CS107, Lecture 11
Assembly Continued

Reading: B&O 3.1-3.4
Lecture Plan

• Recap: mov so far 1
• Data and Register Sizes 11
• The lea Instruction 24
• Logical and Arithmetic Operations 30
• Practice: Reverse Engineering 38

See more guides on Resources page of course website!
The `mov` instruction copies bytes from one place to another; it is similar to the assignment operator (=) in C.

```
    mov src, dst
```

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

- Immediate (constant value, like a number) *(only src)*
- Register
- Memory Location *(at most one of src, dst)*
<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>
## 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>$r_i$</td>
<td>$R[r_i]$</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>$(r_i)$</td>
<td>$M[R[r_i]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^w)$</td>
<td>$M[Imm + R[r^w]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^w, r_h)$</td>
<td>$M[R[r^w] + R[r_h]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^w, r_h, s)$</td>
<td>$M[R[r_h] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^w, r_h, s)$</td>
<td>$M[Imm + R[r_h] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^w, r_h, s)$</td>
<td>$M[R[r^w] + R[r_h] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^w, r_h, s)$</td>
<td>$M[Imm + R[r^w] + R[r_h] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Figure 3.3 from the book:** “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.”
Lecture Plan

- Recap: mov so far
- Data and Register Sizes
- The lea Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

See more guides on Resources page of course website!
Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

• A byte is 1 byte.
• A word is 2 bytes.
• A double word is 4 bytes.
• A quad word is 8 bytes.

Assembly instructions can have suffixes to refer to these sizes:

• b means byte
• w means word
• l means double word
• q means quad word
## Register Sizes

<table>
<thead>
<tr>
<th>Bit</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit: 63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bp1</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%sp1</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit:</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></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 of the stack

See more guides on Resources page of course website!
mov Variants

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

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. movb, movw, movl or movq).

1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. movb, movw, movl or movq).

1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
• 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.

```assembly
movabsq $0x0011223344556677, %rax
```
movz and movs

• 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

MOVZ S,R  \( R \leftarrow \text{ZeroExtend}(S) \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movzbw</td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>movzbl</td>
<td>Move zero-extended byte to double word</td>
</tr>
<tr>
<td>movzwl</td>
<td>Move zero-extended word to double word</td>
</tr>
<tr>
<td>movzbq</td>
<td>Move zero-extended byte to quad word</td>
</tr>
<tr>
<td>movzwq</td>
<td>Move zero-extended word to quad word</td>
</tr>
</tbody>
</table>
# movz and movs

MOVS S,R  
\[ R \leftarrow \text{SignExtend}(S) \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsbw</td>
<td>Move sign-extended byte to word</td>
</tr>
<tr>
<td>movsbl</td>
<td>Move sign-extended byte to double word</td>
</tr>
<tr>
<td>movswl</td>
<td>Move sign-extended word to double word</td>
</tr>
<tr>
<td>movsbq</td>
<td>Move sign-extended byte to quad word</td>
</tr>
<tr>
<td>movswq</td>
<td>Move sign-extended word to quad word</td>
</tr>
<tr>
<td>movslq</td>
<td>Move sign-extended double word to quad word</td>
</tr>
</tbody>
</table>
| cltq        | Sign-extend %eax to %rax \%
|             | \%rax \leftarrow \text{SignExtend}(\%eax) |
Lecture Plan

• Recap: mov so far 7
• Data and Register Sizes 11
• The lea Instruction 24
• Logical and Arithmetic Operations 30
• Practice: Reverse Engineering 38

