CS107, Lecture 13
Assembly: Control Flow and The Runtime Stack

Reading: B&O 3.6
Learning Goals

• Learn how assembly implements loops and control flow
• Learn how assembly calls functions.
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• **Break**: Announcements

• Function Calls and the Stack
mov Variants

• **mov** only updates the specific register bytes or memory locations indicated.

• **Exception:** **movl** writing to a register will also set high order 4 bytes to 0.

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly-code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>Quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8</td>
</tr>
</tbody>
</table>

• Suffix sometimes optional if size can be inferred.
No-Op

• The **nop/nopl** instructions are “no-op” instructions – they do nothing!
• No-op instructions do nothing except increment %rip
• Why? To make functions align on nice multiple-of-8 address boundaries.

“Sometimes, doing nothing is the way to be most productive.” – Philosopher Nick
Sometimes, you’ll see the following: \texttt{mov \%ebx, \%ebx}

What does this do? It zeros out the top 32 register bits, because when \texttt{mov} is performed on an e-register, the rest of the 64 bits are zeroed out.
xor

- Sometimes, you’ll see the following: `xor %ebx, %ebx`
- What does this do? It sets %ebx to zero! May be more efficient than using `mov`. 
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Break: Announcements

• Function Calls and the Stack
Control

• In C, we have control flow statements like if, else, while, for, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.

• This boils down to 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?
  • A way to store conditions that we will check later
  • Assembly instructions whose behavior is dependent on these conditions
• In C, we have control flow statements like if, else, while, for, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.

• This boils down to 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?
  • A way to store conditions that we will check later
  • Assembly instructions whose behavior is dependent on these conditions
Plan For Today

• Control Flow
  • Condition Codes
    • Assembly Instructions

• Break: Announcements

• Function Calls and the Stack
Alongside normal registers, the CPU also has single-bit condition code registers. These can be updated and read to influence what to do next. They are automatically updated by 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.
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.

Which flag would be set after this code?

```c
int a = 5;
int b = -5;
int t = a + b;
```
Condition Codes

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.

Which flag would be set after this code?

```c
int a = 5;
int b = -5;
int t = a + b;
```
Condition Codes

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.

Which flag would be set after this code?

```c
int a = 5;
int b = -20;
int t = a + b;
```
Condition Codes

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.

Which flag would be set after this code?

```cpp
int a = 5;
int b = -20;
int t = a + b;
```
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.
Setting Condition Codes

• In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.

• The `cmp` instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>$S_1, S_2$</td>
<td>$S_2 - S_1$</td>
</tr>
<tr>
<td>cmpb</td>
<td></td>
<td>Compare byte</td>
</tr>
<tr>
<td>cmpw</td>
<td></td>
<td>Compare word</td>
</tr>
<tr>
<td>cmpl</td>
<td></td>
<td>Compare double word</td>
</tr>
<tr>
<td>cmpq</td>
<td></td>
<td>Compare quad word</td>
</tr>
</tbody>
</table>

• **NOTE:** the operand order can be confusing!
Setting Condition Codes

• In addition to being set automatically from logical and arithmetic operations, we can also update condition codes ourselves.

• The test instruction is like the AND instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>$S_1, S_2$</td>
<td>S2 &amp; S1</td>
</tr>
<tr>
<td>testb</td>
<td></td>
<td>Test byte</td>
</tr>
<tr>
<td>testw</td>
<td></td>
<td>Test word</td>
</tr>
<tr>
<td>testl</td>
<td></td>
<td>Test double word</td>
</tr>
<tr>
<td>testq</td>
<td></td>
<td>Test quad word</td>
</tr>
</tbody>
</table>
Setting Condition Codes

• The **test** instruction is like the AND instruction, but it does not store the result anywhere. It just sets condition codes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TEST</strong></td>
<td>$S_1, S_2$</td>
<td>Test</td>
</tr>
<tr>
<td>testb</td>
<td>$S_2 &amp; S_1$</td>
<td>Test byte</td>
</tr>
<tr>
<td>testw</td>
<td></td>
<td>Test word</td>
</tr>
<tr>
<td>testl</td>
<td></td>
<td>Test double word</td>
</tr>
<tr>
<td>testq</td>
<td></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!
Control

