



CS107, Lecture 18

Assembly: Control Flow

Reading: B&O 3.6

Ed Discussion: <https://edstem.org/us/courses/65949/discussion/5619681>

Learning Goals

- Understand how assembly implements loops and control flow
- Learn about how assembly stores comparison and operation results in 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 and main 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**, stored in %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



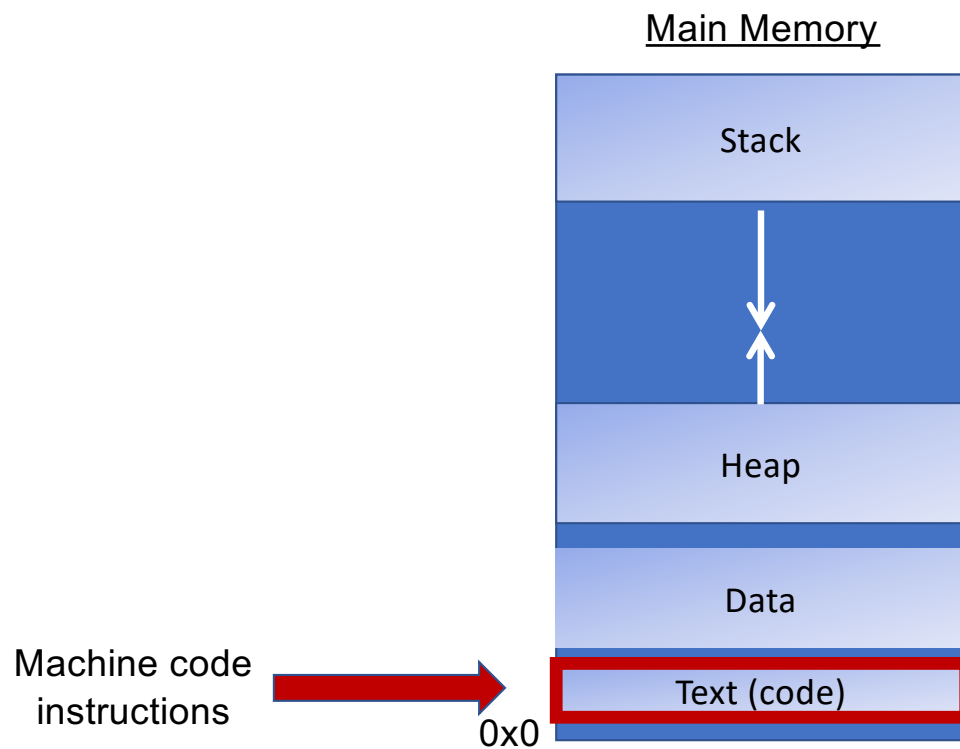
Register Responsibilities

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!



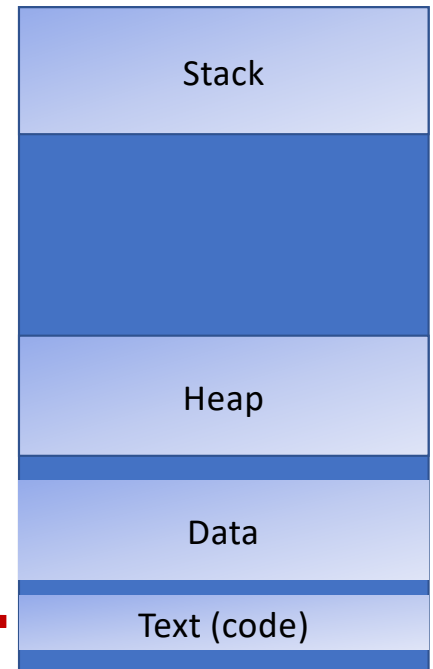
%orig

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

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>

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>

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

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

0x4004fc

%rip

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

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>

Special hardware sets the program counter
to the next instruction:

%rip += size of bytes of current instruction

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

Going In Circles

- How can we use this representation of execution to represent e.g., a **loop**?
- **Key Idea:** we can override what **%rip** stores and populate it with the address of 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>

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>

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>

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 (an "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**?

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 some condition is true, executing other statements if that 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

