

## Instruction-set Design Issues: what is the ML instruction format(s)

### ML instruction

Opcode	Dest. Operand	Source Operand 1	...
--------	---------------	------------------	-----

#### 1) Which instructions to include:

- How many?
- Complexity - simple “ADD R1, R2, R3”  
complex e.g., VAX  
“MATCHC *substrLength, substr, strLength, str*”  
looks for a substring within a string

#### 2) Which built-in data types: integer, floating point, character, etc.

#### 3) Instruction format:

- Length (fixed, variable)
- number of address (2, 3, etc)
- field sizes

#### 4) Number of registers

#### 5) Addressing modes supported - how are the memory addresses of variables/data determining

## Number of Operands

3 Address	2 Address	1 Address (Accumulator machine)	0 Address (Stack machine)
MOVE (X ← Y)	MOVE (X ← Y)	LOAD M	PUSH M
		STORE M	POP M
ADD (X ← Y + Z)	ADD (X ← X + Y)	ADD M	ADD
SUB (X ← Y - Z)	ADD (X ← X - Y)	SUB M	SUB
MUL (X ← Y * Z)	MUL (X ← X * Y)	MUL M	MUL
DIV (X ← Y / Z)	DIV (X ← X / Y)	DIV M	DIV

$$D = A + B * C$$

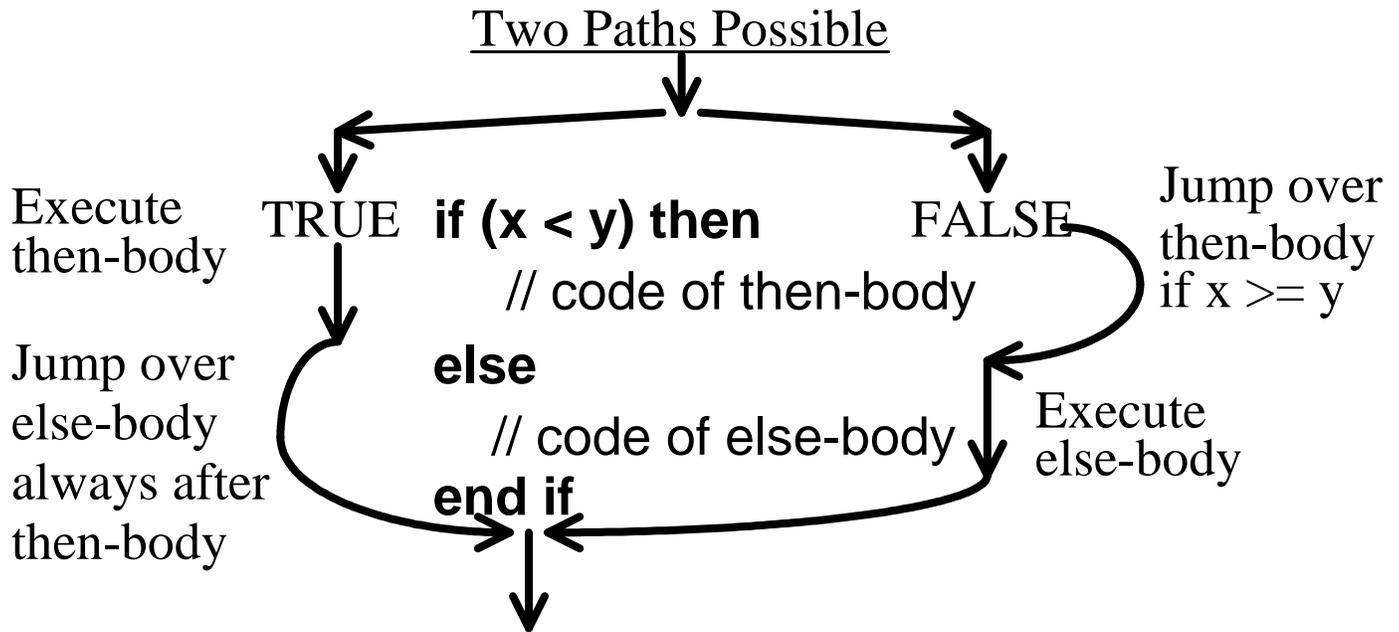
3 Address	2 Address	1 Address (Accumulator machine)	0 Address (Stack machine)
MUL D, B, C ADD D, D, A	MOVE D, B MUL D, C ADD D, A	LOAD B MUL C ADD A STORE D	PUSH B PUSH C MUL PUSH A ADD POP D

