

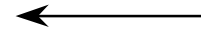
What Are the Components of an ISA?

- Sometimes known as *The Programmer's Model* of the machine
- Storage cells
 - General and special purpose registers in the CPU
 - Many general purpose cells of same size in memory
- 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

What Must an Instruction Specify?

- Which operation to perform
 - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands
 - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result
 - Again CPU register or memory cell
- Location of next instruction
 - Almost always memory cell pointed to by program counter—PC
- Sometimes there *is* no operand, or no result, or no next instruction.
- Can you think of examples?

Data Flow



add r0, r1, r3

add r0, r1, r3

add r0, r1, r3

add r0, r1, r3
br endloop



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**
 - **Change the form of one or more operands to produce a result stored in another location**
 - ***Add, Sub, Shift, etc.***
- **Control flow instructions (branches)**
 - **Alter the normal flow of control from executing the next instruction in sequence**
 - ***Br Loc, Brz Loc2,*—unconditional or conditional branches**

Examples of Data Movement Instructions

Instruction	Meaning	Machine
<code>MOV A, B</code>	Move 16 bits from memory location A to Location B	VAX11
<code>LDA A, Addr</code>	Load accumulator A with the byte at memory location Addr	M6800
<code>lwz R3, A</code>	Move 32-bit data from memory location A to register R3	PPC601
<code>li \$3, 455</code>	Load the 32-bit integer 455 into register \$3	MIPS R3000
<code>mov R4, dout</code>	Move 16-bit data from R4 to output port dout	DEC PDP11
<code>IN, AL, KBD</code>	Load a byte from in port KBD to accumulator	Intel Pentium
<code>LEA.L (A0), A2</code>	Load the address pointed to by A0 into A2	M6800

- Lots of variation, even with one instruction type

Examples of ALU Instructions

<u>Instruction</u>	<u>Meaning</u>	<u>Machine</u>
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 4 bits	Intel 8086

- Notice again the **COMPLETE DISSIMILARITY** of both syntax and semantics.

Change of Control Instructions

<u>Instruction</u>	<u>Meaning</u>	<u>Machine</u>
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. (error in textbook)	Intel 8086

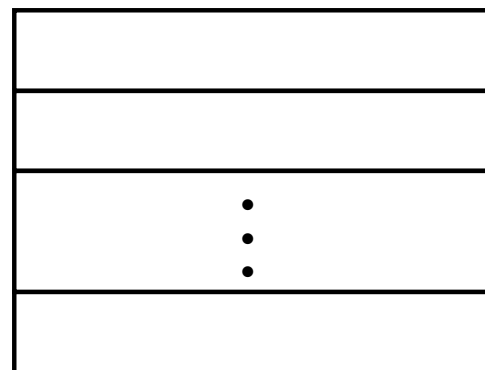
CPU Registers Associated with Flow of Control—Branch Instructions

- Program counter usually locates next instruction
- Condition codes may control branch (Intel 808x6)

Processor State



Program Counter



Branch Targets



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

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

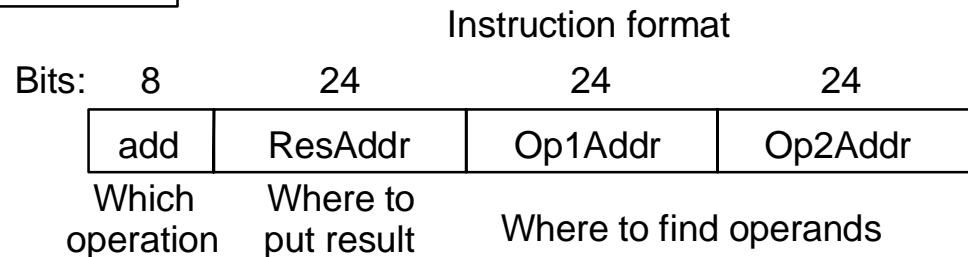
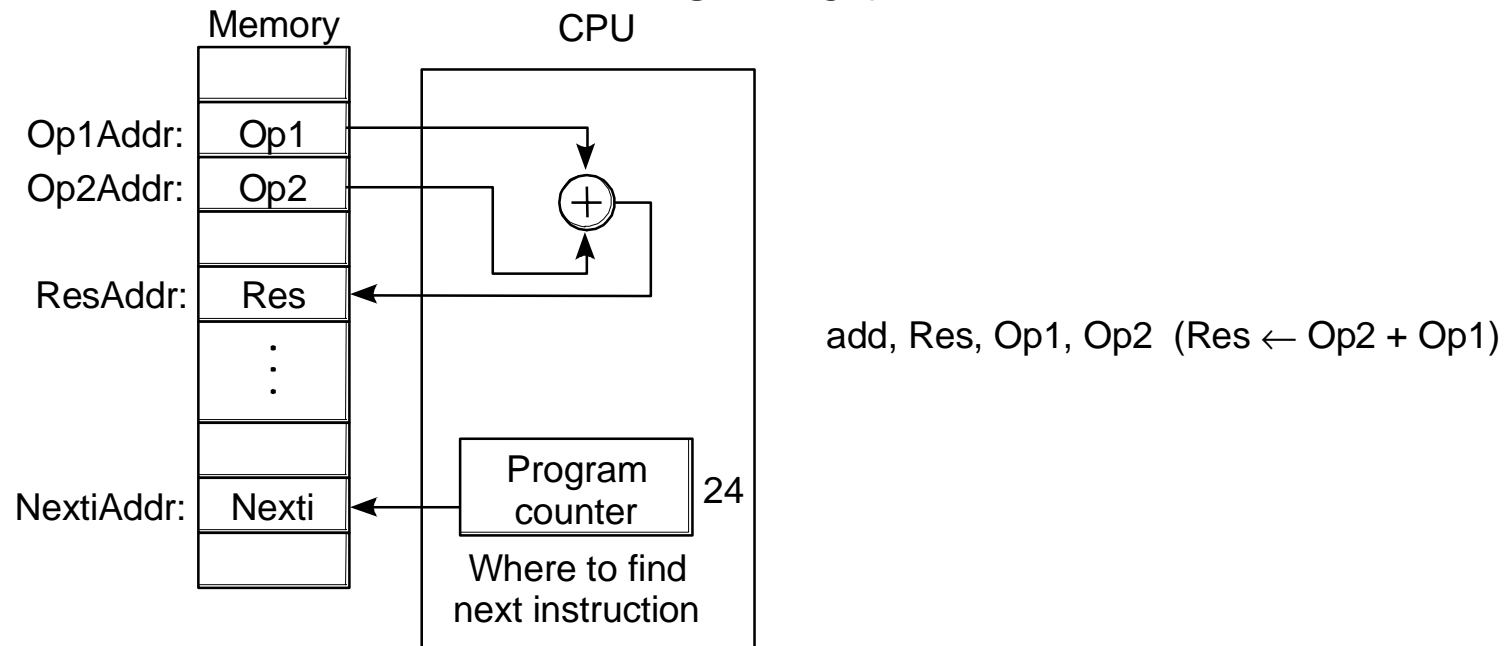
Assembly language

