Lecture #20 – Arithmetic and Logic Operations
## Recap: 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>$\text{Imm}$</td>
<td>$\text{Imm}$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$\text{R}[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}$</td>
<td>$\text{M}[\text{Imm}]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_a)$</td>
<td>$\text{M}[\text{R}[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b, r_i)$</td>
<td>$\text{M}[\text{R}[r_b] + \text{R}[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b, r_i)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b] + \text{R}[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(s, r_i)$</td>
<td>$\text{M}[\text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(s, r_i)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b, r_i, s)$</td>
<td>$\text{M}[\text{R}[r_b] + \text{R}[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$\text{Imm}(r_b, r_i, s)$</td>
<td>$\text{M}[\text{Imm} + \text{R}[r_b] + \text{R}[r_i] \cdot s]$</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.”
Recap: 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.

<table>
<thead>
<tr>
<th>C Type</th>
<th>Suffix</th>
<th>Byte</th>
<th>Intel Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
<td>Word</td>
</tr>
<tr>
<td>int</td>
<td>l</td>
<td>4</td>
<td>Double word</td>
</tr>
<tr>
<td>long</td>
<td>q</td>
<td>8</td>
<td>Quad word</td>
</tr>
<tr>
<td>char *</td>
<td>q</td>
<td>8</td>
<td>Quad word</td>
</tr>
<tr>
<td>float</td>
<td>s</td>
<td>4</td>
<td>Single precision</td>
</tr>
<tr>
<td>double</td>
<td>l</td>
<td>8</td>
<td>Double precision</td>
</tr>
</tbody>
</table>
Recap: \texttt{mov} Variants

- \texttt{mov} can take an optional suffix (\texttt{b,w,l,q}) that specifies the size of data to move: \texttt{movb, movw, movl, movq}
- \texttt{mov} only updates the specific register bytes or memory locations indicated.
  - Exception: \texttt{movl} writing to a register will also set high order 4 bytes to 0.
### 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>movzw1</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

<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>movswq</td>
<td>Move sign-extended double word to quad word</td>
</tr>
</tbody>
</table>
| cltq        | Sign-extend %eax to %rax 
|             | %rax ← SignExtend(%eax) |

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

Moving data around

Arithmetic and logical operations

Control flow

Function calls

Lecture 19

This Lecture

Lecture 21-23

Lecture 24-26
Learning Goals

• Learn how to perform arithmetic and logical operations in assembly
• Begin to learn how to read assembly and understand the C code that generated it
Plan for Today

• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Helpful Assembly Resources

• Course textbook
  Reminder: see relevant readings for each lecture on the Schedule section: https://aykuterdem.github.io/classes/comp201/index.html#div_schedule

• Other resources
  See the guides on the resources section of the course website: https://aykuterdem.github.io/classes/comp201/index.html#div_resources
  - Stanford CS107 Assembly Reference Sheet
  - Stanford CS107 Guide to x86-64
  - CMU 15-213 x86-64 Machine-Level Programming
Lecture Plan

• The lea Instruction
• Logical and Arithmetic Operations
• 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.

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 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + 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

• The 

• Logical and Arithmetic Operations
• 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 ← 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.

  
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]</code> ← <code>S</code> x <code>R[rax]</code></td>
<td>Signed full multiply</td>
</tr>
<tr>
<td><code>mulq S</code></td>
<td><code>R[rdx]:R[rax]</code> ← <code>S</code> x <code>R[rax]</code></td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>
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 \( \%rdx \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>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>
## 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>
Shift Instructions

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 $3,(%rax)$
            shr1 %cl,(%rax,%rdx,8)
            sarl $4,8(%rax)$
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 (0b11111111)`, 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

• The 1ea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering
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/NLYhVf
Code Reference: add_to_first

// 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
Code Reference: full Divide

// 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 %rdx, %rcx
  movq %rdi, %rax
  cqto
  idivq %rsi
  movq %rdx, (%rcx)
  ret

<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>
<tr>
<td>cqto</td>
<td>R[%rdx]:R[%rax] ← SignExtend(R[%rax])</td>
<td>Convert to oct word</td>
</tr>
</tbody>
</table>
Assembly Exercise 1

00000000004005ac <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

0000000000400578 <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

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

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

```c
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

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

https://godbolt.org/z/QQj77g
Reverse Engineering 1

int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------

add_to_ith:
    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_ith:
    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

36
Reverse Engineering 1

```c
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}
```

----------

```assembly
// x in %edi, arr in %rsi, i in %edx
add_to_ith:
    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
Reverse Engineering 2

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
Reverse Engineering 2

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:
    leaq 1(%rdi), %rcx
    movq %rcx, (%rsi)
    movq %rdi, %rax
    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:
    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
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

Recap

• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Next Time: control flow in assembly (while loops, if statements, and more)