Chapter 2: Machines, Machine Languages, and Digital Logic

C S D A 2/e

Instruction sets, SRC, RTN, and the mapping of register transfers to digital logic circuits

Chapter 2 Topics

- **2.1 Classification of Computers and Instructions**

- **2.2 Kinds and Classes of Instruction Sets**

- **2.3 Informal Description of the Simple RISC Computer, SRC**

  - Students may wish to consult Appendix C, Assembly and Assemblers for information about assemblers and assembly.

- **2.4 Formal Description of SRC using Register Transfer Notation (RTN)**

- **2.5 RTN Description of Addressing Modes**

- **2.6 Register Transfers and Logic Circuits: from Behavior to Hardware**

  - Students may wish to consult Appendix A, Digital Logic for additional information about Digital Logic circuits.

What are the components of an ISA?

- Sometimes known as *The Programmers Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The Machine Instruction Set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i. e. Register Transfers
- The Instruction Format
  - Size and meaning of fields within the instruction
- The nature of the Fetch/Execute cycle
  - Things that are done before the operation code is known

Fig. 2.1 Programmer's Models of Various Machines

We saw in Chap. 1 a variation in number and type of storage cells

M6800 (1975)	Pentium 4 (2000)	I8086 (1979)	VAX11 (1981)	PPC G4 (1999)
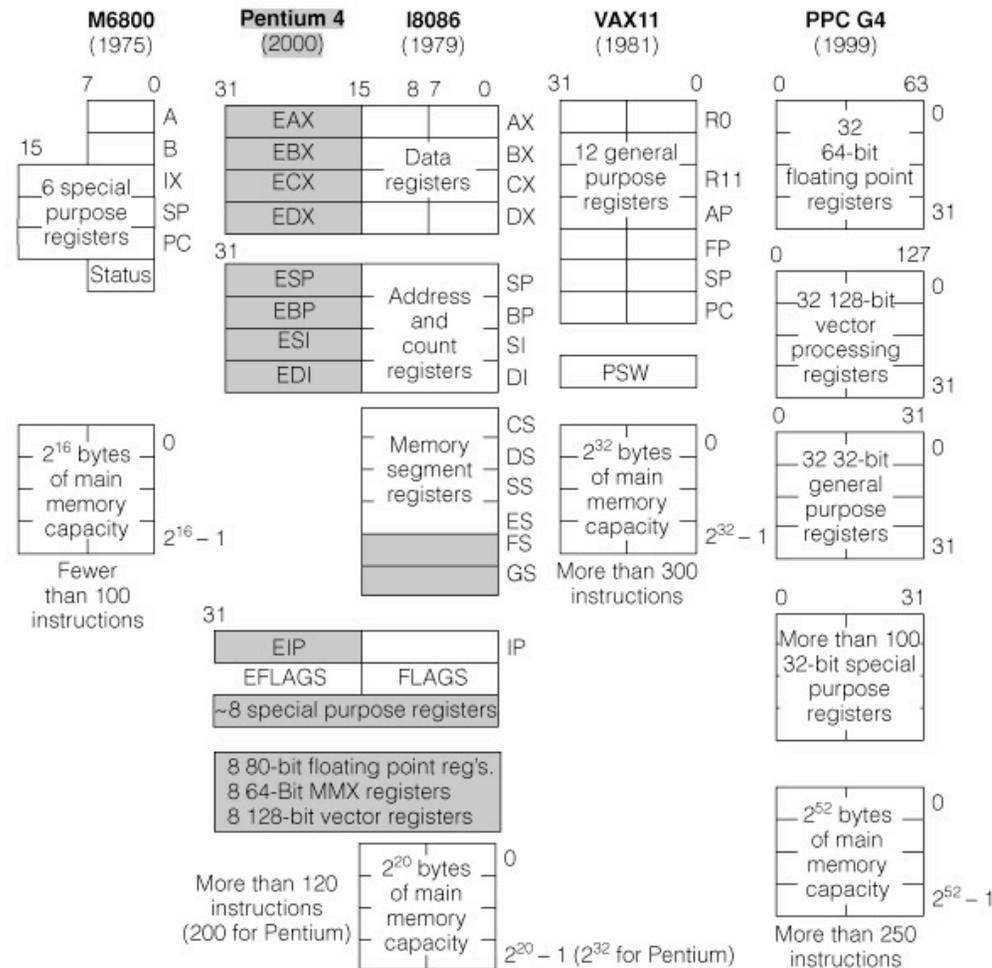
Copyright © 2004 Pearson Prentice Hall, Inc.

What Must an Instruction Specify?

Data Flow

←

- Which operation to perform: add r0, r1, r3
  - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands   add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result   add r0, r1, r3
  - Again CPU register or memory cell
- Location of next instruction   add r0, r1, r3
                                            br endloop

- The default is usually memory cell pointed to by program counter—PC: the next instruction in sequence

- Sometimes there *is* no operand, or no result, or no next instruction. Can you think of examples?
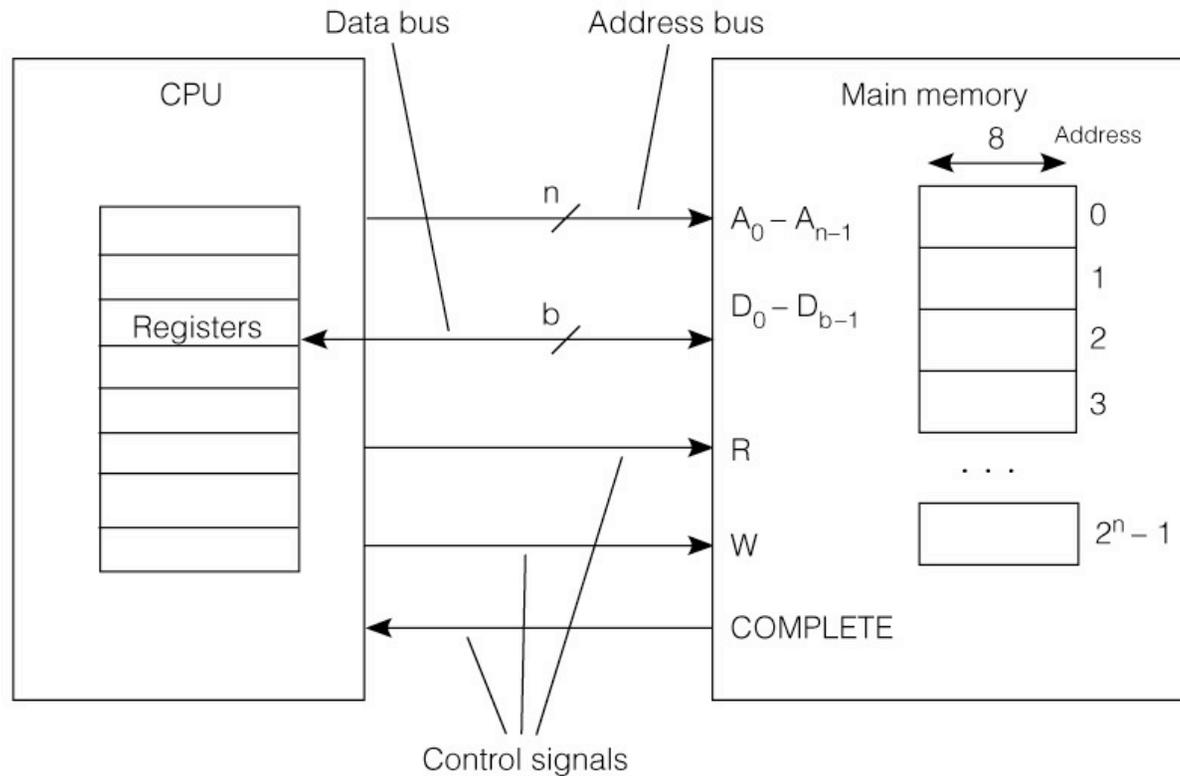
Fig. 2.2 Accessing Memory—Reading from Memory

For a Memory Read:

CPU applies desired address to Address lines A_0-A_{n-1}

CPU issues Read command, R

Memory returns the value at that address on Data lines D_0-D_{b-1} and asserts the COMPLETE signal

Figure 2.2 Accessing Memory—Writing to Memory

Copyright © 2004 Pearson Prentice Hall, Inc.
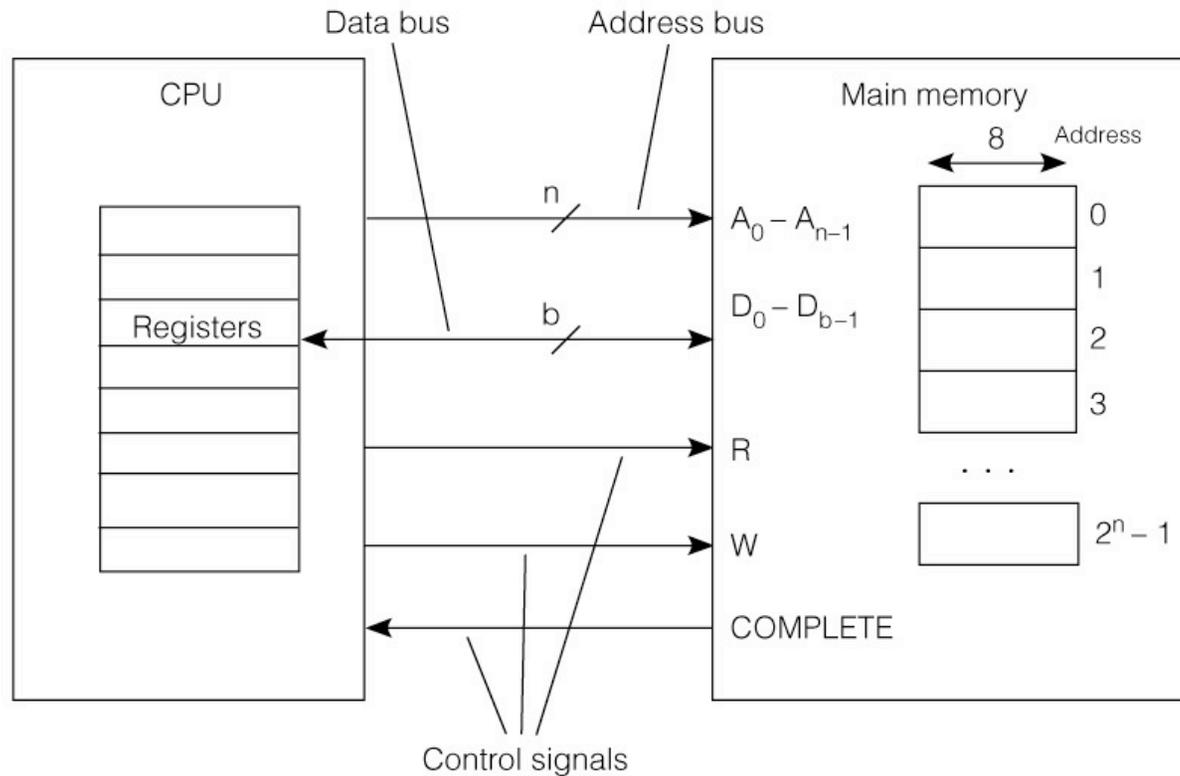
For a Memory Write:

CPU applies desired address to Address lines A_0-A_{n-1} and and data to be written on Data lines D_0-D_{b-1}

CPU issues Write command, W

Memory asserts the COMPLETE signal when the data has been written to memory.

Instructions Can Be Divided into 3 Classes

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - Load—source is memory and destination is register
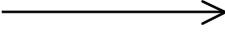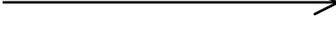 - Store—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Changes the form of one or more operands to produce a result stored in another location
  - Add, Sub, Shift, etc.
- Branch instructions (control flow instructions)
  - Any instruction that alters the normal flow of control from executing the next instruction in sequence
  - Br Loc, Brz Loc2,—unconditional or conditional branches

Tbl. 2.1 Examples of Data Movement Instructions

Instruct.	Meaning	Machine
MOV A, B	Move 16 bits from mem. loc. A to loc. B	VAX11
lwz R3, A	Move 32 bits from mem. loc. A to reg. R3	PPC601
li $3, 455	Load the 32 bit integer 455 into reg. 3	MIPS R3000
mov R4, dout	Move 16 bits from R4 to out port dout	DEC PDP11
IN, AL, KBD	Load a byte from in port KBD to accum.	Intel Pentium
LEA.L (A0), A2	Load address pointed to by A0 into A2	M68000

- Lots of variation, even with one instruction type
- Notice differences in direction of data flow left-to-right or right-to-left

Tbl 2.2 Examples of ALU (Arithmetic and Logic Unit) Instructions

Instruction	Meaning	Machine
MULF A, B, C	**multiply the 32-bit floating point values at mem loc'ns. A and B, store at C**	**VAX11**
nabs r3, r1	**Store abs value of r1 in r3**	**PPC601**
ori $2, $1, 255	**Store logical OR of reg $ 1 with 255 into reg $2**	**MIPS R3000**
DEC R2	**Decrement the 16-bit value stored in reg R2**	**DEC PDP11**
SHL AX, 4	**Shift the 16-bit value in reg AX left by 4 bits**	**Intel 8086**

▪**Notice again the complete dissimilarity of both syntax and semantics**

Tbl 2.3 Examples of Branch Instructions

Instruction	**Meaning**	**Machine**
BLSS A, Tgt	**Branch to address Tgt if the least significant bit of mem loc'n. A is set (i.e. = 1)**	**VAX11**
bun r2	**Branch to location in R2 if result of previous floating point computation was Not a Number (NAN)**	**PPC601**
beq $2, $1, 32	**Branch to location (PC + 4 + 32) if contents of $1 and $2 are equal**	**MIPS R3000**
SOB R4, Loop	**Decrement R4 and branch to Loop if R4 $\neq$ 0**	**DEC PDP11**
JCXZ Addr	**Jump to Addr if contents of register CX = 0.**	**Intel 8086**

CPU Registers Associated with Flow of Control—Branch Insts.