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

3-, 2-, 1-, & 0-Address ISA's (!4)

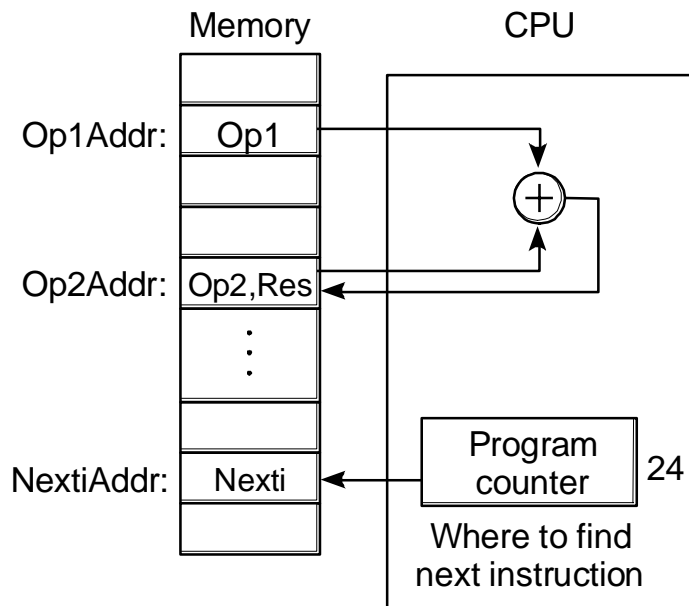
- The classification is based on arithmetic instructions that have two operands and one result
- The key issue is “how many of these are specified by memory addresses, as opposed to being specified implicitly”
- A 3-address instruction specifies memory addresses for both operands and the result: $R \leftarrow \text{Op1 op Op2}$
- 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* to hold one operand & the result (no addr. 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) - STACK ARITHMETIC
- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly

The 3-Address Machine and Instruction Format

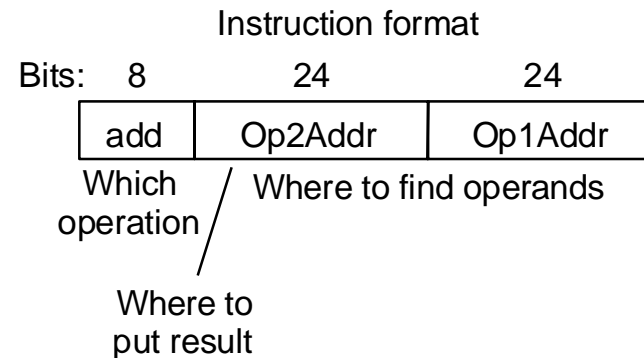


- **Address of next instruction kept in processor state register—the PC (except for explicit branches/jumps)**
- **Rest of addresses in instruction --- may be registers too**

The 2-Address Machine and Instruction Format

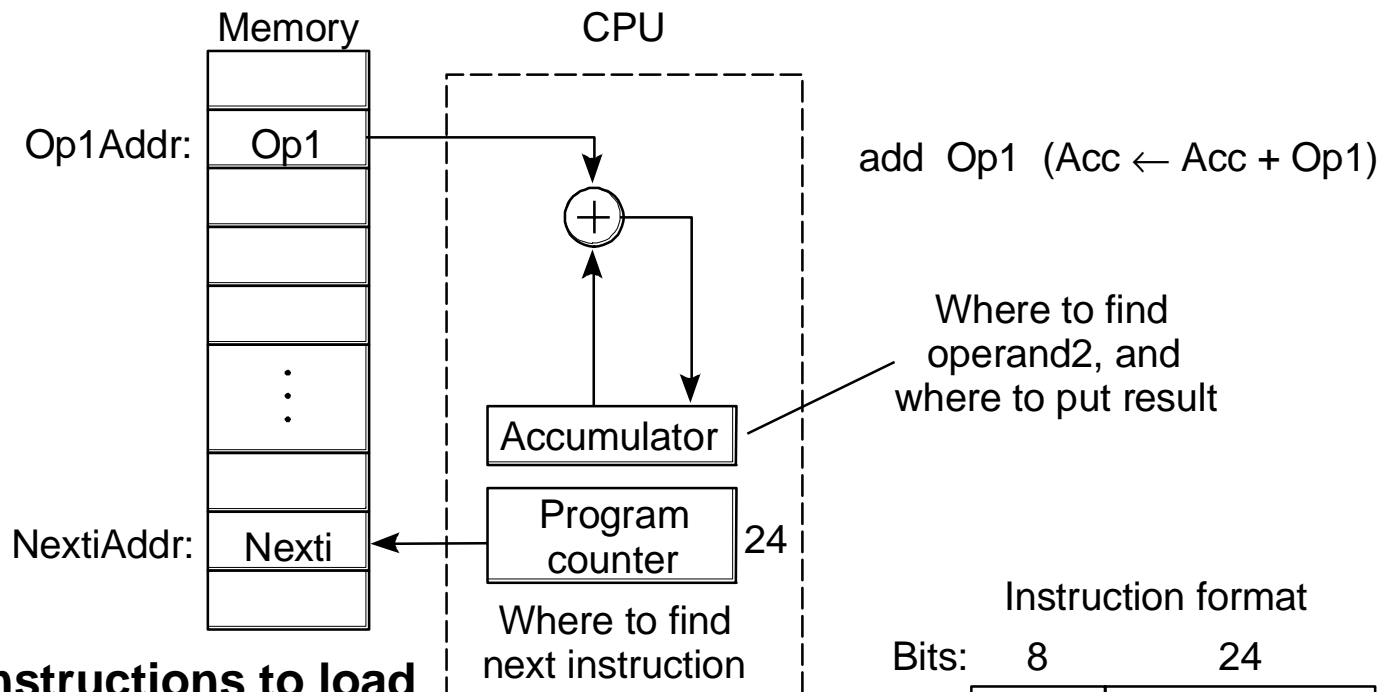


add Op2, Op1 ($Op2 \leftarrow Op2 + Op1$)



