CS107, Lecture 12
Control Flow: When in doubt just JMP!

Reading: B&O 3.1-3.4
Learning Goals

• Learn about how assembly stores comparison and operation results in condition codes
• Understand how assembly implements loops and control flow
• 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
• Live Session Slides
Lecture Plan

• Assembly Execution and %rip
  5
• Control Flow Mechanics
  27
  • Condition Codes
  • Assembly Instructions
• If statements
  46
• Loops
  54
  • While loops
  • For loops
  67
• Other Instructions That Depend On Condition Codes
  73
• Live Session Slides
  81
What does it mean for a program to execute?
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.

<table>
<thead>
<tr>
<th>4004fd</th>
<th>fa</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004fc</td>
<td>eb</td>
</tr>
<tr>
<td>4004fb</td>
<td>01</td>
</tr>
<tr>
<td>4004fa</td>
<td>fc</td>
</tr>
<tr>
<td>4004f9</td>
<td>45</td>
</tr>
<tr>
<td>4004f8</td>
<td>83</td>
</tr>
<tr>
<td>4004f7</td>
<td>00</td>
</tr>
<tr>
<td>4004f6</td>
<td>00</td>
</tr>
<tr>
<td>4004f5</td>
<td>00</td>
</tr>
<tr>
<td>4004f4</td>
<td>00</td>
</tr>
<tr>
<td>4004f3</td>
<td>fc</td>
</tr>
<tr>
<td>4004f2</td>
<td>45</td>
</tr>
<tr>
<td>4004f1</td>
<td>c7</td>
</tr>
<tr>
<td>4004f0</td>
<td>e5</td>
</tr>
<tr>
<td>4004ef</td>
<td>89</td>
</tr>
<tr>
<td>4004ee</td>
<td>48</td>
</tr>
<tr>
<td>4004ed</td>
<td>55</td>
</tr>
</tbody>
</table>
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 Just Bytes!

