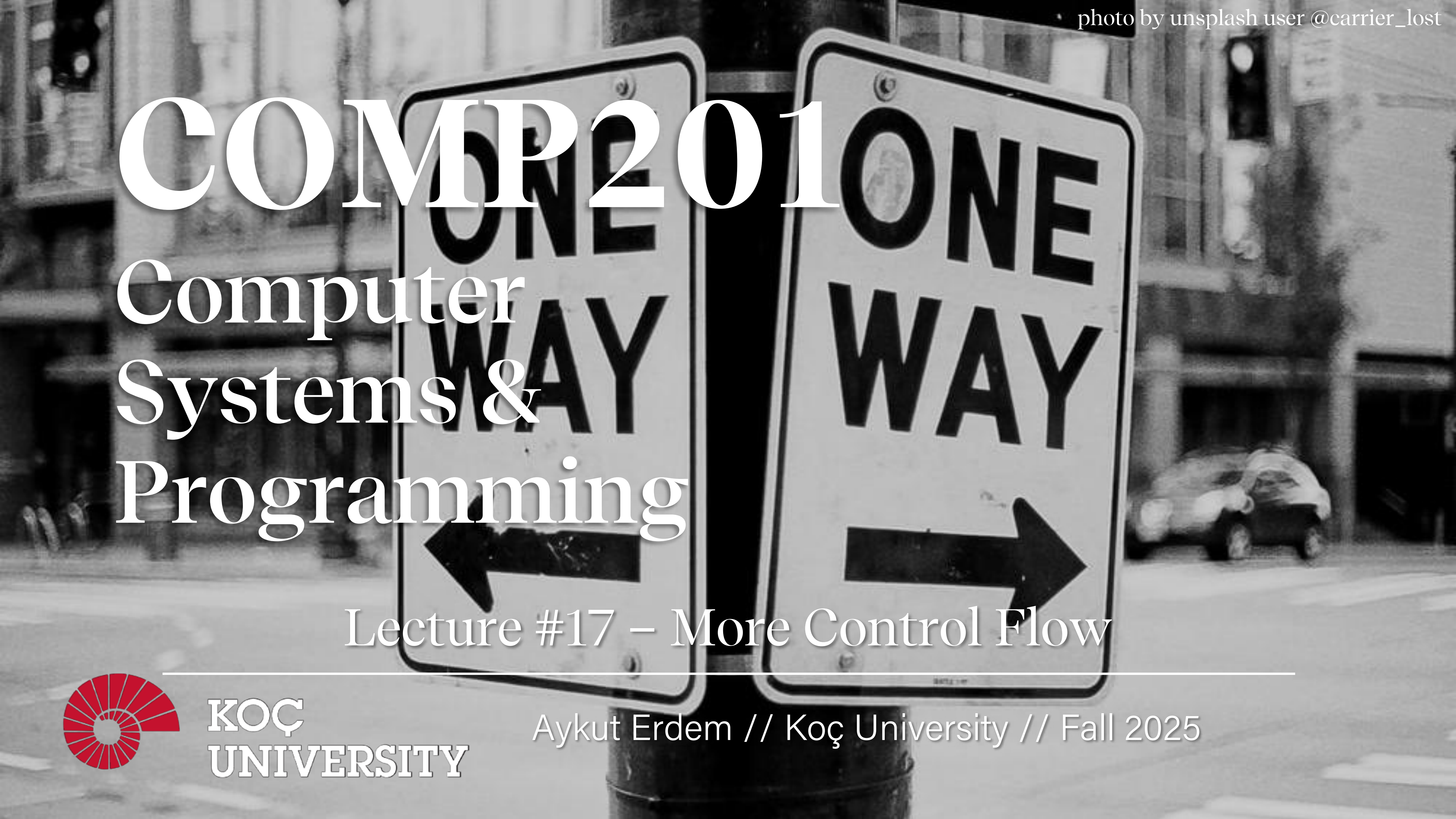


photo by unsplash user @carrier_lost

COMP201

Computer Systems & Programming

Lecture #17 – More Control Flow

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2025

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Assg 3 is out this week
- Lab 6 on machine programming with Assembly this Friday.

