Machine Language, 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

• Last stop on the “language levels” tour
• A power programmer knows the relationship between assembly and machine languages
• A systems programmer knows how an assembler translates assembly 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 format in following slides

- We’ll go over
  - The format of instructions
  - Two example instructions

- Just to convey a sense of how it works…

IA-32 Instruction Format

<table>
<thead>
<tr>
<th>Instruction prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 4 prefixes of 1 byte each (optional)</td>
<td>1, 2, or 3 byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>0, 1, 2, or 4 bytes</td>
<td></td>
</tr>
</tbody>
</table>

ModR/M

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

Mod

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

<table>
<thead>
<tr>
<th>Instruction prefixes</th>
<th>Opcode</th>
<th>Mod/R/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod</td>
<td>Reg/</td>
<td>R/M</td>
<td>7</td>
<td>65</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

#### 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

### Example: Push on to Stack

- **Assembly language:**
  
  ```assembly
  pushl %edx
  ```

- **Machine code:**
  
  ```
  0101 0010
  ```

- **IA32 has a separate opcode for push for each register operand**
  - 50: pushl %eax
  - 51: pushl %ecx
  - 52: pushl %edx
  - ...

- Results in a one-byte instruction

- Sometimes one assembly language instruction can map to a group of different opcodes
Example: Load Effective Address

- **Assembly language:**
  \[
  \text{leal (eax,eax,4), eax}
  \]
- **Machine code:**
  - Byte 1: 8D (opcode for "load effective address")
  - Byte 2: 04 (dest %eax, with scale-index-base)
  - Byte 3: 80 (scale=4, index=%eax, base=%eax)

<table>
<thead>
<tr>
<th>8D</th>
<th>04</th>
<th>80</th>
</tr>
</thead>
</table>

Load the address \( %\text{eax} + 4 \times %\text{eax} \) 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
  - Hardware is plentiful, enabling complex implementations
  - Much motivation for RISC disappears
  - CISC (esp. Intel) dominates the computing world
Part 2: The Assembly Process

The Build/Execute Process

- Compiler
  - myprog.c
  - myprog.s
  - myprog.o
  - libraries
  - Myprog executable

Covered in COS 320: Compiling Techniques

Covered here

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 (file) of code
  - Accesses a global variable by the name of its memory location
  - Calling to and returning from functions defined in other code

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
  - 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

Any assembler must deal with the forward reference problem

Assembler Passes

- Pass 1
  - 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...
    - RDATA section
    - DATA section
    - BSS section
    - TEXT section

The Forward Reference Solution

- Solution
  - Assembler performs 2 passes over assembly language program
  - One to record labels and addresses, e.g.
  - Another to patch them in

- Different assemblers perform different tasks in each pass

- One straightforward design...

References Outside the File

- e.g. Call printf
  - printf is defined in another library; assembler doesn’t know at what address that code will be placed in memory
  - Defer to linker
    - Pass 1 defers symbol resolution to pass 2, pass 2 defers some to linker
  - So, Pass 2 becomes...
    - Assembler traverses assembly program again to create...
      - RDATA section
      - DATA section
      - BSS section
      - TEXT section
      - Relocation record section
        - Each relocation record indicates something that linker must patch
An Example Program

- A simple (nonsensical) program:

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

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

Assembler Data Structures (1)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq #</th>
</tr>
</thead>
</table>

- Relocation Records

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq #</th>
</tr>
</thead>
</table>

- RODATA Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

- TEXT Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

Assembler Pass 1

- Assembler notes that the current section is RODATA

- Assembler adds binding to Symbol Table...

Assembler Data Structures (2)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq #</th>
</tr>
</thead>
</table>

- Relocation Records

- RODATA Section (location counter: 0)

- TEXT Section (location counter: 0)
Assembler Pass 1 (cont.)

msg: .section ".rodata"
    .ascii  "Hi"
    .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 (3)

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler Data Structures (4)

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembler notes that current section is TEXT
Assembler adds binding to Symbol Table...
Assembler does nothing
Assembler increments RODATA section location counter by byte count of the string (4)...
RODATA location counter now is 4
If another label were defined at this point, it would mark a spot in RODATA at offset 4
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
### Assembler Pass 1 (cont.)

```
.msg: .section ".rodata"
    .asciz  "Hi"
.rodata
    .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)

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
</tbody>
</table>

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

### Assembler Pass 1 (cont.)

