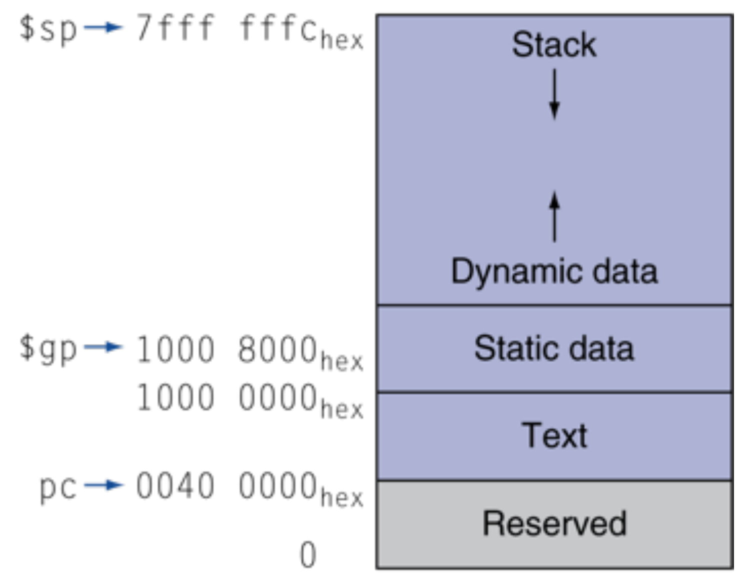
RoadMap: Procedure Calling

- Steps required
  1. Prepare and pass arguments to callee ✓
     - $a registers
  2. Transfer control to callee procedure ✓
     - jal
  3. Acquire storage for procedure ✓
     - Manipulate $sp + lw/sw
  4. Perform procedure's operations ←
  5. Pass result to caller ✓
     - $v registers
  6. Return to place of call ✓
     - jr $ra

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
  - Need a system call in MIPS
- Stack: automatic storage
  - High to low



Procedure Calling

- Steps required
  1. Prepare and pass arguments to callee ✓
  2. Transfer control to callee procedure ✓
  3. Acquire storage for procedure ✓
  4. Perform procedure's operations
     - How to make procedure calls?
  5. Pass result to caller ✓
  6. Return to place of call ✓

Non-Leaf Procedure Example

```
main()
{ ----
  ----
  A (3);
  X: ---
}

int A (int n)
{ ---
  B (5);
  Y: ---
}

int B (int m)
{ ---
}
```

To call A(3): $a0 = 3
jal A ➔ $ra = address X

To call B(5): $a0 = 5
jal B➔ $ra = address Y

- Not able to continue to use $a0 as 3!
- Not able to return to main!

Non-Leaf Procedures

- Procedures that call other procedures
- Caller needs to prepare for making a call
  - Put parameters in argument registers
  - Set the return address
- If caller is a subroutine, caller needs to save on the stack:
  - Its own return address
  - Any arguments (and temporaries) needed after the call
- Restore from the stack after the call

Register Conventions

- Caller preserved registers
  - Return address: $ra (reg 31)
  - Arguments: $a0, $a1, $a2, $a3 (reg's 4 – 7)
  - Return value: $v0, $v1(reg's 2 and 3)
  - Temporaries: $t0, $t1, … , $t7, $t8, $t9 (reg's 8 – 15,24,25)
- Callee preserved registers
  - Local variables: $s0, $s1, … , $s7 (reg's 16 – 23)
  - Stack/frame pointer: $sp, $fp (reg 29,30)
- Only need to save used registers
- A subroutine can be both caller and callee (non-leaf)

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

```
fact:
        addi $sp, $sp, -8      # adjust stack for 2 items
        sw   $ra, 4($sp)       # save return address
        sw   $a0, 0($sp)       # save argument
        slti $t0, $a0, 1       # test for n < 1
        beq  $t0, $zero, L1
        addi $v0, $zero, 1     # if so, result is 1
        addi $sp, $sp, 8       # pop 2 items from stack
        jr   $ra               #   and return
    L1: addi $a0, $a0, -1      # else decrement n
        jal  fact              # recursive call
        lw   $a0, 0($sp)       # restore original n
        lw   $ra, 4($sp)       #   and return address
        addi $sp, $sp, 8       # pop 2 items from stack
        mul  $v0, $a0, $v0     # multiply to get result
        jr   $ra               # and return
```

