

Assemblers and Linkers

Goals for this Lecture

- Help you to learn about:
 - IA-32 machine language
 - The assembly and linking processes

Why Learn Machine Language

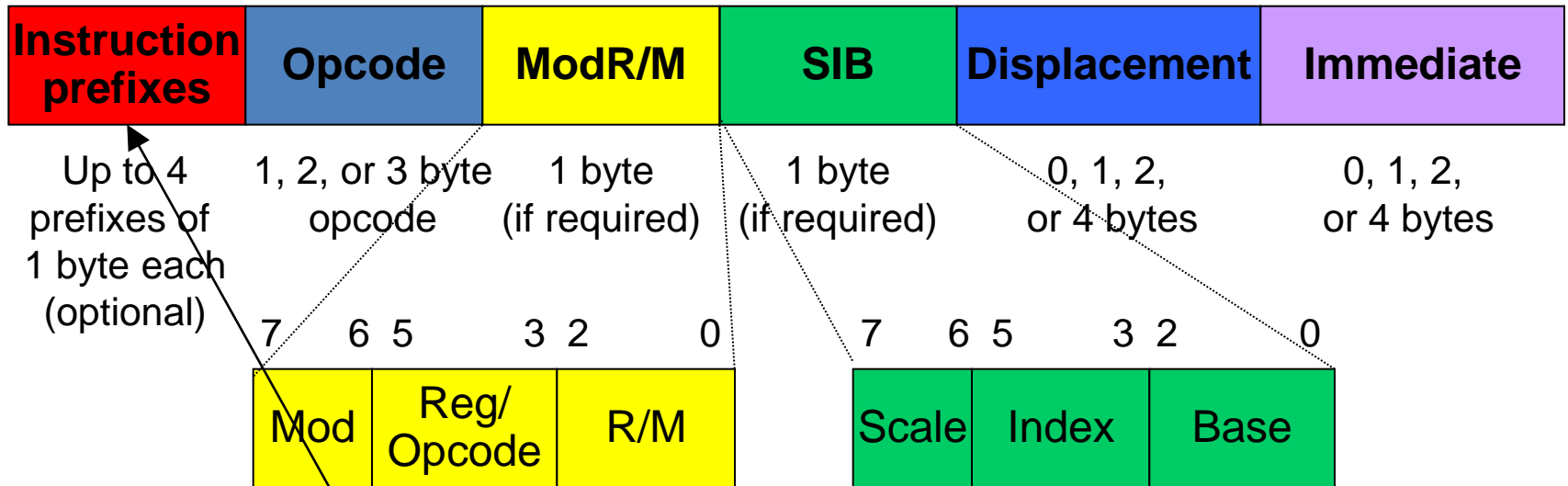
- Machine language is the last stop on the "language levels" tour
- A power programmer knows about the relationship between assembly language and machine language
- A systems programmer knows how an assembler translates assembly language to machine language

Part 1: Machine Language

IA-32 Machine Language

- IA-32 machine language
 - Difficult to generalize about IA-32 instruction format
 - Many (most!) instructions are exceptions to the rules
 - Generally, instructions use the following format shown in following slides
- We'll go over
 - The format of instructions
 - Two example instructions
- Just to give a sense of how it works...

