x86-64 Programming III
CSE 351 Autumn 2020

Instructor:
Justin Hsia

Teaching Assistants:
Aman Mohammed
Ami Oka
Callum Walker
Cosmo Wang
Hang Do
Jim Limprasert
Joy Dang
Julia Wang
Kaelin Laundry
Kyrie Dowling
Mariam Mayanja
Shawn Stanley
Yan Zhe Ong

http://xkcd.com/1652/
Administrivia

- Lab 2 due next Friday (10/30)

- Section tomorrow on Assembly
  - Use the midterm reference sheet!
  - Optional GDB Tutorial slides and Lab 2 phase 1 walkthrough

- Midterm (take home, 10/31–11/2)
  - Find groups of 5 for the group stage
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!
Aside: movz and movs

**movz** __ src, regDest  # Move with zero extension
**movs** __ src, regDest  # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with *zero* (**movz**) or *sign bit* (**movs**)

**movz** _SD_  / **movs** _SD_:

- _S_ – size of source (_b_ = 1 byte, _w_ = 2)
- _D_ – size of dest (_w_ = 2 bytes, _l_ = 4, _q_ = 8)

**Example:**

```
movzbq %al, %rbx
```

<table>
<thead>
<tr>
<th>0x??</th>
<th>0x??</th>
<th>0x??</th>
<th>0x??</th>
<th>0x??</th>
<th>0x??</th>
<th>0x??</th>
<th>0xFF</th>
<th>←%rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0xFF</td>
<td>←%rbx</td>
</tr>
</tbody>
</table>

Aside: movz and movs

movz src, regDest  # Move with zero extension
movs src, regDest  # Move with sign extension

- Copy from a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register
- Fill remaining bits of dest with zero (movz) or sign bit (movs)

movz SD / movs SD:
- S – size of source (b = 1 byte, w = 2)
- D – size of dest (w = 2 bytes, l = 4, q = 8)

Example:
movsbl (%rax), %ebx

Copy 1 byte from memory into 8-byte register & sign extend it

Note: In x86-64, any instruction that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.
GDB Demo

- The `movz` and `movs` examples on a real machine!
  - `movzvbq %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`)
Reading Review

- Terminology:
  - Label, jump target
  - Program counter
  - Jump table, indirect jump