Machine code instructions

Main Memory

0x0

Stack

Heap

Data

Text (code)
0000000000004004ed <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 <loop>:
push %rbp
mov %rsp,%rbp
movl $0x0,-0x4(%rbp)
addl $0x1,-0x4(%rbp)
jmp 4004f8 <loop+0xb>
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004ed</td>
<td>55</td>
</tr>
<tr>
<td>0x4004ee</td>
<td>48 89 e5</td>
</tr>
<tr>
<td>0x4004f1</td>
<td>c7 45 fc 00 00 00 00</td>
</tr>
<tr>
<td>0x4004f8</td>
<td>83 45 fc 01</td>
</tr>
<tr>
<td>0x4004fc</td>
<td>eb fa</td>
</tr>
<tr>
<td>0x4004fd</td>
<td>fa</td>
</tr>
<tr>
<td>0x4004fc</td>
<td>eb</td>
</tr>
<tr>
<td>0x4004fb</td>
<td>01</td>
</tr>
<tr>
<td>0x4004fa</td>
<td>fc</td>
</tr>
<tr>
<td>0x4004f9</td>
<td>45</td>
</tr>
<tr>
<td>0x4004f8</td>
<td>83</td>
</tr>
<tr>
<td>0x4004f7</td>
<td>00</td>
</tr>
<tr>
<td>0x4004f6</td>
<td>00</td>
</tr>
<tr>
<td>0x4004f5</td>
<td>00</td>
</tr>
<tr>
<td>0x4004f4</td>
<td>00</td>
</tr>
<tr>
<td>0x4004f3</td>
<td>fc</td>
</tr>
<tr>
<td>0x4004f2</td>
<td>45</td>
</tr>
<tr>
<td>0x4004f1</td>
<td>c7</td>
</tr>
<tr>
<td>0x4004f0</td>
<td>e5</td>
</tr>
<tr>
<td>0x4004ef</td>
<td>89</td>
</tr>
<tr>
<td>0x4004ee</td>
<td>48</td>
</tr>
<tr>
<td>0x4004ed</td>
<td>55</td>
</tr>
</tbody>
</table>
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
The program counter (PC), known as `%rip` in x86-64, stores the address in memory of the next instruction to be executed.
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
Special hardware sets the program counter to the next instruction:
\[ \%rip += \text{size of bytes of current instruction} \]
\[ 0x4004fc \]
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!
The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

```assembly
0x4004ed <loop>:
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).

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed: 55</td>
<td>push %rbp</td>
</tr>
<tr>
<td>4004ee: 48 89 e5</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>4004f1: c7 45 fc 00 00 00 00</td>
<td>movl $0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004f8: 83 45 fc 01</td>
<td>addl $0x1,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004fc: eb fa</td>
<td>jmp 4004f8 &lt;loop+0xb&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed</td>
<td>55</td>
<td>push</td>
<td>%rbp</td>
</tr>
<tr>
<td>4004ee</td>
<td>48 89 e5</td>
<td>mov</td>
<td>%rsp,%rbp</td>
</tr>
<tr>
<td>4004f1</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl</td>
<td>$0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004f8</td>
<td>83 45 fc 01</td>
<td>addl</td>
<td>$0x1,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004fc</td>
<td>eb fa</td>
<td>jmp</td>
<td>4004f8 &lt;loop+0xb&gt;</td>
</tr>
</tbody>
</table>

0x4004fc

%rip
The `jmp` instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).
This assembly represents an infinite loop in C!

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed</td>
<td>55</td>
<td>push %rbp</td>
</tr>
<tr>
<td>4004ee</td>
<td>48 89 e5</td>
<td>mov %rsp, %rbp</td>
</tr>
<tr>
<td>4004f1</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl $0x0, -0x4(%rbp)</td>
</tr>
<tr>
<td>4004f8</td>
<td>83 45 fc 01</td>
<td>addl $0x1, -0x4(%rbp)</td>
</tr>
<tr>
<td>4004fc</td>
<td>eb fa</td>
<td>jmp 4004f8 &lt;loop+0xb&gt;</td>
</tr>
<tr>
<td>4004fd</td>
<td>fa</td>
<td></td>
</tr>
<tr>
<td>4004fc</td>
<td>eb</td>
<td></td>
</tr>
<tr>
<td>4004fb</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>4004fa</td>
<td>fc</td>
<td></td>
</tr>
<tr>
<td>4004f9</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>4004f8</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>4004f7</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>4004f6</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>4004f5</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>4004f4</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>4004f3</td>
<td>fc</td>
<td></td>
</tr>
<tr>
<td>4004f2</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>4004f1</td>
<td>c7</td>
<td></td>
</tr>
<tr>
<td>4004f0</td>
<td>e5</td>
<td></td>
</tr>
<tr>
<td>4004ef</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>4004ee</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>4004ed</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

This assembly represents an infinite loop in C!
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
```
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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Execution and <code>%rip</code></td>
<td>5</td>
</tr>
<tr>
<td><strong>Control Flow Mechanics</strong></td>
<td>27</td>
</tr>
<tr>
<td>• Condition Codes</td>
<td></td>
</tr>
<tr>
<td>• Assembly Instructions</td>
<td></td>
</tr>
<tr>
<td>If statements</td>
<td>46</td>
</tr>
<tr>
<td>Loops</td>
<td></td>
</tr>
<tr>
<td>• While loops</td>
<td>54</td>
</tr>
<tr>
<td>• For loops</td>
<td>67</td>
</tr>
<tr>
<td>Other Instructions That Depend On Condition Codes</td>
<td>73</td>
</tr>
<tr>
<td>Live Session Slides</td>
<td>81</td>
</tr>
</tbody>
</table>
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?
if (x > y) {
    // a
    // a
} else {
    // b
    // b
}

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je Label</code></td>
<td><code>jz</code></td>
<td>Equal / zero</td>
</tr>
<tr>
<td><code>jne Label</code></td>
<td><code>jnz</code></td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td><code>js Label</code></td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td><code>jns Label</code></td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>jg Label</code></td>
<td><code>jnle</code></td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td><code>jge Label</code></td>
<td><code>jnl</code></td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td><code>jl Label</code></td>
<td><code>jnge</code></td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td><code>jle Label</code></td>
<td><code>jng</code></td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td><code>ja Label</code></td>
<td><code>jnbe</code></td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td><code>jae Label</code></td>
<td><code>jnb</code></td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td><code>jb Label</code></td>
<td><code>jnae</code></td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td><code>jbe Label</code></td>
<td><code>jna</code></td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Read **cmp S1,S2** as “*compare S2 to S1*”:

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

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

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

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

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

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

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

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

Read \texttt{cmp S1,S2} as \textit{“compare S2 to S1”}:

