

Chapter 2

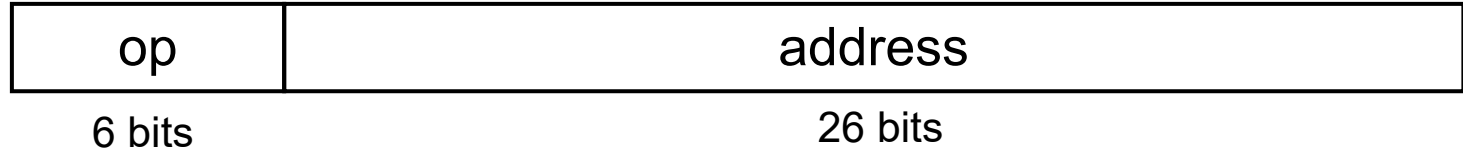
Instructions: Language of the Computer

Lecture #4



Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment



- j 10000 # go to location 10000
 - op = 2, address = 10000
- Direct jump addressing
 - Target address = address × 4

Branch Addressing

- Branch instructions specify
 - opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = PC + offset × 4

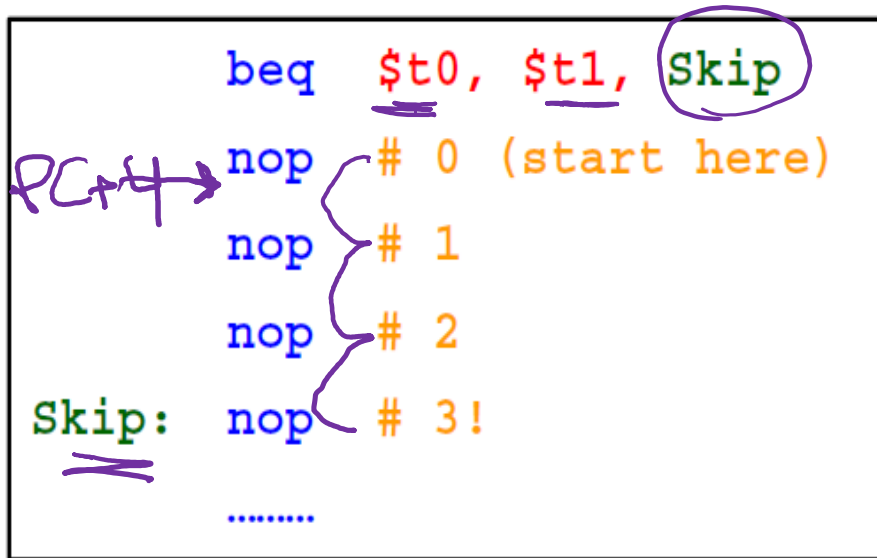
Addressing in Branches and Jumps

- **Design principle 3: Make the common case fast**
 - 16-bits for conditional branch addresses are far too small (maximum allowable program size will be 2^{16} bytes)
 - Conditional branches (`bne` and `beq`) are found in **loops** and **if statements**, so they tend to branch to a nearby instruction
 - About 50% of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away (**principle of locality!**)
- As the **Program Counter (PC)** holds the address of the instruction to be executed, we can branch within $\pm 2^{15}$ bytes of the instruction to be executed
- Use **PC-relative addressing**: add the value of the PC to the branch offset to get the jump-to address
- As it is convenient for the hardware to increment the PC early to point to the next instruction, branch offset is actually relative to the following instruction ($PC + 4$) as opposed to the current one (PC)
- Have the branch offset refer to the number of words to the destination (instead of bytes), which stretches the branch offset to be within $\pm 2^{15}$ words ($\pm 2^{17}$ bytes) of the current instruction