```
Assembler adds binding to Symbol Table...
```

### Assembler Data Structures (6)

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
<tr>
<td>skip</td>
<td>TEXT</td>
<td>26</td>
<td>local</td>
<td>2</td>
</tr>
</tbody>
</table>

- Symbol Table
- 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
- skip marks a spot in the TEXT section at offset 26
- skip is a local label
- Assign skip sequence number 2
### Assembler Pass 1 (cont.)

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

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
<tr>
<td>skip</td>
<td>TEXT</td>
<td>26</td>
<td>local</td>
<td>2</td>
</tr>
</tbody>
</table>

**Symbol Table**
- (Same)

**Relocation Records**
- (Same)

**RODATA Section** (location counter: 4)
- (Same)

**TEXT Section** (location counter: 35)
- (Same)

**Location counters**
- 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)

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
<tr>
<td>skip</td>
<td>TEXT</td>
<td>26</td>
<td>local</td>
<td>2</td>
</tr>
</tbody>
</table>

**Symbol Table**
- (Same)

**Relocation Records**
- (Same)

**RODATA Section** (location counter: 0)
- (Same)

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

- Location counters reset to 0
Assembler notes that the current section is RODATA

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

• Symbol Table
  • (Same)
• Relocation Records
  • (Same)
• RODATA Section (location counter: 4)
  • Location counter incremented to 4
  • RODATA section contains the bytes comprising the string

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

• Symbol Table
  • (Same)
• Relocation Records
  • (Same)
• RODATA Section (location counter: 4)
  • (Same)
• TEXT Section (location counter: 0)
  • (Same)
Assembler Pass 2 (cont.)

Assembler does nothing

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

Assembler Pass 2 (cont.)

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: 3)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>00 05</td>
<td>movl $esp, %ebp</td>
</tr>
<tr>
<td></td>
<td>10001001 11 101 101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This is a &quot;movl $esp, %ebp&quot; instruction</td>
<td></td>
</tr>
</tbody>
</table>

Assembler Data Structures (12)

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td>00 05</td>
<td>movl $esp, %ebp</td>
</tr>
<tr>
<td></td>
<td>10001001 11 101 101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This is a &quot;movl $esp, %ebp&quot; instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The W field designates a register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The source register is ESP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The destination register is EBP</td>
<td></td>
</tr>
</tbody>
</table>
Assembler Pass 2 (cont.)

```assembly
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 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)
- (Same)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| 5-7    | E8 ???????? | call getchar
|        | 11101000 ??????????????????????????????? | This is a call instruction with a 4-byte immediate operand
|        |          | This is the displacement |

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...

Assembler Data Structures (13)

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

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>global</td>
<td>1</td>
</tr>
<tr>
<td>skip</td>
<td>TEXT</td>
<td>26</td>
<td>local</td>
<td>2</td>
</tr>
<tr>
<td>getchar</td>
<td>TEXT</td>
<td>7</td>
<td>global</td>
<td>3</td>
</tr>
</tbody>
</table>

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

Assembler Data Structures (14)

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-10</td>
<td>01 0D 75</td>
<td>jne skip</td>
</tr>
</tbody>
</table>

This is a jne 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'

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-12</td>
<td>01 0D 75</td>
<td>jne skip</td>
</tr>
</tbody>
</table>