See more guides on Resources page of course website!
The **lea** instruction **copies** an “effective address” from one place to another.

```plaintext
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.

The syntax for the destinations is the same as **mov**. The difference is how it handles the src.
## lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6(%rax), %rdx)</td>
<td>Go to the address ((6 + \text{what’s in } %rax)), and copy data there into %rdx</td>
<td>Copy (6 + \text{what’s in } %rax) into %rdx.</td>
</tr>
</tbody>
</table>
## lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6(%rax), %rdx</td>
<td>Go to the address ($6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy $6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>$(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
</tbody>
</table>
# lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(%rax), %rdx</td>
<td>Go to the address (6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx, 4), %rdx</td>
<td>Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.</td>
<td>Copy (%rax + 4 * %rcx) into %rdx.</td>
</tr>
</tbody>
</table>
# lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(%rax), %rdx</td>
<td>Go to the address (6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx, 4), %rdx</td>
<td>Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.</td>
<td>Copy (%rax + 4 * %rcx) into %rdx.</td>
</tr>
<tr>
<td>7(%rax, %rax, 8), %rdx</td>
<td>Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.</td>
<td>Copy (7 + %rax + 8 * %rax) into %rdx.</td>
</tr>
</tbody>
</table>

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

• Recap: `mov` so far 7
• Data and Register Sizes 11
• The `lea` Instruction 24
• Logical and Arithmetic Operations 30
• Practice: Reverse Engineering 38

See more guides on Resources page of course website!
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 ← D + 1</td>
<td>Increment</td>
</tr>
<tr>
<td>dec D</td>
<td>D ← D - 1</td>
<td>Decrement</td>
</tr>
<tr>
<td>neg D</td>
<td>D ← -D</td>
<td>Negate</td>
</tr>
<tr>
<td>not D</td>
<td>D ← ~D</td>
<td>Complement</td>
</tr>
</tbody>
</table>

Examples:

- incq 16(%rax)
- dec %rdx
- not %rcx
Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). 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 ← D + S</td>
<td>Add</td>
</tr>
<tr>
<td>sub S, D</td>
<td>D ← D - S</td>
<td>Subtract</td>
</tr>
<tr>
<td>imul S, D</td>
<td>D ← D * S</td>
<td>Multiply</td>
</tr>
<tr>
<td>xor S, D</td>
<td>D ← D ^ S</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>or S, D</td>
<td>D ← D</td>
<td>S</td>
</tr>
<tr>
<td>and S, D</td>
<td>D ← 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.

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

- 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><code>R[%rdx]:R[%rax] \leftarrow S \times R[%rax]</code></td>
<td>Signed full multiply</td>
</tr>
<tr>
<td><code>mulq S</code></td>
<td><code>R[%rdx]:R[%rax] \leftarrow S \times R[%rax]</code></td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>
Division and Remainder

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

• **Terminology:** dividend / divisor = quotient + remainder

• **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 %rdx, and the low-order 64 bits are in %rax. The divisor is the operand to the instruction.

• The quotient is stored in %rax, and the remainder in %rdx.
# Division and Remainder

**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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>idivq S</code></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><code>divq S</code></td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Unsigned divide</td>
</tr>
<tr>
<td><code>cqto</code></td>
<td>R[%rdx]:R[%rax] ← SignExtend(R[%rax])</td>
<td>Convert to oct word</td>
</tr>
</tbody>
</table>
The following instructions have two operands: the shift amount $k$ and the destination to shift, $D$. $k$ 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)`
- `shrl %cl, (%rax,%rdx,8)`
- `sarh $4,8(%rax)`
Shift Amount

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sal k, D</td>
<td>D ← D &lt;&lt; k</td>
<td>Left shift</td>
</tr>
<tr>
<td>shl k, D</td>
<td>D ← D &lt;&lt; k</td>
<td>Left shift (same as sal)</td>
</tr>
<tr>
<td>sar k, D</td>
<td>D ← D &gt;&gt;ₐ k</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>shr k, D</td>
<td>D ← D &gt;&gt;ₗ k</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

• When using `%cl`, the width of what you are shifting determines what portion of `%cl` is used.

• 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. `shlw` shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

See more guides on Resources page of course website!
Assembly Exploration

• Let’s pull these commands together and see how some C code might be translated to assembly.
• Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let’s check it out!
• https://godbolt.org/z/WPzz6G4a9
What does it mean for a program to execute?
So far:
• Program values can be stored in memory or registers.
• Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
• Assembly instructions are also stored in memory.

So now:
• **Who controls the instructions?**
  How do we know what to do now or next?

Answer:
• The **program counter** (PC), %rip.
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 of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
Instructions Are Just Bytes!
Memory bus

Main memory

"hello, world\n
hello code"
Instructions Are Just Just Bytes!

Machine code instructions

0x0

Main Memory

Stack

Heap

Data

Text (code)
00000000004004ed <loop>:
4004ed:  55  push %rbp
4004ee: 48 89 e5  mov  %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl  $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01  addl  $0x1,-0x4(%rbp)
4004fc: eb fa  jmp  4004f8 <loop+0xb>

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed</td>
<td>55</td>
</tr>
<tr>
<td>4004ee</td>
<td>48 89 e5</td>
</tr>
<tr>
<td>4004f1</td>
<td>c7 45 fc 00 00 00 00</td>
</tr>
<tr>
<td>4004f8</td>
<td>83 45 fc 01</td>
</tr>
<tr>
<td>4004fc</td>
<td>eb fa</td>
</tr>
<tr>
<td>4004f9</td>
<td>45</td>
</tr>
<tr>
<td>4004f8</td>
<td>83</td>
</tr>
<tr>
<td>4004f7</td>
<td>00</td>
</tr>
<tr>
<td>4004f6</td>
<td>00</td>
</tr>
<tr>
<td>4004f5</td>
<td>00</td>
</tr>
<tr>
<td>4004f4</td>
<td>00</td>
</tr>
<tr>
<td>4004f3</td>
<td>fc</td>
</tr>
<tr>
<td>4004f2</td>
<td>45</td>
</tr>
<tr>
<td>4004f1</td>
<td>c7</td>
</tr>
<tr>
<td>4004f0</td>
<td>e5</td>
</tr>
<tr>
<td>4004ef</td>
<td>89</td>
</tr>
<tr>
<td>4004ee</td>
<td>48</td>
</tr>
<tr>
<td>4004ed</td>
<td>55</td>
</tr>
</tbody>
</table>

Main Memory:
- Stack
- Heap
- Data
- Text (code)
00000000004004ed <loop>:

4004ed: 55  push  %rbp
4004ee: 48 89 e5  mov  %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl  $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01  addl  $0x1,-0x4(%rbp)
4004fc: eb fa  jmp  4004f8 <loop+0xb>

The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the next *instruction* to be executed.
The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the **next instruction** to be executed.

The code snippet shown illustrates the use of the %rip register. The instructions:

- **4004ed**: `55` - push %rbp
- **4004ee**: `48 89 e5` - mov %rsp,%rbp
- **4004f1**: `c7 45 fc 00 00 00 00` - movl $0x0,-0x4(%rbp)
- **4004f8**: `83 45 fc 01` - addl $0x1,-0x4(%rbp)
- **4004fc**: `eb fa` - jmp 4004f8 <loop+0xb>
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the **next instruction** to be executed.
Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

0x4004fc

%rip
Going In Circles

• How can we use this representation of execution to represent e.g. a loop?
• Key Idea: we can ”interfere” with %rip and set it back to an earlier instruction!
The `jmp` instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).
Jump!

The jmp instruction is an unconditional jump that sets the program counter to the jump target (the operand).