- **Result overwrites Operand 2**
- **Needs only 2 addresses in instruction but less choice in placing data ---- again, may be register operands**

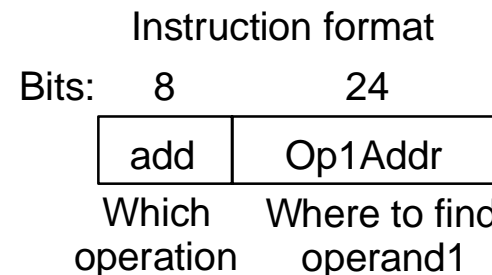
1-Address Machine and Instruction Format



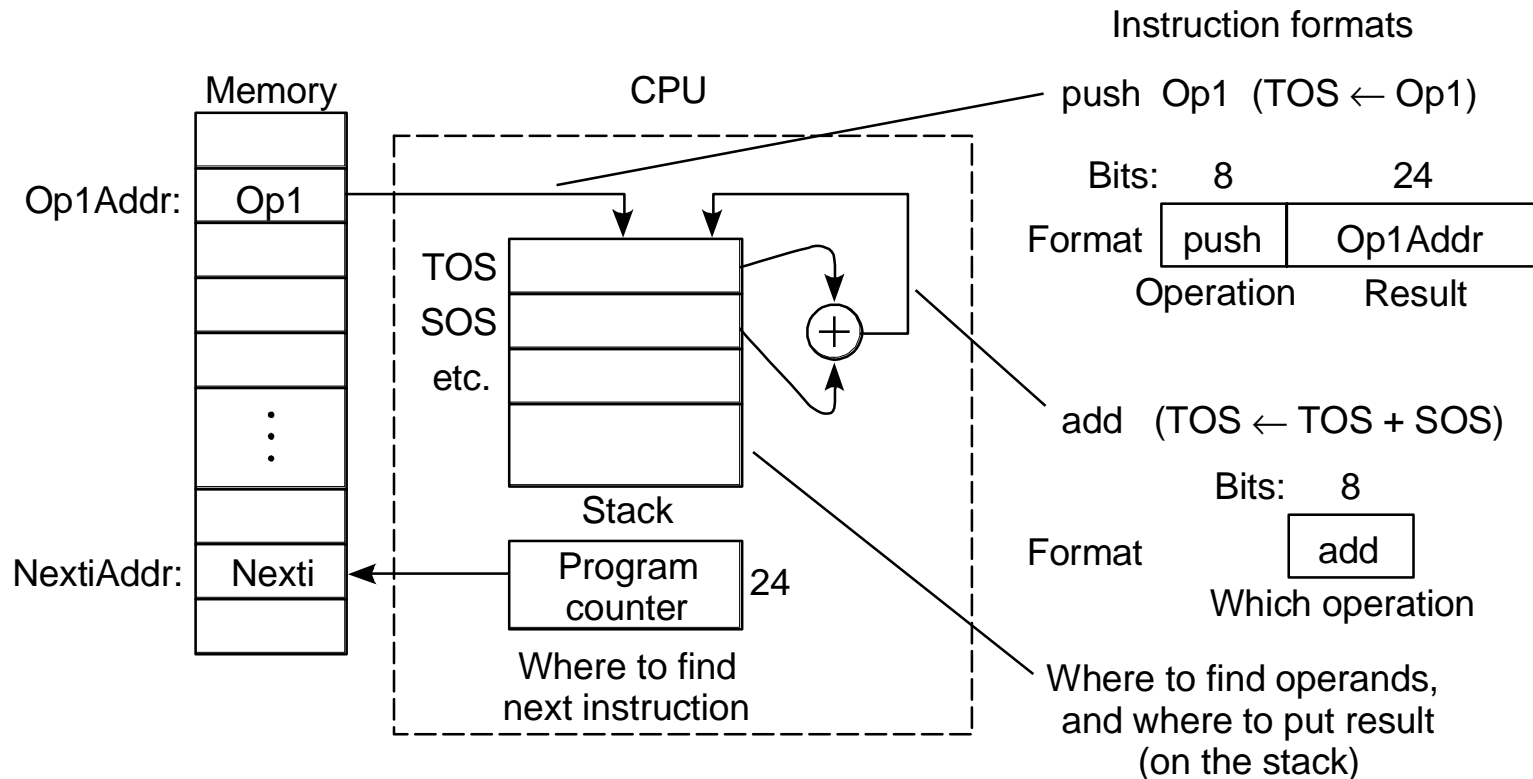
Need instructions to load and store operands:

LDA OpAddr
STA OpAddr

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



The 0-Address, or Stack, Machine and Instruction Format



- Uses a push-down stack in CPU
- Arithmetic *uses stack* for both operands and the result
- Computer must have a 1-address instruction to PUSH and POP operands to and from the stack