// Jump if %edi > 2
\texttt{cmp $2, %edi}
\texttt{jg [target]}

// Jump if %edi < 2
\texttt{cmp $3, %edi}
\texttt{jne [target]}

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

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

• What exactly are the condition codes? How do they store this information?
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.
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).  
  \((a < 0 == b < 0) \&\& (t < 0 != a < 0)\)
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
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpb</td>
<td>Compare byte</td>
</tr>
<tr>
<td>cmpw</td>
<td>Compare word</td>
</tr>
<tr>
<td>cmpl</td>
<td>Compare double word</td>
</tr>
<tr>
<td>cmpq</td>
<td>Compare quad word</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Label</td>
<td>jz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>js Label</td>
<td></td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>jns Label</td>
<td></td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>jg Label</td>
<td>jnlle</td>
<td>Greater (signed &gt;) (ZF = 0 and SF = OF)</td>
</tr>
<tr>
<td>jge Label</td>
<td>jnl</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF! = OF)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testb</td>
<td>Test byte</td>
</tr>
<tr>
<td>testw</td>
<td>Test word</td>
</tr>
<tr>
<td>testl</td>
<td>Test double word</td>
</tr>
<tr>
<td>testq</td>
<td>Test quad word</td>
</tr>
</tbody>
</table>

**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!
• Previously-discussed arithmetic and logical instructions update these flags. \texttt{lea} does not (it was intended only for address computations).
• Logical operations (\texttt{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, \texttt{inc} and \texttt{dec} set the overflow and zero flags, but leave the carry flag unchanged.
Exercise 1: Conditional jump

je target  \hspace{1cm} \textit{jump if ZF is 1}

Let \%edi store 0x10. Will we jump in the following cases? \hspace{1cm} \%edi \hspace{1cm} 0x10

1. \texttt{cmp} $0x10,\%edi  \\
   \hspace{1cm} \texttt{je} 40056f  \\
   \hspace{1cm} \texttt{add} $0x1,\%edi

2. \texttt{test} $0x10,\%edi  \\
   \hspace{1cm} \texttt{je} 40056f  \\
   \hspace{1cm} \texttt{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?  
Assume they are run in order.

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
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

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
• 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
• Live Session Slides
If Statements

How can we use instructions like `cmp` and *conditional jumps* to implement if statements in assembly?
int if_then(int param1) {
    if ( __________ ) {
        __________;
    }
    return __________;
}

00000000000401126 <if_then>:
  401126:  cmp  $0x6,%edi
  401129:  je   40112f
  40112b:  lea  (%rdi,%rdi,1),%eax
  40112e:  retq
  40112f:  add  $0x1,%edi
  401132:  jmp  40112b
int if_then(int param1) {
    if (param1 == 6) {
        param1++;
    }
    return param1 * 2;
}

00000000000401126 <if_then>:
    401126: cmp $0x6,%edi
    401129: je 40112f
    40112b: lea (%rdi,%rdi,1),%eax
    40112e: retq
    40112f: add $0x1,%edi
    401132: jmp 40112b
Common If-Else Construction

**If-Else In C**

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

**If-Else In Assembly pseudocode**

```
Test
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

```c
long absdiff(long x, long y) {
    long result;
    if (integer_expression) {
        expression1;
    } else {
        expression2;
    }
    return result;
}
```

If-Else In Assembly pseudocode

Test
Jump to else-body if test **passes**
**If-body**
Jump to past else-body
**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;
}

If-Else In Assembly pseudocode
Test
Jump to else-body if test passes
If-body
Jump to past else-body
Else-body
Past else body

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 Construction Variations

C Code

```c
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>
```
Lecture Plan

• Assembly Execution and %rip 5
• Control Flow Mechanics 27
  • Condition Codes
  • Assembly Instructions
• If statements 46

• Loops 54
  • While loops 67
  • For loops
• Other Instructions That Depend On Condition Codes 73
• Live Session Slides 81
Loops and Control Flow

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

```assembly
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
```
Loops and Control Flow

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

```assembly
%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

Set %eax (i) to 0.
```
Loops and Control Flow

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

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

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

```assembly
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 += 1;
    }
}

Add 1 to %eax (i).
Loops and Control Flow

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

Jump to another instruction.

```
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
```
Loops and Control Flow

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

Compare `%eax` (i) to 0x63 (99) by calculating `%eax – 0x63`. This is 1 - 99 = -98, so it sets the Sign Flag to 1.
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

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

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

We will stop looping when this comparison says that %eax – 0x63 > 0!
Loops and Control Flow

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

Then, we return from the function.
GCC Common While Loop Construction

C
while (test) {
  body
}

Assembly
Test
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 test
Body
Test
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
Lecture Plan

• Assembly Execution and %rip 5
• Control Flow Mechanics 27
  • Condition Codes
  • Assembly Instructions
• If statements 46