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions

Recap: Executing Instructions

So far:

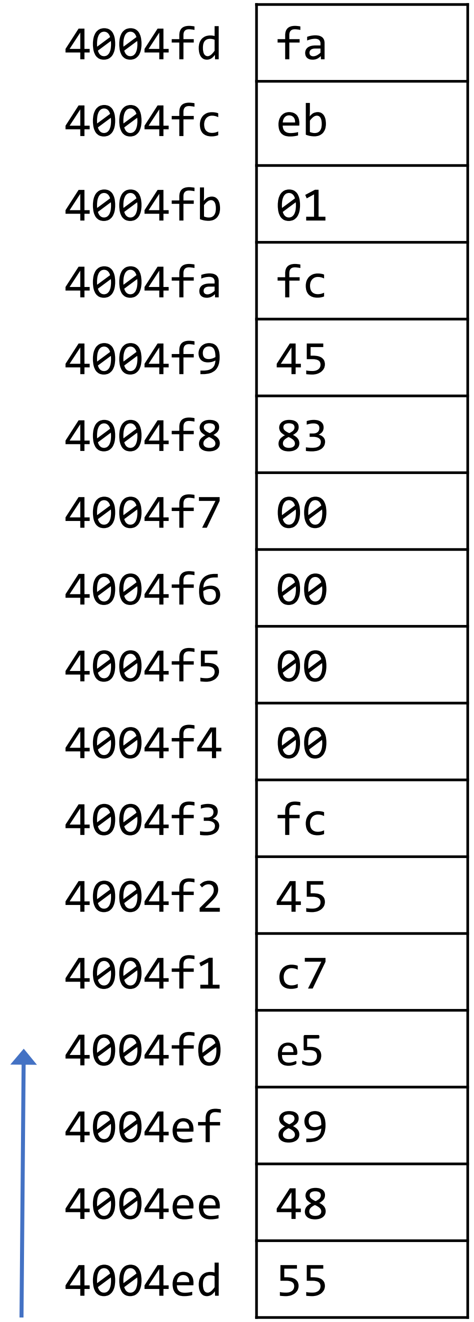
- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Last time:

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

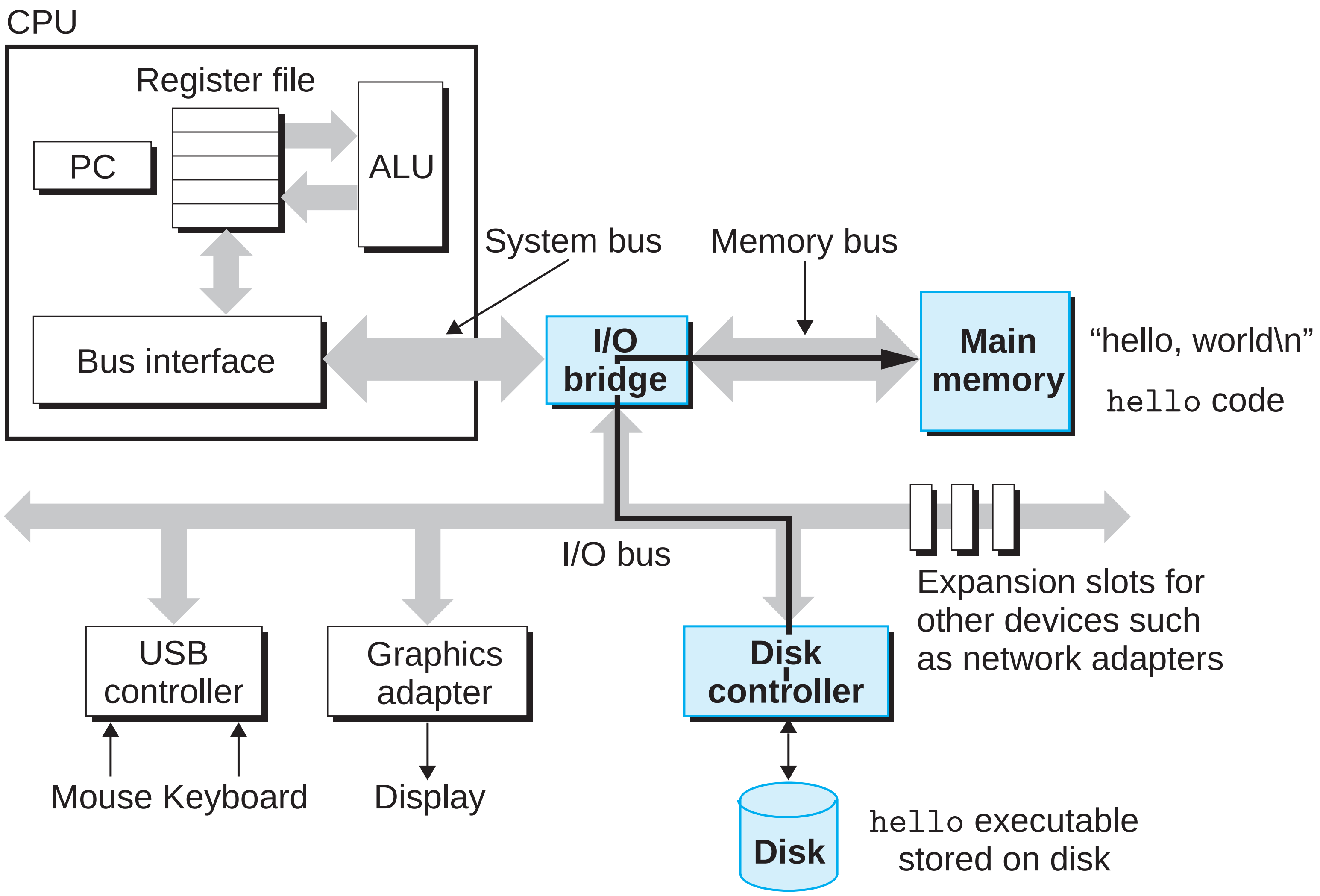
Answer:

