photo: Intel CPU Wafers

# COMP201

# Computer Systems & Programming

## Lecture #20 – Arithmetic and Logic Operations

---

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

# Recap: Operand Forms

| Type | Form | Operand Value | Name |
|---|---|---|---|
| Immediate | $\$Imm$ | $Imm$ | Immediate |
| Register | $r_a$ | $R[r_a]$ | Register |
| Memory | $Imm$ | $M[Imm]$ | Absolute |
| Memory | $(r_a)$ | $M[R[r_a]]$ | Indirect |
| Memory | $Imm(r_b)$ | $M[Imm + R[r_b]]$ | Base + displacement |
| Memory | $(r_b, r_i)$ | $M[R[r_b] + R[r_i]]$ | Indexed |
| Memory | $Imm(r_b, r_i)$ | $M[Imm + R[r_b] + R[r_i]]$ | Indexed |
| Memory | $(, r_i, s)$ | $M[R[r_i] \cdot s]$ | Scaled indexed |
| Memory | $Imm(, r_i, s)$ | $M[Imm + R[r_i] \cdot s]$ | Scaled indexed |
| Memory | $(r_b, r_i, s)$ | $M[R[r_b] + R[r_i] \cdot s]$ | Scaled indexed |
| Memory | $Imm(r_b, r_i, s)$ | $M[Imm + R[r_b] + R[r_i] \cdot s]$ | Scaled indexed |

**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.

| C Type | Suffix | Byte | Intel Data Type |
|---|---|---|---|
| `char` | `b` | 1 | Byte |
| `short` | `w` | 2 | Word |
| `int` | `l` | 4 | Double word |
| `long` | `q` | 8 | Quad word |
| `char *` | `q` | 8 | Quad word |
| `float` | `s` | 4 | Single precision |
| `double` | `l` | 8 | Double precision |

# Recap: `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.

# `movz` and `movs`

```
MOVZ S,R        R ← ZeroExtend(S)
```

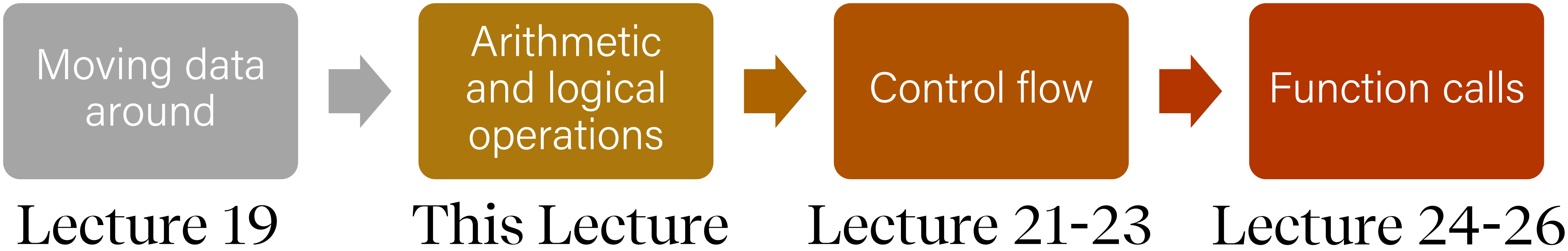
| Instruction | Description |
| --- | --- |
| `movzbw` | Move zero-extended byte to word |
| `movzbl` | Move zero-extended byte to double word |
| `movzwl` | Move zero-extended word to double word |
| `movzbq` | Move zero-extended byte to quad word |
| `movzwq` | Move zero-extended word to quad word |

# `movz` and `movs`

`MOVS S,R` `R ← SignExtend(S)`

| Instruction | Description |
|---|---|
| `movsbw` | Move sign-extended byte to word |
| `movsbl` | Move sign-extended byte to double word |
| `movswl` | Move sign-extended word to double word |
| `movsbq` | Move sign-extended byte to quad word |
| `movswq` | Move sign-extended word to quad word |
| `movslq` | Move sign-extended double word to quad word |
| `cltq` | Sign-extend `%eax` to `%rax`<br>`%rax ← SignExtend(%eax)` |