• Loops 54
  • While loops
  • For loops 67
• Other Instructions That Depend On Condition Codes 73
• Live Session Slides 81
Common For Loop Construction

C For loop

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

Assembly pseudocode

- **Init**
- **Test**
- **Skip loop if test passes**
- **Body**
- **Update**
- **Jump back to test**

C Equivalent While Loop

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

For loops and while loops are treated (essentially) the same when compiled down to assembly.
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}

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 jge instructions doing? (jge: signed jump greater than/equal)
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)
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
<table>
<thead>
<tr>
<th>Lecture Plan</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Assembly Execution and <code>%rip</code></td>
<td>5</td>
</tr>
<tr>
<td>• Control Flow Mechanics</td>
<td>27</td>
</tr>
<tr>
<td>• Condition Codes</td>
<td></td>
</tr>
<tr>
<td>• Assembly Instructions</td>
<td></td>
</tr>
<tr>
<td>• If statements</td>
<td>46</td>
</tr>
<tr>
<td>• Loops</td>
<td></td>
</tr>
<tr>
<td>• While loops</td>
<td>54</td>
</tr>
<tr>
<td>• For loops</td>
<td>67</td>
</tr>
<tr>
<td>• Other Instructions That Depend On Condition Codes</td>
<td>73</td>
</tr>
<tr>
<td>• Live Session Slides</td>
<td>81</td>
</tr>
</tbody>
</table>
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
- Operand 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

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

```asm
    cmp $0xf,%edi
    setle %al
    movzbl %al, %eax
    retq
```
## set: Read condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition (1 if true, 0 if false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>setns D</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnle</td>
<td>Greater (signed $&gt;$)</td>
</tr>
<tr>
<td>setge D</td>
<td>setnl</td>
<td>Greater or equal (signed $\geq$)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnge</td>
<td>Less (signed $&lt;$)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>Less or equal (signed $\leq$)</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>Above (unsigned $&gt;$)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>Above or equal (unsigned $\geq$)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnaae</td>
<td>Below (unsigned $&lt;$)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>Below or equal (unsigned $\leq$)</td>
</tr>
</tbody>
</table>
cmov: Conditional move

\texttt{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;

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

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

76
# cmov: Conditional move

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

- 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
• TEST $S1, S2$ is $S2 \& S1$

test %edi, %edi

jns ...
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
        result = ___(3)___;
        a = ___(4)___;
    }
    return result;
}

GCC common while loop construction:
Test
Jump past loop if fails
Body
Jump to test

https://godbolt.org/z/zrW6c5MGa
long loop(long a, long b) {
    long result = ___(1)___;
    while (___(2)___) {
        result = ___(3)___;
        a = ___(4)___;
    }
    return result;
}

GCC common while loop construction:
Test
Jump past loop if fails
Body
Jump to test
long loop(long a, long b) {
    long result = ________;
    while (_______) {
        result = __________;
        a = ________;
    }
    return result;
}
long loop(long a, long b) {
    long result = 1;
    while (a < b) {
        result = result*(a+b);
        a = a + 1;
    }
    return result;
}

<+0>:  mov  $0x1,%eax
<+5>:  cmp  %rsi,%rdi
<+8>:  jge  0x1151 <loop+24>
<+10>: lea  (%rdi,%rsi,1),%rdx
<+14>: imul  %rdx,%rax
<+18>: add  $0x1,%rdi
<+22>: jmp  0x113e <loop+5>
<+24>: retq
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  $0xfeed,%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 %edi,%edi
0x400548 <test_func+2>  jns 0x400550 <test_func+10>
0x40054a <test_func+4>  mov $0xfeed,%eax
0x40054f <test_func+9>  retq
0x400550 <test_func+10> mov $0xaabbccdd,%eax
0x400555 <test_func+15> retq

int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;  \ (or anything like this)
}
```
Practice: “Escape Room”

```asm
le %rdi,%rdi,1,eax
cmp $0x5,eax
jg 0x114c <escape_room+19>

cmp $0x1,edi
je 0x1152 <escape_room+25>

mov $0x0,eax
retq

mov $0x1,eax
retq

mov $0x1,eax
retq
```

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

You don’t have to reverse-engineer C code exactly!
Practice: “Escape Room”

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

First param > 2 or == 1.
Lecture Plan

• Revisiting %rip 5
• Calling Functions 19
  • The Stack 22
  • Passing Control 36
  • Passing Data 44
  • Local Storage 65
• Register Restrictions 69
• Pulling it all together: recursion example 78
• Optimizations 81
• Live session slides 93