- Program counter usually contains the address of, or "points to" the next instruction
- Condition codes may control branch
- Branch targets may be contained in separate registers

Processor State

Program Counter	**Branch Targets**	**C N V Z**
		Condition Codes
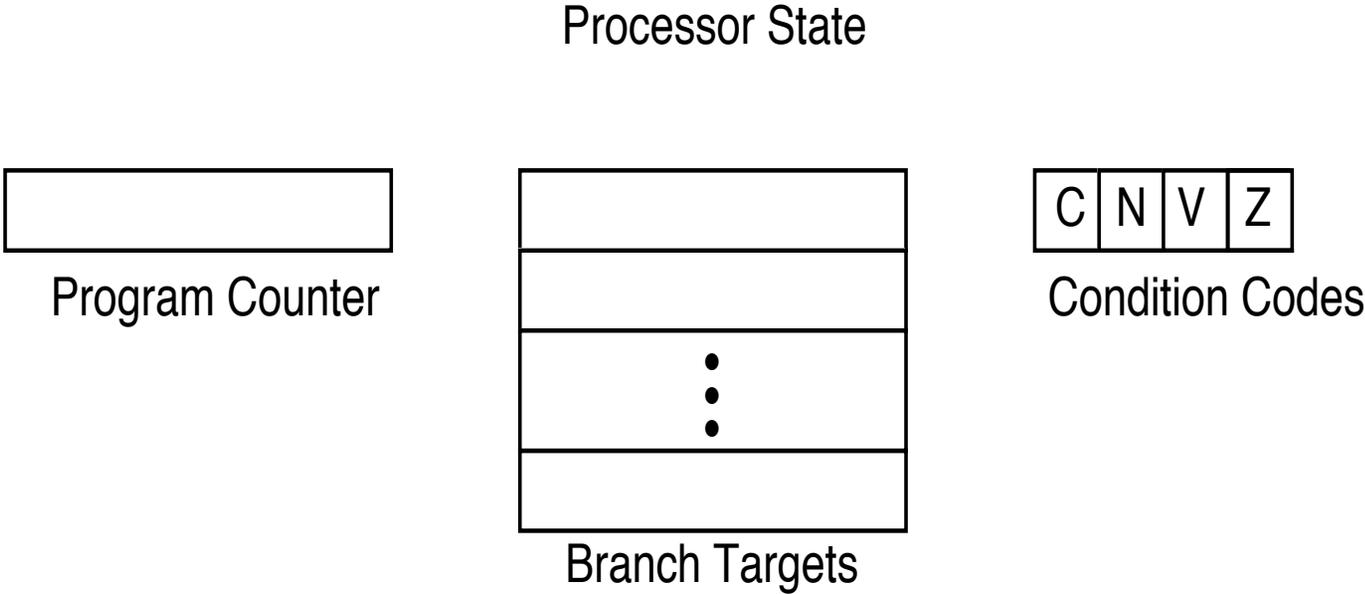
HLL Conditionals Implemented by Control Flow Change

- Conditions are computed by arithmetic instructions
- Program counter is changed to execute only instructions associated with true conditions

C language

Assembly language

```
                   CMP.W  #5, NUM    ;the comparison
                   BNE    L1         ;conditional branch
                   MOV.W  #7, SET    ;action if true
if NUM==5 then SET=7
           L1      ...               ;action if false
```

CPU Registers may have a "personality"

- Architecture classes are often based on how where the operands and result are located and how they are specified by the instruction.

- They can be in CPU registers or main memory

| Stack | Arithmetic Registers | Address Registers | General Purpose Registers |

Push   Pop

Top
Second

⋮
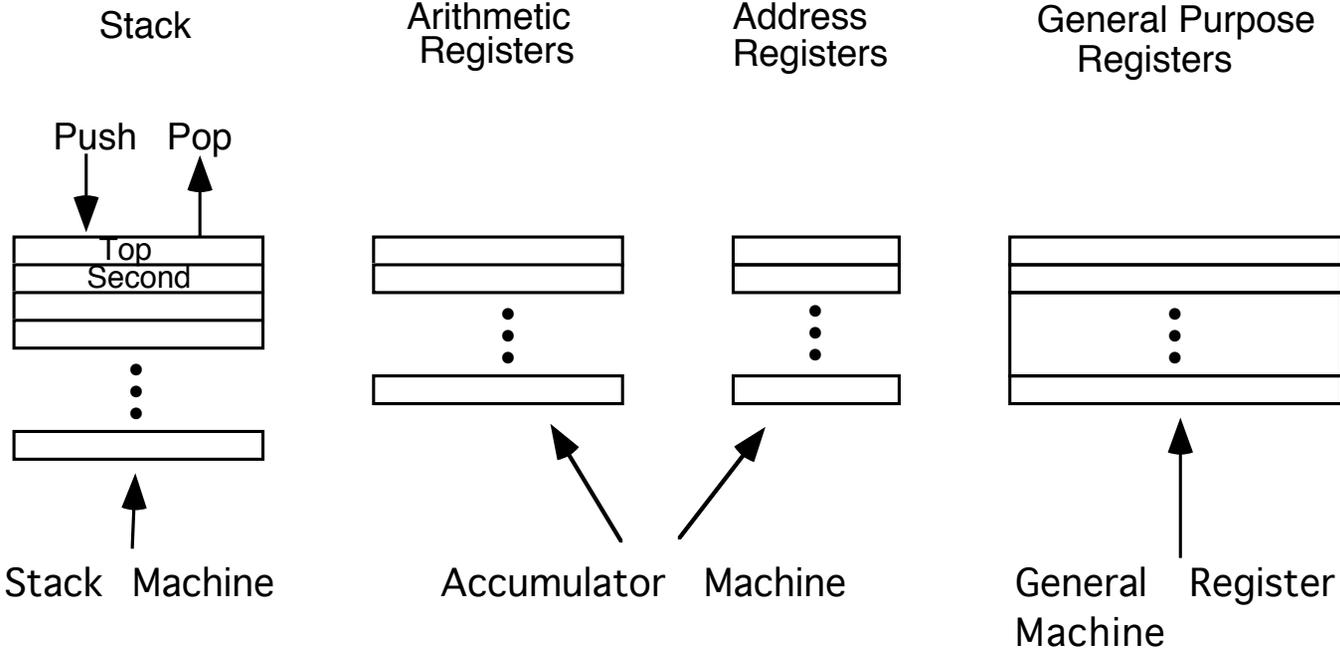
Stack   Machine   Accumulator   Machine   General   Register
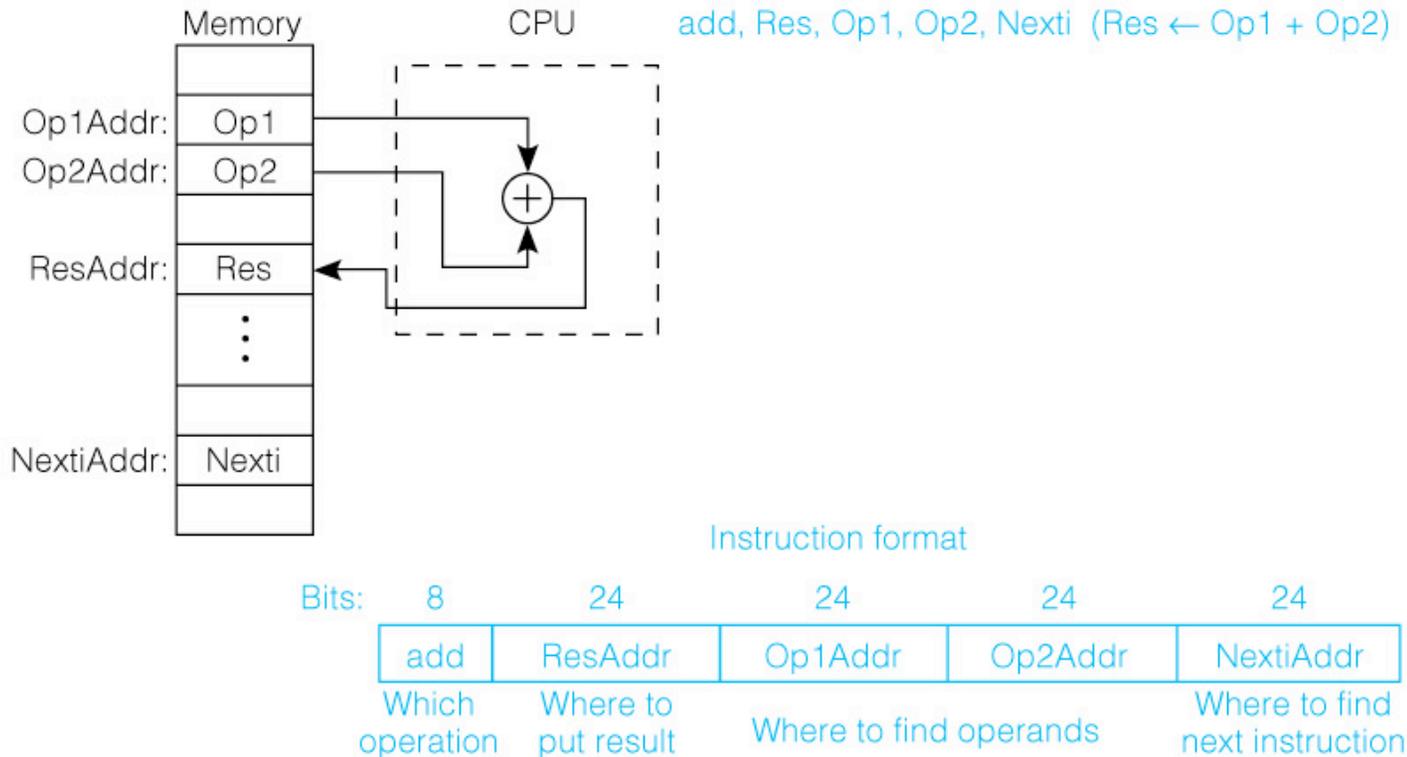   Machine

3, 2, 1, & 0 Address Instructions

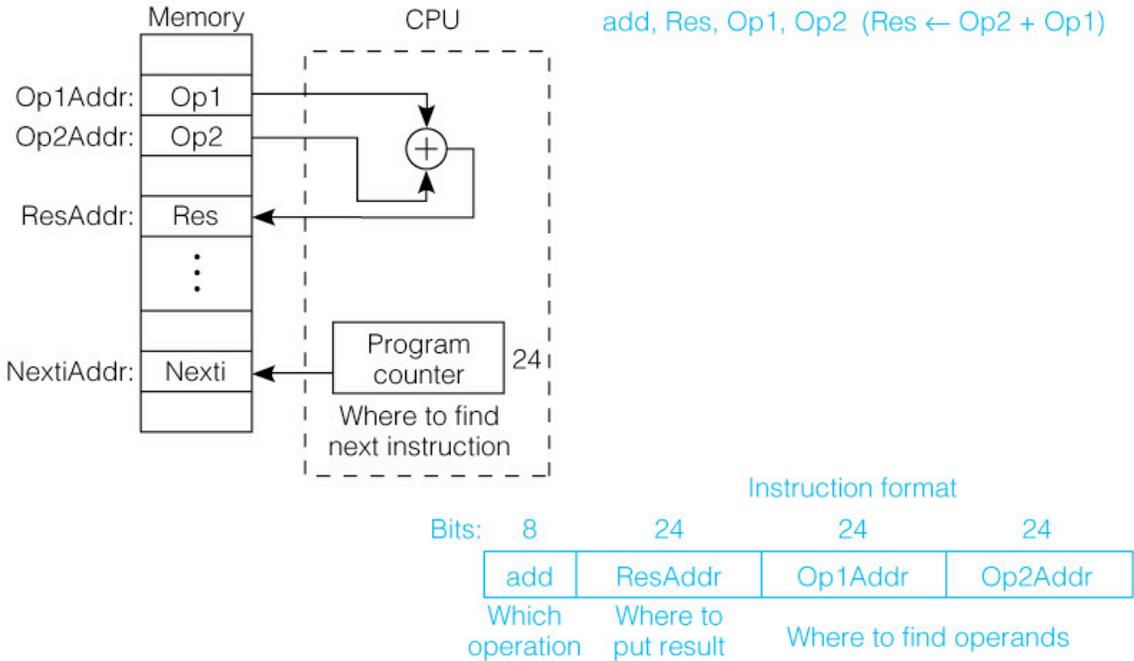- The classification is based on arithmetic instructions that have two operands and one result

- The key issue is "how many of these are specified by memory addresses, as opposed to being specified implicitly"

- A 3 address instruction specifies memory addresses for both operands and the result: R ← Op1 op Op2

- A 2 address instruction overwrites one operand in memory with the result: Op2 ← Op1 op Op2

- A 1 address instruction has a register, called the accumulator register to hold one operand & the result (no address needed):
Acc ← Acc op Op1

- A 0 address + uses a CPU register stack to hold both operands and the result: TOS ← TOS op SOS where TOS is Top Of Stack, SOS is Second On Stack)

- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly.

Fig. 2.3 The 4 Address Instruction

Memory   CPU   add, Res, Op1, Op2, Nexti (Res ← Op1 + Op2)

Op1Addr: Op1
Op2Addr: Op2

⊕

ResAddr: Res

NextiAddr: Nexti

Instruction format

Bits:	8	24	24	24	24
	add	ResAddr	Op1Addr	Op2Addr	NextiAddr
	Which operation	Where to put result	Where to find operands		Where to find next instruction

- Explicit addresses for operands, result & next instruction
- Example assumes 24-bit addresses
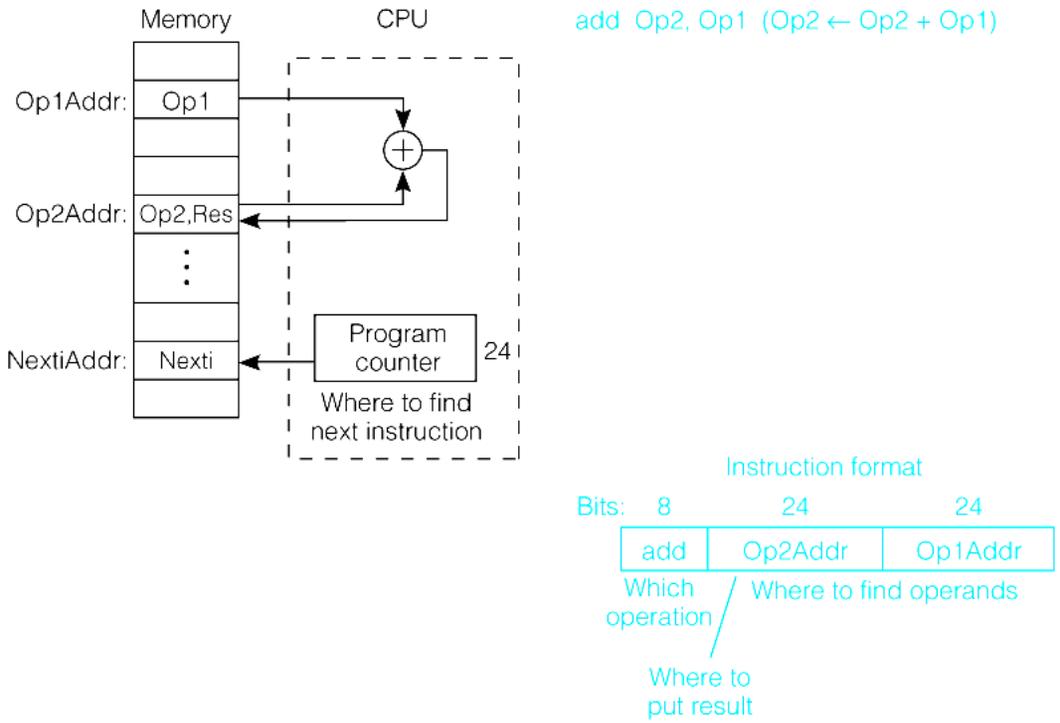 - Discuss: size of instruction in bytes

Fig 2.4 The 3 Address Instruction


```
Memory          CPU          add, Res, Op1, Op2  (Res ← Op2 + Op1)

Op1Addr:  Op1
Op2Addr:  Op2                    (+)
ResAddr:  Res

                 Program counter  24
NextiAddr: Nexti
                 Where to find
                 next instruction

                          Instruction format
              Bits:   8        24         24          24
                     add    ResAddr    Op1Addr     Op2Addr
                    Which    Where to
                  operation  put result  Where to find operands
```

Copyright © 2004 Pearson Prentice Hall, Inc.

