x86-64 Programming III
CSE 351 Autumn 2019

Instructor:
Justin Hsia

Teaching Assistants:
Andrew Hu
Antonio Castelli
Cosmo Wang
Diya Joy
Ivy Yu
Kaelin Laundry
Maurice Montag
Melissa Birchfield
Millicent Li
Suraj Jagadeesh

http://xkcd.com/1652/
Administrivia

- Lab 2 due next Friday (10/25)
  - Optional GDB Tutorial homework on Gradescope

- Section tomorrow on Assembly
  - Bring your midterm reference sheets!

- Midterm: 10/28, 5:30 pm in KNE 130
  - You will be provided a fresh reference sheet
  - You get 1 handwritten, double-sided cheat sheet (letter-size)
  - Midterm Clobber Policy: replace midterm score with score on midterm portion of the final if you “do better”
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!
  - `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 `steppi/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 \( (\text{op}) \)
  - 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>addq 5, (p)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>je:</td>
<td>*p+5 == 0</td>
<td>d (op) s == 0</td>
</tr>
<tr>
<td>jne:</td>
<td>*p+5 != 0</td>
<td>d (op) s != 0</td>
</tr>
<tr>
<td>jg:</td>
<td>*p+5 &gt; 0</td>
<td>d (op) s &gt; 0</td>
</tr>
<tr>
<td>jl:</td>
<td>*p+5 &lt; 0</td>
<td>d (op) s &lt; 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>orq a, b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>je:</td>
<td>b</td>
<td>a == 0</td>
</tr>
<tr>
<td>jne:</td>
<td>b</td>
<td>a != 0</td>
</tr>
<tr>
<td>jg:</td>
<td>b</td>
<td>a &gt; 0</td>
</tr>
<tr>
<td>jl:</td>
<td>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></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-a &gt; 0U</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>b-a &lt; 0U</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>
Choosing instructions for conditionals

### Table: Comparisons and Tests

<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>

### Example Code

```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
```

### Register Use(s)

<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>
Polling Question

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; 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...

```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
    subq %rsi, %rax
    ret

.L4:  # x <= y:
    movq %rsi, %rax
    subq %rdi, %rax
    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>“Equal”</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != 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 &gt;=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 &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= 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>

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

```
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

- C/Java code:
  ```c
  while ( Test ) {
      Body
  }
  ```

- Goto version:
  ```c
  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**:

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

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

```c
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`
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

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

Jump Table

<table>
<thead>
<tr>
<th>JTab</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>•</th>
<th>•</th>
<th>Targ_{n-1}</th>
</tr>
</thead>
</table>

Jump Targets

<table>
<thead>
<tr>
<th>Targ0</th>
<th>Code Block 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1</td>
<td>Code Block 1</td>
</tr>
<tr>
<td>Targ2</td>
<td>Code Block 2</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Targ_{n-1}</td>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>
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;
```