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

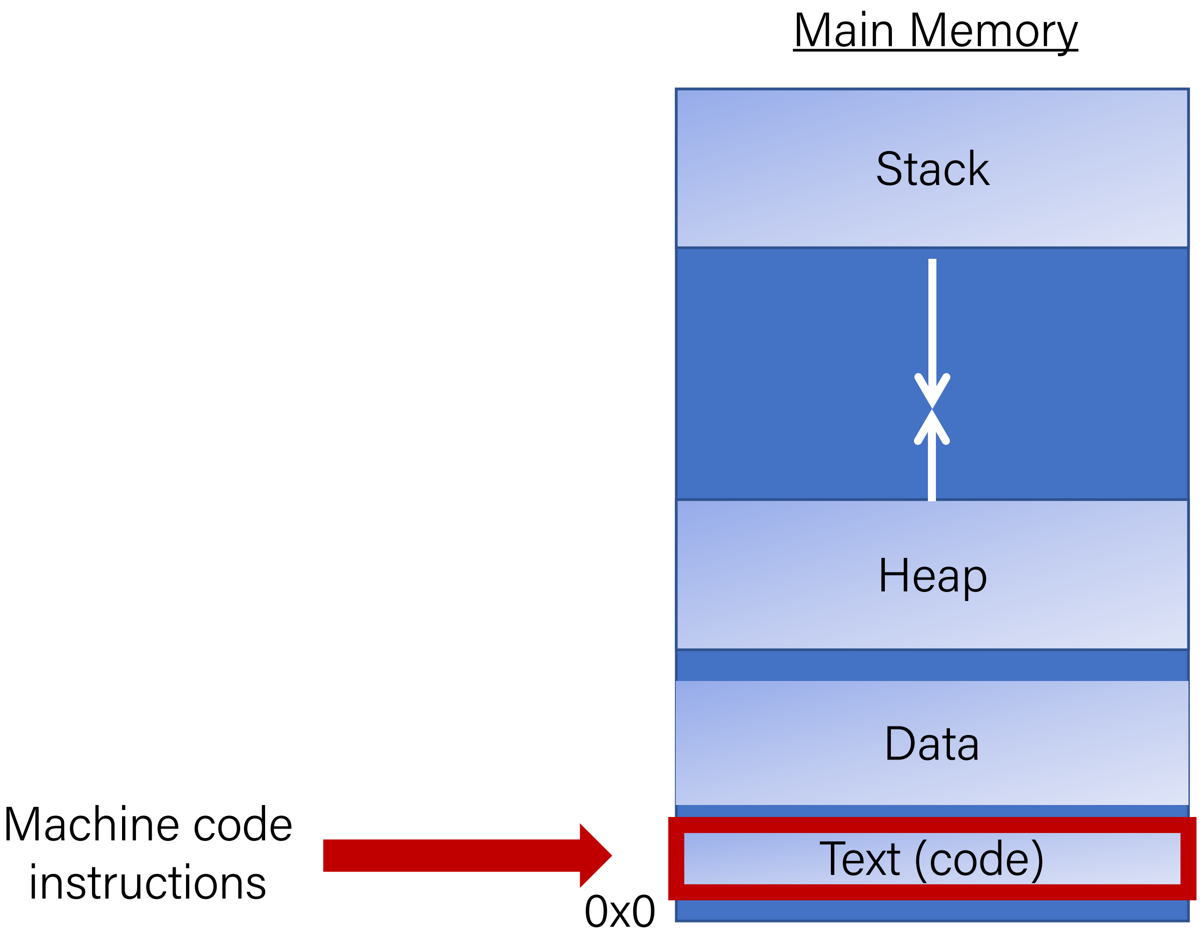


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Recap: Instructions Are Just Bytes!



Recap: Instructions Are Just Bytes!

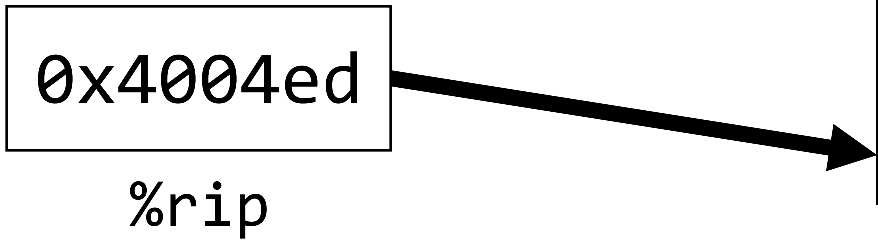


Recap: %rip

```
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

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



0x4004ed

%rip

Recap: `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
```

Recap: Control

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

In Assembly:

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

Recap: 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
`je` *`Label`*	`jz`	Equal / zero
`jne` *`Label`*	`jnz`	Not equal / not zero
`js` *`Label`*		Negative
`jns` *`Label`*		Nonnegative
`jg` *`Label`*	`jnle`	Greater (signed >)
`jge` *`Label`*	`jnl`	Greater or equal (signed >=)
`jl` *`Label`*	`jnge`	Less (signed <)
`jle` *`Label`*	`jng`	Less or equal (signed <=)
`ja` *`Label`*	`jnbe`	Above (unsigned >)
`jae` *`Label`*	`jnb`	Above or equal (unsigned >=)
`jb` *`Label`*	`jnae`	Below (unsigned <)
`jbe` *`Label`*	`jna`	Below or equal (unsigned <=)

Recap: 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.

Recap: 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
`cmpb`	Compare byte
`cmpw`	Compare word
`cmpl`	Compare double word
`cmpq`	Compare quad word

Recap: 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!

Exercise 1: Conditional jump

```
je target              jump if ZF is 1
```

Let `%edi` store 0x10. Will we jump in the following cases?

`%edi` | `0x10`

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

Exercise 1: Conditional jump

```
je target              jump if ZF is 1
```

Let `%edi` store 0x10. Will we jump in the following cases?

`%edi` | `0x10`

1.
```
cmp $0x10,%edi
je  40056f
add  $0x1,%edi
```

S2 - S1 == 0, so jump

2.
```
test $0x10,%edi
je   40056f
add  $0x1,%edi
```

S2 & S1 != 0, so don’t jump 🤔

Exercise 2: Conditional jump

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

`%edi` 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   A. `4004d9`
   B. `4004db`
   C. `4004de`
   D. Other

Do not edit

How to change the design

What is the value of %rip after executing the jne instruction?

