved</td><td>336</td></tr>
<tr><td colspan="16">Reserved</td><td>352</td></tr>
<tr><td colspan="16">Reserved</td><td>368</td></tr>
<tr><td colspan="16">Reserved</td><td>384</td></tr>
<tr><td colspan="16">Reserved</td><td>400</td></tr>
</table>

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td colspan="2">Rsrvd</td><td colspan="2">CS</td><td colspan="5">IP</td><td colspan="2">FOP</td><td colspan="2">FTW</td><td colspan="2">FSW</td><td>FCW</td><td>0</td></tr>
<tr><td colspan="16">Reserved</td><td>416</td></tr>
<tr><td colspan="16">Reserved</td><td>432</td></tr>
<tr><td colspan="16">Reserved</td><td>448</td></tr>
<tr><td colspan="16">Reserved</td><td>464</td></tr>
<tr><td colspan="16">Reserved</td><td>480</td></tr>
<tr><td colspan="16">Reserved</td><td>496</td></tr>
</table>

Three fields in the floating-point save area contain reserved bits that are not indicated in the table:

- FOP: The lower 11-bits contain the opcode, upper 5-bits are reserved.
- IP & DP: 32-bit mode: 32-bit IP-offset.
- 16-bit mode: lower 16-bits are IP-offset and upper 16-bits are reserved.

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to use the FP and MMX technology and SSE units. It cannot be used by an application program to pass a "clean" FP state to a procedure, since it retains the current state. An application must explicitly execute an FINIT instruction after FXSAVE to provide for this functionality.

All of the x87-FP fields retain the same internal format as in FSAVE except for FTW.

Unlike FSAVE, FXSAVE saves only the FTW valid bits rather than the entire x87-FP FTW field. The FTW bits are saved in a non-TOS relative order, which means that FR0 is always saved first, followed by FR1, FR2 and so forth. As an example, if TOS=4 and only ST0, ST1 and ST2 are valid, FSAVE saves the FTW field in the following format:

ST3	ST2	ST1	ST0	ST7	ST6	ST5	ST4 (TOS=4)
FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
11	xx	xx	xx	11	11	11	11

where xx is one of (00, 01, 10). (11) indicates an empty stack elements, and the 00, 01, and 10 indicate Valid, Zero, and Special, respectively. In this example, FXSAVE would save the following vector:

FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
0	1	1	1	0	0	0	0

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using the following table:

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW
0	0	0	0x	1	Special 10
0	0	0	1x	1	Valid 00
0	0	1	00	1	Special 10
0	0	1	10	1	Valid 00
0	1	0	0x	1	Special 10
0	1	0	1x	1	Special 10
0	1	1	00	1	Zero 01
0	1	1	10	1	Special 10

Exponent all 1’s	Exponent all 0’s	Fraction all 0’s	J and M bits	FTW valid bit	x87 FTW
1	0	0	1x	1	Special 10
1	0	0	1x	1	Special 10
1	0	1	00	1	Special 10
1	0	1	10	1	Special 10
For all legal combinations above				0	Empty 11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e. the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0’s.

If the FXSAVE instruction is immediately preceded by an FP instruction which does not use a memory operand, then the FXSAVE instruction does not write/update the DP field, in the FXSAVE image.

MXCSR holds the contents of the SSE Control/Status Register. See the LDMXCSR instruction for a full description of this field.

The fields XMM0-XMM7 contain the content of registers XMM0-XMM7 in exactly the same format as they exist in the registers.

The SSE fields in the save image (XMM0-XMM7 and MXCSR) may not be loaded into the processor if the CR4.OSFXSR bit is not set. This CR4 bit must be set in order to enable execution of SSE instructions.

The destination m512byte is assumed to be aligned on a 16-byte boundary. If m512byte is not aligned on a 16-byte boundary, FXSAVE generates a general protection exception.

FP Exceptions: If #AC exception detection is disabled, a general protection exception is signalled if the address is not aligned on 16-byte boundary. Note that if #AC is enabled (and CPL is 3), signalling of #AC is not guaranteed and may vary with implementation; in all implementations where #AC is not signalled, a general protection fault will instead be signalled. In addition, the width of the alignment check when #AC is enabled may also vary with implementation; for instance, for a given implementation #AC might be signalled for a 2-byte misalignment, whereas #GP might be signalled for all other misalignments (4/8/16-byte). Invalid opcode exception if instruction is preceded by a LOCK override prefix.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

FXSAVE: Store FP and Intel® MMX™ Technology State and SSE State (Continued)

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #NM if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Notes: State saved with FXSAVE and restored with FRSTOR (and vice versa) will result in incorrect restoration of state in the processor. The address size prefix will have the usual effect on address calculation but will have no effect on the format of the FXSAVE image.

If there is a pending unmasked FP exception at the time FXSAVE is executed, the sequence of FXSAVE-FWAIT-FXRSTOR will result in incorrect state in the processor. The FWAIT instruction causes the processor to check and handle pending unmasked FP exceptions. Since the processor does not clear the FP state with FXSAVE (unlike FSAVE), the exception is handled but that fact is not reflected in the saved image. When the image is reloaded using FXRSTOR, the exception bits in FSW will be incorrectly reloaded.

The use of Repeat (F2H, F3H) and Operand Size (66H) prefixes with FXSAVE is reserved. Different processor implementations may handle this prefix differently. Use of these prefixes with FXSAVE risks incompatibility with future processors.

LDMXCSR: Load SSE Control/Status

Opcode	Instruction	Description
0F,AE,/2	LDMXCSR m32	Load SSE control/status word from m32.

Operation: `MXCSR = m32;`

Description: The MXCSR control/status register is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags. The following figure shows the format and encoding of the fields in MXCSR.

Reserved	FZ	RC	RC	PM	UM	OM	ZM	DM	IM	Rsvd	PE	UE	OE	ZE	DE	IE

31-16 15 10 5
0

Bits 5-0 indicate whether an SSE numerical exception has been detected. They are "sticky" flags, and can be cleared by using the LDMXCSR instruction to write zeroes to these fields. If a LDMXCSR instruction clears a mask bit and sets the corresponding exception flag bit, an exception will not be immediately generated. The exception will occur only upon the next SSE instruction to cause this type of exception. The Intel SSE architecture uses only one exception flag for each exception. There is no provision for individual exception reporting within a packed data type. In situations where multiple identical exceptions occur within the same instruction, the associated exception flag is updated and indicates that at least one of these conditions happened. These flags are cleared upon reset.

Bits 12-7 configure numerical exception masking; an exception type is masked if the corresponding bit is set and it is unmasked if the bit is clear. These enables are set upon reset, meaning that all numerical exceptions are masked.

Bits 14-13 encode the rounding-control, which provides for the common round-to-nearest mode, as well as directed rounding and true chop. Rounding control affects the arithmetic instructions and certain conversion instructions. The encoding for RC is as follows:

Rounding Mode	RC Field	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero).
Round down (to minus infinity)	01B	Rounded result is close to but no greater than the infinitely precise result
Round up (toward positive infinity)	10B	Rounded result is close to but no less than the infinitely precise result.
Round toward zero (truncate)	11B	Rounded result is close to but no greater in absolute value than the infinitely precise result.

The rounding-control is set to round to nearest upon reset.

LDMXCSR: Load SSE Control/Status (Continued)

Bit 15 (FZ) is used to turn on the Flush To Zero mode (bit is set). Turning on the Flush To Zero mode has the following effects during underflow situations:

- Zero results are returned with the sign of the true result.
- Precision and underflow exception flags are set.

The IEEE mandated masked response to underflow is to deliver the denormalized result (i.e. gradual underflow); consequently, the flush to zero mode is not compatible with IEEE Std. 754. It is provided primarily for performance reasons. At the cost of a slight precision loss, faster execution can be achieved for applications where underflows are common. Unmasking the underflow exception takes precedence over Flush To Zero mode; this means that an exception handler will be invoked for a SSE instruction that generates an underflow condition while this exception is unmasked, regardless of whether flush to zero is enabled.

The other bits of MXCSR (bits 31-16 and bit 6) are defined as reserved and cleared; attempting to write a non-zero value to these bits, using either the FXRSTOR or LDMXCSR instructions, will result in a general protection exception.

