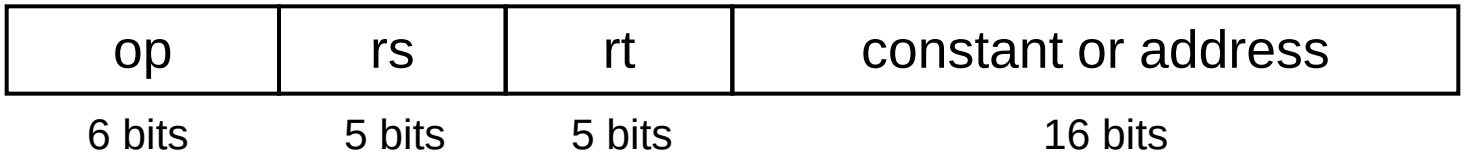
Chapter 2B

Instructions: Language of the Computer

Copyright © 2009 Elsevier, Inc. All rights reserved.

Branch Addressing

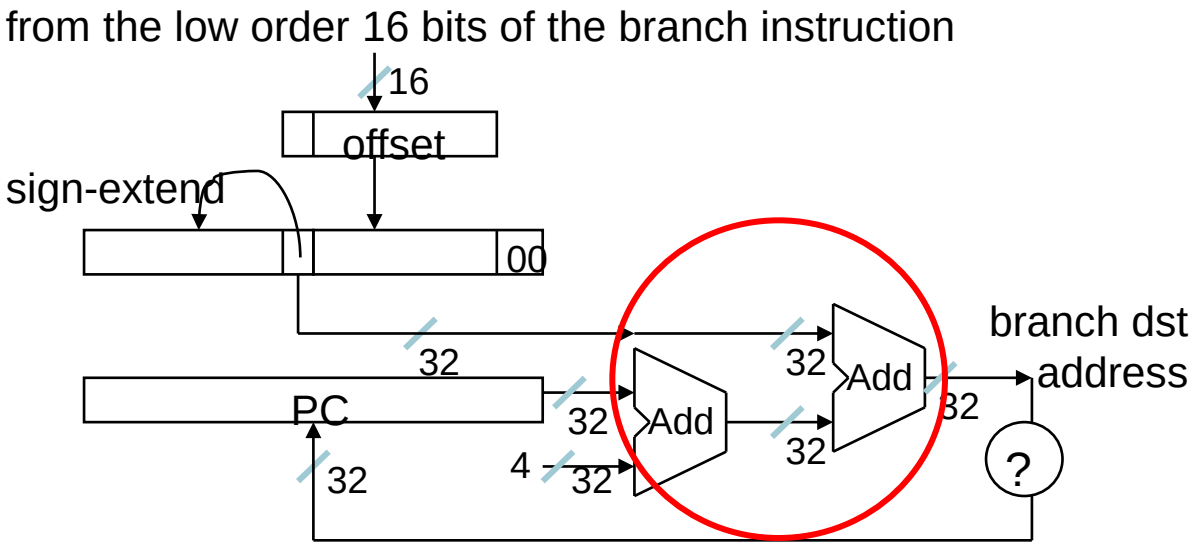
- Branch instructions specify
  - Opcode, two registers, target address
  - I-type
- Most branch targets are near branch
  - Forward or backward



  - PC-relative addressing
    - Target address = (PC+4) + offset × 4
    - PC already incremented by 4 by this time

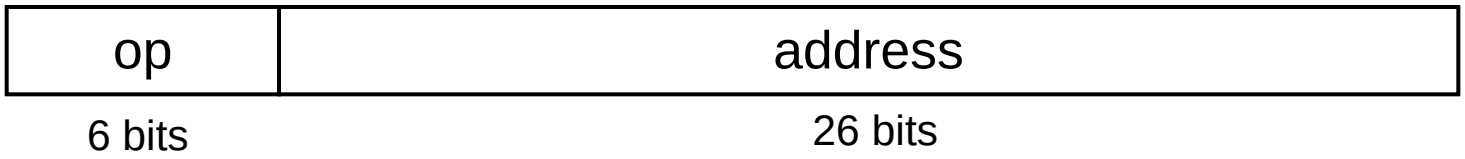
Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway



Jump Addressing

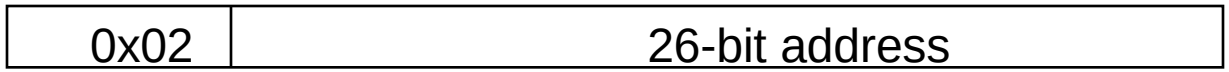
- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction
  - J-type

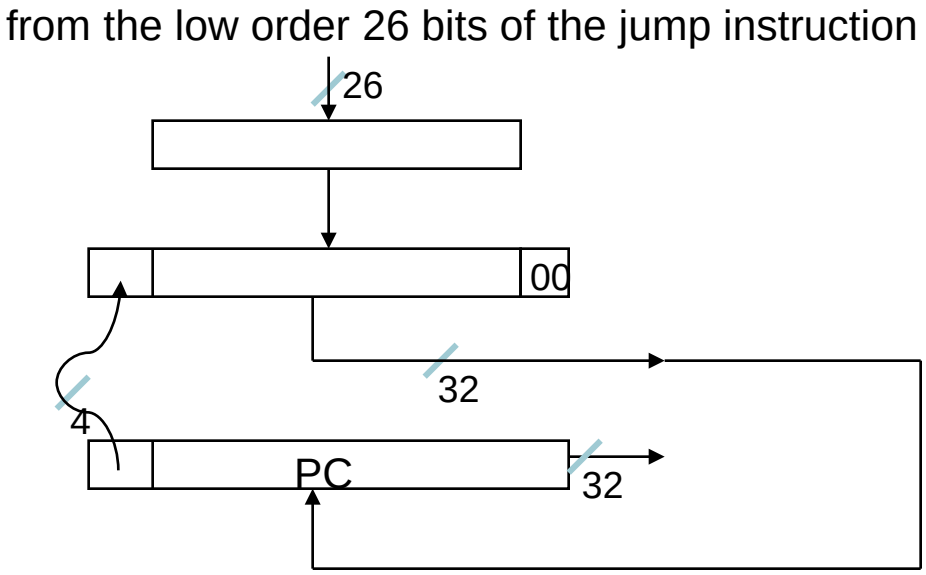


- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)
    = $PC_{31\ldots28}$: (address) : 00

Jump Addressing

- Instruction Format (J Format):





