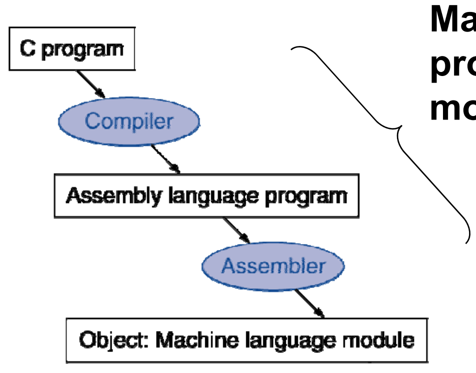
Chapter Two (3/3)

Translation



Many compilers produce object modules directly

What's a Compiler?

- **Compiler: a program that accepts as input a program text in a certain language and produces as output a program text in another language, *while preserving the meaning of that text*.**
- **The text must comply with the syntax rules of whichever programming language it is written in.**
- **A compiler's complexity depends on the syntax of the language and how much abstraction that programming language provides.**
  - **A C compiler is much simpler than C++ Compiler**
- **Compiler executes *before* compiled program runs**

Assembly and Pseudo-instructions

- **Turning textual MIPS instructions into machine code called assembly, program called assembler**
  - **Calculates addresses, maps register names to numbers, produces binary machine language**
  - **Textual language called assembly language**
- **Can also accept instructions convenient for programmer but not in hardware**
  - **Load immediate (li) allows 32-bit constants, assembler turns into lui + ori (if needed)**
  - **Load double (ld) uses two lwc1 instructions to load a pair of 32-bit floating point registers**
  - **Called Pseudo-Instructions**
- **Pseudoinstructions: figments of the assembler's imagination**

```
move $t0, $t1    →    add $t0, $zero, $t1
blt $t0, $t1, L  →    slt $at, $t0, $t1     ; $at: assembler
                      bne $at, $zero, L     ;      temporary ($r1)
```

Producing an Object Module

- Assembler (or compiler) 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

Separate Compilation and Assembly

- No need to compile all code at once
- How put pieces together?

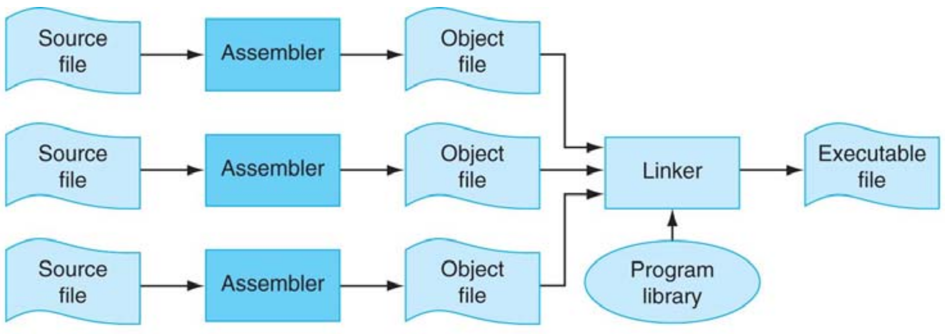
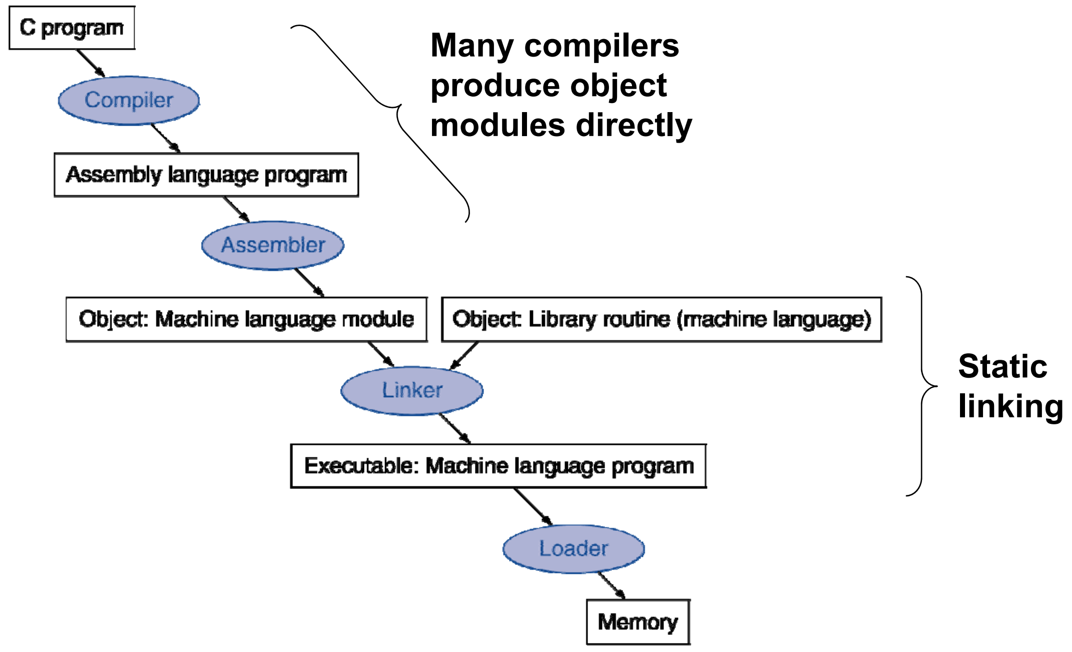


FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Linker Stitches Files Together

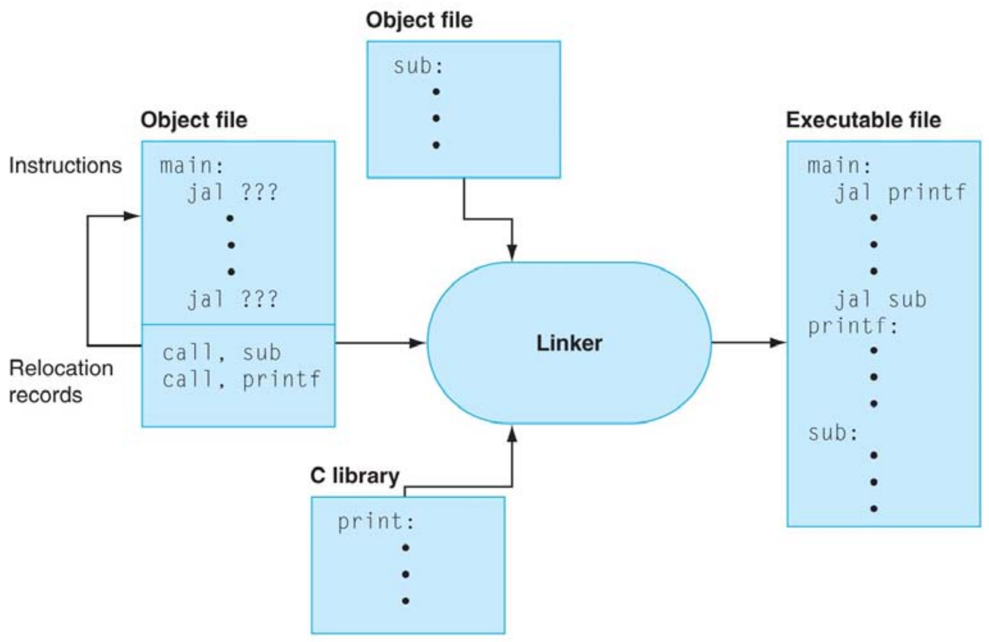


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.