- Address of next instruction kept in a processor state register—the PC (Except for explicit Branches/Jumps)
- Rest of addresses in instruction
  - Discuss: savings in instruction word size
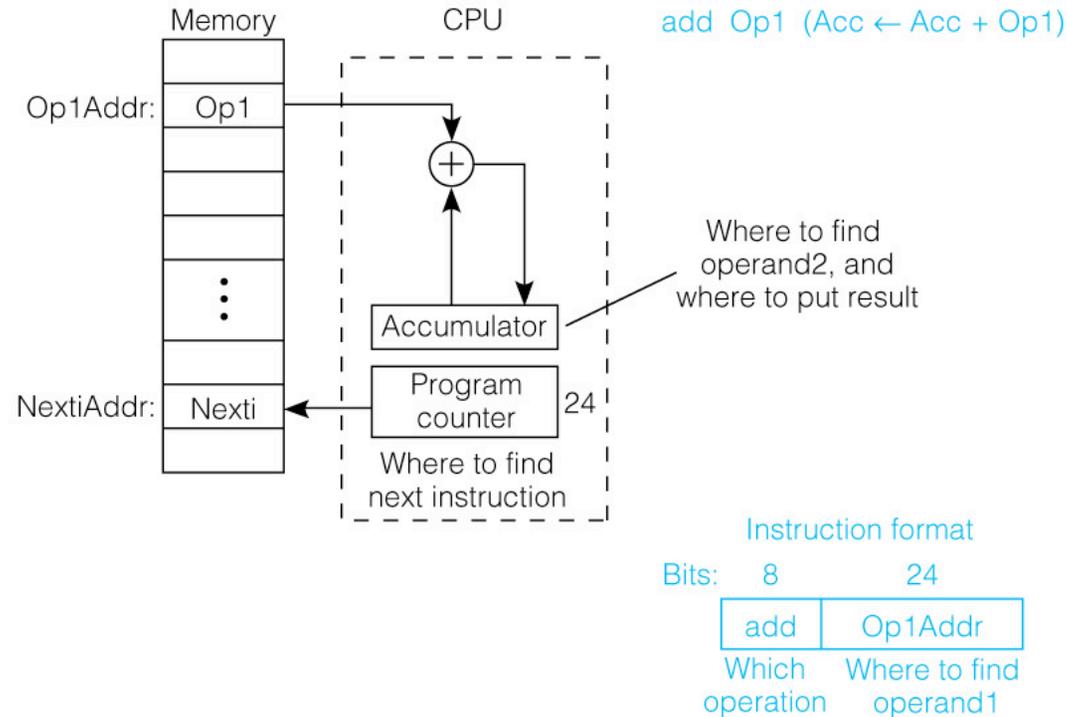
Fig. 2.5 The 2 Address Instruction


```
                Memory          CPU        add  Op2, Op1  (Op2 ← Op2 + Op1)

Op1Addr:    Op1                  (+)
                                  |
Op2Addr:   Op2,Res               |

              :
              :

NextiAddr:   Nexti         Program    24
                           counter
                        Where to find
                        next instruction

                              Instruction format
                   Bits:    8        24         24
                          add    Op2Addr    Op1Addr
                         Which  /  Where to find operands
                       operation/
                            /
                        Where to
                        put result
```

Copyright © 2004 Pearson Prentice Hall, Inc.

- Be aware of the difference between address, Op1Addr, and data stored at that address, Op1.
- Result overwrites Operand 2, Op2, with result, Res
- This format needs only 2 addresses in the instruction but there is less choice in placing data
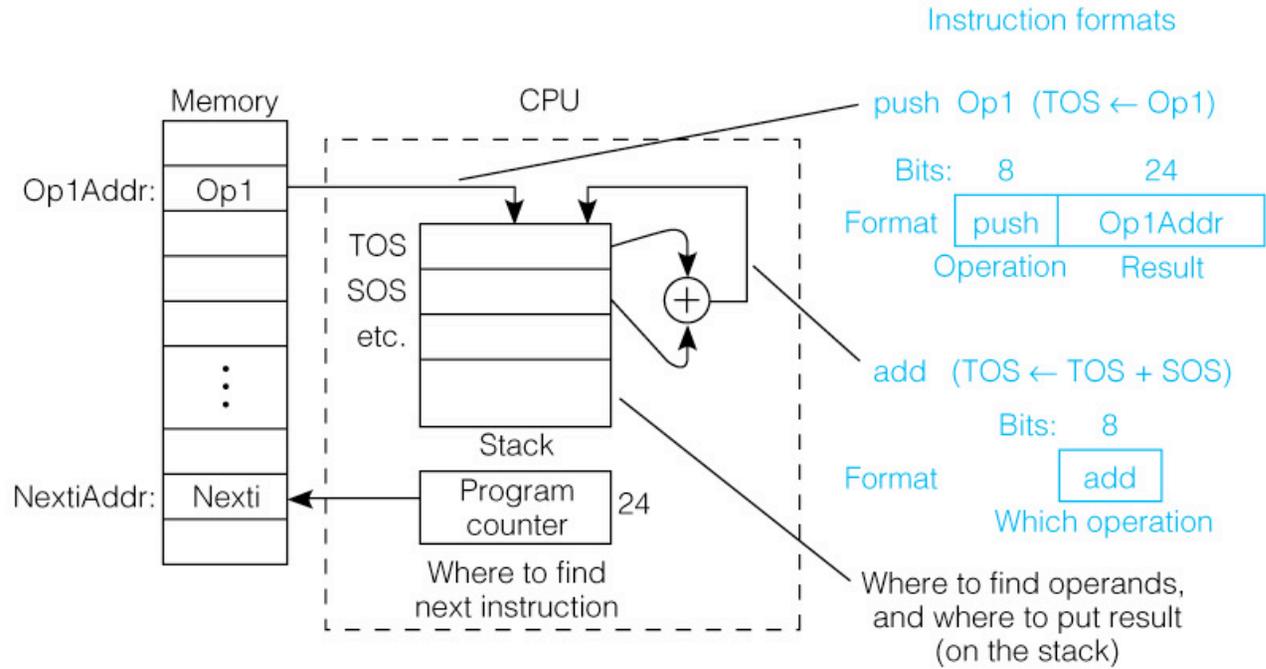
Fig. 2.6 1 Address Instructions

We now need instructions to load and store operands:
LDA OpAddr
STA OpAddr

Memory

CPU

add Op1 (Acc ← Acc + Op1)

Op1Addr: Op1

$+$

Accumulator

Where to find operand2, and where to put result

NextiAddr: Nexti

Program counter   24

Where to find next instruction

Instruction format

Bits: 8   24

add   Op1Addr

Which operation   Where to find operand1

Copyright © 2004 Pearson Prentice Hall, Inc.

- Special CPU register, the accumulator, supplies 1 operand and stores result
- One memory address used for other operand

Fig. 2.7 The 0 Address Instruction

Instruction formats

Memory

Op1Addr: | Op1

NextiAddr: | Nexti

CPU

TOS
SOS
etc.

Stack

Program counter | 24

Where to find next instruction

push Op1 (TOS ← Op1)

Bits: 8   24

Format | push | Op1Addr

Operation   Result

add (TOS ← TOS + SOS)

Bits: 8

Format | add

Which operation

Where to find operands, and where to put result (on the stack)

Copyright © 2004 Pearson Prentice Hall, Inc.

- Uses a push down stack in CPU

- Arithmetic uses stack for both operands. The result replaces them on the TOS

- Computer must have a 1 address instruction to push and pop operands to and from the stack

Example 2.1 Expression evaluation for 3-0 address instructions.

Evaluate a = (b+c)*d-e for 3- 2- 1- and 0-address machines.

3-Address	2-Address	Accumulator	Stack
add a,b,c	load a,b	lda b	push b
mpy a,a,d	add a,c	add c	push c
sub a,a,e	mpy a,d	mpy d	add
	sub a,e	sub e	push d
		sta a	mpy
			push e
			sub
			pop a

- # of instructions & # of addresses both vary
- Discuss as examples: size of code in each case

Fig. 2.8 General Register Machines

CPU

Memory

Registers

Op1Addr: Op1

load → R8

R6

(+) → R4

R2

Nexti ← Program counter

Instruction formats

load R8, Op1 (R8 ← Op1)

load	R8	Op1Addr

add R2, R4, R6 (R2 ← R4 + R6)

add	R2	R4	R6
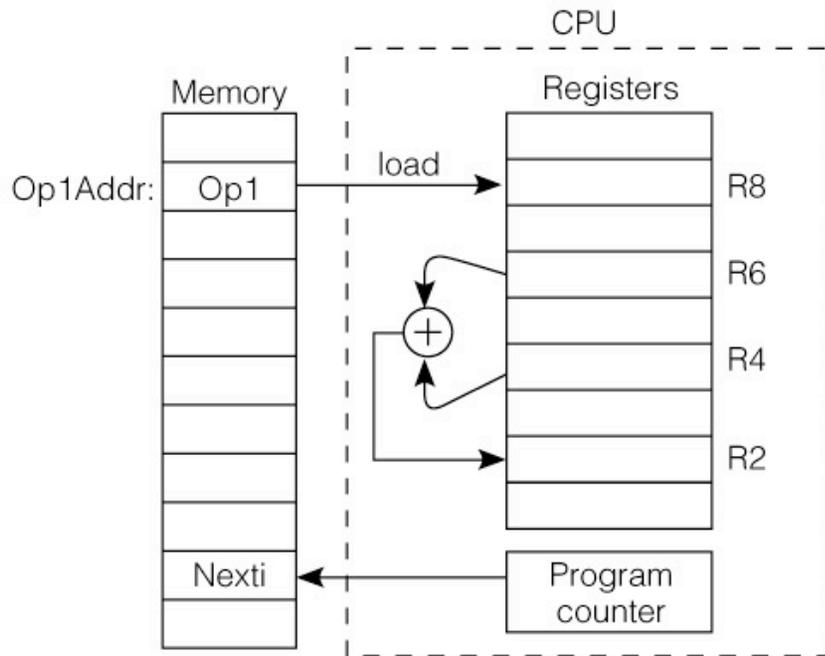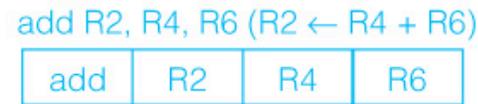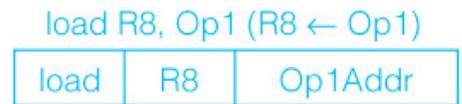
Copyright © 2004 Pearson Prentice Hall, Inc.

- It is the most common choice in today's general purpose computers
- *Which* register is specified by small "address" (3 to 6 bits for 8 to 64 registers)
- Load and store have one long & one short address: 1 1/2 addresses
- 2-Operand arithmetic instruction has 3 "half" addresses

Real Machines are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, 1 1/2 address instructions

- A distinction can be made on whether arithmetic instructions use data from memory

- If ALU instructions only use registers for operands and result, machine type is load-store

  - Only load and store instructions reference memory

- Other machines have a mix of register-memory and memory-memory instructions

Addressing Modes

- An addressing mode is hardware support for a useful way of determining a memory address

- Different addressing modes solve different HLL problems

  - Some addresses may be known at compile time, e.g. global vars.

  - Others may not be known until run time, e.g. pointers

  - Addresses may have to be *computed*: Examples include:

    - Record (struct) components:
      - variable base(full address) + const.(small)
    - Array components:
      - const. base(full address) + index var.(small)

  - Possible to store constant values w/o using another memory cell by storing them with or adjacent to the instruction itself.

HLL Examples of Structured Addresses

- C language: rec -> count
  - rec is a pointer to a record: full address variable
  - count is a field name: fixed byte offset, say 24
- C language: v[i]
  - v is fixed base address of array: full address constant
  - i is name of variable index: no larger than array size
- Variables must be contained in registers or memory cells
- Small constants can be contained in the instruction
- Result: need for "address arithmetic."
  - E.g. Address of Rec -> Count is address of Rec + offset of count.

Count

Rec →

V[i]

V →
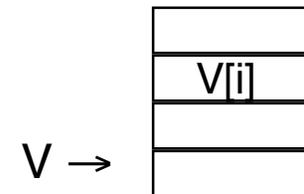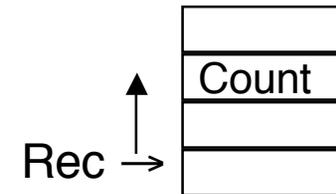
Fig 2.9 Common Addressing Modes a-d

(a) Immediate addressing:

instruction contains
the operand

Instr | Op | 3

load #3, ...

(b) Direct addressing:

instruction contains
address of operand

Instr | Op | Addr of A

load A, ...

