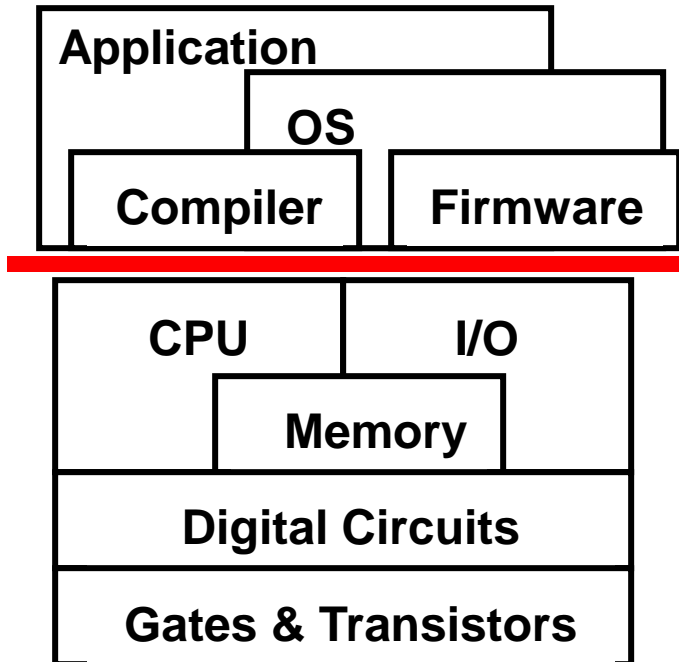


CSE 560  
Computer Systems Architecture

Instruction Set Architecture

# Instruction Set Architecture (ISA)

---



- What is an ISA?
  - A functional contract
- All ISAs similar at a high level
  - Design choices in the details
  - 2 “philosophies”: CISC/RISC
    - Difference is blurring
- Good ISA...
  - Enables high performance
- Importance of Compatibility
  - Tricks: binary translation,  $\mu$ ISAs

# ISWhat?

---

**Question:** How have I lived such a productive geek life without getting my hands on an ISA?

**Likely Answer:** Because you are young/started late enough not to have suffered through pre-compiler days.

**Snarky Answer:** Maybe you weren't as productive as you think you were. ;)

# ISA

---

How far are you willing to go for good performance?

**Ideally:** programmers need only write good programs

**In reality:** programmers want to see/consider the assembly to improve performance (compiler choices, etc.)

**In *some* realities:** designers consider ISA alterations to improve performance further

# **Big Picture (and Review)**

# Program Compilation

## Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

## Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

## Machine Code

```
0x401040
<sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    ...
```

compiler



assembler



**Program** written in high-level programming language (C, C++, Java, C#)

- Hierarchical, structured control: loops, functions, conditionals
- Hierarchical, structured data: scalars, arrays, pointers, structures

## Assembly language

- Human-readable representation of actual machine instructions

## Machine language

- Machine-readable representation of machine instructions
- 1s and 0s (often displayed in hex)

# Code Review

---

Which of the following statements is false?

- A. A compiler needs to be written with both the programming language and the target ISA in mind.
- B. You can determine the number of static instructions in a program by looking at the assembly code.
- C. A compiler can take assembled code for one ISA and prepare efficient object/machine code for a different ISA.
- D. It's easier to convert assembled code into object/machine code than it is to convert C code into assembly code.
- E. Compiler optimizations usually take place before the assembly code is produced.



# Code Review

---

Which of the following statements is false?

- A. A compiler needs to be written with both the programming language and the target ISA in mind.
- B. You can determine the number of static instructions in a program by looking at the assembly code.
- C. A compiler can take assembled code for one ISA and prepare efficient object/machine code for a different ISA.**
- D. It's easier to convert assembled code into object/machine code than it is to convert C code into assembly code.
- E. Compiler optimizations usually take place before the assembly code is produced.



# Reasoning About Performance

---

How long does it take for a program to execute?

## Three factors

1. How many instructions must execute to complete program
2. How fast is a single cycle
3. How many cycles does each instruction take to execute

$$\text{Execution Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}}$$

# Maximizing Performance

$$\text{Execution Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}}$$

## Instructions per program:

- Determined by program, compiler, instruction set architecture (ISA)

## Seconds per cycle: clock period

- Typical range today: 2ns to 0.25ns
- Reciprocal is frequency: 0.5 GHz to 4 GHz (1 Hz = 1 cycle per sec)
- Determined by micro-architecture, technology parameters

## Cycles per Instruction: CPI

- Typical range today: 2 to 0.5
- Determined by program, compiler, ISA, micro-architecture

Minimum execution time → **minimize each term**

- Difficult: *often pull against one another*

# What is an ISA?

# What Is An ISA?

---

## **ISA (instruction set architecture)**

- A well-defined hardware/software interface
- The “**contract**” between software and hardware
  - **Functional definition** of operations, modes, and storage locations supported by hardware
  - **Precise description** of how to invoke, access them
- Included in ISA:
  - Actual machine instructions (instruction set)
  - Storage interface (registers and memory)
  - Operating modes (user mode vs. supervisor mode)

# What Is Not In The ISA?

---

- **Not** in the “contract”: non-functional aspects
  - How operations are implemented
  - Which operations are fast, which are slow and when
  - Which operations take more/less power
  - How memory is implemented
  - Whether or not there is a cache

# ISA as Contract (1)

---

Which of the following **is** considered part of an ISA?