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

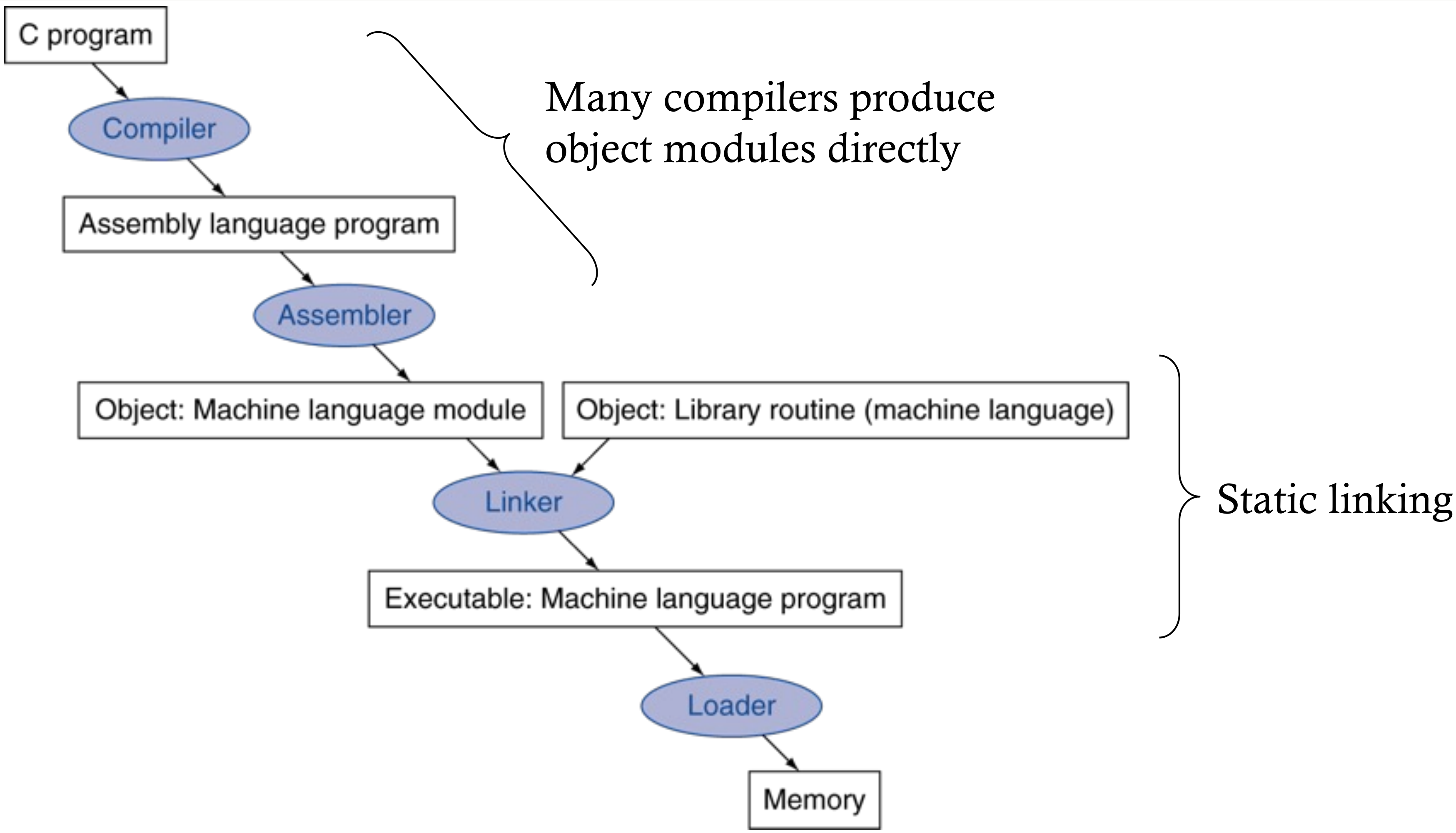
Steps for Making a Procedure Call

- 1) Save necessary values into stack **Prolog**
  - Return address, arguments, …
- 2) Assign argument(s), if any
  - Special registers
- 3) jal call
  - Control transferred to callee after executing this instruction
- 4) Restore values from stack after return from callee **Epilog**

Road Map – MIPS ISA

- MIPS basic instructions
- MIPS instruction format
- Procedure calls
- Other misc.
  - Translating and starting a program
  - Synchronization instructions
  - Arrays and pointers
  - Fallacies and pitfalls