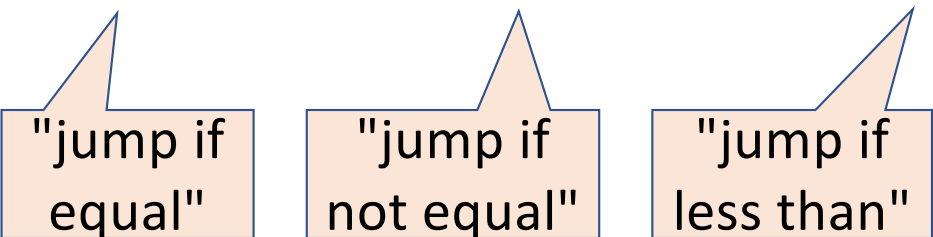
Control

- 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 variants of **jmp** that branch if and only if certain conditions are met. The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
<i>jz Label</i>	<i>jz</i>	Equal / zero
<i>jne Label</i>	<i>jnz</i>	Not equal / not zero
<i>js Label</i>		Negative
<i>jns Label</i>		Nonnegative
<i>jg Label</i>	<i>jnle</i>	Greater (signed >)
<i>jge Label</i>	<i>jnl</i>	Greater or equal (signed >=)
<i>jl Label</i>	<i>jnge</i>	Less (signed <)
<i>jle Label</i>	<i>jng</i>	Less or equal (signed <=)
<i>ja Label</i>	<i>jnb</i>	Above (unsigned >)
<i>jae Label</i>	<i>jnb</i>	Above or equal (unsigned >=)
<i>jb Label</i>	<i>jnae</i>	Below (unsigned <)
<i>jbe Label</i>	<i>jna</i>	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 == 4
```

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

```
je [target]
```

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

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

```
jne [target]
```

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

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

```
jle [target]
```

Wait a minute – how do jump instructions know anything about the comparisons of earlier instructions?

Control

- The CPU has special registers called *condition codes* that act as "global variables". They automatically track 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 information about the most recent arithmetic or logical operation.

Most common condition codes:

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

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]
```

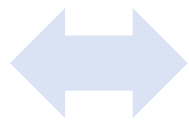
```
// Jump if %edi == 4
// calculates %edi - 4
cmp $4, %edi
je [target]
```

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

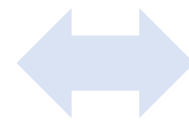
★ How to remember cmp/jmp

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

`cmp S1, S2`
`jg ...`



`S2 > S1`



`S2 - S1 > 0`

Conditional Jumps

Conditional jumps look at a relevant subset of the condition codes to determine whether to branch or fall through without jumping.

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

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

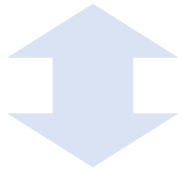
Cool trick: if we pass the same value for both operands, we can check the sign of that value using the **Sign Flag** and **Zero Flag** condition codes!

The test Instruction

- TEST S1, S2 is S2 & S1

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

```
jns ...
```



%edi & %edi is nonnegative

%edi is nonnegative

Condition Codes

- Previously discussed arithmetic and logical instructions update these flags. **lea** does not (it's intended only for address computation and nothing else).
- 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

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>0000000000401126 <if_then>: 401126: cmp \$0x6,%edi 401129: je 40112f 40112b: lea (%rdi,%rdi,1), %eax 40112e: retq 40112f: add \$0x1,%edi 401132: jmp 40112b</pre>
---	---



Practice: Fill In The Blank

<pre>int if_then(int param1) { if (param1 == 6) { param1++; } return param1 * 2; }</pre>	<pre>0000000000401126 <if_then>: 401126: cmp \$0x6,%edi 401129: je 40112f 40112b: lea (%rdi,%rdi,1), %eax 40112e: retq 40112f: add \$0x1,%edi 401132: jmp 40112b</pre>
--	---



Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {  
    long result;  
    if (_____) {  
  
        _____ ;  
    } else {  
  
        _____ ;  
    }  
  
    return result;  
}
```

```
401134 <+0>:  mov    %rsi,%rax  
401137 <+3>:  cmp    %rsi,%rdi  
40113a <+6>:  jge    0x401140 <absdiff+12>  
40113c <+8>:  sub    %rdi,%rax  
40113f <+11>: retq  
401140 <+12>: sub    %rsi,%rdi  
401143 <+15>: mov    %rdi,%rax  
401146 <+18>: retq
```

If-Else In Assembly pseudocode

Check opposite of code condition

Jump to else-body if test passes

If-body

Jump to past else-body

Else-body

Past else body



Practice: Fill in the Blank

If-Else In C

```
long absdiff(long x, long y) {  
    long result;  
    if ( x < y ) {  
        result = y - x ;  
    } else {  
        result = x - y ;  
    }  
    return result;  
}
```

```
401134 <+0>:  mov    %rsi,%rax  
401137 <+3>:  cmp     %rsi,%rdi  
40113a <+6>:  jge     0x401140 <absdiff+12>  
40113c <+8>:  sub     %rdi,%rax  
40113f <+11>: retq  
401140 <+12>: sub     %rsi,%rdi  
401143 <+15>: mov     %rdi,%rax  
401146 <+18>: retq
```

If-Else In Assembly pseudocode

Check opposite of code condition

Jump to else-body if test passes

If-body

Jump to past else-body

Else-body

Past else body

If-Else Construction Variations

C Code

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

Assembly

