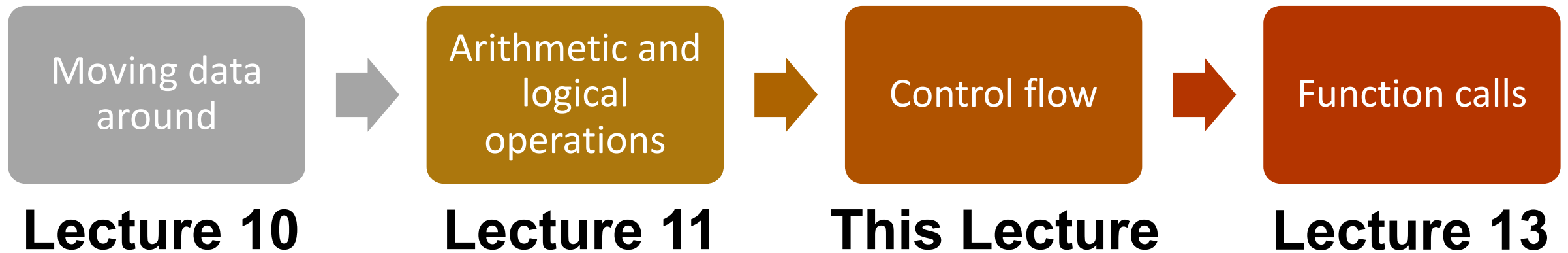


CS107, Lecture 12

Assembly: Control Flow

Reading: B&O 3.6

Learning Assembly



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

Learning Goals

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

Lecture Plan

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

Lecture Plan

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

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



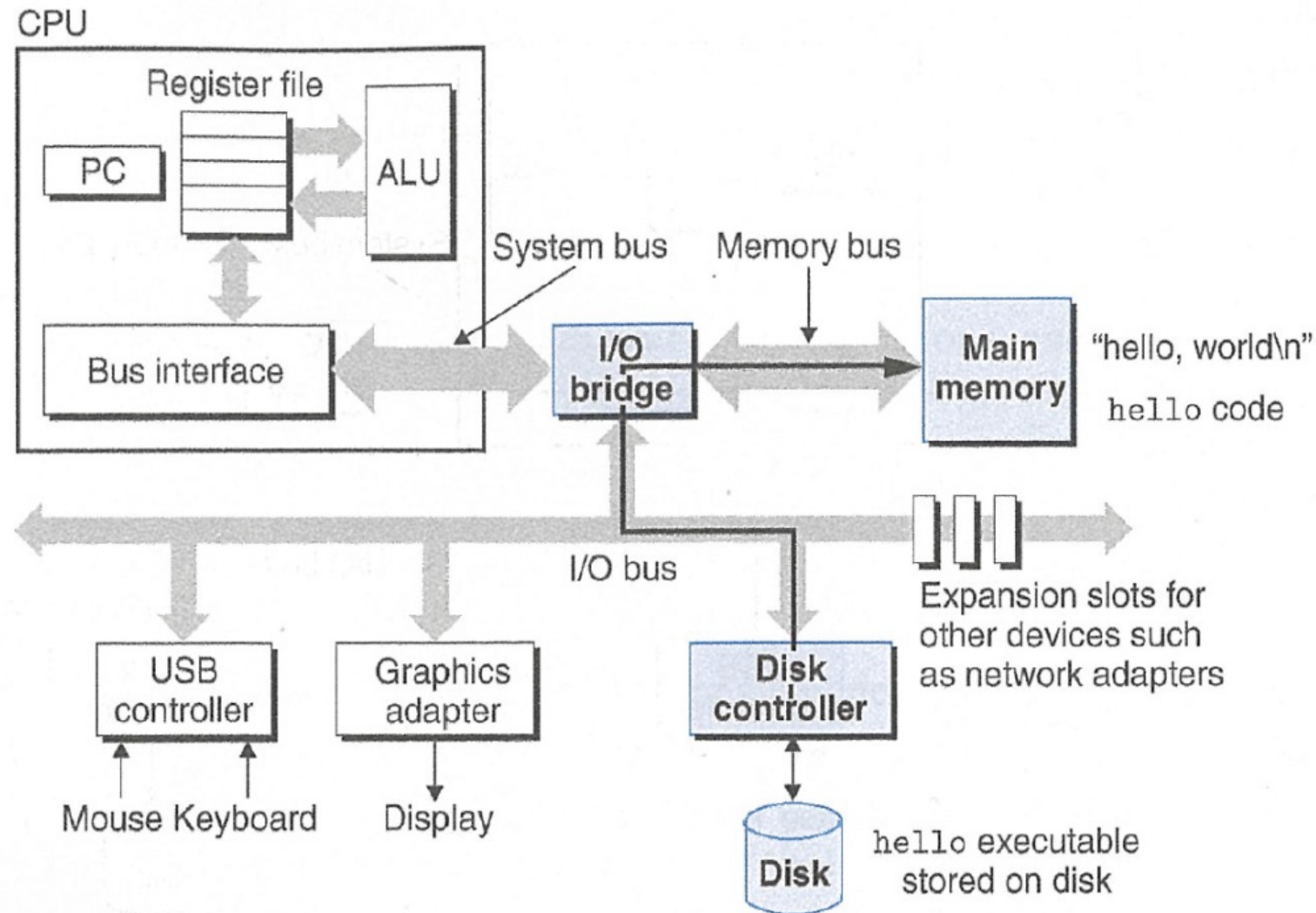
Register Responsibilities