Translation and Startup



Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler's imagination

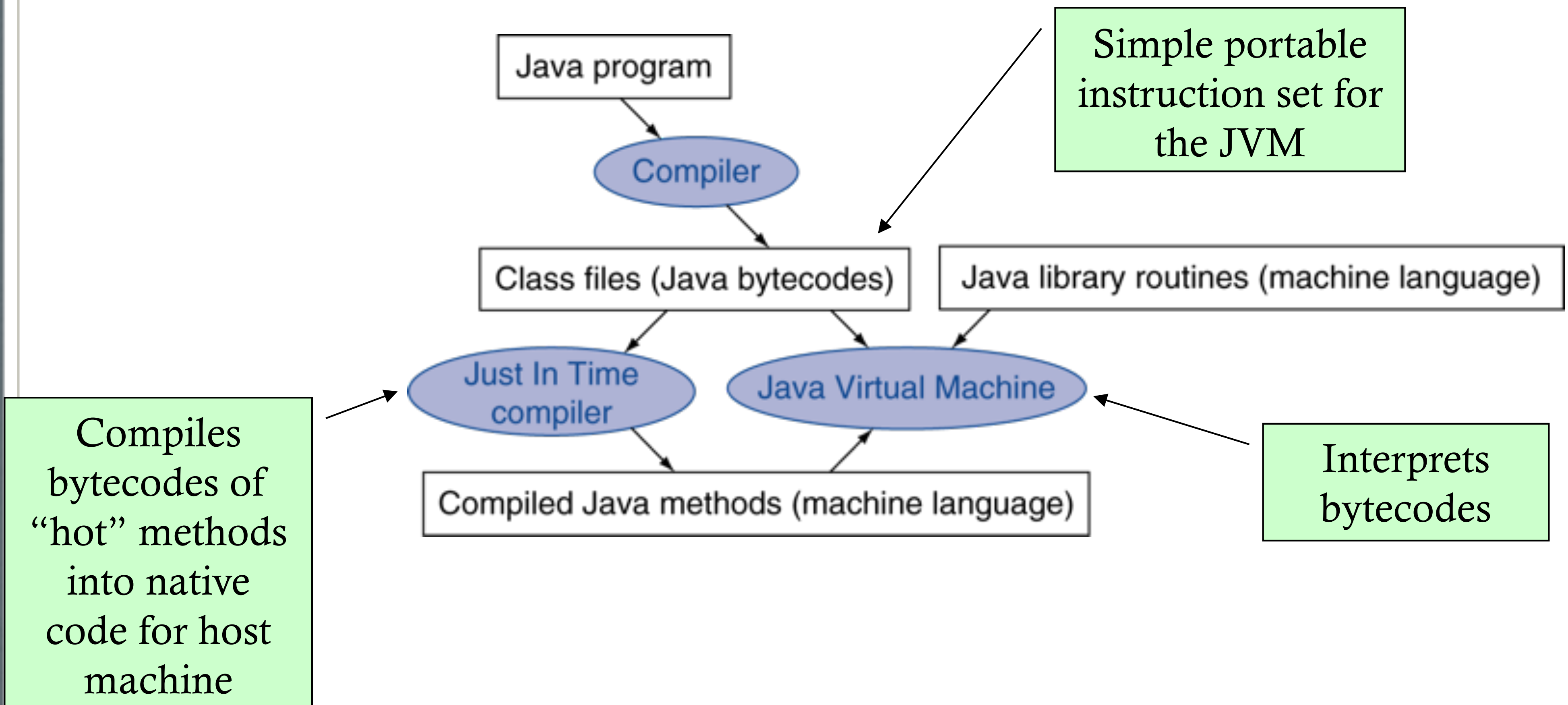
```
move $t0, $t1       →add $t0, $zero, $t1

blt $t0, $t1, L     →slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - \$at (register 1): assembler temporary

Label Resolution

- Labels need to be translated into addresses
  - Branch/jump target, data addresses
- Assembler (or compiler) translates program into machine instructions
  - Label → binary value if possible
  - Symbol table: remaining labels that are undefined in this module
- Linker uses relocation information and symbol table in each object module to resolve undefined labels
  - DLL may delay library linking to runtime

Starting Java Applications



Effect of Language, Compiler, and Algorithm

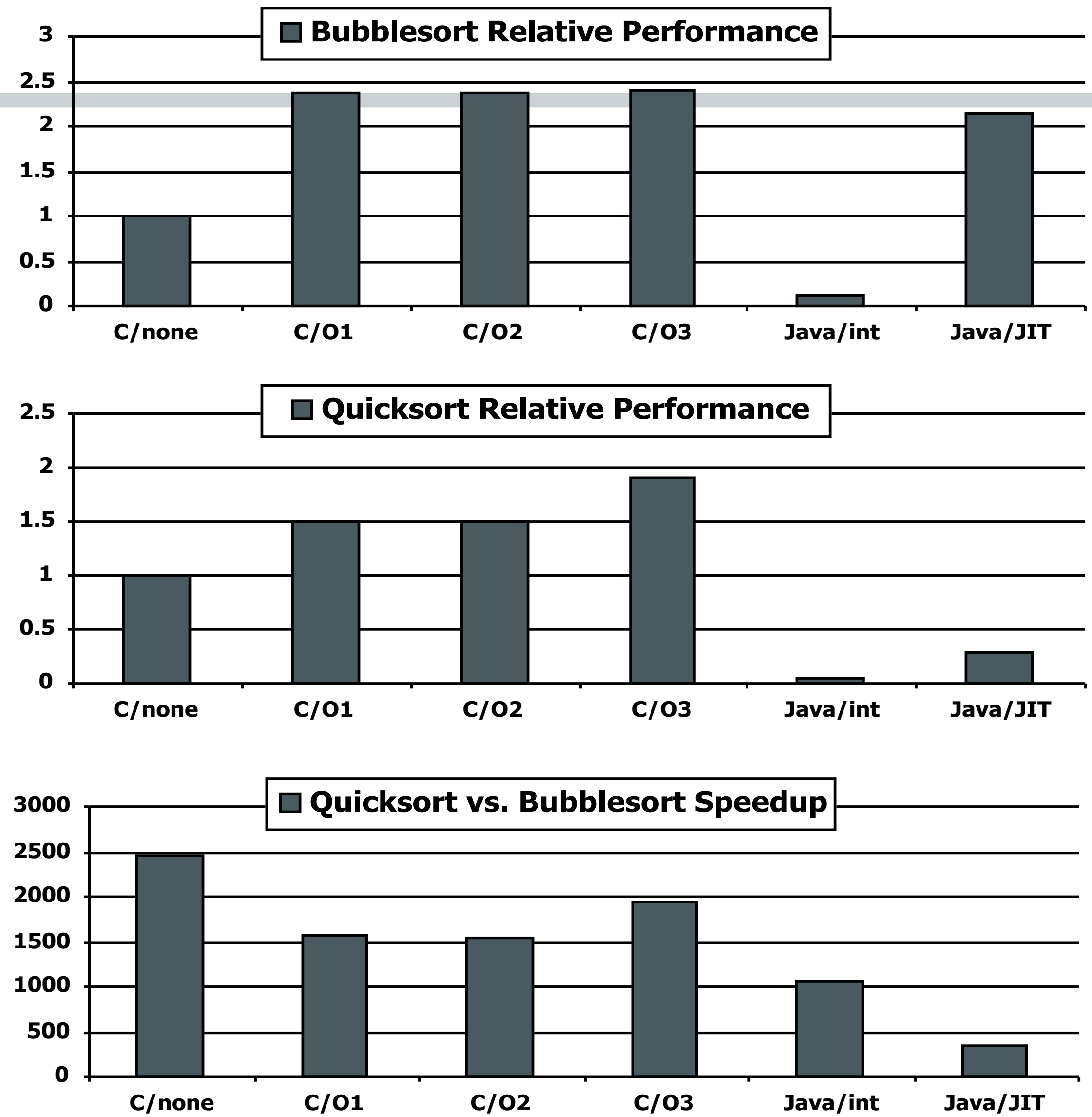


Fig. 2.29

Example: Data Races

Processor P1	Processor P2