Linking Object Modules

- Produces an executable image
  - 1. Merges segments
  - 2. Resolve labels (determine their addresses)
  - 3. Patch location-dependent and external refs
- Often a slower than compiling
  - all the machine code files must be read into memory and linked together

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (cover later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments 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” systems call

Dynamic Linking

- Only link/load library procedure when it is called
  - Automatically picks up new library versions
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
- Dynamic linking is default on UNIX and Windows Systems

- 1st time pay extra overhead of DLL (Dynamically Linked Library), subsequent times almost no cost
- Compiler sets up code and data structures to find desired library first time
- Linker fixes up address at runtime so fast call subsequent times
- Note that return from library is fast every time

Dynamic Linkage

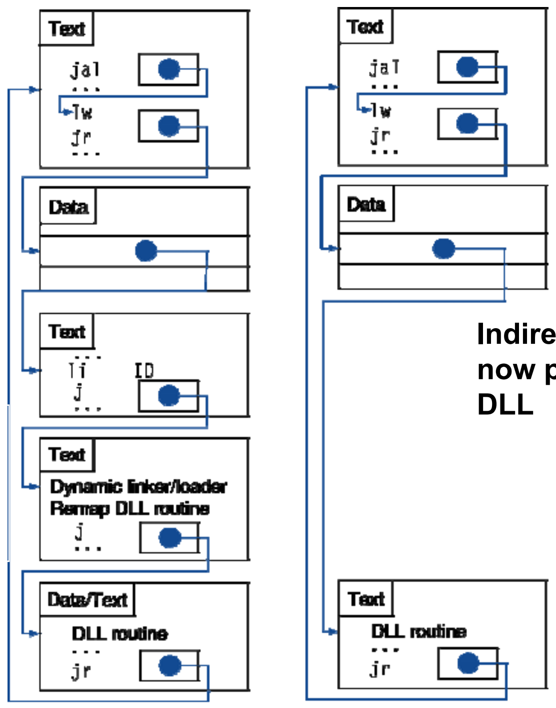
Call to DLL Library

Indirection table that initially points to stub code

Stub: Loads routine ID so can find desired library,
Jump to linker/loader

Linker/loader code finds desired library and edits jump address in indirection table, jumps to desired routine

Dynamically mapped code executes and returns



a. First call to DLL routine

b. Subsequent calls to DLL routine

Indirection table now points to DLL

Compiler Optimization

- gcc compiler options

-O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time

-O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code

-O3: Optimize yet more. All -O2 optimizations and also turns on the inline-functions, …

Typical Benefit of Compiler Optimization?

- What is a typical program?
  For now, try a toy program:
  BubbleSort.c

```
#define ARRAY_SIZE 20000
int main() {
  int iarray[ARRAY_SIZE], x, y, holder;

for(x = 0; x < ARRAY_SIZE; x++)
    for(y = 0; y < ARRAY_SIZE-1; y++)
      if(iarray[y] > iarray[y+1]) {
        holder = iarray[y+1];
        iarray[y+1] = iarray[y];
        iarray[y] = holder;
      }
}
```

Unoptimized MIPS Code

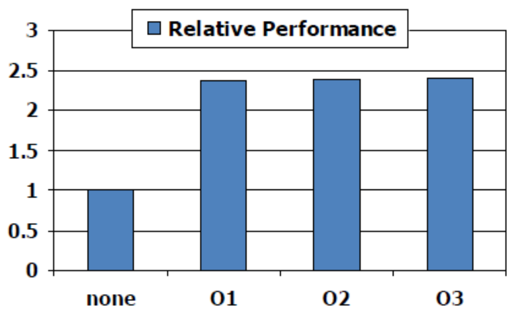
```
$L3:
        lw      $2,80016($sp)
        slt     $3,$2,20000
        bne     $3,$0,$L6
        j       $L4
$L6:
        .set    noreorder
        nop
        .set    reorder
        sw      $0,80020($sp)
$L7:
        lw      $2,80020($sp)
        slt     $3,$2,19999
        bne     $3,$0,$L10
        j       $L5
$L10:
        lw      $2,80020($sp)
        move    $3,$2
        sll     $2,$3,2
        addu    $3,$sp,16
        addu    $2,$3,$2
        lw      $4,80020($sp)
        addu    $3,$4,1
        move    $4,$3
        sll     $3,$4,2
        addu    $4,$sp,16
        addu    $3,$4,$3
        lw      $2,0($2)
        lw      $3,0($3)
        slt     $2,$3,$2
        beq     $2,$0,$L9
        lw      $3,80020($sp)
        addu    $2,$3,1
        move    $3,$2
        sll     $2,$3,2
        addu    $3,$sp,16
        addu    $2,$3,$2
        lw      $3,0($2)
        sw      $3,80024($sp
        lw      $3,80020($sp)
        addu    $2,$3,1
        move    $3,$2
        sll     $2,$3,2
        addu    $3,$sp,16
        addu    $2,$3,$2
        lw      $3,80020($sp)
        move    $4,$3
        sll     $3,$4,2
        addu    $4,$sp,16
        addu    $3,$4,$3
        lw      $4,0($3)
        sw      $4,0($2)
        lw      $2,80020($sp)
        move    $3,$2
        sll     $2,$3,2
        addu    $3,$sp,16
        addu    $2,$3,$2
        lw      $3,80024($sp)
        sw      $3,0($2)
$L11:
$L9:
        lw      $2,80020($sp)
        addu    $3,$2,1
        sw      $3,80020($sp)
        j       $L7
$L8:
$L5:
        lw      $2,80016($sp)
        addu    $3,$2,1
        sw      $3,80016($sp)
        j       $L3
$L4:
$L2:
        li      $12,65536
        ori     $12,$12,0x38b0
        addu    $13,$12,$sp
        addu    $sp,$sp,$12
        j       $31
```

