



# Chapter 2: Machines, Machine Languages, and Digital Logic

---

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

Some modifications by Tom Noack, 2010

## 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.

C

S

D

A

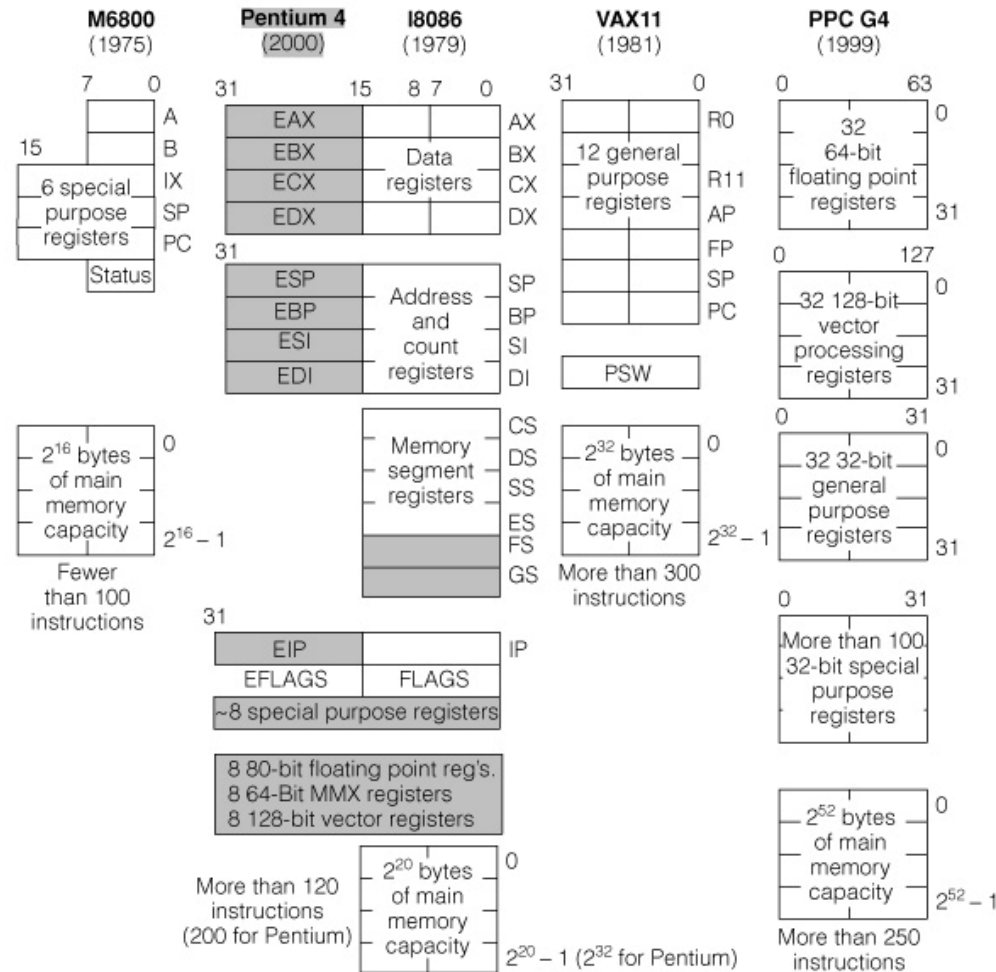
2/e

## 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



Copyright © 2004 Pearson Prentice Hall, Inc.

# What Must an Instruction Specify?

Data Flow



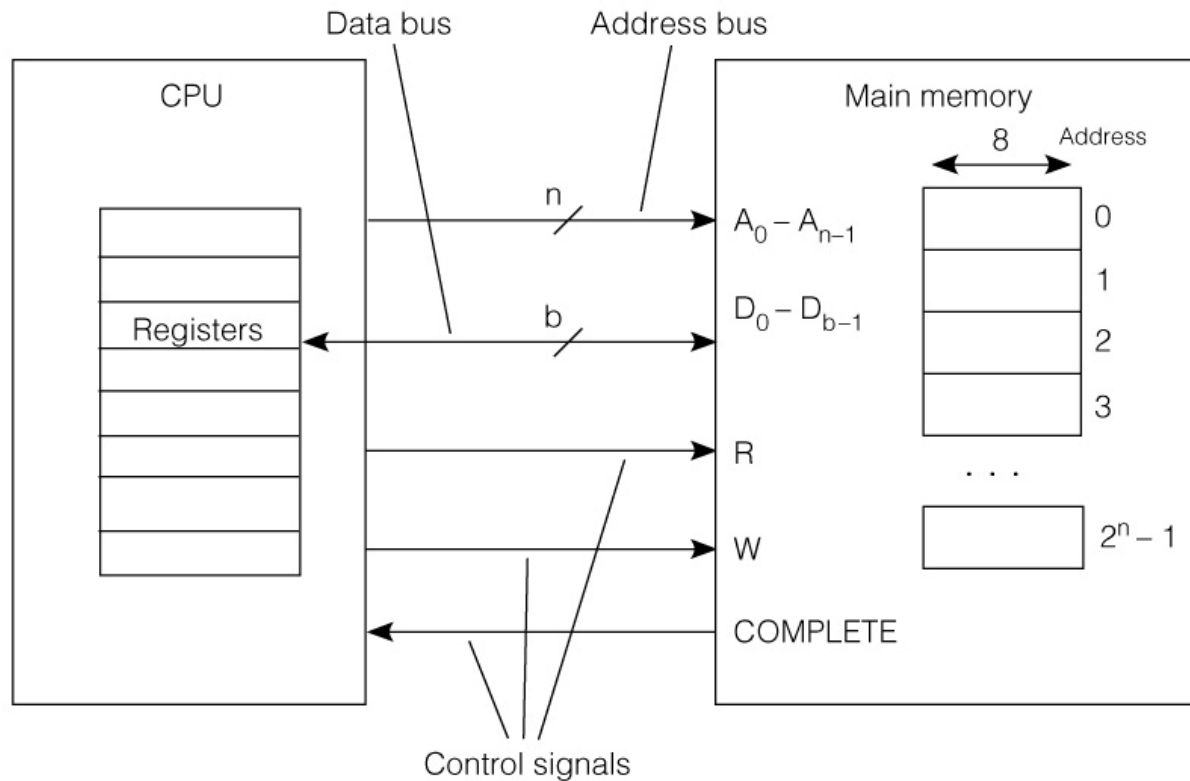
- Which operation to perform: add r0, r1, r3
- Where to find the operand or operands add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add r0, r1, r3
  - Again CPU register or memory cell
- Location of next instruction add r0, r1, r3 br endloop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



# The Bus idea

- A bus is a system for transferring data
  - Components
    - Data – what is being transferred
    - Address – where from/to
    - Control – describes and synchronizes the transfer, R/W, etc.
  - Mechanization
    - Parallel traces on a parentboard or inside a chip
    - Serial – USB, Ethernet, SCSI
  - Interfacing circuits to a bus on a chip
    - Gate – places data on a bus – usually tri-state
    - Strobe – captures data from the bus
    - Transfer is
      - gate (one source at a time, signal is  $Y_{out}$ )
      - Strobe (can be several destinations, signal is  $Y_{in}$ )
      - One transfer per bus per clock cycle

# Fig. 2.2 Accessing Memory—Reading from Memory



**Typical bus transaction**  
**Note data, address, control**

For a Memory Read:

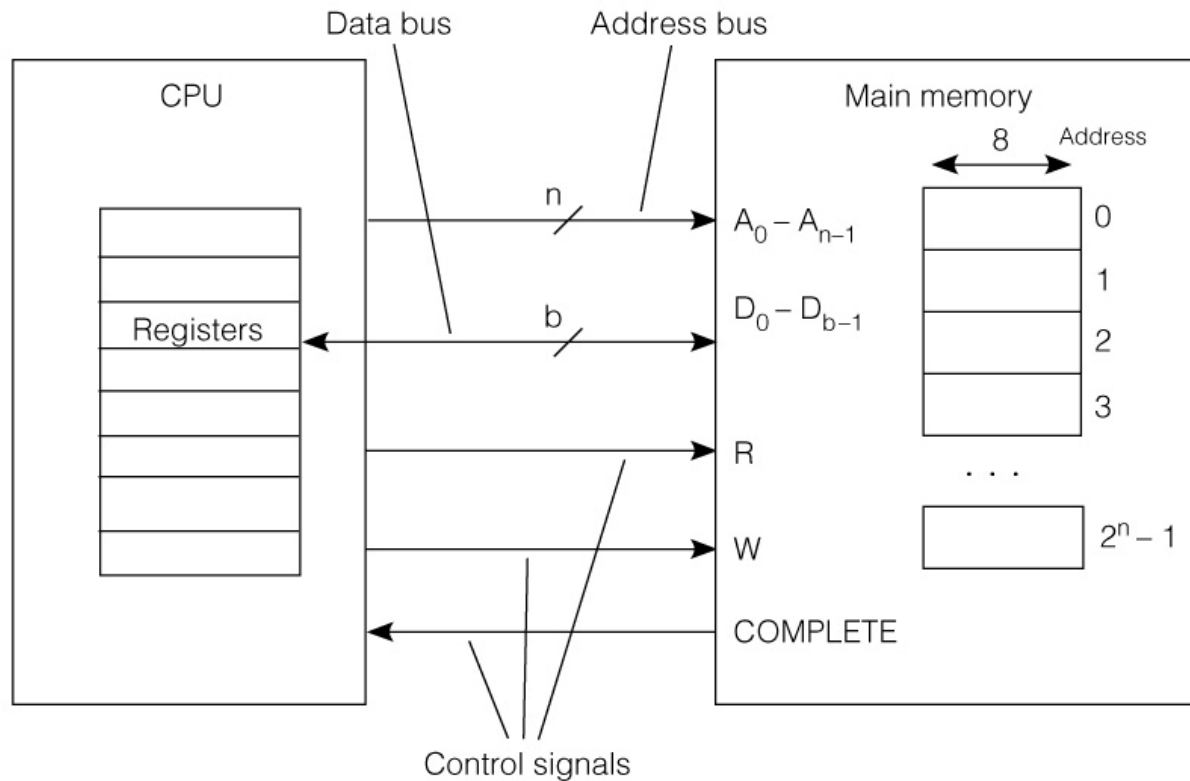
CPU applies desired address to Address lines  $A_0 - A_{n-1}$

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

Copyright © 2004 Pearson Prentice Hall, Inc.

# Figure 2.2 Accessing Memory—Writing to Memory



For a Memory Write:

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

Copyright © 2004 Pearson Prentice Hall, Inc.