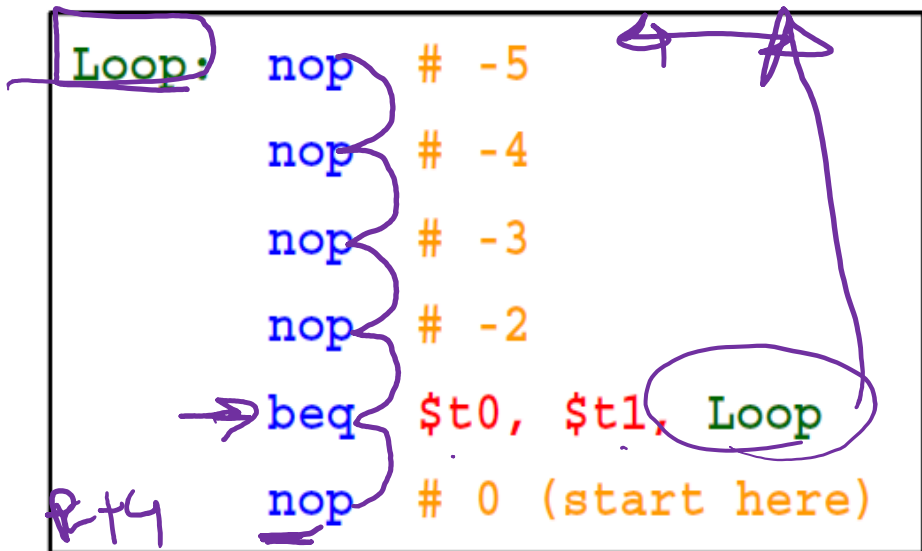
next PC (branch-to address) = (current PC + 4) + branch offset * 

Addressing in Branches and Jumps (cont.)

- **Branch offset:** number of instructions counted from the next instruction



Offset = 3



Offset = -5

Addressing in Branches and Jumps (cont.)

- J-format instructions jump to addresses that have no reason to be near them
- J-format involves absolute (direct) addressing
- The 26-bit field in J-format instructions is a word address
 - it represents 28-bit byte addresses
- Since the PC is 32 bits, the address in J-format instructions replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged (**Pseudodirect addressing**)
- J-format instructions can only be used within memory blocks of a maximum size = 256 MB (28-bit address)
- The loader and linker must be careful to avoid placing a program across an address boundary of 256 MB, for otherwise a jump must be replaced by a jump register (`jr`) instruction preceded by other instructions to load the full 32-bit address into a register



jump instruction: 0000 1010 1100 0101 0001 0100 0110 0010

op-code

26-bit target field from
jump instruction

Shift Left two
positions

32-Bit Jump Address: 0101 1011 0001 0100 0101 0001 1000 1000

High-order four bits from PC

PC: 0101 0110 0111 0110 0111 0010 1001 0100

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
I-format	op	rs	rt	16- bit immediate/address		

Fields of immediate-format branch instructions

op	6 bits	Code for the comparison to perform first
rs	5 bits	1 st argument register
rt	5 bits	2 nd argument register
address	16 bits	Branch offset embedded in the instruction

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
J-format	op	26-bit address				

Fields of jump-format instructions

op	6 bits	Code for jump instructions j or jal
address	26 bits	Target address embedded in the instruction