• In C, we have control flow statements like if, else, while, for, etc. that let us write programs that are more expressive than just “straight-line code” (one instruction following another). We can “control” the “flow” of our programs.
• This boils down to 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?
  • A way to store conditions that we will check later
  • Assembly instructions whose behavior is dependent on these conditions
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Break: Announcements

• Function Calls and the Stack
There are three common instruction types that use condition codes:

- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data
- **jmp** instructions conditionally jump to a different next instruction
# Conditionally Setting Bytes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Set Condition (1 if true, 0 if false)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sete D</code></td>
<td><code>setz</code></td>
<td>Equal / zero</td>
</tr>
<tr>
<td><code>setne D</code></td>
<td><code>setnz</code></td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td><code>sets D</code></td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns D</code></td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>setg D</code></td>
<td><code>setnle</code></td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td><code>setge D</code></td>
<td><code>setnle</code></td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td><code>setl D</code></td>
<td><code>setnge</code></td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td><code>setle D</code></td>
<td><code>setng</code></td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td><code>seta D</code></td>
<td><code>setnbe</code></td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td><code>setae D</code></td>
<td><code>setnbe</code></td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td><code>setb D</code></td>
<td><code>setnae</code></td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td><code>setbe D</code></td>
<td><code>setna</code></td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
# Conditionally Moving Data

<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>cmova S,R</td>
<td>cmovnbe</td>
<td>Above (unsigned &gt;) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>cmovae 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>
Condition Codes

Different combinations of condition codes can indicate different things.

• E.g. To check equality, we can look at the ZERO flag (a = b means a – b = 0)

How can we check whether signed a < b?

• **Overflow may occur!** We have to take that into account.

• **If no overflow:** a < b if a – b < 0, and a >= b if a – b >= 0

• **If overflow:** a < b if a – b > 0 (negative overflow), a > b if a – b < 0 (positive overflow)

**Idea:** a < b when overflow flag is 1 and sign flag is 0, or when overflow flag is 0 and sign flag is 1. OF ^ SF!
The jmp instruction jumps to another instruction in the assembly code (“Unconditional Jump”).

\[
\text{jmp Label} \quad \text{(Direct Jump)} \\
\text{jmp *Operand} \quad \text{(Indirect Jump)}
\]

The destination can be hardcoded into the instruction (direct jump):

\[
\text{jmp 404f8 \langle loop+0xb\rangle} \quad \# \text{ jump to instruction at 0x404f8}
\]

The destination can also be read from a memory location (indirect jump):

\[
\text{jmp *\%rax} \quad \# \text{ jump to instruction at address in \%rax}
\]
Conditional Jumps

There are also variants of \texttt{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 \textit{Label}</td>
<td>jz</td>
<td>Equal / zero (ZF=1)</td>
</tr>
<tr>
<td>jne \textit{Label}</td>
<td>jnz</td>
<td>Not equal / not zero (ZF=0)</td>
</tr>
<tr>
<td>js \textit{Label}</td>
<td></td>
<td>Negative (SF=1)</td>
</tr>
<tr>
<td>jns \textit{Label}</td>
<td></td>
<td>Nonnegative (SF=0)</td>
</tr>
<tr>
<td>jg \textit{Label}</td>
<td>jnle</td>
<td>Greater (signed $&gt;$) (SF=0 and SF=OF)</td>
</tr>
<tr>
<td>jge \textit{Label}</td>
<td>jnl</td>
<td>Greater or equal (signed $\geq$) (SF=OF)</td>
</tr>
<tr>
<td>jl \textit{Label}</td>
<td>jnge</td>
<td>Less (signed $&lt;$) (SF $\neq$ OF)</td>
</tr>
<tr>
<td>jle \textit{Label}</td>
<td>jng</td>
<td>Less or equal (signed $\leq$) (ZF=1 or SF=OF)</td>
</tr>
<tr>
<td>ja \textit{Label}</td>
<td>jnbe</td>
<td>Above (unsigned $&gt;$) (CF = 0 and ZF = 0)</td>
</tr>
<tr>
<td>jae \textit{Label}</td>
<td>jnb</td>
<td>Above or equal (unsigned $\geq$) (CF = 0)</td>
</tr>
<tr>
<td>jb \textit{Label}</td>
<td>jnae</td>
<td>Below (unsigned $&lt;$) (CF = 1)</td>
</tr>
<tr>
<td>jbe \textit{Label}</td>
<td>jna</td>
<td>Below or equal (unsigned $\leq$) (CF = 1 or ZF = 1)</td>
</tr>
</tbody>
</table>
Jump instructions are critical to implementing control flow in assembly. Let’s see why!
Practice: Fill In The Blank