C

S

D

A

2/e

## 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


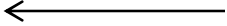
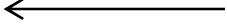
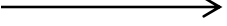

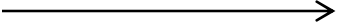
### Arithmetic and logic (ALU) instructions

- 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

Inst ruct.	Meaning	Machine
 MOV A, B	Move 16 bits from mem. loc. A to loc. B	VAX11
 l w z R3, A	Move 32 bits from mem. loc. A to reg. R3	PPC601
 l i \$3, 455	Load the 32 bit integer 455 into reg. 3	MIPS R3000
 m o v R4, d o u t	Move 16 bits from R4 to out port dout	DEC PDP11
 I N, A L, K B D	Load a byte from in port KBD to accum.	Intel Pentium
 L E A . L ( A 0 ), A 2	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

C

S

D

A

2/e

# Assembler and machine differences

- Assembler differences
  - One processor may be supported by several different assemblers
  - Destination-first
    - INTEL/Microsoft
  - Source-first
    - GNU, Motorola, VAX
  - Processor-independent
    - GNU – free software foundation – used in most Linux distributions
    - Register names and opcodes can't be reserved words, so prefixes are used to indicate registers and immediates
    - Table-driven – one assembler skeleton
  - Processor-dependent
    - Specialized, such as INTEL, Microsoft, Motorola
    - Register names and opcodes are reserved words

**C****S****D****A****2/e**

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

<b>Instruction</b>	<b>Meaning</b>	<b>Machine</b>
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

C

S

D

A

2/e

# ALU instruction and assembler conventions

---

- Instruction varieties
  - Zero- to four-address code
  - Combined memory and computation
  - Either-or (RISC/CISC) load-store or operate in registers
  - Operand sizes (byte/half/word/double)
    - Can be specified by opcode (add.w, sub.h)
    - Or by operand attribute (register size or defined name)
    - Or by keyword (word/byte ptr)
- Assembler varieties
  - Processor-independent
  - Processor-dependent

**C****S****D****A**

2/e

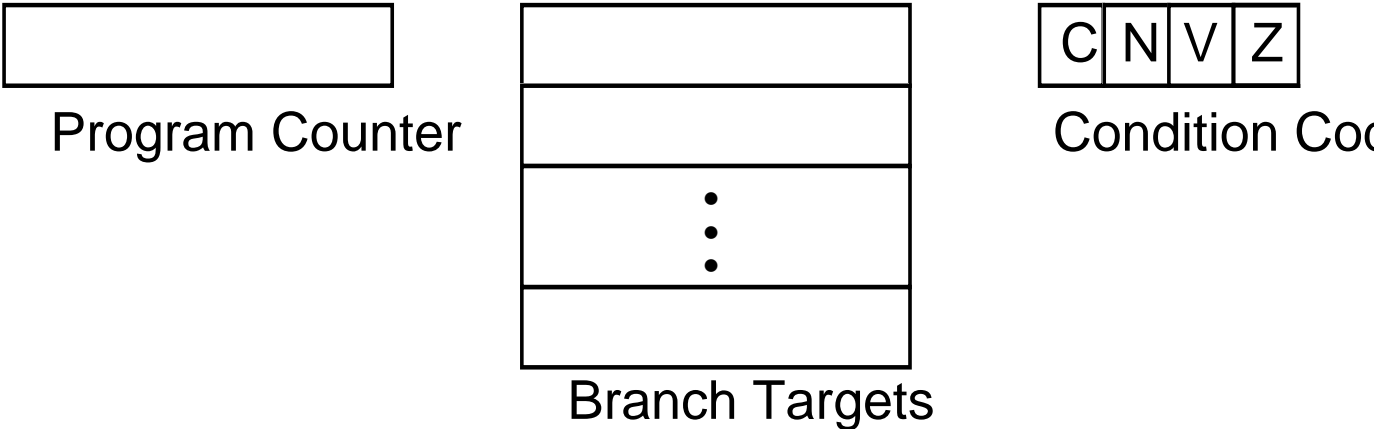
## Tbl 2.3 Examples of Branch Instructions

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

# 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



# 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

```
if NUM==5 then SET=7
```

Assembly language

```

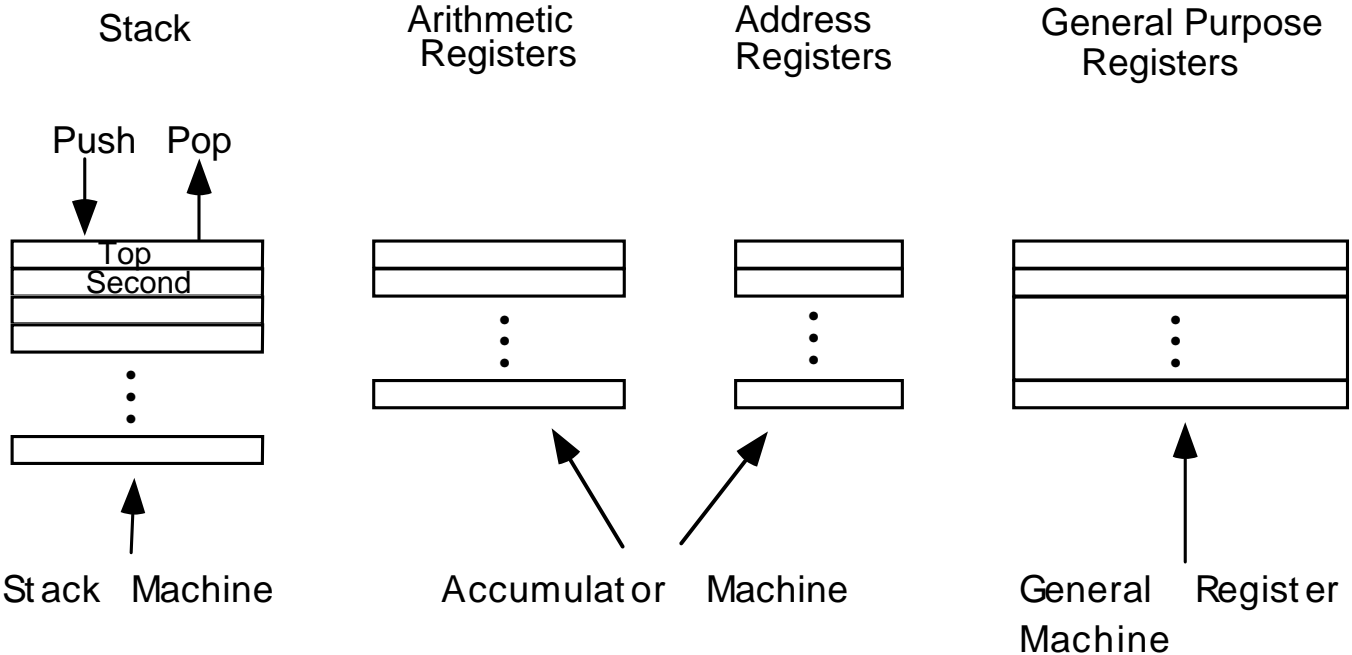
CMP.W #5, NUM ;the comparison
BNE L1 ;conditional branch
MOV.W #7, SET ;action if true
L1 ... ;action if false

```



# 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



C

S

D

A

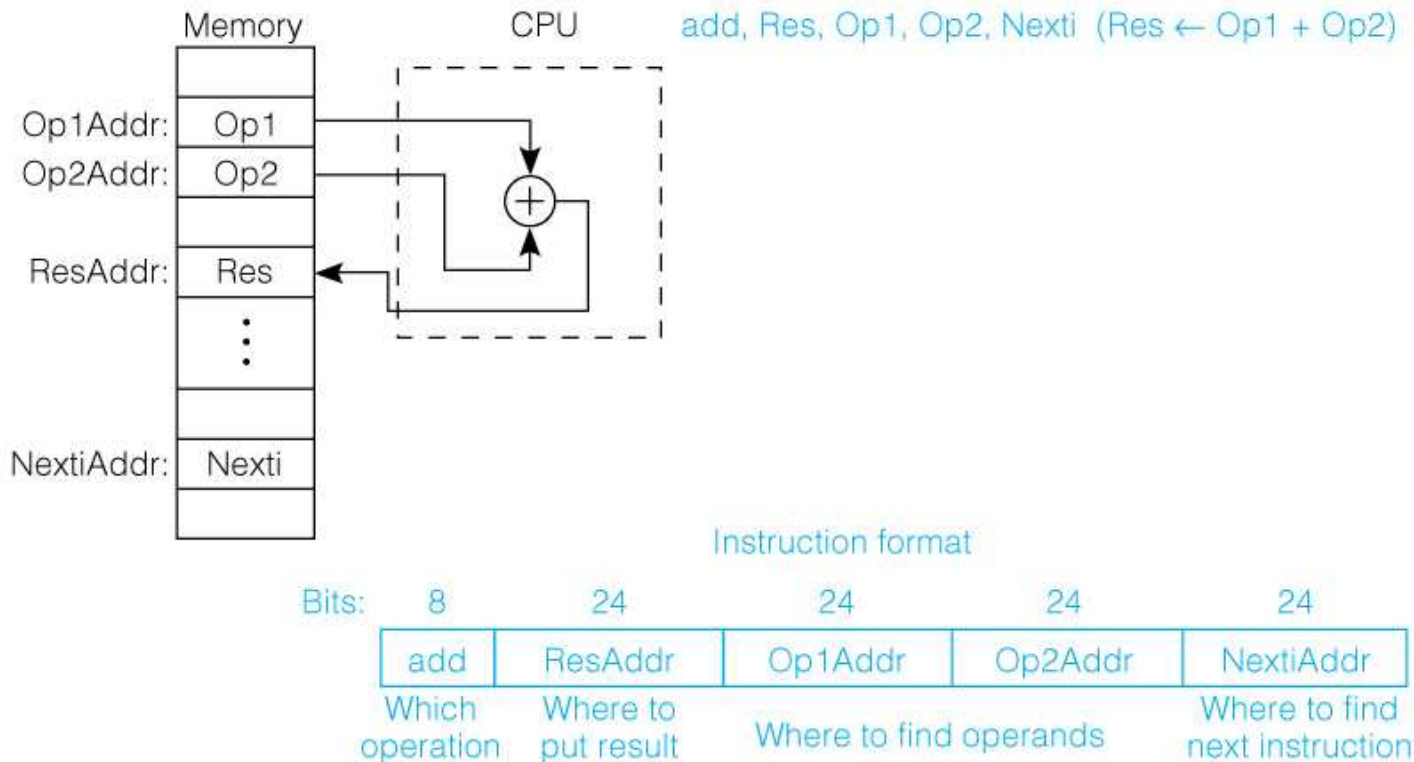
2/e

## 3, 2, 1, & 0 Address Instructions

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

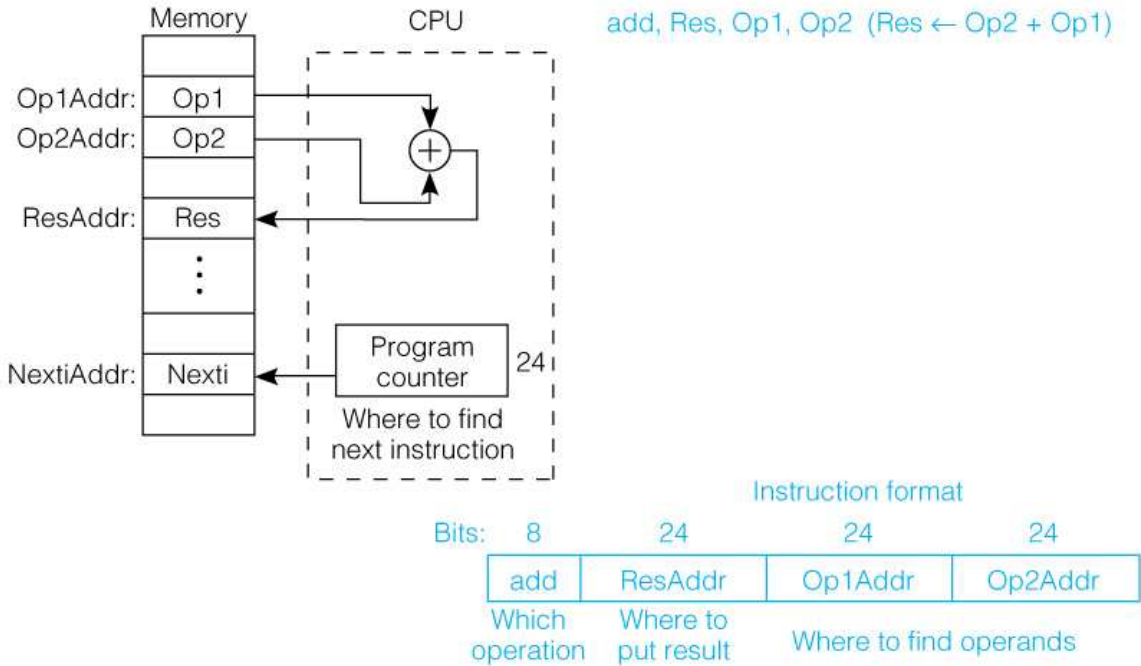
- The key issue is “how many of these are specified by memory addresses, as opposed to being specified implicitly”
- A 3 address instruction specifies memory addresses for both operands and the result:  $R \leftarrow \text{Op1 op Op2}$
- A 2 address instruction overwrites one operand in memory with the result:  $\text{Op2} \leftarrow \text{Op1 op Op2}$
- A 1 address instruction has a register, called the **accumulator register** to hold one operand & the result (no address needed):  
 $\text{Acc} \leftarrow \text{Acc op Op1}$
- A 0 address + uses a CPU register stack to hold both operands and the result:  $\text{TOS} \leftarrow \text{TOS op SOS}$  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