This is a jne instruction that has a 1 byte immediate operand
The displacement between the destination instr. and the next instr. is 13
Assembler Pass 2 (cont.)

```
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)  
  • (Same)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-17</td>
<td>00110100</td>
<td>This is a pushl instruction with a 4 byte immediate operand</td>
</tr>
</tbody>
</table>

• 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...

Assembler Pass 2 (cont.)

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

Assembler Data Structures (17)

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
</tbody>
</table>

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 Data Structures (18)

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>18-22</td>
<td>E8 ???????</td>
<td>call printf</td>
</tr>
<tr>
<td></td>
<td>Intel 32-bit 4-byte immediate operand</td>
<td></td>
</tr>
</tbody>
</table>

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...

Assembler Data Structures (19)

- Symbol Table
- (Same)
- Relocation Records
- (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 23)
  - (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 generates a relocation record, thus asking linker to patch code

Assembler ignores

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

Assembler Pass 2 (cont.)

```
.section .rodata
msg: .asciz "Hi"

.section .text
.globl main
main:
pushl temp
call getchar
cmp $'A', %eax
jne skip
pushl $msg
call printf
addl $4, %esp
skip:
movl $0, %eax
popl %ebp
ret
```
### Assembler Data Structures (21)

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-25</td>
<td>addl $4,%esp</td>
<td>This is a “addl” instruction whose source operand is a register. The destination register is ESP. The immediate operand is 4.</td>
</tr>
<tr>
<td>26-30</td>
<td>movl %ebp,%esp</td>
<td>This is a “movl” instruction whose source operand is a register. The destination register is ESP. The immediate operand is 0.</td>
</tr>
</tbody>
</table>

### Assembler Data Structures (22)

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>25-32</td>
<td>movl $0,%eax</td>
<td>This is an instruction of the form “movl 4-byte-immediate, %eax” in which the immediate operand is 0.</td>
</tr>
<tr>
<td>33</td>
<td>popl %ebp</td>
<td>This is a “popl %ebp” instruction.</td>
</tr>
<tr>
<td>34</td>
<td>ret</td>
<td>This is a “ret” instruction.</td>
</tr>
</tbody>
</table>

### 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
  - 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

  - 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

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

Linker Relocation (cont.)

- For this program

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

  - 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

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF Header</td>
<td>Optional for .o files</td>
</tr>
<tr>
<td>Program Hdr</td>
<td>Optional for a.out files</td>
</tr>
<tr>
<td>Section 1</td>
<td></td>
</tr>
<tr>
<td>Section n</td>
<td></td>
</tr>
<tr>
<td>Section Hdr</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td></td>
</tr>
</tbody>
</table>

**Conclusions**

- **Assembler**: reads assembly language file
  - **Pass 1**: Generates Symbol Table
  - **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**

- Hint for Buffer Overrun assignment…

- 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
  - Option 2:
    - Compose an assembly language program that contains the given assembly language instruction
    - Then use gdb…
Appendix: Generating Machine Lang

**Using gdb**

$ gcc -o detecta detecta.s
$ 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
0x8 <main+8>: 0x83 f8 0x41 0x75 0x0d 0x68 0x00 0x00
0x10 <main+16>: 0x00 0x00 e8 fc ff ff ff
0x18 <main+24>: c4 0x04 b8 0x00 0x00 0x00 0x00 0x00
0x20 <skip+6>: ec 5d c3
(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

---

**Using objdump**

$ gcc -o detecta detecta.s
$ objdump -d detecta

detecta:     file format elf32-i386
... Disassembly of section .text: ...
080483b4 <main>: push %ebp
080483b5 <main+1>: mov %esp,%ebp
080483b7 <main+3>: call 0x8048298 <getchar@plt>
080483bc <main+8>: cmp $0x41,%eax
080483bf <main+11>: jne 0x80483ce <skip>
080483c1 <main+13>: push $0x80484b0
080483c6 <main+18>: call 0x80482c8 <printf@plt>
080483cb <main+23>: add $0x4,%esp
080483ce <skip>: mov $0x0,%eax
080483d3 <skip+5>: mov %ebp,%esp
080483d5 <skip+7>: pop %ebp
080483d6 <skip+8>: ret
...

Build program; run objdump

Machine language

Assembly language

---

**Option 3:**

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