Administrivia

- **Homework 2** due Wednesday (7/17)
  - On Integers, Floating Point, and x86-64
- **Lab 2 (x86-64)**, due Monday (7/22)
  - Ideally want to finish well before the midterm
  - After today’s lecture, you should have everything you need to get started

- **Midterm** (Fri 7/26, 10:50-11:50am)
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation \( \text{op} \)
  - *Kind of weird:* Conditionals are comparisons against 0

- Come in instruction *pairs*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td>*p = <em>p + 5</em></td>
<td>( d \text{ (op) } s )</td>
</tr>
<tr>
<td><code>je:</code></td>
<td>( *p+5 == 0 )</td>
<td></td>
</tr>
<tr>
<td><code>jne:</code></td>
<td>( *p+5 != 0 )</td>
<td></td>
</tr>
<tr>
<td><code>jg:</code></td>
<td>( *p+5 &gt; 0 )</td>
<td></td>
</tr>
<tr>
<td><code>jl:</code></td>
<td>( *p+5 &lt; 0 )</td>
<td></td>
</tr>
<tr>
<td><code>orq a, b</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>je:</code></td>
<td>( b</td>
<td>a == 0 )</td>
</tr>
<tr>
<td><code>jne:</code></td>
<td>( b</td>
<td>a != 0 )</td>
</tr>
<tr>
<td><code>jg:</code></td>
<td>( b</td>
<td>a &gt; 0 )</td>
</tr>
<tr>
<td><code>jl:</code></td>
<td>( b</td>
<td>a &lt; 0 )</td>
</tr>
</tbody>
</table>

```c
orq a, b
je: b|a == 0
jne: b|a != 0
jg: b|a > 0
jl: b|a < 0
```
Choosing instructions for conditionals

- Reminder: \texttt{cmp} is like \texttt{sub}, \texttt{test} is like and
  - Result is not stored anywhere

<table>
<thead>
<tr>
<th></th>
<th>\texttt{cmp a,b}</th>
<th>\texttt{test a,b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{je}</td>
<td>“Equal”</td>
<td>\texttt{b == a}</td>
</tr>
<tr>
<td>\texttt{jne}</td>
<td>“Not equal”</td>
<td>\texttt{b != a}</td>
</tr>
<tr>
<td>\texttt{js}</td>
<td>“Sign” (negative)</td>
<td>\texttt{b-a &lt; 0}</td>
</tr>
<tr>
<td>\texttt{jns}</td>
<td>(non-negative)</td>
<td>\texttt{b-a &gt;=0}</td>
</tr>
<tr>
<td>\texttt{jg}</td>
<td>“Greater”</td>
<td>\texttt{b &gt; a}</td>
</tr>
<tr>
<td>\texttt{jge}</td>
<td>“Greater or equal”</td>
<td>\texttt{b &gt;= a}</td>
</tr>
<tr>
<td>\texttt{jl}</td>
<td>“Less”</td>
<td>\texttt{b &lt; a}</td>
</tr>
<tr>
<td>\texttt{jle}</td>
<td>“Less or equal”</td>
<td>\texttt{b &lt;= a}</td>
</tr>
<tr>
<td>\texttt{ja}</td>
<td>“Above” (unsigned &gt;)</td>
<td>\texttt{b &gt; a}</td>
</tr>
<tr>
<td>\texttt{jb}</td>
<td>“Below” (unsigned &lt;)</td>
<td>\texttt{b &lt; a}</td>
</tr>
</tbody>
</table>

\begin{align*}
\texttt{cmpq} & \ 5, \ (p) \\
\texttt{je:} & \ \ast p = = \ 5 \\
\texttt{jne:} & \ \ast p = ! = \ 5 \\
\texttt{jg:} & \ \ast p > \ 5 \\
\texttt{jl:} & \ \ast p < \ 5 \\
\end{align*}

\begin{align*}
\texttt{testq} & \ a, \ a \\
\texttt{je:} & \ a = = \ 0 \\
\texttt{jne:} & \ a = ! = \ 0 \\
\texttt{jg:} & \ a > \ 0 \\
\texttt{jl:} & \ a < \ 0 \\
\end{align*}

\begin{align*}
\texttt{testb} & \ a, \ 0x1 \\
\texttt{je:} & \ a_{\text{LSB}} = = \ 0 \\
\texttt{jne:} & \ a_{\text{LSB}} = \ 1 \\
\end{align*}
Choosing instructions for conditionals

\[ x - 3 \geq 0 \]
\[ x \geq 3 \]