Note: the jr instruction is a register-format instruction



Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

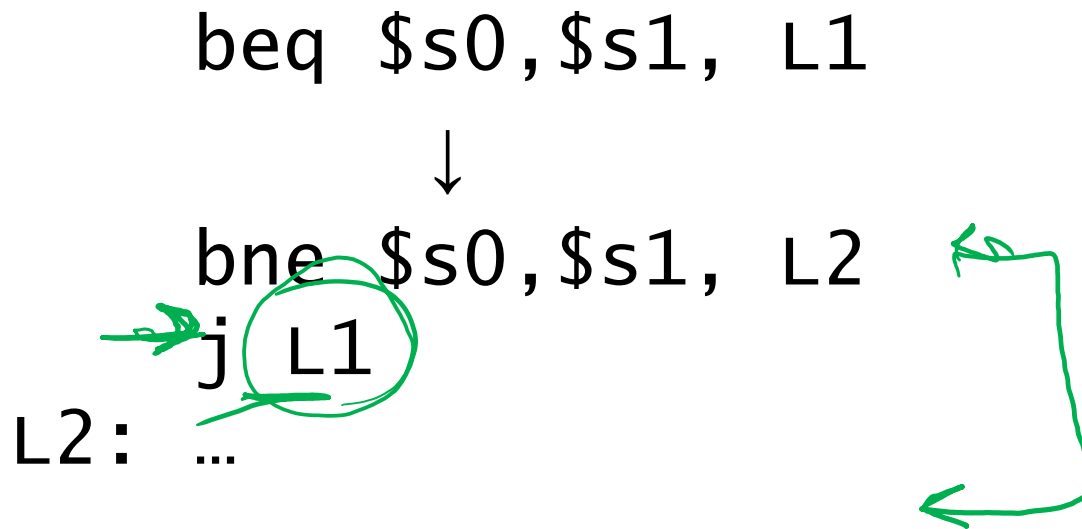
Loop: sll \$t1, \$s3, 2	80000	0	0	19	9	2	0
add \$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw \$t0, 0(\$t1)	80008	35	9	8		0	
bne \$t0, \$s5, Exit	80012	5	8	21		2	
addi \$s3, \$s3, 1	80016	8	19	19		1	
j Loop	80020	2				20000	
Exit: ...	80024						

translation $\Rightarrow 80016 + 2 \times 4 = 80024$ ✓ exit
 $\Rightarrow 20000 \times 4 = 80000$ ✓ loop

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
    beq $s0,$s1, L1
      ↓
    bne $s0,$s1, L2
    j L1
L2:  ...
```



Arrays vs. Pointers

- Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- Pointers correspond directly to memory addresses
 - Can avoid indexing complexity

Example: Clearing an Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
move $t0,$zero    # i = 0  
loop1: sll $t1,$t0,2    # $t1 = i * 4  
add $t2,$a0,$t1    # $t2 =  
                  # &array[i]  
sw $zero, 0($t2)   # array[i] = 0  
addi $t0,$t0,1     # i = i + 1  
slt $t3,$t0,$a1    # $t3 =  
                  # (i < size)  
bne $t3,$zero,loop1 # if (...)  
exit              # goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
        p = p + 4)  
        *p = 0;  
}
```

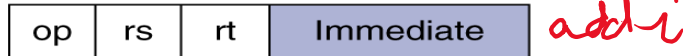
```
move $t0,$a0      # p = & array[0]  
sll $t1,$a1,2     # $t1 = size * 4  
add $t2,$a0,$t1   # $t2 = &array[size]  
loop2: sw $zero,0($t0) # Memory[p] = 0  
addi $t0,$t0,4    # p = p + 4  
slt $t3,$t0,$t2   # $t3 =  
                  # (p < &array[size])  
bne $t3,$zero,loop2 # if (...)  
                  # goto loop2
```

Comparison of Array vs. Ptr

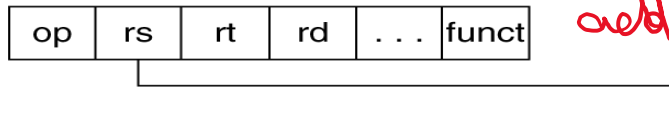
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
 - Better to make program clearer and safer

