CS107, Lecture 18
Assembly: Control Flow

Reading: B&O 3.6

While you’re getting situated: grab a handout and reference sheet in the front or back!

_masks recommended

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}

---------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax
    imull 4(%rdi), %eax
    movslq %esi, %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;
    z -= ________;
    return ________;
}

---------
// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax        // copy y into %eax
    imull 4(%rdi), %eax    // multiply %eax by nums[1]
    movslq %esi, %rsi      // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax
    lea 2(%rax, %rax), %eax
    ret

Work through the last two blanks in groups and input your answer for the first blank on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= nums[y];

    return 2 * z + 2;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax      // copy y into %eax
    imull 4(%rdi), %eax  // multiply %eax by nums[1]
    movslq %esi, %rsi   // sign-extend %esi to %rsi
    subl (%rdi,%rsi,4), %eax  // subtract nums[y] from %eax
    lea 2(%rax, %rax), %eax // multiply %rax by 2, and add 2
    ret
Learning Assembly

Moving data around

Arithmetic and logical operations

Control flow

Function calls

This Lecture

See more guides on Resources page of course website!
Learning Goals

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

• Assembly Execution and %rip
• Control Flow Mechanics
  • Condition Codes
  • Assembly Instructions
Lecture Plan

• Assembly Execution and %rip
• Control Flow Mechanics
  • Condition Codes
  • Assembly Instructions
What does it mean for a program to execute?
Executing Instructions

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

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

Answer:
• The **program counter** (PC), %rip.
Some registers take on special responsibilities during program execution.

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

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
Instructions Are Just Bytes!
Memory bus

Main memory

"hello, world\n hello code"
Instructions Are Just Bytes!

Machine code instructions flow into the Text (code) section of Main Memory.
00000000004004ed <loop>: 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>
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.

%rip

0x4004ee

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

0x4004f1
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
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).
The jmp instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

```
0x4004fc
%rip
```
The `jmp` instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

```asm
0x4004ed <loop>:
push %rbp
mov %rsp,%rbp
movl $0x0,-0x4(%rbp)
addl $0x1,-0x4(%rbp)
jmp 4004f8 <loop+0xb>
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

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

The code snippet shows an example of a `jmp` instruction:

```
0x4004fc e5
```

This instruction jumps to the address `0x4004f8`. The disassembly of the code snippet is as follows:

```
0x4004ed <loop>:
push %rbp
mov %rsp,%rbp
movl $0x0,-0x4(%rbp)
addl $0x1,-0x4(%rbp)
jmp 4004f8 <loop+0xb>
```

Here's a breakdown of what happens:

1. **Push %rbp**: Pushes the value of `%rbp` onto the stack.
2. **Mov %rsp,%rbp**: Moves the value of `%rbp` into `%rsp`.
3. **Movl $0x0,-0x4(%rbp)**: Moves the value of `$0x0` to the location `-0x4(%rbp)`.
4. **Addl $0x1,-0x4(%rbp)**: Adds the value of `$0x1` to the location `-0x4(%rbp)`.
5. **Jmp 4004f8 <loop+0xb>**: Jumps to the address `0x4004f8` after the loop has ended.

This process sets up the environment for the loop and then jumps to the next part of the code.
Jump!

This assembly represents an infinite loop in C!

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

```
0x4004fc <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>
```

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

• Assembly Execution and %rip

• Control Flow Mechanics
  • Condition Codes
  • Assembly Instructions
In C, we have control flow statements like \texttt{if}, \texttt{else}, \texttt{while}, \texttt{for}, etc. to write programs that are more expressive than just one instruction following another.

This is \textit{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
}
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. \texttt{cmp S1, S2} // compare two values

2. \texttt{je [target]} or \texttt{jne [target]} or \texttt{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>je Label</td>
<td>jz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>js Label</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>jns Label</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg Label</td>
<td>jnle</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>jge Label</td>
<td>jnl</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Read `cmp S1,S2` as “compare S2 to S1”:

```plaintext
// 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]
```

Wait a minute – how does the jump instruction know anything about the compared values in the earlier instruction?
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.
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>
## 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 <code>Label</code></td>
<td>jz</td>
<td>Equal / zero (ZF = 1)</td>
</tr>
<tr>
<td>jne <code>Label</code></td>
<td>jnz</td>
<td>Not equal / not zero (ZF = 0)</td>
</tr>
<tr>
<td>js <code>Label</code></td>
<td></td>
<td>Negative (SF = 1)</td>
</tr>
<tr>
<td>jns <code>Label</code></td>
<td></td>
<td>Nonnegative (SF = 0)</td>
</tr>
<tr>
<td>jg <code>Label</code></td>
<td>jnle</td>
<td>Greater (signed &gt;) (ZF = 0 and SF = OF)</td>
</tr>
<tr>
<td>jge <code>Label</code></td>
<td>jn1</td>
<td>Greater or equal (signed &gt;=) (SF = OF)</td>
</tr>
<tr>
<td>jl <code>Label</code></td>
<td>jnge</td>
<td>Less (signed &lt;) (SF != OF)</td>
</tr>
<tr>
<td>jle <code>Label</code></td>
<td>jng</td>
<td>Less or equal (signed &lt;=) (ZF = 1 or SF! = OF)</td>
</tr>
<tr>
<td>ja <code>Label</code></td>
<td>jnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae <code>Label</code></td>
<td>jnb</td>
<td>Above or equal (unsigned &gt;=) (CF = 0)</td>
</tr>
<tr>
<td>jb <code>Label</code></td>
<td>jnae</td>
<td>Below (unsigned &lt;) (CF = 1)</td>
</tr>
<tr>
<td>jbe <code>Label</code></td>
<td>jna</td>
<td>Below or equal (unsigned &lt;=) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
Setting Condition Codes

The different conditional jumps look at appropriate combinations of condition codes to know whether the condition it cares about is true.

• E.g. **je** (“jump equal”) really checks if the ZF (zero flag) is 1

• E.g. **jns** (“jump not signed”) really checks if the SF (sign flag) is 1

• E.g. **jl** (“jump less than”) really checks if SF (sign flag) != OF (overflow flag)
  • SF = 1 and OF = 0 means no signed overflow, and the result was negative
  • SF = 0 and OF = 1 means signed overflow, and the result was positive, meaning it overflowed from the negative direction.
Control

Read \texttt{cmp S1,S2} as “\textit{compare S2 to S1}”. It calculates \(S2 - S1\) and updates the condition codes with the result.

\begin{verbatim}
// 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]
\end{verbatim}
Setting Condition Codes

Usually when `cmp` is paired with conditional jumps, we can read them together. But other instructions use the condition codes in different ways. Example:

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!
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. \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.
Recap

• Assembly Execution and %rip
• Control Flow Mechanics
  • Condition Codes
  • Assembly Instructions

**Lecture 18 takeaway:** We represent control flow in assembly by storing information in condition codes and having instructions that act differently depending on the condition code values. Conditionals commonly use `cmp` or `test` along with jumps to conditionally skip over assembly instructions.