Machine Language
Jennifer Rexford

Goals of this Lecture
Help you to learn about:
• x86-64 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

The Build Process

<table>
<thead>
<tr>
<th>Preprocess</th>
<th>Compile</th>
<th>Assemble</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>mypgm.c</td>
<td></td>
<td>mypgm.s</td>
<td>mypgm</td>
</tr>
<tr>
<td></td>
<td>mypgm.a</td>
<td>libc.a</td>
<td></td>
</tr>
</tbody>
</table>

Covered in COS 320: Compiling Techniques
Covered here

CISC and RISC
x86-64 machine language instructions are complex
x86-64 is a
• Complex Instruction Set Computer (CISC)

Alternative:
• Reduced Instruction Set Computer (RISC)

CISC and RISC Characteristics

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many instructions</td>
<td>Few instructions</td>
<td></td>
</tr>
<tr>
<td>Many memory addressing modes (direct, indirect, base + displacement, indexed, scaled indexed)</td>
<td>Few memory addressing modes (typically only direct and indirect)</td>
<td></td>
</tr>
<tr>
<td>Hardware interpretation is complex</td>
<td>Hardware interpretation is simple</td>
<td></td>
</tr>
<tr>
<td>Need relatively few instructions to accomplish a given job (expressive)</td>
<td>Need relatively many instructions to accomplish a given job (not expressive)</td>
<td></td>
</tr>
<tr>
<td>Example: x86-64</td>
<td>Examples: MIPS, SPARC</td>
<td></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

x86-64 Machine Language
x86-64 Machine Language after Assembly
x86-64 Machine Language after Linking

x86-64 Machine Language

x86-64 machine language
- Difficult to generalize about x86-64 instruction format
- Many (most!) instructions are exceptions to the rules
- Many instructions use this format…

x86-64 Instruction Format

Instruction prefixes | Opcode | ModR/M | SIB | Displacement | Immediate |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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 byte (if required)</td>
<td>1, 2, 4, or 8 bytes (if required)</td>
</tr>
</tbody>
</table>

Instruction prefix
- Sometimes a repeat count
- Rarely used; don’t be concerned

x86-64 Instruction Format (cont.)

Opcode
- Specifies which operation should be performed
  - Add, move, call, etc.
- Sometimes specifies additional (or less) information

x86-64 Instruction Format (cont.)

ModR/M (register mode, register/Opcode, register/memory)
- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes contains an extension of the opcode

Similar mappings exist for 4-byte, 2-byte, and 1-byte registers

<table>
<thead>
<tr>
<th>ModR/M Register</th>
<th>RAX</th>
<th>RCX</th>
<th>RDX</th>
<th>RBX</th>
<th>RSP</th>
<th>RBP</th>
<th>RSI</th>
<th>RDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>001</td>
<td>002</td>
<td>003</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>010</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>111</td>
<td>120</td>
<td>121</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
x86-64 Instruction Format (cont.)

- **Instruction prefixes**: Up to 4 prefixes of 1 byte each (optional).
- **Opcode**: 1, 2, or 3 bytes (if required).
- **ModR/M**: 1 byte (if required).
- **SIB**: 1 byte (if required).
- **Displacement**: 1, 2, 4, or 8 bytes (if required).

**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.

**Example 1**
- **Assembly lang**: addq %rax, %rbx
- **Machine lang**: 4801c3

**Explanation**: 
- Opcode: This is an add instruction whose src operand is an 8-byte register or memory operand and whose dest operand is a 8-byte register.
- ModR/M: The M field of the ModR/M byte designates a register.
- **Example**: %rax
- **Example**: %rbx

**Observation**: Sometimes opcode specifies operation and format(s) of operand(s).

**Example 2**
- **Assembly lang**: movl $1, %ebx
- **Machine lang**: bb01000000

**Explanation**: 
- Opcode: This is a mov instruction whose src operand is a 4-byte immediate.
- **Example**: $1

**Observation**: Immediate operands are in little-endian byte order.

**Example 3, 4**
- **Assembly lang**: pushq %rax
- **Machine lang**: 50

**Explanation**: 
- Opcode: This is a pushq %rax instruction.

**Observation**: Sometimes opcode specifies operation and operand(s).

**Example**: pushq %rcx
- **Machine lang**: 51

**Explanation**: 
- Opcode: This is a pushq %rcx instruction.

**Observation**: Sometimes opcode specifies operation and operand(s).

**Example**: pushq is used often, so is optimized.
Example 5

Assembly lang: `movl -8(%eax,%ebx,4), %edx`
Machine lang: `678b5498f8`

Explanation:
10100111 10001011 01010110 01010000 01011000

Opcode: This is a mov instruction where the src operand is
a 4-byte register or memory operand and whose dest operand is
a 4-byte register.