Addressing Mode Summary

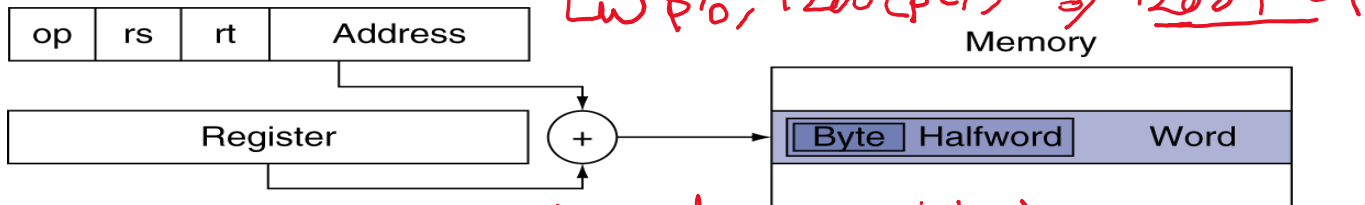
1. Immediate addressing



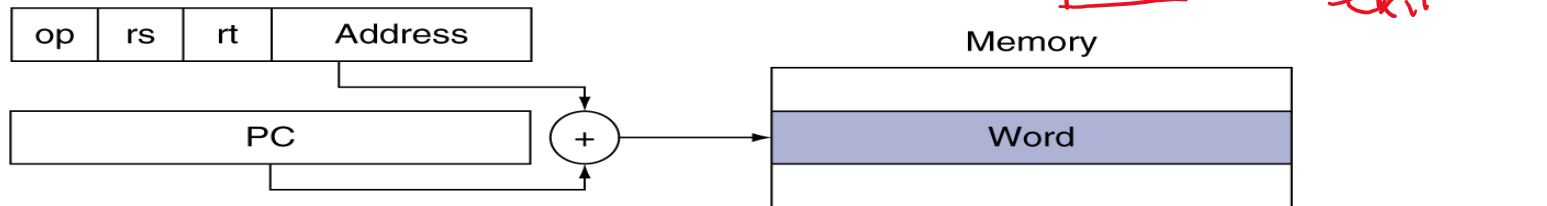
2. Register addressing



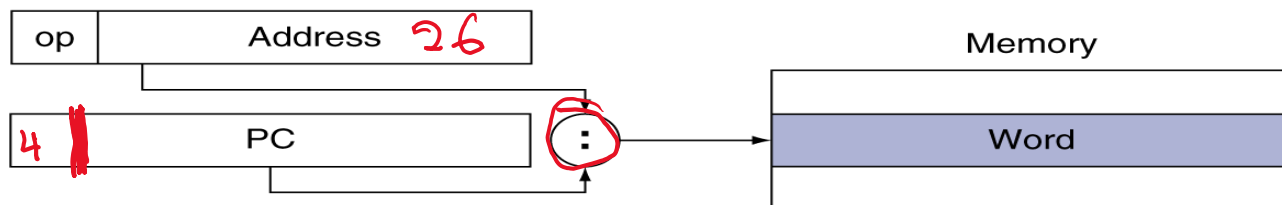
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



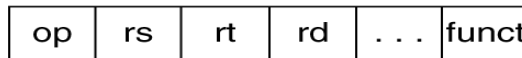
Addressing Mode Summary

1. Immediate addressing



addi

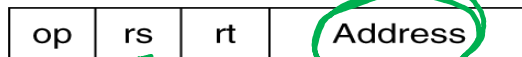
2. Register addressing



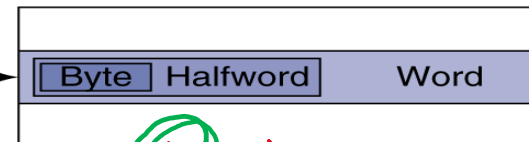
add



3. Base addressing



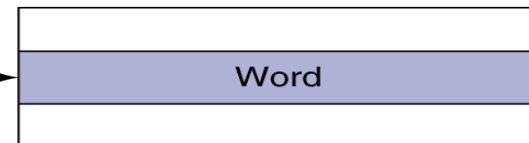
Lw \$t0, 1200(\$t1) $\Rightarrow 1200 + t_1$



4. PC-relative addressing



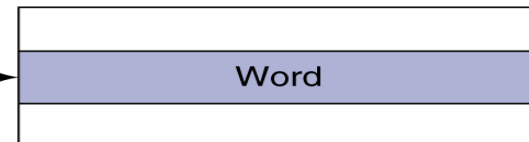
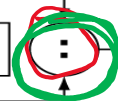
bne \$t0, \$t1, exit $\Rightarrow PC + 4$ + How far is the exit



5. Pseudodirect addressing



26



MIPS Addressing Modes Summary

- **Immediate addressing:** the operand is a constant within the instruction
`addi $s1, $s2, 100`
- **Register addressing:** the operand is a register
`add $s1, $s2, $s3`
- **Base or displacement addressing:** the operand is at the memory location whose address is the sum of a register and a constant in the instruction
`lw $s1, 100 ($s2)`
- **PC-relative addressing:** the address is the sum of the current PC and a constant (multiplied by 4) in the instruction
`beq $s1, $s2, 100`
- **Pseudodirect addressing:** the jump address is a constant (26 bits) in the instruction (multiplied by 4) concatenated with the upper 4 bits of the PC
`j 2500`
- **The assembler calculates effective addresses for branches, jumps, loads, and stores**

MIPS Addressing Modes Summary (cont.)