**Load/Store Architecture** - operands for arithmetic operations must be from/to registers

LOAD R1, B  
LOAD R2, C  
MUL R3, R1, R2  
LOAD R4, A  
ADD R3, R4, R3  
STORE R3, D

## Flow of Control

How do we "jump around" in the code to execute high-level language statements such as if-then-else, while-loops, for-loops, etc.



**Conditional branch** - used to jump to "else" if  $x \geq y$

**Unconditional branch** - used to always jump "end if"

**Labels** are used to name spots in the code (memory)  
("if:", "else:", and "end\_if:" in below example)

**Test-and-Jump** version of the if-then-else (Used in MIPS)

if:

    bge x, y, else

    ...

    j end\_if

else:

    ...

end\_if:

## **Set-Then-Jump** version of the if-then-else (Used in Pentium)

if:

```
    cmp x, y
```

```
    jge else
```

```
    ...
```

```
    j end_if
```

else:

```
    ...
```

```
end_if:
```

The "cmp" instruction performs  $x - y$  with the result used to set the condition codes

SF - (Sign Flag) set if result is  $< 0$

ZF - (Zero Flag) set if result = 0

CF - (Carry Flag) set if unsigned overflow

OF - (Overflow Flag) set if signed overflow

For example, the "jge" instruction checks to see if  $ZF = 1$  or  $SF = 1$ , i.e., if the result of  $x - y$  is zero or negative.

# Machine-Language Representation of Branch/Jump Instructions

## (How are labels (e.g., "end\_if") in the code located???)

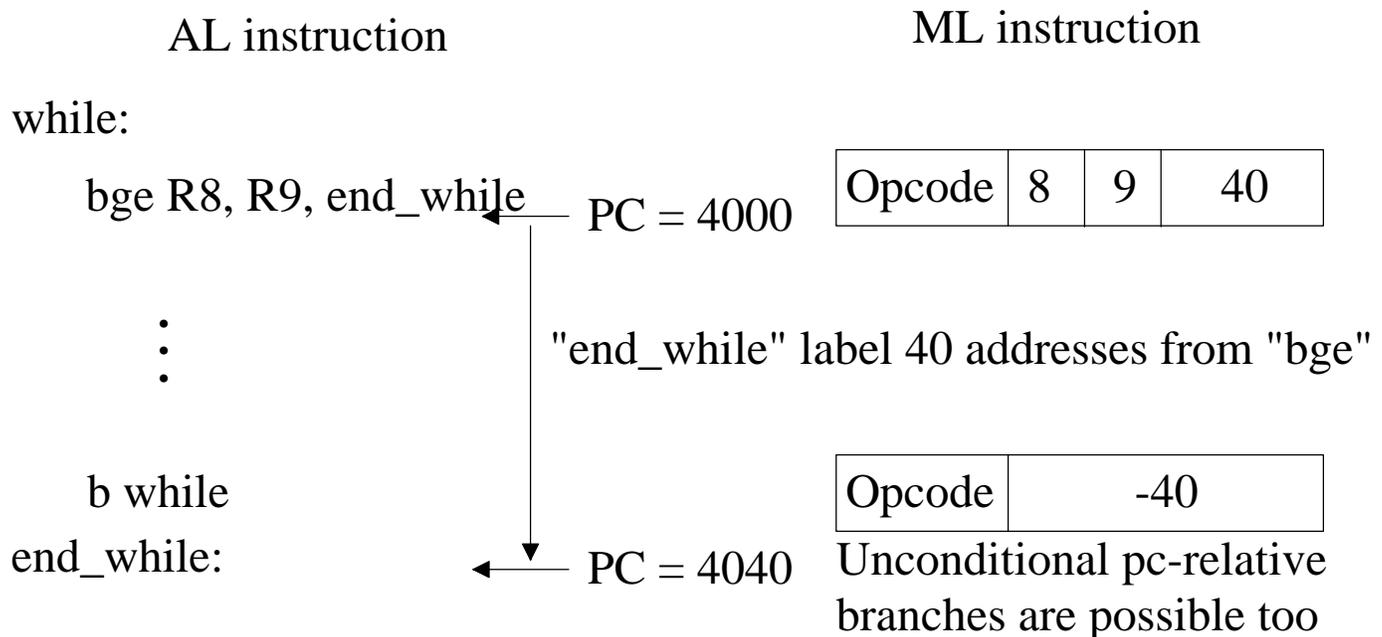
a) *direct/absolute addressing* - the memory address of where the label resides is put into the machine language instruction (EA, effective address = direct)