Some registers take on special responsibilities during program execution.

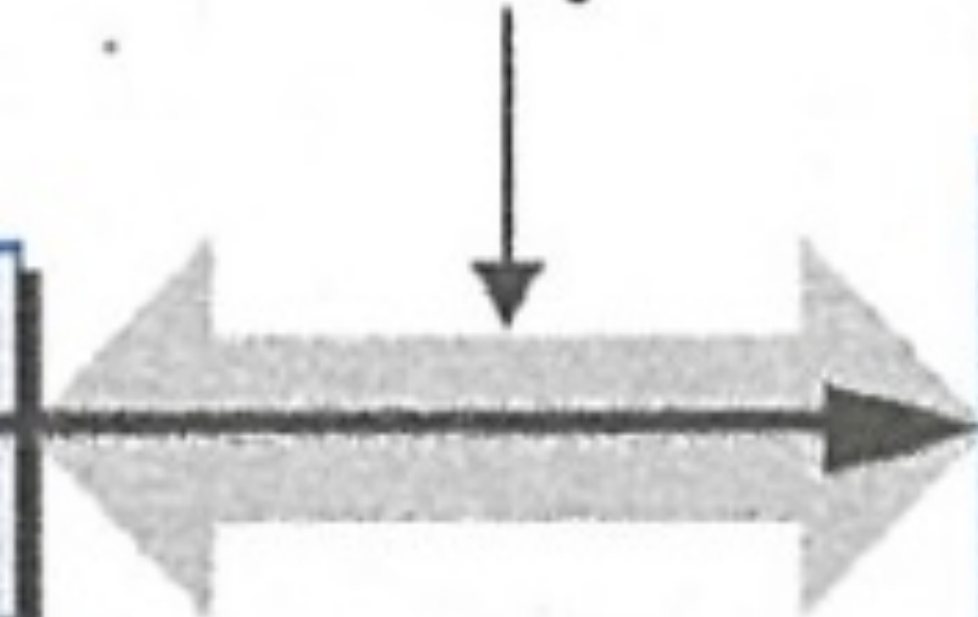
- `%rax` stores the return value
- `%rdi` stores the first parameter to a function
- `%rsi` stores the second parameter to a function
- `%rdx` stores the third parameter to a function
- **`%rip`** stores the address of the next instruction to execute
- `%rsp` stores the address of the current top of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

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

Instructions Are Just Bytes!