Target Addressing Example

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

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

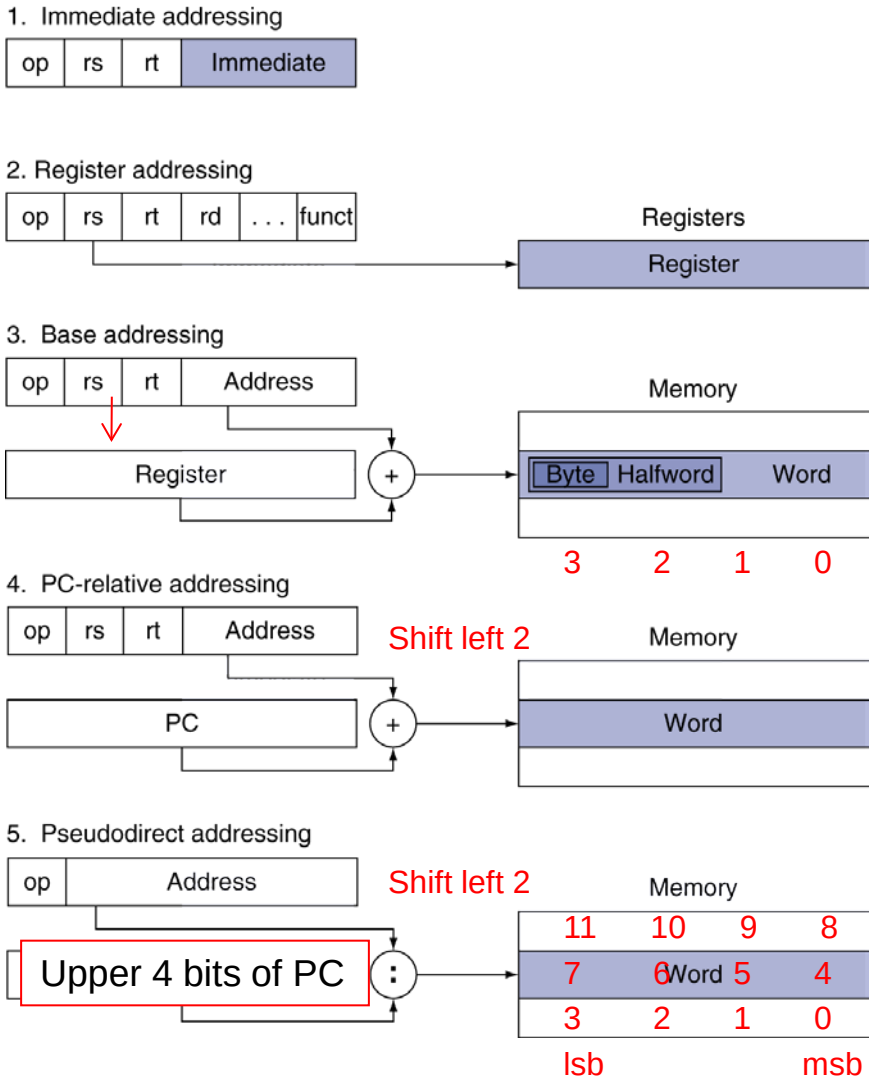
00080020(hex) = 0000 0000 0000 1000 0000 0000 0010 0000 (4 + 28 bits)
00080000(hex) = 0000 0000 0000 1000 0000 0000 0000 0000 (20000 <<2)
80024(hex)= 80012 + 4 + 2 <<2 = (PC + 4) + (offset << 2)

Conditional 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: …
```

Addressing Mode Summary

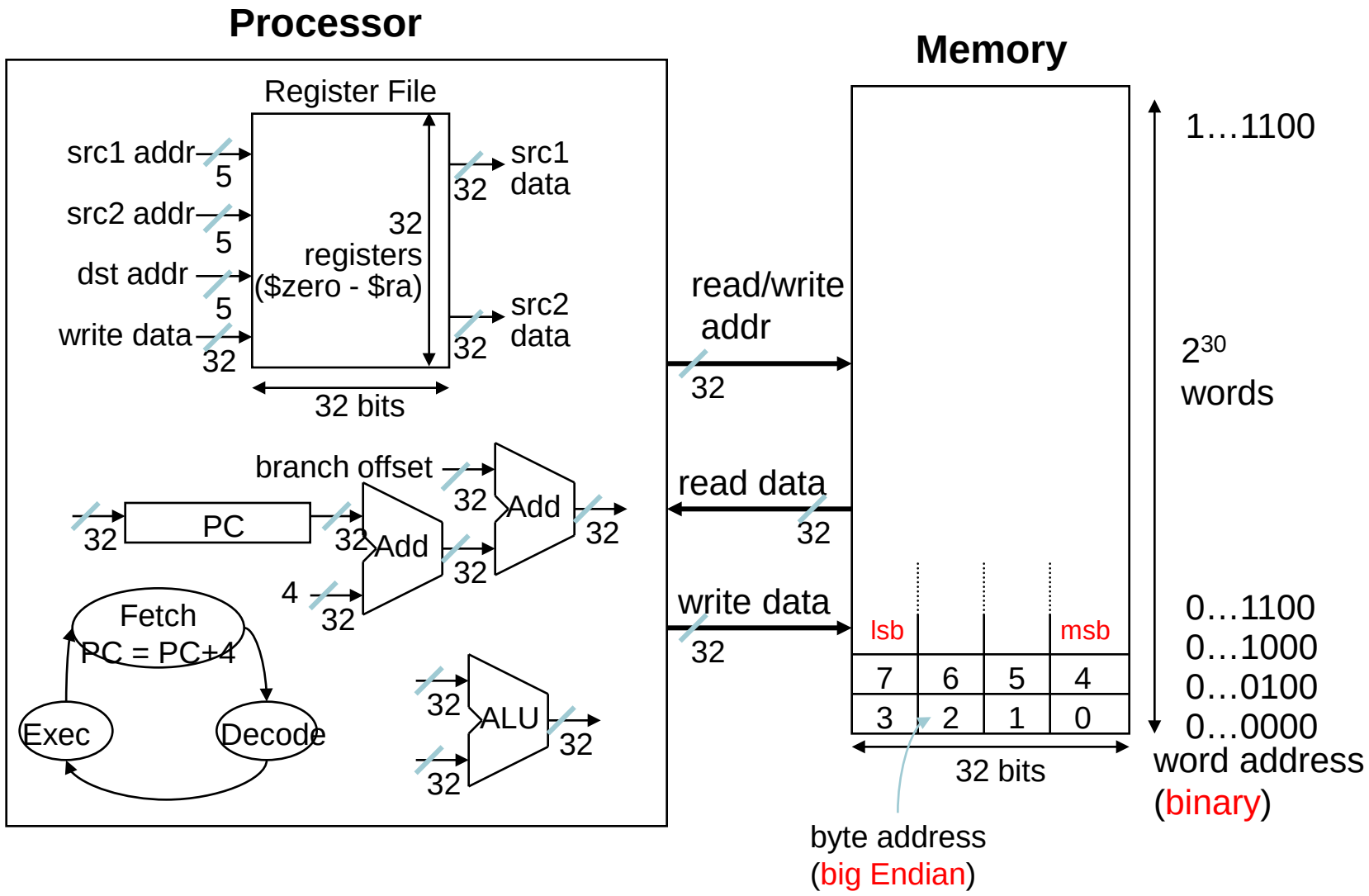


MIPS is big-endian

MIPS instruction formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

MIPS Organization So Far



Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses
- Lock and unlock synchronization operation
  - Used to create (memory) regions where only a single processor can operate, called mutual exclusion, as well as to implement more complex synchronization mechanisms.
- Hardware primitives required
  - Atomic (read and modify) memory operation
  - No other access to the location allowed between the read and write (modify)

Synchronization

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- In general, architects do not expect users to employ the basic hardware primitives.
  - The primitives are used by system programmers to build a synchronization library, a process that is often complex and tricky
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Atomic swap & a simple lock

- A simple lock
  - 0: to indicate that the lock is free: released
  - 1: to indicate that the lock is unavailable: locked
- A processor tries to set the lock by swapping 1 in a register with the memory address of a lock.
  - the returned value is 1 (already locked) if some other processor has already claimed and 0 otherwise.
  - In the latter case, the value is changed to 1 (newly locked), preventing any competing exchange in another processor from retrieving a 0.
- Implementing a single atomic swap with memory is challenging (why? two operations in an instruction)
  - An alternative is to have a pair of instructions

Load linked and store conditional

- These two instructions are used in sequence.
- If the content of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails.
- Load linked (locked): `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Store the value of register rt into memory specified
  - Succeeds if location not changed since the `ll` instruction
    - Returns 1 in rt of sc
  - Fails if location is changed after the `ll` instruction
    - Returns 0 in rt of sc

An atomic swap

- atomic swap (to test/set lock variable) between $s4 and the memory location specified by $s1