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

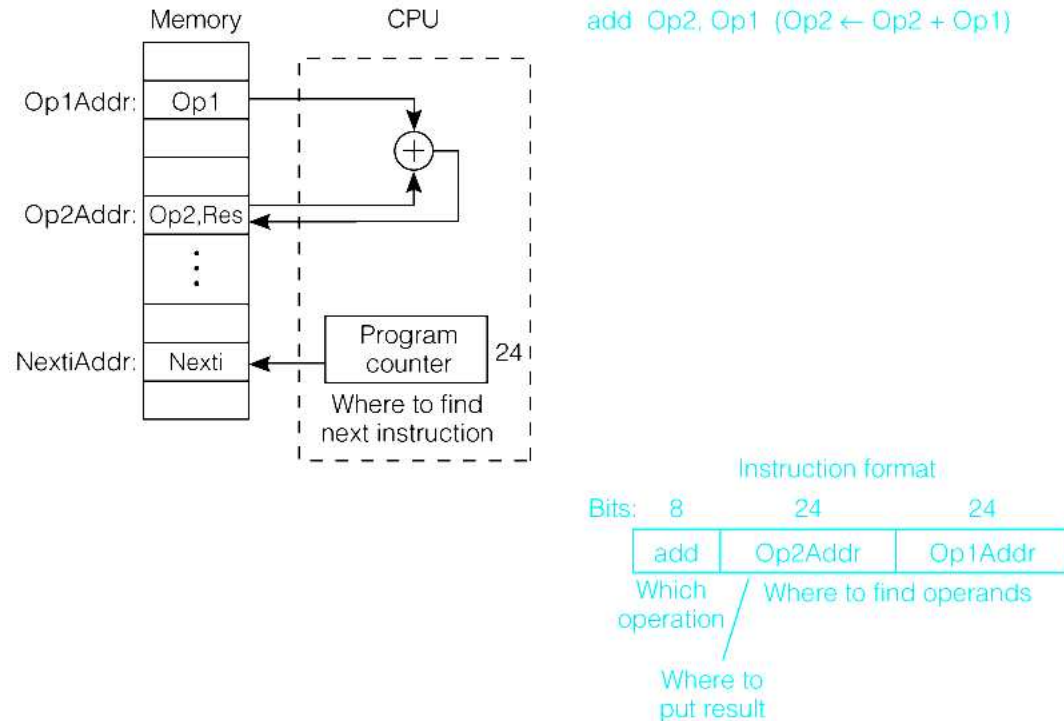
# Fig 2.4 The 3 Address Instruction



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

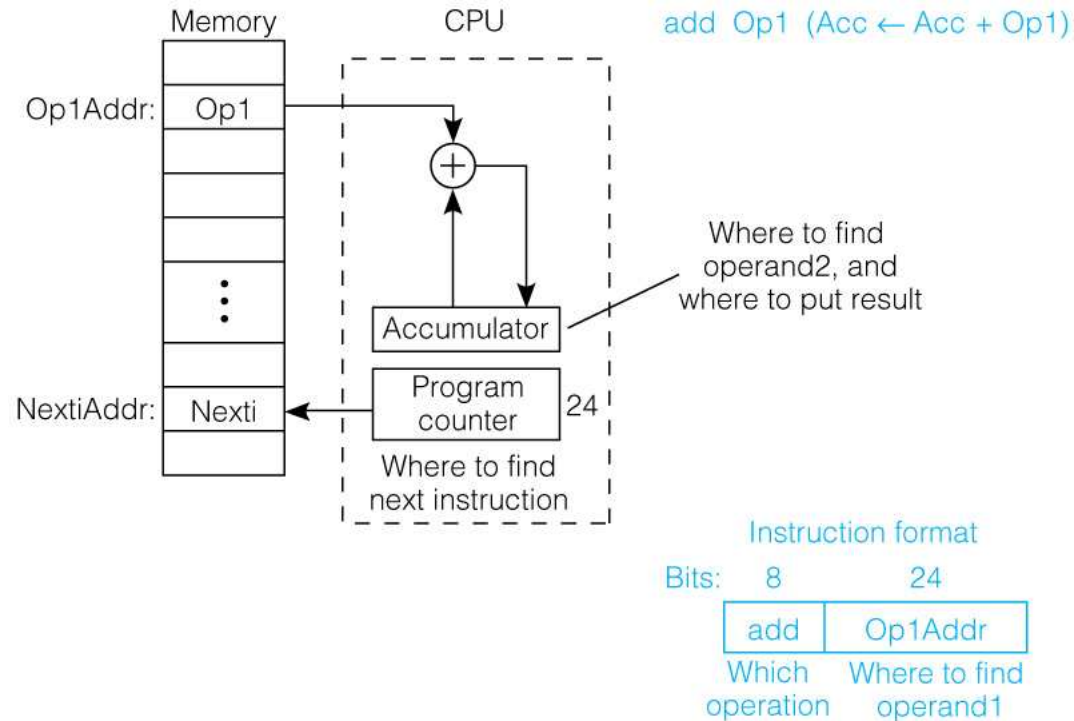


Copyright © 2004 Pearson Prentice Hall, Inc.

- 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

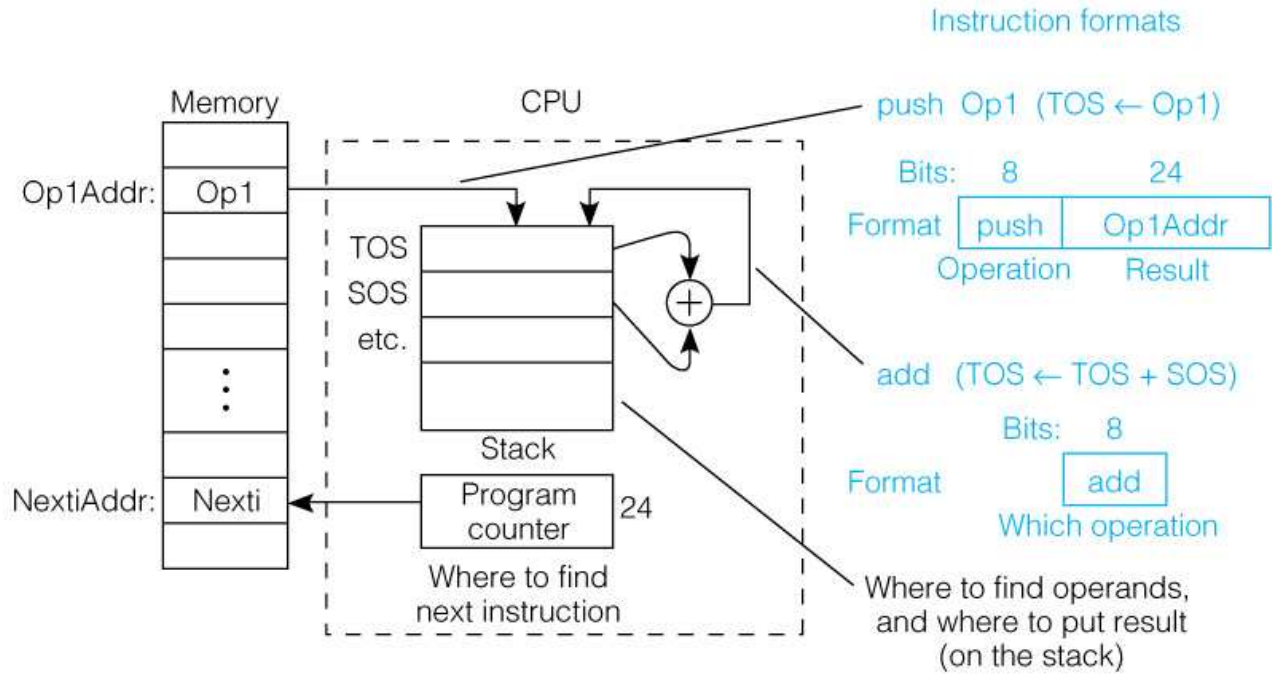
We now need instructions to load and store operands:  
LDA OpAddr  
STA OpAddr



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



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.

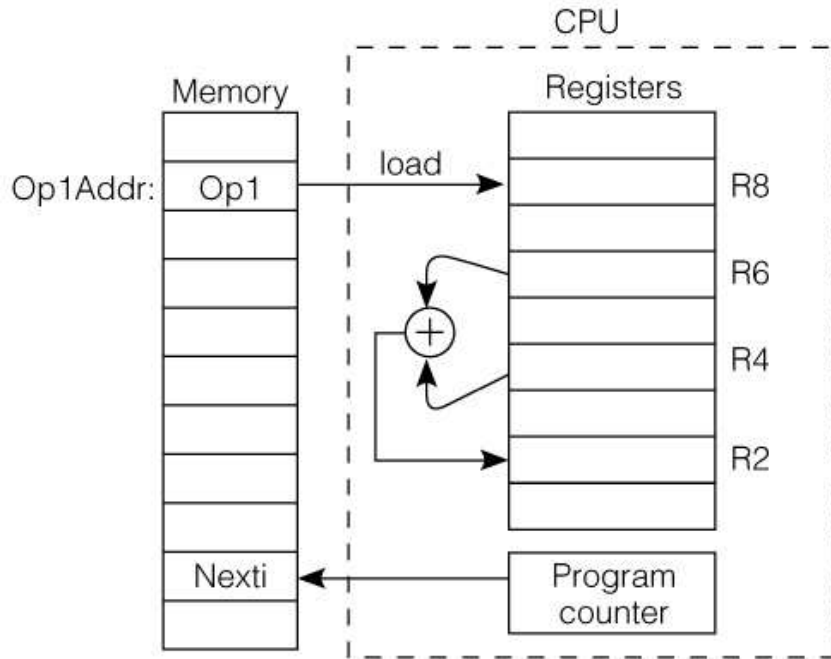
Evaluate  $a = (b+c)*d-e$  for 3- 2- 1- and 0-address machines.

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

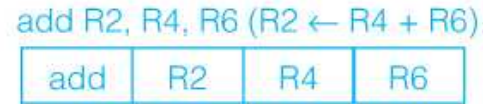
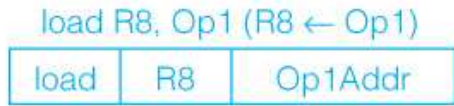
- # of instructions & # of addresses both vary
- Discuss as examples: size of code in each case



# Fig. 2.8 General Register Machines



### Instruction formats



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

C

S

D

A

2/e

## 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.

C

S

D

A

2/e

# Addressing modes are simpler than you think

---

- Operand types (only three basic types)
  - Register specifier (which register)
  - Constant that is part of the instruction
  - Memory contents
- Addressing mode types
  - Register direct
  - Base plus displacement
  - Immediate
  - All others are combinations or degenerate varieties
    - Relative addressing uses PC as a base register
    - Indirect is based with zero displacement
    - Autoincrement/autodecrement is a combined indirect plus increment/decrement

C

S

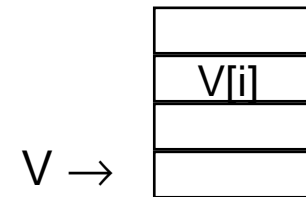
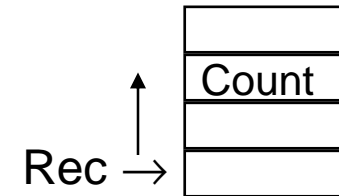
D

A

2/e

## HLL Examples of Structured Addresses

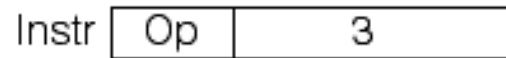
- C language: `rec -> count`
  - `rec` is a pointer to a record: full address variable
  - `count` is a field name: fixed byte offset, say 24
- C language: `v[i]`
  - `v` is fixed base address of array: full address constant
  - `i` is name of variable index: no larger than array size
- Variables must 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`.



# Fig 2.9 Common Addressing Modes a-d

**(a) Immediate addressing:**

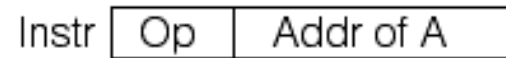
instruction contains the operand



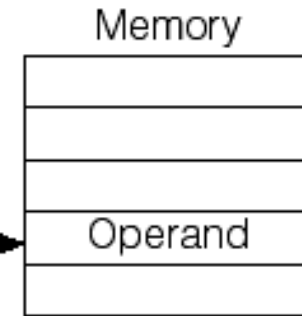
load #3, ...

**(b) Direct addressing:**

instruction contains address of operand



load A, ...