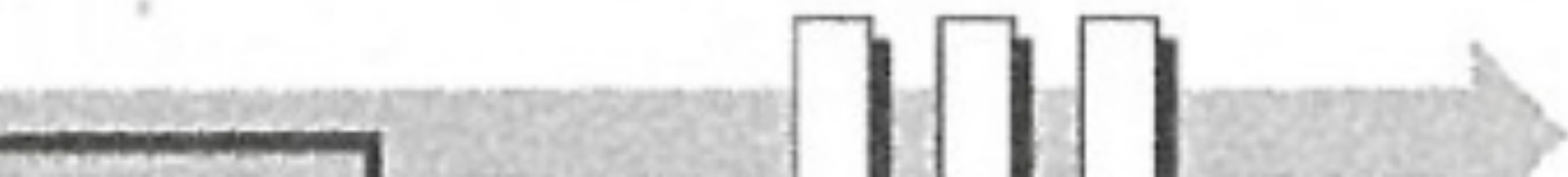
Memory bus



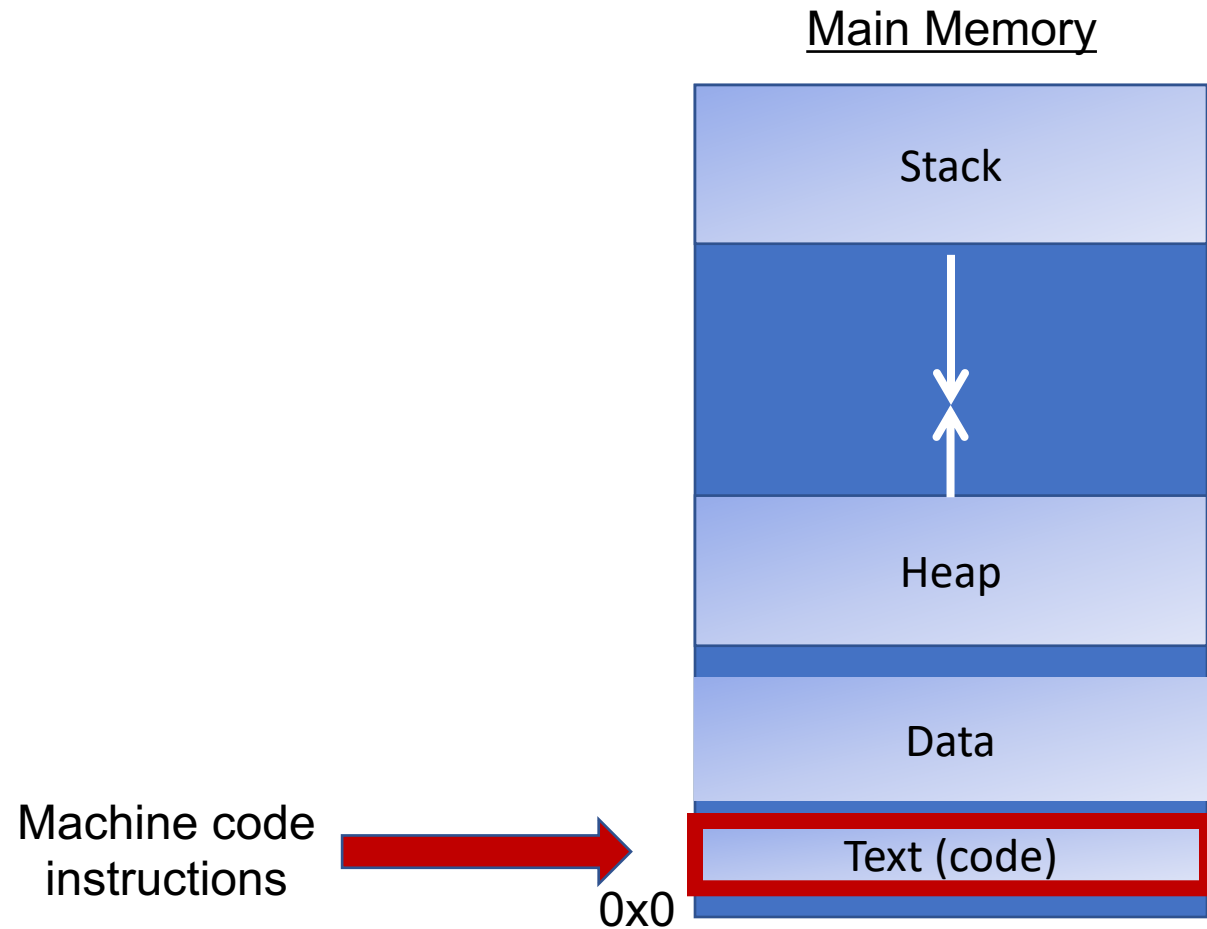
**Main
memory**

"hello, world\n

hello code



Instructions Are Just Bytes!



