CS107, Lecture 12
Assembly: Arithmetic, Logic and Condition Codes

Reading: B&O 3.5-3.6
Learning Goals

• Learn how to perform arithmetic and logical operations in assembly
• Learn how assembly implements loops and control flow
Plan For Today

• Recap: Assembly and mov
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• Practice: Reverse-Engineering
Plan For Today

• Recap: Assembly and mov
  • Data and Register Sizes
  • The lea Instruction
  • Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• Practice: Reverse-Engineering
Assembly

- **Assembly code** is a human-readable form of the machine code your computer actually executes when running your programs.
- Assembly works at a lower level of abstraction than C code. It works with 64-bit spaces called **registers** that act as “scratch paper” for the processor.
- Operations in your C program ultimately are converted to operations that read or write to registers and perform calculations on these registers.
Our First Assembly

```assembly
600000000004005b6 <sum_array>:
4005b6:   ba 00 00 00 00 00   mov    $0x0,%edx
4005bb:   b8 00 00 00 00 00   mov    $0x0,%eax
4005c0:   eb 09             jmp    4005cb <sum_array+0x15>
4005c2:   48 63 ca          movslq  %edx,%rcx
4005c5:   03 04 8f          add    (%rdi,%rcx,4),%eax
4005c8:   83 c2 01          add    $0x1,%edx
4005cb:   39 f2             cmp    %esi,%edx
4005cd:   7c f3             jl     4005c2 <sum_array+0xc>
4005cf:   f3 c3             repz retq
```
Our First Assembly

Each instruction has an operation name ("opcode").
Our First Assembly

00000000004005b6 <sum_array>:

0005b6:  ba 00 00 00 00
0005bb:  b8 00 00 00 00
0005c0:  eb 09
0005c2:  48 63 ca
0005c5:  03 04 8f
0005c8:  83 c2 01
0005cb:  39 f2
0005cd:  7c f3
0005cf:  f3 c3

mov $0x0,%edx
mov $0x0,%eax
jmp 4005cb <sum_array+0x15>
movslq %edx,%rcx
add (%rdi,%rcx,4),%eax
add $0x1,%edx
cmp %esi,%edx
j1 4005c2 <sum_array+0xc>

Each instruction can also have arguments ("operands").
The `mov` instruction copies bytes from one place to another.

\[ \text{mov} \quad \text{src}, \text{dst} \]

The `src` and `dst` can each be one of:

- Immediate (constant value, like a number)
- Register
- Memory Location (at most one of `src`, `dst`
# Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>$Imm$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$R[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_a)$</td>
<td>$M[R[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b)$</td>
<td>$M[Imm + R[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i)$</td>
<td>$M[R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i)$</td>
<td>$M[Imm + R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_i,s)$</td>
<td>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_i,s)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i,s)$</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i,s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Figure 3.3**  **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor $s$ must be either 1, 2, 4, or 8.
## Memory Location Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>Address 0x104 (no $)</td>
</tr>
<tr>
<td>(%rax)</td>
<td>What’s in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>What’s in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of what’s in %rax and %rdx</td>
</tr>
<tr>
<td>4(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx, plus 4</td>
</tr>
<tr>
<td>(, %rcx, 4)</td>
<td>What’s in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx, plus 8</td>
</tr>
</tbody>
</table>
Practice With Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

Reminder:

**Most general form:** $\text{Imm}(r_b, r_i, s)$

$\text{Imm} + R[r_b] + R[r_i] \times s$

Also: $260d = 0x104$
Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

Reminder:

**Most general form:** \( \text{imm}(r_b, r_i, s) \)

\[ \text{imm} + R[r_b] + R[r_i] \times s \]

Also: 260d = 0x104
Plan For Today

• **Recap:** Assembly and `mov`
• Data and Register Sizes
• The `lea` Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• **Practice:** Reverse-Engineering
Data Sizes