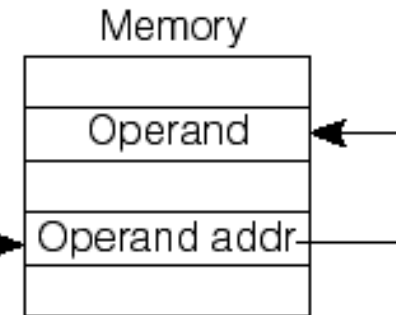
**(c) Indirect addressing:**

instruction contains address of address of operand

Address of address of A



load (A), ...



**(d) Register direct addressing:**

register contains operand



load R1, ...

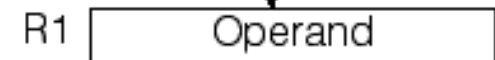
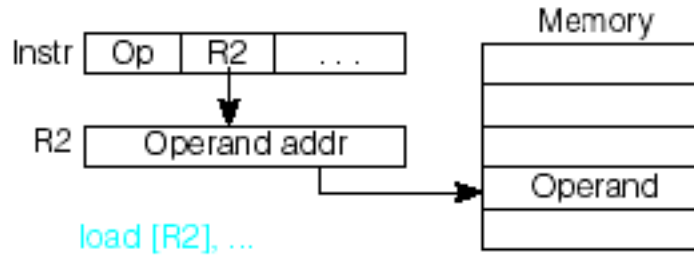


Fig 2.9 Common Addressing Modes e-g

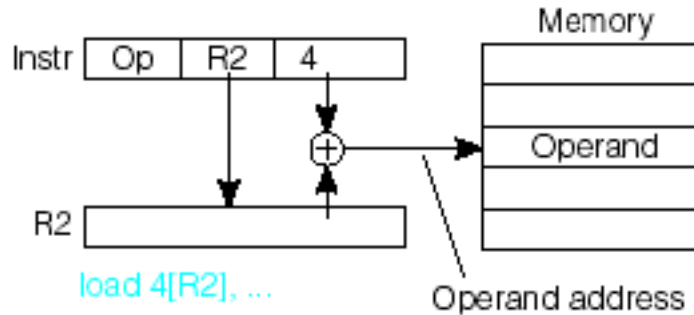
**(e) Register indirect addressing:**

register contains address of operand



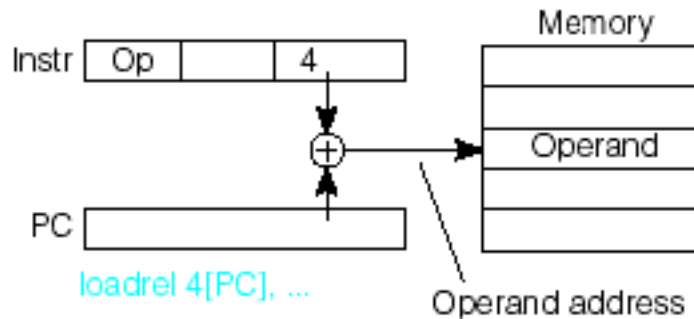
**(f) Displacement (based or indexed) addressing:**

address of operand = register + constant



**(g) Relative addressing:**

address of operand = PC + constant



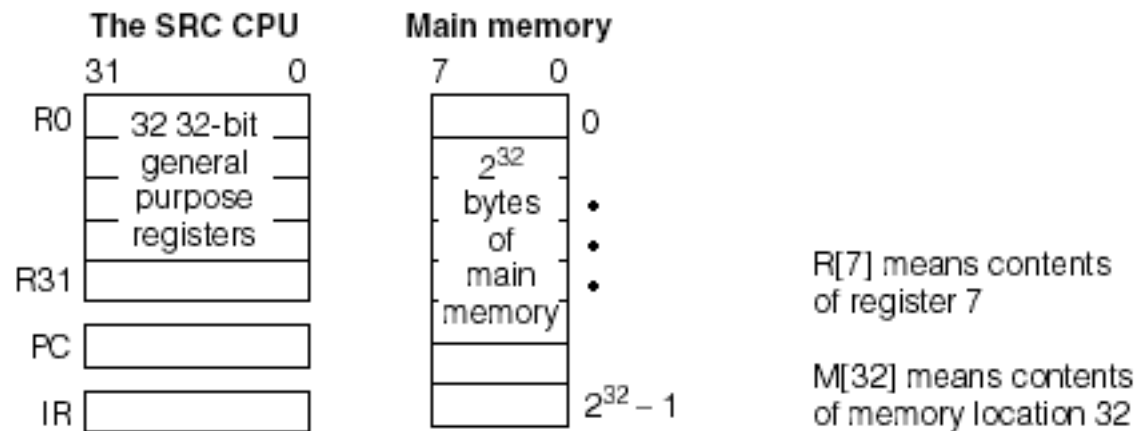
# Many addressing modes are really the same mode

- The base plus displacement mode includes many others
  - The base register can also be the zero register or the PC
  - PC-based causes relative addressing
  - Zero-based is absolute – refers only to base page because displacement is small
  - Based-indexed (8086 style, two registers) can be done by an LA, followed by a based-mode
  - Autoincrement modes require additional steps and are used in CISC machines only
  - The address computation step in the processor is the same for zero-register, relative, and based



# 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



C

S

D

A

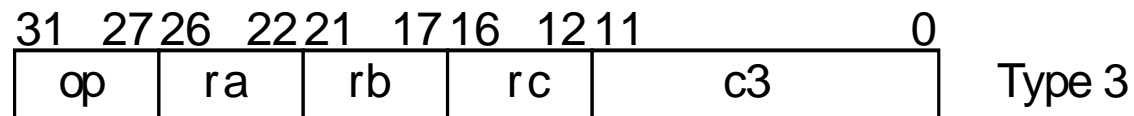
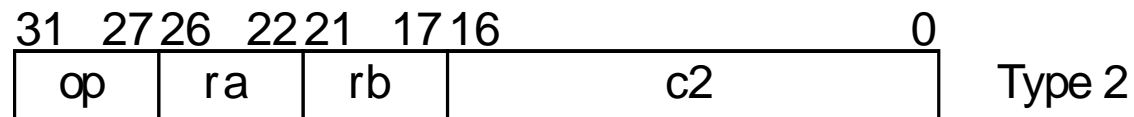
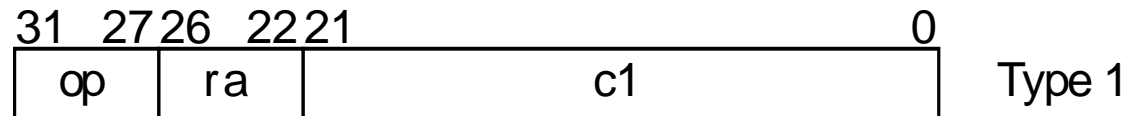
2/e

## 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



• **Details of formats:**

# The repertoire

---

- The instruction set is inside the back cover of the text
- Instruction types
  - Arithmetic/logic
    - Arithmetic (add/sub) (3-register/immediate)
    - Logic (and/or) (3-register/immediate)
    - Shift/rotate (left/logical right/arithmetic right/circular) (count in register/immediate)
  - Load/store
    - Load (base or relative)
    - Load address (base or relative)
    - Store (base or relative)

C

S

D

A

2/e

## The repertoire (more instruction classes)

- Conditional branch
  - Branch on cc (condition code)
  - Branch on condition and link
- Specials
  - Related to interrupt system
  - Nop
  - Stop
- Addressing modes
  - Only one mode – base and displacement
    - This mode encompasses relative and absolute
    - Address = contents of base register + displacement
    - For relative, PC is the base register
    - For absolute, register zero, which is always zero is the base
    - Normally a general register is the base

C

S

D

A

2/e

# Arithmetic and logic instructions

- Unary operations
  - Both have the same format  $op\ ra,rb$  means  $(ra) = op\ rb$
  - Operations are not and neg
- Binary operations
  - Two formats
    - Register – three register operands  
 $op\ ra, rb, rc$  means  $(ra) = rb\ op\ rc$
    - Immediate – two register operands and a number  
 $op\ ra, rb,n$  means  $(ra) = rb\ op\ n$
  - Operations are add, sub, and, or
    - Immediate versions are addi, andi, ori
    - Subi is done by addi with a negative constant
  - Note that all ALU instructions work with registers only, not memory

# Shift instructions

- Operations are:
  - left shift (a zero is shifted in at right)
  - logical right shift (a zero is shifted in at left)
  - arithmetic right shift (the sign bit is shifted in at left)
  - left circular shift (the sign bit is shifted in at right)
- Number of places to shift can be either in a fourth register or an immediate constant (field in the instruction register)

C

S

D

A

2/e

## Load/store instructions

- Only three instructions and one addressing mode
  - ld and ldr
  - st and str
  - la and lar – like ld and ldr but:
    - the address is placed in a register rather than being used for a memory cycle
    - la is really the same as the addi instruction
- Addressing modes
  - Base and displacement is n(rb)
  - Relative is n(PC), coded as a label, assembler calculates n the mnemonic is ldr, str, or lar
  - Absolute is n(r0)
  - Indirect is 0(rb)
- Assemblers often calculate the base and displacement values for you from program or data labels



C

S

D

A

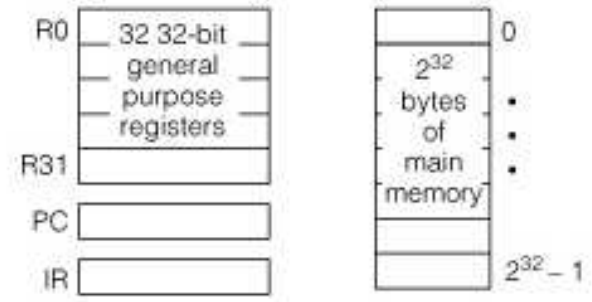
2/e

## Branch (including branch and link) instructions

- The only two branch instructions are:
  - br?? branch on condition code
  - brl?? branch on condition code and save PC in link
  - Condition codes are
    - (blank) always, antonym nv – never
    - Zr zero, antonym nz – nonzero
    - mi minus, antonym pl – plus or zero
  - Condition code is based on register rc of the instruction
  - Destination is address in rb, almost always placed there by a preceding lar instruction
  - Return address is placed in ra if the instruction is a brl
- Examples
  - brlnz ra, rb, rc – branch to address in rb if rc is not zero, store return address in ra
  - br rb – unconditional branch to address in rb , rc not needed
  - brl rb – unconditional gosub to address in rb , return address in ra, rc not needed
  - brlmi rb –conditional gosub to address in rb , return address in ra, if rc is minus

C  
S  
D  
A  
2/e

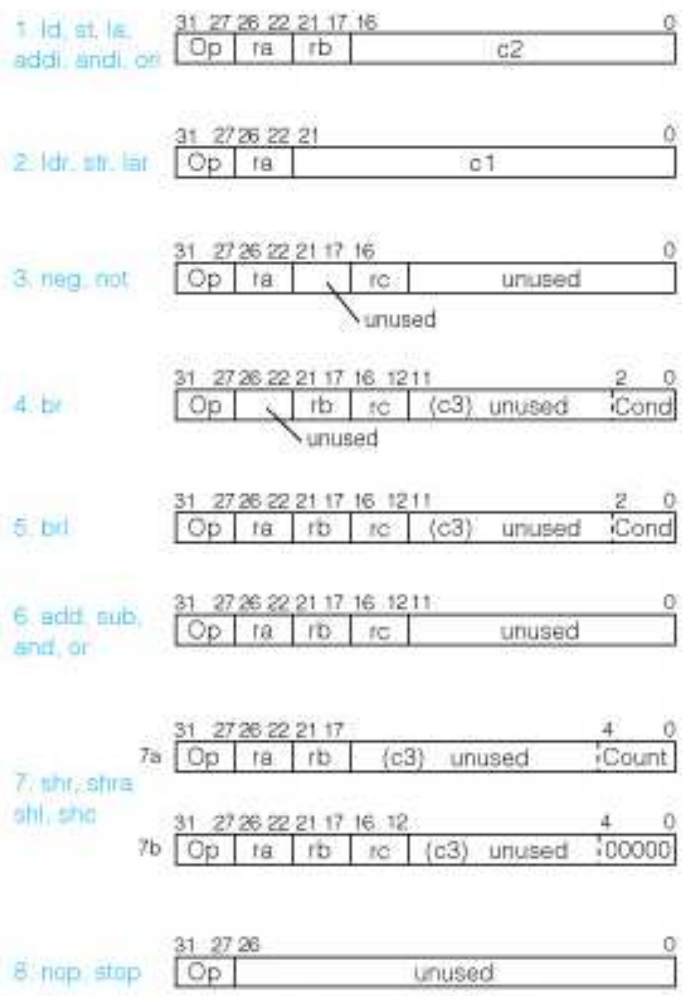
Fig 2.10 cont'd. SR( instructions



R[7] means contents of register 7

M[32] means contents of memory location 32

Instruction formats



Example

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)

ldr r5, 8      (R[5] = M[PC + 8])  
lsr r6, 45      (R[6] = PC + 45)

neg r7, r9      (R[7] = -R[9])

brz r4, r0      (branch to R[4] if R[0] == 0)

brnz r6, r4, r0    (R[6] = PC; branch to R[4] if R[0] ≠ 0)