```
exch $s4, 0($s1) ; atomic swap
```

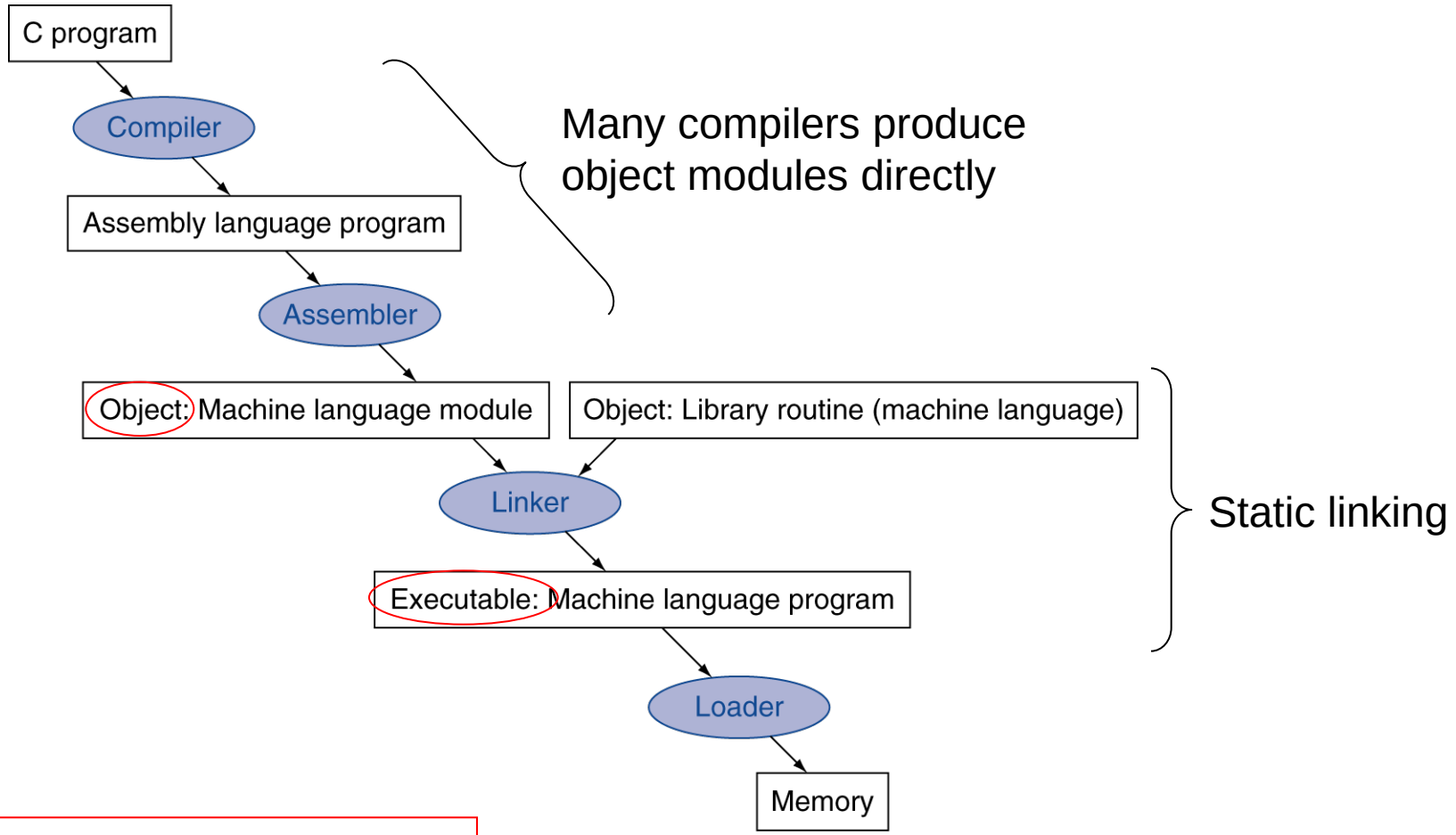
```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

- Although it is proposed for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor.

Implementation of ll and sc

- Typically implemented by keeping track of the address specified in the ll instruction in a register, called the link register.
- If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another sc), the link register is cleared.
- The sc instruction simply checks that its address matches that in the link register to determine its failure or success.
- Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions.
- Only register-register instructions can safely be permitted.

Translation and Startup



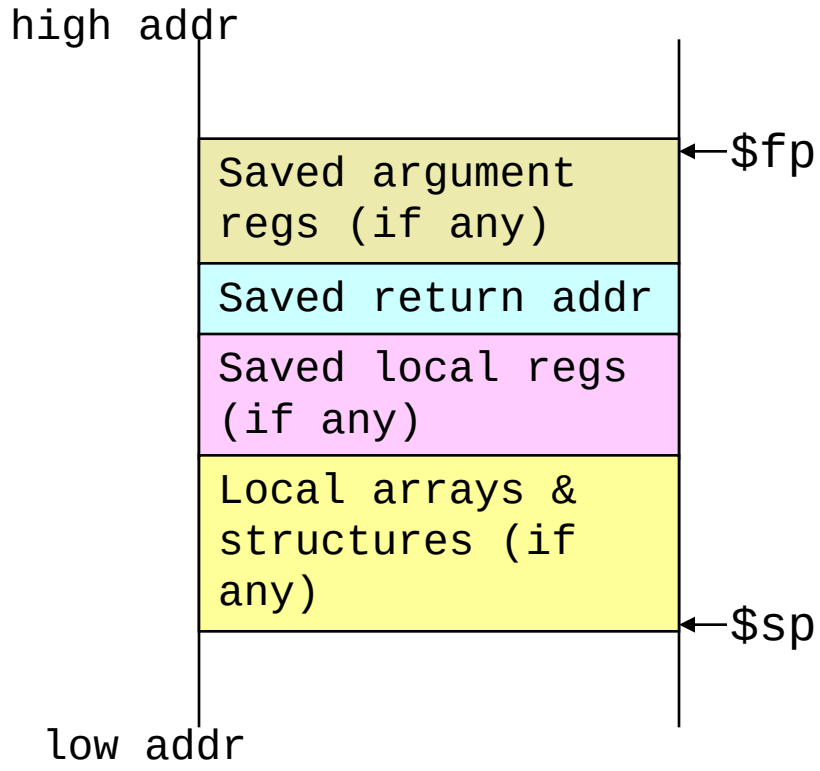
C code – x.c or .TXT
assembly code – x.s or .ASM
object code – x.o or .OBJ
executable – a.out or .EXE

Two storage classes in C

- Automatic variables: local to a procedure, discarded when procedure exits
- Static variables: exist across exits from and entries to procedures
- To simplify access to static data, MIPS software reserves another register, called the global register or $gp.

