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 `steipi/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><strong>addq 5, (p)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>d (op) s == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>d (op) s != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>d (op) s &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>d (op) s &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>d (op) s &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>”Less or equal”</td>
<td>d (op) s &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>d (op) s &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>d (op) s &lt; 0U</td>
</tr>
</tbody>
</table>

**Example: addq 5, (p)**

```asm
addq 5, (p)
je: *p+5 == 0
jne: *p+5 != 0
jg: *p+5 > 0
jl: *p+5 < 0
```

**Example: orq a, b**

```asm
orq a, b
je: b | a == 0
jne: b | a != 0
jg: b | a > 0
jl: b | a < 0
```
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} “Equal”</td>
<td>( b == a )</td>
<td>( b&amp;a == 0 )</td>
</tr>
<tr>
<td>\texttt{jne} “Not equal”</td>
<td>( b != a )</td>
<td>( b&amp;a != 0 )</td>
</tr>
<tr>
<td>\texttt{js} “Sign” (negative)</td>
<td>( b-a &lt; 0 )</td>
<td>( b&amp;a &lt; 0 )</td>
</tr>
<tr>
<td>\texttt{jns} (non-negative)</td>
<td>( b-a &gt;=0 )</td>
<td>( b&amp;a &gt;= 0 )</td>
</tr>
<tr>
<td>\texttt{jg} “Greater”</td>
<td>( b &gt; a )</td>
<td>( b&amp;a &gt; 0 )</td>
</tr>
<tr>
<td>\texttt{jge} “Greater or equal”</td>
<td>( b &gt;= a )</td>
<td>( b&amp;a &gt;= 0 )</td>
</tr>
<tr>
<td>\texttt{jl} “Less”</td>
<td>( b &lt; a )</td>
<td>( b&amp;a &lt; 0 )</td>
</tr>
<tr>
<td>\texttt{jle} “Less or equal”</td>
<td>( b &lt;= a )</td>
<td>( b&amp;a &lt;= 0 )</td>
</tr>
<tr>
<td>\texttt{ja} “Above” (unsigned &gt;)</td>
<td>( b &gt; a )</td>
<td>( b&amp;a &gt; 0U )</td>
</tr>
<tr>
<td>\texttt{jb} “Below” (unsigned &lt;)</td>
<td>( b &lt; a )</td>
<td>( b&amp;a &lt; 0U )</td>
</tr>
</tbody>
</table>
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>

```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
```
Question

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

Register Use(s)
%rdi 1st argument (x)
%rsi 2nd argument (y)
%rax return value

Vote at http://pollev.com/rea

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...

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>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 && x == y) {
    return 1;
} else {
    return 2;
}

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

HTTPS://GODBOLT.ORG/Z/J72AEN
Labels

- 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

```
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 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

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

What are the Goto versions of the following?

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

C/Java code:

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

Goto version:

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

```
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*
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];
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 ≤ 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 \( w \)

**Switch_eg:**

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

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

Indirect jump

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

```
.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:      // .L7
    w -= z;
    break;
  case 6:      // .L7
    w = 2;
  default:     // .L8
    w = 2;
}
```

8-byte memory alignment

this data is 64-bits wide

declaring data, not instructions
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;
    ....
}
```

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;

    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;
    ...
}
```

### 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:
    movq    %rsi, %rax      # y in rax
    cqto    # Div prep
    idivq   %rcx            # y/z
    jmp     .L6            # goto merge
.L9:
    movl    $1, %eax       # w = 1
    addq    %rcx, %rax     # w += z
.L6:
    ret
```

Code Blocks (rest)

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

<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