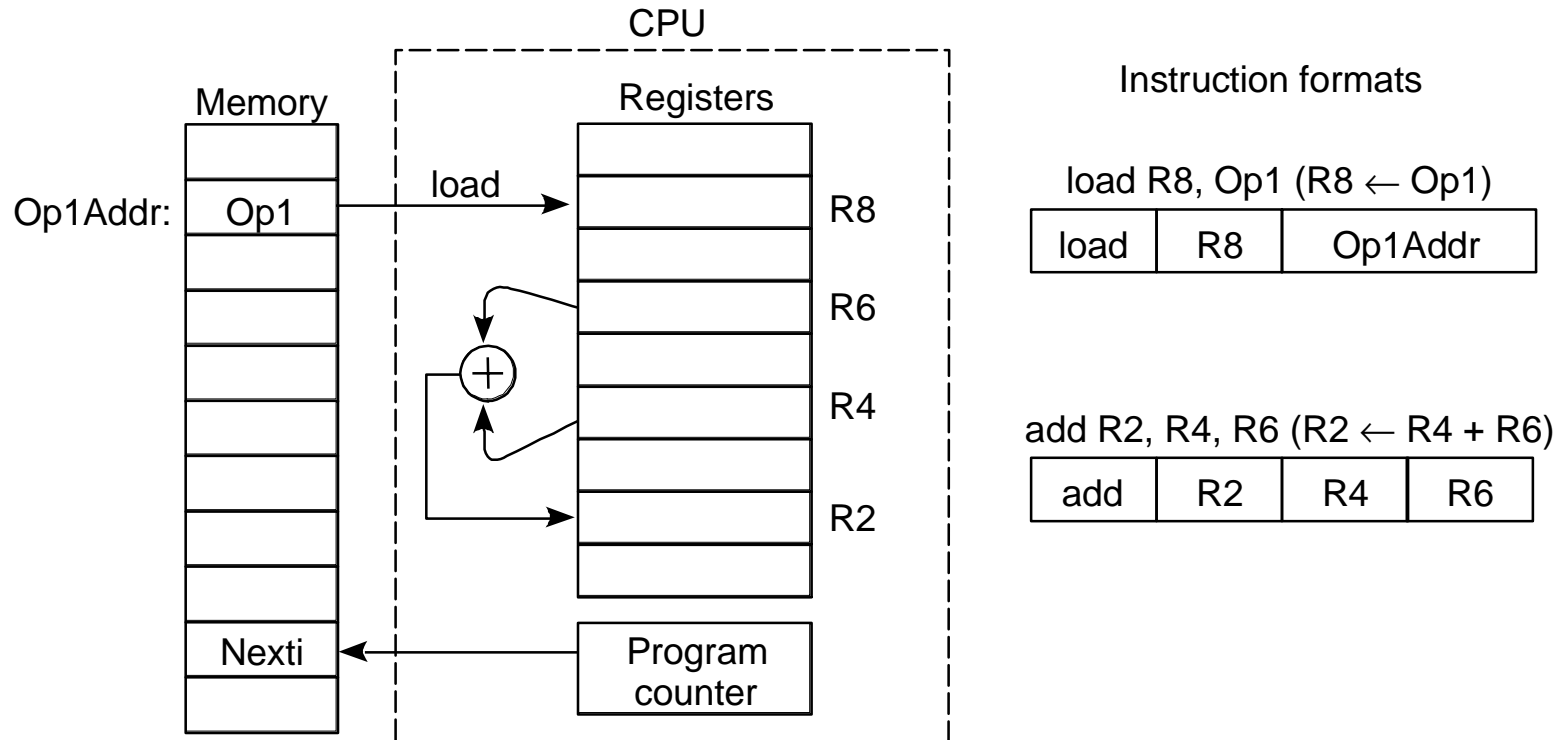
Expression Evaluation for 3-, 2-, 1-, and 0-Address Machines

Evaluate $a = (b+c)*d - e$

<u>3-address</u>	<u>2-address</u>	<u>1-address</u>	<u>Stack</u>
add a, b, c	load a, b	load 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
		store a	mpy
			push e
			sub
			pop a

- Number of instructions & number of addresses both vary
- note numbers of memory references as well as instrs.

General Register (Load-Store) Machine and Instruction Formats



- 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: $1\frac{1}{2}$ addresses
- Arithmetic instruction has 3 "half" addresses

Real Machines Are Not So Simple

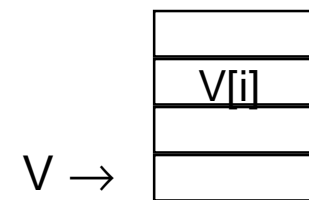
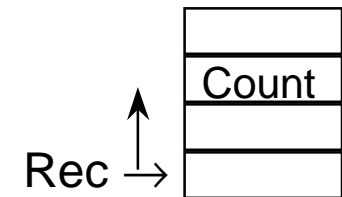
- **Most real machines have a mixture of 3, 2, 1, 0, and 1½ address instructions**
- **A distinction can be made on whether arithmetic instructions use data from memory**
- **If ALU instructions only use registers for operands and result, machine type is load-store**
 - **Only load and store instructions 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 variables
 - 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) + constant (small)
 - Array components:
 - constant base (full address) + index variable (small)
 - Possible to store constant values w/o using another memory cell by storing them with or adjacent to the instruction itself

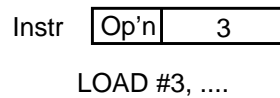
HLL Examples of Structured Addresses

- **C language: $rec \rightarrow count$**
 - **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 $Rec \rightarrow Count$ is address of $Rec + \text{offset of count}$.**

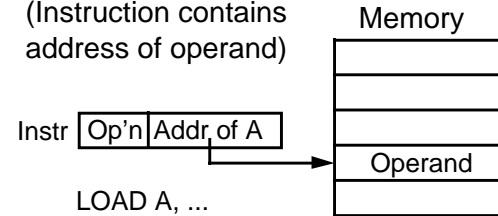


Common Addressing Modes

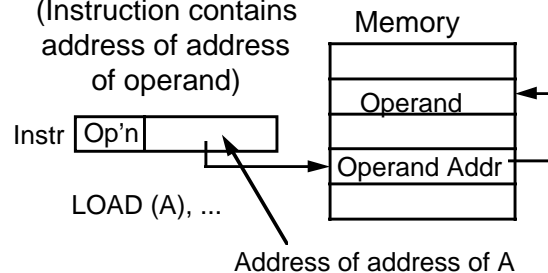
a) Immediate Addressing
(Instruction contains the operand.)



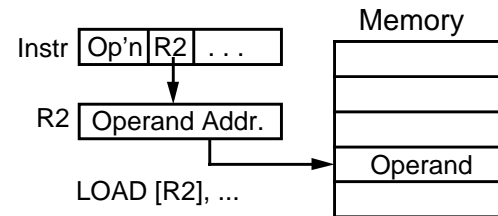
b) Direct Addressing
(Instruction contains address of operand)



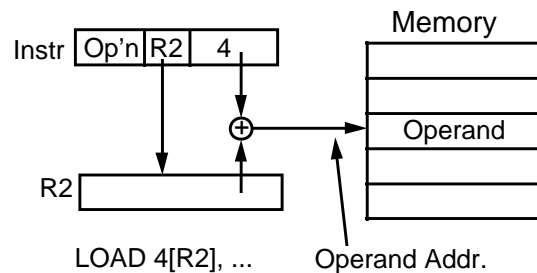
c) Indirect Addressing
(Instruction contains address of address of operand)



