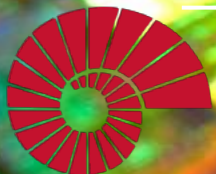


# COMP201

## Computer Systems & Programming

Lecture #15 – Arithmetic and Logic Operations

---



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Fall 2023

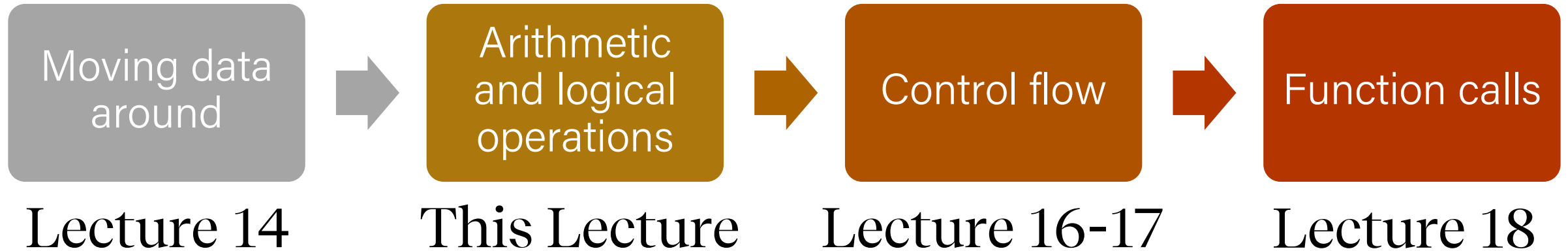
# Good news, everyone!

- No lab this week!



COMP201 Topic 6: How does  
a computer interpret and  
execute C programs?

# 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

- **Recap:** mov so far
- Data and Register Sizes
- 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](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](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

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



# mov

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*)

# Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no \$)
(%rax)	What's in %rax
4(%rax)	What's in %rax, plus 4
(%rax, %rdx)	Sum of what's in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	What's in %rcx, times 4 (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx
8(%rax, %rcx, 2)	What's in %rax, plus 2 times what's in %rcx, plus 8

# Operand Forms

Type	Form	Operand Value	Name
Immediate	\$Imm	Imm	Immediate
Register	$r_a$	$R[r_a]$	Register
Memory	Imm	$M[\text{Imm}]$	Absolute
Memory	$(r_a)$	$M[R[r_a]]$	Indirect
Memory	$\text{Imm}(r_b)$	$M[\text{Imm} + R[r_b]]$	Base + displacement
Memory	$(r_b, r_i)$	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$\text{Imm}(r_b, r_i)$	$M[\text{Imm} + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$\text{Imm}(, r_i, s)$	$M[\text{Imm} + R[r_i] \cdot s]$	Scaled indexed
Memory	$(r_b, r_i, s)$	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$\text{Imm}(r_b, r_i, s)$	$M[\text{Imm} + R[r_b] + R[r_i] \cdot s]$	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: 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 1/4<sup>th</sup> of the way to understanding assembly!  
**What looks understandable right now?**

Some notes:

- Registers store addresses and values
- `mov src, dst` ***copies*** value into `dst`
- `sizeof(int)` is 4
- Instructions executed sequentially

**0000000004005b6 <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	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq



# Practice 1

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
```

```
...
```

```
long num = _____???
```

---

```
// %rdi stores arr, %rcx stores 3, and %rax stores num  
mov (%rdi, %rcx, 8),%rax
```

# Practice 1

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
```

```
...
```

```
long num = arr[3];
```

---

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
```

```
mov (%rdi, %rcx, 8),%rax
```

# Practice 2

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
```

```
int *ptr = malloc(...);
```

```
___???___ = x;
```

---

```
// %ecx stores x, %rax stores ptr
```

```
mov %ecx, (%rax)
```

# Practice 2

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...  
int *ptr = malloc(...);  
*ptr = x;
```

---

```
// %ecx stores x, %rax stores ptr  
mov %ecx, (%rax)
```



# Practice 3

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
```

```
...
```

```
___???___ = 'c';
```