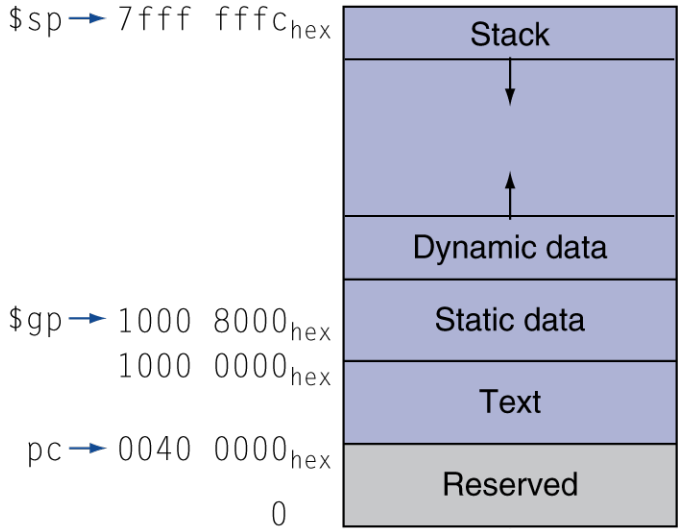
Allocating Space on the Stack

- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
  - The frame pointer (`$fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
    - `$fp` is initialized using `$sp` on a call and `$sp` is restored using `$fp` on a return

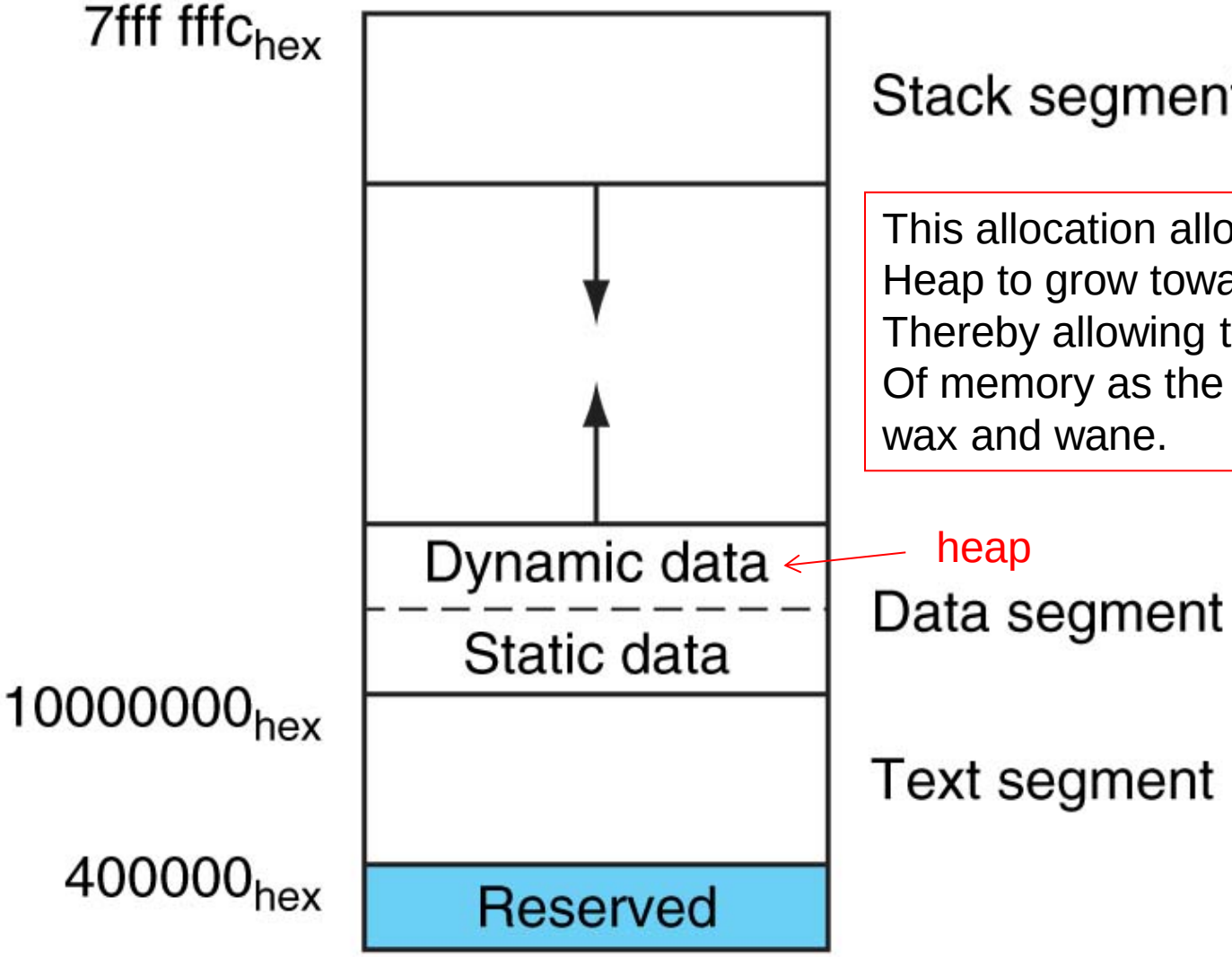


Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±16-bit offsets into this segment (1000 0000 – 1000 ffff)
- Dynamic data: heap
  - e.g., malloc in C, new in Java
- Stack: automatic storage



Layout of memory



This allocation allows the stack and Heap to grow toward each other, Thereby allowing the efficient use Of memory as the two segments wax and wane.

Memory allocation

- Controlled by programs in C
- It is the source of many common and difficult bugs.
- Forgetting to free space leads to a "memory leak," which eventually uses up so much memory that the operating system may crash.
- Freeing space too early leads to "dangling pointer," which can cause pointers to point to thing that the program never intended.
- Java uses automatic memory allocation and garbage collection just to avoid such bugs.

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Example alphabets in Unicode

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.16 Example alphabets in Unicode. Unicode version 4.0 has more than 160 “blocks,” which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16−32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see *www.unicode.org*.