ModR/M: The src operand is a register, the
dest operand is of the form disp(base,index,
scale), the base and index registers are
4-byte registers, and the disp is one-byte
ModR/M: The destination register is EDX

SIB: The scale is 4
SIB: The base reg is EAX
SIB: The index register is EBX

Displacement: The disp is -8

Observation: Two’s complement notation
Observation: Complicated!!!

An Example Program

A simple (nonsensical) program:

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

Let’s consider the machine lang equivalent after assembly…

```assembly
.section .rodata
msg1:   .string "Type a char"
msg2:   .string "Hi

.section .text
.globl  main
main:
   movl    $0, %eax
   movq    $msg1, %rdi
   call    printf
   call    getchar
   cmpl    $'A', %eax
   jne     skip
   movl    $0, %eax
   movq    $msg2, %rdi
   call    printf
   skip:
   movl    $0, %eax
   ret
```

Examining Machine Lang: RODATA

Assemble program; run objdump

`$ gcc217 –c detecta.s`
`$ objdump --full-contents --section .rodata detecta.o`

```
detecta.o:     file format elf64-x86-64
Contents of section .rodata:
0000 54797065 20612063 6861723a 2000
0010 4869
0014 ...
```

Offsets

- Assembler does not know addresses
- Assembler knows only offsets
- "Type a char" starts at offset 0
- "Hi" starts at offset 0e

Examining Machine Lang: TEXT

Assemble program; run objdump

`$ gcc217 –c detecta.s`
`$ objdump --disassemble --reloc detecta.o`

```
detecta.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
0:  b8 00 00 00 00
5:  48 c7 c7 00 00 00 00
8: R_X86_64_32S .rodata
  
  c: e8 00 00 00 00
  11:  callq  11 <main+0x11>
  
  d: R_X86_64_PC32 printf-0x4
  11:  e8 00 00 00 00
  16:  callq  16 <main+0x16>
  
  12: R_X86_64_PC32 getchar-0x4
  16:  83 f8 41
  19:  cmp    $0x41,%eax
  19:  75 11
  21:  jne    2c <skip>
  1b:  b8 00 00 00 00
  20:  48 c7 c7 00 00 00 00
  23: R_X86_64_32S .rodata+0xe
  27:  e8 00 00 00 00
  3c:  callq  2c <skip>
  
  28: R_X86_64_PC32 printf-0x4
```

movl $0, %eax
movl $0, %eax

Assembly lang: movl $0, %eax
Machine lang: b800000000
Explanation:

C3 B8 00 00 00 00

Opcode: The destination register is RDI

01111000 00000000 00000000 00000000

Explanation: This is a mov instruction whose src operand is a 4-byte immediate

 Opcode: This is a mov instruction whose src operand is a 4-byte immediate
 immediate
 Opcode: The destination operand is the EAX register
 Immediate: The immediate operand is 0

• mov must contain an address
• Assembler knew offset marked by msg1
• msg1 marks offset 0 relative to beginning of RDATA section
• But assembler did not know address of RDATA section!
• So assembler didn’t know address marked by msg1
• So assembler couldn’t generate this instruction completely

• movl $0, %eax
• movl $0, %eax

movq $msg1, %rdi

Assembly lang: movq $msg1, %rdi
Machine lang: 48 C7 C7 00 00 00 00
Explanation:

01060000 11000000 00000000 00000000 00000000

Opcode: This is a movq instruction with a 4-byte immediate source operand and a 6-byte register destination operand

 Opcode: The destination register is RDI
 Opcode: The immediate (memory address)

14 0

Dear Linker,

Please patch the TEXT section at offsets 08 through 0B. Do an "absolute" type of patch. When you determine the addr of the RDATA section, place that address in the TEXT section at the prescribed place.