---

```
// %rcx stores str, %rdx stores 2
```

```
mov $0x63, (%rcx,%rdx,1)
```

# Practice 3

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];  
...  
str[2] = 'c';
```

---

```
// %rcx stores str, %rdx stores 2  
mov $0x63, (%rcx,%rdx,1)
```

# Lecture Plan

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

# 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**
- **d** means **double word**
- **q** means **quad word**

# 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.

<b>C Type</b>	<b>Suffix</b>	<b>Byte</b>	<b>Intel Data Type</b>
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

# Register Sizes

Bit: 63

31

15

7

0

<code>%rax</code>	<code>%eax</code>	<code>%ax</code>	<code>%al</code>
<code>%rbx</code>	<code>%ebx</code>	<code>%bx</code>	<code>%bl</code>
<code>%rcx</code>	<code>%ecx</code>	<code>%cx</code>	<code>%cl</code>
<code>%rdx</code>	<code>%edx</code>	<code>%dx</code>	<code>%dl</code>
<code>%rsi</code>	<code>%esi</code>	<code>%si</code>	<code>%sil</code>
<code>%rdi</code>	<code>%edi</code>	<code>%di</code>	<code>%dil</code>

# Register Sizes

Bit: 63	31	15	7	0
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	
%r8	%r8d	%r8w	%r8b	
%r9	%r9d	%r9w	%r9b	
%r10	%r10d	%r10w	%r10b	
%r11	%r11d	%r11w	%r11b	

# Register Sizes

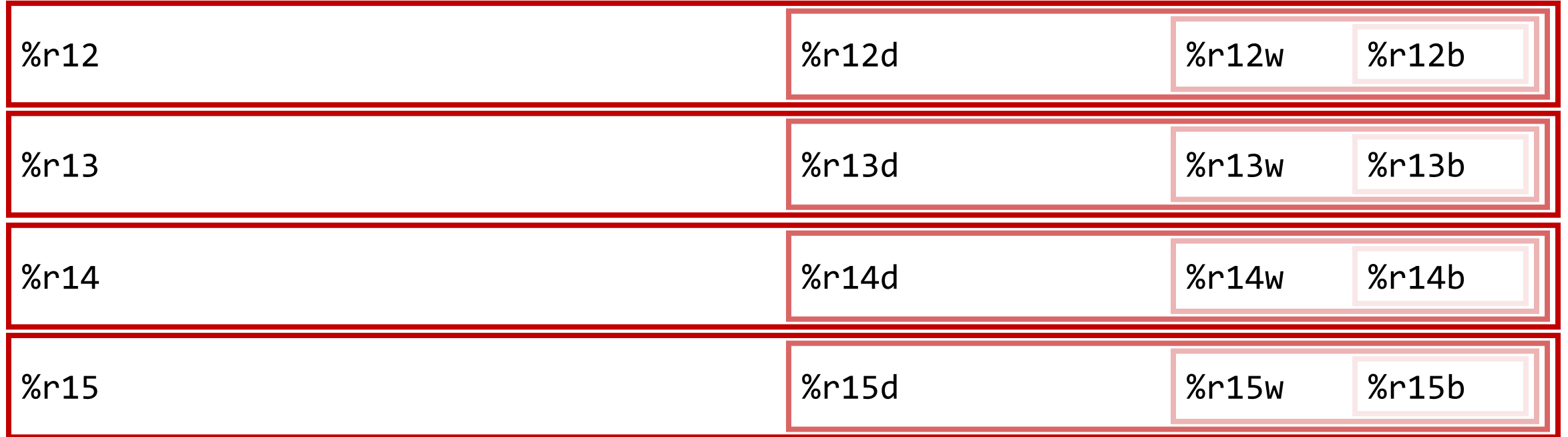
Bit: 63

31

15

7

0





# 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 **Stanford CS107 x86-64 Reference Sheet** on Resources page of the course website!  
[https://aykuterdem.github.io/classes/comp201/index.html#div\\_resources](https://aykuterdem.github.io/classes/comp201/index.html#div_resources)

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

# mov

- 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.