The linear address corresponds to the address of the least-significant byte of the referenced memory data.

FP Exceptions: General protection fault if reserved bits are loaded with non-zero values.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault. #AC for unaligned memory reference.

LDMXCSR: Load SSE Control/Status (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with LDMXCSR is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with LDMXCSR risks incompatibility with future processors.

MAXPS: Packed Single-FP Maximum

Opcode	Instruction	Description
0F,5F,/r	MAXPS xmm1, xmm2/m128	Return the maximum SP FP numbers between XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]       =   (xmm1[31-0] == NAN) ? xmm2[31-0] :

                     (xmm2[31-0] == NAN) ? xmm2[31-0] :

                     (xmm1[31-0] > xmm2/m128[31-0]) ? xmm1[31-0] ?
xmm2/m128[31-0];

xmm1[63-32]      =   (xmm1[63-32] == NAN) ? xmm2[63-32] :

                     (xmm2[63-32] == NAN) ? xmm2[63-32] :

                     (xmm1[63-32] > xmm2/m128[63-32]) ? xmm1[63-32] ?
xmm2/m128[63-32];

xmm1[95-64]      =   (xmm1[95-64] == NAN) ? xmm2[95-64] :

                     (xmm2[95-64] == NAN) ? xmm2[95-64] :

                     (xmm1[95-64] > xmm2/m128[95-64]) ? xmm1[95-64] ?
xmm2/m128[95-64];

xmm1[127-96]     =   (xmm1[127-96] == NAN) ? xmm2[127-96] :

                     (xmm2[127-96] == NAN) ? xmm2[127-96] :

                     (xmm1[127-96] > xmm2/m128[127-96]) ? xmm1[127-96] ?
xmm2/m128[127-96];
```

Description: The MAXPS instruction returns the maximum SP FP numbers from XMM1 and XMM2/Mem. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

MAXPS: Packed Single-FP Maximum (Continued)

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 4-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MAXPS allows compilers to use the MAXPS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MAXSS: Scalar Single-FP Maximum

Opcode	Instruction	Description
F3,0F,5F,/r	MAXSS xmm1, xmm2/m32	Return the maximum SP FP number between the lower SP FP numbers from XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]   = (xmm1[31-0] == NAN) ? xmm2[31-0] :

               (xmm2[31-0] == NAN) ? xmm2[31-0] :

               (xmm1[31-0] > xmm2/m32[31-0]) ? xmm1[31-0] : xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The MAXSS instruction returns the maximum SP FP number from the lower SP FP numbers of XMM1 and XMM2/Mem; the upper 3 fields are passed through from xmm1. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: None

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

MAXSS: Scalar Single-FP Maximum (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 4-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. The upper three operands are still bypassed from the src1 operand, as in all other scalar operations. This approach for MAXSS allows compilers to use the MAXSS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINPS: Packed Single-FP Minimum

Opcode	Instruction	Description
0F,5D,/r	MINPS xmm1, xmm2/m128	Return the minimum SP numbers between XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0]      =   (xmm1[31-0] == NAN) ? xmm2[31-0] :
                    (xmm2[31-0] == NAN) ? xmm2[31-0] :
                    (xmm1[31-0] < xmm2/m128[31-0]) : xmm1[31-0] ?
xmm2/m128[31-0];
xmm1[63-32]     =   (xmm1[63-32] == NAN) ? xmm2[63-32] :
                    (xmm2[63-32] == NAN) ? xmm2[63-32] :
                    (xmm1[63-32] < xmm2/m128[63-32]) : xmm1[63-32] ?
xmm2/m128[63-32];
xmm1[95-64]     =   (xmm1[95-64] == NAN) ? xmm2[95-64] :
                    (xmm2[95-64] == NAN) ? xmm2[95-64] :
                    (xmm1[95-64] < xmm2/m128[95-64]) : xmm1[95-64] ?
xmm2/m128[95-64];
xmm1[127-96]    =   (xmm1[127-96] == NAN) ? xmm2[127-96] :
                    (xmm2[127-96] == NAN) ? xmm2[127-96] :
                    (xmm1[127-96] < xmm2/m128[127-96]) : xmm1[127-96] ?
xmm2/m128[127-96];
```

Description: The MINPS instruction returns the minimum SP FP numbers from XMM1 and XMM2/Mem. If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

MINPS: Packed Single-FP Minimum (Continued)

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 4-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. This approach for MINPS allows compilers to use the MINPS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MINSS: Scalar Single-FP Minimum

Opcode	Instruction	Description
F3,0F,5D,/r	MINSS xmm1, xmm2/m32	Return the minimum SP FP number between the lowest SP FP numbers from XMM2/Mem and XMM1.

Operation:

```
xmm1[31-0] = (xmm1[31-0] == NAN) ? xmm2[31-0] :
              (xmm2[31-0] == NAN) ? xmm2[31-0] :
             (xmm1[31-0] < xmm2/m32[31-0]) ? xmm1[31-0] : xmm2/m32[31-0];
xmm1[63-32]  = xmm1[63-32];
xmm1[95-64]  = xmm1[95-64];
xmm1[127-96] = xmm1[127-96];
```

Description: The MINSS instruction returns the minimum SP FP number from the lower SP FP numbers from XMM1 and XMM2/Mem; the upper 3 fields are passed through from xmm1.If the values being compared are both zeros, source2 (xmm2/m128) would be returned. If source2 (xmm2/m128) is an sNaN, this sNaN is forwarded unchanged to the destination (i.e. a quieted version of the sNaN is not returned).

FP Exceptions: None

Numeric Exceptions: Invalid (including qNaN source operand), Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory references.

MINSS: Scalar Single-FP Minimum (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: Note that if only one source is a NaN for these instructions, the Src2 operand (either NaN or real value) is written to the result; this differs from the behavior for other instructions as defined in Table 4-3, which is to always write the NaN to the result, regardless of which source operand contains the NaN. The upper three operands are still bypassed from the src1 operand, as in all other scalar operations. This approach for MINSS allows compilers to use the MINSS instruction for common C conditional constructs. If instead of this behavior, it is required that the NaN source operand be returned, the min/max functionality can be emulated using a sequence of instructions: comparison followed by AND, ANDN and OR.

MOVAPS: Move Aligned Four Packed Single-FP

Opcode	Instruction	Description
0F,28,/r	MOVAPS xmm1, xmm2/m128	Move 128 bits representing 4 packed SP data from XMM2/Mem to XMM1 register.
0F,29,/r	MOVAPS xmm2/m128, xmm1	Move 128 bits representing 4 packed SP from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127=0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded or stored. When the register-register form of this operation is used, the content of the 128-bit source register is copied into 128-bit destination register.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

MOVAPS: Move Aligned Four Packed Single-FP (Continued)

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: MOVAPS should be used when dealing with 16-byte aligned SP FP numbers. If the data is not known to be aligned, MOVUPS should be used instead of MOVAPS. The usage of this instruction should be limited to the cases where the aligned restriction is easy to meet. Processors that support the Intel SSE architecture will provide optimal aligned performance for the MOVAPS instruction.

The usage of Repeat Prefixes (F2H, F3H) with MOVAPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVAPS risks incompatibility with future processors.

MOVHLPS: Move High to Low Packed Single-FP

Opcode	Instruction	Description
0F,12,/r	MOVHLPS xmm1, xmm2	Move 64 bits representing higher two SP operands from XMM2 to lower two fields of XMM1 register.

Operation:

```
// move instruction

xmm1[127-64] = xmm1[127-64];

xmm1[63-0] = xmm2[127-64];
```

Description: The upper 64-bits of the source register xmm2 are loaded into the lower 64-bits of the 128-bit register xmm1 and the upper 64-bits of xmm1 are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with MOVHLPS is reserved. Different processor implementations may handle these prefixes differently. Usage of these prefixes with MOVHLPS risks incompatibility with future processors.

MOVHPS: Move High Packed Single-FP

Opcode	Instruction	Description
0F,16,/r	MOVHPS xmm, m64	Move 64 bits representing two SP operands from Mem to upper two fields of XMM register.
0F,17,/r	MOVHPS m64, xmm	Move 64 bits representing two SP operands from upper two fields of XMM register to Mem.