C Code

```c
void if_then(int param1) {
    if (__________) {
        __________;
    }
    param1 *= ______;
}
```

What does this assembly code translate to?

```assembly
00000000004004fe <if_then>:
  4004fe:    push %rbp
  4004ff:    mov %rsp,%rbp
  400502:    mov %edi,-0x4(%rbp)
  400505:    cmpl $0x6,-0x4(%rbp)
  400509:    jne 40050f
  40050b:    addl $0x1,-0x4(%rbp)
  40050f:    shll -0x4(%rbp)
  400512:    pop %rbp
  400513:    retq
```
**C Code**

```c
void if_then(int param1) {
    if (param1 == 6) {
        __________;  
    }
    param1 *= ______;
}
```

**What does this assembly code translate to?**

```
00000000004004fe <if_then>:
  4004fe:    push %rbp
  4004ff:    mov %rsp,%rbp
  400502:    mov %edi,-0x4(%rbp)
  400505:    cmpl $0x6,-0x4(%rbp)
  400509:    jne 40050f
  40050b:    addl $0x1,-0x4(%rbp)
  40050f:    shll -0x4(%rbp)
  400512:    pop %rbp
  400513:    retq
```
Practice: Fill In The Blank

C Code

```c
void if_then(int param1) {
    if (param1 == 6) {
        param1++;  
    }
    param1 *= ______;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
    4004fe: push %rbp
    4004ff: mov %rsp,%rbp
    400502: mov %edi,-0x4(%rbp)
    400505: cmpl $0x6,-0x4(%rbp)
    400509: jne 40050f
    40050b: addl $0x1,-0x4(%rbp)
    40050f: shll -0x4(%rbp)
    400512: pop %rbp
    400513: retq
```
C Code

```c
void if_then(int param1) {
    if (param1 == 6) {
        param1++;  
    }
    param1 *= 2;
}
```

What does this assembly code translate to?

```
00000000004004fe <if_then>:
4004fe:    push %rbp
4004ff:    mov %rsp,%rbp
400502:    mov %edi,-0x4(%rbp)
400505:    cmpl $0x6,-0x4(%rbp)
400509:    jne 40050f
40050b:    addl $0x1,-0x4(%rbp)
40050f:    shll -0x4(%rbp)
400512:    pop %rbp
400513:    retq
```
Common If-Else Construction

If-Else In C

```c
if (num > 3) {
    x = 10;
} else {
    x = 7;
}
```

```c
num++;
```

If-Else In Assembly

```
Test
Jump past if-body if test fails
If-body
Jump past else-body
Else-body
Past else body
```
void loop() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
Loops and Control Flow

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++;
    }
}
```

```
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
```

Jump to another instruction.
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

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

jle means “jump if less than or equal”. The sign flag indicates the result was negative, so we jump.
Loops and Control Flow

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

```
0x00000000000400570 <+0>:  mov    $0x0,%eax
0x00000000000400575 <+5>:  jmp    0x40057a <loop+10>
0x00000000000400577 <+7>:  add    $0x1,%eax
0x0000000000040057a <+10>: cmp     $0x63,%eax
0x0000000000040057d <+13>: jle     0x400577 <loop+7>
0x0000000000040057f <+15>: repz retq
```

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

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