- Questions from the Reading?
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><code>addq 5, (p)</code></td>
<td><em>(p)+5 == 0</em></td>
<td><code>*p+5 == 0</code></td>
</tr>
<tr>
<td><code>je:</code></td>
<td></td>
<td>`b</td>
</tr>
<tr>
<td><code>jne:</code></td>
<td><em>(p)+5 != 0</em></td>
<td>`b</td>
</tr>
<tr>
<td><code>jg:</code></td>
<td><em>(p)+5 &gt; 0</em></td>
<td>`b</td>
</tr>
<tr>
<td><code>jl:</code></td>
<td><em>(p)+5 &lt; 0</em></td>
<td>`b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orq a, b</code></td>
<td><code>a == 0</code></td>
<td>`b</td>
</tr>
<tr>
<td><code>je:</code></td>
<td><code>a == 0</code></td>
<td>`b</td>
</tr>
<tr>
<td><code>jne:</code></td>
<td><code>a != 0</code></td>
<td>`b</td>
</tr>
<tr>
<td><code>jg:</code></td>
<td><code>a &gt; 0</code></td>
<td>`b</td>
</tr>
<tr>
<td><code>jl:</code></td>
<td><code>a &lt; 0</code></td>
<td>`b</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></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>(b == a)</td>
</tr>
<tr>
<td>\texttt{jne}</td>
<td>“Not equal”</td>
<td>(b != a)</td>
</tr>
<tr>
<td>\texttt{js}</td>
<td>“Sign” (negative)</td>
<td>(b-a &lt; 0)</td>
</tr>
<tr>
<td>\texttt{jns}</td>
<td>(non-negative)</td>
<td>(b-a &gt;= 0)</td>
</tr>
<tr>
<td>\texttt{jg}</td>
<td>“Greater”</td>
<td>(b &gt; a)</td>
</tr>
<tr>
<td>\texttt{jge}</td>
<td>“Greater or equal”</td>
<td>(b &gt;= a)</td>
</tr>
<tr>
<td>\texttt{jl}</td>
<td>“Less”</td>
<td>(b &lt; a)</td>
</tr>
<tr>
<td>\texttt{jle}</td>
<td>“Less or equal”</td>
<td>(b &lt;= a)</td>
</tr>
<tr>
<td>\texttt{ja}</td>
<td>“Above” (unsigned &gt;)</td>
<td>(b &gt; U a)</td>
</tr>
<tr>
<td>\texttt{jb}</td>
<td>“Below” (unsigned &lt;)</td>
<td>(b &lt; U a)</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Condition</th>
<th>Test Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code> “Equal”</td>
<td>( b == a )</td>
<td>( b &amp; a == 0 )</td>
</tr>
<tr>
<td><code>jne</code> “Not equal”</td>
<td>( b \neq a )</td>
<td>( b &amp; a \neq 0 )</td>
</tr>
<tr>
<td><code>js</code> “Sign” (negative)</td>
<td>( b-a &lt; 0 )</td>
<td>( b &amp; a &lt; 0 )</td>
</tr>
<tr>
<td><code>jns</code> (non-negative)</td>
<td>( b-a \geq 0 )</td>
<td>( b &amp; a \geq 0 )</td>
</tr>
<tr>
<td><code>jg</code> “Greater”</td>
<td>( b &gt; a )</td>
<td>( b &amp; a &gt; 0 )</td>
</tr>
<tr>
<td><code>jge</code> “Greater or equal”</td>
<td>( b \geq a )</td>
<td>( b &amp; a \geq 0 )</td>
</tr>
<tr>
<td><code>jl</code> “Less”</td>
<td>( b &lt; a )</td>
<td>( b &amp; a &lt; 0 )</td>
</tr>
<tr>
<td><code>jle</code> “Less or equal”</td>
<td>( b \leq a )</td>
<td>( b &amp; a \leq 0 )</td>
</tr>
<tr>
<td><code>ja</code> “Above” (unsigned &gt;)</td>
<td>( b &gt; a )</td>
<td>( b &amp; a &gt; 0 )</td>
</tr>
<tr>
<td><code>jb</code> “Below” (unsigned &lt;)</td>
<td>( b &lt; a )</td>
<td>( b &amp; a &lt; 0 )</td>
</tr>
</tbody>
</table>

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

```assembly
cmpq $3, %rdi
jge T2

T1: # x < 3:
    movq $1, %rax
    ret

T2: # !(x < 3):
    movq $2, %rax
    ret
```
Practice Question 1

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

<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>

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…
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 &quot;Equal&quot;</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne &quot;Not equal&quot;</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js &quot;Sign&quot; (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 &quot;Greater&quot;</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge &quot;Greater or equal&quot;</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl &quot;Less&quot;</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle &quot;Less or equal&quot;</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja &quot;Above&quot; (unsigned &gt;)</td>
<td>b &gt;U a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb &quot;Below&quot; (unsigned &lt;)</td>
<td>b &lt;U 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
ej T2
T1: # x < 3 && x == y:
  movq $1, %rax
  ret
T2: # else
  movq $2, %rax
  ret

https://godbolt.org/z/Tfrv33
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
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
Compiling Loops

C/Java code:

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

Assembly code:

```assembly
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

**While Loop:**

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

x86-64:
loopTop:  testq %rax, %rax
          je   loopDone
          <loop body code>
          jmp  loopTop

loopDone:

**Do-while Loop:**

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

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

loopDone:

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

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

x86-64:
loopTop:  testq %rax, %rax
          je   loopDone
          <loop body code>
          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`
Practice Question 2

- The following is assembly code for a for-loop; identify the corresponding parts (Init, Test, Update)
  - i → %eax, x → %rdi, y → %esi

```assembly
movl $0, %eax
.L2: cmpl %esi, %eax
    jge .L4
    movslq %eax, %rdx
    leaq (%rdi,%rdx,4), %rcx
    movl (%rcx), %edx
    addl $1, %edx
    movl %edx, (%rcx)
    addl $1, %eax
    jmp .L2
.L4:  
```