Operation:

```
if (destination == xmm) {
    // load instruction
    xmm[127-64]   = m64;
    xmm[31-0]  = xmm[31-0];
    xmm[63-32] = xmm[63-32];
}
else {
    // store instruction
    m64 = xmm[127-64];
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When the load form of this operation is used, m64 is loaded into the upper 64-bits of the 128-bit register xmm and the lower 64-bits are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVHPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVHPS risks incompatibility with future processors.

MOVLHPS: Move Low to High Packed Single-FP

Opcode	Instruction	Description
0F,16,/r	MOVLHPS xmm1, xmm2	Move 64 bits representing lower two SP operands from XMM2 to upper two fields of XMM1 register.

Operation:

```
// move instruction

xmm1[127-64] = xmm2[63-0];

xmm1[63-0] = xmm1[63-0];
```

Description: The lower 64-bits of the source register xmm2 are loaded into the upper 64-bits of the 128-bit register xmm1 and the lower 64-bits of xmm1 are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1

Comments:

Example: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with MOVLHPS is reserved. Different processor implementations may handle these prefixes differently. Usage of these prefixes with MOVLHPS risks incompatibility with future processors.

MOVLPS: Move Low Packed Single-FP

Opcode	Instruction	Description
0F,12,/r	MOVLPS xmm, m64	Move 64 bits representing two SP operands from Mem to lower two fields of XMM register.
0F,13,/r	MOVLPS m64, xmm	Move 64 bits representing two SP operands from lower two fields of XMM register to Mem.

Operation:

```
if (destination == xmm) {
    // load instruction
    xmm[63-0]    = m64;
    xmm[95-64]   = xmm[95-64];
    xmm[127-96]  = xmm[127-96];
}
else {
    // store instruction
    m64 = xmm[63-0];
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When the load form of this operation is used, m64 is loaded into the lower 64-bits of the 128-bit register xmm and the upper 64-bits are left unchanged.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

MOVLPS: Move Low Packed Single-FP (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVLPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVLPS risks incompatibility with future processors.

MOVMSKPS: Move Mask to Integer

Opcode	Instruction	Description
0F,50,/r	MOVMSKPS r32, xmm	Move the single mask to r32.

Operation:

```
r32[3] = xmm[127]; r32[2] = xmm[95];

r32[1] = xmm[63];  r32[0] = xmm[31];

r32[7-4] = 0x0; r32[15-8] = 0x00;

r32[31-16] = 0x0000;
```

Description: The MOVMSKPS instruction returns to the integer register r32 a 4-bit mask formed of the most significant bits of each SP FP number of its operand.

FP Exceptions: None

Numeric Exceptions: None.

Protected Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

#UD if CR0.EM = 1; #NM if TS bit in CR0 is set.; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with MOVMSKPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVMSKPS risks incompatibility with future processors.

MOVSS: Move Scalar Single-FP

Opcode	Instruction	Description
F3,0F,10,/r	MOVSS xmm1, xmm2/m32	Move 32 bits representing one scalar SP operand from XMM2/Mem to XMM1 register.
F3,0F,11,/r	MOVSS xmm2/m32, xmm1	Move 32 bits representing one scalar SP operand from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m32) {
        // load instruction
        xmm1[31-0]   = m32;
        xmm1[63-32]  = 0x00000000;
        xmm1[95-64]  = 0x00000000;
        xmm1[127-96] = 0x00000000;
    }
    else {
        // move instruction
        xmm1[31-0]   = xmm2[31-0];
        xmm1[63-32]  = xmm1[63-32];
        xmm1[95-64]  = xmm1[95-64];
        xmm1[127-96] = xmm1[127-96];
    }
}
else {
    if (destination == m32) {
        // store instruction
        m32 = xmm1[31-0];
    }
    else {
        // move instruction
        xmm2[31-0]   = xmm1[31-0]
        xmm2[63-32]  = xmm2[63-32];
        xmm2[95-64]  = xmm2[95-64];
```

MOVSS: Move Scalar Single-FP (Continued)

```
        xmm2[127-96] = xmm2[127-96];
    }
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 4 bytes of data at memory location m32 are loaded or stored. When the load form of this operation is used, the 32-bits from memory are copied into the lower 32 bits of the 128-bit register xmm, the 96 most significant bits being cleared.

FP Exceptions: None

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

MOVUPS: Move Unaligned Four Packed Single-FP

Opcode	Instruction	Description
0F,10,/r	MOVUPS xmm1, xmm2/m128	Move 128 bits representing four SP data from XMM2/Mem to XMM1 register.
0F,11,/r	MOVUPS xmm2/m128, xmm1	Move 128 bits representing four SP data from XMM1 register to XMM2/Mem.

Operation:

```
if (destination == xmm1) {
    if (source == m128) {
        // load instruction
        xmm1[127-0] = m128;
    }
    else {
        // move instruction
        xmm1[127-0] = xmm2[127-0];
    }
}
else {
    if (destination == m128) {
        // store instruction
        m128 = xmm1[127-0];
    }
    else {
        // move instruction
        xmm2[127-0] = xmm1[127-0];
    }
}
```

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded to the 128-bit multimedia register xmm or stored from the 128-bit multimedia register xmm. When the register-register form of this operation is used, the content of the 128-bit source register is copied into 128-bit register xmm. No assumption is made about alignment.

FP Exceptions: None

Numeric Exceptions: None

MOVUPS: Move Unaligned Four Packed Single-FP (Continued)

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #AC for unaligned memory reference if the current privilege level is 3; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVUPS should be used with SP FP numbers when that data is known to be unaligned.The usage of this instruction should be limited to the cases where the aligned restriction is hard or impossible to meet. SSE implementations guarantee optimum unaligned support for MOVUPS. Efficient SSE applications should mainly rely on MOVAPS, not MOVUPS, when dealing with aligned data.

The usage of Repeat-NE Prefix (F2H) and Operand Size Prefix (66H) with MOVUPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVUPS risks incompatibility with future processors.

A linear address of the 128 bit data access, while executing in 16-bit mode, that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. Different processor implementations may/may not raise a GP fault in this case if the segment limit has been exceeded; additionally, the address that spans the end of the segment may/may not wrap around to the beginning of the segment.

MULPS: Packed Single-FP Multiply

Opcode	Instruction	Description
0F,59,/r	MULPS xmm1, xmm2/m128	Multiply packed SP FP numbers in XMM2/Mem to XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0]   * xmm2/m128[31-0];

xmm1[63-32]  = xmm1[63-32]  * xmm2/m128[63-32];

xmm1[95-64]  = xmm1[95-64]  * xmm2/m128[95-64];

xmm1[127-96] = xmm1[127-96] * xmm2/m128[127-96];
```

Description: The MULPS instructions multiply the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0).

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

MULSS: Scalar Single-FP Multiply

Opcode	Instruction	Description
F3,0F,59,/r	MULSS xmm1 xmm2/m32	Multiply the lowest SP FP number in XMM2/Mem to XMM1.

```
xmm1[31-0]   = xmm1[31-0] * xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The MULSS instructions multiply the lowest SP FP numbers of both their operands; the upper 3 fields are passed through from xmm1.

FP Exceptions: None

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0).

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

ORPS: Bit-wise Logical OR for Single-FP Data

Opcode	Instruction	Description
0F,56,/r	ORPS xmm1, xmm2/m128	OR 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm1[127-0] |= xmm2/m128[127-0];`

Description: The ORPS instructions return a bit-wise logical OR between xmm1 and xmm2/mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with ORPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with ORPS risks incompatibility with future processors.