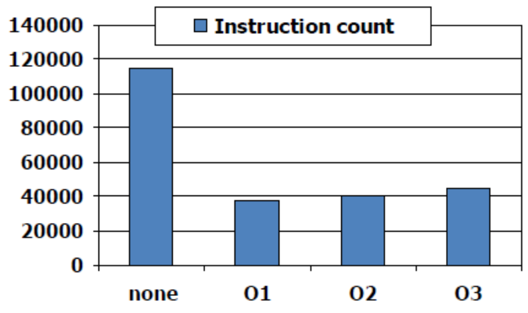
-O2 optimized MIPS Code

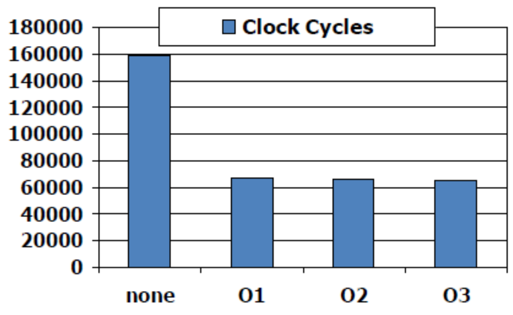
```
        li      $13,65536
        ori     $13,$13,0x3890
        addu    $13,$13,$sp
        sw      $28,0($13)
        move    $4,$0
        addu    $8,$sp,16
$L6:
        move    $3,$0
        addu    $9,$4,1
        .p2align 3
$L10:
        sll     $2,$3,2
        addu    $6,$8,$2
        addu    $7,$3,1
        sll     $2,$7,2
        addu    $5,$8,$2
        lw      $3,0($6)
        lw      $4,0($5)
        slt     $2,$4,$3
        beq     $2,$0,$L9
        sw      $3,0($5)
        sw      $4,0($6)
$L9:
        move    $3,$7
        slt     $2,$3,19999
        bne     $2,$0,$L10
        move    $4,$9
        slt     $2,$4,20000
        bne     $2,$0,$L6
        li      $12,65536
        ori     $12,$12,0x38a0
        addu    $13,$12,$sp
        addu    $sp,$sp,$12
        j       $31
        .
```

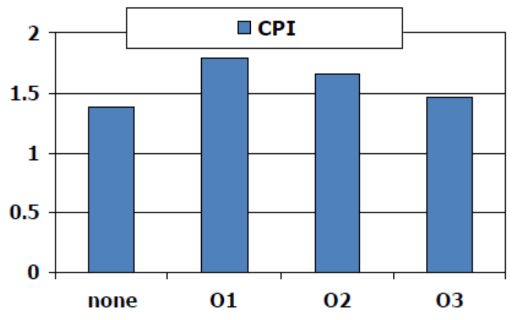
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