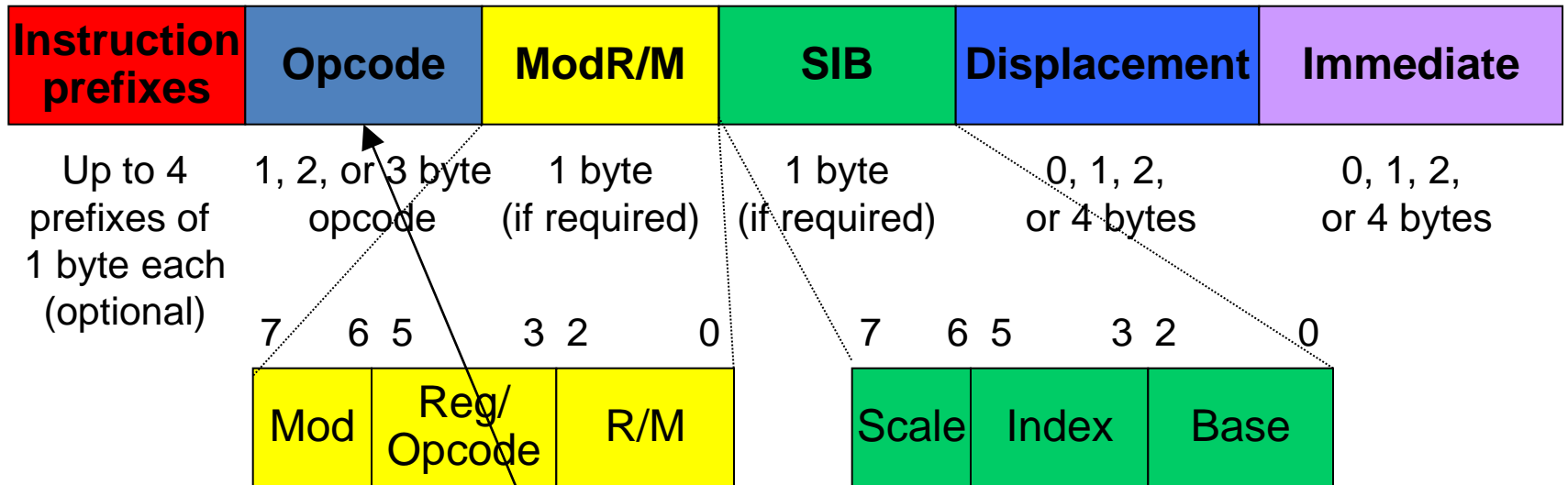
IA-32 Instruction Format



Instruction prefix

- Sometimes a repeat count
- Rarely used; don't be concerned

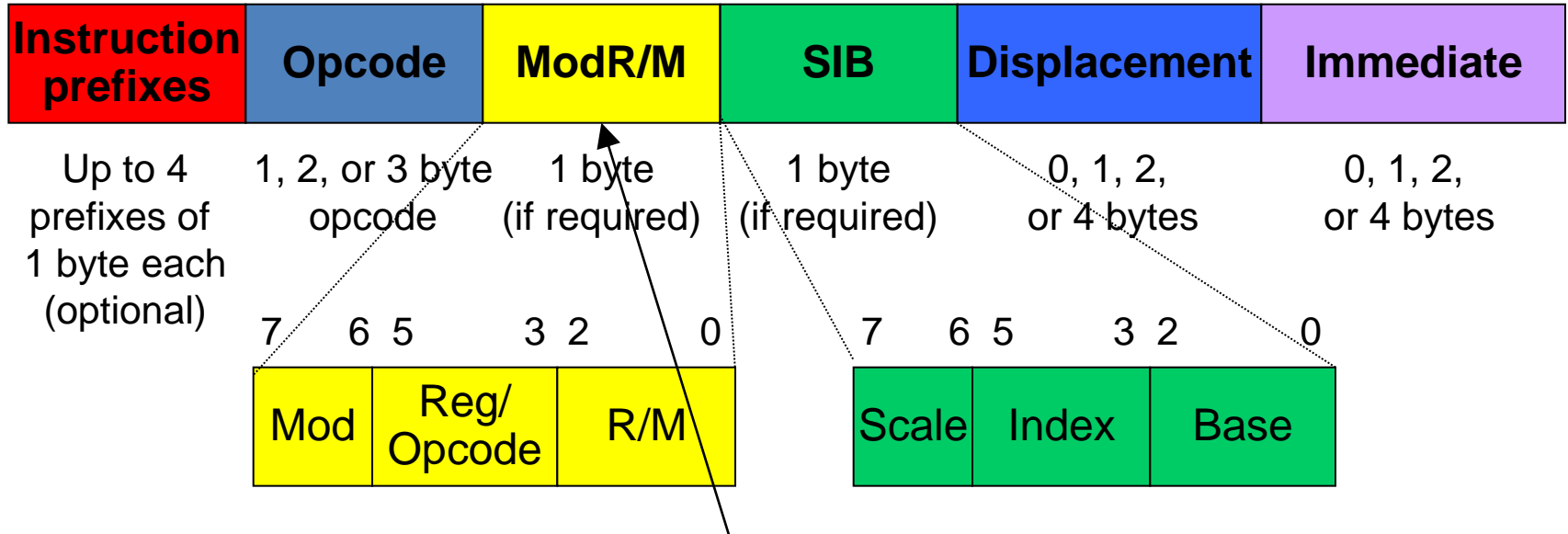
IA-32 Instruction Format (cont.)



Opcode

- Specifies which operation should be performed
- Add, move, call, etc.

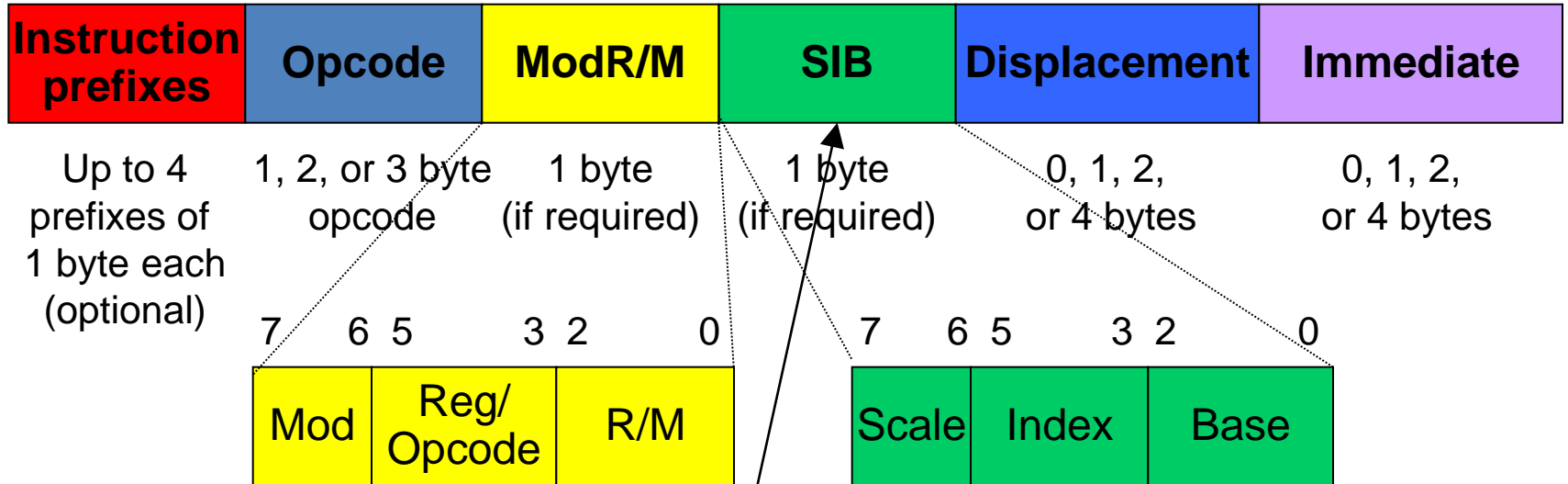
IA-32 Instruction Format (cont.)