e.g., assume label "end\_if" is at address  $8000_{16}$



How *relocatable* is the code in memory if direct addressing is used?  
 How many bits are needed to represent a direct address?

b) *Relative/PC-relative* - base-register addressing where the PC is the implicitly referenced register



# Machine-Language Representation of Variables/Operands

(How are labels (e.g., “sum”, “score”, etc.) in the code located???)

a) *Register* - operand is contained in a register

AL instruction

add r9, r4, r2

ML instruction

Opcode	9	4	2
--------	---	---	---

b) *Direct/absolute addressing* - the memory address of where the label resides is put into the machine language instruction (EA, effective address = direct)

e.g., assume label "sum" is at address  $8000_{16}$  and “score” is at address 8004

AL instruction

add sum, sum, score

⋮

ML instruction

Opcode	8000	8000	8004
	32 bits	32 bits	32 bits

c) *Immediate* - part of the ML instruction contains the value

AL instruction

addi r9, #2

ML instruction

Opcode	9	2
--------	---	---

d) *Indirect* - operand is pointed at by an address in a memory

AL instruction  
addri r9, (A), r2

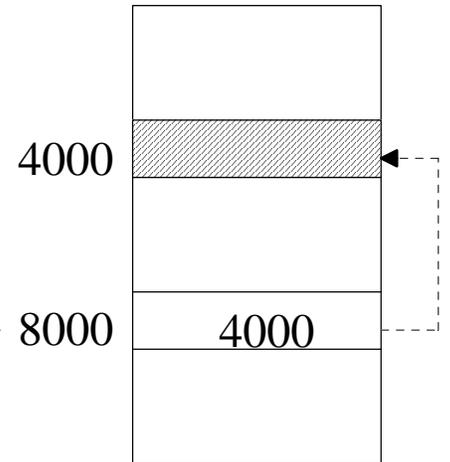
ML instruction

Opcode	9	8000	2
--------	---	------	---

ML instruction

Opcode	9	8000	2
--------	---	------	---

EA = (A)



*Register Indirect* - operand is pointed at by an address in a register

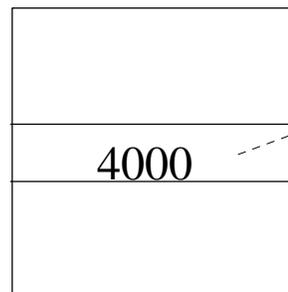
AL instruction  
addri r9, (r4), r2

ML instruction

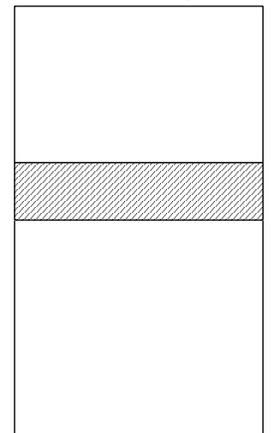
Opcode	9	4	2
--------	---	---	---

EA = (r4)