- **Relative** versus **absolute** (pseudodirect) addressing
 - **Branch instructions (beq, bne)**
 - Offset is **relative**
 - Next PC = (Current PC + 4) + Offset × 4
 - **Jump instructions (j, jal)**
 - Address is **absolute**
 - Next PC = (PC & 0xF0000000) | (Address × 4)
 - Absolute is relative to a 256 MB region of memory

Assembler Pseudoinstructions

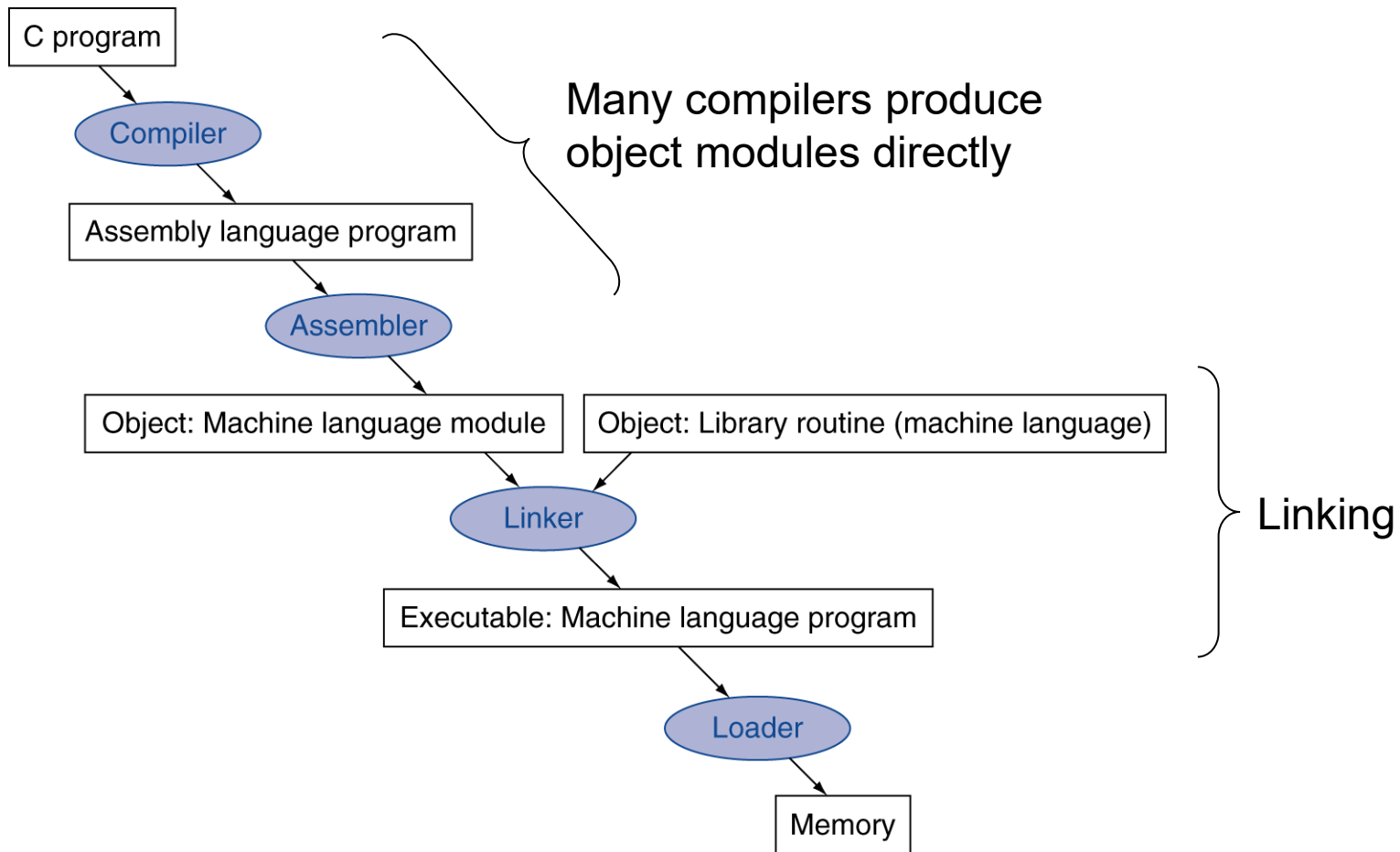
- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: – **Common variations of assembly instructions that are not part of the instruction set**
 - Often appear in MIPS programs and are treated as regular ones
 - Their appearance in assembly language simplifies programming
 - The assembler produces a minimal sequence of actual MIPS instructions to accomplish their operations
 - The assembler uses the \$at register to accomplish this, if needed
 - The hardware need not implement these instructions