```
00000000004004ed <loop>:
4004ed:  55  push  %rbp
4004ee: 48 89 e5  mov  %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl  $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01  addl  $0x1,-0x4(%rbp)
4004fc: eb fa  jmp  4004f8 <loop+0xb>
```

%rip

0x4004fc
The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).
The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

```
0x4004fc
%rip
```

\[ \text{0x4004fc} \]
This assembly represents an infinite loop in C!

```c
while (true) {...}
```
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 one of the usual operand forms (indirect jump):

```
jmp *%rax     # jump to instruction at address in %rax
```
“Interfering” with %rip

1. How do we repeat instructions in a loop?

jmp [target]
• A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a conditional jump?
We’ll leave that for next time....
// 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;
}

------

add_to_first:
    movl %edi, %eax
    addl (%rsi), %eax
    ret
// 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;
}

full_divide:
    movq %rdi, %rax
    movq %rdx, %rcx
    cqto
    idivq %rsi
    movq %rdx, (%rcx)
    ret
Assembly Exercise 1

0000000000040116e <sum_example1>:
  40116e: 8d 04 37 lea (%rdi,%rsi,1),%eax
  401171: 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;
}
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
// B)
int sum_example1(int x, int y) {
    return x + y;
}
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?
Assembly Exercise 3

00000000000401172 <sum_example2>:
  401172: 8b 47 0c           mov 0xc(%rdi),%eax
  401175: 03 07           add (%rdi),%eax
  401177: 2b 47 18        sub 0x18(%rdi),%eax
  40117a: 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 code above represents the C code’s 6 (as in arr[6])?
```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

---

**Our First Assembly**

We’re 1/2 of the way to understanding assembly! What looks understandable right now?

0000000000401136 <sum_array>:

```
401136:   b8 00 00 00 00 00  
40113b:   ba 00 00 00 00 00  
401140:   39 f0              mov $0x0,%eax
401142:   7d 0b              mov $0x0,%edx
401144:   48 63 c8           cmp %esi,%eax
401147:   03 14 8f           jge 40114f <sum_array+0x19>
40114a:   83 c0 01           movslq %eax,%rcx
40114d:   eb f1              add (%rdi,%rcx,4),%edx
40114f:   89 d0              add $0x1,%eax
401151:   c3                jmp 401140 <sum_array+0xa>
```

00000000000000401136 <sum_array>:
**A Note About Operand Forms**

- Many instructions share the same address operand forms that `mov` uses.
  - Eg. `7(%rax, %rcx, 2)`.  

