CS107, Lecture 18
Assembly: Control Flow

Reading: B&O 3.6
Ed Discussion: https://edstem.org/us/courses/28214/discussion/2093991
Learning Goals

• Understand how assembly implements loops and control flow
• Learn about how assembly stores comparison and operation results in condition codes
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 (PC), %rip.
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!

Main Memory

Stack

Heap

Data

Text (code)

Machine code instructions

0x0
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

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

### asm:

```
loop:  
    push %rbp    ; 4004ed
    mov %rsp,%rbp ; 4004ee
    movl $0x0,-0x4(%rbp) ; 4004f1
    addl $0x1,-0x4(%rbp) ; 4004f8
    jmp 4004f8 <loop+0xb> ; 4004fc
```

### x86-64 Assembly:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004ed</td>
<td>push %rbp</td>
</tr>
<tr>
<td>4004ee</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>4004f1</td>
<td>movl $0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004f8</td>
<td>addl $0x1,-0x4(%rbp)</td>
</tr>
<tr>
<td>4004fc</td>
<td>jmp 4004f8 &lt;loop+0xb&gt;</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.

```
0000000000000000 <loop>:
0x4004ed: 55 push %ebp
0x4004ee: 48 89 e5 mov %rsp,%ebp
0x4004f1: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%ebp)
0x4004f8: 83 45 fc 01 addl $0x1,-0x4(%ebp)
0x4004fc: eb fa jmp 4004f8 <loop+0xb>
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004fd</td>
<td>fa</td>
</tr>
<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>4004fb</td>
<td>01</td>
</tr>
<tr>
<td>4004fc</td>
<td>01</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>
The program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.
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 program counter (PC), known as %rip in x86-64, stores the address in memory of the next instruction to be executed.

0x4004f8 %rip
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
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.
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).
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).

```assembly
0x00000000004004ed <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>
```

```
0x4004fc
%rip
```
This assembly represents an infinite loop in C!

while (true) {...}
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
```
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**?
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
} else {
    // 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 variants of `jmp` that branch if and only if certain conditions are met. 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>je <code>Label</code></td>
<td>jz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>jne <code>Label</code></td>
<td>jnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>js <code>Label</code></td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>jns <code>Label</code></td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg <code>Label</code></td>
<td>jnle</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>jge <code>Label</code></td>
<td>jnl</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl <code>Label</code></td>
<td>jnge</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>jle <code>Label</code></td>
<td>jng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja <code>Label</code></td>
<td>jnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae <code>Label</code></td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb <code>Label</code></td>
<td>jnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe <code>Label</code></td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Read `cmp S1, S2` as "compare S2 to S1":

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

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

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

// Jump if %edi <= 1
```assembly
cmp $1, %edi
jle [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 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?
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 yielded 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!)

\[
\text{CMP } S1, S2 \quad \text{S2} - \text{S1}
\]

<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>cmpq</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.

```assembly
// 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:**

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

S2 > S1

S2 - S1 > 0
```
# 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>jn1</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](image)

<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!
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 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  \hspace{2cm} \textit{jump if ZF is 1}

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

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
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 __________;
}

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
int if_then(int param1) {
    if (param1 == 6) {
        param1++;  
    }
    return param1 * 2;
}
Practice: Fill in the Blank

**If-Else In C**

```c
long absdiff(long x, long y) {
    long result;
    if (________) {
        __________________
    } else {
        __________________
    }
    return result;
}
```

**If-Else In Assembly pseudocode**

```
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
```

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;
}

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

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

Assembly

```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

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

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

```c
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

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++;
    }
}

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

```c
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
```

Add 1 to %eax (i).
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
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

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

```c
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!
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
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