ModR/M

- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes denotes a register:
 000 = EAX/AL; 011 = EBX/BL; 001 = ECX/CL; 010 = EDX/DL;
 110 = ESI/DI; 111 = EDI/BH; 101 = EBP/CH; 110 = ESP/AH
- Sometimes contains an extension of the opcode

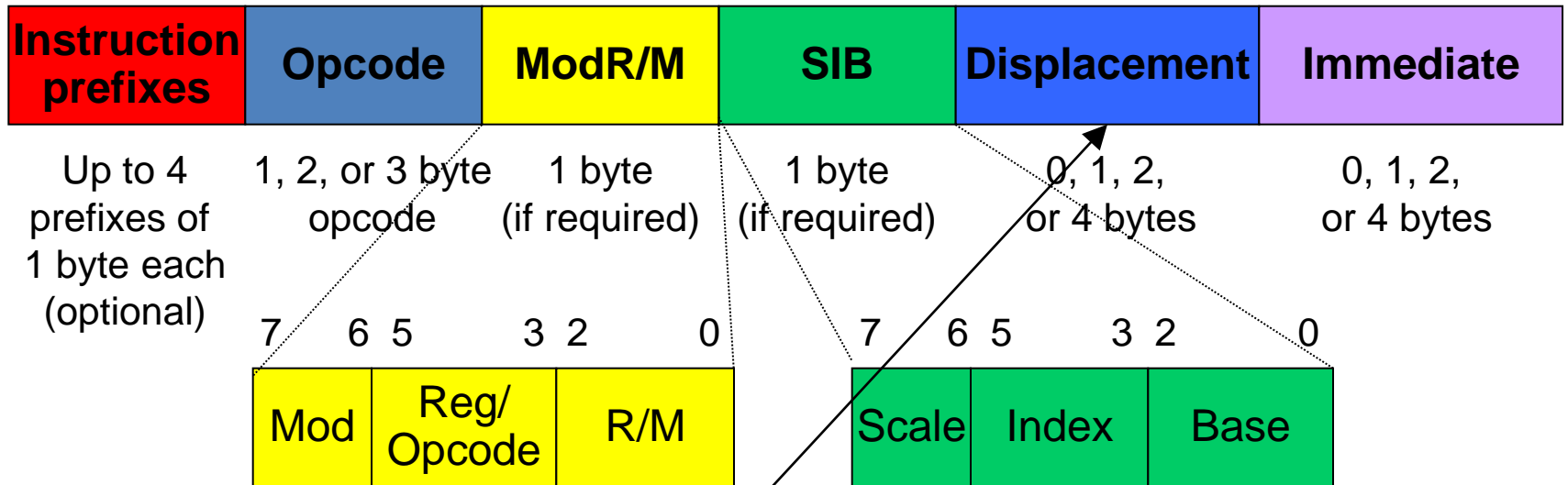
IA-32 Instruction Format (cont.)



SIB

- Used when one of the operands is a memory operand that uses a **scale**, an **index** register, and/or a **base** register

IA-32 Instruction Format (cont.)

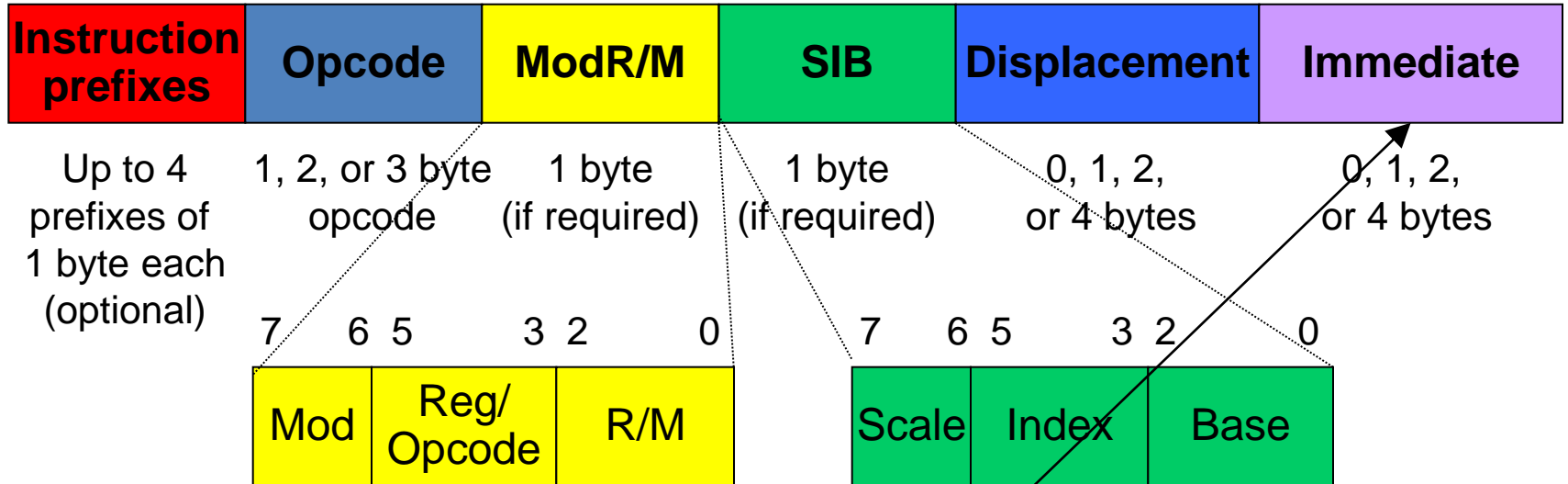


Displacement

- Used in jump and call instructions
- Indicates the displacement between the destination instruction and the jump/call instruction
- More precisely, indicates:

$$[\text{addr of destination instr}] - [\text{addr of instr following the jump/call}]$$
- Uses little-endian byte order

IA-32 Instruction Format (cont.)