- These forms work the same way for other instructions, e.g. `sub`:
  - `sub 8(%rax,%rdx),%rcx` -> Go to 8 + %rax + %rdx, subtract what’s there from %rcx

- The exception is **lea**:
  - It interprets this form as just the calculation, *not the dereferencing*
  - `lea 8(%rax,%rdx),%rcx` -> Calculate 8 + %rax + %rdx, put it in %rcx
When using \%cl, the width of what you are shifting determines what portion of \%cl is used.

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: \texttt{shlb} shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. \texttt{shlw} shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.
Division and Remainder

- **Terminology:** \( \text{dividend} / \text{divisor} = \text{quotient} + \text{remainder} \)
- **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 `%rdx`, and the low-order 64 bits are in `%rax`. The divisor is the operand to the instruction.
- The quotient is stored in `%rax`, and the remainder in `%rdx`.

### Instruction Effect Description

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
Extra Practice

https://godbolt.org/z/hGKPWszq4
int add_to(int x, int arr[], int i)
{
    int sum = ___;
    sum += arr[___];
    return ___;
}

---------

add_to:
movslq %edx, %rdx
movl %edi, %eax
addl (%rsi,%rdx,4), %eax
ret
int add_to(int x, int arr[], int i)
    {
        int sum = ___;
        sum += arr[___];
        return ___;
    }

--------
// x in %edi, arr in %rsi, i in %edx
add_to:
    movslq %edx, %rdx   // sign-extend i into full register
    movl %edi, %eax    // copy x into %eax
    addl (%rsi, %rdx, 4), %eax    // add arr[i] to %eax
    ret
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

// x in %edi, arr in %rsi, i in %edx add_to:
    movslq %edx, %rdx    // sign-extend i into full register
    movl %edi, %eax     // copy x into %eax
    addl (%rsi,%rdx,4), %eax  // add arr[i] to %eax
    ret
int elem_arithmetic(int nums[], int y)
{
    int z = nums[___] * ___;
    z -= ___;
    z >>= ___;
    return ___;
}

elem_arithmetic:
    movl %esi, %eax
    imull (%rdi), %eax
    subl 4(%rdi), %eax
    sarl $2, %eax
    addl $2, %eax
    ret
int elem_arithmetic(int nums[], int y)
{
    int z = nums[__?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax // copy y into %eax
    imull (%rdi), %eax // multiply %eax by nums[0]
    subl 4(%rdi), %eax // subtract nums[1] from %eax
    sarl $2, %eax // shift %eax right by 2
    addl $2, %eax // add 2 to %eax
    ret
int elem_arithmetic(int nums[], int y)
{
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax          // copy y into %eax
    imull (%rdi), %eax       // multiply %eax by nums[0]
    subl 4(%rdi), %eax       // subtract nums[1] from %eax
    sarl $2, %eax            // shift %eax right by 2
    addl $2, %eax            // add 2 to %eax
    ret
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}

---------

func:
    movq %rdi, %rax
    leaq 1(%rdi), %rcx
    movq %rcx, (%rsi)
    cqto
    idivq %rcx
    movq %rdx, %rax
    ret
long func(long x, long *ptr) {
    *ptr = ____ + 1;
    long result = x % ____;
    return ____;
}

// x in %rdi, ptr in %rsi
func:
    movq %rdi, %rax          // copy x into %rax
    leaq 1(%rdi), %rcx      // put x + 1 into %rcx
    movq %rcx, (%rsi)       // copy %rcx into *ptr
    cqto                      // sign-extend x into %rdx
    idivq %rcx               // calculate x / (x + 1)
    movq %rdx, %rax          // copy the remainder into %rax
    ret
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}

// x in %rdi, ptr in %rsi

func:
    movq %rdi, %rax       // copy x into %rax
    leaq 1(%rdi), %rcx    // put x + 1 into %rcx
    movq %rcx, (%rsi)     // copy %rcx into *ptr
    cqto                 // sign-extend x into %rdx
    idivq %rcx           // calculate x / (x + 1)
    movq %rdx, %rax      // copy the remainder into %rax
    ret
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}

// x in %rdi, ptr in %rsi
func:
    leaq 1(%rdi), %rcx          // put x + 1 into %rcx
    movq %rcx, (%rsi)           // copy %rcx into *ptr
    movq %rdi, %rax             // copy x into %rax
    cqto                          // sign-extend x into %rdx
    idivq %rcx                   // calculate x / (x + 1)
    movq %rdx, %rax              // copy the remainder into %rax
    ret