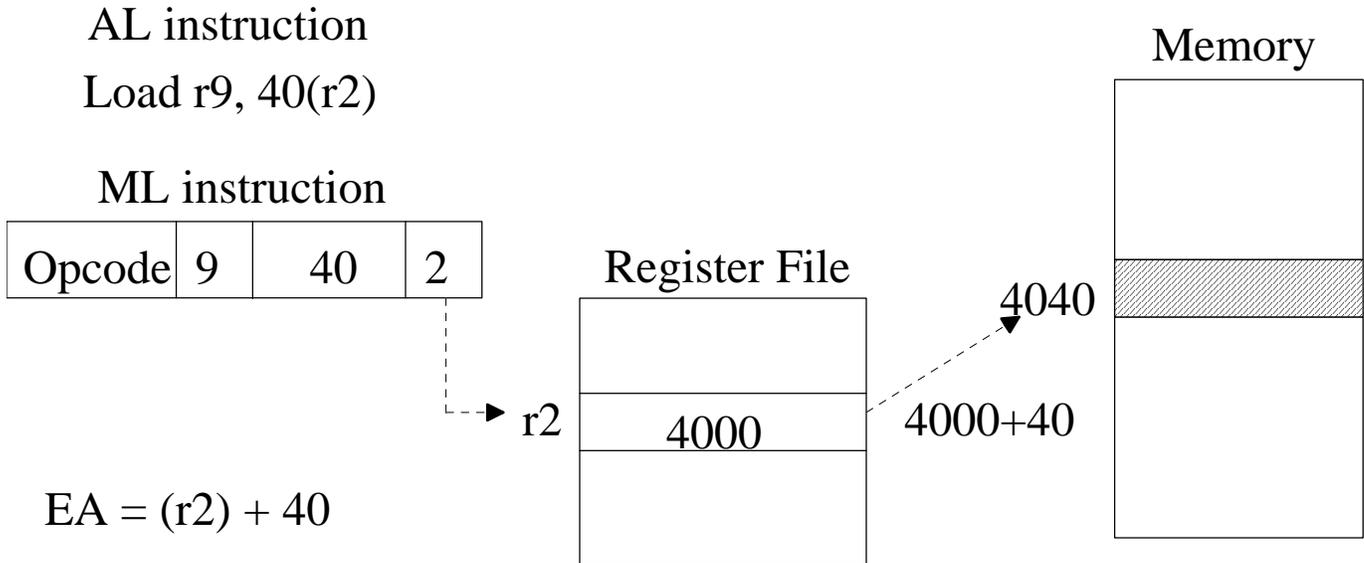
Register File



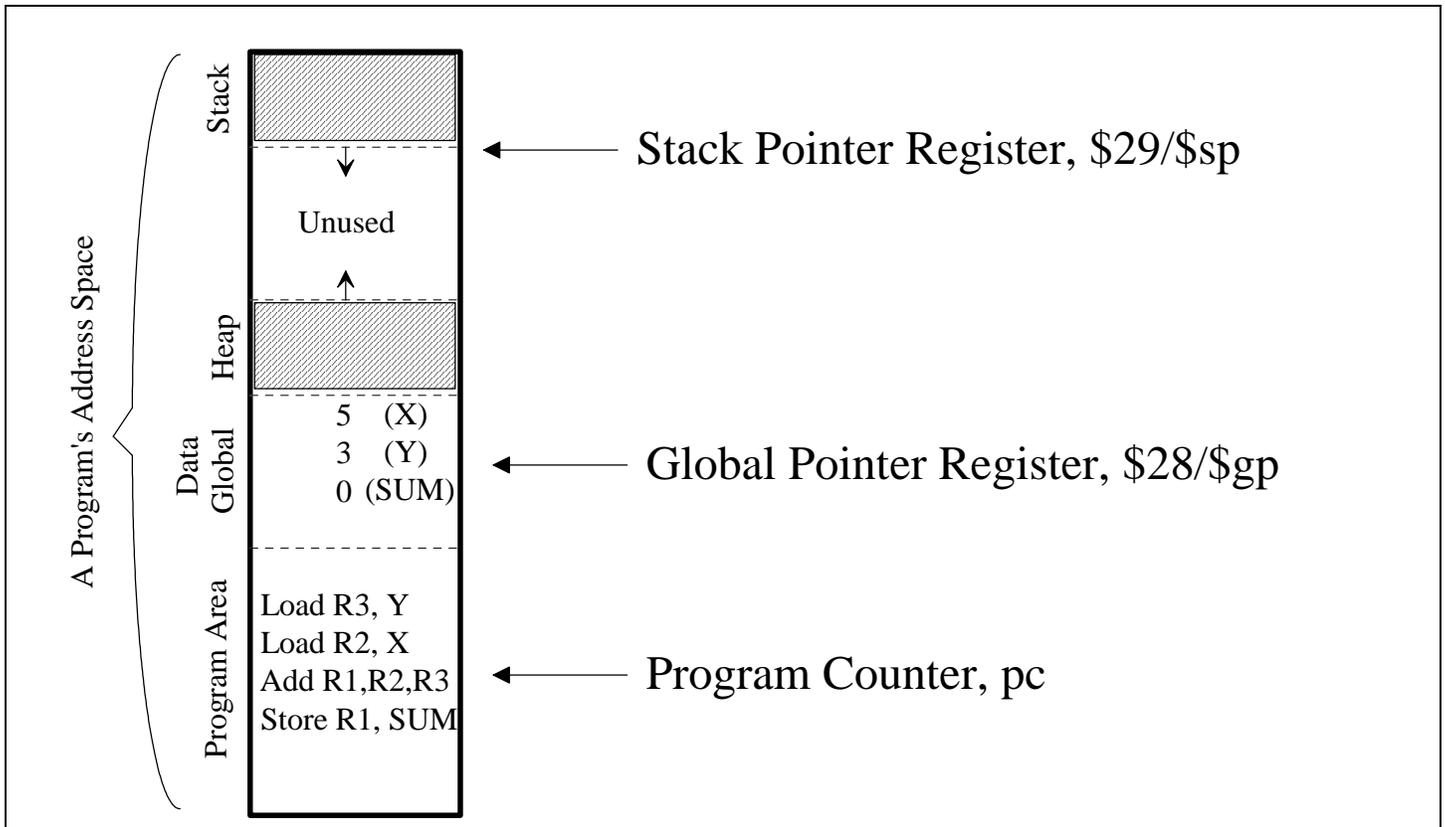
Memory



*Base-register addressing / Displacement* - operand is pointed at by an address in a register plus offset

