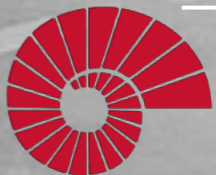


# COMP201

## Computer Systems & Programming

Lecture #16 – x86-64 Condition Codes & Control Flow



KOÇ  
UNIVERSITY

Aykut Erdem // Koç University // Spring 2026

# Recap

- The `leaq` Instruction
- Logical and Arithmetic Operations

# Recap: `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`.

# Recap: 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`

# Recap: 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)
```

# Recap: 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.

$$\text{imul } S, D \quad D \leftarrow D * S$$

- 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

# Recap: 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**.

# Recap: 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.

# Recap: 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)`  
`sarl $4, 8(%rax)`

# Recap: 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.

# Recap: 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`

# Plan for Today

- Practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics

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

# Lecture Plan

- Practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics

# Reverse Engineering Practices

<https://godbolt.org/z/qqj77g>

# Reverse Engineering 1

```
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
```

# 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:
```

```
    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:
```

```
    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
```

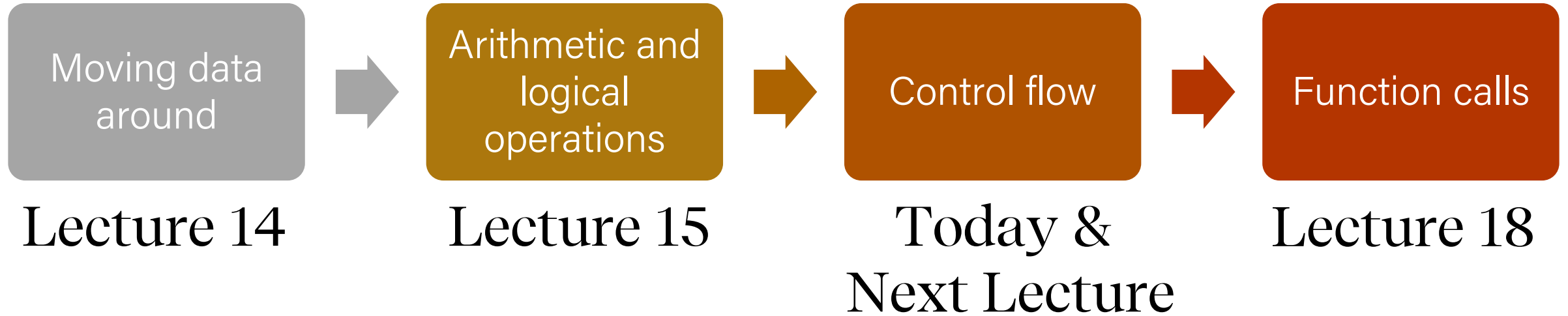
```
func:
```

```
    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
```

# Lecture Plan

- More practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics

# Learning Assembly



# Learning Goals

- Learn about how assembly stores comparison and operation results in condition codes
- Understand how assembly implements loops and control flow

# Executing Instructions

What does it mean for a program  
to execute?

# Executing Instructions

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.


Today:

- **Who controls the instructions?**  
How do we know what to do now or next?

Answer:

- The **program counter** (PC), `%rip`.

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



# 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 the x86-64 Guide and Reference Sheet on the Resources webpage for more!