- Because of its 16-bit origins, Intel uses "word" to mean 16-bits (two bytes)
- 32-bit words are referred to as "double words" ("d" suffix)
- 64-bit quantities are referred to as "quad words" ("q" suffix)
- This table shows the x86 primitive data types of C (Figure 3.1 in the textbook)

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly-code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
</tr>
</tbody>
</table>
# Register Sizes

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
</tr>
<tr>
<td>Register</td>
<td>Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%bp</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%bpl</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%sp</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%spl</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8d</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8w</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8b</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9d</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9w</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9b</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10d</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10w</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10b</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11d</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11w</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11b</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Register Sizes

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
</tr>
</tbody>
</table>
Register Responsibilities

Some registers take on special responsibilities during program execution.

• `%rax` stores the return value
• `%rdi` stores the first parameter to a function
• `%rsi` stores the second parameter to a function
• `%rdx` stores the third parameter to a function
• `%rip` stores the address of the next instruction to execute
• `%rsp` stores the address of the current top element on the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
mov Variants

- **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move: movb, movw, movl, movq

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly-code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
</tr>
</tbody>
</table>
**mov Variants**

- **mov** only updates the specific register bytes or memory locations indicated.
- **Exception:** **movl** writing to a register will also set high order 4 bytes to 0.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly-code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
</tr>
</tbody>
</table>
The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.

The regular **movq** instruction can only take 32-bit immediates.

64-bit immediate as source, only register as destination.

\[
\text{movabsq $0x0011223344556677$, %rax}
\]
• There are two mov instructions that can be used to copy a smaller source to a larger destination: movz and movs.

• movz fills the remaining bytes with zeros

• movs fills the remaining bytes by sign-extending the most significant bit in the source.

• The source must be from memory or a register, and the destination is a register.
# movz and movs

## Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ</td>
<td>S, R</td>
<td>$R \leftarrow \text{ZeroExtend}(S)$</td>
</tr>
<tr>
<td>movzbw</td>
<td></td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>movzbl</td>
<td></td>
<td>Move zero-extended byte to double word</td>
</tr>
<tr>
<td>movzw1</td>
<td></td>
<td>Move zero-extended word to double word</td>
</tr>
<tr>
<td>movzbq</td>
<td></td>
<td>Move zero-extended byte to quad word</td>
</tr>
<tr>
<td>movzwq</td>
<td></td>
<td>Move zero-extended word to quad word</td>
</tr>
</tbody>
</table>
### movz and movs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ $S, R$</td>
<td>$R \leftarrow \text{SignExtend}(S)$</td>
<td>Move with sign extension</td>
</tr>
<tr>
<td>movsbw</td>
<td>Move sign-extended byte to word</td>
<td></td>
</tr>
<tr>
<td>movsbl</td>
<td>Move sign-extended byte to double word</td>
<td></td>
</tr>
<tr>
<td>movswl</td>
<td>Move sign-extended word to double word</td>
<td></td>
</tr>
<tr>
<td>movsbq</td>
<td>Move sign-extended byte to quad word</td>
<td></td>
</tr>
<tr>
<td>movswq</td>
<td>Move sign-extended word to quad word</td>
<td></td>
</tr>
<tr>
<td>movslq</td>
<td>Move sign-extended double word to quad word</td>
<td></td>
</tr>
<tr>
<td>cltq</td>
<td>$%rax \leftarrow \text{SignExtend}(%eax)$</td>
<td>Sign-extend $%eax$ to $%rax$</td>
</tr>
</tbody>
</table>
Practice with mov

- For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., mov can be movb, movw, movl, movq)

```assembly
mov___ %eax, (%rsp)
mov___ (%rax), %dx
mov___ $0xFF, %bl
mov___ (%rsp,%rdx,4), %dl
mov___ (%rdx), %rax
mov___ %dx, (%rax)
```
Practice with mov

• For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., mov can be movb, movw, movl, movq)

  movl  %eax, (%rsp)
  movw  (%rax), %dx
  movb  $0xFF, %bl
  movb  (%rsp,%rdx,4), %dl
  movq  (%rdx), %rax
  movw  %dx, (%rax)
Plan For Today

- Recap: Assembly and mov
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Control
  - Condition Codes
  - Assembly Instructions
- **Practice**: Reverse-Engineering
The `lea` instruction copies an “effective address” from one place to another.

```
lea src, dst
```

Unlike `mov`, which copies data at the address `src` to the destination, `lea` copies the value of `src` itself to the destination.