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

shr r0, r1, 4      (R[0] = R[1] shifted right by 4 bits)

shl r2, r4, r6      (R[2] = R[4] shifted left by count in R[6])

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)**

C

S

D

A

2/e

# Assembly Language Forms of Arithmetic and Logic Instructions

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

C

S

D

A

## Branch Instruction Format

2/ 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:

<u>lsbs</u>	<u>condition</u>	<u>Assy language form</u>	<u>Example</u>
000	never	<code>brInv</code>	<code>brInv r6</code>
001	always	<code>br, brl</code>	<code>br r5, brl r5</code>
010	if <code>rc = 0</code>	<code>brzr, brl zr</code>	<code>brzr r2, r4</code>
011	if <code>rc ≠ 0</code>	<code>brnz, brlnz</code>	
100	if <code>rc ≥ 0</code>	<code>brpl, brlpl</code>	
101	if <code>rc &lt; 0</code>	<code>brmi, brlmi</code>	

- 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	$R[6] \leftarrow PC$	9	6	—	—	000	never
br	br r4	$PC \leftarrow R[4]$	8	—	4	—	001	always
brl	brl r6,r4	$R[6] \leftarrow PC;$ $PC \leftarrow R[4]$	9	6	4	—	001	always
brzr	brzr r5,r1	if (R[1]=0) $PC \leftarrow R[5]$	8	—	5	1	010	zero
brlzt	brlzt r7,r5,r1	$R[7] \leftarrow PC;$	9	7	5	1	010	zero
brnz	brnz r1, r0	if (R[0]≠0) $PC \leftarrow R[1]$	8	—	1	0	011	nonzero
brlnzt	brlnzt r2,r1,r0	$R[2] \leftarrow PC;$ if (R[0]≠0) $PC \leftarrow R[1]$	9	2	1	0	011	nonzero
brpl	brpl r3, r2	if (R[2]□0) $PC \leftarrow R[3]$	8	—	3	2	100	plus
brlpl	brlpl r4,r3,r2	$R[4] \leftarrow PC;$ if (R[2]□0) $PC \leftarrow R[3]$	9	4	3	2		plus
brmi	brmi r0, r1	if (R[1]<0) $PC \leftarrow R[0]$	8	—	0	1	101	minus
brlmi	brlmi r3,r0,r1	$R[3] \leftarrow PC;$ if (r1<0) $PC \leftarrow R[0]$	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 . . .

C

S

D

A

2/e

## Example of conditional branch

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
```



C

S

D

A

2/e

## 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)

C

S

D

A

2/e

## 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

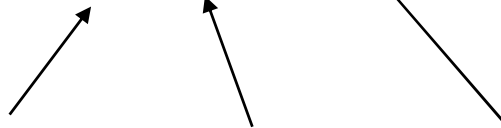
- Specifying registers
  - $IR\langle 31..0 \rangle$  specifies a register named “IR” having 32 bits numbered 31 to 0
- “Naming” using the := naming operator:
  - $op\langle 4..0 \rangle := IR\langle 31..27 \rangle$  specifies that the 5 msbs of IR be called op, with bits 4..0.
  - Notice that this does not create a new register, it just generates another name, or “alias” for an already existing register or part of a register.

# Using RTN to describe Dynamic Properties

## Dynamic Properties

- Conditional expressions:

$(op=12) \rightarrow R[ra] \leftarrow R[rb] + R[rc];$  ; 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

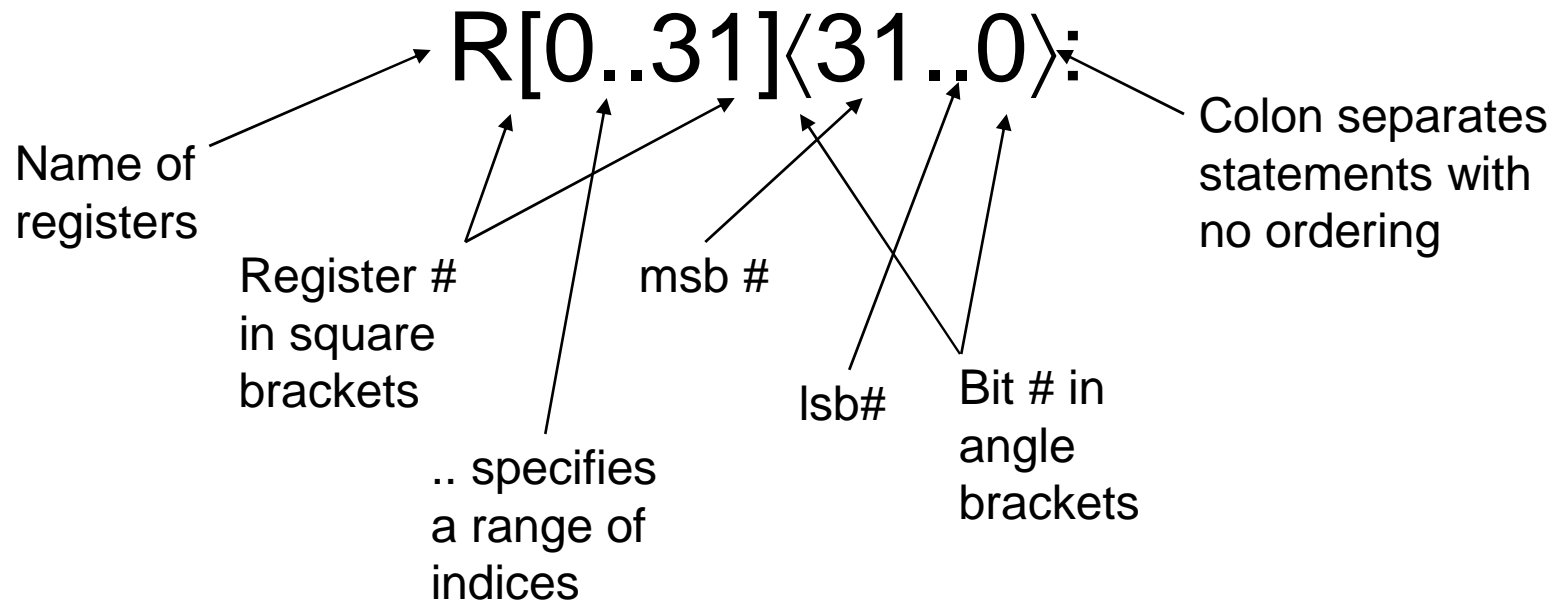
---

## Processor state

- PC $\langle 31..0 \rangle$ : program counter  
(memory addr. of next inst.)
- IR $\langle 31..0 \rangle$ : instruction register
- Run: one bit run/halt indicator
- Strt: start signal
- R[0..31] $\langle 31..0 \rangle$ : 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]



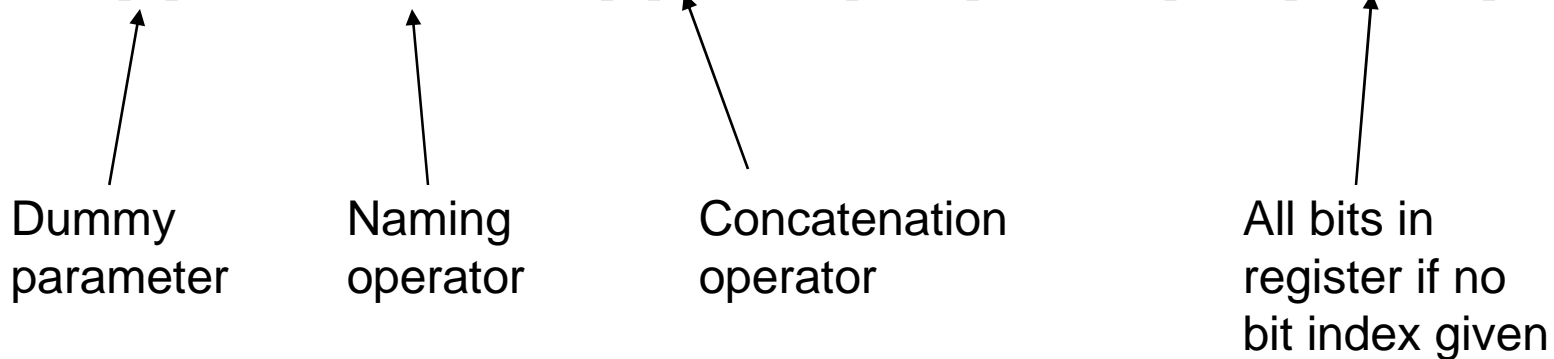
# Memory Declaration: RTN Naming Operator

- 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

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





# 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



C

S

D

A

2/e

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

Effective address calculations (occur at runtime):

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

- Renaming defines displacement and relative addrs.
- New RTN notation is used
  - $\text{condition} \rightarrow \text{expression}$  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

# Detailed Questions Answered by the RTN for Addresses

- 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 := (  
 $\neg$ Run  $\wedge$  Strt  $\rightarrow$  Run  $\leftarrow$  1:  
 Run  $\rightarrow$  (IR  $\leftarrow$  M[PC]; PC  $\leftarrow$  PC + 4; instruction\_execution) );

Register transfer

Separates statements that occur in sequence

C

S

D

A

2/e

## 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

C

S

D

A

2/e

## Synchronous vs. asynchronous logic

- HLL's (imperative languages) describe sequential systems because they operate with a single thread of control
- Real systems are multithreaded
- Simulation languages (Spice, Matlab, VHDL) can describe such dynamical systems
- If you describe a processor in an imperative language (C family) you have to adapt your description to fit the sequential nature of the language
- LogicWorks is really a hybrid
- RTN describes only clocked logic – each clock cycle can include several noninteracting operations active in the same cycle
- The synchronization operators are (:, simultaneous but independent) and (;, sequential)

C

S

D

A

2/e

## More about Instruction Interpretation RTN

- In the expression  $IR \leftarrow M[PC]: PC \leftarrow PC + 4;$  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

C

S

D

A

2/e

## 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.

# RTN Instruction Execution for Load and Store Instructions

```

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



# SRC RTN—The Main Loop

```

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]:
    ldr (:= op= 2) → R[ra] ← M[rel]:
    ...
    stop (:= op= 31) → Run ← 0:
); ii

```

Big switch  
statement  
on the opcode

Thus ii and ie invoke each other, as coroutines.  
*This pair actually defines the fetch-execute cycle; running until a stop instruction halts the processor*

# 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 00000000000001011

- 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.

cond := ( c3<2..0>=0 → 0:	never
c3<2..0>=1 → 1:	always
c3<2..0>=2 → R[rc]=0:	if register is zero
c3<2..0>=3 → R[rc]≠0:	if register is nonzero
c3<2..0>=4 → R[rc]<31>=0:	if positive or zero
c3<2..0>=5 → R[rc]<31>=1 ):	if negative
br (:= op= 8) → (cond → PC ← R[rb]):	conditional branch
brl (:= op= 9) → (R[ra] ← PC: cond → (PC ← R[rb])) ):	branch and link