Memory
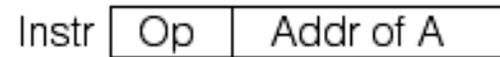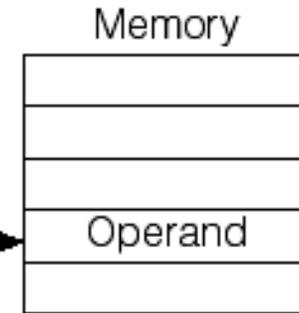
Operand

(c) Indirect addressing:

instruction contains
address of address
of operand

Address of address of A

Instr | Op

load (A), ...

Memory
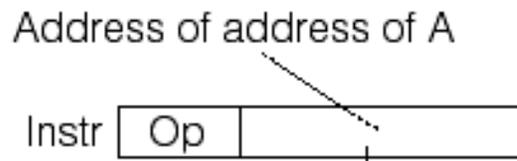
Operand
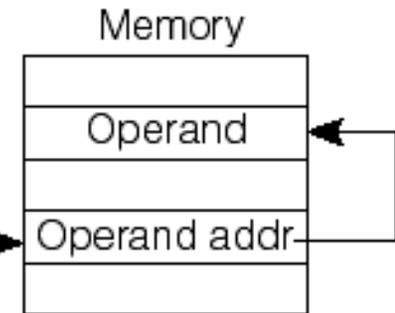
Operand addr

(d) Register direct addressing:

register contains operand

Instr | Op | R1 | . . .

load R1, ...
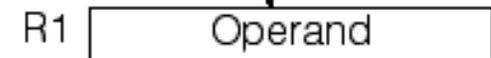
R1 | Operand

Fig 2.9 Common Addressing Modes e-g

(e) Register indirect addressing:

register contains address
of operand

```
Instr  Op | R2 | . . .
              |
              ▼
R2  | Operand addr |
                 └──────────►  Operand
```

Memory

load [R2], ...
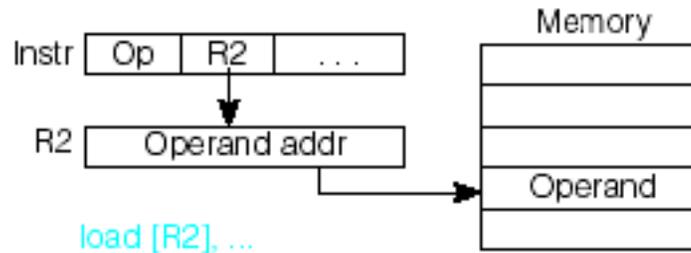
(f) Displacement (based or indexed) addressing:

address of operand =
register + constant

```
Instr  Op | R2 | 4 |
              |    |
              ▼    ▼
          R2 |____| ⊕ ──────► Operand
```

Memory

load 4[R2], ...   Operand address
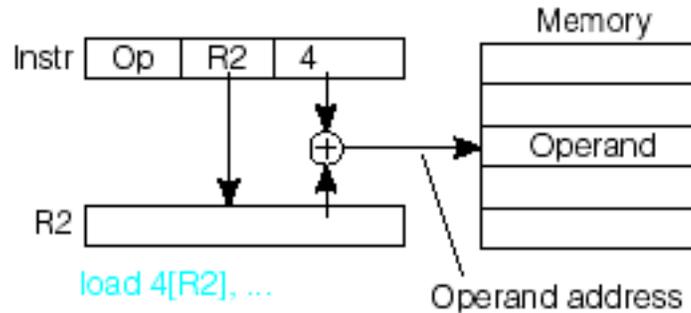
(g) Relative addressing:

address of operand =
PC + constant

```
Instr  Op |      4      |
                 |
                 ▼
          PC |_____| ⊕ ──────► Operand
```

Memory

loadrel 4[PC], ...   Operand address
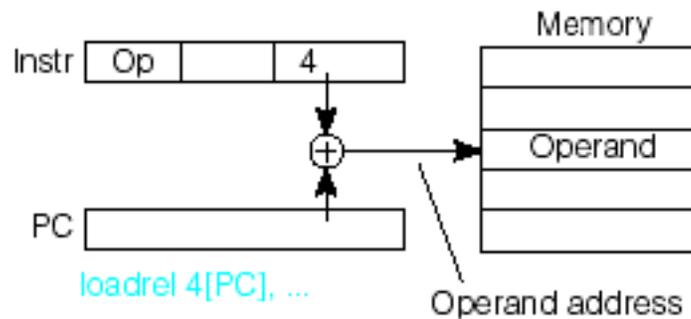
Fig. 2.10a Example Computer, SRC Simple RISC Computer

- 32 general purpose registers of 32 bits
- 32 bit program counter, PC and instruction reg., IR
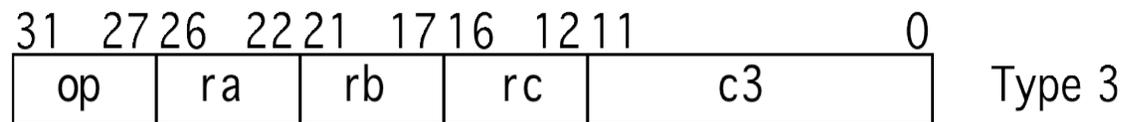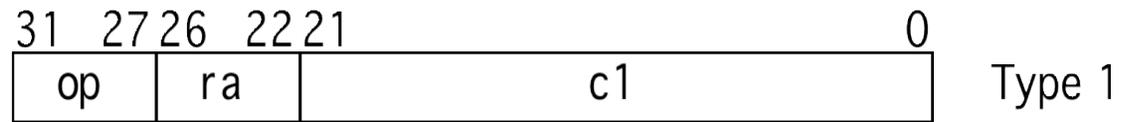- 2^{32} bytes of memory address space

The SRC CPU

```
        31              0
R0   ┌────────────────────┐
     │    32 32-bit       │
     │    general         │
     │    purpose         │
     │    registers       │
R31  └────────────────────┘

PC   ┌────────────────────┐
     └────────────────────┘
IR   ┌────────────────────┐
     └────────────────────┘
```

Main memory

```
 7        0
┌──────────┐  0
│          │
│   2^32   │  •
│  bytes   │  •
│    of    │  •
│  main    │
│ memory   │
│          │
│          │  2^32 − 1
└──────────┘
```

R[7] means contents of register 7

M[32] means contents of memory location 32
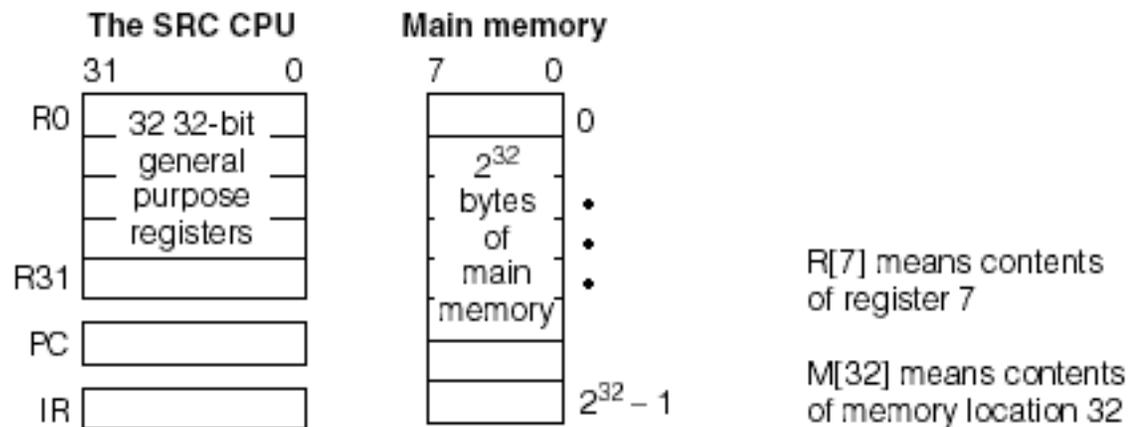
SRC Characteristics

- (=) Load-store design: only way to access memory is through load and store instructions
- (–) Operation on 32-bit words only, no byte or half-word operations.
- (=) Only a few addressing modes are supported
- (=) ALU Instructions are 3-register type
- (–) Branch instructions can branch unconditionally or conditionally on whether the value in a specified register is = 0, <> 0, >= 0, or < 0.
- (–) Branch-and-link instructions are similar, but leave the value of current PC in any register, useful for subroutine return.
- (–) Can only branch to an address in a register, not to a direct address.
- (=) All instructions are 32-bits (1-word) long.

(=) – Similar to commercial RISC machines
(–) – Less powerful than commercial RISC machines.

SRC Basic Instruction Formats

- There are three basic instruction format types

- The number of register specifier fields and length of the constant field vary

- Other formats result from unused fields or parts

```
31   27 26   22 21                              0
┌──────┬──────┬──────────────────────────────┐
│  op  │  ra  │             c1               │    Type 1
└──────┴──────┴──────────────────────────────┘

31   27 26   22 21   17 16                      0
┌──────┬──────┬──────┬────────────────────────┐
│  op  │  ra  │  rb  │           c2           │    Type 2
└──────┴──────┴──────┴────────────────────────┘

31   27 26   22 21   17 16   12 11              0
┌──────┬──────┬──────┬──────┬──────────────────┐
│  op  │  ra  │  rb  │  rc  │        c3        │    Type 3
└──────┴──────┴──────┴──────┴──────────────────┘
```

• **Details of formats:**

Fig 2.10 cont'd. SRC instructions

R0 — 32 32-bit general purpose registers
R31
PC
IR

2^{32} bytes of main memory
0
$2^{32} - 1$
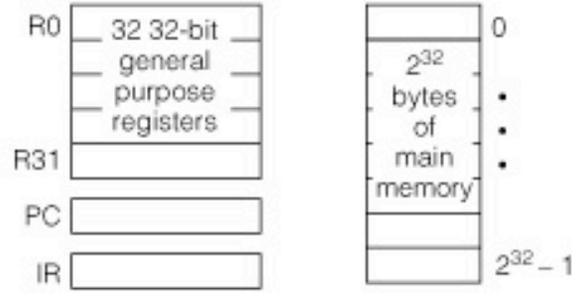
R[7] means contents of register 7

M[32] means contents of memory location 32

Instruction formats

Example

1. ld, st, la, addi, andi, ori
31 27 26 22 21 17 16   0
| Op | ra | rb | c2 |
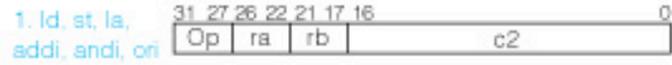
ld r3, A   (R[3] = M[A])
ld r3, 4(r5)   (R[3] = M[R[5] + 4])
addi r2, r4, 1   (R[2] = R[4] +1)

2. ldr, str, lar
31 27 26 22 21   0
| Op | ra | c1 |
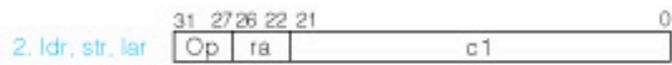
ldr r5, 8   (R[5] = M[PC + 8])
lar r6, 45   (R[6] = PC + 45)

3. neg, not
31 27 26 22 21 17 16   0
| Op | ra | rc | unused |
unused

neg r7, r9   (R[7] = − R[9])

4. br
31 27 26 22 21 17 16 12 11   2 0
| Op | | rb | rc | (c3) unused | Cond |
unused
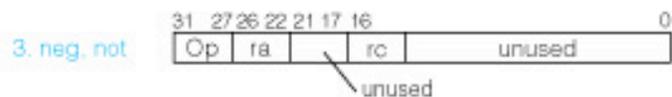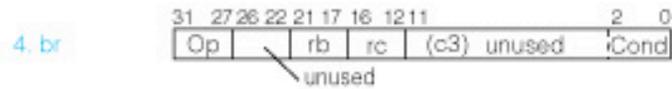
brzr r4, r0
(branch to R[4] if R[0] == 0)

5. brl
31 27 26 22 21 17 16 12 11   2 0
| Op | ra | rb | rc | (c3) unused | Cond |
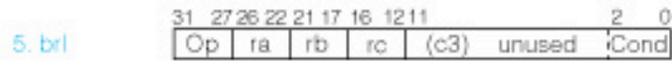
brlnz r6, r4, r0
(R[6] = PC; branch to R[4] if R[0] ≠ 0)

6. add, sub, and, or
31 27 26 22 21 17 16 12 11   0
| Op | ra | rb | rc | unused |

add r0, r2, r4 (R[0] = R[2] + R[4])

7. shr, shra shl, shc
7a
31 27 26 22 21 17   4 0
| Op | ra | rb | (c3) unused | Count |
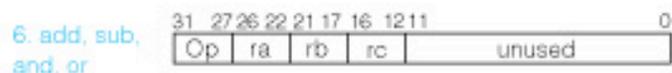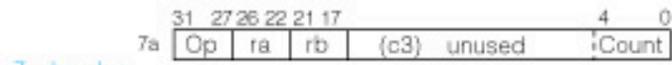
shr r0, r1, 4
(R[0] = R[1] shifted right by 4 bits)

7b
31 27 26 22 21 17 16 12   4 0
| Op | ra | rb | rc | (c3) unused | 00000 |
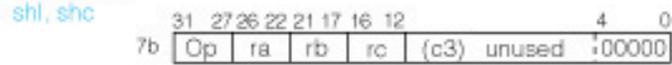
shl r2, r4, r6
(R[2] = R[4] shifted left by count in R[6])

8. nop, stop
31 27 26   0
| Op | unused |

stop

Tbl 2.4 Example Load & Store Instructions: Memory Addressing

- Address can be constant, constant+register, or constant+PC
- Memory contents or address itself can be loaded

Instruction	op	ra	rb	c1	Meaning	Addressing Mode
ld r1, 32	1	1	0	32	$R[1] \leftarrow M[32]$	Direct
ld r22, 24(r4)	1	22	4	24	$R[22] \leftarrow M[24+R[4]]$	Displacement
st r4, 0(r9)	3	4	9	0	$M[R[9]] \leftarrow R[4]$	Register indirect
la r7, 32	5	7	0	32	$R[7] \leftarrow 32$	Immediate
ldr r12, -48	2	12	–	-48	$R[12] \leftarrow M[PC -48]$	Relative
lar r3, 0	6	3	–	0	$R[3] \leftarrow PC$	Register (!)