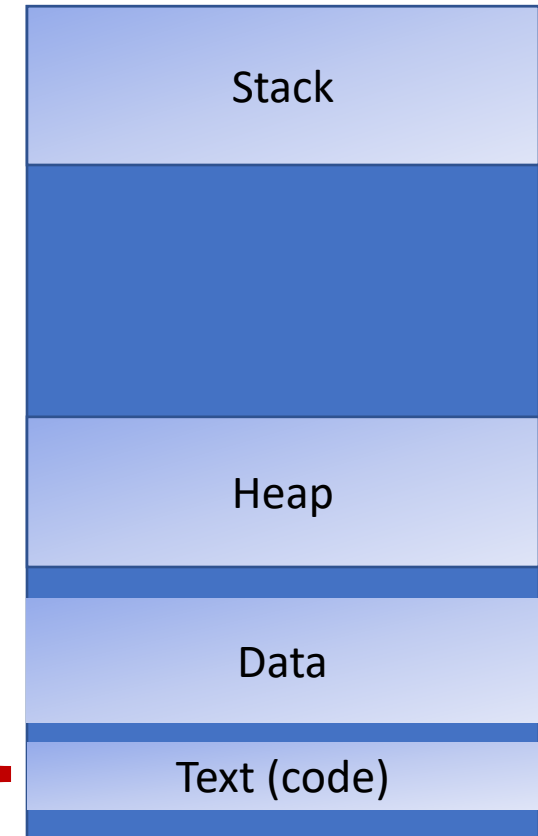
%ori

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

Main Memory



%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>

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

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

%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 += 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>

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

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

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>

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

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

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>

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

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

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>

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

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

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	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

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

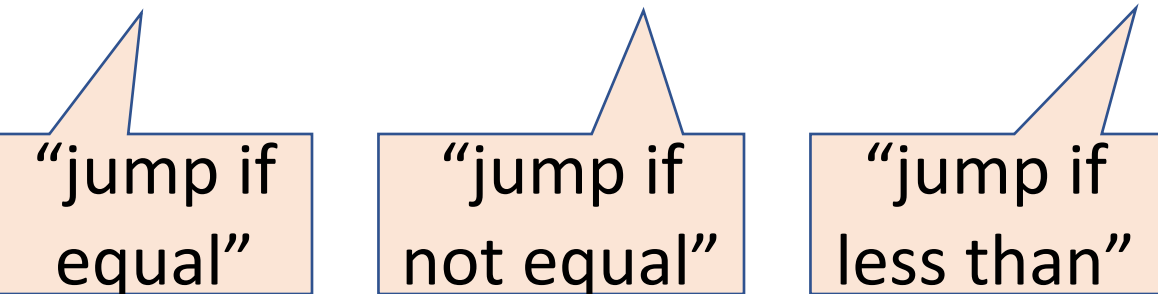
Control

- 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>jl 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 > 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 != 3

cmp \$3, %edi

jne [target]

// Jump if %edi <= 1

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). $(\text{unsigned})\ t < (\text{unsigned})\ a$
- **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
cmpb	Compare byte
cmpw	Compare word
cmpd	Compare double word
cmpq	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
<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>jnb</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)

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.

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`

$S2 \& S1 \neq 0$, 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

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

- A. 4004e1
- B. 0x2
- C. 0xa
- D. 0xc
- E. Other



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

Lecture Plan

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

If Statements

How can we use instructions like **cmp** and *conditional jumps* to implement if statements in assembly?