Immediate

- Specifies an immediate operand
- Uses little-endian byte order

Example: Push on to Stack

- Assembly language:
`pushl %edx`
- Machine code:
 - IA32 has a separate opcode for push for each register operand
 - 50: `pushl %eax`
 - 51: `pushl %ecx`
 - 52: `pushl %edx` → 0101 0010
 - ...
 - Results in a *one-byte* instruction
- Observe: sometimes one assembly language instruction can map to a *group* of different opcodes

Example: Load Effective Address

- Assembly language:

```
leal (%eax,%eax,4), %eax
```

- Machine code:

– Byte 1: 8D (opcode for "load effective address")

1000 1101

– Byte 2: 04 (dest %eax, with scale-index-base)

0000 0100

– Byte 3: 80 (scale=4, index=%eax, base=%eax)

1000 0000

Load the address $\%eax + 4 * \%eax$ into register %eax

CISC and RISC

- IA-32 machine language instructions are **complex**
- IA-32 is a
 - **Complex Instruction Set Computer (CISC)**
- Alternative:
 - **Reduced Instruction Set Computer (RISC)**

Characteristics of CISC and RISC

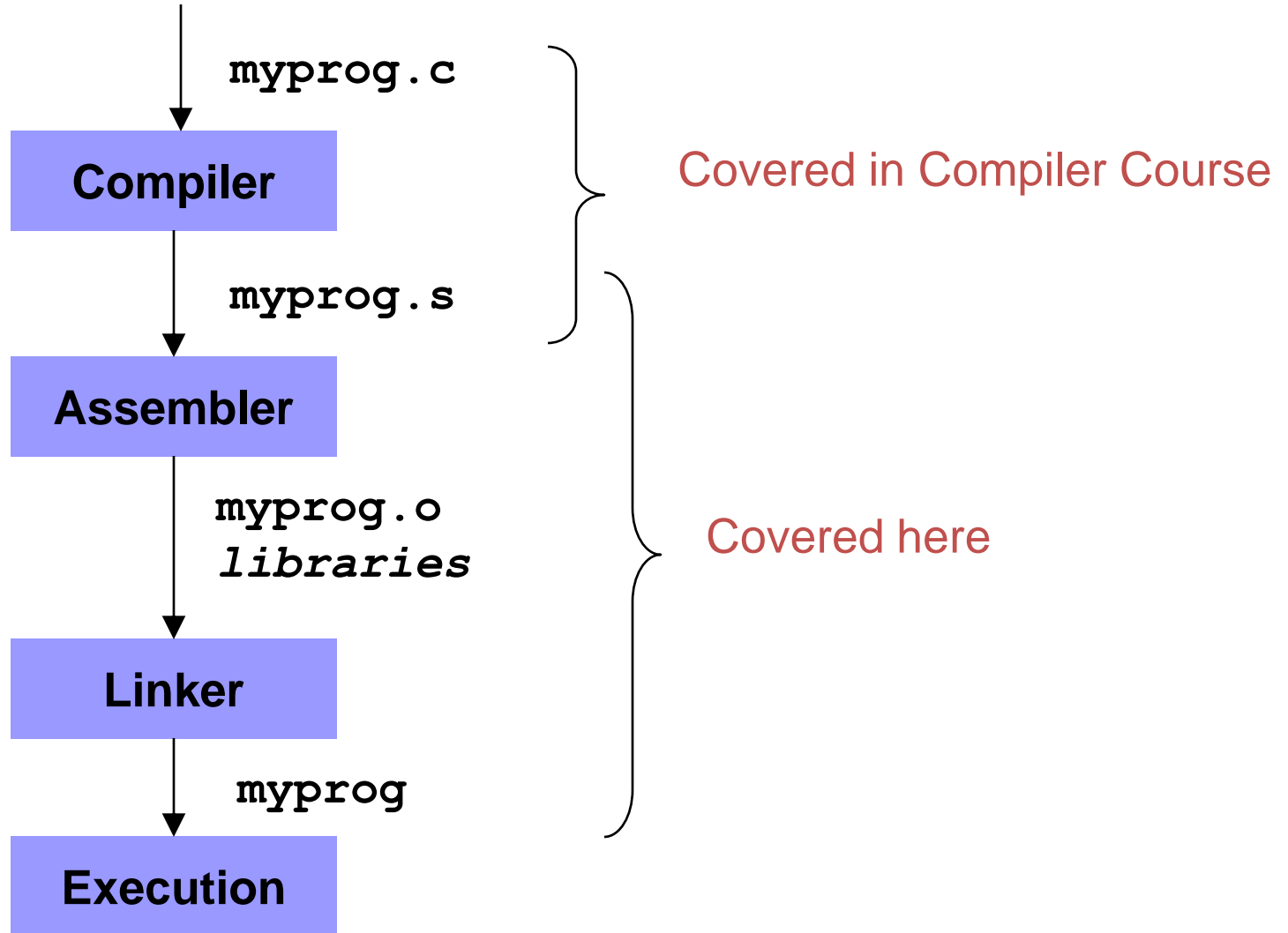
- CISC
 - **Many** instructions
 - **Many** addressing modes (direct, indirect, indexed, base-pointer)
 - Hardware interpretation is **complex**
 - **Few** instructions required to accomplish a given job (expressive)
 - Example: IA-32
- RISC
 - **Few** instructions
 - **Few** addressing modes (typically only direct and indirect)
 - Hardware interpretation is **simple**
 - **Many** instructions required to accomplish a given job (not expressive)
 - Examples: MIPS, SPARC