(note use of la to load a constant)

Assembly Language Forms of Arithmetic and Logic Instructions

CSDA 2/e

Format	Example	Meaning
neg ra, rc	neg r1, r2	;Negate (r1 = -r2)
not ra, rc	not r2, r3	;Not (r2 = r3´)
add ra, rb, rc	add r2, r3, r4	;2's complement addition
sub ra, rb, rc		;2's complement subtraction
and ra, rb, rc		;Logical and
or ra, rb, rc		;Logical or
addi ra, rb, c2	addi r1, r3, 1	;Immediate 2's complement add
andi ra, rb, c2		;Immediate logical and
ori ra, rb, c2		;Immediate logical or

- Immediate subtract not needed since constant in addi may be negative

Branch Instruction Format

There are actually only two branch op codes:

```
br  rb, rc, c3<2..0>              ;branch to R[rb] if R[rc] meets
                                  ; the condition defined by c3<2..0>
brl ra, rb,  rc, c3<2..0>         ; R[ra] ← PC; branch as above
```

- It is c3<2..0>, the 3 lsbs of c3, that governs what the branch condition is:

lsbs	condition	Assy language form	Example
000	never	brlnv	brlnv r6
001	always	br, brl	br r5, brl r5
010	if rc = 0	brzr, brlzr	brzr r2, r4
011	if rc ≠ 0	brnz, brlnz	
100	if rc ≥ 0	brpl, brlpl	
101	if rc < 0	brmi, brlmi	

- Note that branch target address is always in register R[rb].
- It must be placed there explicitly by a previous instruction.

Tbl. 2.6 Branch Instruction Examples

Ass'y lang.	Example instr.	Meaning	op	ra	rb	rc	c3 ⟨2..0⟩	Branch Cond'n.
brlnv	brlnv r6	R[6] ← PC	9	6	—	—	000	never
br	br r4	PC ← R[4]	8	—	4	—	001	always
brl	brl r6,r4	R[6] ← PC; PC ← R[4]	9	6	4	—	001	always
brzr	brzr r5,r1	if (R[1]=0) PC ← R[5]	8	—	5	1	010	zero
brlzr	brlzr r7,r5,r1	R[7] ← PC;	9	7	5	1	010	zero
brnz	brnz r1, r0	if (R[0]≠0) PC← R[1]	8	—	1	0	011	nonzero
brlnz	brlnz r2,r1,r0	R[2] ← PC; if (R[0]≠0) PC← R[1]	9	2	1	0	011	nonzero
brpl	brpl r3, r2	if (R[2]≥0) PC← R[3]	8	—	3	2	100	plus
brlpl	brlpl r4,r3,r2	R[4] ← PC; if (R[2]≥0) PC← R[3]	9	4	3	2		plus
brmi	brmi r0, r1	if (R[1]<0) PC← R[0]	8	—	0	1	101	minus
brlmi	brlmi r3,r0,r1	R[3] ← PC; if (r1<0) PC← R[0]	9	3	0	1		minus

Branch Instructions—Example

C: goto Label3

SRC:

       lar r0, Label3   ; put branch target address into tgt reg.

       br r0   ; and branch

       • • •

Label3   • • •

Example of conditional branch

2/e

```
in C:   #define Cost 125
        if (X<0) then X = -X;
in SRC:
Cost  .equ  125            ;define symbolic constant
      .org  1000           ;next word will be loaded at address 1000₁₀
X:    .dw   1              ;reserve 1 word for variable X
      .org  5000           ;program will be loaded at location 5000₁₀
      lar   r31, Over      ;load address of "false" jump location
      ld    r1, X          ;load value of X into r1
      brpl  r31, r1        ;branch to Else if r1≥0
      neg   r1, r1         ;negate value
Over: • • •                ;continue
```

Address subscripts use 1000_{10} and 5000_{10}; branch condition $r1 \geq 0$.

Computer Systems Design and Architecture Second Edition

© 2004 Prentice Hall

RTN (Register Transfer Notation)

- Provides a formal means of describing machine structure and function

- Is at the "just right" level for machine descriptions

- Does not replace hardware description languages.

- Can be used to describe *what* a machine does (an Abstract RTN) without describing *how* the machine does it.

- Can also be used to describe a particular hardware implementation (A Concrete RTN)

RTN Notation (Cont'd.)

- At first you may find this "meta description" confusing, because it is a language that is used to describe a language.

- You will find that developing a familiarity with RTN will aid greatly in your understanding of new machine design concepts.

- We will describe RTN by using it to describe SRC.

Some RTN Features—
Using RTN to describe a machine's static properties

Static Properties

- Specifying registers
  - IR⟨31..0⟩ specifies a register named "IR" having 32 bits numbered 31 to 0

- "Naming" using the := naming operator:
  - op⟨4..0⟩ := IR⟨31..27⟩ specifies that the 5 msbs of IR be called op, with bits 4..0.
  - Notice that this does not create a new register, it just generates another name, or "alias" for an already existing register or part of a register.

Dynamic Properties

- Conditional expressions:

$$(op=12) \rightarrow R[ra] \leftarrow R[rb] + R[rc]: \quad \text{; defines the add instruction}$$

"if" condition   "then"   RTN Assignment Operator

This fragment of RTN describes the SRC add instruction. It says, "when the op field of IR = 12, then store in the register specified by the ra field, the result of adding the register specified by the rb field to the register specified by the rc field."

Using RTN to describe the SRC (static) Processor State

Processor state

PC⟨31..0⟩:   program counter
                         (memory addr. of next inst.)
IR⟨31..0⟩:   instruction register
Run: one bit run/halt indicator
Strt: start signal
R[0..31]⟨31..0⟩: general purpose registers

RTN Register Declarations

- General register specifications shows some features of the notation

- Describes a set of 32 32-bit registers with names R[0] to R[31]

$$R[0..31]\langle 31..0\rangle:$$

Name of registers

Register # in square brackets

.. specifies a range of indices

msb #

lsb#

Bit # in angle brackets

Colon separates statements with no ordering
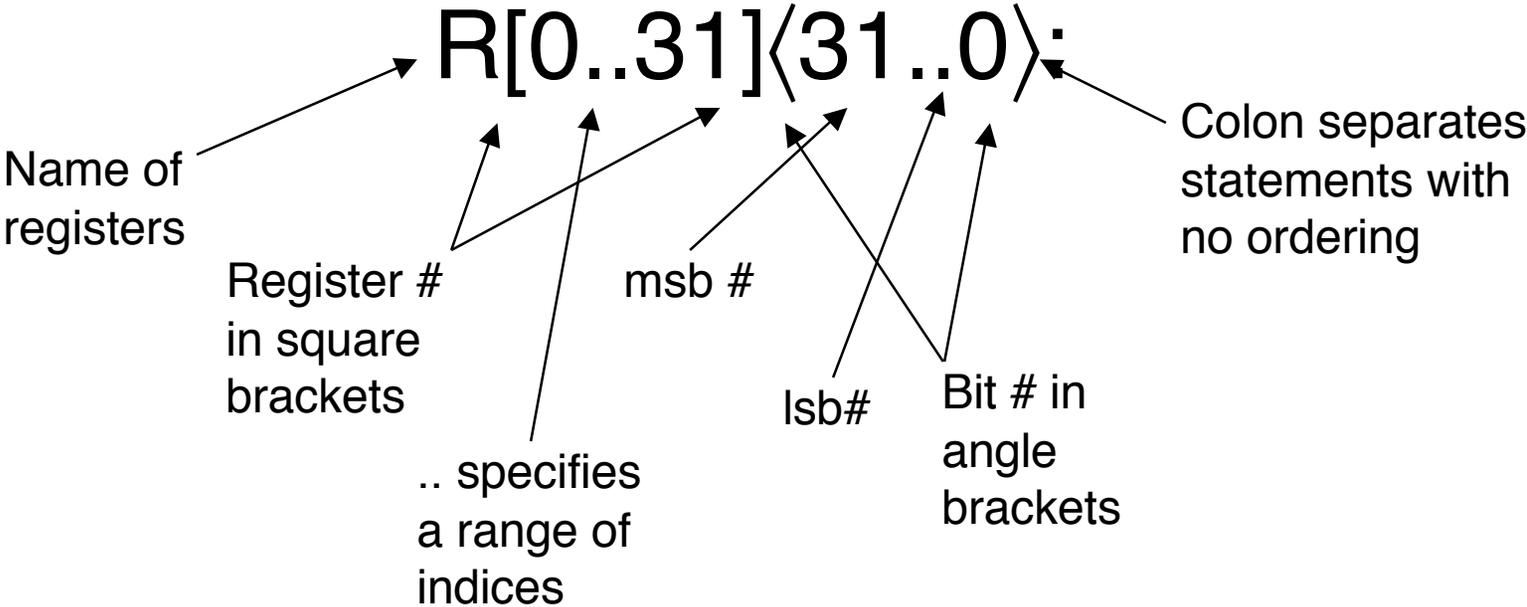
Memory Declaration: RTN Naming Operator

- Defining names with formal parameters is a powerful formatting tool
- Used here to define word memory (big endian)

Main memory state

$\text{Mem}[0..2^{32} - 1]\langle 7..0\rangle$: 2^{32} addressable bytes of memory

$M[x]\langle 31..0\rangle := \text{Mem}[x]\#\text{Mem}[x+1]\#\text{Mem}[x+2]\#\text{Mem}[x+3]$:

Dummy parameter

Naming operator

Concatenation operator

All bits in register if no bit index given

RTN Instruction Formatting Uses Renaming of IR Bits

Instruction formats

$op\langle 4..0 \rangle := IR\langle 31..27 \rangle$:	operation code field
$ra\langle 4..0 \rangle := IR\langle 26..22 \rangle$:	target register field
$rb\langle 4..0 \rangle := IR\langle 21..17 \rangle$:	operand, address index, or branch target register
$rc\langle 4..0 \rangle := IR\langle 16..12 \rangle$:	second operand, conditional test, or shift count register
$c1\langle 21..0 \rangle := IR\langle 21..0 \rangle$:	long displacement field
$c2\langle 16..0 \rangle := IR\langle 16..0 \rangle$:	short displacement or immediate field
$c3\langle 11..0 \rangle := IR\langle 11..0 \rangle$:	count or modifier field

Specifying dynamic properties of SRC: RTN Gives Specifics of Address Calculation

Effective address calculations (occur at runtime):

$$\text{disp}\langle 31..0\rangle := ((rb=0) \rightarrow c2\langle 16..0\rangle \text{ \{sign extend\}}: \qquad \text{displacement}$$
$$(rb\neq 0) \rightarrow R[rb] + c2\langle 16..0\rangle \text{ \{sign extend, 2's comp.\})}: \qquad \text{address}$$
$$\text{rel}\langle 31..0\rangle := PC\langle 31..0\rangle + c1\langle 21..0\rangle \text{ \{sign extend, 2's comp.\}}: \qquad \text{relative}$$
address

- Renaming defines displacement and relative addrs.
- New RTN notation is used
  - condition → expression means _if_ condition _then_ expression
  - modifiers in { } describe type of arithmetic or how short numbers are extended to longer ones
  - arithmetic operators (+ - * / etc.) can be used in expressions
- Register R[0] cannot be added to a displacement

- What set of memory cells can be addressed by direct addressing (displacement with rb=0)
  - If $c2\langle 16\rangle=0$ (positive displacement) absolute addresses range from 00000000H to 0000FFFFH
  - If $c2\langle 16\rangle=1$ (negative displacement) absolute addresses range from FFFF0000H to FFFFFFFFH
- What range of memory addresses can be specified by a relative address
  - The largest positive value of $C1\langle 21..0\rangle$ is $2^{21}-1$ and its most negative value is -2^{21}, so addresses up to $2^{21}-1$ forward and 2^{21} backward from the current PC value can be specified
- Note the difference between rb and R[rb]

Instruction Interpretation: RTN Description of Fetch/Execute

- Need to describe actions (not just declarations)
- Some new notation

Logical NOT

Logical AND

instruction_interpretation := (
¬Run∧Strt → Run ← 1:
Run → (IR ← M[PC]: PC ← PC + 4; instruction_execution));

Register transfer

Separates statements
that occur in sequence

RTN Sequence and Clocking

- In general, RTN statements separated by : take place during the same clock pulse

- Statements separated by ; take place on successive clock pulses

- This is not entirely accurate since some things written with one RTN statement can take several clocks to perform

- More precise difference between : and ;

    - The order of execution of statements separated by : does not matter

    - If statements are separated by ; the one on the left must be complete before the one on the right starts

- In the expression IR ← M[PC]: PC ← PC + 4; which value of PC applies to M[PC] ?

- The rule in RTN is that all right hand sides of   ":" - separated RTs are evaluated before any LHS is changed

  - In logic design, this corresponds to "master-slave" operation of flip-flops

- We see what happens when Run is true and when Run is false but Strt is true. What about the case of Run and Strt both false?

  - Since no action is specified for this case, the RTN implicitly says that no action occurs in this case

Individual Instructions

- instruction_interpretation contained a forward reference to instruction_execution
- instruction_execution is a long list of conditional operations
  - The condition is that the op code specifies a given inst.
  - The operation describes what that instruction does
- Note that the operations of the instruction are done after (;) the instruction is put into IR and the PC has been advanced to the next inst.

instruction_execution := (
    ld (:= op= 1) → R[ra] ← M[disp]:   load register
    ldr (:= op= 2) → R[ra] ← M[rel]:   load register relative
    st (:= op= 3) → M[disp] ← R[ra]:   store register
    str (:= op= 4) → M[rel] ← R[ra]:   store register relative
    la (:= op= 5) → R[ra] ← disp:load displacement address
    lar (:= op= 6) → R[ra] ← rel: load relative address

- The in-line definition (:= op=1) saves writing a separate definition ld := op=1 for the ld mnemonic

- The previous definitions of disp and rel are needed to understand all the details

ii := instruction_interpretation:

ie := instruction_execution :

ii := (¬Run∧Strt → Run ← 1:
         Run → (IR ← M[PC]: PC ← PC + 4;
         ie));

ie := (
    ld (:= op= 1) → R[ra] ← M[disp]:   Big switch
    ldr (:= op= 2) → R[ra] ← M[rel]:   statement
    . . .   on the opcode
    stop (:= op= 31) → Run ← 0:
); ii

Thus ii and ie invoke each other, as coroutines.