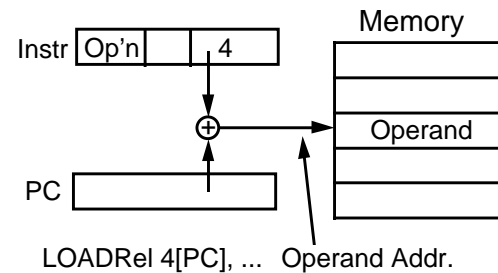
d) Register Indirect Addressing
(register contains address of operand)



e) Displacement (Based) (Indexed) Addressing
(address of operand = register + constant)



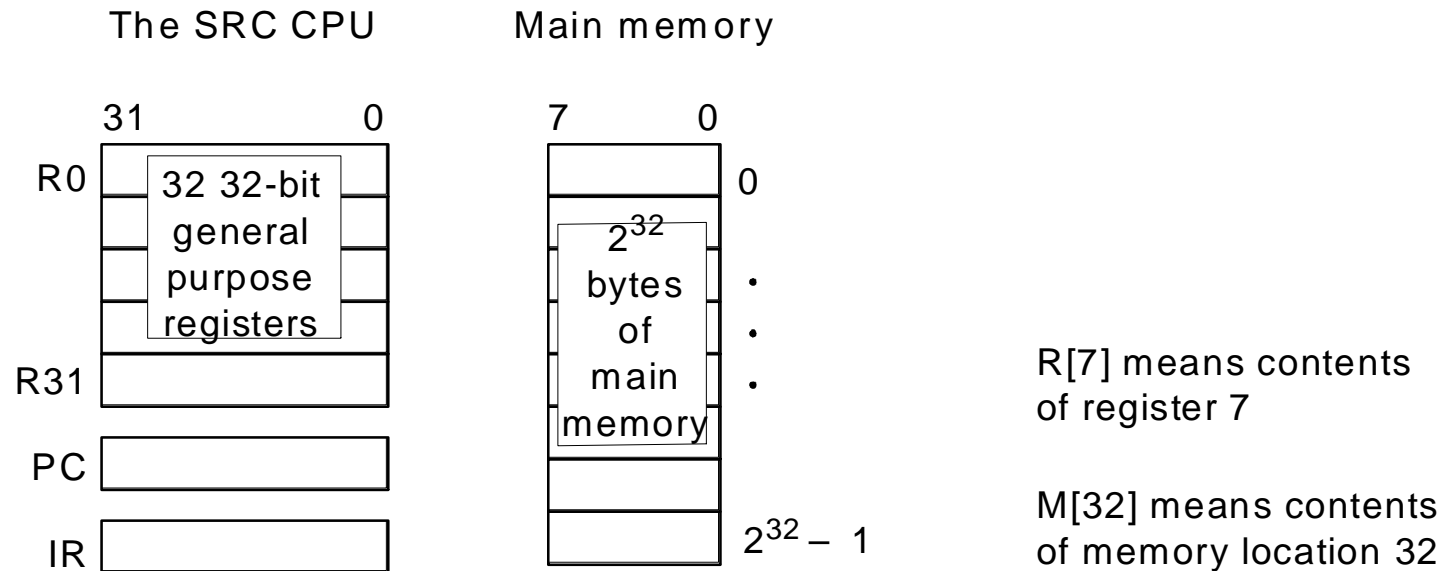
f) Relative Addressing
(Address of operand = PC + constant)



Example Computer

Simple RISC Computer (SRC)

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

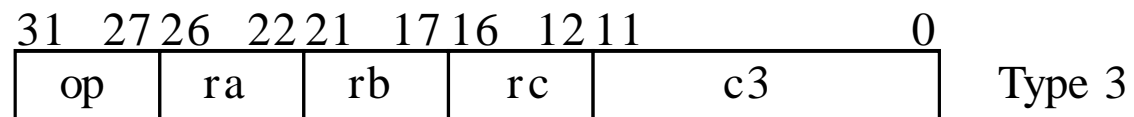
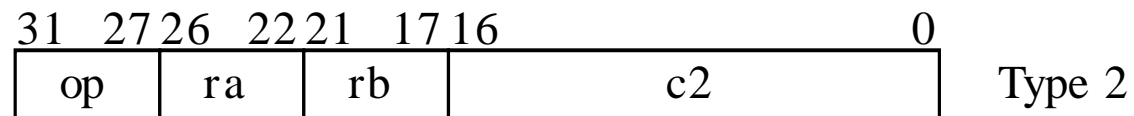
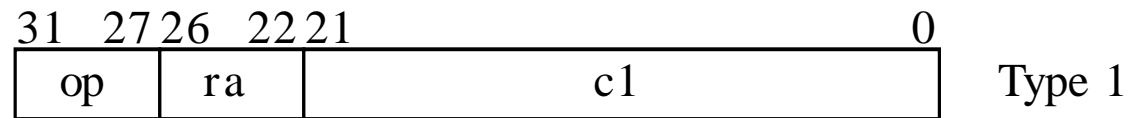


SRC Characteristics

- **Load-store design: only way to access memory is through load and store instructions**
- **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, \neq 0, \geq 0, or $<$ 0**
- **Branch and link instructions are similar, but leave the value of current PC in any register, useful for subroutine return**
- **All instructions are 32 bits (1 word) long**

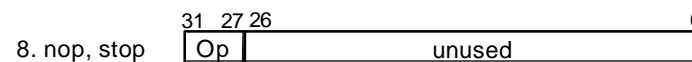
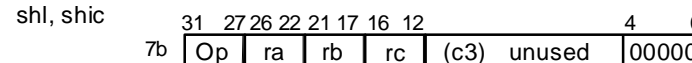
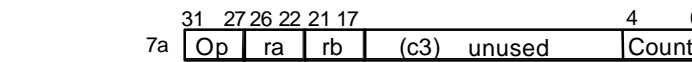
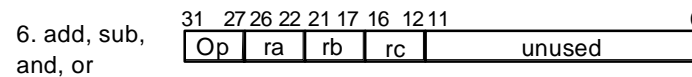
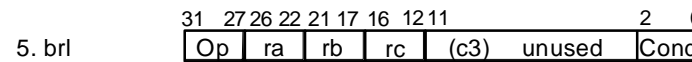
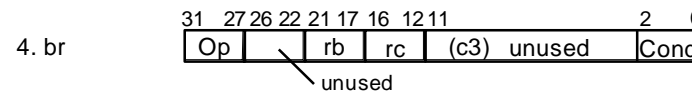
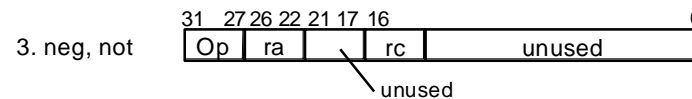
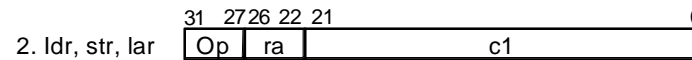
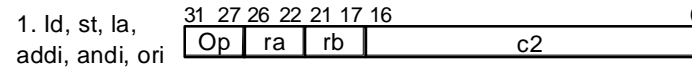
SRC Basic Instruction Formats