Practice: Fill In The Blank

<pre>int if_then(int param1) { if (_____) { _____; } return _____; }</pre>	<pre>00000000004004d6 <if_then>: 4004d6: cmp \$0x6,%edi 4004d9: jne 4004de 4004db: add \$0x1,%edi 4004de: lea (%rdi,%rdi,1),%eax 4004e1: retq</pre>
---	---



Practice: Fill In The Blank

<pre>int if_then(int param1) { if (param1 == 6) { param1++; } return param1 * 2; }</pre>	<pre>00000000004004d6 <if_then>: 4004d6: cmp \$0x6,%edi 4004d9: jne 4004de 4004db: add \$0x1,%edi 4004de: lea (%rdi,%rdi,1),%eax 4004e1: retq</pre>
--	---



Common If-Else Construction

If-Else In C

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

```
ret++;
```

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

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

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

Loops and Control Flow

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

0x000000000000400570	<+0>:	mov	\$0x0,%eax
0x000000000000400575	<+5>:	jmp	0x40057a <loop+10>
0x000000000000400577	<+7>:	add	\$0x1,%eax
0x00000000000040057a	<+10>:	cmp	\$0x63,%eax
0x00000000000040057d	<+13>:	jle	0x400577 <loop+7>
0x00000000000040057f	<+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>:

0x0000000000400575 <+5>:

0x0000000000400577 <+7>:

0x000000000040057a <+10>:

0x000000000040057d <+13>:

0x000000000040057f <+15>:

mov \$0x0,%eax

jmp 0x40057a <loop+10>

add \$0x1,%eax

cmp \$0x63,%eax

jle 0x400577 <loop+7>

repz retq

Lecture Plan

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

Common For 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 `j1` instructions doing?
(`j1`: jump less; signed <)



Demo: GDB and Assembly



sum_array.c

`gdb tips`



`layout split` (ctrl-x a: exit,
ctrl-l: resize)

`info reg`

`p $eax`

`p $eflags`

`b *0x400546`

`b *0x400550 if $eax > 98`

`ni`

`si`

View C, assembly, and gdb (lab5)

Print all registers

Print register value

Print all condition codes currently set

Set breakpoint at assembly instruction

Set **conditional breakpoint**

Next assembly instruction

Step into assembly instruction (will step into function calls)

gdb tips



`p/x $rdi`

Print register value in hex

`p/t $rsi`

Print register value in binary

`x $rdi`

Examine the byte stored at this address

`x/4bx $rdi`

Examine 4 bytes starting at this address

`x/4wx $rdi`

Examine 4 ints starting at this address

Lecture Plan

• Assembly Execution and %rip	5
• Control Flow Mechanics	27
• Condition Codes	
• Assembly Instructions	
• If statements	46
• Loops	
• While loops	53
• For loops	66
• Other Instructions That Depend On Condition Codes	72
• Live Session Slides	80

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
```

cmov: Conditional move

Instruction	Synonym	Move Condition
cmovz 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	cmovnl	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)

Last Lab: Conditional Move

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

signed_division:

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

Put $x + 3$ into %eax

Check the sign of x

If x is positive, put x into %eax

Divide %eax by 4

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

80

Live Session Slides

Practice: Fill in the blanks

```
long loop(long a, long b) {  
    long result = ____ (1) ____;  
    while (____ (2) ____) {  
        result = ____ (3) ____;  
        a = ____ (4) ____;  
    }  
    return result;  
}
```