Use of RTN Definitions: Text Substitution Semantics

ld (:= op= 1) → R[ra] ← M[disp]:

disp⟨31..0⟩ := ((rb=0) → c2⟨16..0⟩ {sign extend}:
      (rb≠0) → R[rb] + c2⟨16..0⟩ {sign extend, 2's comp.}):

ld (:= op= 1) → R[ra] ← M[
           ((rb=0) → c2⟨16..0⟩ {sign extend}:
           (rb≠0) → R[rb] + c2⟨16..0⟩ {sign extend, 2's comp.}):
                      ]:

- An example:
  - If IR = 00001 00101 00011 0000000000001011
  - then ld → R[5] ← M[R[3] + 11]:

RTN Descriptions of SRC Branch Instructions

- Branch condition determined by 3 lsbs of inst.
- Link register (R[ra]) set to point to next inst.

$$
\begin{aligned}
\text{cond} := (\ & c3\langle 2..0\rangle = 0 \rightarrow 0: && \text{never} \\
& c3\langle 2..0\rangle = 1 \rightarrow 1: && \text{always} \\
& c3\langle 2..0\rangle = 2 \rightarrow R[rc] = 0: && \text{if register is zero} \\
& c3\langle 2..0\rangle = 3 \rightarrow R[rc] \neq 0: && \text{if register is nonzero} \\
& c3\langle 2..0\rangle = 4 \rightarrow R[rc]\langle 31\rangle = 0: && \text{if positive or zero} \\
& c3\langle 2..0\rangle = 5 \rightarrow R[rc]\langle 31\rangle = 1\ ): && \text{if negative}
\end{aligned}
$$

br (:= op= 8) → (cond → PC ← R[rb]):   conditional branch

brl (:= op= 9) → (R[ra] ← PC:

          cond → (PC ← R[rb])):   branch and link

RTN for Arithmetic and Logic

add (:= op=12) → R[ra] ← R[rb] + R[rc]:
addi (:= op=13) → R[ra] ← R[rb] + c2⟨16..0⟩ {2's comp. sign ext.}:
sub (:= op=14) → R[ra] ← R[rb] - R[rc]:
neg (:= op=15) → R[ra] ← -R[rc]:
and (:= op=20) → R[ra] ← R[rb] ∧ R[rc]:
andi (:= op=21) → R[ra] ← R[rb] ∧ c2⟨16..0⟩ {sign extend}:
or (:= op=22) → R[ra] ← R[rb] ∨ R[rc]:
ori (:= op=23) → R[ra] ← R[rb] ∨ c2⟨16..0⟩ {sign extend}:
not (:= op=24) → R[ra] ← ¬R[rc]:

- Logical operators: <u>and</u> ∧ <u>or</u> ∨ and <u>not</u> ¬

RTN for Shift Instructions

- Count may be 5 lsbs of a register or the instruction
- Notation: @ - replication, # - concatenation

$$n := (\quad (c3\langle 4..0\rangle = 0) \rightarrow R[rc]\langle 4..0\rangle:$$
$$(c3\langle 4..0\rangle \neq 0) \rightarrow c3\langle 4..0\rangle):$$

shr (:= op=26) $\rightarrow$ R[ra]$\langle 31..0\rangle \leftarrow$ (n @ 0) # R[rb]$\langle 31..n\rangle$:

shra (:= op=27) $\rightarrow$ R[ra]$\langle 31..0\rangle \leftarrow$ (n @ R[rb]$\langle 31\rangle$) # R[rb]$\langle 31..n\rangle$:

shl (:= op=28) $\rightarrow$ R[ra]$\langle 31..0\rangle \leftarrow$ R[rb]$\langle 31-n..0\rangle$ # (n @ 0):

shc (:= op=29) $\rightarrow$ R[ra]$\langle 31..0\rangle \leftarrow$ R[rb]$\langle 31-n..0\rangle$ # R[rb]$\langle 31..32-n\rangle$:

Example of Replication and Concatenation in Shift

- Arithmetic shift right by 13 concatenates 13 copies of the sign bit with the upper 19 bits of the operand

shra r1, r2, 13

R[2]= | 1001 0111 1110 1010 1110 1100 0001 0110 |

$13@R[2]\langle 31 \rangle$ # $R[2]\langle 31..13 \rangle$

R[1]= | 1111 1111 1111 1 | 100 1011 1111 0101 0111 |

Assembly Language for Shift

- Form of assembly language instruction tells whether to set c3=0

```
shr ra, rb, rc           ;Shift rb right into ra by 5 lsbs of rc
shr ra, rb, count ;Shift rb right into ra by 5 lsbs of inst
shra ra, rb, rc           ;AShift rb right into ra by 5 lsbs of rc
shra ra, rb, count        ;AShift rb right into ra by 5 lsbs of inst
shl ra, rb, rc            ;Shift rb left into ra by 5 lsbs of rc
shl ra, rb, count ;Shift rb left into ra by 5 lsbs of inst
shc ra, rb, rc            ;Shift rb circ. into ra by 5 lsbs of rc
shc ra, rb, count;Shift rb circ. into ra by 5 lsbs of inst
```

End of RTN Definition of instruction_execution

nop (:= op= 0) → :   No operation

stop (:= op= 31) → Run ← 0:   Stop instruction

);   End of instruction_execution

 instruction_interpretation.

- We will find special use for nop in pipelining

- The machine waits for Strt after executing stop

- The long conditional statement defining instruction_execution ends with a direction to go repeat instruction_interpretation, which will fetch and execute the next instruction (if Run still =1)

Confused about RTN and SRC?

- **SRC is a Machine Language**
  - It can be interpreted by either hardware or software simulator.

- **RTN is a *Specification Language***
  - Specification languages are languages that are used to specify other languages or systems—a *metalanguage*.
  - Other examples: LEX, YACC, VHDL, Verilog

Figure 2.10 may help clear this up...

Fig 2.11 The Relationship of RTN to SRC

SRC specification written in RTN

RTN compiler

Generated processor

SRC program
and data

SRC interpreter
or simulator

Data output
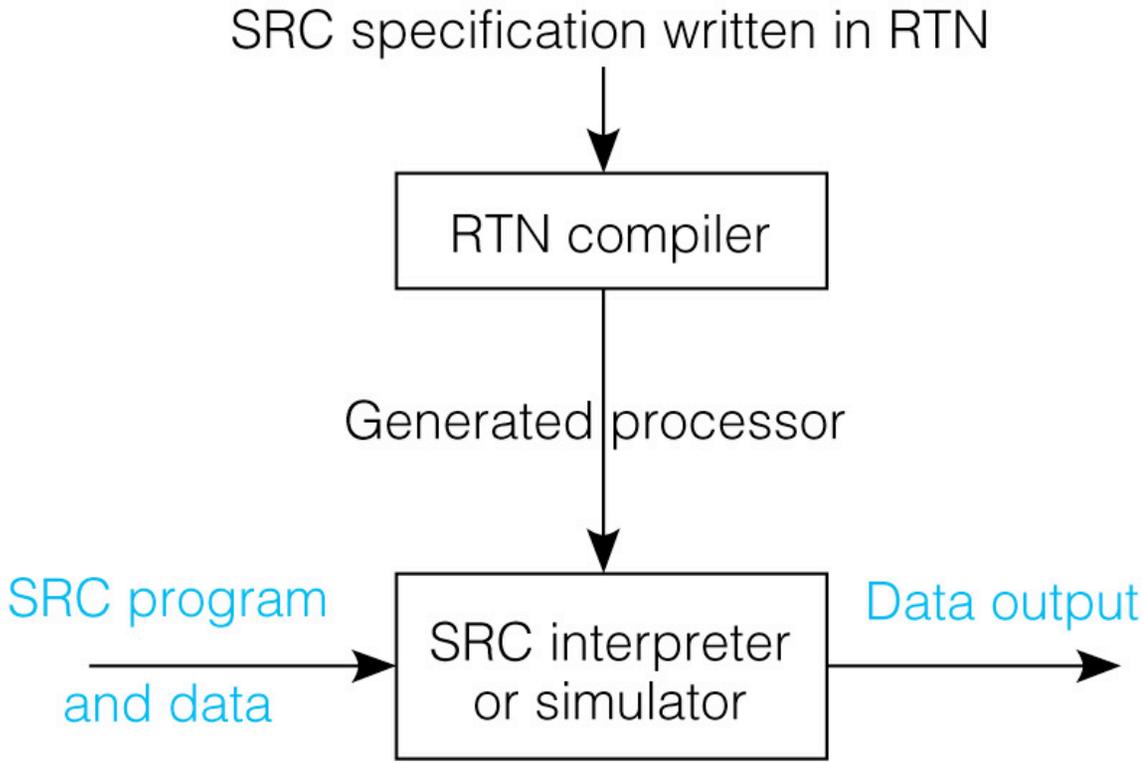
Copyright © 2004 Pearson Prentice Hall, Inc.

A Note about Specification Languages

- They allow the description of *what* without having to specify *how.*
- They allow precise and unambiguous specifications, unlike natural language.
- They reduce errors:
    - errors due to misinterpretation of imprecise specifications written in natural language
    - errors due to confusion in design and implementation - "human error."
- Now the designer must debug the specification!
- Specifications can be automatically checked and processed by tools.
    - An RTN specification could be input to a simulator generator that would produce a simulator for the specified machine.
    - An RTN specification could be input to a compiler generator that would generate a compiler for the language, whose output could be run on the simulator.

Addressing Modes Described in RTN (Not SRC)

Target register

Mode name	Assembler Syntax	RTN meaning	Use
Register	Ra	R[t] ← R[a]	Tmp. Var.
Register indirect	(Ra)	R[t] ← M[R[a]]	Pointer
Immediate	#X	R[t] ← X	Constant
Direct, absolute	X	R[t] ← M[X]	Global Var.
Indirect	(X)	R[t] ← M[M[X]]	Pointer Var.
Indexed, based, or displacement	X(Ra)	R[t] ← M[X + R[a]]	Arrays, structs
Relative	X(PC)	R[t] ← M[X + PC]	Vals stored w pgm
Autoincrement	(Ra)+	R[t] ← M[R[a]]; R[a] ← R[a] + 1	Sequential
Autodecrement	- (Ra)	R[a] ← R[a] - 1; R[t] ← M[R[a]]	access.

Fig. 2.12 Register transfers can be mapped to Digital Logic Circuits.

- Implementing the RTN statement A ← B
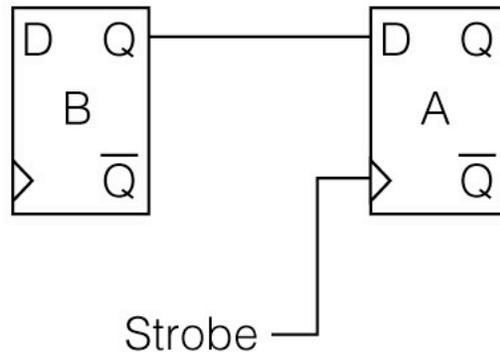
(a) Hardware   (b) Timing
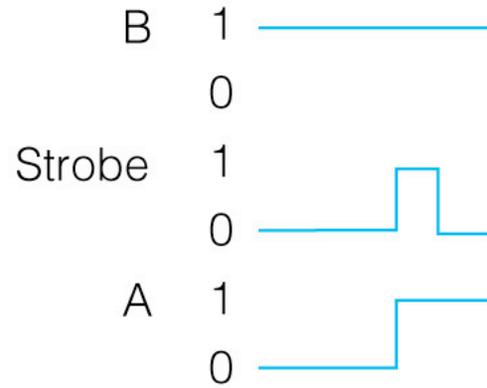
Fig. 2.13 Multiple Bit Register Transfer

- Implementing A⟨m..1⟩ ← B⟨m..1⟩
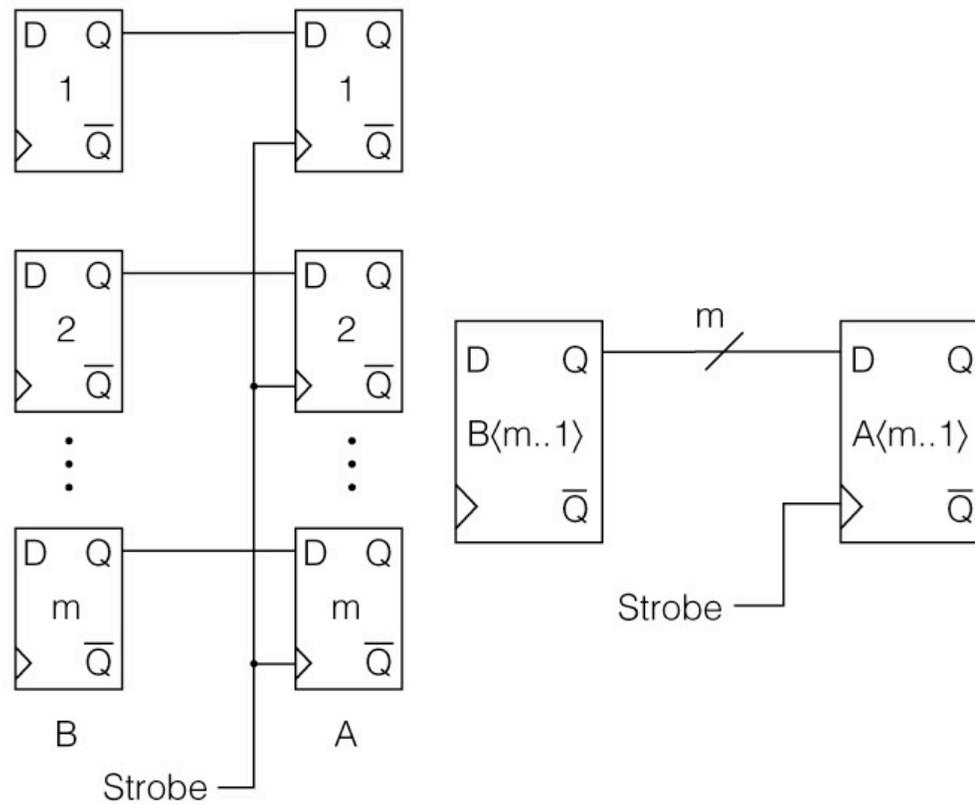
(a) Individual flip-flops

(b) Abbreviated notation

Copyright © 2004 Pearson Prentice Hall, Inc.