RCPPS: Packed Single-FP Reciprocal

Opcode	Instruction	Description
0F,53,/r	RCPPS xmm1, xmm2/m128	Return a packed approximation of the reciprocal of XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/(xmm2/m128[31-0]));

xmm1[63-32]  = approx (1.0/(xmm2/m128[63-32]));

xmm1[95-64]  = approx (1.0/(xmm2/m128[95-64]));

xmm1[127-96] = approx (1.0/(xmm2/m128[127-96]));
```

Description: RCPPS returns an approximation of the reciprocal of the SP FP numbers from xmm2/m128. The relative error for this approximation is Error, which satisfies:

$|\mathrm{Error}| <= 1.5x2^{-12}$

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RCPPS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and tiny results are always flushed to zero, with the sign of the operand.

Results are guaranteed not to be tiny, and therefore not flushed to zero, for input values x which satisfy

$|x| <= 1.11111111110100000000000_B \times 2^{125}$

RCPPS: Packed Single-FP Reciprocal (Continued)

For input values x which satisfy

$1.11111111110100000000001_B \times 2^{125}$ <= |x| <=
$1.00000000000110000000000_B \times 2^{126}$

flush-to-zero might or might not occur, depending on the implementation (this interval contains 6144 + 3072 = 9216 single precision floating-point numbers).

Results are guaranteed to be tiny, and therefore flushed to zero, for input values x which satisfy

|x| <= $1.00000000000110000000001_B \times 2^{126}$

The decimal approximations of the single precision numbers that delimit the three intervals specified above, are as follows:

$1.11111111110100000000000_B \times 2^{125}$ ~= 8.5039437×10^{37}

$1.11111111110100000000001_B \times 2^{125}$ ~= 8.5039443×10^{37}

$1.00000000000110000000000_B \times 2^{126}$ ~= 4.2550872×10^{37}

$1.00000000000110000000001_B \times 2^{126}$ ~= 4.2550877×10^{37}

The hexadecimal representations of the single precision numbers that delimit the three intervals specified above, are as follows:

$1.11111111110100000000000_B \times 2^{125}$ = 0x7e7fe800

$1.11111111110100000000001_B \times 2^{125}$ = 0x7e7fe801

$1.00000000000110000000000_B \times 2^{126}$ = 0x7e800c00

$1.00000000000110000000001_B \times 2^{126}$ = 0x7e800c01

RCPSS: Scalar Single-FP Reciprocal

Opcode	Instruction	Description
F3,0F,53,/r	RCPSS xmm1, xmm2/m32	Return an approximation of the reciprocal of the lower SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/(xmm2/m32[31-0]));

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: RCPSS returns an approximation of the reciprocal of the lower SP FP number from xmm2/m32; the upper 3 fields are passed through from xmm1. The relative error for this approximation is Error, which satisfies:

$|Error| <= 1.5x2^{-12}$

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #AC for unaligned memory reference if the current privilege level is 3; #NM if TS bit in CR0 is set.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RCPSS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign) and tiny results are always flushed to zero, with the sign of the operand.

Results are guaranteed not to be tiny, and therefore not flushed to zero, for input values x which satisfy

$|x| <= 1.11111111110100000000000_B \times 2^{125}$

RCPSS: Scalar Single-FP Reciprocal (Continued)

For input values x which satisfy

$1.11111111110100000000001_B\times 2^{125}$ <= |x| <=
$1.00000000000110000000000_B\times 2^{126}$

flush-to-zero might or might not occur, depending on the implementation (this interval contains 6144 + 3072 = 9216 single precision floating-point numbers).

Results are guaranteed to be tiny, and therefore flushed to zero, for input values x which satisfy

|x| <= $1.00000000000110000000001_B\times 2^{126}$

The decimal approximations of the single precision numbers that delimit the three intervals specified above, are as follows:

$1.11111111110100000000000_B\times 2^{125}$ ~= 8.5039437×10^{37}

$1.11111111110100000000001_B\times 2^{125}$ ~= 8.5039443×10^{37}

$1.00000000000110000000000_B\times 2^{126}$ ~= 4.2550872×10^{37}

$1.00000000000110000000001_B\times 2^{126}$ ~= 4.2550877×10^{37}

The hexadecimal representations of the single precision numbers that delimit the three intervals specified above, are as follows:

$1.11111111110100000000000_B\times 2^{125}$ = 0x7e7fe800

$1.11111111110100000000001_B\times 2^{125}$ = 0x7e7fe801

$1.00000000000110000000000_B\times 2^{126}$ = 0x7e800c00

$1.00000000000110000000001_B\times 2^{126}$ = 0x7e800c01

RSQRTPS: Packed Single-FP Square Root Reciprocal

Opcode	Instruction	Description
0F,52,/r	RSQRTPS xmm1, xmm2/m128	Return a packed approximation of the square root of the reciprocal of XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/sqrt(xmm2/m128[31-0]));

xmm1[63-32]  = approx (1.0/sqrt(xmm2/m128[63-32]));

xmm1[95-64]  = approx (1.0/sqrt(xmm2/m128[95-64]));

xmm1[127-96] = approx (1.0/sqrt(xmm2/m128[127-96]));
```

Description: RSQRTPS returns an approximation of the reciprocal of the square root of the SP FP numbers from xmm2/m128. The relative error for this approximation is Error, which satisfies:

$|Error| <= 1.5x2^{-12}$

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: RSQRTPS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign).

RSQRTSS: Scalar Single-FP Square Root Reciprocal

Opcode	Instruction	Description
F3,0F,52,/r	RSQRTSS xmm1, xmm2/m32	Return an approximation of the square root of the reciprocal of the lowest SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = approx (1.0/sqrt(xmm2/m32[31-0]));

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: RSQRTSS returns an approximation of the reciprocal of the square root of the lowest SP FP number from xmm2/m32; the upper 3 fields are passed through from xmm1. The relative error for this approximation is Error, which satisfies:

$|Error| <= 1.5x2^{-12}$

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments:

Example: RSQRTSS is not affected by the rounding control in MXCSR. Denormal inputs are treated as zeros (of the same sign).

SHUFPS: Shuffle Single-FP

Opcode	Instruction	Description
0F,C6,/r, ib	SHUFPS xmm1, xmm2/m128, imm8	Shuffle Single.

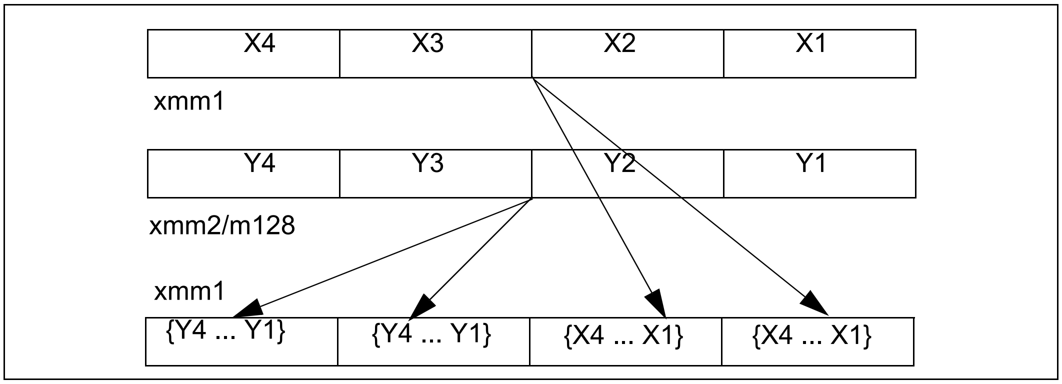
Operation:

```
fp_select  = (imm8 >> 0) & 0x3;

xmm1[31-0]    = (fp_select == 0) ? xmm1[31-0]   :

                (fp_select == 1) ? xmm1[63-32]  :

                (fp_select == 2) ? xmm1[95-64]  :

                                   xmm1[127-96];

fp_select  = (imm8 >> 2) & 0x3;

xmm1[63-32]   = (fp_select == 0) ? xmm1[31-0]   :

                (fp_select == 1) ? xmm1[63-32]  :

                (fp_select == 2) ? xmm1[95-64]  :

                                   xmm1[127-96];

fp_select  = (imm8 >> 4) & 0x3;

xmm1[95-64]   = (fp_select == 0) ? xmm2/m128[31-0]   :

                (fp_select == 1) ? xmm2/m128[63-32]  :

                (fp_select == 2) ? xmm2/m128[95-64]  :

                                   xmm2/m128[127-96];

fp_select  = (imm8 >> 6) & 0x3;

xmm1[127-96] = (fp_select == 0) ? xmm2/m128[31-0]   :

                (fp_select == 1) ? xmm2/m128[63-32]  :

                (fp_select == 2) ? xmm2/m128[95-64]  :

                                   xmm2/m128[127-96];
```

Description: The SHUFPS instruction is able to shuffle any of the four SP FP numbers from xmm1 to the lower 2 destination fields; the upper 2 destination fields are generated from a shuffle of any of the four SP FP numbers from xmm2/m128. By using the same register for both sources, SHUFPS can return any combination of the four SP FP numbers from this register. Bits 0 and 1 of the immediate field are used to select which of the four input SP FP numbers will be put in the first SP FP number of the result; bits 3 and 2 of the immediate field are used to select which of the four input SP FP will be put in the second SP FP number of the result; etc.