1.1 Read X from memory	2.1 Read X from memory
1.2 Add 1 to X (local copy)	2.2 Add 2 to X (local copy)
1.3 Write X to memory	2.3 Write X to memory

Possible sequence: **All updates are applied!**

Operation	P1	P2	Memory
			X = 100
1.1	X = 100		X = 100
1.2	X = 101		X = 100
1.3	X = 101		X = 101
2.1		X = 101	X = 101
2.2		X = 103	X = 101
2.3		X = 103	X = 103

Example: Data Races

Processor P1	Processor P2
1.1 Read X from memory	2.1 Read X from memory
1.2 Add 1 to X (local copy)	2.2 Add 2 to X (local copy)
1.3 Write X to memory	2.3 Write X to memory

Possible sequence: **Inconsistent / lost updates!**

Operation	P1	P2	Memory
			X = 100
1.1	X = 100		X = 100
2.1	X = 100	X = 100	X = 100
1.2	X = 101	X = 100	X = 100
2.2	X = 101	X = 102	X = 100
2.3	X = 101	X = 102	X = 102
1.3	X = 101	X = 102	X = 101

Synchronization

- Parallel computing: coordination needed between multiple cooperating tasks
  - Enforce a certain order / avoid data races
- Hardware support required to ensure atomic read/modify memory operations
  - Atomic: no other access to the location allowed between the read and write
  - Basis of building locks / critical sections / …
- Implementation options
  - A single atomic memory instruction
  - A pair of instructions (more efficient, may succeed/fail)

Synchronization in MIPS

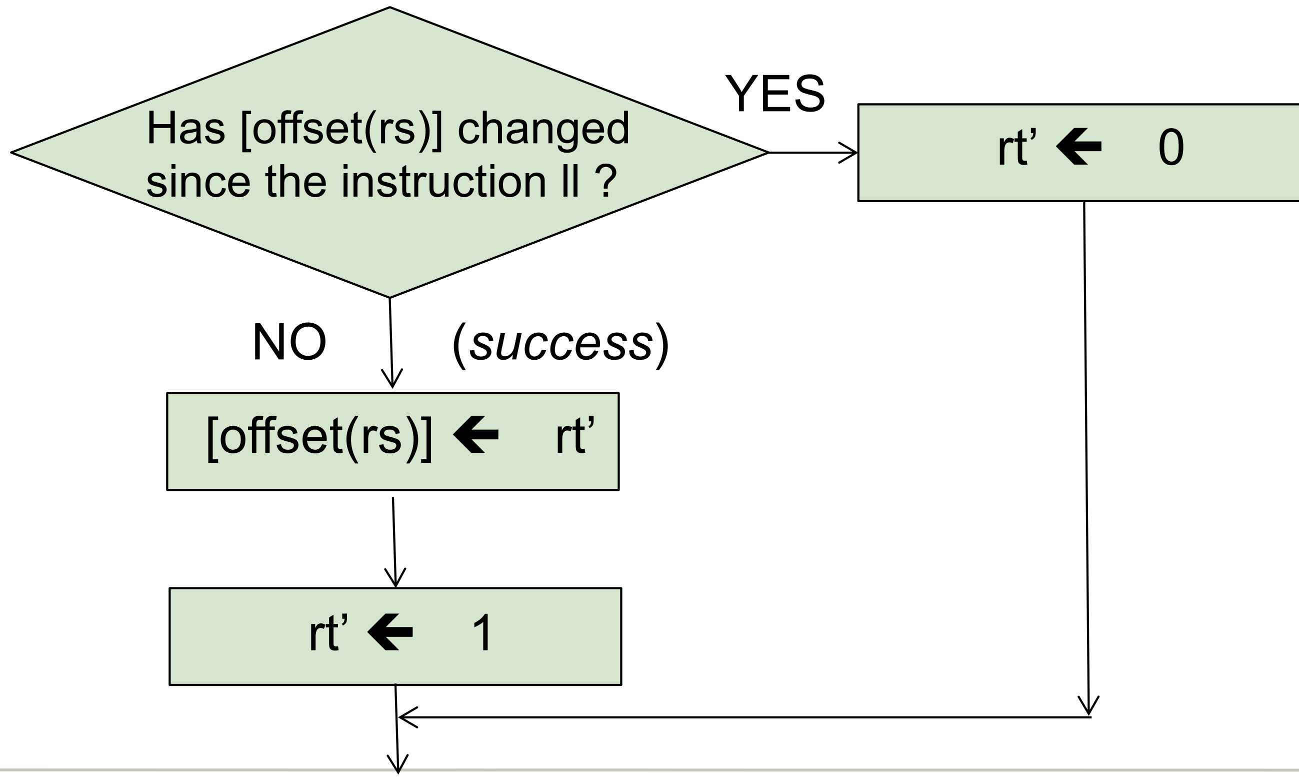
- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Always set rt as a report
  - Return 1 in rt
    - Succeed/atomic: memory location not changed since the previous ll
  - Return 0 in rt
    - Fail: if location has been changed since the previous ll

Synchronization in MIPS

ll rt, offset(rs)

rt ← [offset(rs)] (start monitoring [offset(rs)]

sc rt', offset(rs)



Uses of LL-SC

- Atomic updating one memory location
  - E.g. a[0] = a[0]+1