Examples: if `%rax` holds value `x` and `%rcx` holds value `y`:

- `leaq 6(%rax), %rdx` : `%rdx` now holds `x + 6`
- `leaq (%rax,%rcx), %rdx` : `%rdx` now holds `x + y`
- `leaq (%rax,%rcx,4), %rdx` : `%rdx` now holds `x + 4*y`
- `leaq 7(%rax,%rax,8), %rdx` : `%rdx` now holds `7 + 9x`
- `leaq 0xA(,%rcx,4), %rdx` : `%rdx` now holds `10 + 4y`
- `leaq 9(%rax,%rcx,2), %rdx` : `%rdx` now holds `9 + x + 2y`
Plan For Today

- **Recap**: Assembly and `mov`
- Data and Register Sizes
- The `lea` Instruction
- **Logical and Arithmetic Operations**
- Control
  - Condition Codes
  - Assembly Instructions
- **Practice**: Reverse-Engineering
Unary Instructions

The following instructions operate on a single operand (register or memory):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc D</td>
<td>$D \leftarrow D + 1$</td>
<td>Increment</td>
</tr>
<tr>
<td>dec D</td>
<td>$D \leftarrow D - 1$</td>
<td>Decrement</td>
</tr>
<tr>
<td>neg D</td>
<td>$D \leftarrow -D$</td>
<td>Negate</td>
</tr>
<tr>
<td>not D</td>
<td>$D \leftarrow \sim D$</td>
<td>Complement</td>
</tr>
</tbody>
</table>

Examples:

- incq 16(%rax)
- dec %rdx
- not %rcx
The following instructions operate on two operands (register or memory). Both cannot be memory locations. Read it as, e.g. “Subtract S from D”:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add S, D</td>
<td>$D \leftarrow D + S$</td>
<td>Add</td>
</tr>
<tr>
<td>sub S, D</td>
<td>$D \leftarrow D - S$</td>
<td>Subtract</td>
</tr>
<tr>
<td>imul S, D</td>
<td>$D \leftarrow D \times S$</td>
<td>Multiply</td>
</tr>
<tr>
<td>xor S, D</td>
<td>$D \leftarrow D \oplus S$</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>or S, D</td>
<td>$D \leftarrow D</td>
<td>S$</td>
</tr>
<tr>
<td>and S, D</td>
<td>$D \leftarrow D &amp; S$</td>
<td>And</td>
</tr>
</tbody>
</table>

Examples:

- `addq %rcx,(%rax)`
- `xorq $16,(%rax, %rdx, 8)`
- `subq %rdx,8(%rax)`
Large Multiplication

• Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?

• If you specify two operands to `imul`, it multiplies them together and truncates until it fits in a 64-bit register.

\[
\text{imul } S, D \quad D \leftarrow D \times S \quad \text{Multiply}
\]

• If you specify one operand, it multiplies that by `%rax`, and splits the product across 2 registers. It puts the high-order 64 bits in `%rdx` and the low-order 64 bits in `%rax`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imulq S</code></td>
<td>R[%rdx]:R[%rax] ← S × R[%rax]</td>
<td>Signed full multiply</td>
</tr>
<tr>
<td><code>mulq S</code></td>
<td>R[%rdx]:R[%rax] ← S × R[%rax]</td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>
Division and Remainder