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)     lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)    lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)     sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

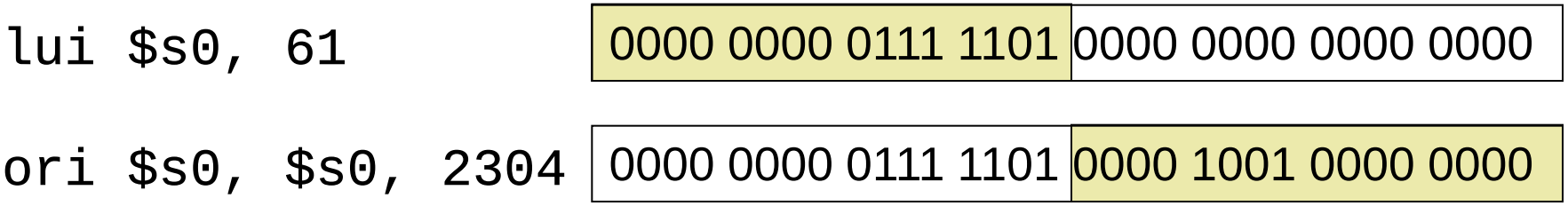
- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0 to use it for i
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rt, constant`
  - lui: load upper immediate
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0



Effect of lui instruction

The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

Linker and loader

- A loader does program loading
- A linker does symbol resolution
- Either can do relocation
- There have been all-in-one linking loaders that do all three functions
  - Program loading
  - Relocation
  - Symbol resolution

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: 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

MIPS Register Convention

Name	Register Number	Usage	Preserve on call by callee?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Object file header: size and location of each piece, source file name, creation date
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: a list of the places in the object code that have to be fixed up by the linker, which depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for debugger: source code, line number, data structure

Linking Object Modules

- Produces an executable image
  1. Find library routines used by the program
  2. Merges code and data modules
  3. Resolve labels (determine their addresses)
  4. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Linker (link editor)

Ref: pp. 143 - 144

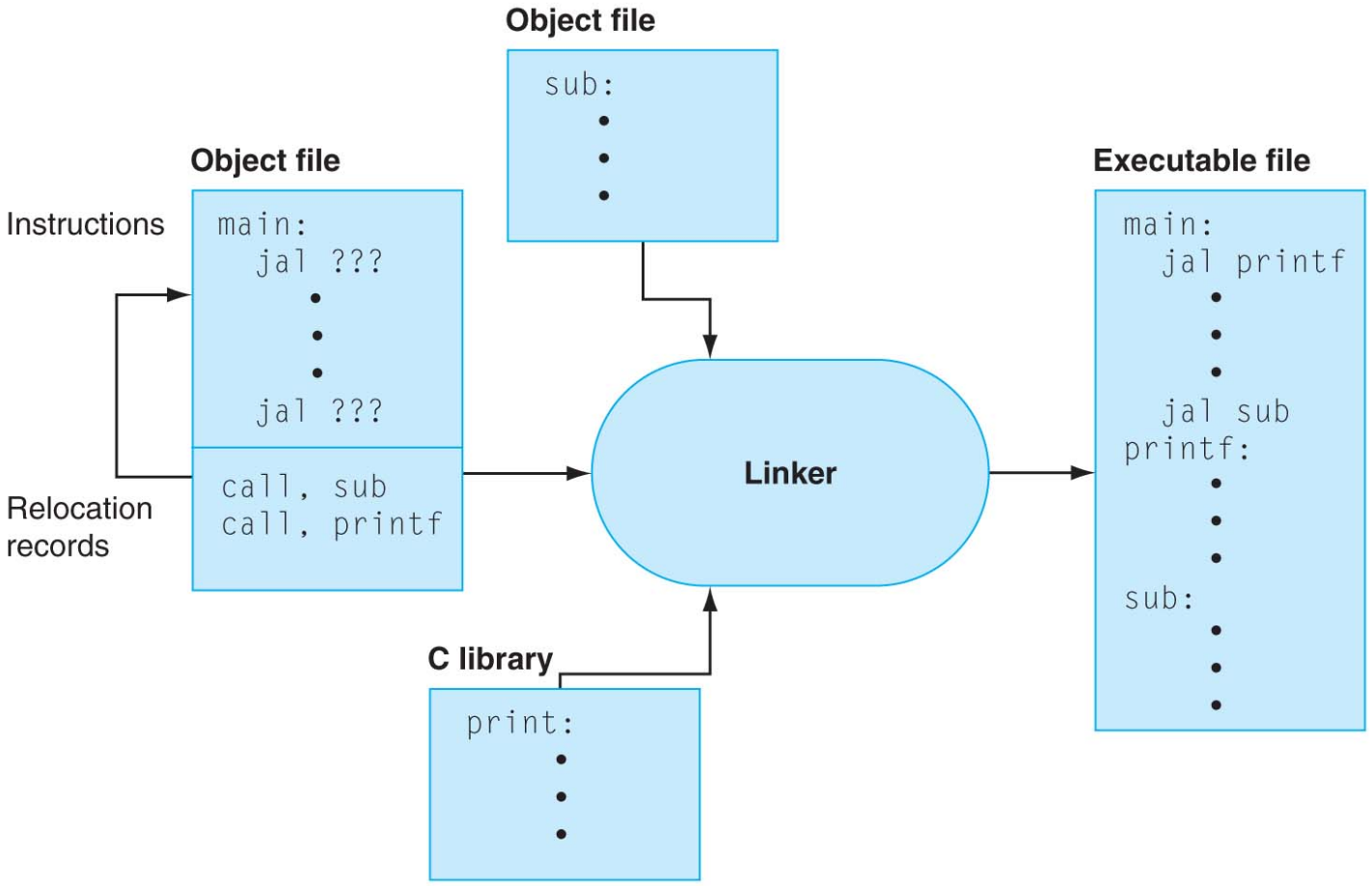


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Linking Object Modules

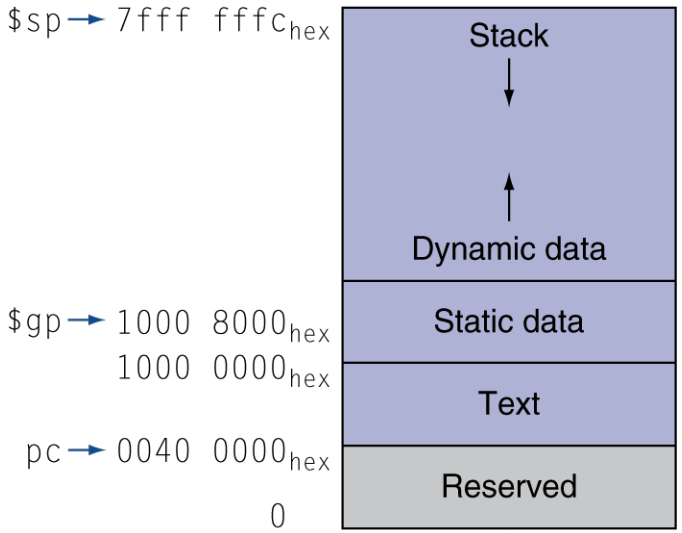
- Two object files

Object file header		
name: procedure A text size: 100 (hex) data size: 20 (hex)		
Text segment address	instruction	
0	lw $a0, 0($gp)	
4	jal 0	
…	…	
Data segment 0	(X)	
…	…	
Relocation info addr	I type	Depend
0	lw	X
4	jal	B
Symbol table label	address	
X	-	
B	-	

Object file header		
name: procedure B text size: 200 (hex) data size: 30 (hex)		
Text segment address	instruction	
0	sw $a1, 0($gp)	
4	jal 0	
…	…	
Data segment 0	(Y)	
…	…	
Relocation info addr	I type	Depend
0	sw	Y
4	jal	A
Symbol table label	address	
Y	-	
A	-	