```
try: ll   $t0,offset($s1)     #load linked
     addi $t0,$t0,1           #increment by 1
     sc   $t0,offset($s1)     #store conditional
     beq  $t0,$zero,try       #re-try if fails

     << successfully updated offset($s1) >>
```

- Locks, critical sections, …

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

<pre>clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}</pre>
	<pre>clear2(int *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	       p = p + 1)
	    *p = 0;
	}</pre>

<pre>   move $t0,$zero   # i = 0
       j test1
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
test1: slt $t3,$t0,$a1 # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1</pre>
	<pre> move $t0,$a0 # p = & array[0]
	       sll $t1,$a1,2 # $t1 = size * 4
	       add $t2,$a0,$t1 # $t2 =
	                        # &array[size]
	       j test2
	loop2: sw $zero,0($t0) # Memory[p] = 0
	       addi $t0,$t0,4 # p = p + 4
	test2: slt $t3,$t0,$t2 # $t3 =
	                        #(p<&array[size])
	       bne $t3,$zero,loop2 # if (…)
	                           # goto loop2</pre>

*code changed slightly from textbook 2.14

Pitfalls

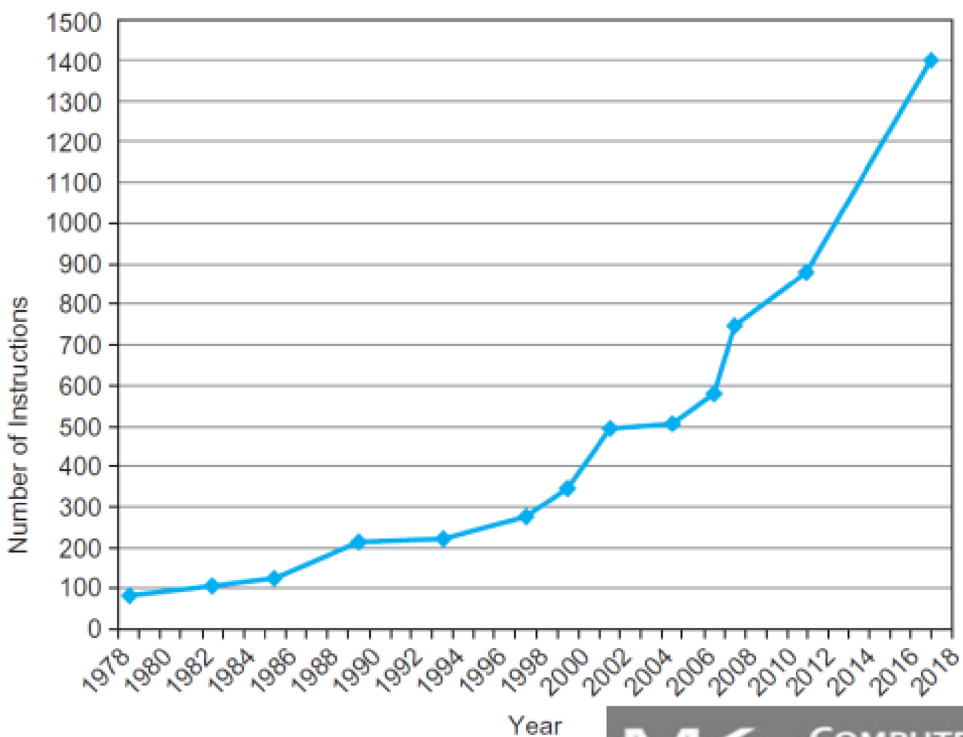
- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Fallacies

- Powerful instruction ⇒ 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
- Assembly programs: manually developed vs. automatically generated
  - Modern compilers improves a lot for performance
  - More lines of code ⇒ more errors and less productivity

Fallacies

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



x86 instruction set

MIPS Common Instructions

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

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: typical of RISC ISAs
  - c.f. x86