```
0x00000000000400570 <+0>:   mov   $0x0,%eax
0x00000000000400575 <+5>:   jmp   0x40057a <loop+10>
0x00000000000400577 <+7>:   add   $0x1,%eax
0x0000000000040057a <+10>:  cmp   $0x63,%eax
0x0000000000040057d <+13>:  jle    0x400577 <loop+7>
0x0000000000040057f <+15>:  repz retq
```

`jle` means “jump if less than or equal”. The sign flag indicates the result was negative, so we jump.
Loops and Control Flow

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

We continue in this pattern until we do not 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.
Common Loop Construction

For Loop In C
for (int i = 0; i < n; i++) {
    // body
}

/* equivalent while loop */
int i = 0;
while (i < n) {
    // body
    i++;
}

For Loop In Assembly
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
GCC For Loop Output

For Loop In Assembly

- Initialization
- Test
- Jump past loop if fails
- Body
- Increment
- Jump to test

GCC For Loop Output

- Initialization
- Jump to test
- Body
- Increment
- Test
- Jump to body if success
GCC For Loop Output

For Loop In Assembly

Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test

GCC For Loop Output

Initialization
Jump to test
Body
Increment
Test
Jump to body if success
GCC For Loop Output

For Loop In Assembly

Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test

for (int i = 0; i < n; i++)  // n = 100
For Loop In Assembly

Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test

 GCC For Loop Output

```
for (int i = 0; i < n; i++) // n = 100
{
    Initialization
    Test
    No jump
    Body
    Increment
    Jump to test
    Test
    No jump
    Body
    Increment
    Jump to test
    ...
```
For Loop In Assembly

Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test
GCC For Loop Output

```
for (int i = 0; i < n; i++) // n = 100
```

GCC For Loop Output

- Initialization
- Jump to test
- Body
- Increment
- Test
- Jump to body if success
for (int i = 0; i < n; i++) // n = 100

Initialization
Jump to test
Body
Increment
Test
Jump to body
Body
Increment
Test
Jump to body
Body
...

GCC For Loop Output
Initialization
Jump to test
Body
Increment
Test
Jump to body if success
for (int i = 0; i < n; i++)    // n = 100

Initialization
Jump to test
Body
Increment
Test
Jump to body
Body
Increment
Test
Jump to body
Body
...

GCC For Loop Output
Initialization
Jump to test
Body
Increment
Test
Jump to body if success
GCC For Loop Output

For Loop In Assembly
Initialization
Test
Jump past loop if fails
Body
Increment
Jump to test

GCC For Loop Output
Initialization
Jump to test
Body
Increment
Test
Jump to body if success

Which instructions are better when n = 0?

for (int i = 0; i < n; i++) // n = 100
Both of these loop forms have the same static instruction count – same number of written instructions.

But they have different dynamic instruction counts – the number of times these instructions are executed when the program is run.

- If $n = 0$, left is best
- If $n$ is large, right is best

The compiler may emit static instruction counts many times longer than alternatives, but which is more efficient if loop executes many times.

Problem: the compiler may not know whether the loop will execute many times! Hard problem..... (take EE108, EE180, CS316 for more!)
Optimizations

• **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.

• Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.
Practice: Fill In The Blank

C Code

```c
long loop(long a, long b) {
    long result = _______;  // initial result
    while (_________) {  // while condition
        result = __________;  // result calculation
        a = __________;  // update a
    }
    return result;
}
```

What does this assembly code translate to?