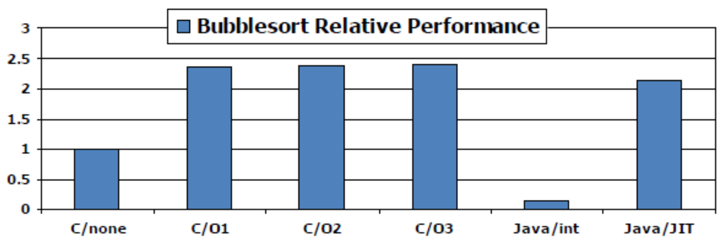


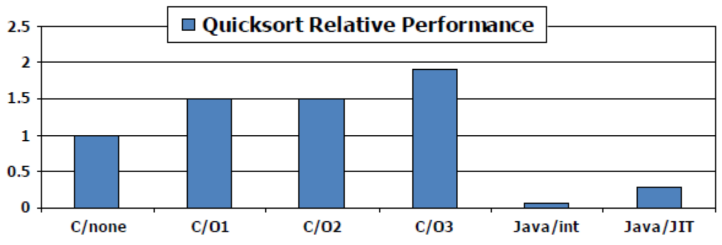


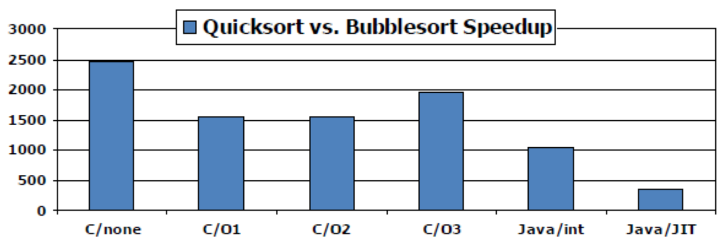




Effect of Language and Algorithm







What's an Interpreter?

- It reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Start executing quicker, but run much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- The interpreter stays around during execution
  - Unlike compiler, some work is done after program starts
- Writing an interpreter is much less work than writing a compiler

Compiler vs. Interpreter Advantages

Compilation:

- Faster Execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors before run program
- Compiler can optimize code

Interpreter:

- Easier to debug program
- Faster development time

Compiler vs. Interpreter Disadvantages

Compilation:

- Harder to debug program
- Takes longer to change source code, recompile, and relink

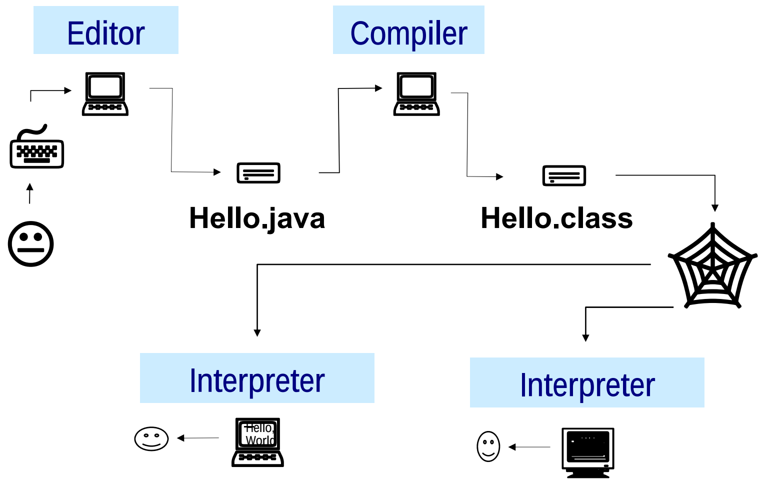
Interpreter:

- Slower execution times
- No optimization
- Need all of source code available
- Source code larger than executable for large systems
- Interpreter must remain installed while the program is interpreted

Java's Hybrid Approach:

Compiler + Interpreter

- A Java compiler converts Java source code into instructions for the *Java Virtual Machine (JVM)*
- These instructions, called *bytecodes*, are same for any computer / OS
- A CPU-specific Java interpreter interprets bytecodes on a particular computer



Why Bytecodes?