`move $t0, $t1` \rightarrow `add $t0, $zero, $t1`

`blt $t0, $t1, L` \rightarrow `slt $at, $t0, $t1`
 `bne $at, $zero, L`

- **\$at (register 1): assembler temporary**

Translation and Startup



Producing an Object Module

- Assembler translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable file
 1. Place code and data modules in memory
 2. Resolve labels (determine their addresses)
 3. Patch all references



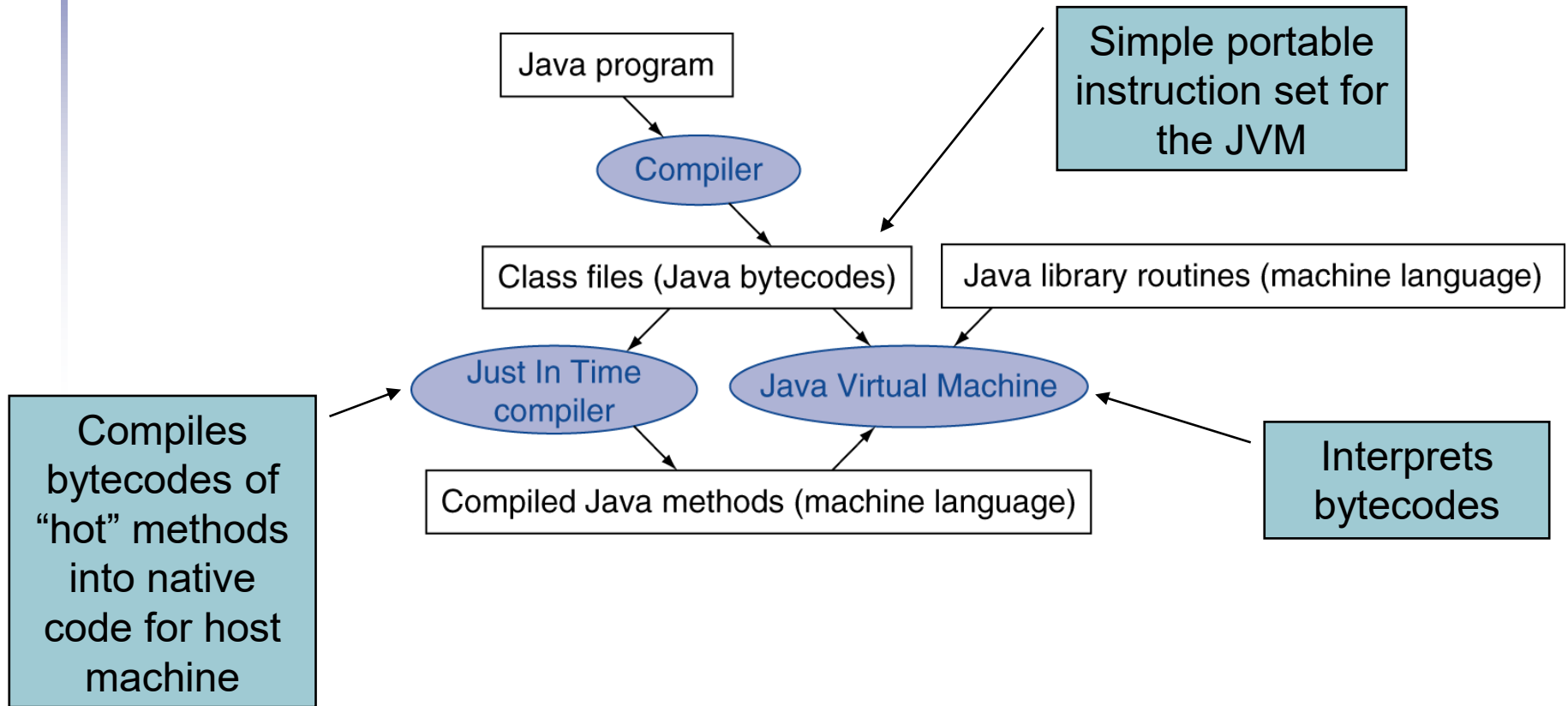
Loading a Program

- Load from file on disk into memory
 1. Read header to determine the size
 2. Create address space
 3. Copy the instructions and data from the executable file into memory
 4. Copy parameters on stack
 5. Initialize the machine registers
 6. Jump to startup routine