```assembly
// a in %rdi, b in %rsi
long loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```
Practice: Fill In The Blank

C Code

```c
long loop(long a, long b) {
    long result = 1;
    while (a < b) {
        result = result*(a+b);
        a = a + 1;
    }
    return result;
}
```

What does this assembly code translate to?

```assembly
// a in %rdi, b in %rsi
loop:
    movl $1, %eax
    jmp .L2
.L3
    leaq (%rdi,%rsi), %rdx
    imulq %rdx, %rax
    addq $1, %rdi
.L2
    cmpq %rsi, %rdi
    jl .L3
rep; ret
```
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Break: Announcements

• Function Calls and the Stack
Announcements

• Midterms have been graded, and will be returned after class
• Visit gradescope.com (you should receive an email) to view your exam and score
• Regrades accepted until next Friday at 2PM
Plan For Today

• Control Flow
  • Condition Codes
  • Assembly Instructions

• Instruction Pointer

• Break: Announcements

• Function Calls and the Stack
How do we call functions in assembly?
Calling Functions In Assembly

To call a function in assembly, we must do a few things:

• **Pass Control** – `%rip` must be adjusted to execute the function being called and then resume the caller function afterwards.

• **Pass Data** – we must pass any parameters and receive any return value.

• **Manage Memory** – we must handle any space needs of the caller on the stack.

Terminology: **caller** function calls the **callee** function.
Stack Frame

main()
myfunction()
foo()
foo2()
Heap
Data
Text (code)
0x0
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 element on the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!
The push and pop operations write and read from the stack, and they also modify the stack pointer, `%rsp`:

<table>
<thead>
<tr>
<th>Instruct</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq S</td>
<td>R[%rsp] ← R[%rsp]-8; M[R[%rsp]] ← S</td>
<td>Push quad word</td>
</tr>
<tr>
<td>popq D</td>
<td>D ← M[R[%rsp]]; R[%rsp] ← R[%rsp]+8</td>
<td>Push quad word</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing address

Stack "top"

0x108
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

![Stack Diagram]

Stack "top"

Stack "bottom"

Increasing address 0x108
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq</th>
<th>rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Initially:
- \%rax: 0x123
- \%rdx: 0
- \%rsp: 0x108

After pushq \%rax:
- \%rax: 0x123
- \%rdx: 0
- \%rsp: 0x100
Pushing onto the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq %rax</th>
<th>popq %rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
<td>%rdx</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
<td>%rsp</td>
</tr>
</tbody>
</table>

Initially, the stack is at 0x108 with %rdx set to 0. After pushing %rax (0x123), the stack moves up to 0x108. Then, after popping %rdx, the stack moves back to 0x108.
Pushing/Popping with the Stack

• Pushing a quad word onto the stack means decrementing the stack pointer by 8, and writing the value to the new top-of-stack address. Equivalent (e.g. with %rax):
  
  \[
  \text{subq} \ $8, \ %\text{rsp} \\
  \text{movq} \ %\text{rax}, \ (%\text{rsp})
  \]

• Popping a quad word off the stack means reading the value at %rsp, and then incrementing the stack pointer by 8. Equivalent (e.g. with %rdx):
  
  \[
  \text{movq} \ (%\text{rsp}), \ %\text{rdx} \\
  \text{addq} \ $8, \ %\text{rsp}
  \]
Key Idea: `%rsp`

`%rsp` must point to the same place before and after a function is called.
Stack Frame

%rsp

main()
myfunction()
foo()
foo2()

Heap

Data

Text (code)

0x0
To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.

- **Pass Data** – we must pass any parameters and receive any return value.

- **Manage Memory** – we must handle any space needs of the caller on the stack.

Terminology: **caller** function calls the **callee** function.
Passing Control

**Problem:** %rip stores the current instruction being executed. If we execute the callee’s instructions, we must **remember** what instruction to resume at in the caller after!

**Solution:** use the `callq` command to push the current value of %rip at the bottom of the caller’s stack frame before calling the function. Then after the callee is finished, use the `ret` instruction to put this value back into %rip and continue executing.
The **call** instruction pushes the value of %rip onto the stack and sets %rip to point to the beginning of the specified function.

```assembly
    call Label
    call *Operand
```

The **ret** instruction pops the value of %rip from the stack and sets %rip to store this value.

```assembly
    ret
```
Recap

• Control Flow
  • Condition Codes
  • Assembly Instructions
• Instruction Pointer
• Break: Announcements
• Function Calls and the Stack

Next time: more function calls and optimizations