```
401134 <+0>:  cmp    $0x3,%edi  
401137 <+3>:  jle     0x401142 <test+14>  
401139 <+5>:  mov     $0xa,%eax  
40113e <+10>:  add     $0x1,%eax  
401141 <+13>:  retq  
401142 <+14>:  mov     $0x0,%eax  
401147 <+19>:  jmp     0x40113e <test+10>
```

Loops and Control Flow

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

```
0x00000000000040115c <+0>:    mov    $0x0,%eax  
0x000000000000401161 <+5>:    cmp    $0x63,%eax  
0x000000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x000000000000401166 <+10>:   add    $0x1,%eax  
0x000000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x00000000000040116b <+15>:   retq
```

Loops and Control Flow

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

0x00000000000040115c	<+0>:	mov	\$0x0,%eax
0x000000000000401161	<+5>:	cmp	\$0x63,%eax
0x000000000000401164	<+8>:	jg	0x40116b <loop+15>
0x000000000000401166	<+10>:	add	\$0x1,%eax
0x000000000000401169	<+13>:	jmp	0x401161 <loop+5>
0x00000000000040116b	<+15>:	retq	

Set %eax (i) to 0.

Loops and Control Flow

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

0x00000000000040115c <+0>:	mov	\$0x0,%eax
0x000000000000401161 <+5>:	cmp	\$0x63,%eax
0x000000000000401164 <+8>:	jg	0x40116b <loop+15>
0x000000000000401166 <+10>:	add	\$0x1,%eax
0x000000000000401169 <+13>:	jmp	0x401161 <loop+5>
0x00000000000040116b <+15>:	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++;  
    }  
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax  
0x0000000000401161 <+5>:    cmp    $0x63,%eax  
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x0000000000401166 <+10>:   add    $0x1,%eax  
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x000000000040116b <+15>:   retq
```

jg means "jump if greater than". This jumps if `%eax > 0x63`. The flags indicate this is false, so we do not jump.

Loops and Control Flow

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

```
0x00000000000040115c <+0>:    mov    $0x0,%eax  
0x000000000000401161 <+5>:    cmp    $0x63,%eax  
0x000000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x000000000000401166 <+10>:   add    $0x1,%eax  
0x000000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x00000000000040116b <+15>:   retq
```

Add 1 to %eax (i).

Loops and Control Flow

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

```
0x0000000000040115c <+0>:    mov    $0x0,%eax  
0x00000000000401161 <+5>:    cmp    $0x63,%eax  
0x00000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x00000000000401166 <+10>:   add    $0x1,%eax  
0x00000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x0000000000040116b <+15>:   retq
```

Jump to another instruction.

Loops and Control Flow

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

0x00000000000040115c <+0>:	mov	\$0x0,%eax
0x000000000000401161 <+5>:	cmp	\$0x63,%eax
0x000000000000401164 <+8>:	jg	0x40116b <loop+15>
0x000000000000401166 <+10>:	add	\$0x1,%eax
0x000000000000401169 <+13>:	jmp	0x401161 <loop+5>
0x00000000000040116b <+15>:	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++;  
    }  
}
```

```
0x000000000040115c <+0>:    mov    $0x0,%eax  
0x0000000000401161 <+5>:    cmp    $0x63,%eax  
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x0000000000401166 <+10>:   add    $0x1,%eax  
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x000000000040116b <+15>:   retq
```

We continue in this pattern until we make this conditional jump. When will that be?

Loops and Control Flow

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

0x0000000000040115c	<+0>:	mov	\$0x0,%eax
0x00000000000401161	<+5>:	cmp	\$0x63,%eax
0x00000000000401164	<+8>:	jg	0x40116b <loop+15>
0x00000000000401166	<+10>:	add	\$0x1,%eax
0x00000000000401169	<+13>:	jmp	0x401161 <loop+5>
0x0000000000040116b	<+15>:	retq	

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

Loops and Control Flow

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

```
0x000000000040115c <+0>:    mov    $0x0,%eax  
0x0000000000401161 <+5>:    cmp    $0x63,%eax  
0x0000000000401164 <+8>:    jg     0x40116b <loop+15>  
0x0000000000401166 <+10>:   add    $0x1,%eax  
0x0000000000401169 <+13>:   jmp    0x401161 <loop+5>  
0x000000000040116b <+15>:   retq
```

Then, we return from the function.

GCC Common While Loop Construction

C

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

Assembly

Check *opposite of code condition*
Skip loop if test passes
Body
Jump back to test

From Previous Slide:

0x000000000040115c <+0>:	mov	\$0x0,%eax
0x0000000000401161 <+5>:	cmp	\$0x63,%eax
0x0000000000401164 <+8>:	jg	0x40116b <loop+15>
0x0000000000401166 <+10>:	add	\$0x1,%eax
0x0000000000401169 <+13>:	jmp	0x401161 <loop+5>
0x000000000040116b <+15>:	retq	

GCC Other While Loop Construction

C

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

Assembly

Jump to check

Body

Check code condition

Jump to body if test passes

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	