

CS107, Lecture 18

Assembly: Control Flow

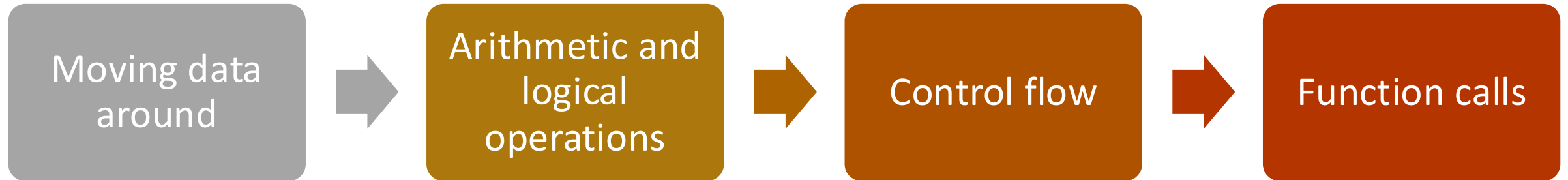
Reading: B&O 3.6

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

Learning Assembly



This Lecture

Reference Sheet: cs107.stanford.edu/resources/x86-64-reference.pdf
See more guides on Resources page of course website!

Learning Goals

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

Lecture Plan

- Assembly Execution and %rip
- Control Flow Mechanics
 - **cmp**
 - Condition Codes
- If Statements
- **test**, and **more** about Condition Codes

Lecture Plan

- **Assembly Execution and %rip**
- Control Flow Mechanics
 - **cmp**
 - Condition Codes
- If Statements
- **test**, and **more** about Condition Codes