# Learning Assembly



# 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

# `lea`

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`

| Operands | mov Interpretation | lea Interpretation |
|---|---|---|
| **`6(%rax), %rdx`** | Go to the address (6 + what's in `%rax`), and copy data there into `%rdx` | Copy 6 + what's in `%rax` into `%rdx`. |

# lea vs. mov

| Operands | mov Interpretation | lea Interpretation |
|---|---|---|
| `6(%rax), %rdx` | Go to the address (6 + what's in `%rax`), and copy data there into `%rdx` | Copy 6 + what's in `%rax` into `%rdx`. |
| `(%rax, %rcx), %rdx` | Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx` | Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`. |

# `lea` vs. `mov`

| Operands | mov Interpretation | lea Interpretation |
|---|---|---|
| `6(%rax), %rdx` | Go to the address (6 + what's in `%rax`), and copy data there into `%rdx` | Copy 6 + what's in `%rax` into `%rdx`. |
| `(%rax, %rcx), %rdx` | Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx` | Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`. |
| `(%rax, %rcx, 4), %rdx` | Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`. | Copy (`%rax` + 4 * `%rcx`) into `%rdx`. |

# `lea` vs. `mov`

| Operands | mov Interpretation | lea Interpretation |
|---|---|---|
| `6(%rax), %rdx` | Go to the address (6 + what's in `%rax`), and copy data there into `%rdx` | Copy 6 + what's in `%rax` into `%rdx`. |
| `(%rax, %rcx), %rdx` | Go to the address (what's in `%rax` + what's in `%rcx`) and copy data there into `%rdx` | Copy (what's in `%rax` + what's in `%rcx`) into `%rdx`. |
| `(%rax, %rcx, 4), %rdx` | Go to the address (`%rax` + 4 * `%rcx`) and copy data there into `%rdx`. | Copy (`%rax` + 4 * `%rcx`) into `%rdx`. |
| `7(%rax, %rax, 8), %rdx` | Go to the address (7 + `%rax` + 8 * `%rax`) and copy data there into `%rdx`. | Copy (7 + `%rax` + 8 * `%rax`) into `%rdx`. |

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 `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

# Unary Instructions

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

| Instruction | Effect | Description |
|---|---|---|
| `inc D` | `D ← D + 1` | Increment |
| `dec D` | `D ← D - 1` | Decrement |
| `neg D` | `D ← -D` | Negate |
| `not D` | `D ← ~D` | Complement |

**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”:

| Instruction | Effect | Description |
|---|---|---|
| `add S, D` | `D ← D + S` | Add |
| `sub S, D` | `D ← D - S` | Subtract |
| `imul S, D` | `D ← D * S` | Multiply |
| `xor S, D` | `D ← D ^ S` | Exclusive-or |
| `or S, D` | `D ← D | S` | Or |
| `and S, D` | `D ← D & S` | And |

**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      D ← D * 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`**.

| Instruction | Effect | Description |
|---|---|---|
| `imulq S` | `R[%rdx]:R[%rax] ← S x R[%rax]` | Signed full multiply |
| `mulq S` | `R[%rdx]:R[%rax] ← S x R[%rax]` | Unsigned full multiply |

# Division and Remainder

| Instruction | Effect | Description |
|---|---|---|
| `idivq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Signed divide |
| `divq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Unsigned divide |

- 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

| Instruction | Effect | Description |
|---|---|---|
| `idivq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Signed divide |
| `divq S` | `R[%rdx] ← R[%rdx]:R[%rax] mod S;`<br>`R[%rax] ← R[%rdx]:R[%rax] ÷ S` | Unsigned divide |
| `cqto` | `R[%rdx]:R[%rax] ← SignExtend(R[%rax])` | Convert to oct word |

- 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 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!)