- Terminology: \texttt{dividend} \ divisor = \texttt{quotient} + \texttt{remainder}
- \texttt{x86-64} supports dividing up to a 128-bit value by a 64 bit value.
- The high-order 64 bits of the dividend are in \texttt{%rdx}, and the low-order 64 bits are in \texttt{%rax}. The divisor is the operand to the instruction.
- The quotient is stored in \texttt{%rax}, and the remainder in \texttt{%rdx}.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{idivq S}</td>
<td>\texttt{R[%rdx] \leftarrow R[%rdx]:R[%rax] \mod S}; \texttt{R[%rax] \leftarrow R[%rdx]:R[%rax] \div S}</td>
<td>Signed divide</td>
</tr>
<tr>
<td>\texttt{divq S}</td>
<td>\texttt{R[%rdx] \leftarrow R[%rdx]:R[%rax] \mod S}; \texttt{R[%rax] \leftarrow R[%rdx]:R[%rax] \div S}</td>
<td>Unsigned divide</td>
</tr>
</tbody>
</table>
Division and Remainder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cqto</td>
<td>R[%rdx]:R[%rax] ← SignExtend(R[%rax])</td>
<td>Convert to oct word</td>
</tr>
<tr>
<td>idivq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Signed divide</td>
</tr>
<tr>
<td>divq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Unsigned divide</td>
</tr>
</tbody>
</table>

- **Terminology:** **dividend**  divisor = quotient + remainder
- The high-order 64 bits of the dividend are in %rdx, and the low-order 64 bits are in %rax. The divisor is the operand to the instruction.

- Most division uses only 64 bit dividends. The **cqto** instruction sign-extends the 64-bit value in %rax into %rdx to fill both registers with the dividend, as the division instruction expects.
Shift Instructions

The following instructions operate on two operands, one the shift amount and the other the destination to shift. The shift amount \( k \) can be either an immediate value, or the byte register \( %cl \) (and only that register!)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sal ( k, D )</td>
<td>( D \leftarrow D \ll k )</td>
<td>Left shift</td>
</tr>
<tr>
<td>shl ( k, D )</td>
<td>( D \leftarrow D \ll k )</td>
<td>Left shift (same as sal)</td>
</tr>
<tr>
<td>sar ( k, D )</td>
<td>( D \leftarrow D \gg_A k )</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>shr ( k, D )</td>
<td>( D \leftarrow D \gg_L k )</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

Examples:

- `shll $3,(%rax)`
- `shr %cl,(%rax,%rdx,8)`
- `sar $4,8(%rax)`
When using `%cl`, the width of what you are shifting determines how much of `%cl` it is shifted by.

For `w` bits of data, it looks at the low-order $\log_2(w)$ bits of `%cl` to know how much to shift.

- If `%cl = 0xff`, then: `shlb` shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. `shll` shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sal k, D</code></td>
<td>$D \leftarrow D &lt;&lt; k$</td>
<td>Left shift</td>
</tr>
<tr>
<td><code>shl k, D</code></td>
<td>$D \leftarrow D &lt;&lt; k$</td>
<td>Left shift (same as <code>sal</code>)</td>
</tr>
<tr>
<td><code>sar k, D</code></td>
<td>$D \leftarrow D &gt;&gt;_A k$</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td><code>shr k, D</code></td>
<td>$D \leftarrow D &gt;&gt;_L k$</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>
Assembly Exploration

• Let’s pull these different commands together and see how some C code we write may be translated into assembly.

• Compiler Explorer is a handy website that lets you write C code and see its assembly translation without having to log into Myth or compile/disassemble a program. Let’s check it out!

• [https://godbolt.org/z/NLYhVf](https://godbolt.org/z/NLYhVf)
// Returns the sum of x and the first element in arr
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}

// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}
Assembly Exercise 1

000000000004005ac <sum_example1>:
  4005bd:  8b 45 e8    mov   %esi,%eax
  4005c3:  01 d0    add   %edi,%eax
  4005cc:  c3    retq

Which of the following is most likely to have generated the above assembly?

// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}

// B)
int sum_example1(int x, int y) {
    return x + y;
}

// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
Assembly Exercise 2

00000000000400578 <sum_example2>:

400578:  8b 47 0c            mov 0xc(%rdi),%eax
40057b:  03 07            add (%rdi),%eax
40057d:  2b 47 18            sub 0x18(%rdi),%eax
400580:  c3            retq

int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly above represents the C code’s sum variable?
What location or value in the assembly above represents the C code’s `sum` variable?

[%eax]
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly code above represents the C code’s 6 (as in arr[6])?
int sum_example2(int arr[]) {  
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly code above represents the C code’s 6 (as in arr[6])?

0x18
Plan For Today

• **Recap:** Assembly and *mov*
• Data and Register Sizes
• The *lea* Instruction
• Logical and Arithmetic Operations

• **Control**
  • Condition Codes
  • Assembly Instructions

• **Practice:** Reverse-Engineering
Control

• In C, we have control flow statements like if, else, while, for, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.

• This boils down to *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.

• How is this represented in assembly?
  • A way to store conditions that we will check later
  • Assembly instructions whose behavior is dependent on these conditions
Control

• In C, we have control flow statements like if, else, while, for, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.

• This boils down to *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.

• How is this represented in assembly?
  • A way to store conditions that we will check later
  • Assembly instructions whose behavior is dependent on these conditions
Plan For Today

• Recap: Assembly and mov
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• Practice: Reverse-Engineering
Alongside normal registers, the CPU also has single-bit condition code registers. These can be updated and read to influence what to do next. They are automatically updated by the most recent arithmetic or logical operation.

Most common condition codes:

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
- **ZF**: Zero flag. The most recent operation yielded zero.
- **SF**: Sign flag. The most recent operation yielded a negative value.
- **OF**: Overflow flag. The most recent operation caused a two’s-complement overflow—either negative or positive.
Condition Codes

Common Condition Codes

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

- **ZF**: Zero flag. The most recent operation yielded zero.

- **SF**: Sign flag. The most recent operation yielded a negative value.

- **OF**: Overflow flag. The most recent operation caused a two’s-complement overflow—either negative or positive.

Which flag would be set after this code?

```c
int a = 5;
int b = -5;
int t = a + b;
```
Common Condition Codes

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

- **ZF**: Zero flag. The most recent operation yielded zero.

- **SF**: Sign flag. The most recent operation yielded a negative value.

- **OF**: Overflow flag. The most recent operation caused a two’s-complement overflow—either negative or positive.

Which flag would be set after this code?

```c
int a = 5;
int b = -5;
int t = a + b;
```
Condition Codes

Common Condition Codes

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

- **ZF**: Zero flag. The most recent operation yielded zero.

- **SF**: Sign flag. The most recent operation yielded a negative value.

- **OF**: Overflow flag. The most recent operation caused a two’s-complement overflow—either negative or positive.

Which flag would be set after this code?

```cpp
int a = 5;
int b = -20;
int t = a + b;
```
Condition Codes

Common Condition Codes

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

- **ZF**: Zero flag. The most recent operation yielded zero.

- **SF**: Sign flag. The most recent operation yielded a negative value.

- **OF**: Overflow flag. The most recent operation caused a two’s-complement overflow—either negative or positive.

Which flag would be set after this code?

```c
int a = 5;
int b = -20;
int t = a + b;
```
Condition Codes

- Previously-discussed arithmetic and logical instructions update these flags. `lea` does not (it was intended only for address computations).
- Logical operations (`xor`, etc.) set carry and overflow flags to zero.
- Shift operations set the carry flag to the last bit shifted out, and set the overflow flag to zero.
- For more complicated reasons, `inc` and `dec` set the overflow and zero flags, but leave the carry flag unchanged.
Setting Condition Codes

• In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.

• The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP</strong></td>
<td><strong>S_1, S_2</strong></td>
<td><strong>S_2 - S_1</strong></td>
</tr>
<tr>
<td><strong>cmpb</strong></td>
<td><strong>S_2 - S_1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cmpw</strong></td>
<td><strong>S_2 - S_1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cmp1</strong></td>
<td><strong>S_2 - S_1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cmpq</strong></td>
<td><strong>S_2 - S_1</strong></td>
<td></td>
</tr>
</tbody>
</table>

• **NOTE:** the operand order can be confusing!
Setting Condition Codes

• In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.