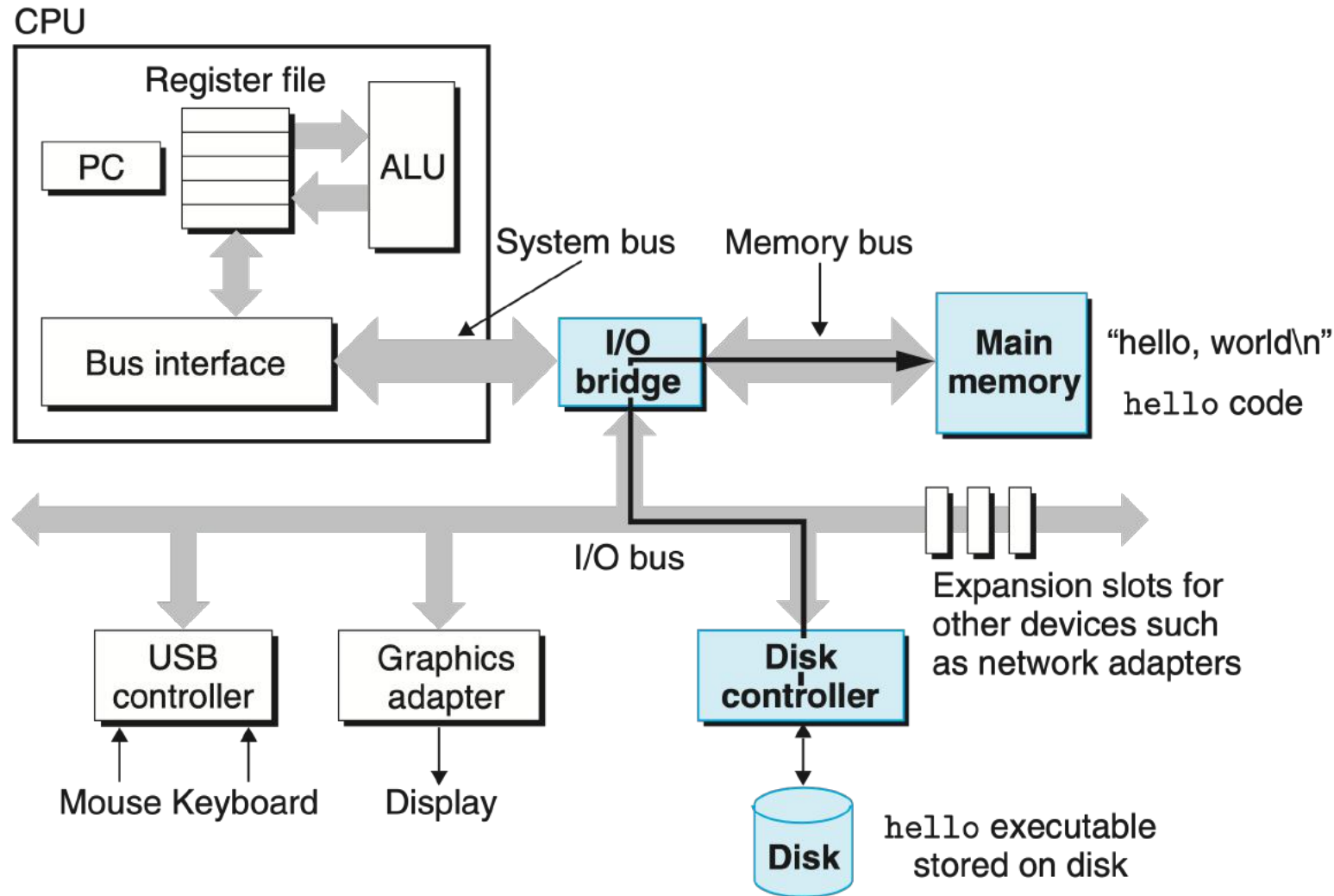
# Register Responsibilities

Some registers take on special responsibilities during program execution.