| Instruction | Effect | Description |
|---|---|---|
| `sal k, D` | D ← D << k | Left shift |
| `shl k, D` | D ← D << k | Left shift (same as `sal`) |
| `sar k, D` | $D \leftarrow D >>_A k$ | Arithmetic right shift |
| `shr k, D` | $D \leftarrow D >>_L k$ | Logical right shift |

**Examples**:
```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

# Shift Amount

| Instruction | Effect | Description |
|---|---|---|
| `sal k, D` | $D \leftarrow D << k$ | Left shift |
| `shl k, D` | $D \leftarrow D << k$ | Left shift (same as `sal`) |
| `sar k, D` | $D \leftarrow D >>_A k$ | Arithmetic right shift |
| `shr k, D` | $D \leftarrow D >>_L k$ | Logical right shift |

- 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 **`log2(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 log2(8) = 3 bits, which represent 7. **`shlw`** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

# Lecture Plan

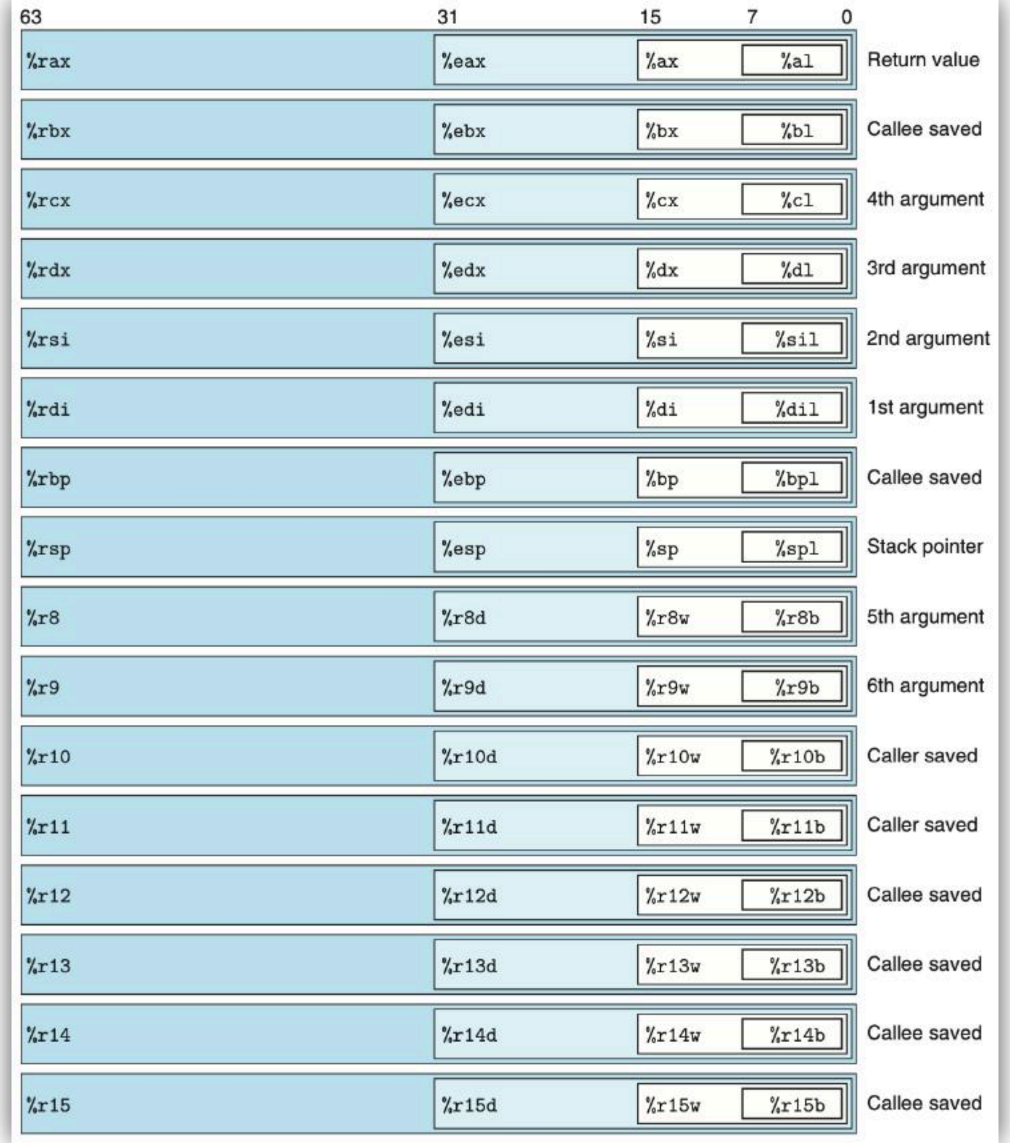
- The `lea` Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering

# 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

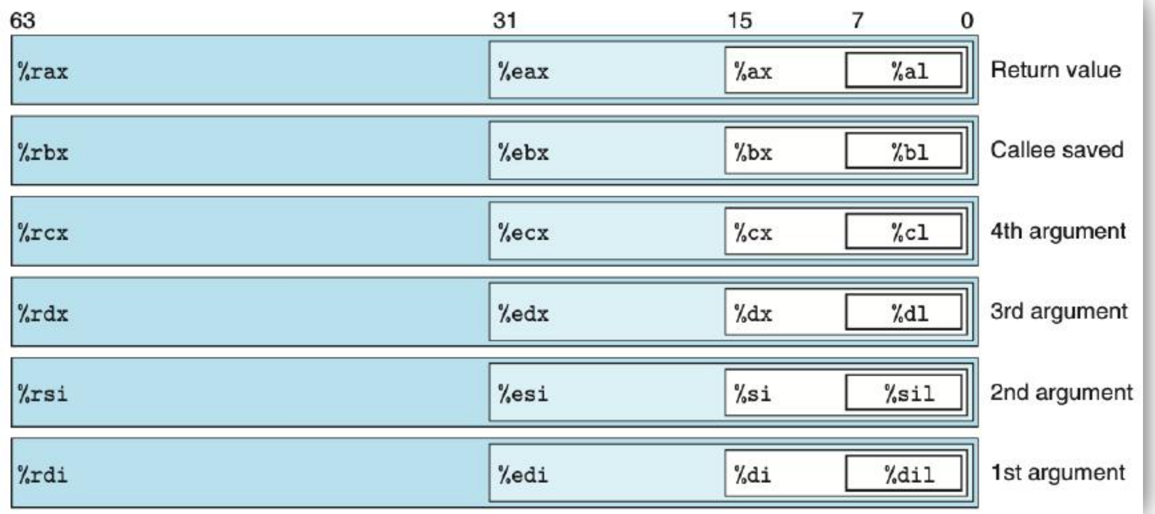
# Code Reference: add_to_first

```
// 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
```

| Instruction | Effect | Description |
|---|---|---|
| idivq S | R[%rdx] ← R[%rdx]:R[%rax] mod S;<br>R[%rax] ← R[%rdx]:R[%rax] ÷ S | Signed divide |
| divq S | R[%rdx] ← R[%rdx]:R[%rax] mod S;<br>R[%rax] ← R[%rdx]:R[%rax] ÷ S | Unsigned divide |
| cqto | R[%rdx]:R[%rax] ← SignExtend(R[%rax]) | Convert to oct word |

# Assembly Exercise 1

slido Event code: 73165

```
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

slido Event code: 73165

```
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
```

```
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?

**%eax**

# Assembly Exercise 3

**slido** Event code: 73165

```
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
```

```
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

```
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?**

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00          mov    $0x0,%edx
  4005bb:    b8 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
```

# 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
```

# Reverse Engineering 1

```
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
```

# Reverse Engineering 1

```
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_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
```

# Reverse Engineering 2

```
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
```

# Reverse Engineering 3

```
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
```

# Reverse Engineering 3

```
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
```

# Reverse Engineering 3

```
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)