- A. Whether a multiply can operate on data still in memory.
- B. Whether branch prediction is used.
- C. The number of cycles it takes to execute a multiply.
- D. The number of physical registers the machine has.
- E. Whether instructions can be executed out of order.



# ISA as Contract (1)

---

Which of the following **is** considered part of an ISA?

- A. Whether a multiply can operate on data still in memory.**
- B. Whether branch prediction is used.
- C. The number of cycles it takes to execute a multiply.
- D. The number of physical registers the machine has.
- E. Whether instructions can be executed out of order.

# ISA Design Goals



# What Makes a Good ISA?

---

## 1. Programmability

- Easy to express programs efficiently?

## 2. Implementability

- Easy to design high-performance implementations?
- More recently: low-power, high-reliability, low-cost

## 3. Compatibility

- Easy to maintain programmability as languages and programs evolve?
- x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2...

# 1. Programmability

---

- Easy to express programs efficiently? For whom?
- Before 1985: **human**
  - Compilers were terrible, often code hand-assembled
  - Want high-level coarse-grain instructions
    - As similar to high-level language as possible
- After 1985: **compiler**
  - Optimizing compilers generate better code than we do
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly

# Human Programmability

---

- What makes an ISA easy for a human to program in?
  - Proximity to a high-level language (HLL)
    - Closing the “**semantic gap**”
  - Semantically heavy (CISC-like) insns that capture complete idioms
    - “Access array element”, “loop”, “procedure call”
    - Example: SPARC **save/restore**
    - Bad example: x86 **rep movsb** (copy string)
    - Ridiculous example: VAX **insque** (insert-into-queue)
  - “**Semantic clash**”: what if you have many high-level languages?
- Stranger than fiction
  - People once thought computers would execute language directly
  - Fortunately, never materialized (but keeps coming back around)
    - Now we just ask LLM to write the code for us

# Today's Semantic Gap

---

- Today's ISAs are actually targeted to one language...
- ...Just so happens that this language is very low level
  - **The C programming language**
- Will ISAs be different when Java/C# become dominant?
  - Object-oriented? **Probably not**
  - Support for garbage collection? **Maybe**
  - Support for bounds-checking? **Maybe**
  - **Why?**
    - Smart compilers transform HLL to simple instructions
    - Any benefit of tailored ISA is likely small

# Compiler Optimizations (1)

---

Compilers do two things

- **Code generation**
  - Translate HLL to machine insns, 1 statement at a time
- **Optimization**
  - Preserve meaning but improve performance
  - Active research area, but some standard optimizations
    - Register allocation, common sub-expression elimination, loop-invariant code motion, loop unrolling, function inlining, code scheduling (to increase insn-level parallelism), *etc.*

# Compiler Optimizations (2)

---

- Reduce dynamic insn count primarily
  - Redundant computation, store things in registers
    - + Registers are faster, fewer loads/stores
    - ISA can make this difficult by having too few registers
- Also reduce:
  - Branches and jumps
  - Cache misses
  - Dependences between nearby insns (for parallelism)
    - ISA can make this difficult by having implicit dependences
- How effective are these?
  - + Can give 4X performance over unoptimized code
  - Collective wisdom of 40 years (“Proebsting’s Law”: *Compiler Advances Double Computing Power Every 18 Years*)
  - Funny but ... don’t laugh at 4X performance

# ISA as Contract (2)

---

Which of the following **is not** considered part of an ISA?

- A. The number of architected registers a machine has.
- B. Whether the compiler can inject prefetch instructions.
- C. Whether the machine supports vector operations.
- D. Whether a data is stored in the cache after it is fetched from memory.
- E. The number of bits it takes to encode each instruction.



# ISA as Contract (2)

---

Which of the following **is not** considered part of an ISA?

- A. The number of architected registers a machine has.
- B. Whether the compiler can inject prefetch instructions.
- C. Whether the machine supports vector operations.
- D. Whether a data is stored in the cache after it is fetched from memory.**
- E. The number of bits it takes to encode each instruction.



## 2. Implementability

---

- **Every ISA can be implemented**
  - Not every ISA can be implemented efficiently
- **Classic high-performance implementation techniques**
  - Pipelining, parallel execution, out-of-order execution
- **Certain ISA features make these difficult**
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicate many things
    - Example: memory copy instruction

# 3. Compatibility

---

- In many domains, ISA must remain compatible
  - IBM's 360/370 (the *first* "ISA family")
  - Another example: Intel's x86 and Microsoft Windows
    - x86 one of the worst designed ISAs **EVER**, but survives
- **Backward compatibility**
  - New processors supporting old programs
    - Can't drop features (cumbersome)
    - Or, update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs
    - Include a "CPU ID" so the software can test for features
    - Add ISA hints by overloading no-ops (example: x86's PAUSE)
    - New firmware/software on old processors to emulate new insn

# The Compatibility Trap

---

- Easy compatibility requires forethought
  - Temptation: some ISA extension gives 5% perf. gain
  - Often: gain diminishes, disappears, or turns to loss
    - **Must continue to support gadget for eternity**
  - Example: register windows (SPARC) makes OoO difficult
- Compatibility trap door
  - How to rid yourself of some ISA mistake in the past?
  - Make old insns an “illegal” insn on new machine
  - OS handles exception, emulates instruction, returns
    - Slow unless extremely uncommon

# Aspects of ISAs