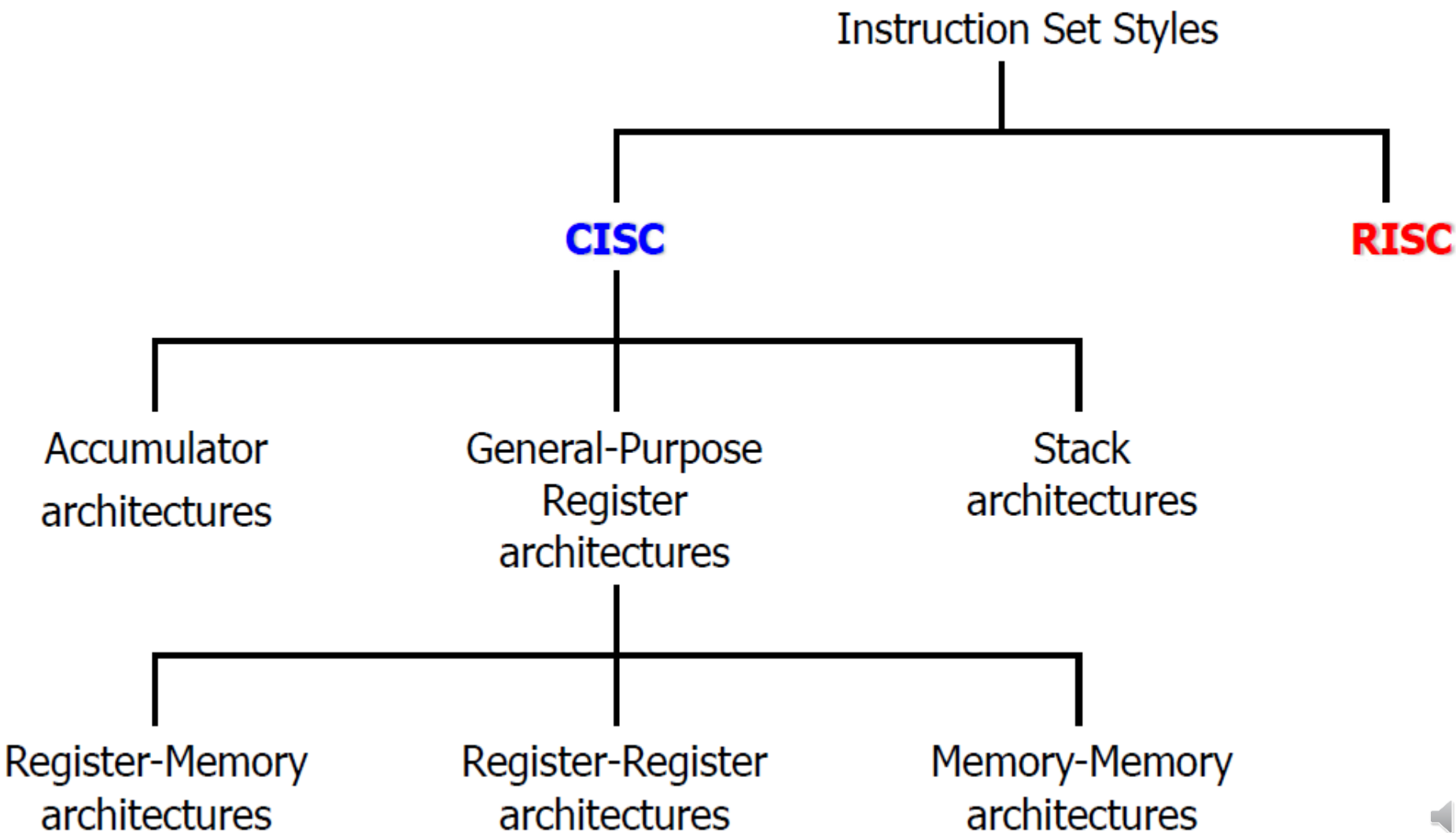
Dynamic Linking

- Only link/load library procedure when it is called
 - Avoids image bloat caused by static linking of all referenced libraries
 - Automatically picks up new library versions