Exercise 2: Conditional jump

slido

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp     $0x6,%edi
  4004d9:   75 03       jne     4004de <if_then+0x8>
  400rdb:   83 c7 01    add     $0x1,%edi
  4004de:   8d 04 3f    lea     (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

%edi 0x5

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. 4004d9
   - B. 4004db
   - [✓ C. 4004de]
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. 4004e1
   - B. 0x2
   - C. 0xa
   - D. 0xc
   - E. Other

Do not edit
How to change the design

What is the value of %eax
when we hit the retq instruction?

Exercise 2: Conditional jump

```
00000000004004d6 <if_then>:
  4004d6:   83 ff 06    cmp    $0x6,%edi
  4004d9:   75 03       jne    4004de <if_then+0x8>
  400rdb:   83 c7 01    add    $0x1,%edi
  4004de:   8d 04 3f    lea    (%rdi,%rdi,1),%eax
  4004e1:   c3          retq
```

`%edi` `0x5`

1. What is the value of `%rip` after executing the `jne` instruction?
   - A. 4004d9
   - B. 4004db
   - [✓ C. 4004de]
   - D. Other

2. What is the value of `%eax` when we hit the `retq` instruction?
   - A. 4004e1
   - B. 0x2
   - [✓ C. 0xa]
   - D. 0xc
   - E. Other

Plan for Today

- If statements
- Loops
- Other Instructions That Depend On Condition Codes

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- If statements
- Loops
- Other Instructions That Depend On Condition Codes

Practice: Fill In The Blank

```
int if_then(int param1) {
  if ( __________ ) {
    _________ ;
  }

  return __________ ;
}
```

```
00000000004004d6 <if_then>:
  4004d6:    cmp  $0x6,%edi
  4004d9:    jne  4004de
  4004db:    add  $0x1,%edi
  4004de:    lea  (%rdi,%rdi,1),%eax
  4004e1:    retq
```

🤔

Practice: Fill In The Blank

```c
int if_then(int param1) {
  if (param1 == 6 ) {
        param1++;
  }

  return  param1 * 2;
}
```

```
00000000004004d6 <if_then>:
  4004d6:    cmp  $0x6,%edi
  4004d9:    jne  4004de
  4004db:    add  $0x1,%edi
  4004de:    lea  (%rdi,%rdi,1),%eax
  4004e1:    retq
```

Practice: Fill In The Blank

If-Else In C

```
if (__________) {
    _________;
} else {
    _________;
}

_________;
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill In The Blank

If-Else In C

```
if ( _________ ) {
    ________;
} else {
    _______;
}
______;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

Practice: Fill In The Blank

slido

If-Else In C

```
if (  arg > 3   ) {
      ________;
} else {
      ________;
}
______;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

🤔

Practice: Fill In The Blank

slido

If-Else In C

```
if (  arg > 3   ) {
    ret = 10;
} else {
    ________;
}
______;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

🤔

Practice: Fill In The Blank

slido

If-Else In C

```
if (  arg > 3   ) {
    ret = 10;
} else {
    ret = 0;
}
______;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

🤔

Practice: Fill In The Blank

slido

If-Else In C

```
if (  arg > 3   ) {
    ret = 10;
} else {
    ret = 0;
}
ret++;
```

```
400552 <+0>:   cmp   $0x3,%edi
400555 <+3>:   jle   0x40055e <if_else+12>
400557 <+5>:   mov   $0xa,%eax
40055c <+10>:  jmp   0x400563 <if_else+17>
40055e <+12>:  mov   $0x0,%eax
400563 <+17>:  add   $0x1,%eax
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```

🤔

Lecture Plan

- If statements (cont'd.)
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Set `%eax` (`i`) to 0.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Jump to another instruction.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Compare `%eax` (`i`) to 0x63 (99) by calculating `%eax` – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

`jle` means “jump if less than or equal”. This jumps if `%eax` <= 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Add 1 to %eax (`i`).

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Compare `%eax` (`i`) to 0x63 (99) by calculating `%eax` – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

`jle` means "jump if less than or equal". This jumps if **`%eax`** <= 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

We continue in this pattern until we do not make this conditional jump. When will that be?

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

We will stop looping when this comparison says that `%eax` – 0x63 > 0!