```
<loop>      mov     $0x1,%eax  
<loop+5>    jmp     <loop+0x19>  
<loop+7>    lea     (%rdi,%rsi,1),%rdx  
<loop+11>   mul     %rdx,%rax  
<loop+15>   add     $0x1,%rdi  
<loop+19>   cmp     %rsi,%rdi  
<loop+22>   jl      <loop+0x7>  
<loop+24>   repz    retq
```



Common while loop construction:

Jump to test

Body

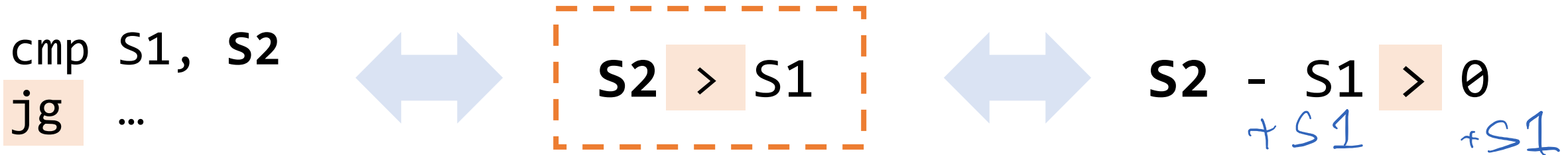
Test

Jump to body if success



★ How to remember cmp/jmp

- `CMP S1, S2` is $S2 - S1$ (just sets condition codes). **But generally:**



- Much less important to remember exact condition codes
 - Yes, they fully explain conditional jmp...
 - ...but more important to know how to translate assembly back into C
 - If you're interested, B&O p. 206 has details

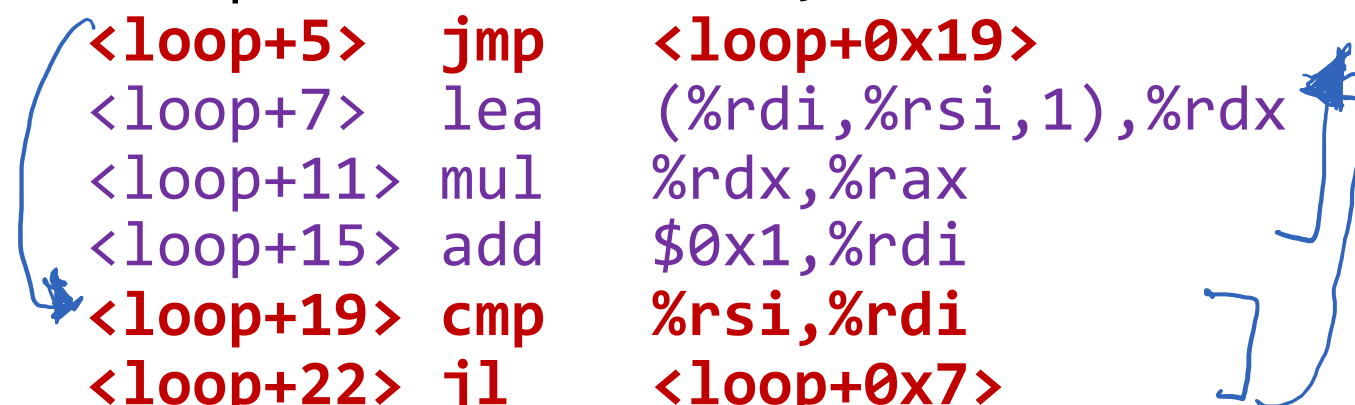
Instruction	Synonym	Jump condition	Description
<code>jmp Label</code>		1	Direct jump
<code>jmp *Operand</code>		1	Indirect jump
<code>je Label</code>	<code>jz</code>	ZF	Equal / zero
<code>jne Label</code>	<code>jnz</code>	$\sim ZF$	Not equal / not zero
<code>js Label</code>		SF	Negative
<code>jns Label</code>		$\sim SF$	Nonnegative
<code>jg Label</code>	<code>jnl</code>	$\sim (SF \wedge OF) \wedge \sim ZF$	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	$\sim (SF \wedge OF)$	Greater or equal (signed >=)
<code>jle Label</code>	<code>jnge</code>	$SF \wedge OF$	Less (signed <)
<code>jle Label</code>	<code>jng</code>	$(SF \wedge OF) \vee ZF$	Less or equal (signed <=)
<code>ja Label</code>	<code>jnb</code>	$\sim CF \wedge \sim ZF$	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	$\sim CF$	Above or equal (unsigned >=)
<code>jb Label</code>	<code>jnae</code>	CF	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	$CF \vee ZF$	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Practice: Fill in the blanks

```
long loop(long a, long b) {  
    long result = ____ (1) ____;  
    while (____ (2) ____) {  
        result = ____ (3) ____;  
        a = ____ (4) ____;  
    }  
    return result;  
}
```

<loop>	mov	\$0x1,%eax
<loop+5>	jmp	<loop+0x19>
<loop+7>	lea	(%rdi,%rsi,1),%rdx
<loop+11>	mul	%rdx,%rax
<loop+15>	add	\$0x1,%rdi
<loop+19>	cmp	%rsi,%rdi
<loop+22>	j1	<loop+0x7>
<loop+24>	repz	retq



Common while loop construction:

Jump to test

Body 

Test

Jump to body if success



Practice: Fill in the blanks