Executable module

Executable file header	
Text size: 300 (hex) data size: 50 (hex)	
Text segment	
address	instruction
0040 0000	lw $a0, 8000($gp)
0040 0004	jal 10 0040
…	…
0040 0100	sw $a1, 8020($gp)
0040 0104	jal 10 0000
Data segment	
address	
1000 00000	(X)
…	…
1000 00020	(Y)
…	…



8000(hex) = 1000 0000 0000 0000
(16-bit 2's complement number)
= - 8000 (hex)

8020(hex) = 1000 0000 0010 0000
(16-bit 2's complement number)
= - 8000 (hex) + 0020 (hex)

8000($gp) = 1000 8000 – 8000 = 1000 00000; 0:010 0040<<2 = 0040 0100

8020($gp) = 1000 8020 – 8000 + 0020 = 1000 00020; 0:010 0000<<2 = 0040 0000

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create (virtual) address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments (if any) to the main program on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Statically Linked Libraries

- The fastest way to call library routines
- A few disadvantages
  - The library routines become part of the executable code. If a new version of the library is released, the statically linked program keeps using the old version.
  - It loads all routines in the library that are called anywhere in the executable even if those calls are not executed. The library can be large relative to the program. (for example, the standard C library is 2.5 MB)

Dynamic Linking Libraries (DLL)

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
- Initial version of DLL
  - The loader ran a dynamic linker
  - Downside: it still linked all routines of library that might be called.

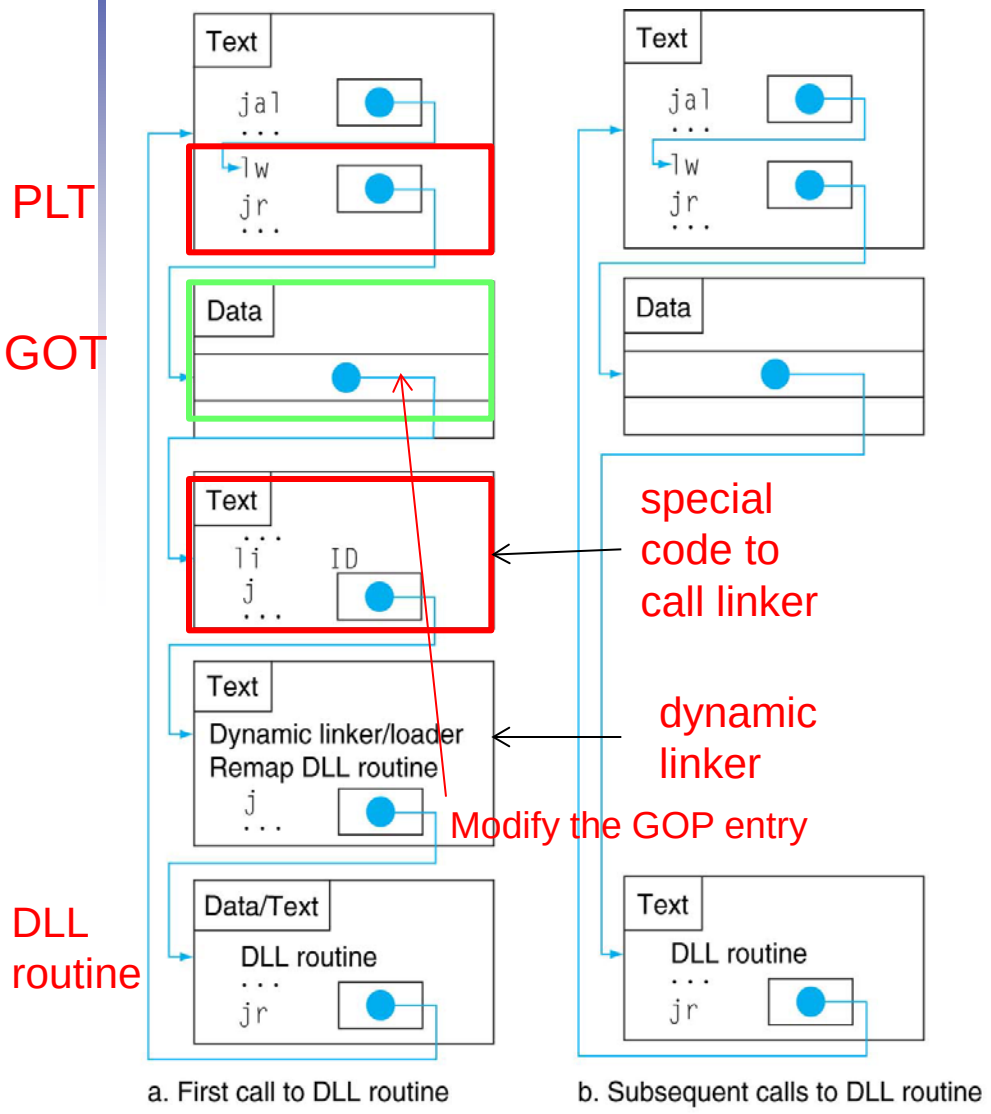
Lazy procedure linkage

- Each routine is linked only after it is called.
- It relies on another level of indirection, as shown in Fig. 2.22.
  - It starts with nonlocal routines calling a set of dummy routine in [PLT] at the end of the program, with one entry per nonlocal routine.
  - These dummy entries each contain an indirect jump after loading an offset from [GOT] .
  - The first time the library routine is called, the program calls the dummy entry and follows the indirect jump.

Lazy procedure linkage

- It initially points to code that puts a number in a register to identify the desired library routine and then jumps to the dynamic linker/loader.
- The linker/loader finds the desired routine remaps the GOT entry by changing the address in the indirect jump location to point to that routine. It then jump to the routine.
- When the routine completes, it returns to the original calling site.
- Thereafter, the call to the library jump indirectly to the routine without the extra hops.

Lazy procedure linkage



PLT: procedure linkage table
* A level of indirection for position independent code (PIC)
* A set of dummy routines at the end of the program, each entry per nonlocal routine.
* Each entry contains an indirect jump after loading an offset from GOT

GOT: global offset table
* Each PLT entry has a corresponding GOT entry which was initially set to code that put the routine ID in a register and then jump to the linker.

Dynamic linker:
* Find the desired routine with ID
* Relocate it and change its GOT entry
* Jump to it.