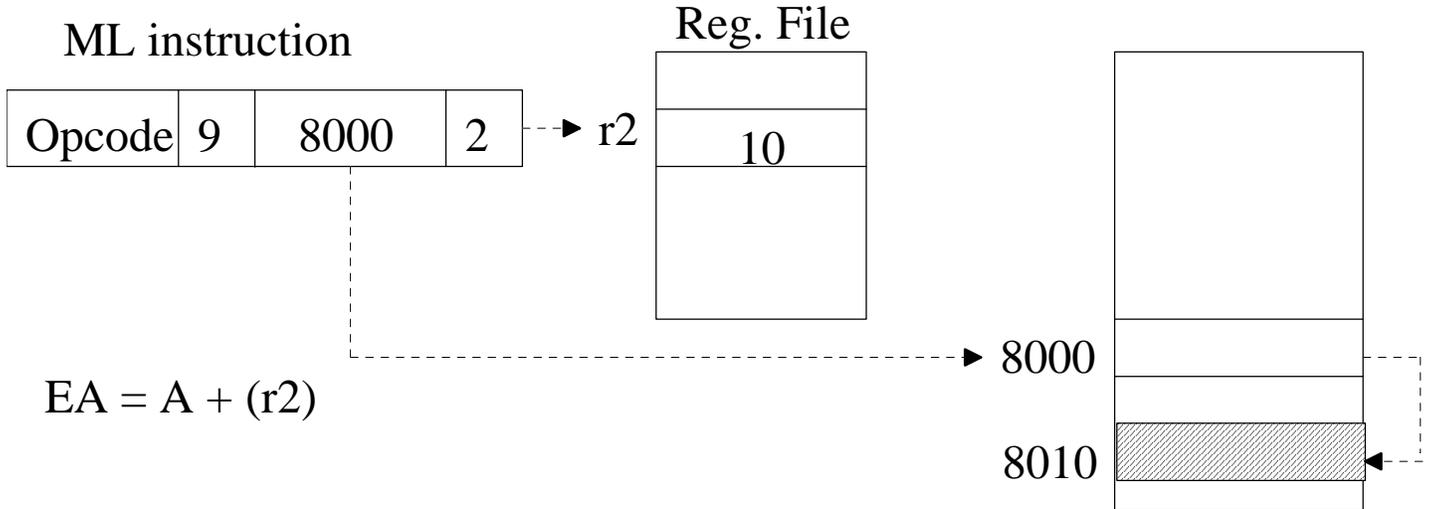
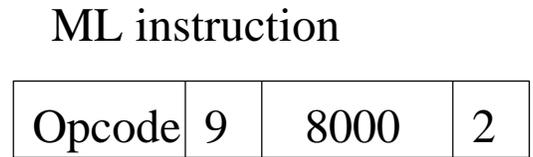


Often the reference register is the stack pointer register to manipulate the run-time stack, or a global pointer to a block of global variables.



f) *Indexing* - ML instruction contains a memory address and a register containing an index

AL instruction  
addindex r9, A(r2)



Useful for array access.

# Reduced Instruction Set Computers (RISC)

Two approaches to instruction set design:

1) CISC (Complex Instruction Set Computer) e.g., VAX or IBM 370  
1960's: Make assembly language (AL) as much like high-level language (HLL) as possible to reduce the "semantic gap" between AL and HLL

Alleged Reasons:

- reduce compiler complexity and aid assembly language programming - compilers not too good at the time (e.g., they did not allocate registers very efficiently)
- reduce the code size - (memory limited at this time)
- improve code efficiency - complex sequence of instructions implemented in microcode (e.g., VAX "MATCHC *substrLength, substr, strLength, str*" that looks for a substring within a string)

Characteristics of CISC:

- high-level like AL instructions
- variable format and number of cycles
- many addressing modes (VAX 22 addressing modes)

Problems with CISC:

- complex hardware needed to implement more and complex instructions which slows the execution of simpler instructions
- compiler can rarely figure out when to use complex instructions (verified by studies of programs)
- variability in instruction format and instruction execution time made CISC hard to pipeline

2) RISC (mid 1980's) Addresses these problems to improve speed.

(Table 9.1 - characteristics of some CISC and RISC processors)

Table 9.1 - characteristics of some CISC and RISC processors

RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., add R1, R2, R3)	One or two register operands allowed per instruction (e.g., add R1, R2)
Parameter passing through efficient on-chip register windows	Parameter passing through inefficient off-chip memory
Single-cycle instructions (except for load and store)	Multiple-cycle instructions
Hardwired control	Microprogrammed control
Highly pipelined	Less pipelined
Simple instructions that are few in number	Many complex instructions
Fixed length instructions	Variable length instructions
Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

## General Characteristics of RISC:

- emphasis on optimizing instruction pipeline
  - a) one instruction completion per cycle
  - b) register-to-register operations
  - c) simple addressing modes
  - d) simple, fixed-length instruction formats
- limited and simple instruction set and addressing modes
- large number of registers or use of compiler technology to optimize register usage
- hardwired control unit

RISC Instruction-Set Architecture (ISA) can be effectively pipelined