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



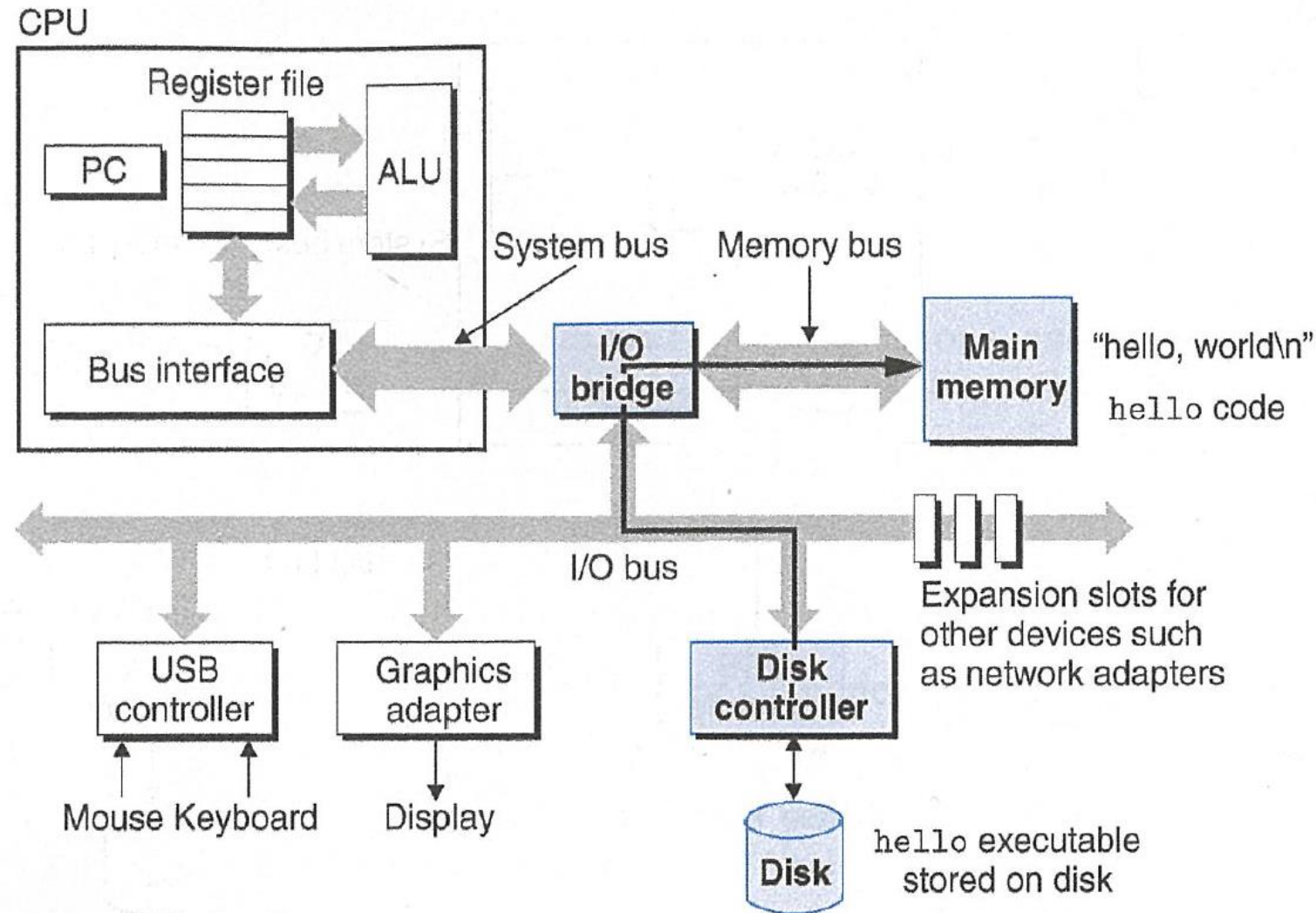
Register Responsibilities

Some registers take on special responsibilities during program execution.

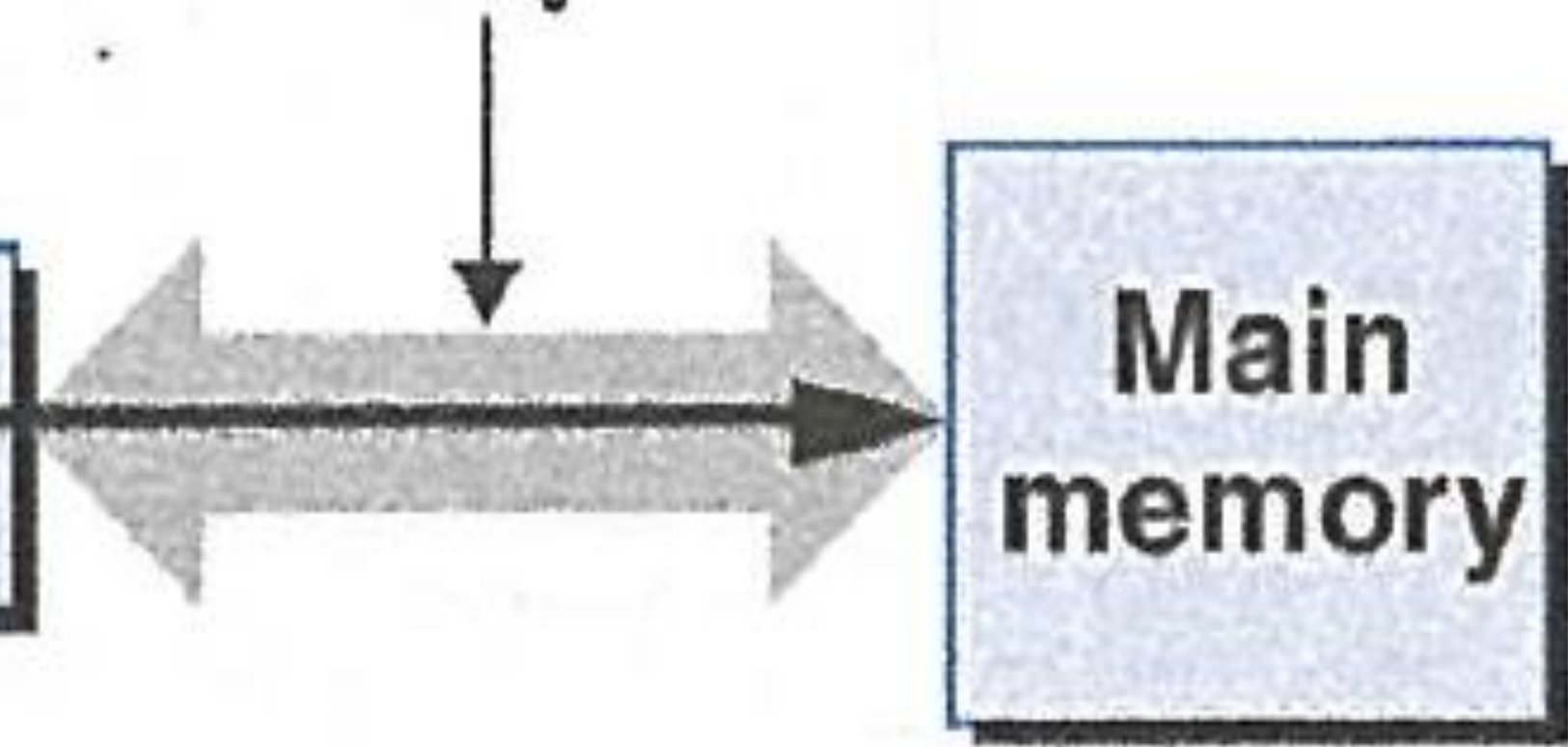
- `%rax` stores the return value
- `%rdi` stores the first parameter to a function
- `%rsi` stores the second parameter to a function
- `%rdx` stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- `%rsp` stores the address of the current top of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