• The test instruction is like the AND instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>S₁, S₂</td>
<td>S₂ &amp; S₁</td>
</tr>
<tr>
<td>testb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>testw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>testl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>testq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Setting Condition Codes

- The **test** instruction is like the AND instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>S₁, S₂</td>
<td>S₂ &amp; S₁</td>
</tr>
<tr>
<td>testb</td>
<td></td>
<td>Test byte</td>
</tr>
<tr>
<td>testw</td>
<td></td>
<td>Test word</td>
</tr>
<tr>
<td>testl</td>
<td></td>
<td>Test double word</td>
</tr>
<tr>
<td>testq</td>
<td></td>
<td>Test quad word</td>
</tr>
</tbody>
</table>

- **Cool trick:** if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!
In C, we have control flow statements like *if*, *else*, *while*, *for*, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.

This boils down to *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false, etc.

How is this represented in assembly?

- A way to store conditions that we will check later
  - *Assembly instructions whose behavior is dependent on these conditions*
Plan For Today

• Recap: Assembly and mov
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• Practice: Reverse-Engineering
There are three common instruction types that use condition codes:
• **set** instructions conditionally set a byte to 0 or 1
• new versions of **mov** instructions conditionally move data
• **jmp** instructions conditionally jump to a different next instruction
## Conditionally Setting Bytes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition (1 if true, 0 if false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>setns D</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnle</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setnl</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnge</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
## Conditionally Moving Data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Move Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove S,R</td>
<td>cmovz</td>
<td>Equal / zero (ZF=1)</td>
</tr>
<tr>
<td>cmovne S,R</td>
<td>cmovnz</td>
<td>Not equal / not zero (ZF=0)</td>
</tr>
<tr>
<td>cmovs S,R</td>
<td></td>
<td>Negative (SF=1)</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td></td>
<td>Nonnegative (SF=0)</td>
</tr>
<tr>
<td>cmovg S,R</td>
<td>cmovnle</td>
<td>Greater (signed &gt;) (SF=0 and SF=OF)</td>
</tr>
<tr>
<td>cmovge S,R</td>
<td>cmovnl</td>
<td>Greater or equal (signed &gt;=) (SF=OF)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>cmovle S,R</td>
<td>cmovng</td>
<td>Less or equal (signed &lt;=) (ZF=1 or SF!=OF)</td>
</tr>
<tr>
<td>cmova S,R</td>
<td>cmovnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>cmovae S,R</td>
<td>cmovnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>cmovb S,R</td>
<td>cmovnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
The `jmp` instruction jumps to another instruction in the assembly code ("Unconditional Jump").

```
jmp Label          (Direct Jump)
jmp *Operand       (Indirect Jump)
```

The destination can be hardcoded into the instruction (direct jump):

```
jmp 404f8 <loop+0xb>  # jump to instruction at 0x404f8
```

The destination can also be read from a memory location (indirect jump):

```
jmp *%rax           # jump to instruction at address in %rax
```
Conditional Jumps

There are also variants of `jmp` that jump only if certain conditions are true ("Conditional Jump"). The jump location for these must be hardcoded into the instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Label</td>
<td>jz</td>
<td>Equal / zero (ZF=1)</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>Not equal / not zero (ZF=0)</td>
</tr>
<tr>
<td>js Label</td>
<td></td>
<td>Negative (SF=1)</td>
</tr>
<tr>
<td>jns Label</td>
<td></td>
<td>Nonnegative (SF=0)</td>
</tr>
<tr>
<td>jg Label</td>
<td>jnle</td>
<td>Greater (signed &gt;) (SF=0 and SF=OF)</td>
</tr>
<tr>
<td>jge Label</td>
<td>jnl</td>
<td>Greater or equal (signed &gt;=) (SF=OF)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>Less or equal (signed &lt;=) (ZF=1 or SF!=OF)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
Loops and Control Flow