C

S

D

A

2/e

## RTN for Arithmetic and Logic

add (:= op=12)  $\rightarrow R[ra] \leftarrow R[rb] + R[rc]$ :

addi (:= op=13)  $\rightarrow R[ra] \leftarrow R[rb] + c2\langle 16..0 \rangle$  {2's comp. sign ext.}:

sub (:= op=14)  $\rightarrow R[ra] \leftarrow R[rb] - R[rc]$ :

neg (:= op=15)  $\rightarrow R[ra] \leftarrow -R[rc]$ :

and (:= op=20)  $\rightarrow R[ra] \leftarrow R[rb] \wedge R[rc]$ :

andi (:= op=21)  $\rightarrow R[ra] \leftarrow R[rb] \wedge c2\langle 16..0 \rangle$  {sign extend}:

or (:= op=22)  $\rightarrow R[ra] \leftarrow R[rb] \vee R[rc]$ :

ori (:= op=23)  $\rightarrow R[ra] \leftarrow R[rb] \vee c2\langle 16..0 \rangle$  {sign extend}:

not (:= op=24)  $\rightarrow R[ra] \leftarrow \neg R[rc]$ :

- Logical operators: and  $\wedge$  or  $\vee$  and not  $\neg$

## RTN for Shift Instructions

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

$$n := ( \quad (c3\langle 4..0 \rangle = 0) \rightarrow R[rc]\langle 4..0 \rangle : \\ (c3\langle 4..0 \rangle \neq 0) \rightarrow c3\langle 4..0 \rangle ) :$$
$$\text{shr} (:= \text{op}=26) \rightarrow R[ra]\langle 31..0 \rangle \leftarrow (n @ 0) \# R[rb]\langle 31..n \rangle :$$
$$\text{shra} (:= \text{op}=27) \rightarrow R[ra]\langle 31..0 \rangle \leftarrow (n @ R[rb]\langle 31 \rangle) \# R[rb]\langle 31..n \rangle :$$
$$\text{shl} (:= \text{op}=28) \rightarrow R[ra]\langle 31..0 \rangle \leftarrow R[rb]\langle 31-n..0 \rangle \# (n @ 0) :$$
$$\text{shc} (:= \text{op}=29) \rightarrow R[ra]\langle 31..0 \rangle \leftarrow R[rb]\langle 31-n..0 \rangle \# R[rb]\langle 31..32-n \rangle :$$

