x86-64 Programming III
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/1652/
Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-13
- Lab 1b due **tonight** at 11:59pm (7/13)
  - Submit aisle_manager.c, store_client.c, and lab1Breflect.txt
  - Can still use late days until 7/15
- Unit Summary 1 due Friday (7/17) – **11:59pm**
  - Can still use late days until 7/20
- Mid-quarter Survey due Friday (7/17) – **11:59pm**
  - Submit via Canvas!
- hw8, hw9, hw10 now due Monday (7/20) – **10:30am**
  - hw11 also due Monday (7/20)
  - See course schedule for original/suggested deadlines
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 of Instruction Pairs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>je</strong></td>
<td>“Equal”</td>
<td>(d \ (\text{op}) \ s \ == \ 0)</td>
</tr>
<tr>
<td><strong>jne</strong></td>
<td>“Not equal”</td>
<td>(d \ (\text{op}) \ s \ != \ 0)</td>
</tr>
<tr>
<td><strong>js</strong></td>
<td>“Sign” (negative)</td>
<td>(d \ (\text{op}) \ s &lt; 0)</td>
</tr>
<tr>
<td><strong>jns</strong></td>
<td>(non-negative)</td>
<td>(d \ (\text{op}) \ s &gt;= 0)</td>
</tr>
<tr>
<td><strong>jg</strong></td>
<td>“Greater”</td>
<td>(d \ (\text{op}) \ s &gt; 0)</td>
</tr>
<tr>
<td><strong>jge</strong></td>
<td>“Greater or equal”</td>
<td>(d \ (\text{op}) \ s &gt;= 0)</td>
</tr>
<tr>
<td><strong>jl</strong></td>
<td>“Less”</td>
<td>(d \ (\text{op}) \ s &lt; 0)</td>
</tr>
<tr>
<td><strong>jle</strong></td>
<td>”Less or equal”</td>
<td>(d \ (\text{op}) \ s &lt;= 0)</td>
</tr>
</tbody>
</table>
| **ja** | “Above” (unsigned 
| | >) | \(d \ (\text{op}) \ s > 0\) |
| **jb** | “Below” (unsigned <) | \(d \ (\text{op}) \ s < 0\) |
| **orq** a, b | | |
| je: | b | a == 0 |
| jne: | b | a != 0 |
| jg: | b | a > 0 |
| jl: | b | a < 0 |

\(\text{addq 5, (p)}\)

\(\text{je: } *p+5 == 0\)
\(\text{jne: } *p+5 != 0\)
\(\text{jg: } *p+5 > 0\)
\(\text{jl: } *p+5 < 0\)

\(\text{orq a}, \ b\)
\(\text{je: } b | a == 0\)
\(\text{jne: } b | a != 0\)
\(\text{jg: } b | a > 0\)
\(\text{jl: } b | a < 0\)
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>Instruction</th>
<th>First options</th>
<th>\texttt{cmp} a,b</th>
<th>\texttt{test} a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>&quot;Equal&quot;</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne</td>
<td>&quot;Not equal&quot;</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js</td>
<td>&quot;Sign&quot; (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>&quot;Greater&quot;</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>&quot;Greater or equal&quot;</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>&quot;Less&quot;</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>&quot;Less or equal&quot;</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>&quot;Above&quot; (unsigned &gt;)</td>
<td>b &gt;U a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>&quot;Below&quot; (unsigned &lt;)</td>
<td>b &lt;U a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

\texttt{cmpq} 5, (p)
- \texttt{je}: *p == 5
- \texttt{jne}: *p != 5
- \texttt{jg}: *p > 5
- \texttt{jl}: *p < 5

\texttt{testq} a, a
- \texttt{je}: a == 0
- \texttt{jne}: a != 0
- \texttt{jg}: a > 0
- \texttt{jl}: a < 0

\texttt{testb} a, 0x1
- \texttt{je}: a_{LSB} == 0
- \texttt{jne}: a_{LSB} == 1
Choosing instructions for conditionals

We want to jump when \( x \geq 3 \)

\[
\text{cmpq } 3, \%rdi \\
\text{jge } T2
\]

where to jump

<table>
<thead>
<tr>
<th>Instruction</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: Choosing Instructions for Conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Test</th>
<th>Return Value</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>

### Code Snippet

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

**Register Used**

<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>

**Assembly Code**

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

```
Polling Question [Asm III - a]

We want to jump when \( x < y \)

<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/phiones

A. \texttt{cmpq} \%rsi, \%rdi \texttt{jle .L4}  
B. \texttt{cmpq} \%rsi, \%rdi \texttt{jg .L4}  
C. \texttt{testq} \%rsi, \%rdi \texttt{jle .L4}  
D. \texttt{testq} \%rsi, \%rdi \texttt{jg .L4}  
E. We’re lost...

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

absdiff:

\begin{verbatim}
    _____________________________
    # x > y:  
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret

    _____________________________  # x <= y:  
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
\end{verbatim}
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>&quot;Equal&quot;</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>&quot;Not equal&quot;</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>&quot;Sign&quot; (negative)</td>
<td>b-a &lt; 0, b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0, b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>&quot;Greater&quot;</td>
<td>b &gt; a, b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>&quot;Greater or equal&quot;</td>
<td>b &gt;= a, b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>&quot;Less&quot;</td>
<td>b &lt; a, b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>&quot;Less or equal&quot;</td>
<td>b &lt;= a, b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>&quot;Above&quot; (unsigned &gt;)</td>
<td>b &gt; a, b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>&quot;Below&quot; (unsigned &lt;)</td>
<td>b &lt; a, 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 jump if (%a&%b1)=2>0

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

T2: # else
    movq $2, %rax
    ret

https://godbolt.org/z/GNxpqv
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

```plaintext
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
Compiling Loops

C/Java code:

```java
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:

\[
\text{while ( Test ) } \{
\text{Body}
\}
\]

Goto version:

\[
\text{Loop: if ( !Test ) goto Exit;}
\text{Body}
\text{goto Loop;}
\text{Exit:}
\]

- 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>
}
```

x86-64:
```assembly
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:
```assembly
loopTop:    <loop body code>
            testq  %rax, %rax
            jne   loopTop
            fall through/exit
loopDone:
```

While Loop (ver. 2):

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

x86-64:
```assembly
loopTop:    testq  %rax, %rax
            je    loopDone
            <loop body code>
            testq  %rax, %rax
            jne   loopTop
            fall through/exit
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*
Jump Table Structure

Switch Form

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

Approximate Translation

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

Jump Table

- Targ0
- Targ1
- Targ2
- Targn-1

Jump Targets

- Code Block 0
- Code Block 1
- Code Block 2
- Code Block n-1

Jump Table:

- JTab: address of jump table
- Targ0: target = JTab[0]; goto target;
- Targ1: target = JTab[1]; goto target;
- Targ2: target = JTab[2]; goto target;
- Targn-1: target = JTab[n-1]; 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 $x \leq 6$:

```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;
}
```

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

Note compiler chose to not initialize `w`

Jump above – unsigned > catches negative default cases

Take a look!
https://godbolt.org/z/aY24el

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

Negative numbers are large unsigned values
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
```

Indirect jump
```
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**

```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
```
BONUS SLIDES

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

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

8-byte memory alignment

this data is 64-bits wide

declaring data, not instructions

8-byte memory alignment

this data is 64-bits wide
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;
```

```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;
  ...
}
```

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

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

.L6:
  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)

```
switch (x) {
    ... 
    case 5:  // .L7
        w -= z;
        break;
    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
```
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 `steipi/nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining `memory` (`x`)