Jump instructions are critical to implementing loops and control flow in assembly. Let’s see why!
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

Set %eax (i) to 0.
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:    mov    $0x0,%eax
0x0000000000400575 <+5>:    jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:    add    $0x1,%eax
0x000000000040057a <+10>:   cmp    $0x63,%eax
0x000000000040057d <+13>:   jle     0x400577 <loop+7>
0x000000000040057f <+15>:   repz retq
```

Jump to another instruction.
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

Compare `%eax` (i) to 0x63 (99) by calculating `%eax` – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

```assembly
0x0000000000400570 <+0>:    mov     $0x0,%eax
0x0000000000400575 <+5>:    jmp     0x40057a <loop+10>
0x0000000000400577 <+7>:    add     $0x1,%eax
0x000000000040057a <+10>:   cmp     $0x63,%eax
0x000000000040057d <+13>:   jle     0x400577 <loop+7>
0x000000000040057f <+15>:   repz retq
```
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

The `jle` instruction means “jump if less than or equal”. The sign flag indicates the result was negative, so we jump.

```
0x0000000000400570 <+0>:    mov  $0x0,%eax
0x0000000000400575 <+5>:    jmp  0x40057a <loop+10>
0x0000000000400577 <+7>:    add  $0x1,%eax
0x000000000040057a <+10>:   cmp  $0x63,%eax
0x000000000040057d <+13>:   jle  0x400577 <loop+7>
0x000000000040057f <+15>:   repz retq
```
Loops and Control Flow

void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

jle means “jump if less than or equal”. The sign flag indicates the result was negative, so we jump.
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

We continue in this pattern until we do not make this conditional jump. When will that be?
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

We will stop looping when this comparison says that \( \text{eax} - 0x63 > 0 \)!
Loops and Control Flow

```c
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

Then, we return from the function.
Plan For Today

• Recap: Assembly and mov
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• Practice: Reverse-Engineering
long test(long x, long y, long z) {
    long val = _____________;

    if (__________) {
        if (__________) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

long test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = __________;
    if (________) {
        if (________) {
            val = ________;
        } else {
            val = ________;
        }
    } else if (______) {
        val = ________;
    }
    return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret

    .L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret

    .L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax

    .L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y;

    if (__________) {
        if (__________) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y;
    if (__________) {
        if (__________) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }
    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;

    if (__________) {
        if (__________) {
            val = __________;
        } else {
            val = __________;
        }
    } else if (_______) {
        val = __________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
    .L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
    .L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
    .L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (__________) {
        if (__________) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }
    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (__________) {
            val = __________;
        } else {
            val = __________;
        }
    } else if (_______) {
        val = __________;
    }

    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (__________) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

# test:
leaq (%rdi,%rsi), %rax
addq %rdx, %rax
cmpq $-3, %rdi
jge .L2

cmpq %rdx, %rsi
jge .L3
movq %rdi, %rax
imulq %rsi, %rax
ret

.L3:
    movq %rsi, %rax
imulq %rdx, %rax
ret

.L2:
    cmpq $2, %rdi
jle .L4
movq %rdi, %rax
imulq %rdx, %rax
.L4:
    rep; ret
```c
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }
    return val;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
```
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = _________;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }

    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x;
        } else {
            val = _________;
        }
    } else if (_______) {
        val = _________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret

.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret

.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
    ret

.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x;
        } else {
            val = __________;
        }
    } else if (_______) {
        val = __________;
    }
    return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
  .L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
  .L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
  .L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < 3) {
        if (y < z) {
            val = x * y;
        } else {
            val = _________;
        }
    } else if (______) {
        val = _________;
    }
    return val;
}

# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = __________;
        }
    } else if (_______) {
        val = __________;
    }
    return val;
}