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

Instructions Are Just Bytes!

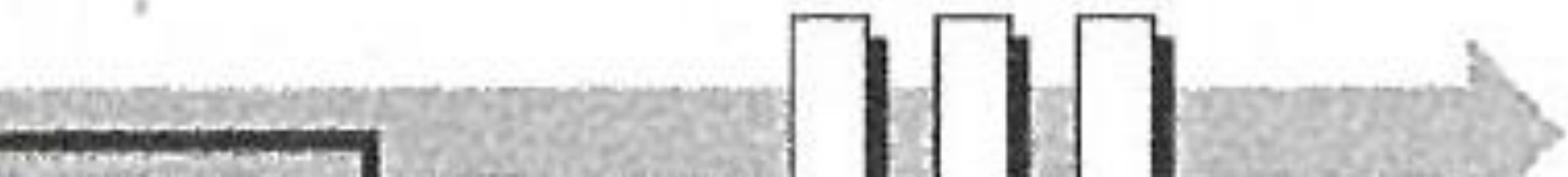


Memory bus

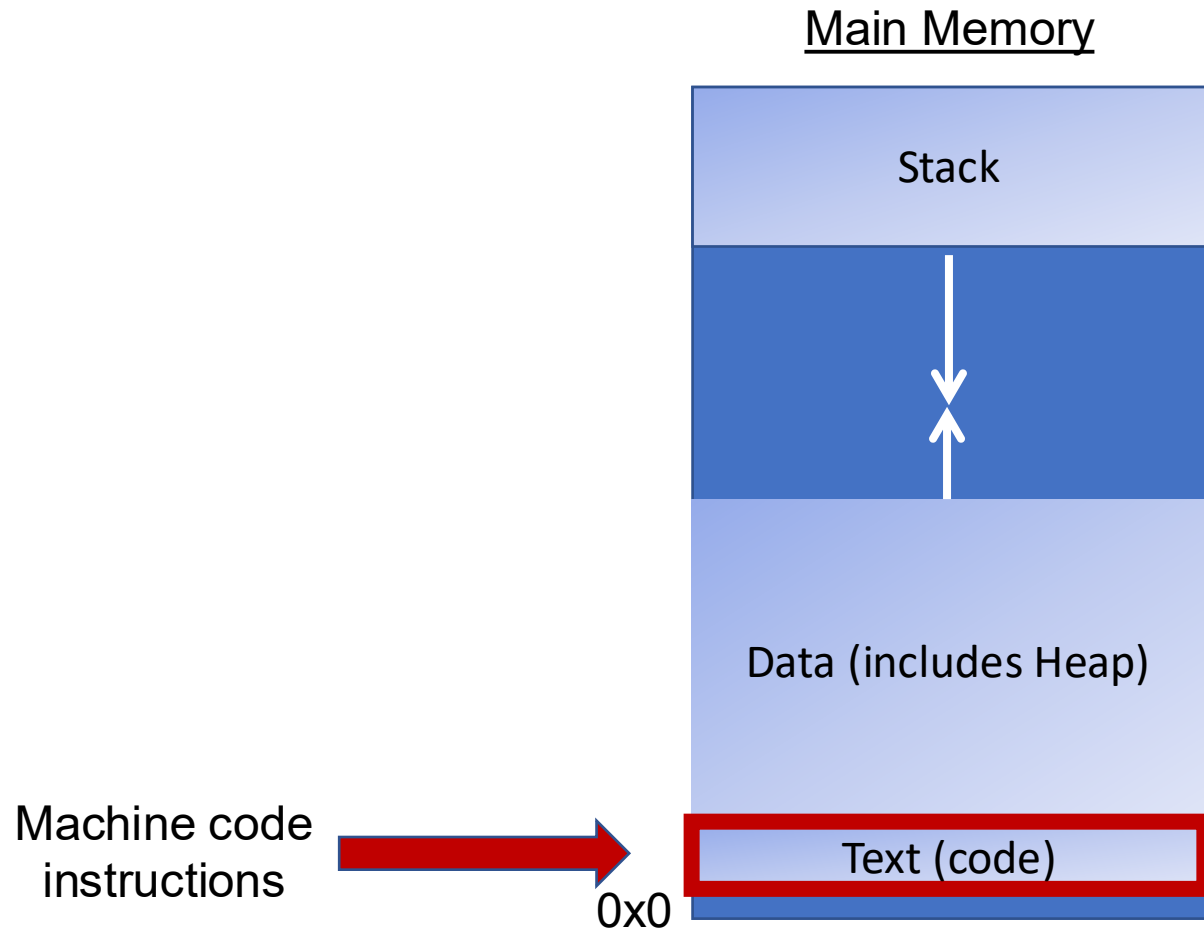


"hello, world\n"

hello code



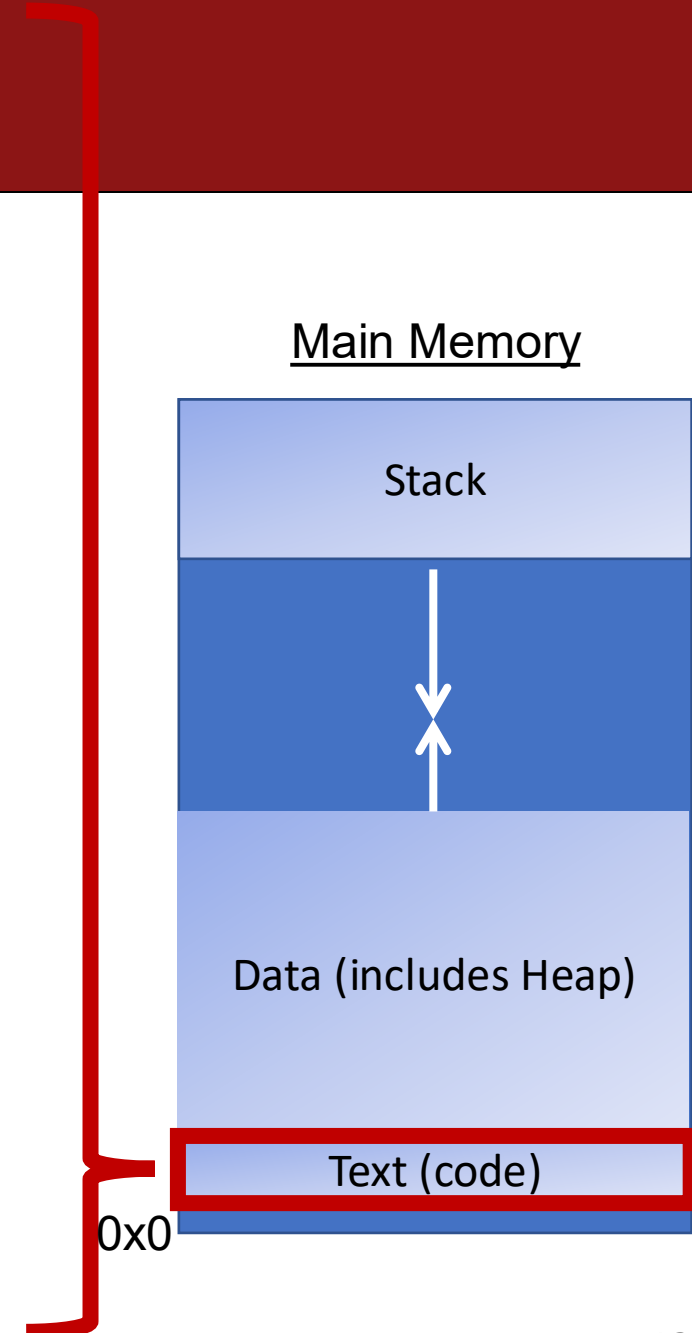
Instructions Are Just Bytes!



