Machine Language, Assemblers, and Linkers
Goals of this Lecture

Help you to learn about:
  • IA-32 machine language (in general)
  • The assembly and linking processes

Why?
  • 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 language code to machine language code
Agenda

Machine Language

The Assembly Process

The Linking Process
IA-32 machine language

- Difficult to generalize about IA-32 instruction format
  - Many (most!) instructions are exceptions to the rules
  - Many instructions use this format...
## 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></td>
<td>1, 2, or 3 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
</tr>
</tbody>
</table>

### Instruction prefixes
- Up to 4 prefixes of 1 byte each (optional)

### Opcode

### ModR/M

### SIB

### Displacement

### Immediate

### Instruction prefix
- Sometimes a repeat count
- Rarely used; don’t be concerned
IA-32 Instruction Format (cont.)

Instruction prefixes | Opcode | ModR/M | SIB | Displacement | Immediate
---|---|---|---|---|---
Up to 4 prefixes of 1 byte each (optional) | 1, 2, or 3 bytes | 1 byte (if required) | 1 byte (if required) | 1, 2, or 4 bytes (if required) | 1, 2, or 4 bytes (if required)

**Mod** | **Reg/Opcode** | **R/M** | **Scale** | **Index** | **Base**
7 | 6 | 5 | 3 | 2 | 0

**Opcode**
- Specifies which operation should be performed
  - Add, move, call, etc.
- Sometimes specifies additional (or less) information
### IA-32 Instruction Format (cont.)

<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 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
</tr>
</tbody>
</table>

#### ModR/M (register mode, register/opcode, register/memory)
- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes specifies register(s):
  - 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
### IA-32 Instruction Format (cont.)

<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 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
</tr>
<tr>
<td>Mod</td>
<td>Reg/Opcode</td>
<td>R/M</td>
<td>Scale</td>
<td>Index</td>
<td>Base</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<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 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
<td>1, 2, or 4 bytes (if required)</td>
</tr>
</tbody>
</table>

#### Mod Reg/Opcode R/M

#### Scale Index Base

### Displacement

- Part of memory operand, or…
- 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.)

- **Instruction prefixes**: Up to 4 prefixes of 1 byte each (optional)
  - 1, 2, or 3 bytes
  - 1 byte (if required)

- **Opcode**: 1 byte
  - 1, 2, or 3 bytes
  - 1 byte (if required)

- **ModR/M**: 1 byte
  - 1 byte (if required)

- **SIB**: 1 byte
  - 1, 2, or 4 bytes (if required)

- **Displacement**: 1 byte
  - 1, 2, or 4 bytes (if required)

- **Immediate**: 1, 2, or 4 bytes (if required)

**Immediate**
- Specifies an immediate operand
- Uses little-endian byte order
Example 1

Assembly lang:       addl %eax, %ebx
Machine lang:        01C3
Explanation:

00000001 11000011

Opcode: This is an add instruction whose src operand is a 32-bit register and whose dest operand is a 32-bit register or memory operand
  ModR/M: The M field of the ModR/M byte designates a register
  ModR/M: The src register is EAX
  ModR/M: The dest register is EBX

Observation: Sometimes opcode specifies operation (e.g. add) and format(s) of operand(s)
Example 2

Assembly lang: movl $1, %ebx
Machine lang: BB010000
Explanation:

10111011 00000001 00000000 00000000 00000000

Opcode: This is a mov instruction whose src operand is a 4-byte immediate and whose destination operand is the EBX register
Immediate: The immediate operand is 1

Observation: Sometimes opcode specifies operation and operand(s)
Observation: Immediate operands are in little-endian byte order
Examples 3, 4

Assembly lang:    pushl %eax
Machine lang:    50
Explanation:

01010000
 Opcode: This is a pushl %eax instruction

Assembly lang:    pushl %ecx
Machine lang:    51
Explanation:

01010001
 Opcode: This is a pushl %ecx instruction

Observation: Sometimes opcode specifies operation and operand(s)
Observation: pushl is used often, so is optimized
Example 5

Assembly lang: movl -8(%eax,%ebx,4), %edx
Machine lang: 8B5498F8

Explanation:

10001011 01010100 10011000 11111000

Opcode: This is a mov instruction whose src operand is a 32-bit register or memory operand and whose dest operand is a 32-bit register

ModR/M: The src operand is a 32-bit register, the dest operand is of the form disp(base,index,scale), and the disp is one-byte

ModR/M: The destination register is EDX
SIB: The scale is 4
SIB: The index register is EBX
SIB: The base register is EAX

Displacement: The disp is -8

Observation: Two’s complement notation
Observation: Complicated!!!
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)
## CISC and RISC Characteristics