- Platform-independent
- Load from the Internet faster than source code
- Interpreter is faster and smaller than it would be for Java source
- Source code is not revealed to end users
- Interpreter performs additional security checks, screens out malicious code

Java Bytecodes (Stack) vs. MIPS (Reg.)

Category	Operation	Java bytecode	Size (bits)	MIPS Instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3

Alternative Architectures

- **Design alternative:**
  - **provide more powerful operations**
  - **goal is to reduce number of instructions executed**
  - **danger is a slower cycle time and/or a higher CPI**
- **Sometimes referred to as "RISC vs. CISC"**
  - **virtually all new instruction sets since 1982 have been RISC**
  - **VAX-11/780: minimize code size, make assembly language easy**

    instructions from 1 (NOP) to 56 (POLYG) bytes long!
- **We'll look at others**

ARM & MIPS Similarities

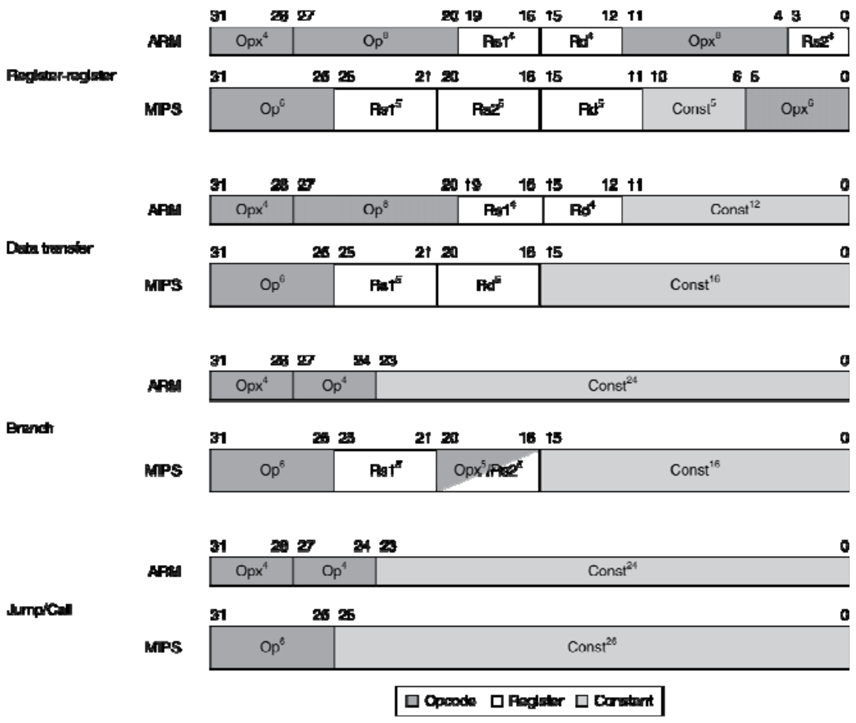
- **ARM: the most popular embedded core**
- **Similar basic set of instructions to MIPS**