Loops and Control Flow

```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Then, we return from the function.

Common While Loop Construction

C

```
while (test) {
        body
}
```

Assembly

```
Jump to test
Body
Test
Jump to body if success
```

From Previous Slide:

```
0x0000000000400570 <+0>:     mov    $0x0,%eax
0x0000000000400575 <+5>:     jmp    0x40057a <loop+10>
0x0000000000400577 <+7>:     add    $0x1,%eax
0x000000000040057a <+10>:    cmp    $0x63,%eax
0x000000000040057d <+13>:    jle    0x400577 <loop+7>
0x000000000040057f <+15>:    repz retq
```

Lecture Plan

- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Common While Loop Construction

C For loop

```
for (init; test; update) {
        body
}
```

C Equivalent While Loop

```
init
while(test) {
      body
      update
}
```

Assembly pseudocode

```
Init
Jump to test
Body
Update
Test
Jump to body if success
```

`for` loops and `while` loops are treated (essentially) the same when compiled down to assembly.

Back to 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;
}
```

```
00000000004005b6 <sum_array>:
  4005b6:         mov    $0x0,%edx
  4005bb<+5>:     mov    $0x0,%eax
  4005c0<+10>:    jmp    4005cb <sum_array+21>
  4005c2<+12>:    movslq %edx,%rcx
  4005c5<+15>:    add    (%rdi,%rcx,4),%eax
  4005c8<+18>:    add    $0x1,%edx
  4005cb<+21>:    cmp    %esi,%edx
  4005cd<+23>:    jl     4005c2 <sum_array+12>
  4005cf<+25>:    repz retq
```

1. Which register is C code's `sum`?
2. Which register is C code's `i`?
3. Which assembly instruction is C code's `sum += arr[i]`?
4. What are the **`cmp`** and **`jl`** instructions doing?
   (**`jl`**: jump less; signed <)

Lecture Plan

- If Statements
- Loops
- Other Instructions That Depend On Condition Codes

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **`jmp`** instructions conditionally jump to a different next instruction
- **`set`** instructions conditionally set a byte to 0 or 1
- new versions of **`mov`** instructions conditionally move data

`set`: Read condition codes

`set` instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte register (e.g., `%al`) or single-byte memory location
- Does not perturb other bytes of register
- Typically followed by `movzbl` to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

cmov: Conditional move

cmovx src,dst conditionally moves data in src to data in dst.

- Mov src to dst if condition x holds; no change otherwise
- src is memory address/register, dst is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: result = test ? then: else;

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp    %edi,%esi
mov    %edi, %eax
cmovge %esi, %eax
retq
```

Ternary Operator

The ternary operator is a shorthand for using if/else to evaluate to a value.

```
condition ? expressionIfTrue : expressionIfFalse
```

```
int x;
if (argc > 1) {
     x = 50;
} else {
     x = 0;
}

// equivalent to
int x = argc > 1 ? 50 : 0;
```

cmov: Conditional move

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		Nonnegative (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Practice: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

-14/4 should yield -3 rather than -4

(See Sec. 2.3.7)

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

Put **x** + 3 into **%eax** (add appropriate bias, 2^2-1)

To see whether **x** is negative, zero, or positive

If **x** is positive, put **x** into **%eax**

Divide **%eax** by 4

Extra Practice

Practice: Fill In The Blank

Note: .L2/.L3 are "labels" that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = ________;
    while (__________) {
      result = ______________;
      a = __________;
    }
    return result;
}
```

Common while loop construction:

```
Jump to test
Body
Test
Jump to body if success
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice: Fill In The Blank

Note: .L2/.L3 are "labels" that make jumps easier to read.

C Code

```
long loop(long a, long b) {
    long result = ___1___;
    while (__a < b__) {
      result = result*(a+b);
      a = __a + 1__;
    }
    return result;
}
```

Common while loop construction:

```
Jump to test
Body
Test
Jump to body if success
```

What does this assembly code translate to?

```
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```

Practice: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

Practice: “Escape Room”

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the `escapeRoom` function such that it returns true (1) and not false (0)?

First param > 2 or == 1.

Recap

- Assembly Execution and `%rip`
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If statements
- Loops
  - While loops
  - For loops
- Other Instructions That Depend On Condition Codes

Next time: *Function calls in assembly*