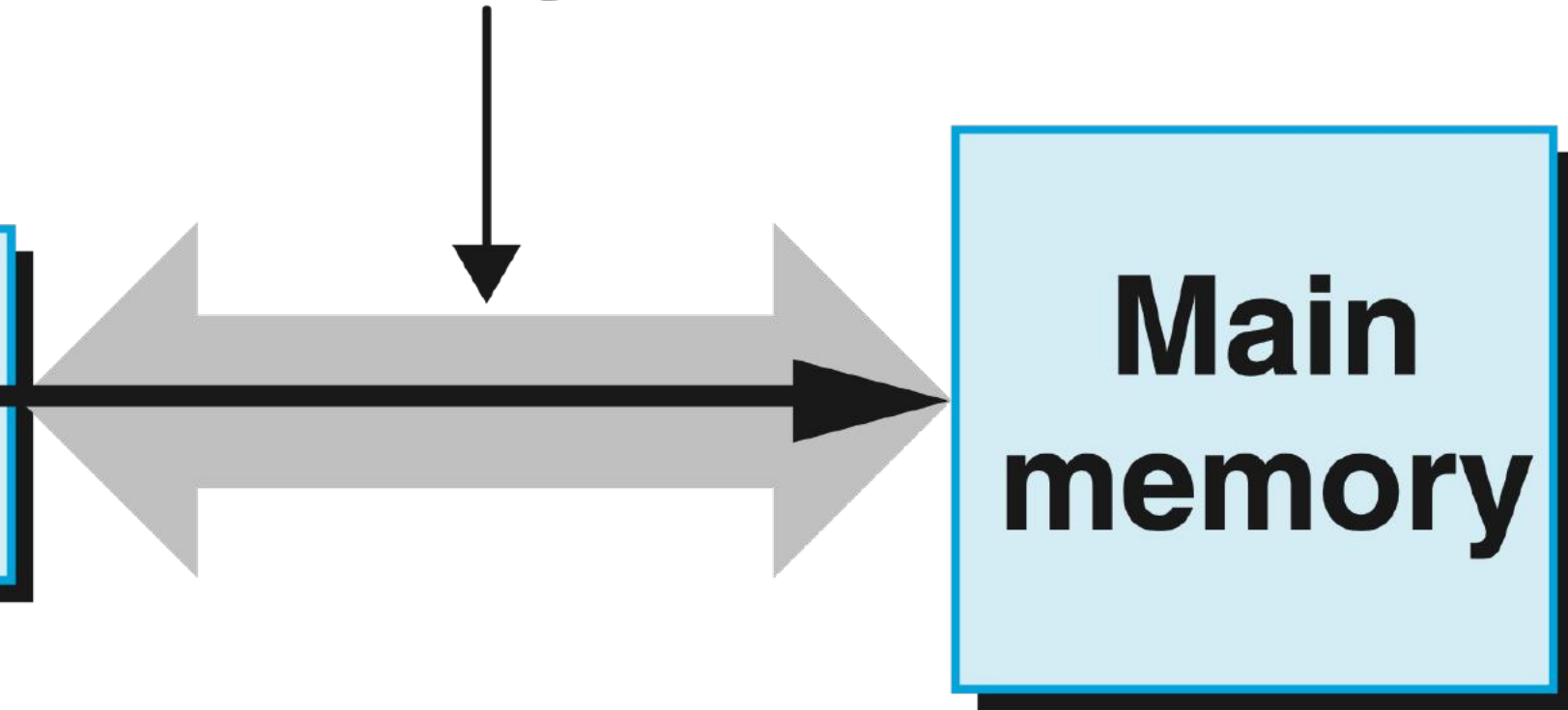
- `%rax` stores the return value
- `%rdi` stores the first parameter to a function
- `%rsi` stores the second parameter to a function
- `%rdx` stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- `%rsp` stores the address of the current top of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