SHUFPS: Shuffle Single-FP (Continued)

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: The usage of Repeat Prefixes (F2H, F3H) with SHUFPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with SHUFPS risks incompatibility with future processors.

SQRTPS: Packed Single-FP Square Root

Opcode	Instruction	Description
0F,51,/r	SQRTPS xmm1, xmm2/m128	Square Root of the packed SP FP numbers in XMM2/Mem.

Operation:

```
xmm1[31-0]   = sqrt (xmm2/m128[31-0]);

xmm1[63-32]  = sqrt (xmm2/m128[63-32]);

xmm1[95-64]  = sqrt (xmm2/m128[95-64]);

xmm1[127-96] = sqrt (xmm2/m128[127-96]);
```

Description: The SQRTPS instruction returns the square root of the packed SP FP numbers from xmm2/m128.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

SQRTSS: Scalar Single-FP Square Root

Opcode	Instruction	Description
F3,0F,51,/r	SQRTSS xmm1, xmm2/m32	Square Root of the lower SP FP number in XMM2/Mem.

Operation:

```
xmm1[31-0]   = sqrt (xmm2/m32[31-0]);

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The SQRTSS instructions return the square root of the lowest SP FP numbers of their operand.

FP Exceptions: None

Numeric Exceptions: Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

STMXCSR: Store SSE Control/Status

Opcode	Instruction	Description
0F,AE,/3	STMXCSR m32	Store SSE control/status word to m32.

Operation: `m32 = MXCSR;`

Description: The MXCSR control/status register is used to enable masked/unmasked exception handling, to set rounding modes, to set flush-to-zero mode, and to view exception status flags. Refer to LDMXCSR for a description of the format of MXCSR. The linear address corresponds to the address of the least-significant byte of the referenced memory data. The reserved bits in the MXCSR are stored as zeroes.

FP Exceptions: None.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault. #AC for unaligned memory reference.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: The usage of Repeat (F2H, F3H) and Operand Size (66H) prefixes with STMXCSR is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with STMXCSR risks incompatibility with future processors.

SUBPS: Packed Single-FP Subtract

Opcode	Instruction	Description
0F,5C,/r	SUBPS xmm1 xmm2/m128	Subtract packed SP FP numbers in XMM2/Mem from XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0]   - xmm2/m128[31-0];

xmm1[63-32]  = xmm1[63-32]  - xmm2/m128[63-32];

xmm1[95-64]  = xmm1[95-64]  - xmm2/m128[95-64];

xmm1[127-96] = xmm1[127-96] - xmm2/m128[127-96];
```

Description: The SUBPS instruction subtracts the packed SP FP numbers of both their operands.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault;.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

SUBSS: Scalar Single-FP Subtract

Opcode	Instruction	Description
F3,0F,5C, /r	SUBSS xmm1, xmm2/m32	Subtract the lower SP FP numbers in XMM2/Mem from XMM1.

Operation:

```
xmm1[31-0]   = xmm1[31-0] - xmm2/m32[31-0];

xmm1[63-32]  = xmm1[63-32];

xmm1[95-64]  = xmm1[95-64];

xmm1[127-96] = xmm1[127-96];
```

Description: The SUBSS instruction subtracts the lower SP FP numbers of both their operands.

FP Exceptions: None.

Numeric Exceptions: Overflow, Underflow, Invalid, Precision, Denormal.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

UCOMISS: Unordered Scalar Single-FP Compare and Set EFLAGS

Opcode	Instruction	Description
0F,2E,/r	UCOMISS xmm1, xmm2/m32	Compare lower SP FP number in XMM1 register with lower SP FP number in XMM2/Mem and set the status flags accordingly.

Operation:

```
switch (xmm1[31-0] <> xmm2/m32[31-0]) {

   OF,SF,AF = 000;

   case UNORDERED:     ZF,PF,CF = 111;

   case GREATER_THAN:  ZF,PF,CF = 000;

   case LESS_THAN:     ZF,PF,CF = 001;

   case EQUAL:         ZF,PF,CF = 100;

}
```

Description: The UCOMISS instructions compare the two lowest scalar SP FP numbers and sets the ZF,PF,CF bits in the EFLAGS register as described above. In addition, the OF, SF and AF bits in the EFLAGS register are zeroed out. The unordered predicate is returned if either source operand is a NaN (qNaN or sNaN).

FP Exceptions: None.

Numeric Exceptions: Invalid (if SNaN operands), Denormal. Integer EFLAGS values will not be updated in the presence of unmasked numeric exceptions.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3); #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #XM for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =1); #UD for an unmasked SSE numeric exception (CR4.OSXMMEXCPT =0); #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

UCOMISS: Unordered Scalar Single-FP Compare and Set EFLAGS (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: UCOMISS differs from COMISS in that it signals an invalid numeric exception when a source operand is an sNaN; COMISS signals invalid if a source operand is either a qNaN or an sNaN.

The usage of Repeat (F2H, F3H) and Operand-Size prefixes with UCOMISS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UCOMISS risks incompatibility with future processors.

UNPCKHPS: Unpack High Packed Single-FP Data

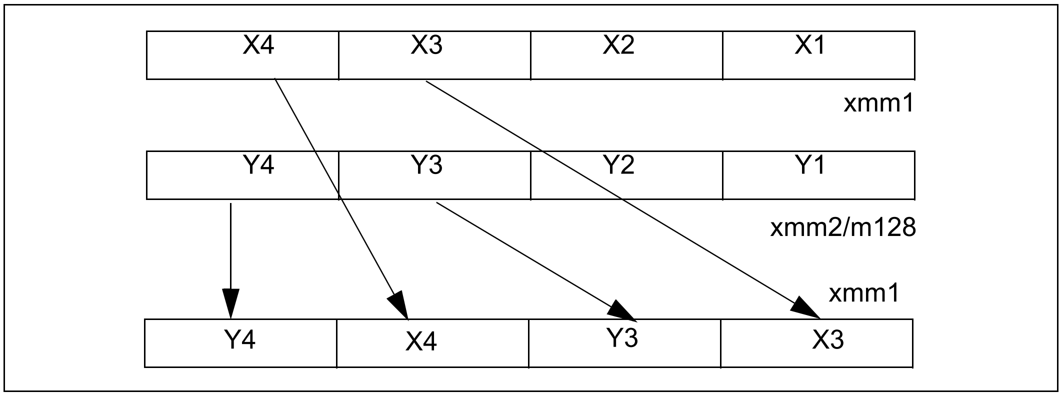
Opcode	Instruction	Description
0F,15,/r	UNPCKHPS xmm1, xmm2/m128	Interleaves SP FP numbers from the high halves of XMM1 and XMM2/Mem into XMM1 register.

Operation:

```
xmm1[31-0]   = xmm1[95-64];

xmm1[63-32]  = xmm2/m128[95-64];

xmm1[95-64]  = xmm1[127-96];

xmm1[127-96] = xmm2/m128[127-96];
```

Description: The UNPCKHPS instruction performs an interleaved unpack of the high-order data elements of XMM1 and XMM2/Mem. It ignores the lower half of the sources.

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

UNPCKHPS: Unpack High Packed Single-FP Data (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

The usage of Repeat Prefixes (F2H, F3H) with UNPCKHPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UNPCKHPS risks incompatibility with future processors.