Sincerely,
Assembler
call printf

Assembly lang: call printf
Machine lang: e8 00 00 00 00

Explanation:

11101000 00000000 00000000 00000000 00000000
Opcode: This is a call instruction with a 4-byte displacement
Disp: The displacement is 00000000, (0)

- call must contain a displacement
- Assembler had to generate the displacement:
  [addr of printf] – [addr after call instr]
- But assembler didn’t know addr of printf
  - printf isn’t even present yet!
- So assembler couldn’t generate this instruction completely

Relocation Record 2

$ gcc217 –c detecta.s
$ objdump –-disassemble --reloc detecta.o
detecta.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
  0:  b8 00 00 00 00    mov    $0x0,%eax
  5:  48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  8: R_X86_64_32S    .rodata
  c:  e8 00 00 00 00    callq  11 <main+0x11>
  d: R_X86_64_PC32    printf-0x4
  11: e8 00 00 00 00    callq  16 <main+0x16>
  12: R_X86_64_PC32    getchar-0x4
  16:  83 f8 41    cmp    $0x41,%eax
  19:  75 11    jne    2c <skip>
  1b:  b8 00 00 00 00    mov    $0x0,%eax
  20:  48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  23: R_X86_64_32S    .rodata+0xe
  27:  e8 00 00 00 00    callq  2c <skip>
  28: R_X86_64_PC32    printf-0x4

Relocation Record 3

Dear Linker,
Please patch the TEXT section at offsets 0d through 10H. Do a “relative” type of patch. When you determine the addr of printf, compute [addr of printf] – [addr after call] and place the result at the prescribed place.

Sincerely,
Assembler

Relocation Record 2

call getchar

Assembly lang: call getchar
Machine lang: e8 00 00 00 00

Explanation:

11101000 00000000 00000000 00000000 00000000
Opcode: This is a call instruction with a 4-byte displacement
Disp: The displacement is 00000000, (0)

- call must contain a displacement
- Assembler had to generate the displacement:
  [addr of getchar] – [addr after call instr]
- But assembler didn’t know addr of getchar
  - getchar isn’t even present yet!
- So assembler couldn’t generate this instruction completely

Relocation Record 3
Dear Linker,

Please patch the TEXT section at offsets 12 through 15. Do a "relative" type of patch. When you determine the addr of getchar, compute $[\text{offset of } \text{getchar}] - [\text{addr after call}]$ and place the result at the prescribed place.

Sincerely,
Assembler

Assembly lang:
```
cmpl $'A', %eax
```

Machine lang:
```
83 f8 41
```

Explanation:

- $\text{cmp}l$ must contain a displacement
- Assembler had to generate the displacement: $[\text{addr of skip}] - [\text{addr after the instr}]$
- Assembler did not know addr of skip
- So assembler could generate this instruction completely $2c_8 = 15_8 = 11_2 = 17_0$
movl $0, %eax

```
# Assembly lang:
movl $0, %eax
# Machine lang:

```

**Explanation:**

10111

- **Opcode:** This is a mov instruction whose src operand is a 4-byte immediate
- **Operand:** The destination operand is the EAX register
- **Immediate:** The immediate operand is 0

---

movq $msg2, %rdi

```
# Assembly lang:
movq $msg2, %rdi
# Machine lang:

```

**Explanation:**

0100100

- **Opcode:** This is a movq instruction with a 4-byte immediate source operand and an 8 byte register destination operand
- **Operand:** The destination register is RDI (cont.)
- **Disp:** The immediate (memory address) is 0
- **Address:** The immediate (memory address) is 0
- **Address:** The immediate (memory address) is 0

---

Relocation Record 4

```
# Assembly lang:

```

**Explanation:**

- movq must contain an address
- Assembler knew offset marked by msg2
- msg2 marks offset 0e, relative to beginning of RODATA section
- But assembler did not know address of RODATA section!
- So assembler didn’t know address marked by msg2
- So assembler couldn’t generate this instruction completely