- 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



Total of 7 Detailed Formats (fig. 2.9)

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])
lar r6, 45 (R[6] = PC + 45)

neg r7, r9 (R[7] = - R[9])

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

brlnz 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

Example SRC Load and Store Instructions

- 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

<u>Format</u>	<u>Example</u>	<u>Meaning</u>
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 instructions:

`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>brlnv</code>	<code>brlnv r6</code>
001	always	<code>br, brl</code>	<code>br r5, brl r5</code>
010	if <code>rc = 0</code>	<code>brzr, brlzt</code>	<code>brzr r2, r4, r5</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 < 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.

Forms and Formats of the br and brl Instructions

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
brlzr	brlzr 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
brlnz	brlnz 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

In C:

```
goto Label3;
```

In SRC:

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

```
br r0 ; and branch
```

```
• • •
```

```
Label3 • • •
```

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 r0, Over      ;load address of "false" jump location
      ld r1, X          ;load value of X into r1
      brpl r0, r1       ;branch to Else if r1 ≥ 0
      neg r1, r1        ;negate value
Over: . . .            ;continue
```

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 (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<31..0>: program counter
(memory addr. of next inst.)

IR<31..0>: instruction register

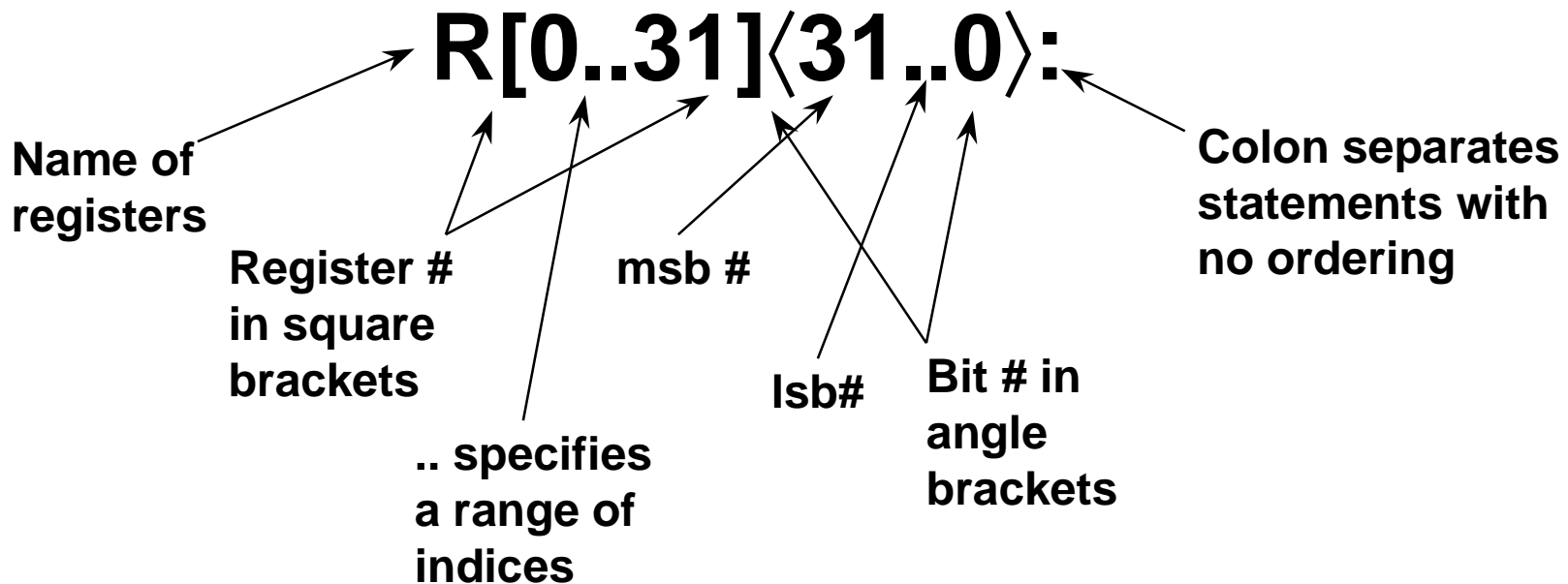
Run: one bit run/halt indicator

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]



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

Mem[0..2³² - 1]⟨7..0⟩: 2³² addressable bytes of memory

M[x]⟨31..0⟩:= Mem[x]#Mem[x+1]#Mem[x+2]#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 $\langle 4..0 \rangle := \text{IR}\langle 31..27 \rangle$:	operation code field