```
movabsq $0x0011223344556677, %rax
```

# Practice #2: mov And Data Sizes

For each of the following mov instructions, determine how data movement instructions modify the upper bytes of a destination register.

1. `movabs $0x0011223344556677, %rax`    `%rax = 0011223344556677`
2. `movb $-1, %al`    `%rax = 00112233445566FF`
3. `movw $-1, %ax`    `%rax = 001122334455FFFF`
4. `movl $-1, %eax`    `%rax = 00000000FFFFFFFF`
5. `movq $-1, %rax`    `%rax = FFFFFFFFFFFFFFFFFF`

# 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

$R \leftarrow \text{ZeroExtend}(S)$

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

$R \leftarrow \text{SignExtend}(S)$

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
cvtq	Sign-extend %eax to %rax $\%rax \leftarrow \text{SignExtend}(\%eax)$



# Lecture Plan

- Recap: mov so far
- Data and Register Sizes
- 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
<b>6(%rax), %rdx</b>	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
<b>6(%rax), %rdx</b>	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
<b>(%rax, %rcx), %rdx</b>	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
<b>6(%rax), %rdx</b>	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
<b>(%rax, %rcx), %rdx</b>	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.
<b>(%rax, %rcx, 4), %rdx</b>	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
<b>6(%rax), %rdx</b>	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
<b>(%rax, %rcx), %rdx</b>	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.
<b>(%rax, %rcx, 4), %rdx</b>	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
<b>7(%rax, %rax, 8), %rdx</b>	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

- Recap: mov so far
- Data and Register Sizes
- 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
<code>inc D</code>	$D \leftarrow D + 1$	Increment
<code>dec D</code>	$D \leftarrow D - 1$	Decrement
<code>neg D</code>	$D \leftarrow -D$	Negate
<code>not D</code>	$D \leftarrow \sim D$	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	$D \leftarrow D + S$	Add
sub S, D	$D \leftarrow D - S$	Subtract
imul S, D	$D \leftarrow D * S$	Multiply
xor S, D	$D \leftarrow D \wedge S$	Exclusive-or
or S, D	$D \leftarrow D   S$	Or
and S, D	$D \leftarrow D \& S$	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`       $D \leftarrow D * S$

- 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
<code>imulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply

# Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	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
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide
<code>cqto</code>	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	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 **%c1** (and only that register!)

Instruction	Effect	Description
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code> )
<code>sar k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

**Examples:** `shll $3, (%rax)`  
`shr1 %c1, (%rax, %rdx, 8)`  
`sar1 $4, 8(%rax)`

# Shift Amount

Instruction	Effect	Description
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code> )
<code>sar k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

- When using **%c1**, the width of what you are shifting determines what portion of **%c1** is used.
- For **w** bits of data, it looks at the low-order **log<sub>2</sub>(w)** bits of **%c1** to know how much to shift.
  - If **%c1** = 0xff (0b11111111), then: **sh1b** shifts by 7 because it considers only the low-order  $\log_2(8) = 3$  bits, which represent 7. **sh1w** 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

# 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
```