```
long loop(long a, long b) {
    1) long result = 1;
    2) while ( a < b ) {
    3) result = result * (a + b);
    4) a = a + 1;
    }
    return result;
}
```



<loop>	mov	\$0x1,%eax	
<loop+5>	jmp	<loop+0x19>	
			$rdi + rsi = a + b$
<loop+7>	lea	(%rdi,%rsi,1),%rdx	
<loop+11>	mul	%rdx,%rax	$D = D * S$
<loop+15>	add	\$0x1,%rdi	$D = D + 1$
			$b \quad a$
<loop+19>	cmp	%rsi,%rdi	$a < b$
<loop+22>	j1	<loop+0x7>	
<loop+24>	repz	retq	

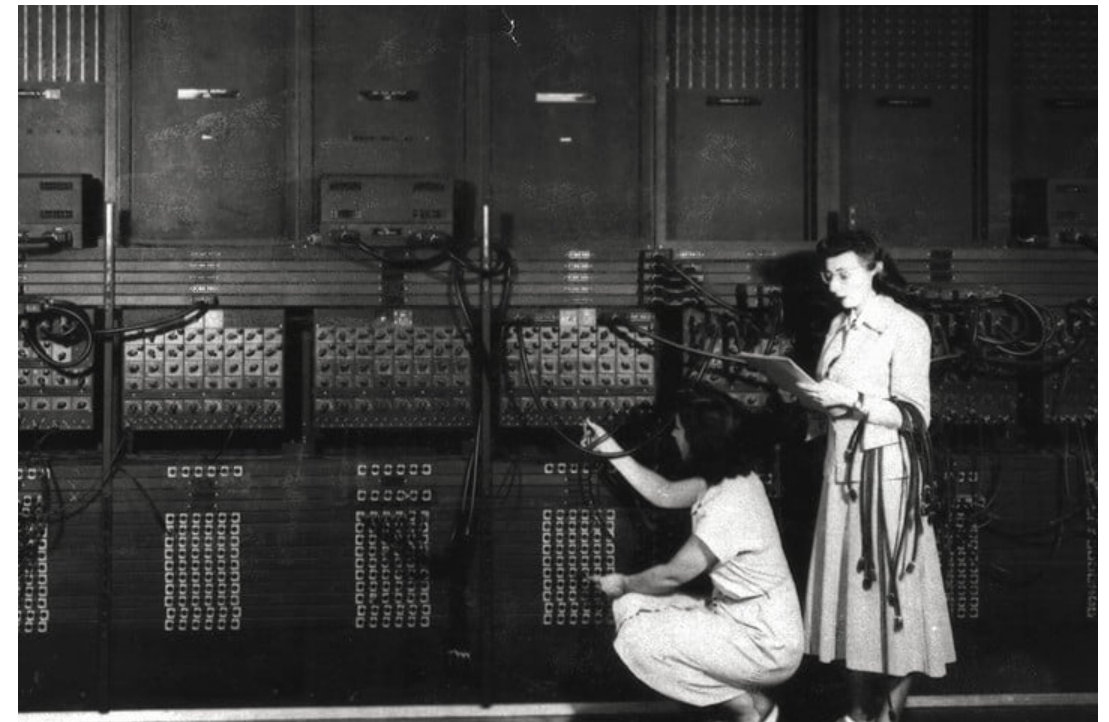
Practice: Fill in the blanks

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

```
<loop>    mov    $0x1,%eax  
<loop+5>  jmp    <loop+0x19>   
  
<loop+7>  lea    (%rdi,%rsi,1),%rdx  
<loop+11> mul    %rdx,%rax  
<loop+15> add    $0x1,%rdi  
  
<loop+19> cmp    %rsi,%rdi  
<loop+22> jl     <loop+0x7>   
  
<loop+24> repz  retq
```

News: ENIAC Day

- Electronic Numerical Integrator And Computer (ENIAC) was the first programmable, general-purpose, Turing-complete computer
- Unveiled at U Penn on February 15, 1946 (celebrated since 2011)
- Fran Bilas, Betty Jennings, Ruth Lichterman, Kay McNulty, Betty Snyder, and Marlyn Wescoff were the first programmers and structure to determine how to wire and manipulate its switches and cables, as programming languages did not yet exist.
- The programmers' efforts were not recognized until many years later, as programming was considered a clerical task



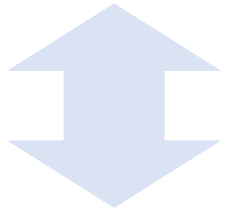
Ester Gerston and Gloria Gordon work on the ENIAC computer

★ Remember test exists

- TEST S1, S2 is S2 & S1