Starting Java Applications



Instruction Set Styles

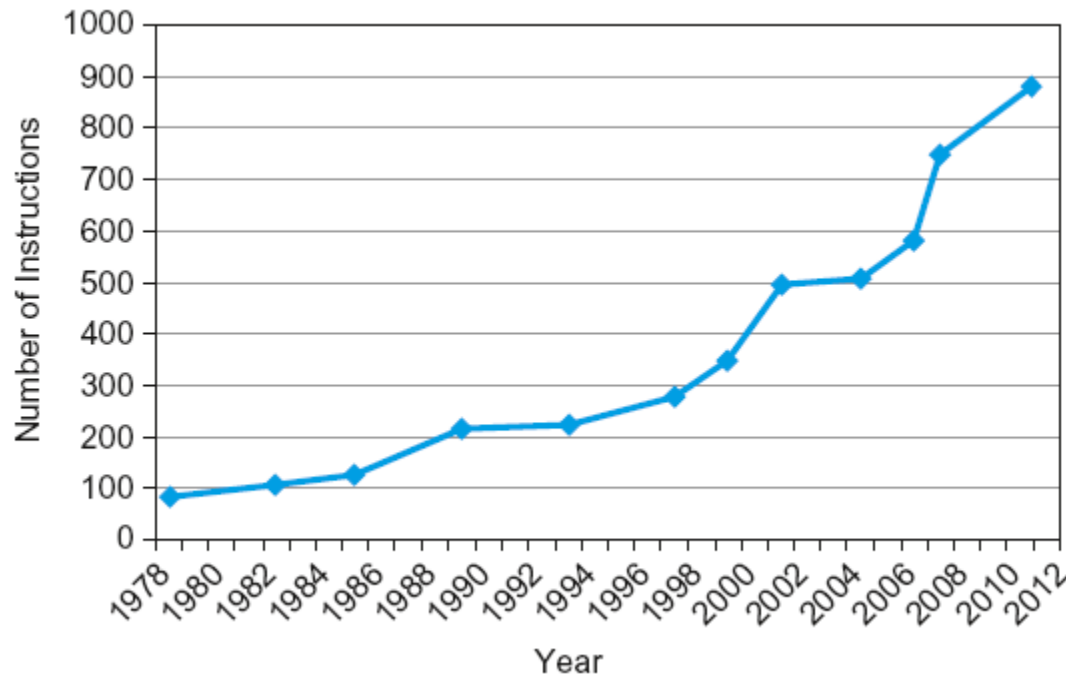


Fallacies

- Powerful instruction \Rightarrow higher performance
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS
 - ARM and x86