Brief History of CISC and RISC

- Stage 1: Programmers write assembly language
 - Important that assembly/machine language be expressive
 - CISC dominates (esp. Intel)
- Stage 2: Programmers write high-level language
 - Not important that assembly/machine language be expressive; the compiler generates it
 - Important that compilers work well => assembly/machine language should be simple
 - RISC takes a foothold (but CISC, esp. Intel, persists)
- Stage 3: Compilers get smarter
 - Less important that assembly/machine language be simple
 - Much motivation for RISC disappears
 - CISC (esp. Intel) dominates the computing world

Part 2: The Assembly Process

The Build/Execute Process



Two Aspects of the Assembler/Linker

- Translating each instruction
 - Mapping an assembly-language instruction
 - ... into the corresponding machine-language instruction
- Dealing with references across instructions
 - Jumps to other locations in same chunk of code
 - Accesses a global variable by the name of its memory location
 - Calling to and returning from functions defined in other code

```
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    $'A', %eax
    jne     skip
    pushl    $msg
    call    printf
    addl    $4, %esp

skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

References Across Instructions

- Many instructions can be assembled independently
 - `pushl %edx`
 - `leal (%eax, %eax, 4), %eax`
 - `movl $0, %eax`
 - `addl %ebx, %ecx`
- But, some make references to other data or code
 - `jne skip`
 - `pushl $msg`
 - `call printf`
- Need to fill in those references
 - To generate a final executable binary

The Forward Reference Problem

- Problem

```
...  
    jmp mylabel  
...  
mylabel:  
...
```

Any assembler must
deal with the
forward reference
problem

- Assembler must generate machine language code for “jmp mylabel”
- But assembler hasn't yet *seen* the definition of mylabel
 - I.e., the jmp instruction contains a **forward reference** to mylabel

The Forward Reference Solution

- Solution
 - Assembler performs **2 passes** over assembly language program
- Different assemblers perform different tasks in each pass
- One straightforward design...

Assembler Passes

- Pass1
 - Assembler traverses assembly program to create...
 - Symbol table
 - Key: label
 - Value: information about label
 - Label name, which section, what offset within that section, ...
- Pass 2
 - Assembler traverses assembly program again to create...
 - RODATA section
 - DATA section
 - BSS section
 - TEXT section
 - Relocation record section
 - Each relocation record indicates an area that the linker must patch

An Example Program

- A simple (nonsensical) program:

```
#include <stdio.h>
int main(void) {
    if (getchar() == 'A')
        printf("Hi\n");
    return 0;
}
```

- Let's consider how the assembler handles that program...

```
        .section ".rodata"
msg:
        .asciz  "Hi\n"
        .section ".text"
        .globl  main

main:
        pushl   %ebp
        movl    %esp, %ebp
        call   getchar
        cmpl   $'A', %eax
        jne    skip
        pushl   $msg
        call   printf
        addl   $4, %esp

skip:
        movl   $0, %eax
        movl   %ebp, %esp
        popl   %ebp
        ret
```


Assembler Data Structures (1)

- Symbol Table

Label	Section	Offset	Local?	Seq#

- Relocation Records

Section	Offset	Rel Type	Seq#

- RODATA Section (location counter: 0)

Offset	Contents	Explanation

- No DATA or BSS section in this program
- Initially all sections are empty

- TEXT Section (location counter: 0)

Offset	Contents	Explanation

Assembler Pass 1

```
msg:
    .section ".rodata"
    .asciz "Hi\n"
    .section ".text"
    .globl main
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne     skip
    pushl   $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler notes that
the current section is
RODATA

Assembler adds binding
to Symbol Table...

Assembler Data Structures (2)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0

- msg marks a spot in the RODATA section at offset 0
- msg is a local label
- Assign msg sequence number 0

- Relocation Records

- (Same)

- RODATA Section (location counter: 0)

- (Same)

- TEXT Section (location counter: 0)

- (Same)

Assembler Pass 1 (cont.)

```
.section ".rodata"
msg:
.asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl   '$A', %eax
    jne    skip
    pushl   $msg
    call   printf
    addl   $4, %esp
skip:
    movl   $0, %eax
    movl   %ebp, %esp
    popl   %ebp
    ret
```

Assembler increments
RODATA section
location counter by
byte count of the
string (4)...

Assembler Data Structures (3)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 0)

- (Same)

- RODATA location counter now is 4
- If another label were defined in at this point, it would mark a spot in RODATA at offset 4

Assembler Pass 1 (cont.)

```
msg:      .section ".rodata"
          .asciz "Hi\n"
          .section ".text"
          .globl main
main:
    pushl   %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne     skip
    pushl   $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler notes
that current section
is TEXT

Assembler does
nothing

Assembler adds binding
to Symbol Table...

Assembler Data Structures (4)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1

- main marks a spot in the TEXT section at offset 0
- main is a local label (assembler will discover otherwise in Pass 2)
- Assign main sequence number 1

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 0)

- (Same)

Assembler Pass 1 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl   %ebp
    movl   %esp, %ebp
    call   getchar
    cmpl   '$A', %eax
    jne    skip
    pushl   $msg
    call   printf
    addl   $4, %esp
skip:
    movl   $0, %eax
    movl   %ebp, %esp
    popl   %ebp
    ret
```

Assembler increments
TEXT section location
counter by the length
of each instruction...

Assembler Data Structures (5)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 26)

- (Same)

- TEXT location counter now is 26
- If another label were defined at this point, it would mark a spot in TEXT at offset 26

Assembler Pass 1 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne     skip
    pushl    $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler adds binding
to Symbol Table...