	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

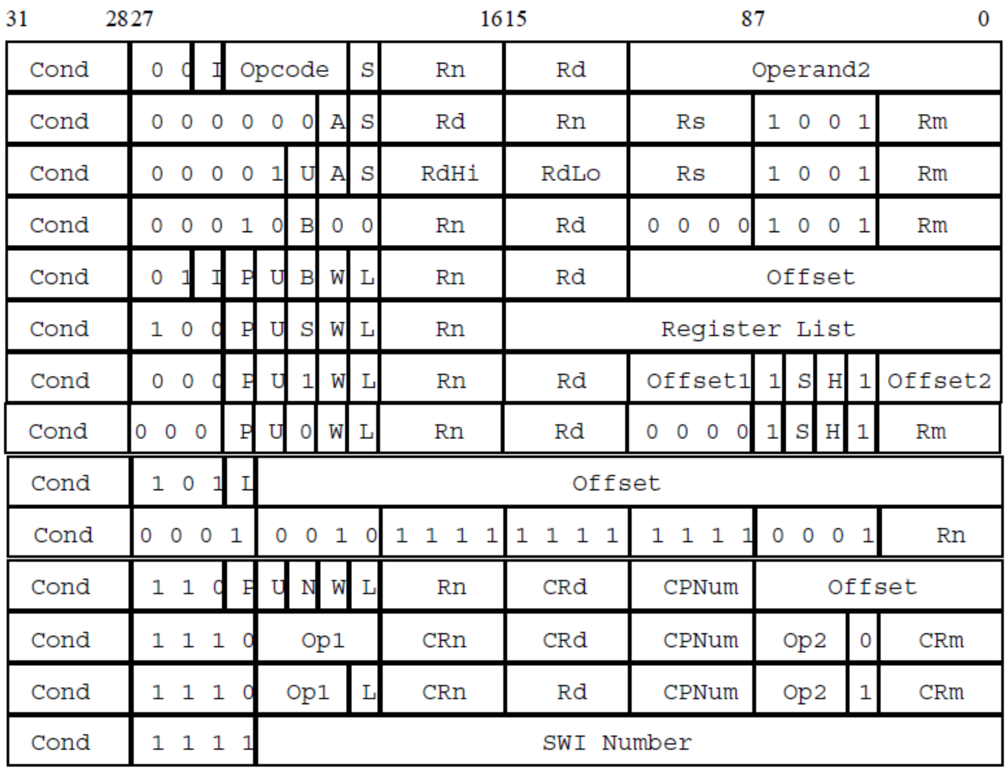
Compare and Branch in ARM

- Uses 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

Instruction Encoding



ARM Instruction Set Formats



31 2827 1615 87 0

Cond	0 0 I Opcode S	Rn	Rd	Operand2		
Cond	0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm
Cond	0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm
Cond	0 0 0 1 0 B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm
Cond	0 1 I P U B W L	Rn	Rd	Offset		
Cond	1 0 0 P U S W L	Rn	Register List			
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm
Cond	1 0 1 L	Offset				
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset	
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm
Cond	1 1 1 1	SWI Number				

PowerPC

- **Indexed addressing**
  - **example:** `lw $t1,$a0+$s3` **`#$t1=Memory[$a0+$s3]`**
  - **What do we have to do in MIPS?**

```
add $t0, $a0, $s3
lw $t1, 0($t0)
```

- **Update addressing**
  - **update a register as part of load (for marching through arrays)**
  - **example:** `lwu $t0,4($s3)`

    `#$t0=Memory[$s3+4];$s3=$s3+4`
  - **What do we have to do in MIPS?**

```
lw $t0, 4($s3)
addi $s3, $s3, 4
```

- **Others:**
  - **load multiple/store multiple: up to 32 words**
  - **a special counter register**

```
bc Loop, $ctr!=0
```

  - **What do we have to do in MIPS?**

```
Loop:   …
        addi, $t0, $t0, -1
        bne $t0, $zero, Loop
```