%ori

```
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>

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

0x4004ee

%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>

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

0x4004f1

%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>

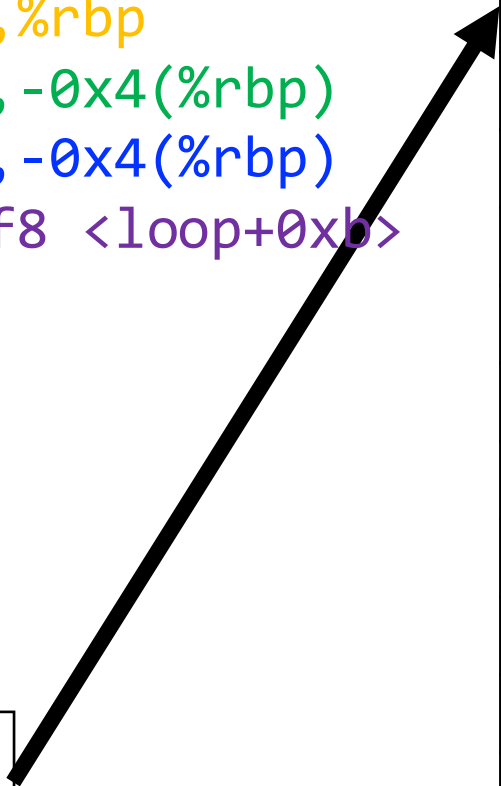


The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the **next instruction** to be executed.

0x4004f8

%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

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

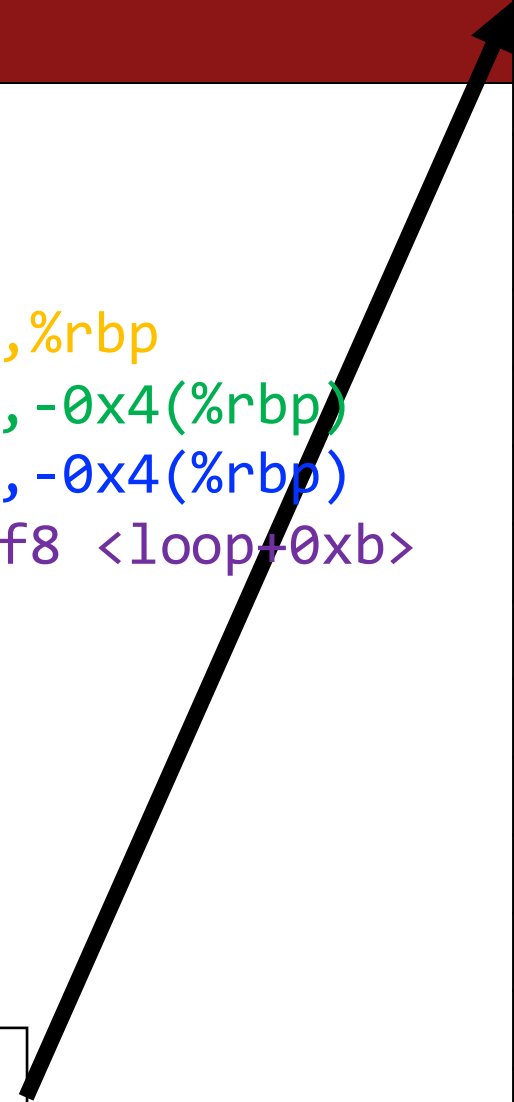
mov %rsp,%rbp

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



Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

0x4004fc

%rip

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>



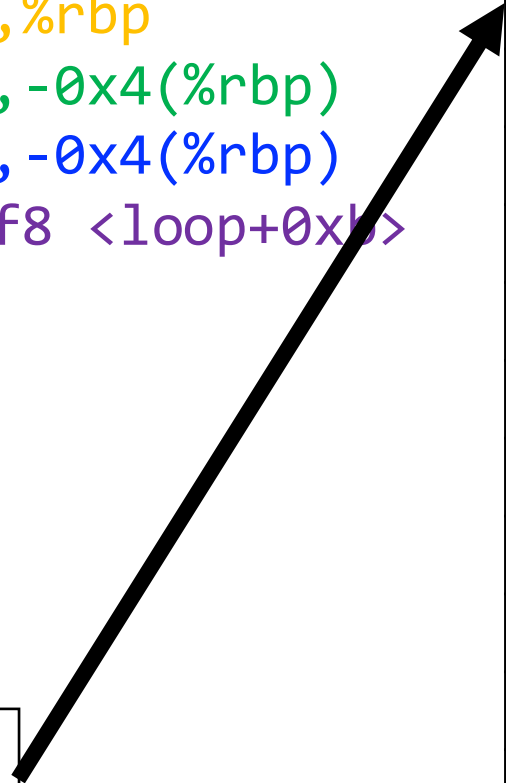
This assembly represents an infinite loop in C!

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

0x4004fc

%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



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

- Assembly Execution and %rip
- **Control Flow Mechanics**
 - **cmp**
 - Condition Codes
- If Statements
- **test**, and **more** about Condition Codes

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

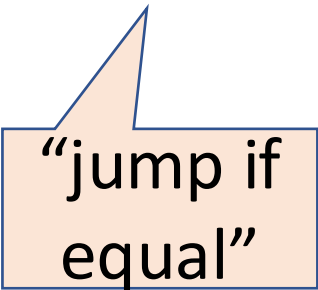
Control

- 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

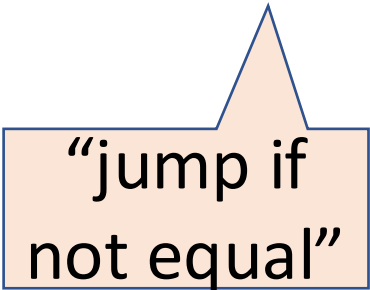
Common Pattern:

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

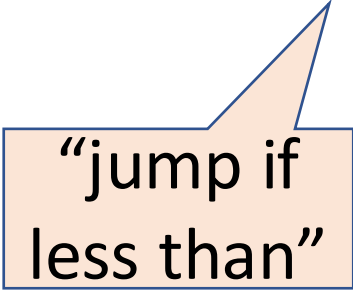
2. **je [target]** or **jne [target]** or **jl [target]** or ... // conditionally jump



“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>jnbe</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 == 4  
cmp $4, %edi  
je [target]
```