Assembler Data Structures (6)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- skip marks a spot in the TEXT section at offset 26
- skip is a local label
- Assign skip sequence number 2

- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 26)
 - (Same)

Assembler Pass 1 (cont.)

```
.section ".rodata"
msg:
.asciz "Hi\n"
.section ".text"
.globl main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl    '$A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler increments
TEXT section location
counter by the length
of each instruction...

Assembler Data Structures (7)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 35)

- (Same)

- TEXT location counter now is 35
- If another label were defined at this point, it would mark a spot in TEXT at offset 35

From Assembler Pass 1 to Pass 2

- End of Pass 1
 - Assembler has (partially) created Symbol Table
 - So assembler now knows which location each label marks
- Beginning of Pass 2
 - Assembler resets all section location counters...

Assembler Data Structures (8)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	local	1
skip	TEXT	26	local	2

- Relocation Records

- (Same)

- RODATA Section (location counter: 0)

- (Same)

- TEXT Section (location counter: 0)

- (Same)

• Location counters reset to 0

Assembler Pass 2

```
msg:
    .section ".rodata"
    .asciz "Hi\n"
main:
    .section ".text"
    .globl main
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne    skip
    pushl    $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler notes that the current section is RODATA

Assembler does nothing

Assembler places bytes in RODATA section, and increments location counter...

Assembler Data Structures (9)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)

• Location counter incremented to 4

Offset	Contents (hex)	Explanation
0	48	ASCII code for 'H'
1	69	ASCII code for 'i'
2	0A	ASCII code for '\n'
3	00	ASCII code for null char

- TEXT Section (location counter: 0)
 - (Same)

• RODATA section contains the bytes comprising the string

Assembler Pass 2 (cont.)

```
        .section ".rodata"
msg:
        .asciz "Hi\n"
        .section ".text"
        .globl main
main:
        pushl   %ebp
        movl    %esp, %ebp
        call   getchar
        cmpl   '$A', %eax
        jne    skip
        pushl   $msg
        call   printf
        addl   $4, %esp
skip:
        movl   $0, %eax
        movl   %ebp, %esp
        popl   %ebp
        ret
```

Assembler notes that
the current section is
TEXT

Assembler updates
Symbol Table...

Assembler Data Structures (10)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2

- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 0)
 - (Same)



• main is a global label

Assembler Pass 2 (cont.)

```
msg:      .section ".rodata"
          .asciz  "Hi\n"
          .section ".text"
          .globl main

main:     pushl   %ebp
          movl   %esp, %ebp
          call  getchar
          cmpl  '$A', %eax
          jne   skip
          pushl $msg
          call  printf
          addl  $4, %esp

skip:    movl   $0, %eax
          movl  %ebp, %esp
          popl  %ebp
          ret
```

Assembler does nothing

Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (11)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 1)

Offset	Contents	Explanation
0	55	<code>pushl %ebp</code> 01010101 This is a "pushl %ebp" instruction

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl     %esp, %ebp
    call     getchar
    cmpl     '$A', %eax
    jne     skip
    pushl    $msg
    call     printf
    addl     $4, %esp
skip:
    movl     $0, %eax
    movl     %ebp, %esp
    popl     %ebp
    ret
```

Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (12)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 3)

Offset	Contents	Explanation
...
1-2	89 E5	<code>movl %esp,%ebp</code> <code>10001001 11 100 101</code> This is a "movl" instruction whose source operand is a register The M field designates a register The source register is ESP The destination register is EBP

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl     %esp, %ebp
    call     getchar
    cmpl     '$A', %eax
    jne     skip
    pushl    $msg
    call     printf
    addl     $4, %esp
skip:
    movl     $0, %eax
    movl     %ebp, %esp
    popl     %ebp
    ret
```

Assembler generates
machine language
code in current
(TEXT) section...

Assembler Data Structures (12)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 8)

- Assembler looks in Symbol Table to find offset of getchar
- getchar is not in Symbol Table
- Assembler cannot compute displacement that belongs at offset 4
- So...

Offset	Contents	Explanation
...
3-7	E8 ????????	call getchar 11101000 ?? This is a "call" instruction with a 4-byte immediate operand This the displacement

Assembler Data Structures (13)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2
getchar	?	?	global	3

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 8)

- (Same)

- Assembler adds getchar to Symbol Table

- Then...

Assembler Data Structures (14)

- Symbol Table
 - (Same)
- Relocation Records

- Assembler generates a relocation record, thus asking linker to patch code

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3

- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 8)
 - (Same)

***Dear Linker,
Please patch the TEXT section at offset 4. Do a “displacement” type of patch. The patch is with respect to the label whose seq number is 3 (i.e. getchar).
Sincerely,
Assembler***

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
.asciz "Hi\n"
.section ".text"
.globl main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl   '$A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler generates
machine language
code in current
(TEXT) section...

Assembler Data Structures (15)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 11)

Offset	Contents	Explanation
...
8-10	83 F8 41	<pre>cmpl %'A',%eax 10000011 11 111 000 01000001 This is some "l" instruction that has a 1 byte immediate operand The M field designates a register This is a "cmp" instruction The destination register is EAX The immediate operand is 'A'</pre>

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne     skip
    pushl    $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (16)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 13)

- Assembler looks in Symbol Table to find offset of skip (26)
- Assembler subtracts offset of next instruction (13)
- Resulting displacement is 13

Offset	Contents	Explanation
...
11-12	75 0D	<code>jne skip</code> <code>01110101 00001101</code> This is a <code>jne</code> instruction that has a 1 byte immediate operand The displacement between the destination instr. and the next instr. is 13

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
.asciz "Hi\n"
.section ".text"
.globl main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl    '$A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (16)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 18)

- Assembler knows offset of msg (0) within RODATA section
- But assembler does not know location RODATA section
- So assembler does not know location of msg
- So...