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

# Instructions Are Just Bytes!

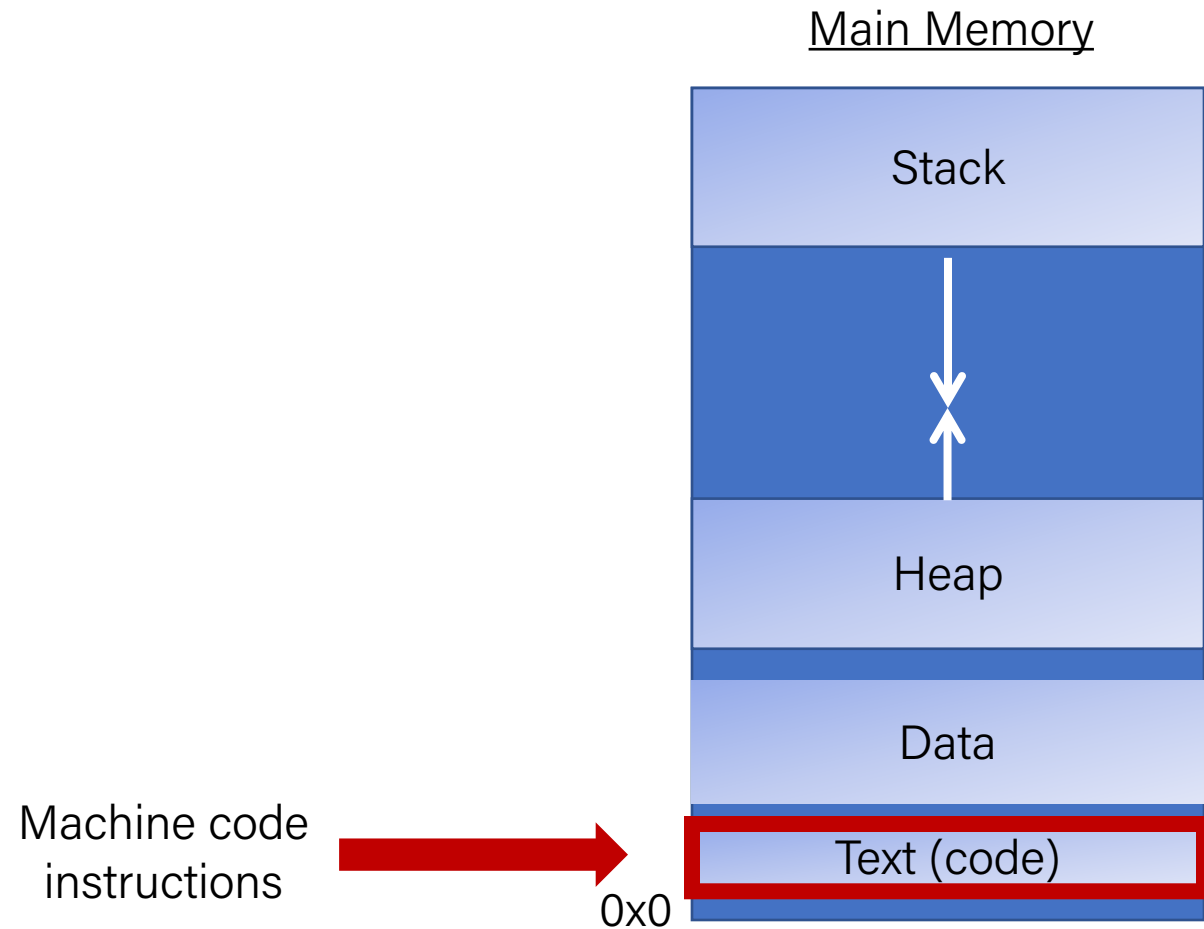


Memory bus



“hello, world\n  
hello code

# Instructions Are Just Bytes!



# %rip

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>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

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.

0x4004ed

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

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

0x4004ee

%rip

# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

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

0x4004f1

%rip

# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl   $0x0,-0x4(%rbp)
addl   $0x1,-0x4(%rbp)
jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

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

0x4004f8

%rip

# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

0x4004fc

%rip

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

# %rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

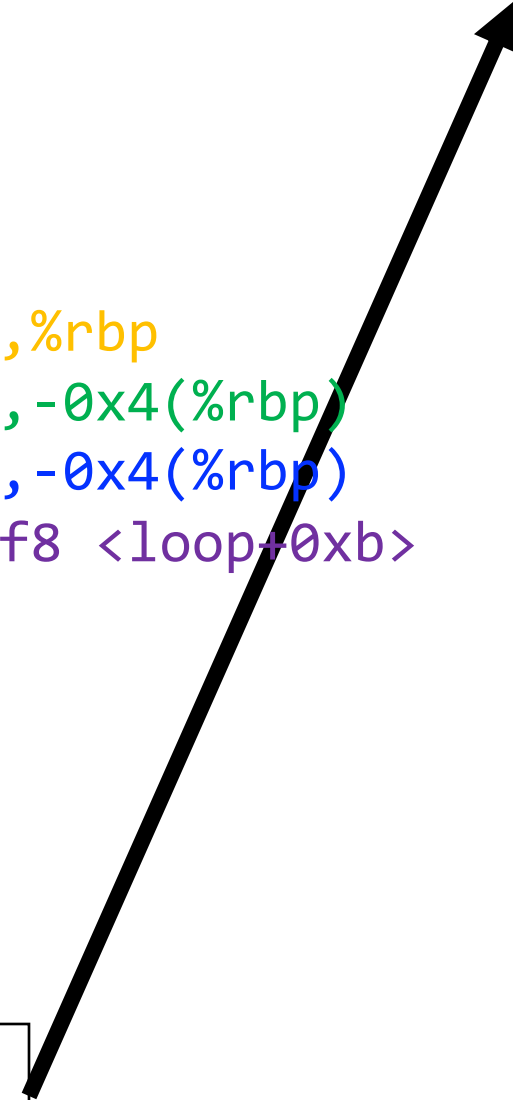
mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