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

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

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

Exercise 1: Conditional jump

`je target` `jump if equal`
`jle target` `jump if less than or equal`

Let `%edi` store `0x10`, and `%esi` store `0x15`. Will we jump in the following cases?

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

`%edi`

`%esi`

2. `cmp %edi,%esi`
`jle 40056f`
`add $0x1,%edi`

Exercise 1: Conditional jump

`je target` `jump if equal`
`jle target` `jump if less than or equal`

Let `%edi` store `0x10`, and `%esi` store `0x15`. Will we jump in the following cases?

1. `cmp $0x10,%edi`
`je 40056f` “jump if `%edi == 0x10`” (true)
`add $0x1,%edi`

`%edi`

`%esi`

2. `cmp %edi,%esi`
`jle 40056f` “jump if `%esi <= %edi`” (false)
`add $0x1,%edi`

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.

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

Conditional Jumps

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

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

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 == 4:  
// calculates %edi - 4,  
// jump if result = 0 (ZF=1)  
cmp $4, %edi  
je [target]
```

```
// Jump if %edi != 3:  
// calculates %edi - 3,  
// jump if result != 0 (ZF=0)  
cmp $3, %edi  
jne [target]
```

```
// Jump if %edi > 2:  
// calculates %edi - 2,  
// jump if result != 0 (ZF=0) AND either  
// (nonnegative and no signed overflow)  
// OR (negative and signed overflow)  
// (SF=OF)  
cmp $2, %edi  
jg [target]
```

```
// Jump if %edi <= 1:  
// calculates %edi - 1,  
// jump if result = 0 (ZF=1) OR (negative  
// and no signed overflow) OR  
// (nonnegative and signed overflow)  
// (SF!=OF)  
cmp $1, %edi  
jle [target]
```

Control

If a **cmp** and jump are adjacent, we can use the instructions themselves to understand what is being checked, without needing to dig into condition codes. But it's still important to understand the condition codes to know how the conditions are really checked.

Lecture Plan

- Assembly Execution and %rip
- Control Flow Mechanics
 - **cmp**
 - Condition Codes
- **If Statements**
- **test**, and **more** about Condition Codes

Practice: Fill In The Blank