Practice: Reverse Engineering

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret

.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret

.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax

.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = __________;
        }
    } else if (_______) {
        val = __________;
    }

    return val;
}
long test(long x, long y, long z) {
  long val = x + y + z;

  if (x < -3) {
    if (y < z) {
      val = x * y;
    } else {
      val = y;
    }
  } else if (_______) {
    val = __________;
  }

  return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
  leaq (%rdi,%rsi), %rax
  addq %rdx, %rax
  cmpq $-3, %rdi
  jge .L2
  cmpq %rdx, %rsi
  jge .L3
  movq %rdi, %rax
  imulq %rsi, %rax
  ret

.L3:
  movq %rsi, %rax
  imulq %rdx, %rax
  ret

.L2:
  cmpq $2, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
  .L4:
  rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y;
        }
    } else if (_______) {
        val = __________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

# test:
leaq (%rdi,%rsi), %rax
addq %rdx, %rax
cmpq $.-3, %rdi
jge .L2
cmpq %rdx, %rsi
jge .L3
movq %rdi, %rax
imulq %rsi, %rax
ret

.L3:
movq %rsi, %rax
imulq %rdx, %rax
ret

.L2:
    cmpq $.2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
    ret

.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (_______) {
        val = _________;
    }
    return val;
}

# x in %rdi, y in %rsi, z in %rdx
# test:
leaq (%rdi,%rsi), %rax
addq %rdx, %rax
cmpq $-3, %rdi
jge .L2
cmpq %rdx, %rsi
jge .L3
movq %rdi, %rax
imulq %rsi, %rax
ret
.L3:
movq %rsi, %rax
imulq %rdx, %rax
ret
.L2:
cmpq $2, %rdi
jle .L4
movq %rdi, %rax
imulq %rdx, %rax
.L4:
rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (_______) {
        val = __________;
    }
    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (_______) {
        val = _________;
    }
    return val;
}
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = _________;
    }
    return val;
}

Practice: Reverse Engineering

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret

.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret

.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax

.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = ________;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
 .L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
 .L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
 .L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = x;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret

.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret

.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax

.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = x;
    }
    return val;
}

# x in %rdi, y in %rsi, z in %rdx
int test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = x * z;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
long test(long x, long y, long z) {
  long val = x + y + z;
  if (x < -3) {
    if (y < z) {
      val = x * y;
    } else {
      val = y * z;
    }
  } else if (x > 2) {
    val = x * z;
  }
  return val;
}

Practice: Reverse Engineering

# x in %rdi, y in %rsi, z in %rdx
test:
  leaq (%rdi,%rsi), %rax
  addq %rdx, %rax
  cmpq $-3, %rdi
  jge .L2
  cmpq %rdx, %rsi
  jge .L3
  movq %rdi, %rax
  imulq %rsi, %rax
  ret
.L3:
  movq %rsi, %rax
  imulq %rdx, %rax
  ret
.L2:
  cmpq $2, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
.L4:
  rep; ret
long test(long x, long y, long z) {
    long val = x + y + z;

    if (x < -3) {
        if (y < z) {
            val = x * y;
        } else {
            val = y * z;
        }
    } else if (x > 2) {
        val = x * z;
    }

    return val;
}

# x in %rdi, y in %rsi, z in %rdx

test:
leaq (%rdi,%rsi), %rax
addq %rdx, %rax
cmpq $-3, %rdi
jge .L2
cmpq %rdx, %rsi
jge .L3
movq %rdi, %rax
imulq %rsi, %rax
ret
.L3:
movq %rsi, %rax
imulq %rdx, %rax
ret
.L2:
cmpq $2, %rdi
jle .L4
movq %rdi, %rax
imulq %rdx, %rax
.L4:
rep; ret
Recap

• **Recap:** Assembly and *mov*
• Data and Register Sizes
• The *lea* Instruction
• Logical and Arithmetic Operations
• Control
  • Condition Codes
  • Assembly Instructions
• **Practice:** Reverse-Engineering

**Next time:** More control flow, and the runtime stack