# 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
```

C

S

D

A

2/e

## 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)



C

S

D

A

2/e

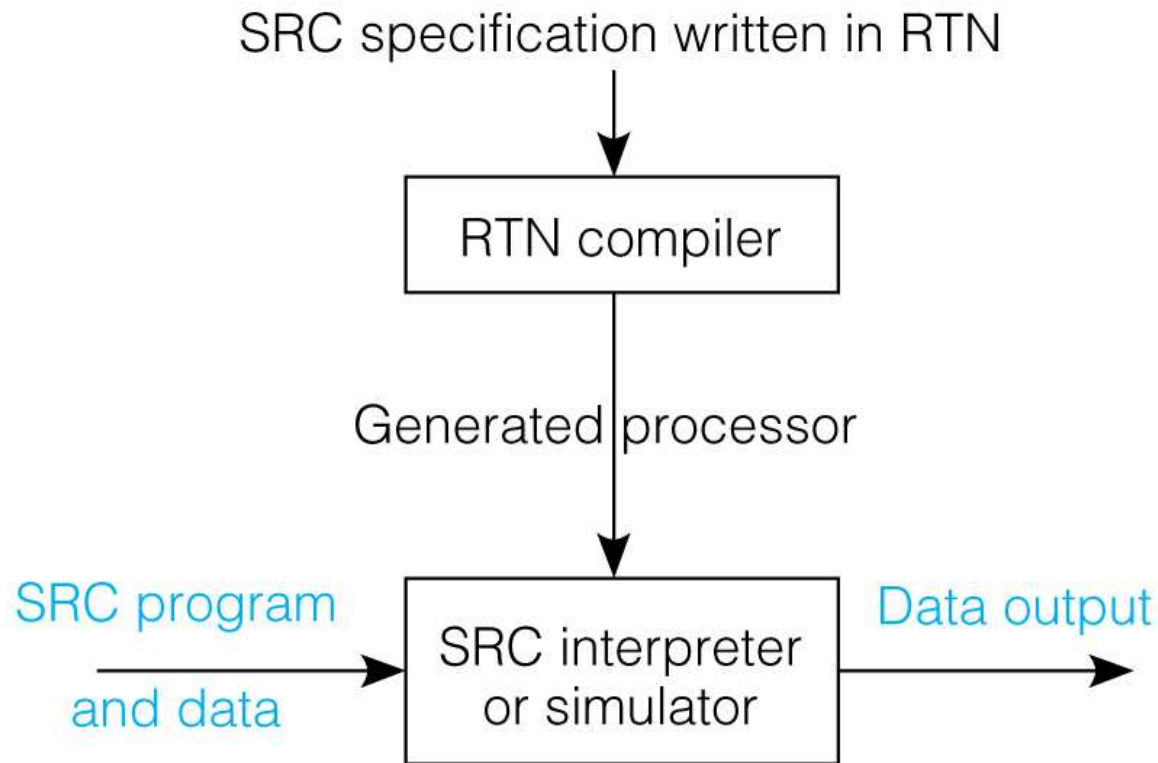
## Confused about RTN and SRC?

- 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...**

C  
S  
D  
A  
2/e

# Fig 2.11 The Relationship of RTN to SRC



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.

C

S

D

A

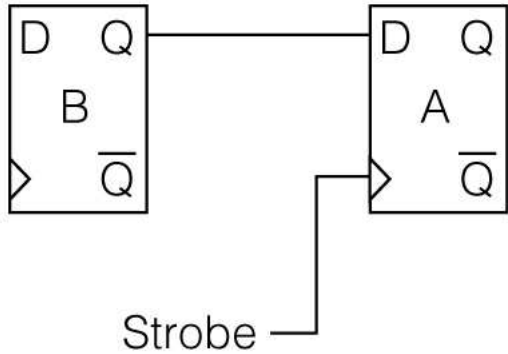
2/e

# Addressing Modes Described in RTN (Not SRC)

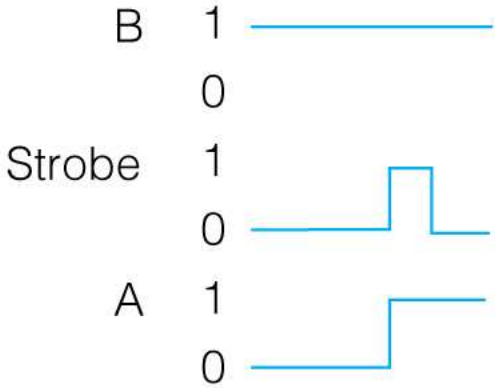
<u>Mode name</u>	<u>Assembler Syntax</u>	<u>RTN meaning</u>	<u>Use</u>
Register	Ra	$R[t] \leftarrow R[a]$	Target register Tmp. Var.
Register indirect	(Ra)	$R[t] \leftarrow M[R[a]]$	Pointer
Immediate	#X	$R[t] \leftarrow X$	Constant
Direct, absolute	X	$R[t] \leftarrow M[X]$	Global Var.
Indirect	(X)	$R[t]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

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

- Implementing the RTN statement  $A \leftarrow B$



**(a) Hardware**

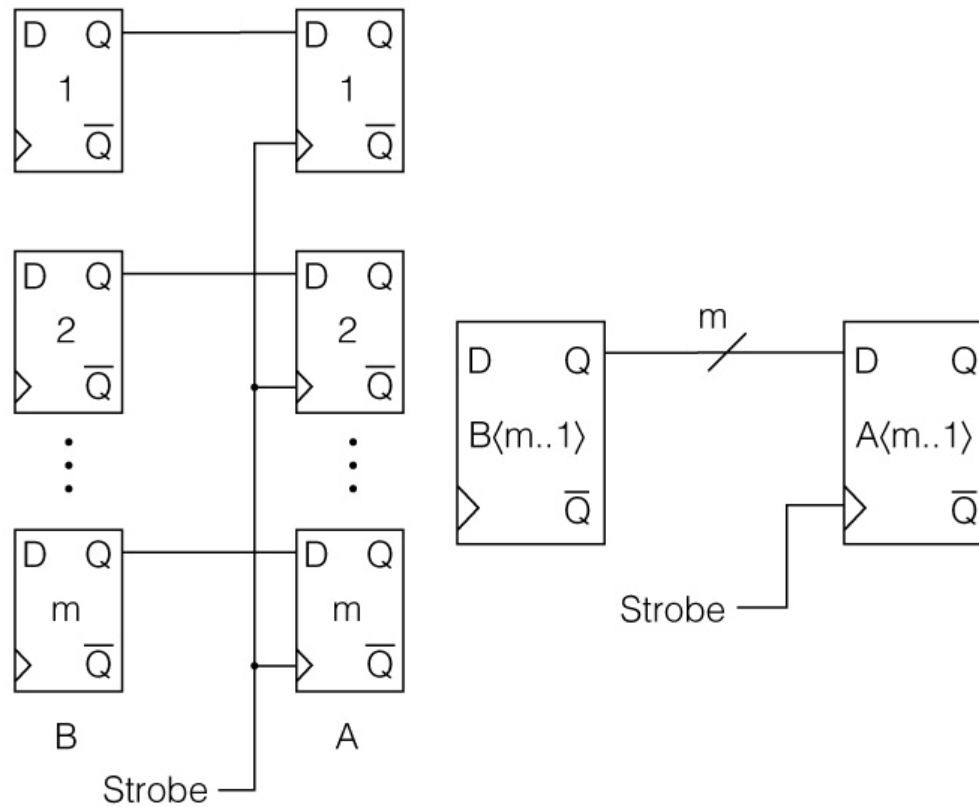


**(b) Timing**

Copyright © 2004 Pearson Prentice Hall, Inc.

# Fig. 2.13 Multiple Bit Register Transfer

- Implementing  $A\langle m..1 \rangle \leftarrow B\langle m..1 \rangle$



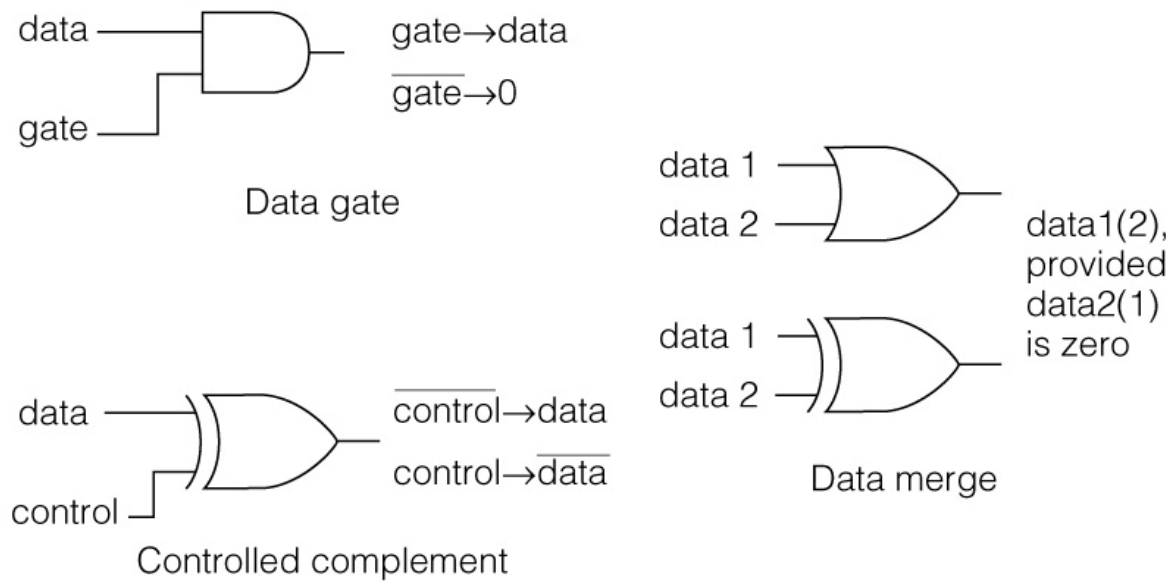
**(a) Individual flip-flops**

**(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:



C

S

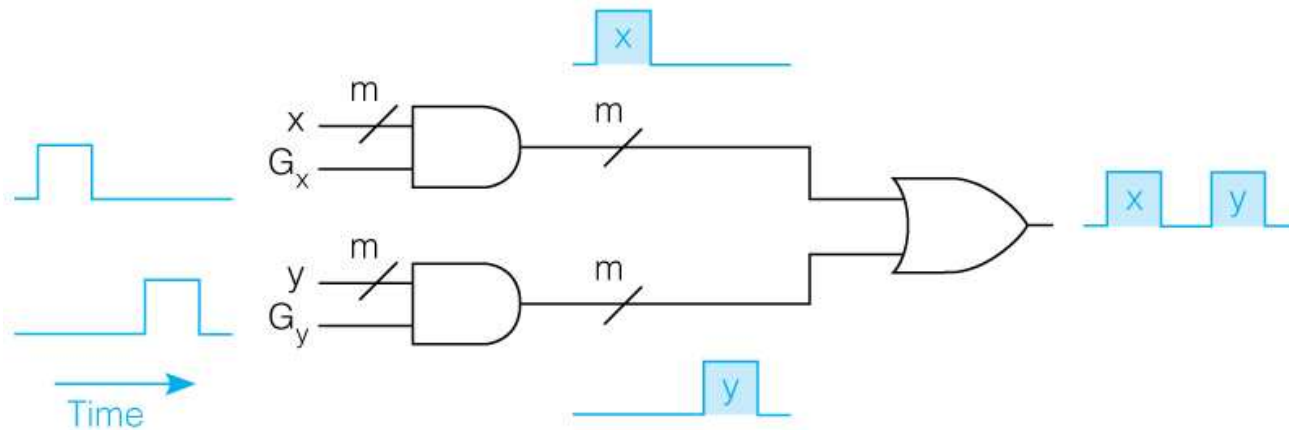
D

A

2/e

## Fig. 2.15 Multiplexer as a 2 Way Gated Merge

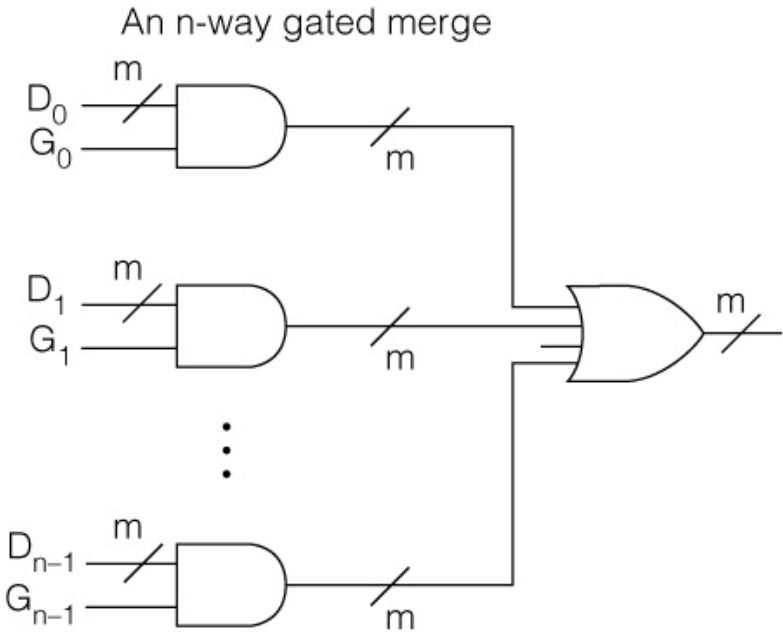
- Data from multiple sources can be selected for transmission



Copyright © 2004 Pearson Prentice Hall, Inc.

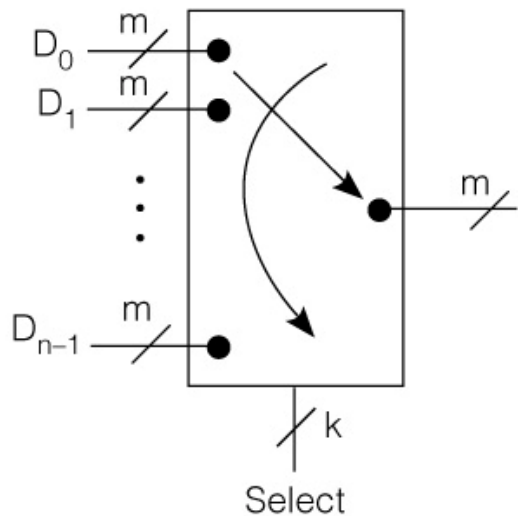


# Fig. 2.16 m-bit Multiplexer and Symbol



**(a) Multiplexer in terms of gates**

An n-way multiplexer with decoder

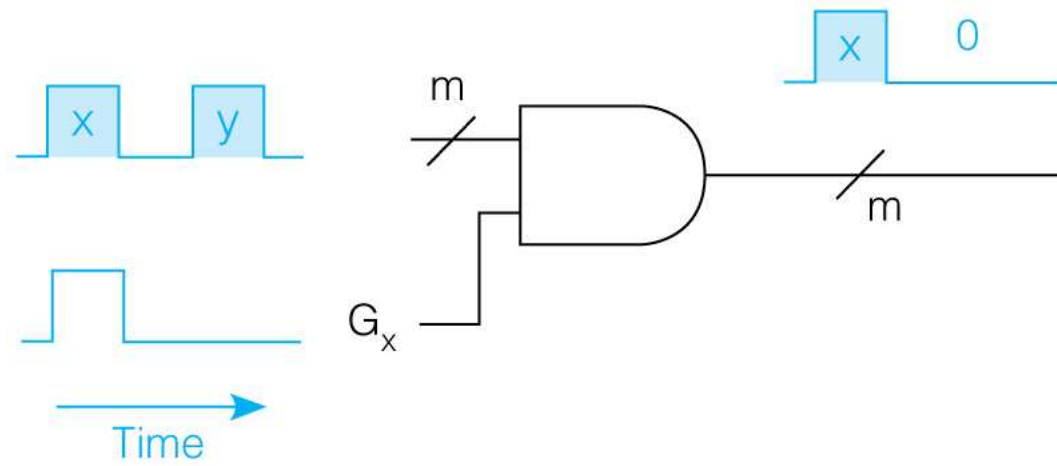


**(b) Symbol abbreviation**

Copyright © 2004 Pearson Prentice Hall, Inc.

- Multiplexer gate signals  $G_i$  may be produced by a binary to one-out-of-n decoder

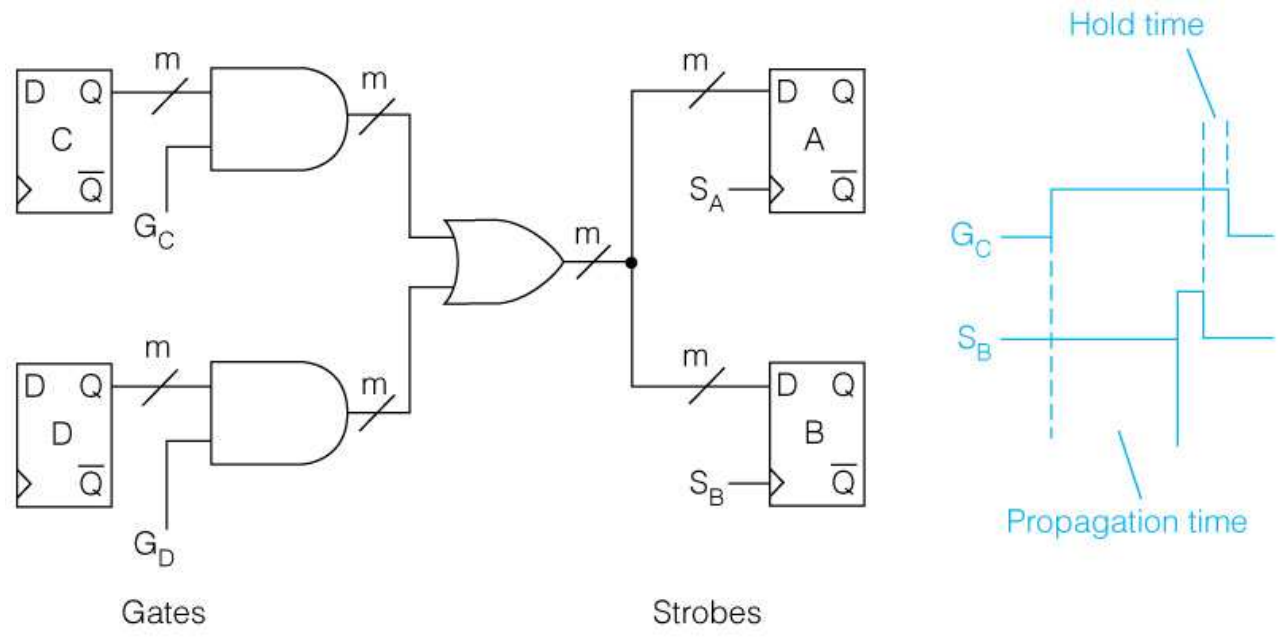
# Fig. 2.17 Separating Merged Data



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



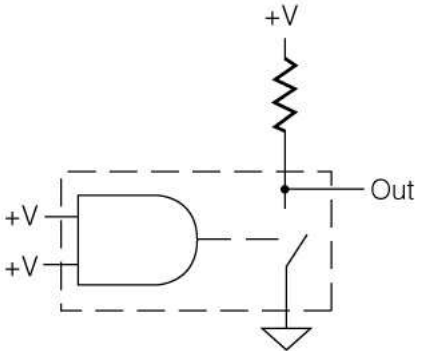
Copyright © 2004 Pearson Prentice Hall, Inc.