Lazy Linkage

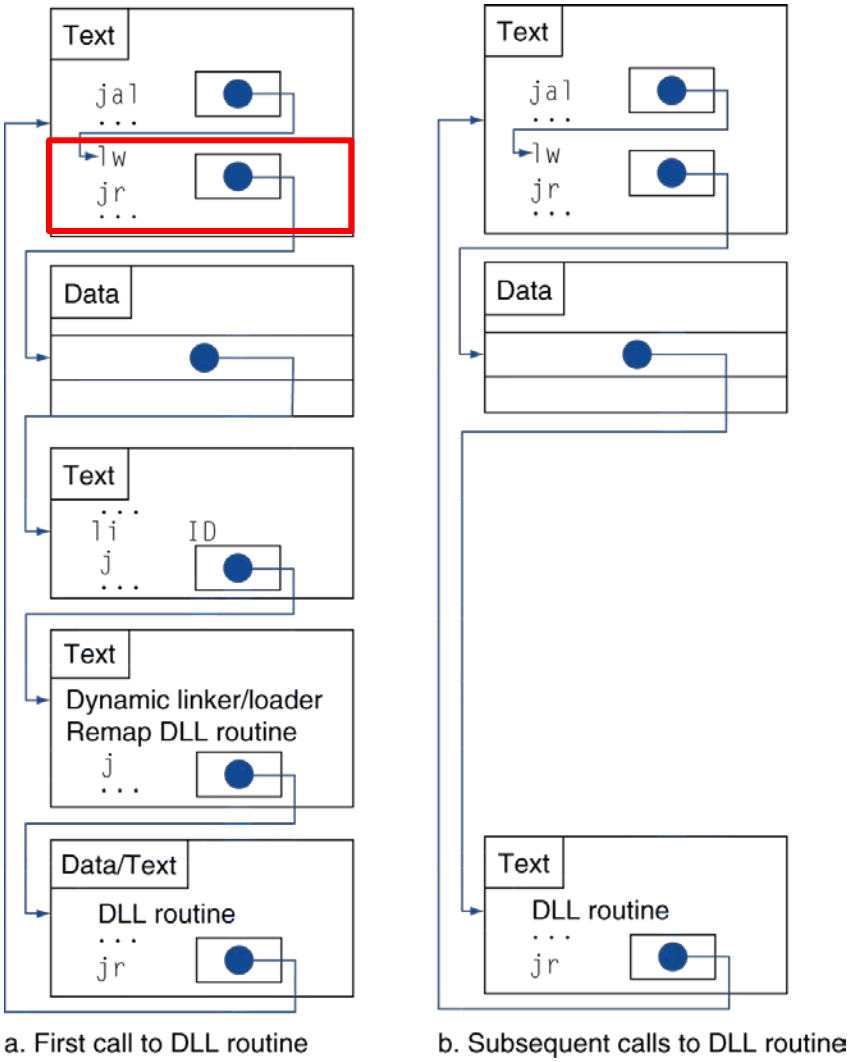
PLT: Procedure linkage table

GOT: Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



DLL Summary

- speed program startup
- require extra space for information needed for dynamic linking
- a good deal of overhead for the first call
  - only an indirect jump for subsequent calls
- Microsoft's Window relies extensively on DLL and it is also the default when executing programs on UNIX systems today.

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

- v in $a0, k in $a1, temp in $t0

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n){
  int i, j;
  for (i = 0; i < n; i++) {
   for (j = i–1;
         j >= 0 && v[j]>v[j+1]; j--) {
        swap(v,j);
   }
  }
 }
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
          move $s2, $a0            # save $a0 into $s2
          move $s3, $a1            # save $a1 into $s3
          move $s0, $zero          # i = 0
for1tst:  slt  $t0, $s0, $s3       # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, –1        # j = i – 1
for2tst:  slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2         # $t1 = j * 4
          add  $t2, $s2, $t1       # $t2 = v + (j * 4)
          lw   $t3, 0($t2)         # $t3 = v[j]
          lw   $t4, 4($t2)         # $t4 = v[j + 1]
          slt  $t0, $t4, $t3       # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
          move $a0, $s2            # 1st param of swap is v (old $a0)
          move $a1, $s1            # 2nd param of swap is j
          jal  swap                # call swap procedure
          addi $s1, $s1, –1        # j –= 1
          j    for2tst             # jump to test of inner loop
exit2:    addi $s0, $s0, 1         # i += 1
          j    for1tst             # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

The Full Procedure

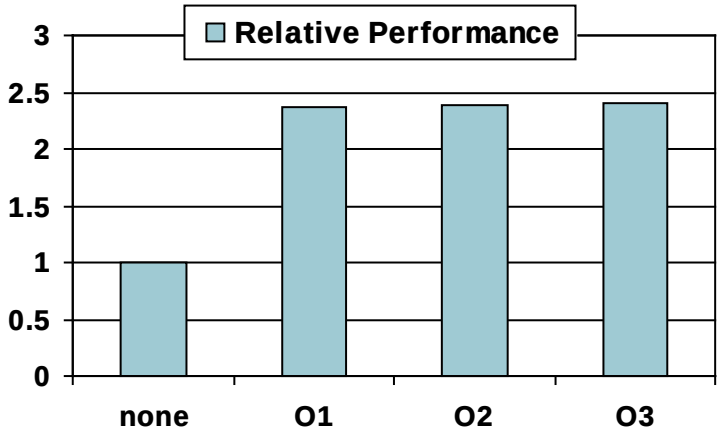
```
sort:    addi $sp,$sp, –20      # make room on stack for 5 registers
         sw $ra, 16($sp)        # save $ra on stack
         sw $s3,12($sp)         # save $s3 on stack
         sw $s2, 8($sp)         # save $s2 on stack
         sw $s1, 4($sp)         # save $s1 on stack
         sw $s0, 0($sp)         # save $s0 on stack
         …                      # procedure body
         …
         exit1: lw $s0, 0($sp)  # restore $s0 from stack
         lw $s1, 4($sp)         # restore $s1 from stack
         lw $s2, 8($sp)         # restore $s2 from stack
         lw $s3,12($sp)         # restore $s3 from stack
         lw $ra,16($sp)         # restore $ra from stack
         addi $sp,$sp, 20       # restore stack pointer
         jr $ra                 # return to calling routine
```

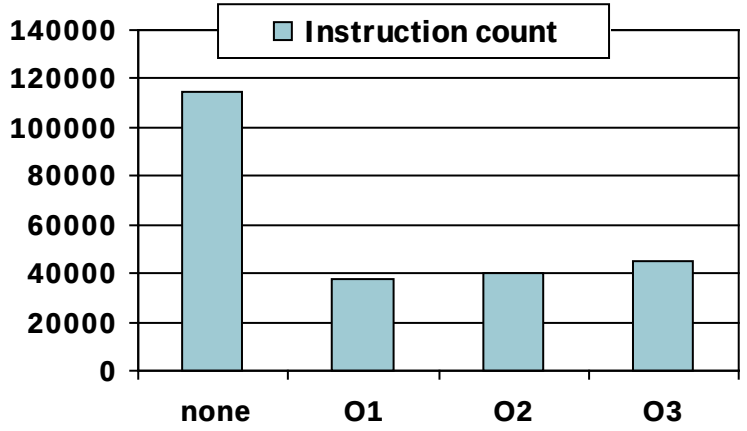
Optimization types

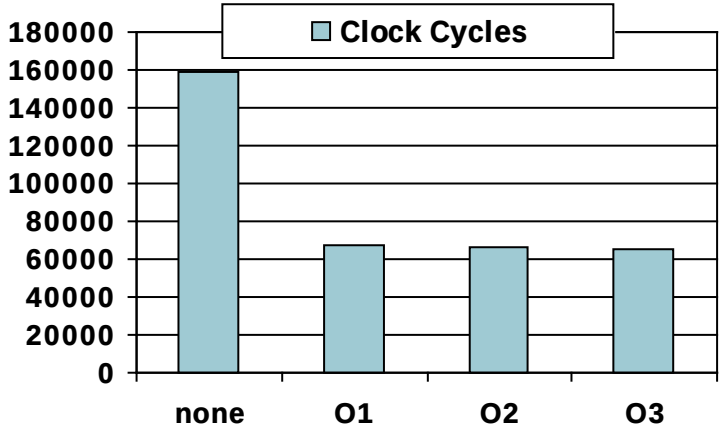
Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