ra $\langle 4..0 \rangle := \text{IR}\langle 26..22 \rangle$:	target register field
rb $\langle 4..0 \rangle := \text{IR}\langle 21..17 \rangle$:	operand, address index, or branch target register
rc $\langle 4..0 \rangle := \text{IR}\langle 16..12 \rangle$:	second operand, conditional test, or shift count register
c1 $\langle 21..0 \rangle := \text{IR}\langle 21..0 \rangle$:	long displacement field
c2 $\langle 16..0 \rangle := \text{IR}\langle 16..0 \rangle$:	short displacement or immediate field
c3 $\langle 11..0 \rangle := \text{IR}\langle 11..0 \rangle$:	count or modifier field

Specifying Dynamic Properties of SRC: RTN Gives Specifics of Address Calculation

Effective address calculations (occur at runtime):

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

- Renaming defines displacement and relative addresses
- 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 := (  
¬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

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

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 instruction
 - 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 instruction

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 := (\neg Run ^ Strt \rightarrow Run \leftarrow 1:
Run \rightarrow (IR \leftarrow M[PC]: PC \leftarrow PC + 4;
ie));**

**ie := (
ld (:= op= 1) \rightarrow R[ra] \leftarrow M[disp]:
ldr (:= op= 2) \rightarrow R[ra] \leftarrow M[rel]:
...
stop (:= op= 31) \rightarrow Run \leftarrow 0:
); ii**

**Big switch
statement
on the opcode**

Thus ii and ie invoke each other, as coroutines.

Use of RTN Definitions: Text Substitution Semantics

Id (:= 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.}):**

**Id (:= 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 Id → R[5] ← M[R[3] + 11]:**

RTN Descriptions of SRC Branch Instructions

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

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

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<16..0> {2's comp. sign ext.}:
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<16..0> {sign extend}:
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<16..0> {sign extend}:
