Lecture #22 – Control Flow
Good news, everyone!

• Assignment 4 will be out on Nov 27 (due Dec 11)
• Midterm exam will be held on December 5, 2020 (Saturday) at 11:45am
• More info on Friday
Learning Goals

• Learn about how assembly stores comparison and operation results in condition codes

• Understand how assembly implements loops and control flow
Plan for Today

• Assembly Execution and `%rip` (cont’d.)
• Control Flow Mechanics
• If statements

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

• Assembly Execution and %rip (cont’d.)
• Control Flow Mechanics
• If statements
Recap: Executing Instructions

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

Today:
• **Who controls the instructions?** How do we know what to do now or next?

Answer:
• The **program counter** (PC), %rip.
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!

Figure 1.6 Loading the executable from disk into main memory.

Figure 1.7 Writing the output string from memory to the display.
Memory bus

Main memory

“hello, world\n”

hello code
Instructions Are Just Bytes!

Main Memory

Stack

Heap

Data

Text (code)

Machine code instructions 0x0
%rip

00000000004004ed <loop>:
4004ed: 55                    push %rbp
4004ee: 48 89 e5              mov %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01            addl $0x1,-0x4(%rbp)
4004fc: eb fa                  jmp 4004f8 <loop+0xb>
4004fd: fa
4004fc: eb
4004fb: 01
4004fa: fc
4004f9: 45
4004f8: 83
4004f7: 00
4004f6: 00
4004f5: 00
4004f4: 00
4004f3: fc
4004f2: 45
4004f1: c7
4004f0: e5
4004ef: 89
4004ee: 48
4004ed: 55
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
The program counter (PC), known as `%rip` in x86-64, stores the address in memory of the next instruction to be executed.

```
0x4004ee <loop>:
push %rbp
mov %rsp,%rbp
movl $0x0,-0x4(%rbp)
addl $0x1,-0x4(%rbp)
jmp 4004f8 <loop+0xb>
```

%rip

00000000004004ed <loop>:
4004ed: 55 push %rbp
4004ee: 48 89 e5 mov %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01 addl $0x1,-0x4(%rbp)
4004fc: eb fa jmp 4004f8 <loop+0xb>
%rip

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004ed</td>
<td>push %rbp</td>
<td>Store %rbp on the stack</td>
</tr>
<tr>
<td>0x4004ee</td>
<td>mov %rsp,%rbp</td>
<td>Move %rbp to %rsp</td>
</tr>
<tr>
<td></td>
<td>movl $0x0,-0x4(%rbp)</td>
<td>Load 0x0 into memory offset of %rbp</td>
</tr>
<tr>
<td></td>
<td>addl $0x1,-0x4(%rbp)</td>
<td>Add 0x1 to memory offset of %rbp</td>
</tr>
<tr>
<td>0x4004fa</td>
<td>jmp 4004f8 &lt;loop+0xb&gt;</td>
<td>Jump to address 4004f8 + 0xb</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.
Special hardware sets the program counter to the next instruction:

\%rip += size of bytes of current instruction

0x4004fc

000000000004004ed <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>
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).
Jump!

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

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

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>

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

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

```
000000000004004ed <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>
```

%rip

0x4004fc
This assembly represents an infinite loop in C!

```
while (true) {...}
```
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 (cont’d.)
- Control Flow Mechanics
  - Condition Codes
  - Assembly Instructions
- If statements
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

```java
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.

<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”:

```assembly
// 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 \texttt{cmp \texttt{S1,S2}} as “compare \textit{S2} to \textit{S1}”: 

\begin{verbatim}
// 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]
\end{verbatim}

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).  
  \[(a < 0 == b < 0) \ & \ (t < 0 != a < 0)\]
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

<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>
Control

Read `cmp S1, S2` as “compare S2 to S1”. It calculates S2 – S1 and updates the condition codes with the result.

```c
// 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>jnle</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><code>testb</code></td>
<td>Test byte</td>
</tr>
<tr>
<td><code>testw</code></td>
<td>Test word</td>
</tr>
<tr>
<td><code>testl</code></td>
<td>Test double word</td>
</tr>
<tr>
<td><code>testq</code></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!
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?

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

2. `test $0x10,%edi`
   ```
   je  40056f
   add  $0x1,%edi
   ```
Exercise 1: Conditional jump

je target  

jump if ZF is 1

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

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

   S2 - S1 == 0, so jump

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

   S2 & S1 != 0, so don't jump
Exercise 2: Conditional jump

00000000004004d6 <if_then>:

4004d6:  83 ff 06    cmp    $0x6,%edi
4004d9:  75 03    jne    4004de <if_then+0x8>
4004db:  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
Exercise 2: Conditional jump

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

A. 4004d9  
B. 4004db  
C. 4004de  
D. Other

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

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

000000000004004d6 <if_then>:
  4004d6:  83  ff  06   cmp   $0x6,%edi
  4004d9:  75  03   jne   4004de <ifThen+0x8>
  4004db:  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
Lecture Plan

• Assembly Execution and `%rip` (cont’d.)
• Control Flow Mechanics
• If statements
If Statements

How can we use instructions like `cmp` and *conditional jumps* to implement if statements in assembly?
Practice: Fill In The Blank

```c
int if_then(int param1) {
    if ( __________ ) {
        __________;
    }
    return __________;
}
```

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

```c
int if_then(int param1) {
    if (param1 == 6) {
        param1++;  // Fill In The Blank
    }
    return param1 * 2;  // Fill In The Blank
}
```

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

If-Else In C
if (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

```c
if ( _________ ) {
    ______;
} else {
    ______;
}
____;
```

If-Else In Assembly pseudocode

```
Test
Jump to else-body if test fails
If-body
Jump to past else-body
Else-body
Past else body
```
Practice: Fill In The Blank

If-Else In C

```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
Recap

• Assembly Execution and %rip (cont’d.)
• Control Flow Mechanics
  • Condition Codes
  • Assembly Instructions
• If statements

Next time: Loops, other instructions that depend on condition codes