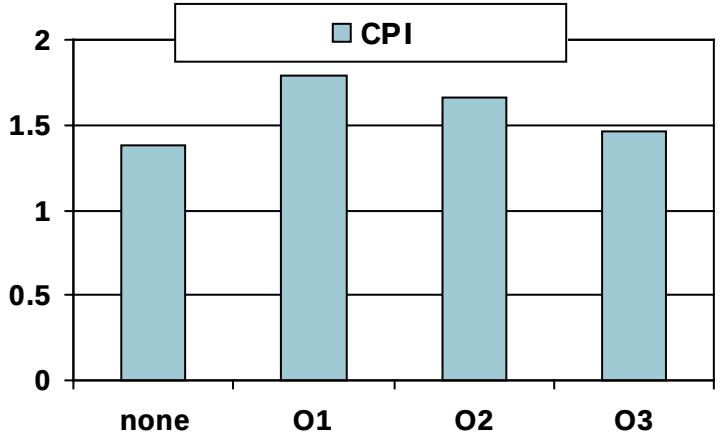
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

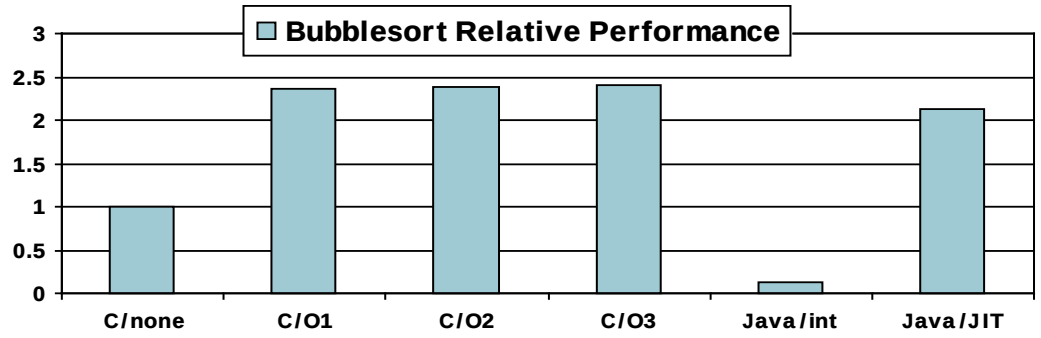


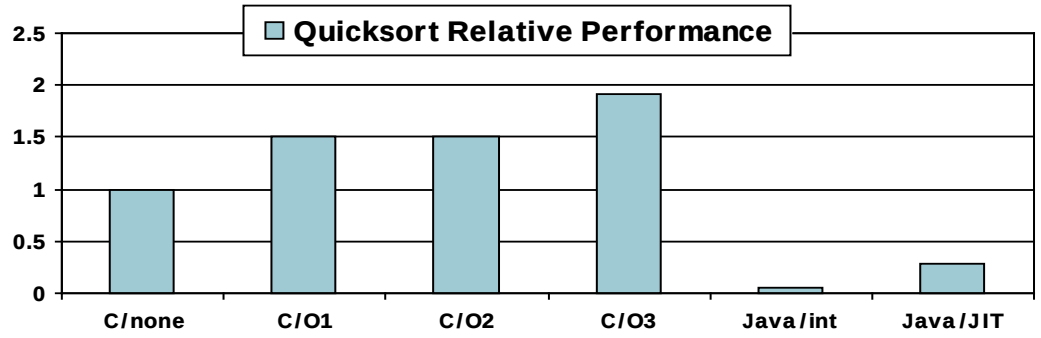


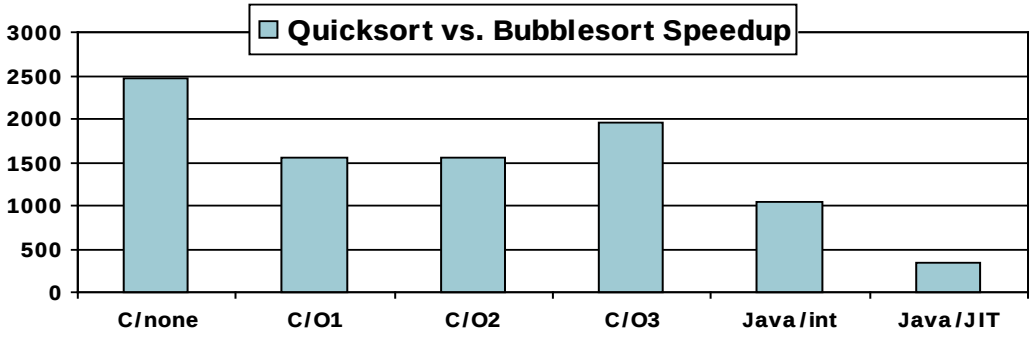




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

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 and Array

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

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 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 (…)
                           # goto loop1
```

```
       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
  - Induction variable elimination
  - Better to make program clearer and safer

Compiling C and Interpreting Java

Homework: read section 2.15 in CD

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

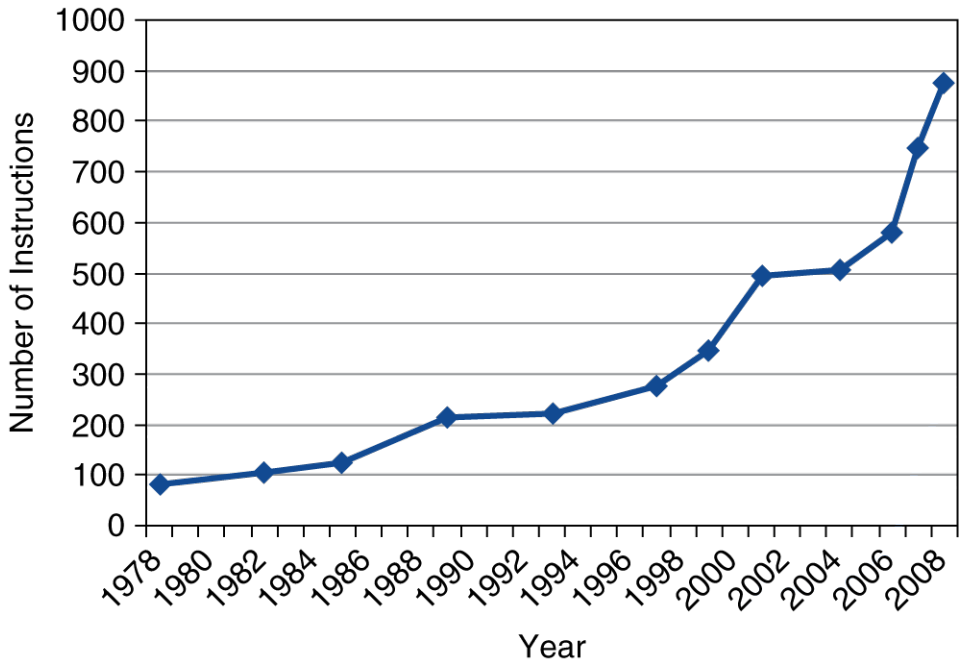
- Uses four condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

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
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - 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

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: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

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

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%