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Many</strong> instructions</td>
<td><strong>Few</strong> instructions</td>
</tr>
<tr>
<td><strong>Many</strong> memory addressing modes (direct, indirect, base+displacement, indexed, scaled indexed)</td>
<td><strong>Few</strong> memory addressing modes (typically only direct and indirect)</td>
</tr>
<tr>
<td>Hardware interpretation is complex</td>
<td>Hardware interpretation is simple</td>
</tr>
<tr>
<td>Need relatively <strong>few</strong> instructions to accomplish a given job (expressive)</td>
<td>Need relatively <strong>many</strong> instructions to accomplish a given job (not expressive)</td>
</tr>
<tr>
<td>Example: IA-32</td>
<td>Examples: MIPS, SPARC</td>
</tr>
</tbody>
</table>
CISC and RISC History

Stage 1: Programmers compose assembly language
- Important that assembly/machine language be expressive
- CISC dominated (esp. Intel)

Stage 2: Programmers compose 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 took 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
Agenda

Machine Language

The Assembly Process

The Linking Process
The Build Process

1. Preprocess: `mypygm.c` → `mypygm.i`
2. Compile: `mypygm.i` → `mypygm.s`
3. Assemble: `mypygm.s` → `mypygm.o`
4. Link: `mypygm.o` → `mypygm`

Covered in COS 320: Compiling Techniques

Covered here
The “Forward Reference” Problem

Problem

Any assembler must deal with the forward reference problem

• Assembler must generate machine lang code for `jmp mylabel`
• Machine lang `jmp` instr must contain displacement between `mylabel` label and `jmp` instr
• But assembler hasn’t yet seen the def of `mylabel`
  • I.e., the `jmp` instr contains a forward reference to `mylabel`
The “Forward Reference” Solution

Solution
- Assembler performs **2 passes** over assembly lang program
- One to record labels and the address that they denote
- Another to generate code

Different assemblers perform different tasks in each pass

One straightforward design…
The “Forward Reference” Solution

Pass 1
- Assembler traverses assembly lang program to create…
- **Symbol table**
  - Key: label
  - Value: information about label
    - Which section, what offset within that section, …

Pass 2
- Assembler traverses assembly lang program again to create…
- RODATA section
- DATA section
- BSS section
- TEXT section
The “Relocation” Problem

**Problem**

- Assembler must generate machine lang code for `call printf`
- Machine lang `call` instr must contain displacement between `printf` label and `call` instr
- But assembler hasn’t yet seen the def of `printf` label
- And assembler *never will* see the def of `printf` label!!!
  - `printf` label isn’t defined in this .s file

Any assembler must deal with the *relocation* problem
The “Relocation” Solution

Solution:

- Assembler generates as much code as it can
- Assembler generates relocation records

Relocation record

- Request from assembler to linker to patch code at a specified place
The “Relocation” Solution

Pass 1
- Assembler traverses assembly language program to create...
  - Symbol table
    - Key: label
    - Value: information about label
      - Which section, what offset within that section, …

Pass 2
- Assembler traverses assembly language program again to create…
  - RODATA section
  - DATA section
  - BSS section
  - TEXT section
  - Relocation records
    - Each describes a patch that the linker must perform
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...

```assembly
.sect .rodata
msg:
.string "Hi\n"
.sect .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

<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 adds binding to Symbol Table…
Assembler notes that the current section is RODATA
```

```
.section "".rodata"

msg:
.string "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
```
Symbol Table

<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>

- 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)
.section "!.rodata"

msg:
.string "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

<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>

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 at this point, it would mark a spot in RODATA at offset 4
Assembler Pass 1 (cont.)

```assembly
Assembler notes that current section is TEXT
```

```assembly
Assembler does nothing
```

```assembly
Assembler adds binding to Symbol Table…
```

```assembly
.section " .rodata"
msg:
.string "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 (4)

Symbol Table

<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>

- **msg** marks a spot in the RODATA section at offset 0
- **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 "\rodatal"
msg:
   .string "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

<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>

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.)

```assembly
.section " .rodata"
msg:
    .string "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

<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>

- 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:
  .string "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…
Symbol Table

<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>

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 denotes

Beginning of Pass 2
- Assembler resets all section location counters…
Assembler Data Structures (8)

Symbol Table

<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>

Relocation Records

- (Same)

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

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

Location counters reset to 0
Assembler Pass 2

```
Assembler notes that the current section is RODATA

Assembler does nothing

Assembler places bytes in RODATA section, and increments location counter…
```

```
.msg:
    .section "".rodata"
    .string "Hi\n"

.main:
    .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
```
Symbol Table
  • (Same)

Relocation Records
  • (Same)

RODATA Section (location counter: 4)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents (hex)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>ASCII code for ‘H’</td>
</tr>
<tr>
<td>1</td>
<td>69</td>
<td>ASCII code for ‘i’</td>
</tr>
<tr>
<td>2</td>
<td>0A</td>
<td>ASCII code for ‘\n’</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
<td>ASCII code for null char</td>
</tr>
</tbody>
</table>

Location counter incremented to 4

RODATA section contains the bytes comprising the string

TEXT Section (location counter: 0)
  • (Same)
Assembler Pass 2 (cont.)

```
Assembler notes that the current section is TEXT

Assembler updates Symbol Table…
```

```
.section ".rodata"

msg:
    .string "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 (10)

Symbol Table

<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>
</tbody>
</table>

Relocation Records
- (Same)

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

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

main is a global label
Assembler Pass 2 (cont.)