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Use(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpq $3, %rdi</td>
<td></td>
<td>T1: # x &lt; 3:</td>
</tr>
<tr>
<td>jge T2</td>
<td></td>
<td>movq $1, %rax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T2: # !(x &lt; 3):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movq $2, %rax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

```c
if (x < 3) {
  return 1;
}
return 2;
```
Question

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

### A.
\[
\text{cmpq} \ %\text{rsi}, \ %\text{rdi} \\
\text{jle} \ .L4
\]

### B.
\[
\text{cmpq} \ %\text{rsi}, \ %\text{rdi} \\
\text{jg} \ .L4
\]

### C.
\[
\text{testq} \ %\text{rsi}, \ %\text{rdi} \\
\text{jle} \ .L4
\]

### D.
\[
\text{testq} \ %\text{rsi}, \ %\text{rdi} \\
\text{jg} \ .L4
\]

### E.
We’re lost…

---

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

**absdiff:**

```

    # x > y:
    movq %rdi, %rax  \text{result} = x
    subq %rsi, %rax  \text{result} -= y
    ret

    # x <= y:
    movq %rsi, %rax  \text{result} = y
    subq %rdi, %rax  \text{result} -= x
    ret
```

Vote at [http://pollev.com/wolfson](http://pollev.com/wolfson)
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Condition</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je “Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne “Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js “Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns (non-negative)</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg “Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge “Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl “Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle “Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja “Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb “Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

cmpq $3, %rdi
setl %al
cmpq %rsi, %rdi
sete %bl
testb %al, %bl
je T2
T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret

https://godbolt.org/z/j72AEn
Labels

- A jump changes the program counter (%rip)
  - %rip tells the CPU the address of the next instr to execute
- **Labels** give us a way to refer to a specific instruction in our assembly/machine code
  - Associated with the *next* instruction found in the assembly code (ignores whitespace)
  - Each *use* of the label will eventually be replaced with something that indicates the final address of the instruction that it is associated with

```
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret

max:
  movq %rdi, %rax
  cmpq %rsi, %rdi
  jg done
  movq %rsi, %rax
done:
  ret
```
GDB Demo

- See files on course schedule:
  - mov.s – assembly file
  - mov_demo.txt – commands to use with gdb
  - mov_tui_demo.txt – commands for gdb using TUI

- The movz and movs examples on a real machine!

- You will need to use GDB to get through Lab 2

- Pay attention to:
  - Setting breakpoints (break)
  - Stepping through code (step/next and stepti/nextti)
  - Printing out expressions (print – works with regs & vars)
  - Examining memory (x)
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- **Loops**
- Switches
Expressing with Goto Code

C allows `goto` as means of transferring control (jump)

- Closer to assembly programming style
- Generally considered bad coding style!!! Do not write this in your code!
  - If you use `goto`, I can’t promise that you won’t get eaten by a t-rex

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

```c
long absdiff_j(long x, long y) {
    long result;
    int ntest = (x <= y);
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```
Compiling Loops

Other loops compiled similarly
- Will show variations and complications in coming slides, but may skip a few examples in the interest of time

Most important to consider:
- When should conditionals be evaluated? (while vs. do-while)
- How much jumping is involved?

C/Java code:
```java
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:
```assembly
loopTop:
    testq %rax, %rax
    je    loopDone
    <loop body code>
    jmp  loopTop
loopDone:
```

Sum & sum = sum
Compiling Loops

C/Java code:

\begin{Verbatim}
while ( Test ) {
    Body
}
\end{Verbatim}

Goto version:

\begin{Verbatim}
Loop: if ( !Test ) goto Exit;
    Body
    goto Loop;
Exit:
\end{Verbatim}

- What are the Goto versions of the following?
  - Do...while: Test and Body
  - For loop: Init, Test, Update, and Body
Compiling Loops

**While Loop:**

C:
```c
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```assembly
loopTop:  testq  %rax, %rax
          je   loopDone
          jmp  loopTop
loopDone:  
```

**Do-while Loop:**

C:
```c
do {
    <loop body>
} while ( sum != 0 )
```

x86-64:
```assembly
loopTop:  <loop body code>
          testq  %rax, %rax
          jne   loopTop
loopDone:  
```