UNPCKLPS: Unpack Low Packed Single-FP Data

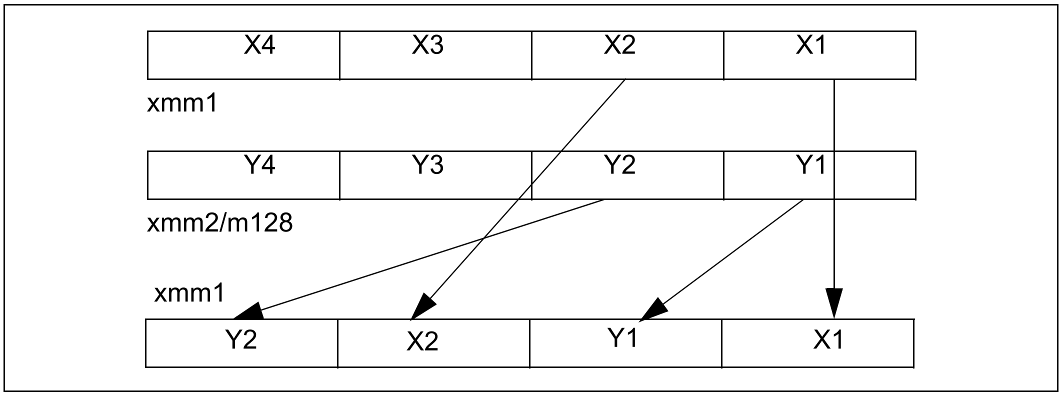
Opcode	Instruction	Description
0F,14,/r	UNPCKLPS xmm1, xmm2/m128	Interleaves SP FP numbers from the low halves of XMM1 and XMM2/Mem into XMM1 register.

Operation:

```
xmm1[31-0]   = xmm1[31-0];

xmm1[63-32]  = xmm2/m128[31-0];

xmm1[95-64]  = xmm1[63-32];

xmm1[127-96] = xmm2/m128[63-32];
```

Description: The UNPCKLPS instruction performs an interleaved unpack of the low-order data elements of XMM1 and XMM2/Mem. It ignores the upper half part of the sources.

Example:



FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

UNPCKLPS: Unpack Low Packed Single-FP Data (Continued)

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments: When unpacking from a memory operand, an implementation may decide to fetch only the appropriate 64 bits. Alignment to 16-byte boundary and normal segment checking will still be enforced.

The usage of Repeat Prefixes (F2H, F3H) with UNPCKLPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with UNPCKLPS risks incompatibility with future processors.

XORPS: Bit-wise Logical Xor for Single-FP Data

Opcode	Instruction	Description
0F,57,/r	XORPS xmm1, xmm2/m128	XOR 128 bits from XMM2/Mem to XMM1 register.

Operation: `xmm[127-0] ^= xmm/m128[127-0];`

Description: The XORPS instruction returns a bit-wise logical XOR between XMM1 and XMM2/Mem.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

Comments:

The usage of Repeat Prefixes (F2H, F3H) with XORPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with XORPS risks incompatibility with future processors.

4.13 SIMD Integer Instruction Set Extensions

Additional new SIMD Integer instructions have been added to accelerate the performance of 3D graphics, video decoding and encoding and other applications. These instructions operate on the MMX technology registers and on 64-bit memory operands.

PAVGB/PAVGW: Packed Average

Opcode	Instruction	Description
0F,E0, /r	PAVGB mm1,mm2/m64	Average with rounding packed unsigned bytes from MM2/Mem to packed bytes in MM1 register.
0F,E3, /r	PAVGW mm1, mm2/m64	Average with rounding packed unsigned words from MM2/Mem to packed words in MM1 register.

Operation:

```
if (instruction == PAVGB) {
    x[0]   = mm1[7-0]          y[0] = mm2/m64[7-0];
    x[1]   = mm1[15-8]         y[1] = mm2/m64[15-8];
    x[2]   = mm1[23-16]        y[2] = mm2/m64[23-16];
    x[3]   = mm1[31-24]        y[3] = mm2/m64[31-24];
    x[4]   = mm1[39-32]        y[4] = mm2/m64[39-32];
    x[5]   = mm1[47-40]        y[5] = mm2/m64[47-40];
    x[6]   = mm1[55-48]        y[6] = mm2/m64[55-48];
    x[7]   = mm1[63-56]        y[7] = mm2/m64[63-56];

    for (i = 0; i < 8; i++) {
        temp[i] = zero_ext(x[i], 8) + zero_ext(y[i], 8);
        res[i] = (temp[i] +1) >> 1;
        }
    mm1[7-0]      =   res[0];
    ...
    mm1[63-56]    =   res[7];
}
else if (instruction == PAVGW){
    x[0]   = mm1[15-0]         y[0] = mm2/m64[15-0];
    x[1]   = mm1[31-16]        y[1] = mm2/m64[31-16];
    x[2]   = mm1[47-32]        y[2] = mm2/m64[47-32];
    x[3]   = mm1[63-48]        y[3] = mm2/m64[63-48];


    for (i = 0; i < 4; i++) {
```

PAVGB/PAVGW: Packed Average (Continued)

```
        temp[i] = zero_ext(x[i], 16) + zero_ext(y[i], 16);

        res[i] = (temp[i] +1) >> 1;

    }

    mm1[15-0]     =  res[0];

    ...

    mm1[63-48]    =  res[3];

}
```

Description: The PAVG instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register, along with a carry-in. The results of the add are then each independently right shifted by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

The destination operand is a MMX technology register. The source operand can either be a MMX technology register or a 64-bit memory operand.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory references (if the current privilege level is 3).

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PEXTRW: Extract Word

Opcode	Instruction	Description
0F,C5, /r, ib	PEXTRW r32, mm, imm8	Extract the word pointed to by imm8 from MM and move it to a 32-bit integer register.

Operation:

```
sel = imm8 & 0x3;

mm_temp = (mm >> (sel * 16)) & 0xffff;

r[15-0] = mm_temp[15-0];

r[31-16] = 0x0000;
```

Description: The PEXTRW instruction moves the word in MM selected by the two least significant bits of imm8 to the lower half of a 32-bit integer register.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults Disabled FP Register Fault if PSR.dfl is 1

PINSRW: Insert Word

Opcode	Instruction	Description
0F,C4,/r,ib	PINSRW mm, r32/m16, imm8	Insert the word from the lower half of r32 or from Mem16 into the position in MM pointed to by imm8 without touching the other words.

Operation:

```
sel = imm8 & 0x3;

mask = (sel == 0)? 0x000000000000ffff :

       (sel == 1)? 0x00000000ffff0000 :

       (sel == 2)? 0x0000ffff00000000 :

                   0xffff000000000000;

mm = (mm & ~mask) | ((m16/r32[15-0] << (sel * 16)) & mask);
```

Description: The PINSRW instruction loads a word from the lower half of a 32-bit integer register (or from memory) and inserts it in the MM destination register at a position defined by the two least significant bits of the imm8 constant. The insertion is done in such a way that the three other words from the destination register are left untouched.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMAXSW: Packed Signed Integer Word Maximum

Opcode	Instruction	Description
0F,EE, /r	PMAXSW mm1, mm2/m64	Return the maximum words between MM2/Mem and MM1.

Operation:

```
mm1[15-0]   = (mm1[15-0] > mm2/m64[15-0]) ? mm1[15-0] : mm2/m64[15-0];

mm1[31-16]  = (mm1[31-16] > mm2/m64[31-16]) ? mm1[31-16] : mm2/m64[31-16];

mm1[47-32]  = (mm1[47-32] > mm2/m64[47-32]) ? mm1[47-32] : mm2/m64[47-32];

mm1[63-48]  = (mm1[63-48] > mm2/m64[63-48]) ? mm1[63-48] : mm2/m64[63-48];
```

Description: The PMAXSW instruction returns the maximum between the four signed words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMAXUB: Packed Unsigned Integer Byte Maximum

Opcode	Instruction	Description
0F,DE, /r	PMAXUB mm1, mm2/m64	Return the maximum bytes between MM2/Mem and MM1.