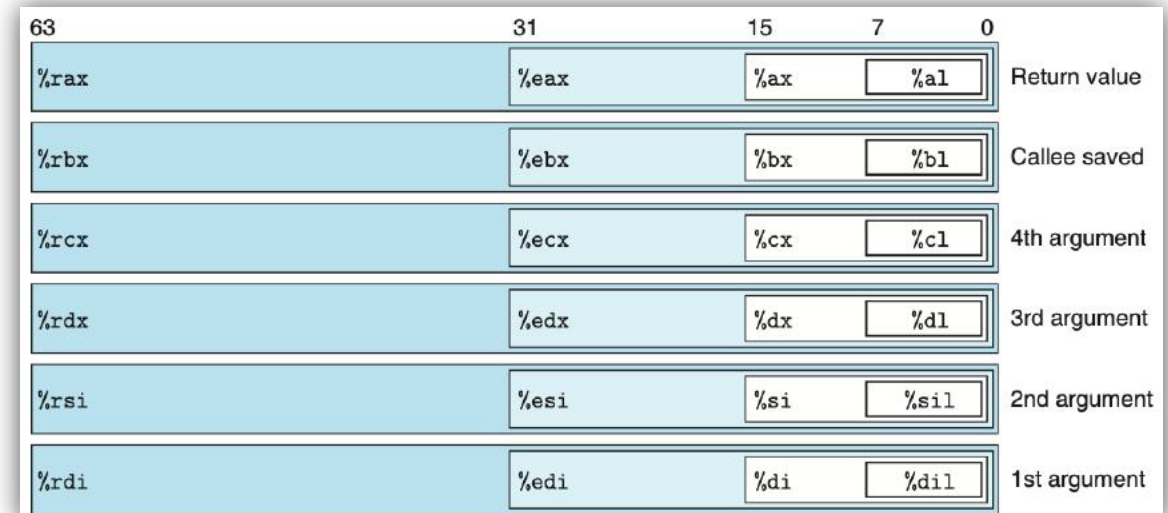
63	31	15	7	0	
%rax	%eax	%ax	%al		Return value
%rbx	%ebx	%bx	%bl		Callee saved
%rcx	%ecx	%cx	%cl		4th argument
%rdx	%edx	%dx	%dl		3rd argument
%rsi	%esi	%si	%sil		2nd argument
%rdi	%edi	%di	%dil		1st argument
%rbp	%ebp	%bp	%bpl		Callee saved
%rsp	%esp	%sp	%spl		Stack pointer
%r8	%r8d	%r8w	%r8b		5th argument
%r9	%r9d	%r9w	%r9b		6th argument
%r10	%r10d	%r10w	%r10b		Caller saved
%r11	%r11d	%r11w	%r11b		Caller saved
%r12	%r12d	%r12w	%r12b		Callee saved
%r13	%r13d	%r13w	%r13b		Callee saved
%r14	%r14d	%r14w	%r14b		Callee saved
%r15	%r15d	%r15w	%r15b		Callee saved

# 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	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
divq S	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide
cqto	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word

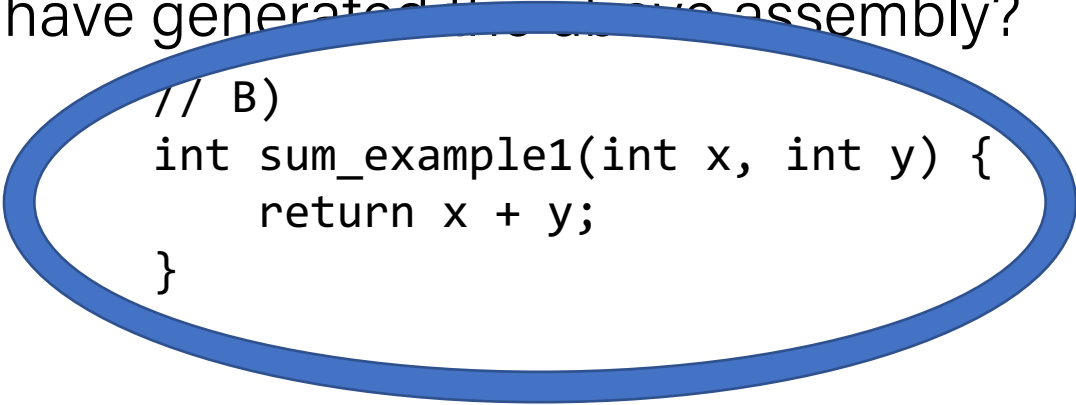
# Assembly Exercise 1

```
000000004005ac <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;  
}
```

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



```
// B)  
int sum_example1(int x, int y) {  
    return 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
```

```
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

```
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	j1	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.
  - E.g. `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`

# Recap

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

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