```
int if_then(int param1) { 000000000401126 <if_then>:
    if ( _____ ) {    401126:    cmp     $0x5,%edi
                          401129:    jg     40112e
                          40112b:    add     $0x1,%edi
                          40112e:    lea    (%rdi,%rdi,1),%eax
                          401131:    ret
    }
    return _____;
}
```


Weird common structure – frequently checks *opposite* of code condition to know when to skip the if body.



Practice: Fill In The Blank

```
int if_then(int param1) { 000000000401126 <if_then>:
    if ( param1 < 6 ) {    401126:      cmp      $0x5,%edi
        param1++ ;        401129:      jg       40112e
    }                       40112b:      add     $0x1,%edi
                                40112e:      lea    (%rdi,%rdi,1),%eax
                                401131:      ret

    return param1 * 2;
}
```



Alternative Construction

```
int if_then(int param1) { 000000000401126 <if_then>:
    if ( param1 == 6 ) {   401126:    cmp     $0x6,%edi
        param1++ ;        401129:    je     40112f
    }                       40112b:    lea   (%rdi,%rdi,1),%eax
                                40112e:    ret
                                40112f:    add   $0x1,%edi
                                401132:    jmp   40112b
    return param1 * 2;
}
```

The structure may vary – to identify the **if** statement condition, identify the case where you do *extra operations*.

Practice: Reverse Engineering

```
int func(int nums[], int y) {
    int z = nums[_____] * _____;
    if (_____) {
        z -= _____;
    }
    return _____;
}
```

```
000000000401126 <elem_arithmetic>:
401126: movslq %esi,%rax
401129: mov    %esi,%edx
40112b: imul  (%rdi,%rax,4),%edx
40112f: mov    %edx,%eax
401131: cmp    %edx,%esi
401133: jge   401138
401135: sub    $0x2,%eax
401138: lea   0x2(%rax,%rax,1),%eax
40113c: ret
```

Work through the remaining blanks in groups, and input your responses on PollEv: pollev.com/cs107



Practice: Reverse Engineering

```
int func(int nums[], int y) {
    int z = nums[y] * y;
    if (z > y) {
        z -= 2;
    }
    return 2 * z + 2;
}
```

```
000000000401126 <elem_arithmetic>:
 401126: movslq %esi,%rax // nums in %rdi, y in %esi
 401129: mov    %esi,%edx // sign-extend y into %rax
 40112b: imul  (%rdi,%rax,4),%edx // copy y into %edx
 40112f: mov    %edx,%eax // multiply %edx by nums[y]
 401131: cmp    %edx,%esi // copy %edx (z) into %eax
 401133: jge   401138 // compute y - z
 401135: sub    $0x2,%eax // jump if %esi (y) >= %edx (z)
 401138: lea   0x2(%rax,%rax,1),%eax // subtract 2 from %eax
 40113c: ret   // put 2 * %eax + 2 into %eax (2*z + 2)
```

L18. Reverse Engineering

0 surveys completed



0 surveys underway

What goes in blank #3 for the if statement condition?

Join by Web

Pollev.com/cs107

Join by QR code
Scan with your camera app



Common If-Else Construction

If-Else In C

```
int if_else(int param1) {  
    if (param1 != 6) {  
        param1++;  
    } else {  
        param1--;  
    }  
  