Fig. 2.14 Data Transmission View of Logic Gates

■ Logic gates can be used to control the transmission of data:

Data gate

Controlled complement
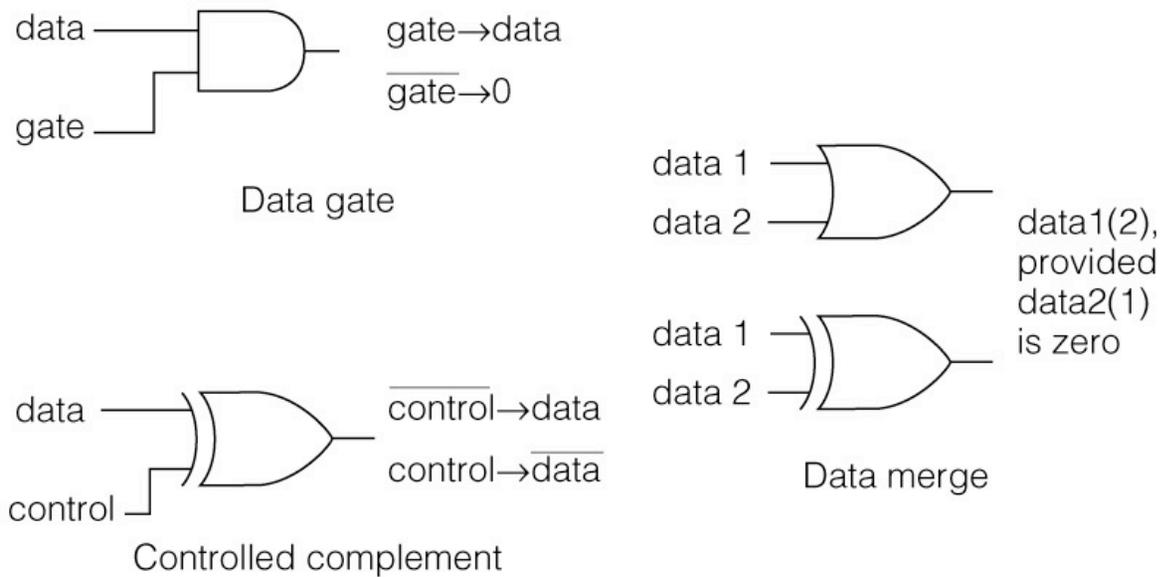
Data merge

Fig. 2.15 Multiplexer as a 2 Way Gated Merge

- Data from multiple sources can be selected for transmission
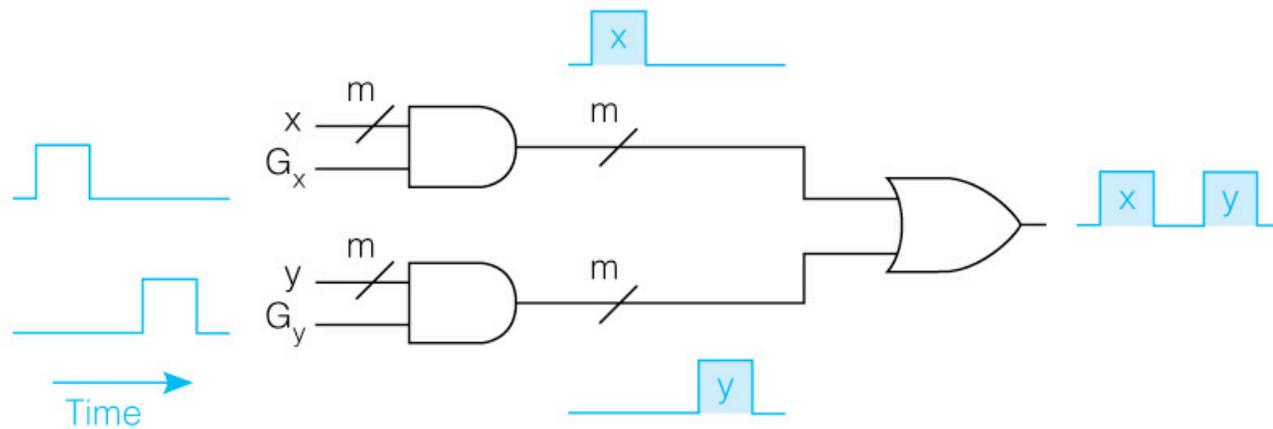
Copyright © 2004 Pearson Prentice Hall, Inc.

Fig. 2.16 m-bit Multiplexer and Symbol

An n-way gated merge

D_0
G_0
m
m

D_1
G_1
m
m

$\vdots$

D_{n-1}
G_{n-1}
m
m

m

(a) Multiplexer in terms of gates

An n-way multiplexer with decoder

D_0 m

D_1 m

$\vdots$

D_{n-1} m

m

k

Select

(b) Symbol abbreviation
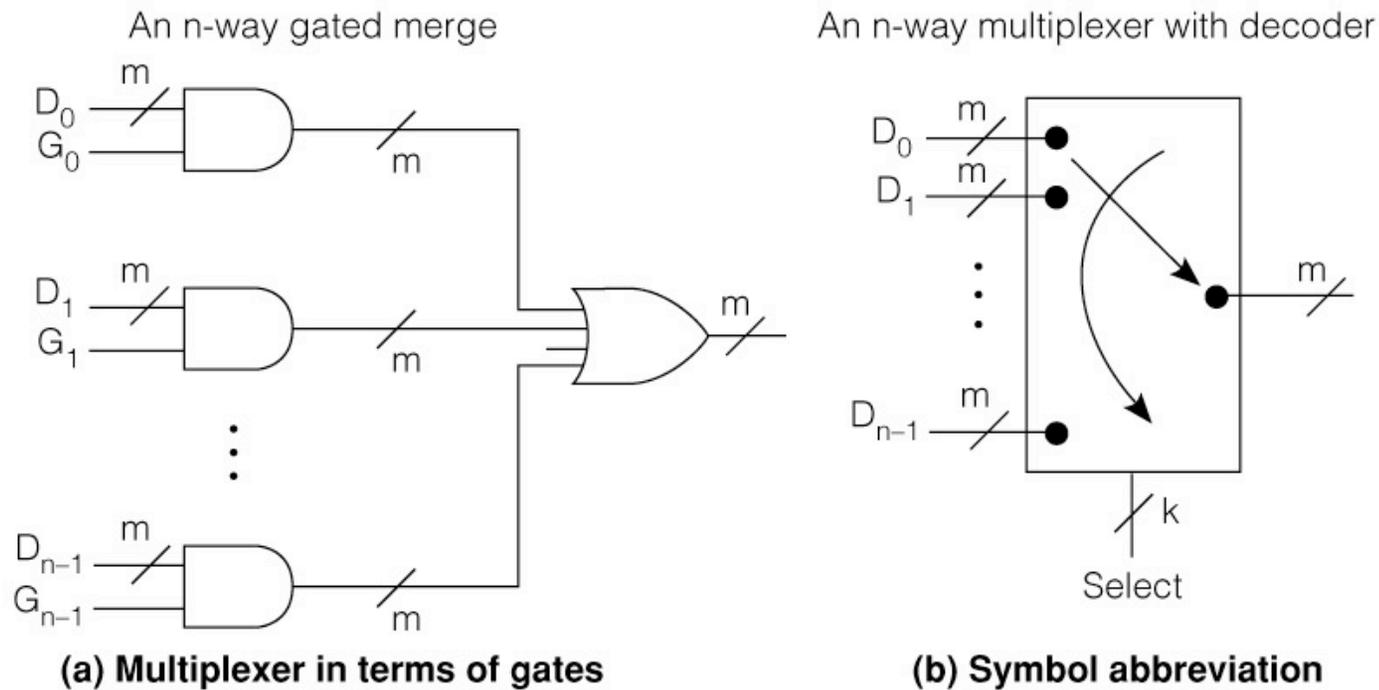
- Multiplexer gate signals G_i may be produced by a binary to one-out-of-n decoder

Fig. 2.17 Separating Merged Data
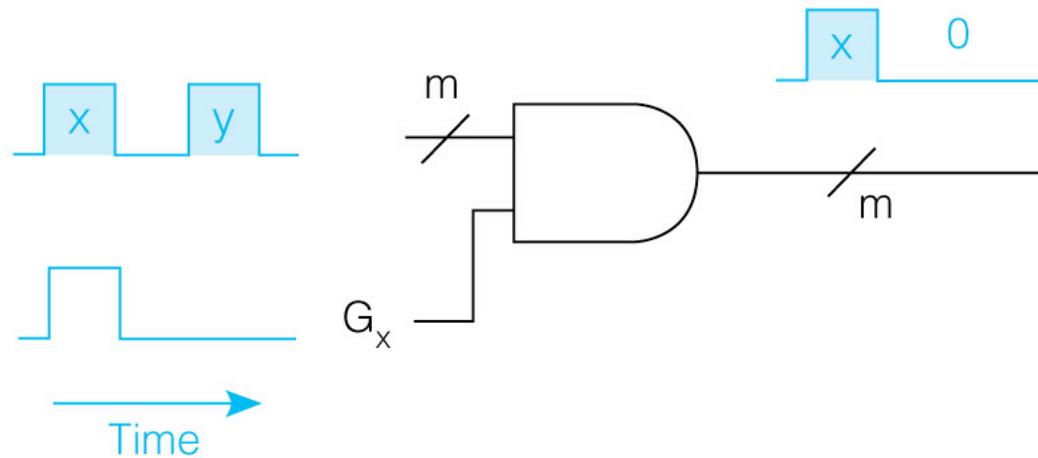
Copyright © 2004 Pearson Prentice Hall, Inc.

- Merged data can be separated by gating at the right time
- It can also be strobed into a flip-flop when valid

Fig. 2.18   Multiplexed Register Transfers using Gates and Strobes
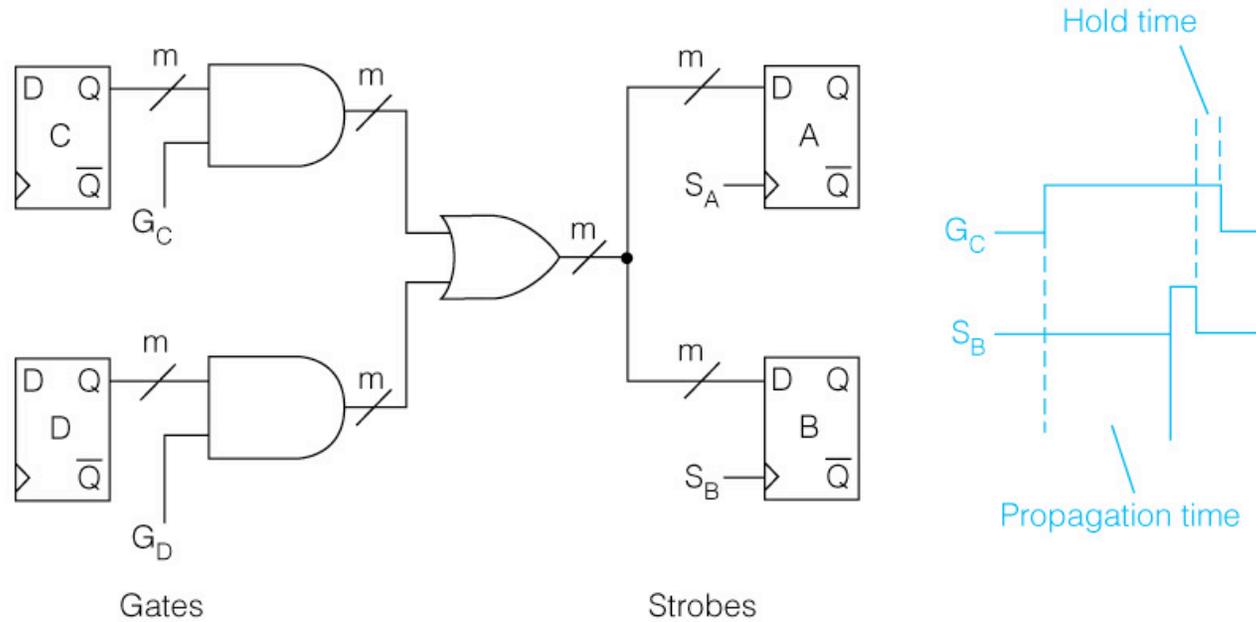
Copyright © 2004 Pearson Prentice Hall, Inc.

- Selected gate and strobe determine which Register is Transferred to where.
- A←C, and B←C can occur together, but not A←C, and B←D

Fig. 2.19 Open-Collector NAND Gate Output Circuit

Inputs		Output	
0v	0v	Open	(Out = +V)
0v	+V	Open	(Out = +V)
+V	0v	Open	(Out = +V)
+V	+V	Closed	(Out = 0v)

(a) Open-collector NAND truth table
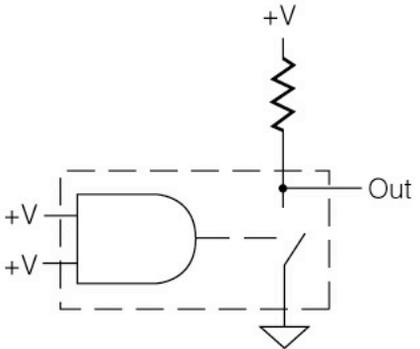
(b) Open-collector NAND
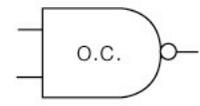
(c) Symbol

Copyright © 2004 Pearson Prentice Hall, Inc.

Fig. 2.20 Wired AND Connection of Open-Collector Gates

+V

Out

a

b
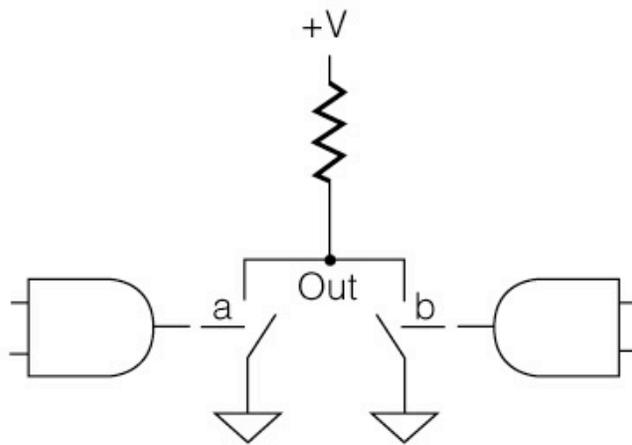
(a) Wired AND connection

+V

o.c.

o.c.
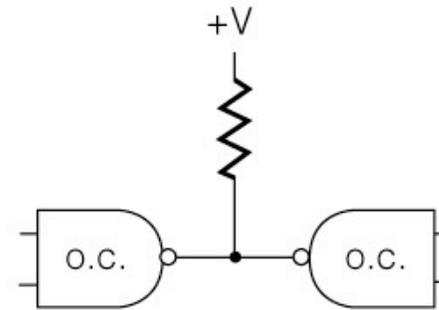
(b) With symbols

Switch		Wired AND output
a	b	
Closed(0)	Closed(0)	0v (0)
Closed(0)	Open (1)	0v (0)
Open (1)	Closed(0)	0v (0)
Open (1)	Open (1)	+V (1)

(c) Truth table

Copyright © 2004 Pearson Prentice Hall, Inc.

