Instructor:  
Sam Wolfson  

Teaching Assistants:  
Rehaan Bhimani  
Corbin Modica  
Daniel Hsu

https://xkcd.com/292/
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>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{addq} 5, (p)</td>
<td>( \text{je: } \ast p+5 == 0 )</td>
<td>(op) s == 0</td>
</tr>
<tr>
<td>\texttt{jne}</td>
<td>( \text{jne: } \ast p+5 != 0 )</td>
<td>(op) s != 0</td>
</tr>
<tr>
<td>\texttt{js}</td>
<td>( \text{js: } \ast p+5 &gt; 0 )</td>
<td>(op) s &gt; 0</td>
</tr>
<tr>
<td>\texttt{jns}</td>
<td>( \text{jns: } \ast p+5 &gt;= 0 )</td>
<td>(op) s &gt;= 0</td>
</tr>
<tr>
<td>\texttt{ja}</td>
<td>( \text{ja: } \ast p+5 &lt; 0 )</td>
<td>(op) s &lt; 0</td>
</tr>
<tr>
<td>\texttt{jb}</td>
<td>( \text{jb: } \ast p+5 &lt; 0 )</td>
<td>(op) s &lt; 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{orq} a, b</td>
<td>( \text{je: } b</td>
<td>a == 0 )</td>
</tr>
<tr>
<td>\texttt{jne}</td>
<td>( \text{jne: } b</td>
<td>a != 0 )</td>
</tr>
<tr>
<td>\texttt{jg}</td>
<td>( \text{jg: } b</td>
<td>a &gt; 0 )</td>
</tr>
<tr>
<td>\texttt{jle}</td>
<td>( \text{jle: } b</td>
<td>a &lt; 0 )</td>
</tr>
<tr>
<td>\texttt{ja}</td>
<td>( \text{ja: } b</td>
<td>a &gt; 0 )</td>
</tr>
<tr>
<td>\texttt{jb}</td>
<td>( \text{jb: } b</td>
<td>a &lt; 0 )</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

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

```c
cmpq 5, (p)
  je:  *p == 5
  jne:  *p != 5
  jg:  *p > 5
  jl:  *p < 5

testq a, a
  je:  a == 0
  jne:  a != 0
  jg:  a > 0
  jl:  a < 0

testb a, 0x1
  je:  a_{LSB} == 0
  jne:  a_{LSB} == 1
```
Choosing instructions for conditionals

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

If \( x < 3 \) {
    return 1;
}
return 2;

cmpq $3, %rdi
jge T2
T1: # x < 3:
    movq $1, %rax
    ret
T2: # !(x < 3):
    movq $2, %rax
    ret
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>

Vote at [http://pollev.com/wolfson](http://pollev.com/wolfson)

A. \texttt{cmpq} \%rsi, \%rdi \texttt{jle} .L4
B. \texttt{cmpq} \%rsi, \%rdi \texttt{jg} .L4
C. \texttt{testq} \%rsi, \%rdi \texttt{jle} .L4
D. \texttt{testq} \%rsi, \%rdi \texttt{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;
}
```

\texttt{absdiff:}

\[
\]

\# x > y:

\[
\]

\texttt{movq} \%rdi, \%rax
\texttt{subq} \%rsi, \%rax
\texttt{ret}

\# x \leq y:

\[
\]

\texttt{movq} \%rsi, \%rax
\texttt{subq} \%rdi, \%rax
\texttt{ret}
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th></th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

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

```assembly
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
```
Labels

- A jump changes the program counter (%rip)
  - %rip tells the CPU the *address* of the next instruction 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

```assembly
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 for 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 `steepi/nexti`)
  - 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
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;
}
```

- 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
Compiling Loops

C/Java code:

```java
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:

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

- 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?
Compiling Loops

What are the Goto versions of the following?

- Do...while: Test and Body
- For loop: Init, Test, Update, and Body

C/Java code:

```
while ( Test ) {
    Body
}
```

Goto version

```
Loop: if (!Test) goto Exit;
    Body
    goto Loop;
Exit:
```
Compiling Loops

**While Loop:**

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

x86-64:
```assembly
loopTop:       testq  %rax, %rax
               je    loopDone
               <loop body code>
               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:       testq  %rax, %rax
               je    loopDone
               <loop body code>
               testq  %rax, %rax
               jne   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**
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - Jump table
  - Indirect jump instruction

```c
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
}
```

Jump Table

- JTab: Targ0
- Targ0
- Targ1
- Targ2
- ...
- Targn-1

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- ...
- Targn-1: Code Block n-1

Approximate Translation

```java
target = JTab[N];
goto target;
```
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;
}
```

### Register Use(s)

<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>%rdx</td>
<td>3\textsuperscript{rd} argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Note compiler chose to not initialize `w`

Take a look!
[https://godbolt.org/z/dOWSFR](https://godbolt.org/z/dOWSFR)

Jump above – unsigned > catches negative default cases
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:
    .align 8
    .section .rodata
    .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
```

Indirect jump
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

8-byte memory alignment

declaring data, not instructions

8-byte memory alignment

this data is 64-bits wide

.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;
}

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

this data is 64-bits wide

8-byte memory alignment

Jump table
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;
    ...
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

More complicated choice than “just fall-through” forced by “migration” of \( w = 1 \);

- Example compilation trade-off
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;
    . . .
}
```

### 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>

```
.L5:
    # Case 2:
    movq %rsi, %rax  # y in rax
    cqto          # Div prep
    idivq %rcx  # y/z
    jmp .L6  # goto merge
.L9:
    # Case 3:
    movl $1, %eax  # w = 1
.L6:
    # merge:
    addq %rcx, %rax  # w += z
    ret
```
Code Blocks (rest)

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

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

### Register Use(s)

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