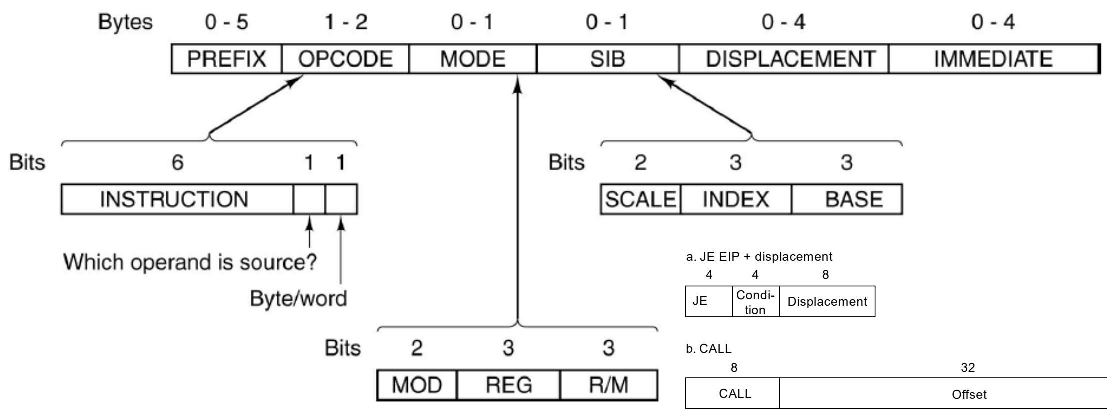
PowerPC Instruction Set Formats

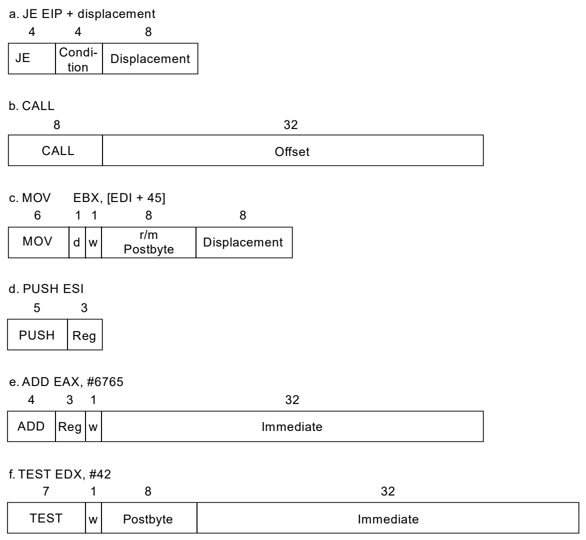
Format	0	6	11	16	21	26	30	31
D-form	opcd	tgt/src	src/tgt	immediate				
X-form	opcd	tgt/src	src/tgt	src	extended opcd			
A-form	opcd	tgt/src	src/tgt	src	src	extended opcd		Rc
BD-form	opcd	BO	BI	BD			AA	LK
I-form	opcd	LI					AA	LK

80x86

- **1978: The Intel 8086 is announced (16 bit architecture)**
- **1980: The 8087 floating point coprocessor is added**
- **1982: The 80286 increases address space to 24 bits, +instructions**
- **1985: The 80386 extends to 32 bits, new addressing modes**
- **1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)**
- **1997: MMX is added**
- **1999: The Pentium III added another 70 instructions (SSE)**
- **2001: Another 144 instructions (SSE2)**
- **2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)**
- **2004: Intel capitulates and embraces AMD64 (calls it EM64T, Extended Memory 64 Technology) and adds more media extensions(SSE3)**
- **2006: Intel adds SSE4 instructions, virtual machine support**
- **2007: Competition continues…**

Pentium 4 Instruction Set Formats (IA-32)

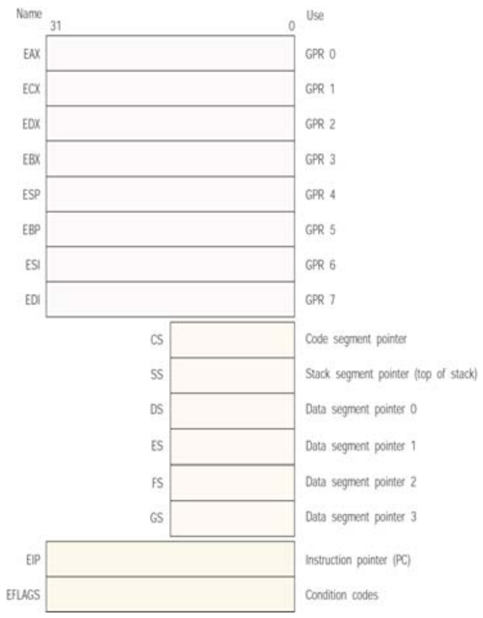




Basic x86 Addressing Modes

- **Two operands per instruction**

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate



- **Memory addressing modes**
  - **Address in register**
  - **Address = R_{base} + displacement**
  - **Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)**
  - **Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement**

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!