Offset	Contents	Explanation
...
13-17	68 ????????	pushl \$msg 001101000 ?? This is a pushl instruction with a 4 byte immediate operand This is the data to be pushed

Assembler Data Structures (17)

- Symbol Table
 - (Same)
- Relocation Records

Section	Offset	Rel Type	Seq#
...
TEXT	14	absolute	0

- Assembler generates a relocation record, thus asking linker to patch code

- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 18)
 - (Same)

***Dear Linker,
Please patch the TEXT section at offset 14. Do an “absolute” type of patch. The patch is with respect to the label whose seq number is 0 (i.e. msg).***

***Sincerely,
Assembler***

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
    .asciz "Hi\n"
.section ".text"
.globl main
main:
    pushl    %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    '$A', %eax
    jne     skip
    pushl    $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler generates
machine language
code in current
(TEXT) section...

Assembler Data Structures (18)

- Symbol Table
 - (Same)
- Relocation Records
 - (Same)
- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 23)

- Assembler looks in Symbol Table to find offset of printf
- printf is not in Symbol Table
- Assembler cannot compute displacement that belongs at offset 19
- So...

Offset	Contents	Explanation
...
18-22	E8 ????????	call printf 11101000 ?? This is a "call" instruction with a 4-byte immediate operand This the displacement

Assembler Data Structures (19)

- Symbol Table

Label	Section	Offset	Local?	Seq#
msg	RODATA	0	local	0
main	TEXT	0	global	1
skip	TEXT	26	local	2
getchar	?	?	global	3
printf	?	?	global	4

- Relocation Records

- (Same)

- RODATA Section (location counter: 4)

- (Same)

- TEXT Section (location counter: 23)

- (Same)

- Assembler adds printf to Symbol Table
- Then...

Assembler Data Structures (20)

- Symbol Table
 - (Same)
- Relocation Records

- Assembler generates a relocation record, thus asking linker to patch code

Section	Offset	Rel Type	Seq#
...
TEXT	19	displacement	4

- RODATA Section (location counter: 4)
 - (Same)
- TEXT Section (location counter: 8)
 - (Same)

***Dear Linker,
Please patch the TEXT section at offset 19. Do a “displacement” type of patch. The patch is with respect to the label whose seq number is 4 (i.e. printf).
Sincerely,
Assembler***

Assembler Pass 2 (cont.)

```
.section ".rodata"
msg:
.asciz "Hi\n"
.section ".text"
.globl main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl    '$A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler ignores

Assembler generates
machine language
code in current
(TEXT) section...

skip:

Assembler Data Structures (21)

- Symbol Table, Relocation Records, RODATA Section
 - (Same)
- TEXT Section (location counter: 31)

Offset	Contents	Explanation
...
23-25	83 C4 04	<pre>addl \$4,%esp 10000011 11 000 100 00000100</pre> <p>This is some "l" instruction that has a 1 byte immediate operand</p> <p style="padding-left: 40px;">The M field designates a register</p> <p style="padding-left: 40px;">This is an "add" instruction</p> <p style="padding-left: 40px;">The destination register is ESP</p> <p style="padding-left: 40px;">The immediate operand is 4</p>
26-30	B8 00000000	<pre>movl \$0,%eax 10111000 00000000000000000000000000000000</pre> <p>This is an instruction of the form "movl 4-byte-immediate, %eax"</p> <p style="padding-left: 40px;">The immediate operand is 0</p>

Assembler Data Structures (22)

- Symbol Table, Relocation Records, RODATA Section
 - (Same)
- TEXT Section (location counter: 35)

Offset	Contents	Explanation
...
31-32	89 EC	<code>movl %ebp,%esp</code> 10001001 11 101 100 This is a "movl" instruction whose source operand is a register The M field designates a register The source register is EBP The destination register is ESP
33	5D	<code>popl %ebp</code> 01011101 This is a "popl %ebp" instruction
34	C3	<code>ret</code> 11000011 This is a "ret" instruction

From Assembler to Linker

- Assembler writes its data structures to .o file
- Linker:
 - Reads .o file
 - Works in two phases: **resolution** and **relocation**

Linker Resolution

- Resolution
 - Linker resolves references
- For this program, linker:
 - Notes that Symbol Table contains undefined labels
 - getchar and printf
 - Fetches, from libc.a, machine language code defining getchar and printf
 - Adds that code to TEXT section
 - (May add code to other sections too)
 - Updates Symbol Table to note offsets of getchar and printf
 - Adds column to Symbol Table to note addresses of all labels

Linker Relocation

- Relocation
 - Linker patches (“relocates”) code
 - Linker traverses relocation records, patching code as specified
- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

- Linker looks up offset of getchar
- Linker computes:
[offset of getchar] – 8
- Linker places difference in TEXT section at offset 4

Linker Relocation (cont.)

- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

- Linker looks up addr of msg
- Linker places addr in TEXT section at offset 14

Linker Relocation (cont.)

- For this program

Section	Offset	Rel Type	Seq#
TEXT	4	displacement	3
TEXT	14	absolute	0
TEXT	19	displacement	4

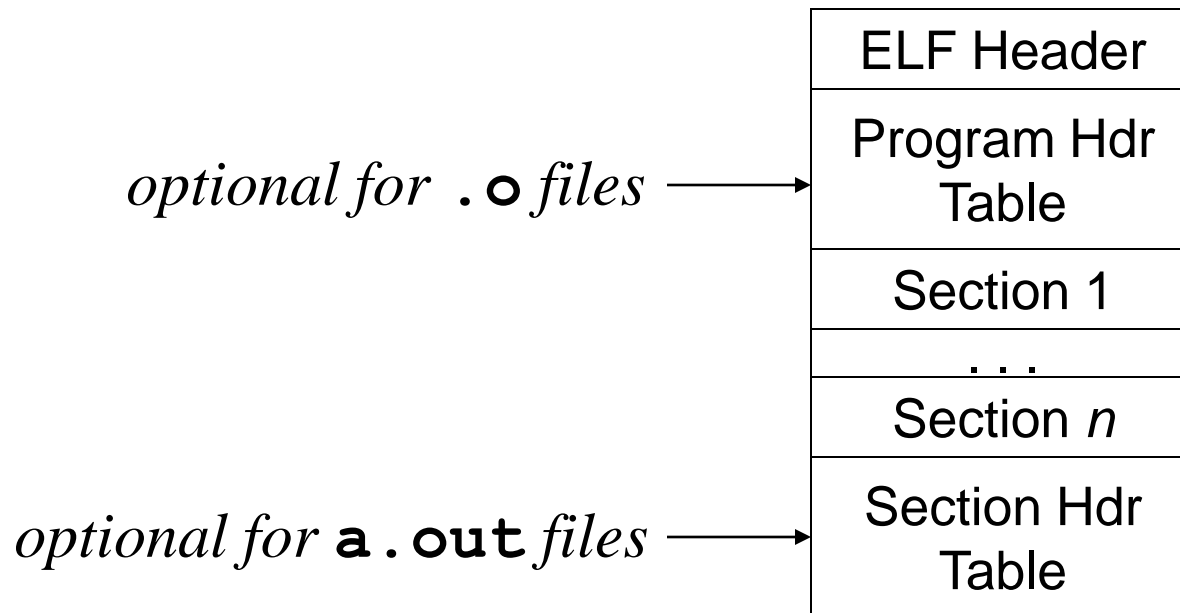
- Linker looks up offset of printf
- Linker computes:
[offset of printf] – 23
- Linker places difference in TEXT section at offset 19

Linker Finishes

- Linker writes resulting TEXT, RODATA, DATA, BSS sections to executable binary file

ELF: Executable and Linking Format

- Unix format of object and executable files
 - Output by the assembler
 - Input and output of linker



Conclusions

- **Assembler:** reads assembly language file
 - **Pass 1:** Generates Symbol Table
 - Contains info about labels
 - **Pass 2:** Uses Symbol Table to generate code
 - TEXT, RODATA, DATA, BSS sections
 - Relocation Records
 - Writes object file (ELF)
- **Linker:** reads object files
 - **Resolution:** Resolves references to make Symbol Table complete
 - **Relocation:** Uses Symbol Table and Relocation Records to patch code
 - Writes executable binary file (ELF)

Appendix: Generating Machine Lang

- Given an assembly language instruction, how can you find the machine language equivalent?
- Option 1: Consult IA-32 reference manuals
 - See course Web pages for links to the manuals

Appendix: Generating Machine Lang

- Option 2:
 - Compose an assembly language program that contains the given assembly language instruction
 - Then use gdb...

Appendix: Generating Machine Lang