0x4004fc

%rip

Special hardware sets the program counter to the next instruction:

$\%rip += \text{size of bytes of current instruction}$

# 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!

# Jump!

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl   $0x0,-0x4(%rbp)
addl   $0x1,-0x4(%rbp)
jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

0x4004fc

%rip

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

# Jump!

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp  
mov %rsp,%rbp  
movl \$0x0,-0x4(%rbp)  
addl \$0x1,-0x4(%rbp)  
jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

0x4004fc

%rip

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

# Jump!

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl   $0x0,-0x4(%rbp)
addl   $0x1,-0x4(%rbp)
jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

0x4004fc

%rip

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

# Jump!

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl   $0x0,-0x4(%rbp)
addl   $0x1,-0x4(%rbp)
jmp    4004f8 <loop+0xb>
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

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

0x4004fc

%rip

# Jump!

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp  
mov %rsp,%rbp  
movl \$0x0,-0x4(%rbp)  
addl \$0x1,-0x4(%rbp)  
jmp 4004f8 <loop+0xb>

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

This assembly represents an infinite loop in C!

```
while (true) {...}
```

0x4004fc

%rip

# jmp

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**?

# Lecture Plan

- More practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions

# Control

- In C, we have control flow statements like **if**, **else**, **while**, **for**, etc. to write programs that are more expressive than just one instruction following another.
- This is conditional execution of statements: executing statements if one condition is true, executing other statements if one condition is false, etc.
- How is this represented in assembly?

# Control

```
if (x > y) {  
    // a  
}  
else {  
    // b  
}
```

In Assembly:

1. Calculate the condition result
2. Based on the result, go to a or b

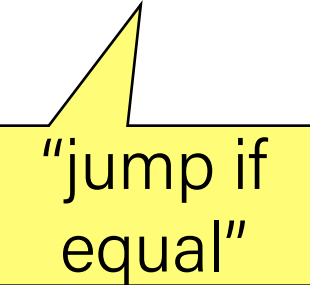
# Control

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

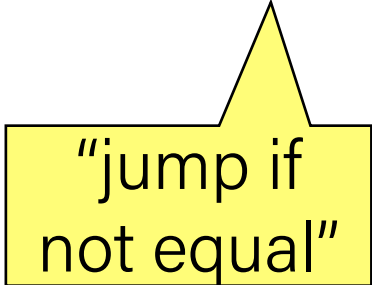
Common Pattern:

1. **cmp S1, S2** // compare two values

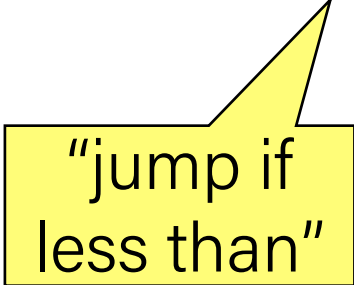
2. **je [target]** or **jne [target]** or **j1 [target]** or ... // conditionally  
// jump



"jump if  
equal"



"jump if  
not equal"



"jump if  
less than"

# Conditional Jumps

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

Instruction	Synonym	Set Condition
<code>je Label</code>	<code>jz</code>	Equal / zero
<code>jne Label</code>	<code>jnz</code>	Not equal / not zero
<code>js Label</code>		Negative
<code>jns Label</code>		Nonnegative
<code>jg Label</code>	<code>jnl</code>	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	Greater or equal (signed >=)
<code>j1 Label</code>	<code>jnge</code>	Less (signed <)
<code>jle Label</code>	<code>jng</code>	Less or equal (signed <=)
<code>ja Label</code>	<code>jnb</code>	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	Above or equal (unsigned >=)
<code>jb Label</code>	<code>jnae</code>	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	Below or equal (unsigned <=)