Operation:

```
mm1[7-0]    = (mm1[7-0] > mm2/m64[7-0]) ? mm1[7-0] : mm2/m64[7-0];

mm1[15-8]   = (mm1[15-8] > mm2/m64[15-8]) ? mm1[15-8] : mm2/m64[15-8];

mm1[23-16]  = (mm1[23-16] > mm2/m64[23-16]) ? mm1[23-16] : mm2/m64[23-16];

mm1[31-24]  = (mm1[31-24] > mm2/m64[31-24]) ? mm1[31-24] : mm2/m64[31-24];

mm1[39-32]  = (mm1[39-32] > mm2/m64[39-32]) ? mm1[39-32] : mm2/m64[39-32];

mm1[47-40]  = (mm1[47-40] > mm2/m64[47-40]) ? mm1[47-40] : mm2/m64[47-40];

mm1[55-48]  = (mm1[55-48] > mm2/m64[55-48]) ? mm1[55-48] : mm2/m64[55-48];

mm1[63-56]  = (mm1[63-56] > mm2/m64[63-56]) ? mm1[63-56] : mm2/m64[63-56];
```

Description: The PMAXUB instruction returns the maximum between the eight unsigned words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMINSW: Packed Signed Integer Word Minimum

Opcode	Instruction	Description
0F,EA, /r	PMINSW mm1, mm2/m64	Return the minimum words between MM2/Mem and MM1.

Operation:

```
mm1[15-0]   = (mm1[15-0] < mm2/m64[15-0]) ? mm1[15-0] : mm2/m64[15-0];
mm1[31-16]  = (mm1[31-16] < mm2/m64[31-16]) ? mm1[31-16] : mm2/m64[31-16];
mm1[47-32]  = (mm1[47-32] < mm2/m64[47-32]) ? mm1[47-32] : mm2/m64[47-32];
mm1[63-48]  = (mm1[63-48] < mm2/m64[63-48]) ? mm1[63-48] : mm2/m64[63-48];
```

Description: The PMINSW instruction returns the minimum between the four signed words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception#AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true (CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set. #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMINUB: Packed Unsigned Integer Byte Minimum

Opcode	Instruction	Description
0F,DA, /r	PMINUB mm1, mm2/m64	Return the minimum bytes between MM2/Mem and MM1.

Operation:

```
mm1[7-0]   = (mm1[7-0] < mm2/m64[7-0]) ? mm1[7-0] : mm2/m64[7-0];

mm1[15-8]  = (mm1[15-8] < mm2/m64[15-8]) ? mm1[15-8] : mm2/m64[15-8];

mm1[23-16]  = (mm1[23-16] < mm2/m64[23-16]) ? mm1[23-16] : mm2/m64[23-16];

mm1[31-24]  = (mm1[31-24] < mm2/m64[31-24]) ? mm1[31-24] : mm2/m64[31-24];

mm1[39-32]  = (mm1[39-32] < mm2/m64[39-32]) ? mm1[39-32] : mm2/m64[39-32];

mm1[47-40]  = (mm1[47-40] < mm2/m64[47-40]) ? mm1[47-40] : mm2/m64[47-40];

mm1[55-48]  = (mm1[55-48] < mm2/m64[55-48]) ? mm1[55-48] : mm2/m64[55-48];

mm1[63-56]  = (mm1[63-56] < mm2/m64[63-56]) ? mm1[63-56] : mm2/m64[63-56];
```

Description: The PMINUB instruction returns the minimum between the eight unsigned words in MM1 and MM2/Mem.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PMOVMSKB: Move Byte Mask To Integer

Opcode	Instruction	Description
0F,D7,/r	PMOVMSKB r32, mm	Move the byte mask of MM to r32.

Operation:

```
r32[7]  = mm[63];  r32[6]  = mm[55];

r32[5]  = mm[47];  r32[4]  = mm[39];

r32[3]  = mm[31];  r32[2]  = mm[23];

r32[1]  = mm[15];  r32[0]  = mm[7];

r32[31-8] = 0x000000;
```

Description: The PMOVMSKB instruction returns a 8-bit mask formed of the most significant bits of each byte of its source operand.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1

PMULHUW: Packed Multiply High Unsigned

Opcode	Instruction	Description
0F,E4,/r	PMULHUW mm1, mm2/m64	Multiply the packed unsigned words in MM1 register with the packed unsigned words in MM2/Mem, then store the high-order 16 bits of the results in MM1.

Operation:

```
mm1[15-0]    = (mm1[15-0]    * mm2/m64[15-0])[31-16];

mm1[31-16]   = (mm1[31-16]   * mm2/m64[31-16])[31-16];

mm1[47-32]   = (mm1[47-32]   * mm2/m64[47-32])[31-16];

mm1[63-48]   = (mm1[63-48]   * mm2/m64[63-48])[31-16];
```

Description: The PMULHUW instruction multiplies the four unsigned words in the destination operand with the four unsigned words in the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand.

Numeric Exceptions: None.

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PSADBW: Packed Sum of Absolute Differences

Opcode	Instruction	Description
0F,F6, /r	PSADBW mm1,mm2/m64	Absolute difference of packed unsigned bytes from MM2 /Mem and MM1; these differences are then summed to produce a word result.

Operation:

```
temp1   = ABS(mm1[7-0]      - mm2/m64[7-0]);

temp2   = ABS(mm1[15-8]     - mm2/m64[15-8]);

temp3   = ABS(mm1[23-16]    - mm2/m64[23-16]);

temp4   = ABS(mm1[31-24]    - mm2/m64[31-24]);

temp5   = ABS(mm1[39-32]    - mm2/m64[39-32]);

temp6   = ABS(mm1[47-40]    - mm2/m64[47-40]);

temp7   = ABS(mm1[55-48]    - mm2/m64[55-48]);

temp8   = ABS(mm1[63-56]    - mm2/m64[63-56]);


mm1[15:0] = temp1 + temp2 + temp3 + temp4 + temp5 + temp6 + temp7 + temp8;

mm1[31:16] = 0x00000000;

mm1[47:32] = 0x00000000;

mm1[63:48] = 0x00000000;
```

Description: The PSADBW instruction computes the absolute value of the difference of unsigned bytes for mm1 and mm2/m64. These differences are then summed to produce a word result in the lower 16-bit field; the upper 3 words are cleared.

The destination operand is a MMX technology register. The source operand can either be a MMX technology register or a 64-bit memory operand.

Numeric Exceptions: None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

PSADBW: Packed Sum of Absolute Differences (Continued)

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

PSHUFW: Packed Shuffle Word

Opcode	Instruction	Description
0F,70,/r,ib	PSHUFW mm1, mm2/m64, imm8	Shuffle the words in MM2/Mem based on the encoding in imm8 and store in MM1.

Operation:

```
mm1[15-0]   = (mm2/m64 >> (imm8[1-0] * 16) )[15-0]

mm1[31-16]  = (mm2/m64 >> (imm8[3-2] * 16) )[15-0]

mm1[47-32]  = (mm2/m64 >> (imm8[5-4] * 16) )[15-0]

mm1[63-48]  = (mm2/m64 >> (imm8[7-6] * 16) )[15-0]
```

Description: The PSHUF instruction uses the imm8 operand to select which of the four words in MM2/Mem will be placed in each of the words in MM1. Bits 1 and 0 of imm8 encode the source for destination word 0 (MM1[15-0]), bits 3 and 2 encode for word 1, bits 5 and 4 encode for word 2, and bits 7 and 6 encode for word 3 (MM1[63-48]). Similarly, the two bit encoding represents which source word is to be used, e.g. an binary encoding of 10 indicates that source word 2 (MM2/Mem[47-32]) will be used.

Numeric Exceptions: None.

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault

4.14 Cacheability Control Instructions

This section describes the cacheability control instructions which enable an application writer to minimize data access latency and cache pollution.

MASKMOVQ: Byte Mask Write

Opcode	Instruction	Description
0F,F7,/r	MASKMOVQ mm1, mm2	Move 64-bits representing integer data from MM1 register to memory location specified by the edi register, using the byte mask in MM2 register.