```
$ gcc209 detecta.s -o detecta
$ gdb detecta
(gdb) x/12i main
0x80483b4 <main>:      push   %ebp
0x80483b5 <main+1>:    mov    %esp,%ebp
0x80483b7 <main+3>:    call  0x8048298 <getchar@plt>
0x80483bc <main+8>:    cmp    $0x41,%eax
0x80483bf <main+11>:   jne   0x80483ce <skip>
0x80483c1 <main+13>:   push  $0x80484b0
0x80483c6 <main+18>:   call  0x80482c8 <printf@plt>
0x80483cb <main+23>:   add   $0x4,%esp
0x80483ce <skip>:      mov    $0x0,%eax
0x80483d3 <skip+5>:    mov   %ebp,%esp
0x80483d5 <skip+7>:    pop   %ebp
0x80483d6 <skip+8>:    ret
(gdb) x/35b main
0x0 <main>:      0x55  0x89  0xe5  0xe8  0xfc  0xff  0xff  0xff  0xff
0x8 <main+8>:    0x83  0xf8  0x41  0x75  0x0d  0x68  0x00  0x00  0x00
0x10 <main+16>: 0x00  0x00  0xe8  0xfc  0xff  0xff  0xff  0xff  0x83
0x18 <main+24>: 0xc4  0x04  0xb8  0x00  0x00  0x00  0x00  0x00  0x89
0x20 <skip+6>:  0xec  0x5d  0xc3
(gdb) quit
```

Build program; run gdb from shell

Issue x/i command to examine memory as instructions

Issue x/b command to examine memory as raw bytes

Match instructions to bytes

Appendix: Generating Machine Lang

- Option 3:
 - Compose an assembly language program that contains the given assembly language instruction
 - Then use objdump - a special purpose tool...

Appendix: Generating Machine Lang

Build program; run objdump

```
$ gcc209 detecta.s -o detecta
$ objdump -d detecta
detecta:      file format elf32-i386
```

Machine language

Assembly language

```
...
Disassembly of section .text:
...
```

```
080483b4 <main>:
```

```
80483b4:  55
80483b5:  89 e5
80483b7:  e8 dc fe ff ff
80483bc:  83 f8 41
80483bf:  75 0d
80483c1:  68 b0 84 04 08
80483c6:  e8 fd fe ff ff
80483cb:  83 c4 04
```

```
push  %ebp
mov   %esp,%ebp
call  8048298 <getchar@plt>
cmp   $0x41,%eax
jne   80483ce <skip>
push  $0x80484b0
call  80482c8 <printf@plt>
add   $0x4,%esp
```

```
080483ce <skip>:
```

```
80483ce:  b8 00 00 00 00
80483d3:  89 ec
80483d5:  5d
80483d6:  c3
```

```
mov   $0x0,%eax
mov   %ebp,%esp
pop   %ebp
ret
```

```
...
```