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

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

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

R[1]=

13@R[2]<31> #	R[2]<31..13>
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)**

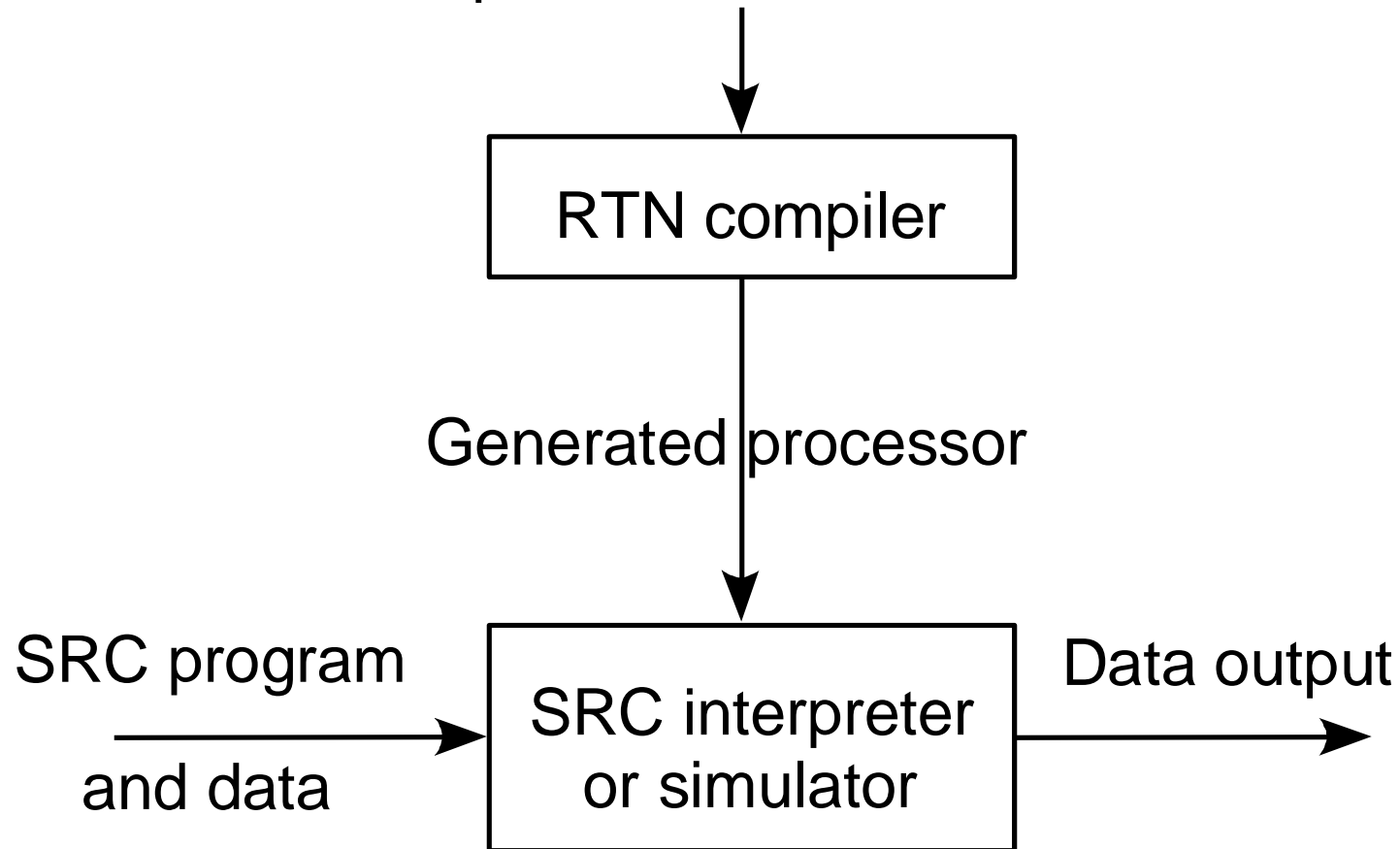
Confused about RTN and SRC?

- 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...

The Relationship of RTN to SRC

SRC specification written in RTN



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)

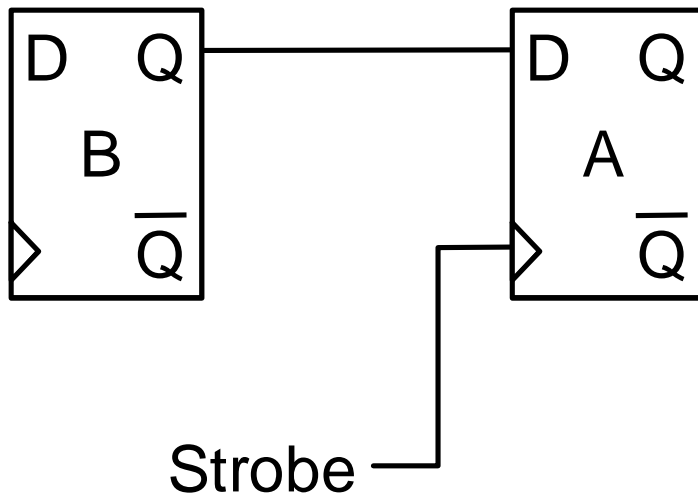
<u>Mode name</u>	<u>Assembler Syntax</u>	<u>RTN meaning</u>	<u>Use</u>
Register	Ra	$R[t] \leftarrow R[a]$	Tmp. Var.
Register indirect	(Ra)	$R[t] \leftarrow M[R[a]]$	Pointer
Immediate	#X	$R[t] \leftarrow X$	Constant
Direct, absolute	X	$R[t] \leftarrow M[X]$	Global Var.
Indirect	(X)	$R[t] \leftarrow M[M[X]]$	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 access.
Autodecrement	-(Ra)	$R[a] \leftarrow R[a] - 1; R[t] \leftarrow M[R[a]]$	

Target register

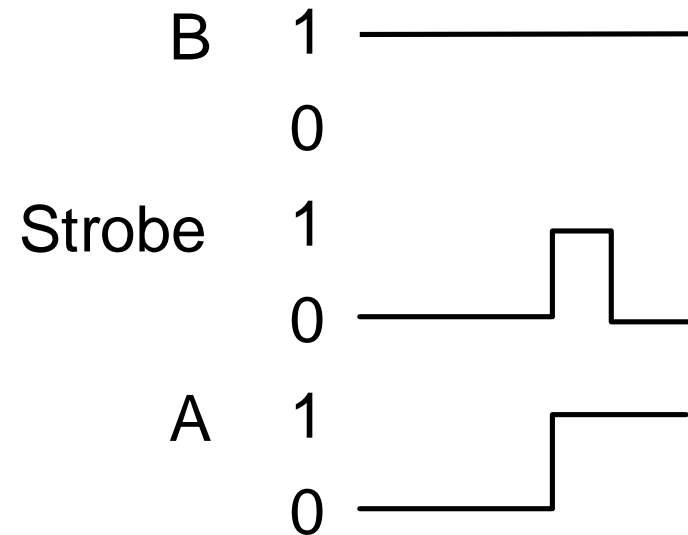


Register Transfers Hardware and Timing for a Single-Bit Register Transfer: $A \leftarrow B$

- Implementing the RTN statement $A \leftarrow B$

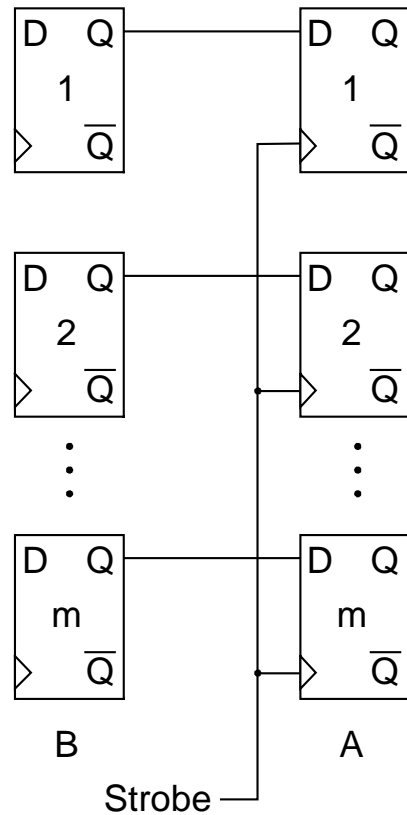


(a) Hardware

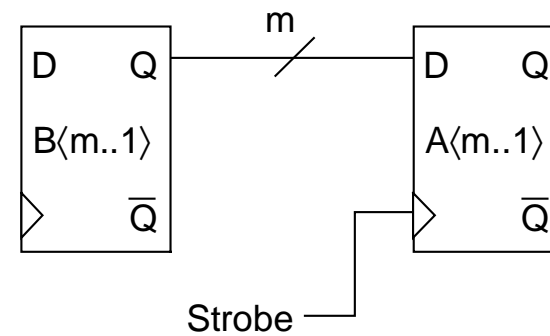


(b) Timing

Multiple Bit Register Transfer: $A\langle m..1 \rangle \leftarrow B\langle m..1 \rangle$



(a) Individual flip-flops



(b) Abbreviated notation

Registers and Arithmetic Units Connected by One Bus

Example:

Abstract RTN

$R[3] \leftarrow R[1] + R[2];$

Concrete RTN

$Y \leftarrow R[2];$

$Z \leftarrow R[1] + Y;$

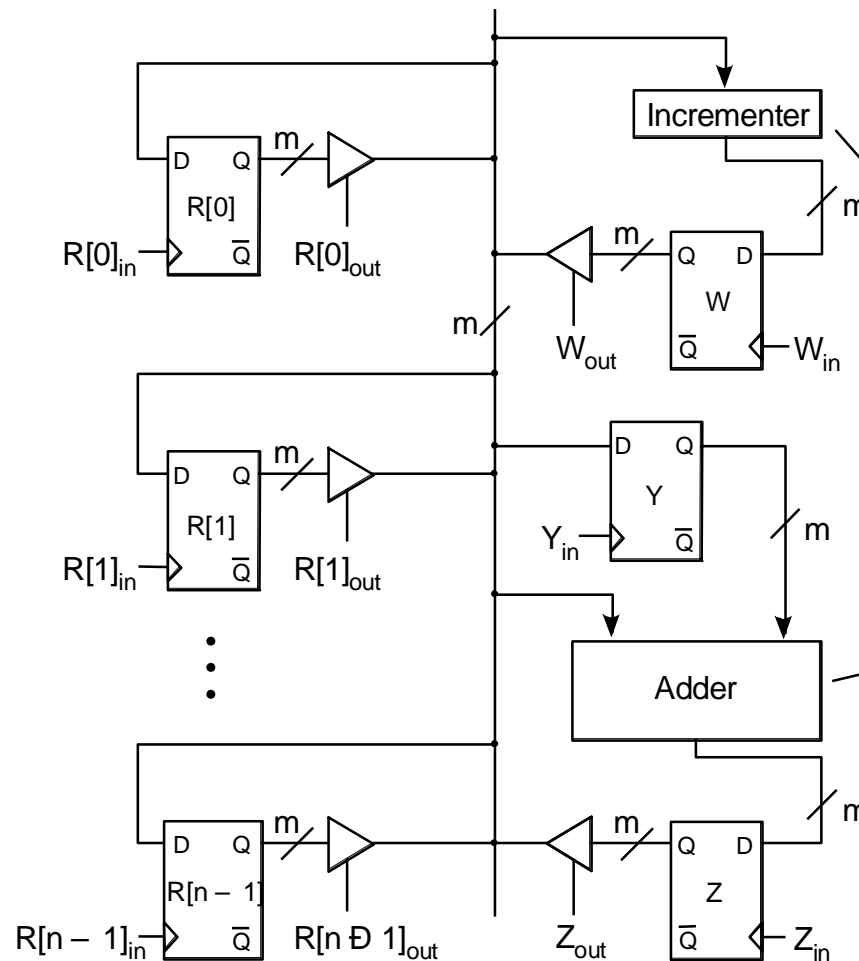
$R[3] \leftarrow Z;$

Control Sequence

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

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

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



**Combinational
logic—no
memory**

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

RTs 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

- **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 to develop various hardware designs for SRC.**