Operation:

```
if (mm2[7])   m64[edi]    = mm1[7-0];

if (mm2[15])  m64[edi+1]  = mm1[15-8];

if (mm2[23])  m64[edi+2]  = mm1[23-16];

if (mm2[31])  m64[edi+3]  = mm1[31-24];

if (mm2[39])  m64[edi+4]  = mm1[39-32];

if (mm2[47])  m64[edi+5]  = mm1[47-40];

if (mm2[55])  m64[edi+6]  = mm1[55-48];

if (mm2[63])  m64[edi+7]  = mm1[63-56];
```

Description: Data is stored from the mm1 register to the location specified by the di/edi register (using DS segment). The size of the store address depends on the address-size attribute. The most significant bit in each byte of the mask register mm2 is used to selectively write the data (0 = no write, 1 = write), on a per-byte basis. Behavior with a mask of all zeroes is as follows:

- No data will be written to memory. However, transition from FP to MMX technology state (if necessary) will occur, irrespective of the value of the mask.
- For memory references, a zero byte mask does not prevent addressing faults (i.e. #GP, #SS) from being signalled.
- Signalling of page faults (#PF) is implementation specific.
- #UD, #NM, #MF, and #AC faults are signalled irrespective of the value of the mask.
- Signalling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (i.e. is reserved) and is implementation specific. Dependency on the behavior of a specific implementation in this case is not recommended, and may lead to future incompatibility.

The Mod field of the ModR/M byte must be 11, or an Invalid Opcode Exception will result.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

MASKMOVQ: Byte Mask Write (Continued)

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1

Comments: MASKMOVQ can be used to improve performance for algorithms which need to merge data on a byte granularity.MASKMOVQ should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store. Similar to the SSE non-temporal store instructions, MASKMOVQ minimizes pollution of the cache hierarchy. MASKMOVQ implicitly uses weakly-ordered, write-combining stores (WC). See Section 4.6.1.9, "Cacheability Control Instructions" for further information about non-temporal stores.

As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation such as SFENCE should be used if multiple processors may use different memory types to read/write the same memory location specified by edi.

This instruction behaves identically to MMX technology instructions, in the presence of x87-FP instructions: transition from x87-FP to MMX technology (TOS=0, FP valid bits set to all valid).

MASMOVQ ignores the value of CR4.OSFXSR. Since it does not affect the new SSE state, they will not generate an invalid exception if CR4.OSFXSR = 0.

MOVNTPS: Move Aligned Four Packed Single-FP Non-temporal

Opcode	Instruction	Description
0F,2B, /r	MOVNTPS m128, xmm	Move 128 bits representing four packed SP FP data from XMM register to Mem, minimizing pollution in the cache hierarchy.

Operation: `m128 = xmm;`

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

FP Exceptions: General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #UD if CRCR4.OSFXSR(bit 9) = 0; #UD if CPUID.XMM(EDX bit 25) = 0.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVTNPS should be used when dealing with 16-byte aligned single-precision FP numbers. MOVNTPS minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation should be used if multiple processors may use different memory types to read/write the memory location. See Section 4.6.1.9, "Cacheability Control Instructions" for further information about non-temporal stores.

The usage of Repeat Prefixes(F2H, F3H) with MOVNTPS is reserved. Different processor implementations may handle this prefix differently. Usage of this prefix with MOVNTPS risks incompatibility with future processors.

MOVNTQ: Move 64 Bits Non-temporal

Opcode	Instruction	Description
0F,E7,/r	MOVNTQ m64, mm	Move 64 bits representing integer operands (8b, 16b, 32b) from MM register to memory, minimizing pollution within cache hierarchy.

Operation: `m64 = mm;`

Description: The linear address corresponds to the address of the least-significant byte of the referenced memory data. This store instruction minimizes cache pollution.

Numeric Exceptions: None

Protected Mode Exceptions:

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception; #AC for unaligned memory reference. To enable #AC exceptions, three conditions must be true(CR0.AM is set; EFLAGS.AC is set; current CPL is 3).

Real Address Mode Exceptions:

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; #UD if CR0.EM = 1; #NM if TS bit in CR0 is set; #MF if there is a pending FPU exception.

Virtual 8086 Mode Exceptions:

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3; #PF (fault-code) for a page fault.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	Disabled FP Register Fault if PSR.dfl is 1, NaT Register Consumption Fault
Itanium Mem Faults	VHPT Data Fault, Data TLB Fault, Alternate Data TLB Fault, Data Page Not Present Fault, Data NaT Page Consumption Abort, Data Key Miss Fault, Data Key Permission Fault, Data Access Rights Fault, Data Access Bit Fault, Data Dirty Bit Fault

Comments: MOVNTQ minimizes pollution in the cache hierarchy. As a consequence of the resulting weakly-ordered memory consistency model, a fencing operation should be used if multiple processors may use different memory types to read/write the memory location. See Section 4.6.1.9, "Cacheability Control Instructions" for further information about non-temporal stores.

MOVNTQ ignores the value of CR4.OSFXSR. Since it does not affect the new SSE state, they will not generate an invalid exception if CR4.OSFXSR = 0.

PREFETCH: Prefetch

Opcode	Instruction	Description
0F,18,/1	PREFETCHT0 m8	Move data specified by address closer to the processor using the t0 hint.
0F,18,/2	PREFETCHT1 m8	Move data specified by address closer to the processor using the t1 hint.
0F,18,/3	PREFETCHT2 m8	Move data specified by address closer to the processor using the t2 hint.
0F,18,/0	PREFETCHNTA m8	Move data specified by address closer to the processor using the nta hint.

Operation:

```
fetch (m8);
```

Description: If there are no excepting conditions, the prefetch instruction fetches the line containing the addresses byte to a location in the cache hierarchy specified by a locality hint. If the line is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. The bits 5:3 of the ModR/M byte specify locality hints as follows:

- Temporal data(t0) - prefetch data into all cache levels.
- Temporal with respect to first level cache (t1) – prefetch data in all cache levels except 0th cache level.
- Temporal with respect to second level cache (t2) – prefetch data in all cache levels, except 0th and 1st cache levels.
- Non-temporal with respect to all cache levels (nta) – prefetch data into non-temporal cache structure.

Locality hints do not affect the functional behavior of the program. They are implementation dependent, and can be overloaded or ignored by an implementation. The prefetch instruction does not cause any exceptions (except for code breakpoints), does not affect program behavior and may be ignored by the implementation. The amount of data prefetched is implementation dependent. It will however be a minimum of 32 bytes. Prefetches to uncacheable memory (UC or WC memory types) will be ignored. Additional ModRM encodings, besides those specified above, are defined to be reserved and the use of reserved encodings risks future incompatibility.

Numeric Exceptions: None

Protected Mode Exceptions: None

Real Address Mode Exceptions: None

Virtual 8086 Mode Exceptions: None

Additional Itanium System Environment Exceptions: None

Comments: This instruction is merely a hint.If executed, this instruction moves data closer to the processor in anticipation of future use. The performance of these instructions in application code can be implementation specific. To achieve maximum speedup, code tuning might be necessary for each implementation. The non temporal hint also minimizes pollution of useful cache data.

PREFETCH instructions ignore the value of CR4.OSFXSR. Since they do not affect the new SSE state, they will not generate an invalid exception if CR4.OSFXSR = 0.

SFENCE: Store Fence

Opcode	Instruction	Description
0F AE /7	SFENCE	Guarantees that every store instruction that precedes in program order the store fence instruction is globally visible before any store instruction which follows the fence is globally visible.

Operation:

```
while (!(preceding_stores_globally_visible)) wait();
```

Description: Weakly ordered memory types can enable higher performance through such techniques as out-of-order issue, write-combining, and write-collapsing. Memory ordering issues can arise between a producer and a consumer of data and there are a number of common usage models which may be affected by weakly ordered stores: (1) library functions, which use weakly ordered memory to write results (2) compiler-generated code, which also benefit from writing weakly-ordered results, and (3) hand-written code. The degree to which a consumer of data knows that the data is weakly ordered can vary for these cases. As a result, the SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

SFENCE uses the following ModRM encoding:

Mod (7:6) = 11B

Reg/Opcode (5:3) = 111B

R/M (2:0) = 000B

All other ModRM encodings are defined to be reserved, and use of these encodings risks incompatibility with future processors.

Numeric Exceptions: None

Protected Mode Exceptions: None

Real Address Mode Exceptions: None

Virtual 8086 Mode Exceptions: None

Additional Itanium System Environment Exceptions: None

Comments: SFENCE ignores the value of CR4.OSFXSR. SFENCE will not generate an invalid exception if CR4.OSFXSR = 0

Index

INDEX FOR VOLUMES 1, 2, 3 AND 4