# Control

Read `cmp S1,S2` as “compare S2 to S1”:

```
// Jump if %edi > 2
```

```
cmp $2, %edi
```

```
jg [target]
```

```
// Jump if %edi != 3
```

```
cmp $3, %edi
```

```
jne [target]
```

```
// Jump if %edi == 4
```

```
cmp $4, %edi
```

```
je [target]
```

```
// Jump if %edi <= 1
```

```
cmp $1, %edi
```

```
jle [target]
```

# Control

Read `cmp S1,S2` as “compare S2 to S1”:

```
// Jump if %edi > 2  
cmp $2, %edi  
jg [target]
```

```
// Jump if %edi == 4  
cmp $4, %edi  
je [target]
```

```
// Jump if %edi <= 1  
cmp $3, %edi  
jne [target]
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?

# Control

- The CPU has special registers called ***condition codes*** that are like “global variables”. They automatically keep track of information about the most recent arithmetic or logical operation.
  - **cmp** compares via calculation (subtraction) and info is stored in the condition codes
  - conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes? How do they store this information?

# Condition Codes

Alongside normal registers, the CPU also has single-bit condition code registers. They store the results of the most recent arithmetic or logical operation.

Most common condition codes:

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

# Condition Codes

Alongside normal registers, the CPU also has single-bit condition code registers. They store the results of the most recent arithmetic or logical operation.

Example: if we calculate  $t = a + b$ , condition codes are set according to:

- **CF**: Carry flag (Unsigned Overflow).  $(\textit{unsigned})\ t < (\textit{unsigned})\ a$
- **ZF**: Zero flag (Zero).  $(t == 0)$
- **SF**: Sign flag (Negative).  $(t < 0)$
- **OF**: Overflow flag (Signed Overflow).  $(a < 0 == b < 0) \ \&\& \ (t < 0 \neq a < 0)$

# Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

`CMP S1, S2`

`S2 - S1`

Instruction	Description
<code>cmpb</code>	Compare byte
<code>cmpw</code>	Compare word
<code>cmpd</code>	Compare double word
<code>cmpq</code>	Compare quad word

# Control

Read **cmp S1, S2** as “compare S2 to S1”. It calculates  $S2 - S1$  and updates the condition codes with the result.

```
// Jump if %edi > 2
// calculates %edi - 2
cmp $2, %edi
jg [target]
```

```
// Jump if %edi != 3
// calculates %edi - 3
cmp $3, %edi
jne [target]
```

```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

```
// Jump if %edi <= 1
// calculates %edi - 1
cmp $1, %edi
jle [target]
```

# Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

Instruction	Synonym	Set Condition
<i>je Label</i>	<i>jz</i>	Equal / zero (ZF = 1)
<i>jne Label</i>	<i>jnz</i>	Not equal / not zero (ZF = 0)
<i>js Label</i>		Negative (SF = 1)
<i>jns Label</i>		Nonnegative (SF = 0)
<i>jg Label</i>	<i>jnle</i>	Greater (signed >) (ZF = 0 and SF = OF)
<i>jge Label</i>	<i>jnl</i>	Greater or equal (signed >=) (SF = OF)
<i>jl Label</i>	<i>jnge</i>	Less (signed <) (SF != OF)
<i>jle Label</i>	<i>jng</i>	Less or equal (signed <=) (ZF = 1 or SF != OF)
<i>ja Label</i>	<i>jnbe</i>	Above (unsigned >) (CF = 0 and ZF = 0)
<i>jae Label</i>	<i>jnb</i>	Above or equal (unsigned >=) (CF = 0)
<i>jb Label</i>	<i>jnae</i>	Below (unsigned <) (CF = 1)
<i>jbe Label</i>	<i>jna</i>	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

# Setting Condition Codes

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

TEST S1, S2

S2 & S1

Instruction	Description
testb	Test byte
testw	Test word
testl	Test double word
testq	Test quad word

**Cool trick:** if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

# Condition Codes

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

# Recap

- More practice: Reverse Engineering
- Assembly Execution and `%rip`
- Control Flow Mechanics

**Next time:** *Conditional branches*