x86-64 Programming III
CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou

http://xkcd.com/1652/
Administrivia

- Lab 1b due TONIGHT, Monday (4/20) – 11:59pm
  - Submit `bits.c` and `lab1Breflect.txt`
  - Submissions that fail the autograder get a ZERO
    - No excuses – make full use of tools & Gradescope’s interface
- Unit Summary #1 due this Friday (4/24) on Canvas
- Lab 2 (x86-64) due next Friday (5/01)
  - Optional GDB Tutorial homework on Gradescope

- You must log on with your @uw google account to access!!
  - Google doc for 11:30 Lecture: https://tinyurl.com/351-04-20A
  - Google doc for 2:30 Lecture: https://tinyurl.com/351-04-20B
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (op)
  - 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>addq 5, (p)</td>
<td>je: *p+5 == 0</td>
<td>jne: *p+5 != 0</td>
</tr>
<tr>
<td>orq a, b</td>
<td>je: b</td>
<td>a == 0</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

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

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

\[ \text{cmpq} \ 5, \ (p) \]
- \texttt{je}: \ *p == 5 \n- \texttt{jne}: \ *p != 5 \n- \texttt{jg}: \ *p > 5 \n- \texttt{jle}: \ *p < 5 \n
\[ \text{testq} \ a, \ a \]
- \texttt{je}: \ a == 0 \n- \texttt{jne}: \ a != 0 \n- \texttt{jg}: \ a > 0 \n- \texttt{jle}: \ a < 0 \n
\[ \text{testb} \ a, \ 0x1 \]
- \texttt{je}: \ a_{LSB} == 0 \n- \texttt{jne}: \ a_{LSB} == 1 \n
\[ *p - 5 \ ? \ 0 \]
- \text{cmpq} 5, (p)
Choosing instructions for conditionals

<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>Operator</th>
<th>Description</th>
<th>Condition</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp a,b</td>
<td>Compare a,b</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>

```c
if (x < 3) {
    return 1;
}
return 2;
```
CSE351, Spring 2020
L10: x86-64 Programming III

Polling Question [Asm III - a]

<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>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Vote at http://pollev.com/rea

A. \( \text{cmpq} \quad %\text{rsi}, \quad %\text{rdi} \quad \text{jle} \quad .L4 \)
B. \( \text{cmpq} \quad %\text{rsi}, \quad %\text{rdi} \quad \text{jg} \quad .L4 \)
C. \( \text{testq} \quad %\text{rsi}, \quad %\text{rdi} \quad \text{jle} \quad .L4 \)
D. \( \text{testq} \quad %\text{rsi}, \quad %\text{rdi} \quad \text{jg} \quad .L4 \)
E. We’re lost…

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

absdiff:

\[ x > y: \]
\[
\begin{align*}
    &\text{movq} \quad %\text{rdi}, \quad %\text{rax} \\
    &\text{subq} \quad %\text{rsi}, \quad %\text{rax} \\
    &\text{ret}
\end{align*}
\]

\[ x \leq y: \]
\[
\begin{align*}
    &\text{movq} \quad %\text{rsi}, \quad %\text{rax} \\
    &\text{subq} \quad %\text{rdi}, \quad %\text{rax} \\
    &\text{ret}
\end{align*}
\]
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</td>
<td>“Equal”</td>
<td>$b \leq x$ &amp; $x = y$</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>$b \neq a$</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>$b - a &lt; 0$</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>$b - a \geq 0$</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>$b &gt; a$</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>$b \geq a$</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>$b &lt; 3$</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>$b \leq a$</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>$b &gt; a$</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>$b &lt; a$</td>
</tr>
</tbody>
</table>

https://godbolt.org/z/GNxpqpv
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
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
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?
Compiling Loops

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:

```
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:

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

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

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

x86-64:

```
loopTop:
        testq %rax, %rax
        je  loopDone
        <loop body code>
        do while loop
        testq %rax, %rax
        jne  loopTop
loopDone:
```
For-Loop → While-Loop

For-Loop:

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

While-Loop Version:

```c
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

```c
switch (x) {
    case val_0: Block 0
    case val_1: Block 1
    ...
    case val_n-1: Block n-1
}
```

Jump Table

- JTab: 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

```c
target = JTab[x];
go to 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>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>

Note compiler chose to not initialize w

Take a look!

[https://godbolt.org/z/aY24el](https://godbolt.org/z/aY24el)
Switch Statement Example

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

**Jump table**

```assembly
.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
```

**Example code**

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

**Indirect jump**

Address of jump table: \( D + R_i + S \)  
Size of \( \text{void}^\* \)
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

```assembly
.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
```
Slides that expand on the simple switch code in assembly. These slides expand on material covered today, so while you don’t need to read these, the information is “fair game.”
Jump Table

Jump table

8-byte memory alignment

declaring data, not instructions

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;
}
```
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:

```assembly
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
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:
    # 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
```

<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
GDB Demo

- The **movz** and **movs** examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step/next` and `stepi/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining **memory** (`x`)