```
test %edi, %edi
```

```
jns ...
```



%edi & %edi is ~~unsigned~~

%edi is ~~unsigned~~

Instruction	Synonym	Jump condition	Description
jmp <i>Label</i>		1	Direct jump
jmp <i>*Operand</i>		1	Indirect jump
j e <i>Label</i>	jz	ZF	Equal / zero
j ne <i>Label</i>	jnz	~ZF	Not equal / not zero
j s <i>Label</i>		SF	Negative
j ns <i>Label</i>		~SF	Nonnegative
j g <i>Label</i>	jnl e	~(SF ^ OF) & ~ZF	Greater (signed >)
j ge <i>Label</i>	jnl	~(SF ^ OF)	Greater or equal (signed >=)
j l <i>Label</i>	jnge	SF ^ OF	Less (signed <)
j le <i>Label</i>	jng	(SF ^ OF) ZF	Less or equal (signed <=)
j a <i>Label</i>	jnb e	~CF & ~ZF	Above (unsigned >)
j ae <i>Label</i>	jnb	~CF	Above or equal (unsigned >=)
j b <i>Label</i>	jna e	CF	Below (unsigned <)
j be <i>Label</i>	jna	CF ZF	Below or equal (unsigned <=)

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

test practice: What's the C code?

0x400546	<test_func>	test	%edi,%edi
0x400548	<test_func+2>	jns	0x400550 <test_func+10>
0x40054a	<test_func+4>	mov	\$0xf eed,%eax
0x40054f	<test_func+9>	retq	
0x400550	<test_func+10>	mov	\$0xaabbccdd,%eax
0x400555	<test_func+15>	retq	



test practice: What's the C code?

```
0x400546 <test_func> test
0x400548 <test_func+2> jns
0x40054a <test_func+4> mov
0x40054f <test_func+9> retq
0x400550 <test_func+10> mov
0x400555 <test_func+15> retq
```

```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;
}
```

```
test    %edi,%edi
jns     0x400550 <test_func+10>
mov     $0xfeed,%eax
retq
mov     $0xaabbccdd,%eax
retq
```

X-X
(or anything
like this)

0
1
return 0xaabbccdd
return 0xfeed

1 1 1 0 0 0
1 1 1 0 0 0
1 1 1 0 0 0

SF ZF OF

Practice: "Escape Room"

<escape_room>	lea	(%rdi,%rdi,1),%eax
<escape_room+3>	cmpl	\$0x5,%eax
<escape_room+6>	jg	<escape_room+19>
<escape_room+8>	cmpl	\$0x1,%edi
<escape_room+11>	jne	<escape_room+25>
<escape_room+13>	movl	\$0x1,%eax
<escape_room+18>	retq	←
<escape_room+19>	movl	\$0x1,%eax
<escape_room+24>	retq	←
<escape_room+25>	movl	\$0x0,%eax
<escape_room+30>	retq	×

What must be passed to the escape^{Room} function such that it returns true (1) and not false (0)?

You don't have to reverse-engineer C code exactly!

Just figure out the big picture!

Practice: "Escape Room"

```

<escape_room>      lea
<escape_room+3>     [cmpl
<escape_room+6>     ]jg
<escape_room+8>     [cmpl
<escape_room+11>    ]jne
<escape_room+13>    [movl
<escape_room+18>    ]retq
<escape_room+19>    [movl
<escape_room+24>    ]retq
<escape_room+25>    [movl
<escape_room+30>    ]retq
    
```

$2x > 5 \checkmark +19, +24$

~~(%rdi,%rdi,1),%eax~~
~~\$0x5,%eax~~ $2x > 5$
 <escape_room+19>
~~\$0x1,%edi~~ $x \neq 1$
 <escape_room+25>
 \$0x1,%eax
 \$0x1,%eax
 \$0x0,%eax

$x \neq 1 \otimes +25, +30$

What must be passed to the escape^{Room}~~Room~~ function such that it returns true (1) and not false (0)?

$x == 1 \checkmark +13, +18$

First param > 2 or $== 1$.