```assembly
Assembler does nothing

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

```assembly
.section "".rodata"
msg:
.string "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 (11)

Symbol Table
• (Same)

Relocation Records
• (Same)

RODATA Section (location counter: 4)
• (Same)

TEXT Section (location counter: 1)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>55</td>
<td>pushl %ebp&lt;br&gt;01010101&lt;br&gt;This is a “pushl %ebp” instruction</td>
</tr>
</tbody>
</table>
Assembler Pass 2 (cont.)

```
.section "".rodata"

msg:
    .string "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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 1-2    | 89 E5    | movl %esp,%ebp  
         |          | 10001001 11 100 101  
         |          | 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:
.string "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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 3-7    | 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

<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>?</td>
<td>?</td>
<td>global</td>
<td>3</td>
</tr>
</tbody>
</table>

Relocation Records
• (Same)

RODATA Section (location counter: 4)
• (Same)

TEXT Section (location counter: 8)
• (Same)
Assembler Data Structures (14)

Symbol Table
• (Same)

Relocation Records

<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

RODATA Section
(location counter: 4)
• (Same)

TEXT Section
(location counter: 8)
• (Same)
Assembler Pass 2 (cont.)

```assembly

.section ".rodata"

msg:
  .string "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...
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>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 8-10   | 83 F8 41 | 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’ |
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: 13)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 11-12  | 75 OD    | jne skip 01110101 00001101  
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 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.)

```assembly
.section " .rodata"
msg:
   .string "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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

13-17  68 ?????????

pushl $msg
001101000 ??????????????4 byte immediate operand
This is the data to be pushed

• 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 Data Structures (17)

Symbol Table
  • (Same)

Relocation Records

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<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 generates a relocation record, thus asking linker to patch code.

RODATA Section
(location counter: 4)
  • (Same)

TEXT Section
(location counter: 18)
  • (Same)
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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 18-22  | E8 ??????? | call printf  
11101000 ????????????????????????????????????????  
This is a “call” instruction with a 4-byte immediate operand  
This is the displacement |
Assembler Data Structures (19)

Symbol Table

<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>?</td>
<td>?</td>
<td>global</td>
<td>3</td>
</tr>
<tr>
<td>printf</td>
<td>?</td>
<td>?</td>
<td>global</td>
<td>4</td>
</tr>
</tbody>
</table>

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

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

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.

RODATA Section
(location counter: 4)
• (Same)

TEXT Section
(location counter: 8)
• (Same)
Assembler Pass 2 (cont.)

Assembler ignores

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

```asm
section "".rodata"
msg:
.string "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
```
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>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 23–25  | 83 C4 04 | addl $4,%esp  
10000011 11 000 100 00000100  
This is some “l” instruction that has a 1 byte immediate operand  
The M field designates a register  
This is an “add” instruction  
The destination register is ESP  
The immediate operand is 4 |
| 26–30  | B8 00000000 | movl $0,%eax  
1011000 00000000000000000000000000000000  
This is an instruction of the form “movl 4-byte-immediate, %eax”  
The immediate operand is 0 |
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>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| 31-32  | 89 EC    | movl %ebp,%esp  
          | 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       | popl %ebp  
          | 01011101  
          | This is a “popl %ebp” instruction |
| 34     | C3       | ret  
          | 11000011  
          | This is a “ret” instruction |
Agenda

Machine Language
The Assembly Process
The Linking Process
Assembler writes its data structures to .o file

Linker:
- Reads .o file
- Write executable binary 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

<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
• Thus linker completes translation of call getchar
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
- Thus linker completes translation of `pushl $msg`
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
- Thus linker completes translation of `call printf`
Linker Finishes

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

**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

**Linker:** reads object files
- **Resolution:** Resolves references to make Symbol Table and code complete
- **Relocation:** Uses Symbol Table and Relocation Records to patch code
- Writes executable binary file
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
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

Using gdb

```bash
$ gcc217 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>
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 0x83 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…
Using objdump

`$ gcc -o detecta detecta.s`

`$ objdump -d detecta`

detecta: file format elf32-i386

Disassembly of section .text:

```
080483b4 <main>:
  80483b4:       55                      push   %ebp
  80483b5:       89 e5                   mov    %esp,%ebp
  80483b7:       e8 dc fe ff ff          call   8048298 <getchar@plt>
  80483bc:       83 f8 41                cmp    $0x41,%eax
  80483bf:       75 0d                   jne    80483ce <skip>
  80483c1:       68 b0 84 04 08          push   $0x80484b0
  80483c6:       e8 fd fe ff ff          call   80482c8 <printf@plt>
  80483cb:       83 c4 04                add    $0x4,%esp
```

```
080483ce <skip>:
  80483ce:       b8 00 00 00 00          mov    $0x0,%eax
  80483d3:       89 ec                   mov    %esp,%ebp
  80483d5:       5d                      pop    %ebp
  80483d6:       c3                      ret
```

Build program; run objdump

Machine language

Assembly language