    return param1 * 2;  
}
```

If-Else In Assembly pseudocode

Check opposite of code condition
Jump to else-body if test passes

If-body

Past if body

Else-body

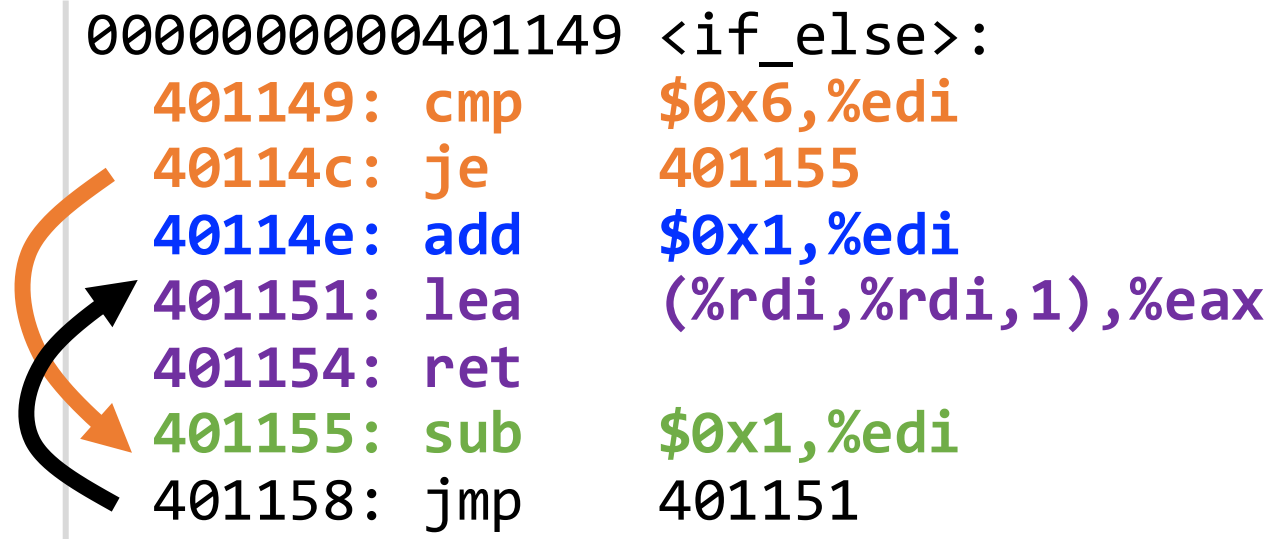
Jump up to past if body

Common If-Else Construction

If-Else In C

```
int if_else(int param1) {  
    if (param1 != 6) {  
        param1++;  
    } else {  
        param1--;  
    }  
  
    return param1 * 2;  
}
```

```
0000000000401149 <if_else>:  
401149: cmp     $0x6,%edi  
40114c: je     401155  
40114e: add     $0x1,%edi  
401151: lea    (%rdi,%rdi,1),%eax  
401154: ret  
401155: sub     $0x1,%edi  
401158: jmp    401151
```



If-Else In Assembly pseudocode

Check opposite of code condition
Jump to else-body if test passes

If-body

Past if body

Else-body

Jump up to past if body


If-Else Construction Variations

C Code

```
int if_else_2(int param1) {  
    if (param1 == 6) {  
        param1++;  
    } else {  
        param1--;  
    }  
  
    return param1 * 2;  
}
```

Assembly

```
000000000040115a <if_else_2>:  
    40115a: cmp     $0x6,%edi  
    40115d: je     401166  
    40115f: sub     $0x1,%edi  
    401162: lea   (%rdi,%rdi,1),%eax  
    401165: ret  
    401166: add     $0x1,%edi  
    401169: jmp    401162
```



The structure may vary – key idea is to identify what operation(s) go with what check.

Lecture Plan

- Assembly Execution and %rip
- Control Flow Mechanics
 - **cmp**
 - Condition Codes
- If Statements
- **test, and more about Condition Codes**

Setting Condition Codes

Usually when **cmp** is paired with conditional jumps, we can read them together. But other instructions use the condition codes in different ways. Example:

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

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



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`

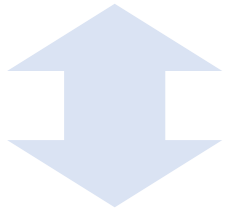
2. `test $0x10,%edi`
`je 40056f`
`add $0x1,%edi`

`S2 & S1 != 0, so don't jump`

The test Instruction

- TEST S1, S2 is S2 & S1

```
test %edi, %edi  
jns ...
```



%edi & %edi is nonnegative

%edi is nonnegative

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

- Assembly Execution and %rip
- Control Flow Mechanics
 - `cmp`
 - Condition Codes
- If Statements
- **test**, and more about Condition Codes

Lecture 18 takeaway: We represent control flow in assembly by storing information in condition codes and having instructions that act differently depending on the condition code values. Conditionals commonly use cmp or test along with jumps to conditionally skip over assembly instructions.

Next time: loops