---

Relocation Record 4

```

- Dear Linker,

  Please patch the TEXT section at offsets 23, through 26. Do an "absolute" type of patch. When you determine the addr of the RODATA section, add 0e, to that address, and place the result in the TEXT section at the prescribed place.

  Sincerely,
  Assembler
$ gcc217 –c detecta.s
$ objdump –-disassemble --reloc detecta.o

detecta.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
0:
   b8 00 00 00 00
   mov    $0x0,%eax
5:
   48 c7 c7 00 00 00 00
   mov    $0x0,%rdi
8: R_X86_64_32S
   .rodata
   c:
   e8 00 00 00 00
   callq  11 <main+0x11>
d: R_X86_64_PC32
   printf-0x4
   11:
   e8 00 00 00 00
   callq  16 <main+0x16>
12: R_X86_64_PC32
   getchar-0x4
   16:
   83 f8 41
   cmp    $0x41,%eax
19:
   75 11
   jne    2c <skip>
1b:
   b8 00 00 00 00
   mov    $0x0,%eax
20:
   48 c7 c7 00 00 00 00
   mov    $0x0,%rdi
23: R_X86_64_32S
   .rodata+0xe
   27:
   e8 00 00 00 00
   callq  2c <skip>
28: R_X86_64_PC32
   printf-0x4
   2c:
   b8 00 00 00 00
   mov    $0x0,%eax
31:
   c3
   retq

Assembly lang: call printf
Machine lang: e8 00 00 00 00

Explanation:

11101000
00000000 00000000 00000000 00000000
 Opcode: This is a call instruction with a 4-byte displacement
 Disp: The displacement is 00000000, [0]

- call must contain a displacement
- Assembler must generate the displacement: [addr of printf] – [addr after call instr]
- But assembler didn’t know addr of printf
- printf isn’t even present yet
- So assembler couldn’t generate this instruction completely

Dear Linker,

Please patch the TEXT section at offsets 28H through 2bH. Do a “relative” type of patch. When you determine the addr of printf, compute [addr of printf] – [addr after call] and place the result at the prescribed place.

Sincerely,
Assembler
$ gcc217 –c detecta.s
$ objdump --disassemble --reloc detecta.o
detecta.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
0:
  b8 00 00 00 00
     mov    $0x0,%eax
5:
  48 c7 c7 00 00 00 00
     mov    $0x0,%rdi
8: R_X86_64_32S
    .rodata
11:
  e8 00 00 00 00
     callq  11 <main+0x11>
14:
  00 00 00 00 00
     mov    $0x41,%eax
c:
  e8 00 00 00 00
     callq  16 <main+0x16>
18: R_X86_64_PC32
    printf-0x4
11:
  e8 00 00 00 00
     callq  16 <main+0x16>
12: R_X86_64_PC32
    getchar-0x4
16:
  83 f8 41
     cmp    $0x41,%eax
19:
  75 11
     jne    2c <skip>
1b:
  b8 00 00 00 00
     mov    $0x0,%eax
20:
  48 c7 c7 00 00 00 00
     mov    $0x0,%rdi
23: R_X86_64_32S
    .rodata+0xe
27:
  e8 00 00 00 00
     callq  2c <skip>
28: R_X86_64_PC32
    printf-0x4
000000000000002c <skip>:
2c:
  b8 00 00 00 00
     mov    $0x0,%eax
31:
  c3
     retq

Assembly lang: ret
Machine lang: c3
Explanation:
11000011
Opcode: This is a ret (alias retq) instruction

Agenda

x86-64 Machine Language
x86-64 Machine Language after Assembly
x86-64 Machine Language after Linking

From Assembler to Linker

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

Linker Resolution

Resolution
- Linker resolves references
For this program, linker:
  - Notes that labels getchar and printf are unresolved
  - Fetches machine language code defining getchar and printf from libc.a
  - Adds that code to TEXT section
  - Adds more code (e.g. definition of _start) to TEXT section too
  - Adds code to other sections too

Linker Relocation

Relocation
- Linker patches ("relocates") code
- Linker traverses relocation records, patching code as specified
Examine Machine Lang: RODATA

- Link program; run objdump
- Additional Code
- Examining Machine Lang: TEXT
- Link program; run objdump

RODATA addresses, not offsets
- RODATA is at .00400638
- Starts with some header info
- Real start of RODATA is at .00400648
- "Type a char: " starts at .00400648
- "Hi\n" starts at .00400656

Additional code

movq $msg1, %rdi

Recall: Real addr of RODATA = .00400648

Linker replaced 00000000 with real addr of RODATA + 0
= .00400648 + 0
= .00400648

Addr of printf = .00400318

Linker replaced 00000000, with [addr after call]
= .00400318, ... .00400525
= .fffffff3
= -301

Addr of getchar = .00400418

Linker replaced 00000000, with [addr after call]
= .00400418, ... .0040052a
= .fffffffa
= -274
Summary

x86-64 Machine Language
- CISC: many instructions, complex format
- Fields: prefix, opcode, modR/M, SIB, displacement, immediate

Assembler
- Reads assembly language file
- Generates TEXT, RODATA, DATA, BSS sections
- Containing machine language code
- Generates relocation records
- Writes object (.o) file

Linker
- Reads object (.o) file(s)
- Generates TEXT, RODATA, DATA, BSS sections
- Containing machine language code
- Resolves references to make code complete
- Does relocation: traverses relocation records to patch code
- Writes executable binary file