Fig. 2.21 Open Collector Wired OR Bus

- DeMorgan's OR by not of AND of nots
- Pull-up resistor removed from each gate - open collector
- One pull-up resistor for whole bus
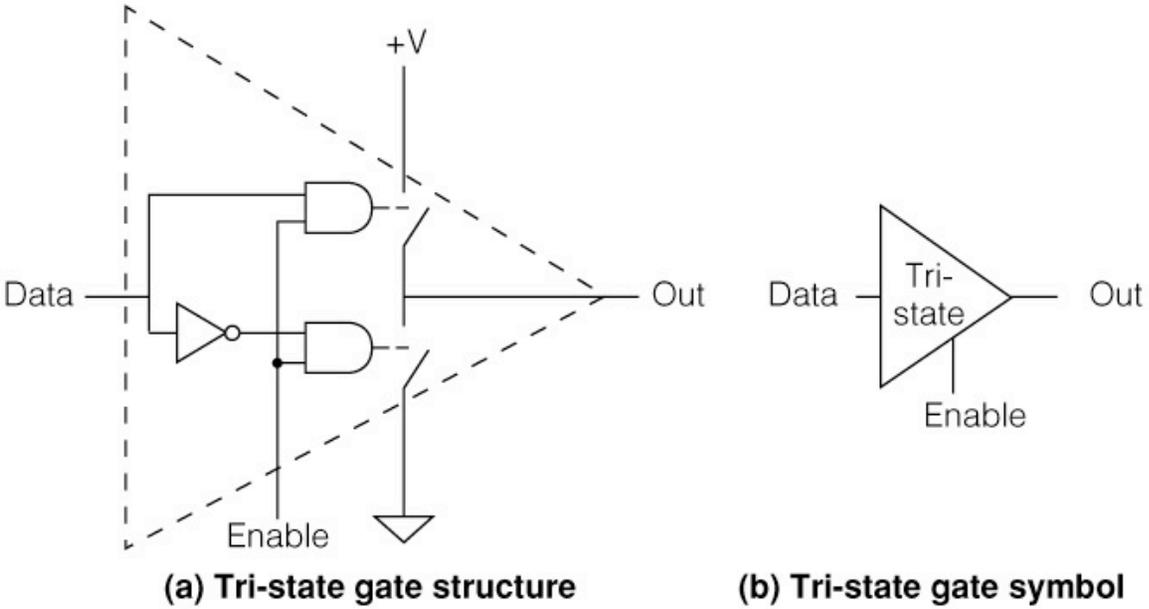- Forms an OR distributed over the connection
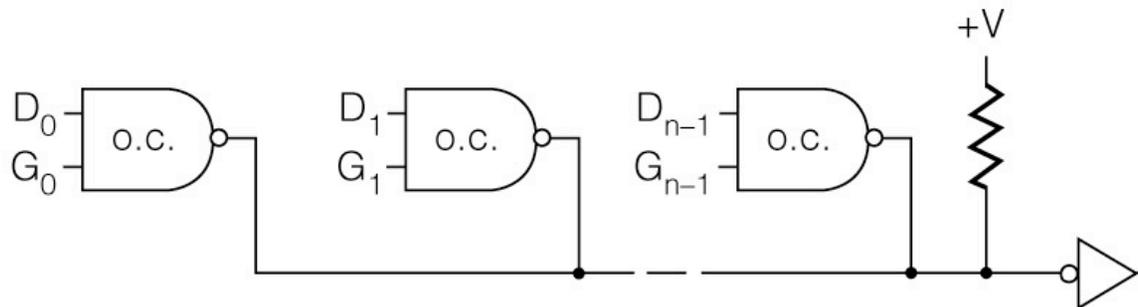
Fig. 2.22 Tri-state Gate Internal Structure and Symbol
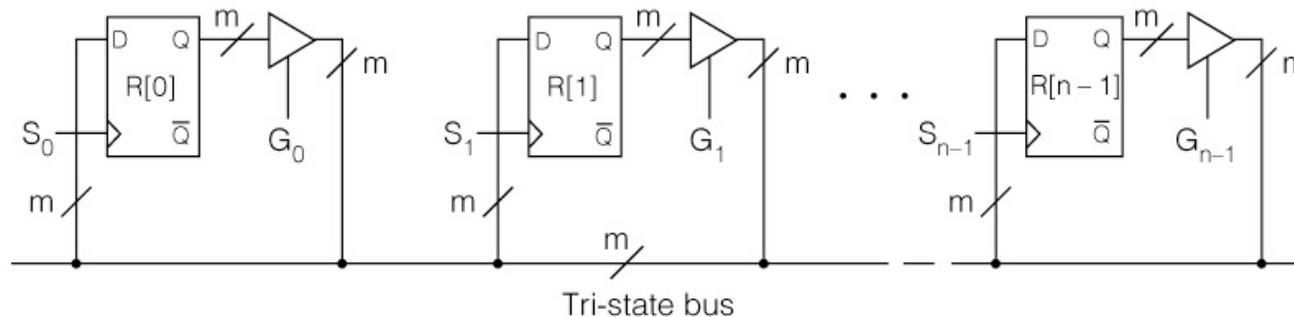
(a) Tri-state gate structure

(b) Tri-state gate symbol

Enable	Data	Output
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(c) Tri-state gate truth table

Fig. 2.23 Registers Connected by a Tri-state Bus

Tri-state bus

- Can make any register transfer R[i]←R[j]
- Can't have $G_i = G_j = 1$ for i≠j
- Violating this constraint gives low resistance path from power supply to ground—with predictable results!

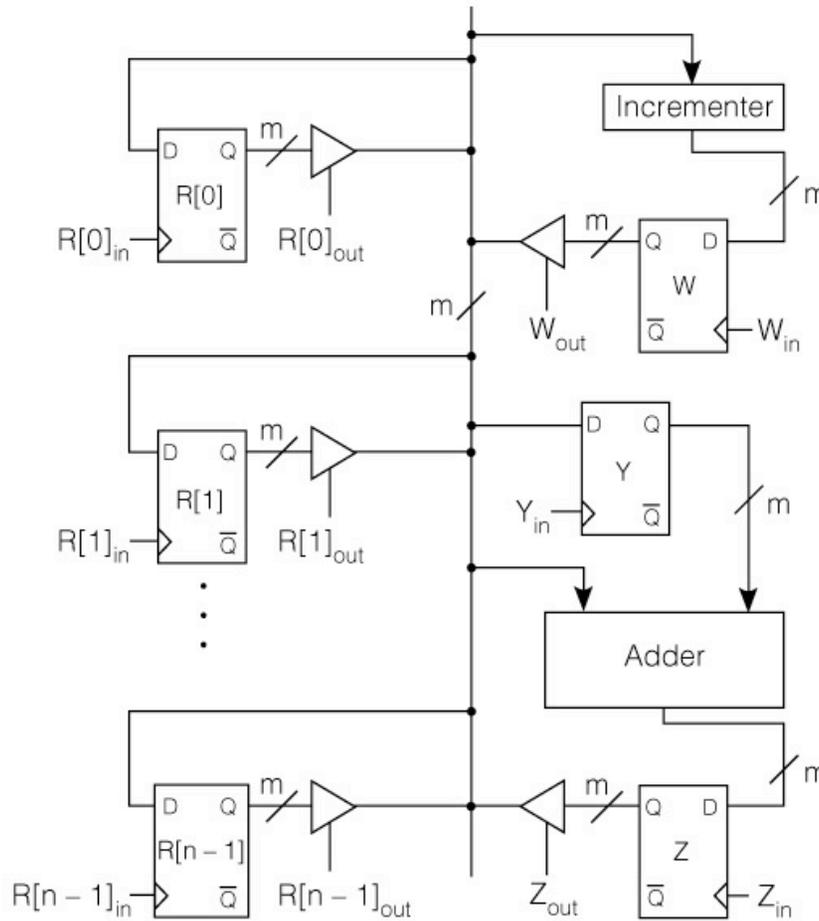# Fig. 2.24 Registers and Arithmetic Connected by One Bus

Example

Abstract RTN

R[3] ← R[1]+R[2];

Concrete RTN

Y ← R[2];

Z ← R[1]+Y;

R[3] ← Z;

Control Sequence

R[2]$_{out}$, Y$_{in}$;

R[1]$_{out}$, Z$_{in}$;

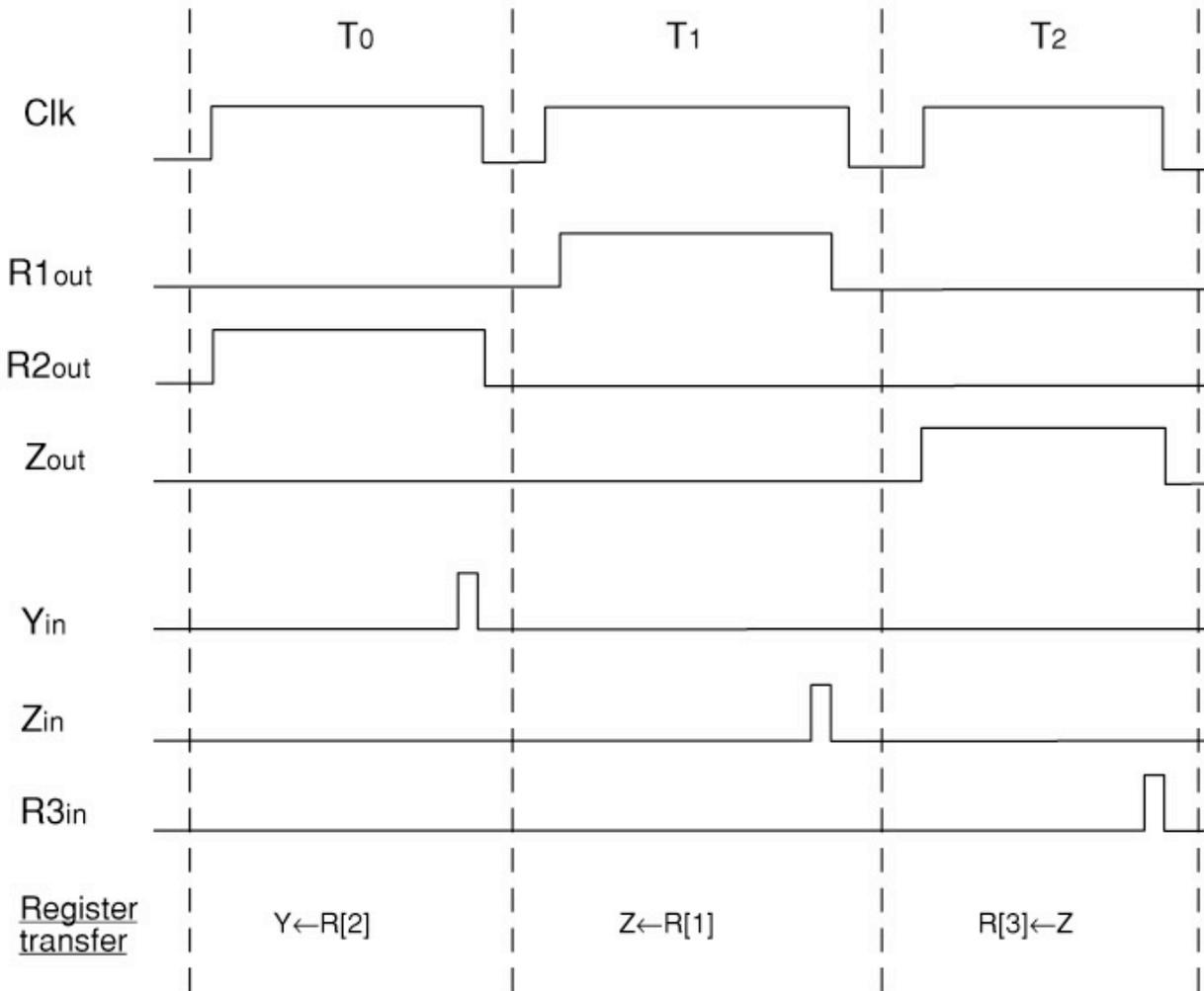**Z$_{out}$, R[3]$_{in}$;**

Combinational Logic—no memory

Copyright © 2004 Pearson Prentice Hall, Inc.

Notice that what could be described in one step in the abstract RTN took three steps on this particular hardware

Figure 2.25 Timing of the Register Transfers

- Discuss: difference between gating signals and strobing signals

- Discuss factors influencing minimum clock period.

Copyright © 2004 Pearson Prentice Hall, Inc.

RT's Possible with the One Bus Structure

- R[i] or Y can get the contents of anything but Y
- Since result different from operand, it cannot go on the bus that is carrying the operand
- Arithmetic units thus have result registers
- Only one of two operands can be on the bus at a time, so adder has register for one operand
- R[i] ← R[j] + R[k] is performed in 3 steps: Y←R[k]; Z←R[j] + Y; R[i]←Z;
- R[i] ← R[j] + R[k] is high level RTN description
- Y←R[k]; Z←R[j] + Y; R[i]←Z; is concrete RTN
- Map to control sequence is: $R[2]_{out}$, Y_{in}; $R[1]_{out}$, Z_{in}; Z_{out}, $R[3]_{in}$;

From Abstract RTN to Concrete RTN to Control Sequences

- The ability to begin with an abstract description, then describe a hardware design and resulting concrete RTN and control sequence is powerful.

- We shall use this method in Chapter 4 to develop various hardware designs for SRC

Chapter 2 Summary

- Classes of computer ISAs

- Memory addressing modes

- SRC: a complete example ISA

- RTN as a description method for ISAs

- RTN description of addressing modes

- Implementation of RTN operations with digital logic circuits

- Gates, strobes, and multiplexers