**While Loop (ver. 2):**

C:
```c
while ( sum != 0 ) {
    <loop body>
}
```

x86-64:
```assembly
loopTop:  <loop body code>
          testq  %rax, %rax
          je   loopDone
          jmp  loopTop
loopDone:  
```
For-Loop → While-Loop

For-Loop:

```
for (Init; Test; Update) {
    Body
}
```

While-Loop Version:

```
Init;
while (Test) {
    Body
    Update;
}
```

Caveat: C and Java have break and continue

- Conversion works fine for break
  - Jump to same label as loop exit condition
- But not continue: would skip doing Update, which it should do with for-loops
  - Introduce new label at Update
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
long switch_ex (long x, long y, long z) {
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
Jump Table Structure

Switch Form

```java
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    case val_{n-1}:
        Block n-1
}
```

Approximate Translation

```java
target = JTab[x];
goto target;
```

Jump Table

```
 JTab:
     | Targ0 |
     | Targ1 |
     | Targ2 |
     | ...  |
     | Targ_{n-1} |
```

Jump Targets

```
 Targ0: Code Block 0
 Targ1: Code Block 1
 Targ_{n-1}: Code Block n-1
```
Jump Table Structure

C code:

```c
switch (x) {
    case 1: <some code>
        break;
    case 2: <some code>
    case 3: <some code>
        break;
    case 5:
    case 6: <some code>
        break;
    default: <some code>
}
```

Use the jump table when $x \leq 6$:

```c
if (x <= 6)
    target = JTab[x];
goto target;
else
    goto default;
```
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

Note compiler chose to not initialize \( w \)

Jump above – unsigned > catches negative default cases

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Take a look!

[https://godbolt.org/z/dOWSFR](https://godbolt.org/z/dOWSFR)
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

Jump table

```
.switch (x) {
    case 1: <code>
        break;
    case 2: <code>
    case 3: <code>
        break;
    case 5: 
    case 6: <code>
        break;
    default: <code>
}
```

```
.switch_ex:
    movq       %rdx, %rcx
    cmpq       $6, %rdi     # x:6
    ja         .L8          # default
    jmp        *.L4(%rdi,8) # jump table
```

Indirect jump

```
x = 3
.L4 + 8x3 = .L4 + 24
```
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump**: `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump**: `jmp *(.L4, %rdi, 8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

---

### Jump table

```
.section .rodata
.align 8
.L4:
    .quad .L8  # x = 0
    .quad .L3  # x = 1
    .quad .L5  # x = 2
    .quad .L9  # x = 3
    .quad .L8  # x = 4
    .quad .L7  # x = 5
    .quad .L7  # x = 6
```
Jump Table

declaring data, not instructions

Jump table

.switch .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6

8-byte memory alignment

switch(x) {
case 1: // .L3
    w = y*z;
    break;

case 2: // .L5
    w = y/z;
    /* Fall Through */

case 3: // .L9
    w += z;
    break;

case 5:
case 6: // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}

this data is 64-bits wide

declaring data, not instructions

8-byte memory alignment
Code Blocks ($x == 1$)

```c
switch(x) {
    case 1:  // .L3
        w = y*Z;
        break;
    ...
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument ($x$)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument ($y$)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument ($z$)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
.L3:
    movq %rsi, %rax  # y
    imulq %rdx, %rax # y*Z
    ret
```
Handling Fall-Through

```c
long w = 1;
...
switch (x) {
    ...
    case 2: // .L5
        w = y/z;
        /* Fall Through */
        case 3: // .L9
            w += z;
            break;
    ...
}
```

- More complicated choice than “just fall-through” forced by “migration” of `w = 1`;
  - Example compilation trade-off

```c
case 2:
    w = y/z;
    goto merge;
```
Code Blocks (x == 2, x == 3)

```c
long w = 1;
    ...
switch (x) {
    ...
    case 2: // .L5
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    ...
}
```

```
.L5:
    movq %rsi, %rax # y in rax
    cqto # Div prep
    idivq %rcx # y/z
    jmp .L6 # goto merge

.L9:
    movl $1, %eax # w = 1
    addq %rcx, %rax # w += z
    ret
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
## Code Blocks (rest)

```c
switch (x) {
    ...  
    case 5: // .L7
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
.L7: # Case 5,6:
    movl $1, %eax  # w = 1
    subq %rdx, %rax # w -= z
    ret
.L8: # Default:
    movl $2, %eax  # 2
    ret
```