x86-64 Programming III
CSE 351 Spring 2019

Instructor:
Ruth Anderson

Teaching Assistants:
Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

http://xkcd.com/1652/
Administrivia

- Lab 1b due TONIGHT Monday (4/22)
  - Submit `bits.c` and `lab1Breflect.txt`
- Homework 2 due Wednesday (4/24)
  - On Integers, Floating Point, and x86-64
- Lab 2 (x86-64), due Wednesday (5/01)
  - Ideally want to finish well before the midterm

- Midterm (Fri 5/03, 4:30-5:30pm in KNE 130)
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 `stedi/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining `memory` (`x`)
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>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addq 5, (p)</strong></td>
<td></td>
<td><code>d (op) s == 0</code></td>
</tr>
<tr>
<td>je:</td>
<td>“Equal”</td>
<td></td>
</tr>
<tr>
<td>jne:</td>
<td>“Not equal”</td>
<td><code>d (op) s != 0</code></td>
</tr>
<tr>
<td>jg:</td>
<td>“Sign” (negative)</td>
<td><code>d (op) s &lt; 0</code></td>
</tr>
<tr>
<td>jl:</td>
<td>(non-negative)</td>
<td><code>d (op) s &gt;= 0</code></td>
</tr>
<tr>
<td><strong>orq a, b</strong></td>
<td></td>
<td><code>d (op) s &gt; 0</code></td>
</tr>
<tr>
<td>je:</td>
<td>“Greater”</td>
<td></td>
</tr>
<tr>
<td>jne:</td>
<td>“Greater or equal”</td>
<td><code>d (op) s &gt;= 0</code></td>
</tr>
<tr>
<td>jg:</td>
<td>“Less”</td>
<td><code>d (op) s &lt; 0</code></td>
</tr>
<tr>
<td>jl:</td>
<td>“Less or equal”</td>
<td><code>d (op) s &lt;= 0</code></td>
</tr>
<tr>
<td>ja:</td>
<td>“Above” (unsigned &gt;)</td>
<td><code>d (op) s &gt; 0U</code></td>
</tr>
<tr>
<td>jb:</td>
<td>“Below” (unsigned &lt;)</td>
<td><code>d (op) s &lt; 0U</code></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></th>
<th>cmp $a,b$</th>
<th>test $a,b$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>$b == a$</td>
<td>$b&amp;a == 0$</td>
</tr>
<tr>
<td><code>jne</code></td>
<td>$b \neq a$</td>
<td>$b&amp;a \neq 0$</td>
</tr>
<tr>
<td><code>js</code></td>
<td>$b-a &lt; 0$</td>
<td>$b&amp;a &lt; 0$</td>
</tr>
<tr>
<td><code>jns</code></td>
<td>$b-a \geq 0$</td>
<td>$b&amp;a \geq 0$</td>
</tr>
<tr>
<td><code>jg</code></td>
<td>$b &gt; a$</td>
<td>$b&amp;a &gt; 0$</td>
</tr>
<tr>
<td><code>jge</code></td>
<td>$b \geq a$</td>
<td>$b&amp;a \geq 0$</td>
</tr>
<tr>
<td><code>jl</code></td>
<td>$b &lt; a$</td>
<td>$b&amp;a &lt; 0$</td>
</tr>
<tr>
<td><code>jle</code></td>
<td>$b \leq a$</td>
<td>$b&amp;a \leq 0$</td>
</tr>
<tr>
<td><code>ja</code></td>
<td>$b &gt; a$</td>
<td>$b&amp;a &gt; 0U$</td>
</tr>
<tr>
<td><code>jb</code></td>
<td>$b &lt; a$</td>
<td>$b&amp;a &lt; 0U$</td>
</tr>
</tbody>
</table>

**Example:***

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

- `testq a, a`
  - `je`: $a == 0$
  - `jne`: $a \neq 0$
  - `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>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>

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

```assembly
    cmpq $3, %rdi
    jge T2
T1:  # x < 3: (if)
        movq $1, %rax
        ret
T2:  # !(x < 3): (else)
        movq $2, %rax
        ret
```
Question

A. `cmpq %rsi, %rdi
   jle .L4`

B. `cmpq %rsi, %rdi
   jg .L4`

C. `testq %rsi, %rdi
   jle .L4`

D. `testq %rsi, %rdi
   jg .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;
}`

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

Vote at [http://pollev.com/rea](http://pollev.com/rea)
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 \times 3)</td>
<td>(b \times 3)</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>(b \times a)</td>
<td>(b \times a)</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>(b-a &lt; 0)</td>
<td>(b-a &lt; 0)</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>(b-a &gt; 0)</td>
<td>(b-a &gt; 0)</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>(b &gt; a)</td>
<td>(b &gt; a)</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>(b \geq a)</td>
<td>(b \geq a)</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>(b &lt; 3)</td>
<td>(b &lt; 3)</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>(b \leq a)</td>
<td>(b \leq a)</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>(b &gt; a)</td>
<td>(b &gt; a)</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>(b &lt; a)</td>
<td>(b &lt; a)</td>
</tr>
</tbody>
</table>

- https://godbolt.org/z/j72AEn
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
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!
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

C/Java code:

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

Goto version

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

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>
}
```

**Do-while Loop:**

C:

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

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

C:

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

---

x86-64:

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

```
loopDone:
```

---

```
loopTop:    testq  %rax, %rax
            je     loopDone
            jne   loopTop
```

```
loopDone:
```

---

```
loopTop:    testq  %rax, %rax
            je     loopDone
            jne   loopTop
```

```
loopDone:
```

---

**Do-while Loop:**

C:

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

```c
loopTop:    testq  %rax, %rax
            jne   loopTop
```

```
loopDone:
```

---

```
loopTop:    testq  %rax, %rax
            jne   loopTop
```

```
loopDone:
```

---

```
loopTop:    testq  %rax, %rax
            jne   loopTop
```

```
loopDone:
```

---

**All jump instructions update the program counter (rip):**

```
sum == 0
```
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;
}

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
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
}
```

Approximate Translation

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

Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>...</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

Jump Targets

<table>
<thead>
<tr>
<th>Target</th>
<th>Code Block 0</th>
<th>Code Block 1</th>
<th>Code Block 2</th>
<th>Code Block n-1</th>
</tr>
</thead>
</table>

Memory

Addresses (8 bytes wide)
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)

- `%rdi` 1st argument (x)
- `%rsi` 2nd argument (y)
- `%rdx` 3rd argument (z)
- `%rax` return value

Note compiler chose to not initialize `w`

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

```
.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
```

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

Jump table

Declaring data, not instructions

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

8-byte memory alignment

This data is 64-bits wide
Code Blocks (x == 1)

```c
switch(x) {
    case 1:     // .L3
        w = y*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>

```assembly
.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;
```

```c
case 3:
    w = 1;
merge:
    w += z;
```
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;
  . . .
}
```

Registers:

<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>

Assembly:

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

.L9:  
  movl $1, %eax       # w = 1

.L6:  
  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;
}
```

### 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
```