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

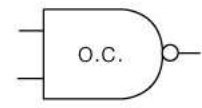
# Fig. 2.19 Open-Collector NAND Gate Output Circuit

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



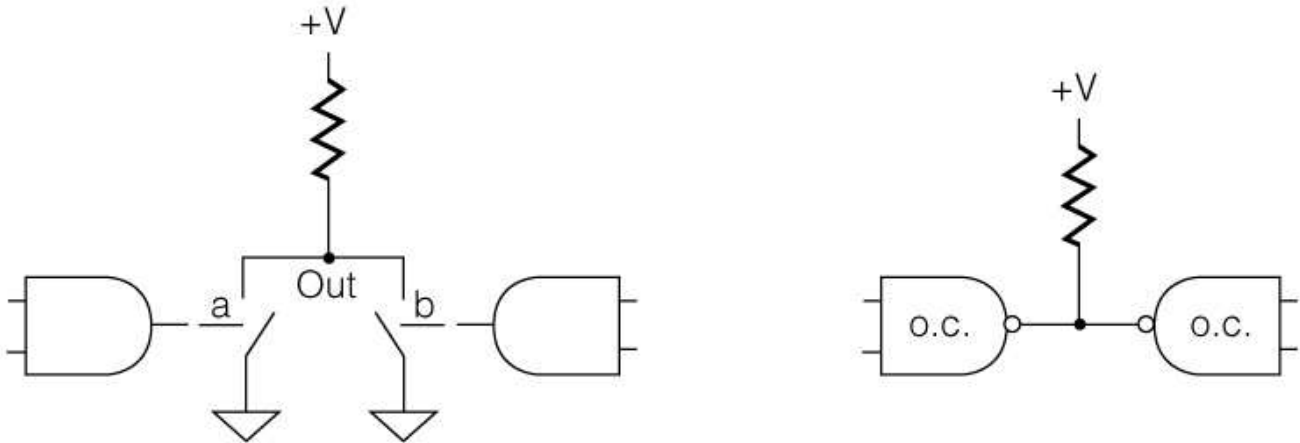
(b) Open-collector NAND



(c) Symbol

Copyright © 2004 Pearson Prentice Hall, Inc.

# Fig. 2.20 Wired AND Connection of Open-Collector Gates



(a) Wired AND connection

(b) With symbols

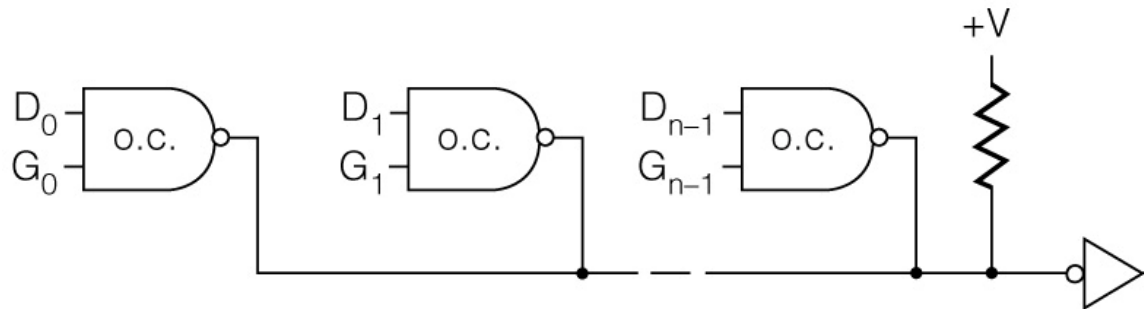
Switch		Wired AND output
a	b	
Closed(0)	Closed(0)	0v (0)
Closed(0)	Open (1)	0v (0)
Open (1)	Closed(0)	0v (0)
Open (1)	Open (1)	+V (1)

(c) Truth table

Copyright © 2004 Pearson Prentice Hall, Inc.

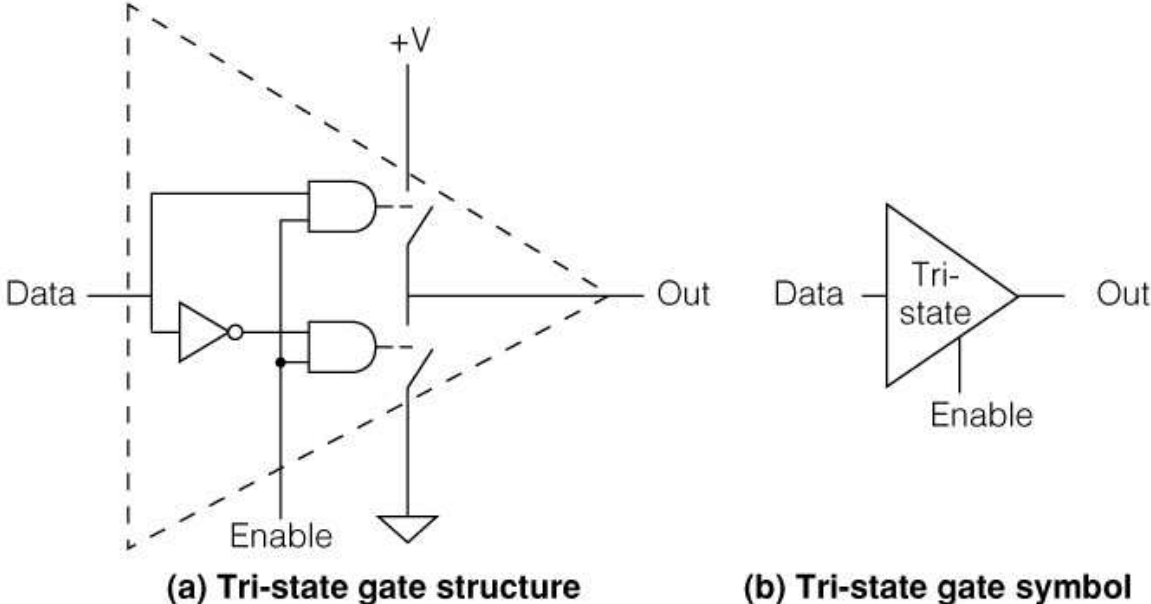
## Fig. 2.21 Open Collector Wired OR Bus

- 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



Copyright © 2004 Pearson Prentice Hall, Inc.

# Fig. 2.22 Tri-state Gate Internal Structure and Symbol

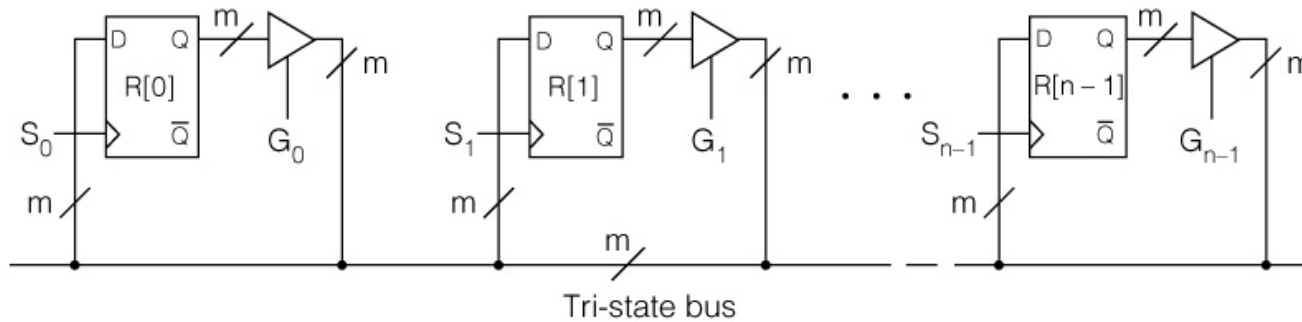


Enable	Data	Output
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(c) Tri-state gate truth table

Copyright © 2004 Pearson Prentice Hall, Inc.

# Fig. 2.23 Registers Connected by a Tri-state Bus



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



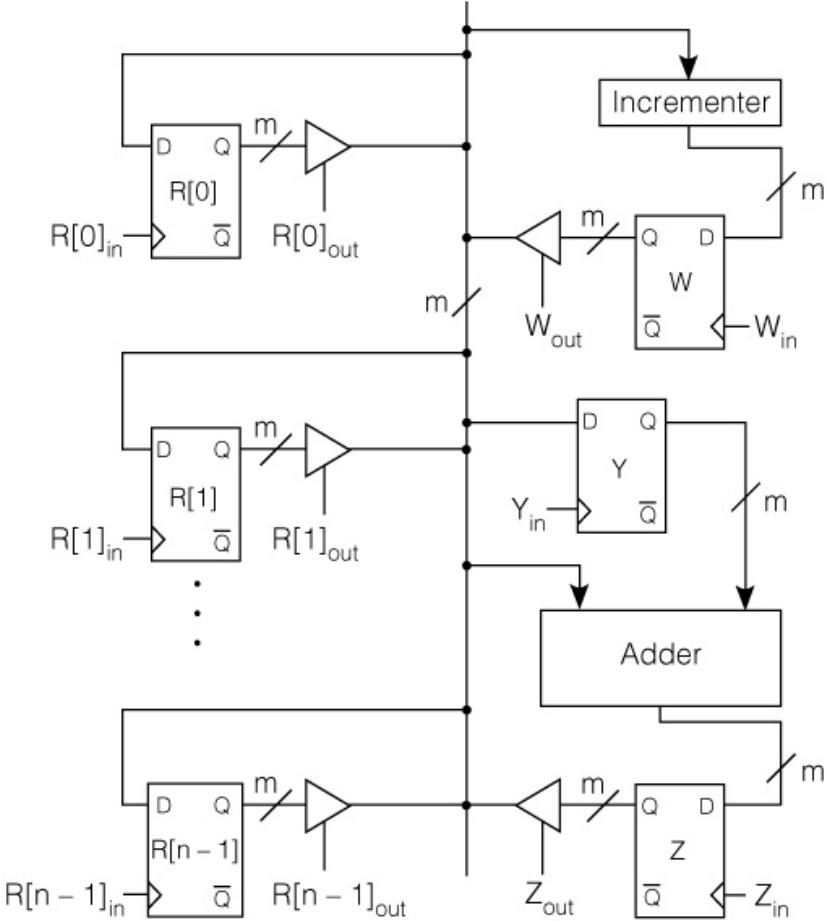
C  
S  
D  
A  
2/e

# Fig. 2.24 Registers and Arithmetic Connected by One Bus

**Example**  
Abstract RTN  
 $R[3] \leftarrow R[1]+R[2];$

Concrete RTN  
 $Y \leftarrow R[2];$   
 $Z \leftarrow R[1]+Y;$   
 $R[3] \leftarrow Z;$

Control Sequence  
 $R[2]_{out}, Y_{in};$   
 $R[1]_{out}, Z_{in};$   
 $Z_{out}, R[3]_{in};$



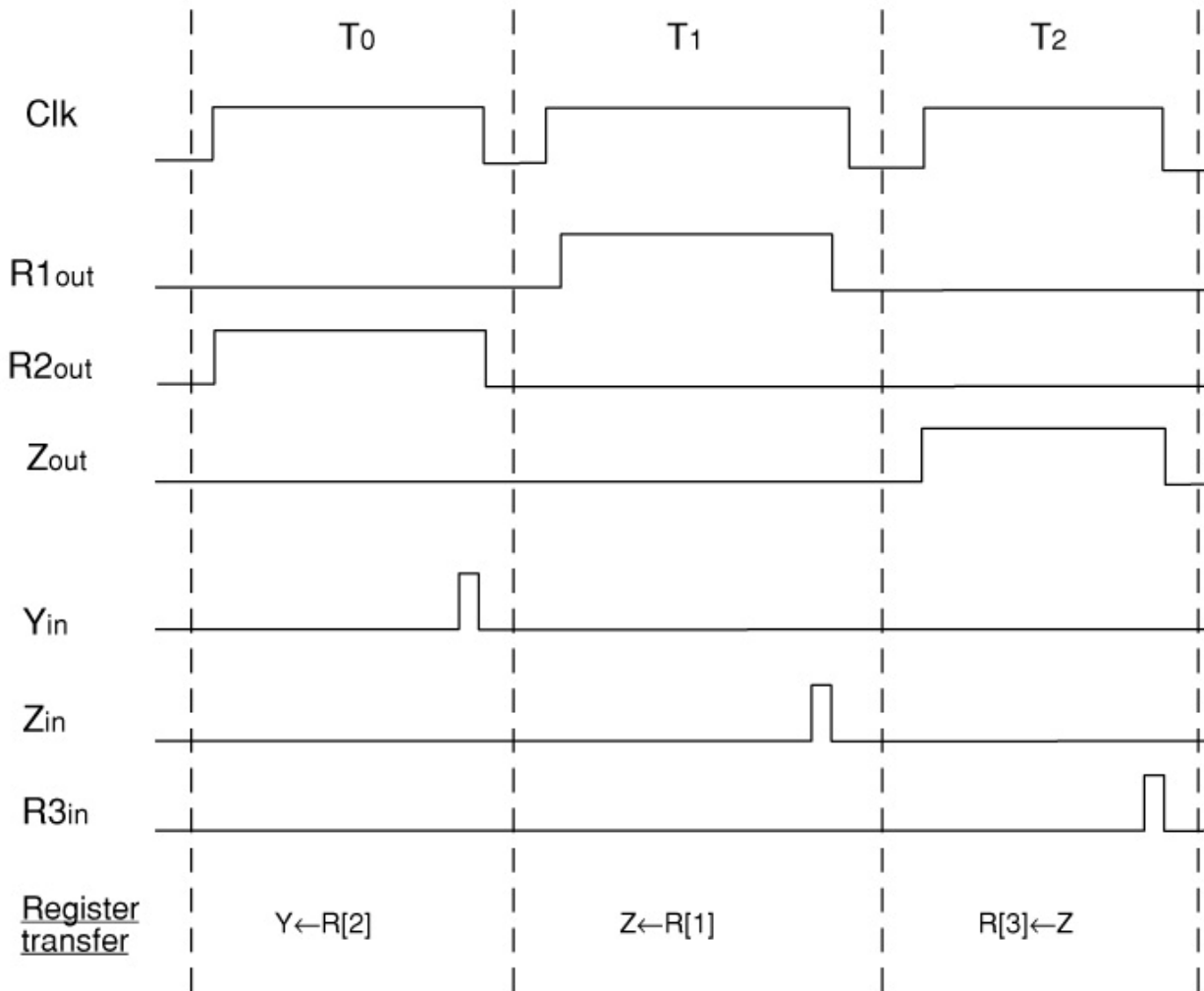
Combinational Logic—no memory

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.

C

S

D

A

2/e

## 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
- $R[i] \leftarrow R[j] + R[k]$  is performed in 3 steps:  $Y \leftarrow R[k]$ ;  $Z \leftarrow R[j] + Y$ ;  $R[i] \leftarrow Z$ ;
- $R[i] \leftarrow R[j] + R[k]$  is high level RTN description
- $Y \leftarrow R[k]$ ;  $Z \leftarrow R[j] + Y$ ;  $R[i] \leftarrow Z$ ; 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

C  
S  
D  
A  
2/e

- 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, strobos, and multiplexers