x86-64 Programming III
CSE 351 Winter 2021

Instructor:
Mark Wyse

Teaching Assistants:
Kyrie Dowling
Catherine Guevara
Ian Hsiao
Jim Limprasert
Armin Magness
Allie Pfleger
Cosmo Wang
Ronald Widjaja

http://xkcd.com/1652/
Administrivia

❖ Study Guide 1 due Friday (1/29)
  ▪ may use late days if needed
❖ hw9 due Friday, hw10 due Monday
❖ Lab 2 due next Friday (2/5)

❖ Section tomorrow on Assembly
  ▪ turn in worksheet by Friday 11:59pm PST
Reading Review

❖ Terminology:
  ▪ Label, jump target
  ▪ Program counter
  ▪ Jump table, indirect jump

❖ Any questions from reading?
  ▪ can also post reading questions to Ed Discussion
Aside: \texttt{movz} and \texttt{movs}

\texttt{movz} \_\_ src, regDest \quad \# \text{Move with } \underline{\text{zero}} \text{ extension}
\texttt{movs} \_\_ src, regDest \quad \# \text{Move with } \underline{\text{sign}} \text{ extension}

\begin{itemize}
  \item Copy from a \textit{smaller} source value to a \textit{larger} destination
  \item Source can be memory or register; Destination \textit{must} be a register
  \item Fill remaining bits of dest with \texttt{zero} (\texttt{movz}) or \texttt{sign bit} (\texttt{movs})
\end{itemize}

\texttt{movz SD} / \texttt{movs SD}:

\begin{itemize}
  \item \texttt{S} – size of source (\texttt{b} = 1 byte, \texttt{w} = 2)
  \item \texttt{D} – size of dest (\texttt{w} = 2 bytes, \texttt{l} = 4, \texttt{q} = 8)
\end{itemize}

\textbf{Example:}
\texttt{movzbq} %al, %rbx
\begin{verbatim}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \leftarrow %rax
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF \leftarrow %rbx
\end{verbatim}
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)

movzSD / movsSD:

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.
Using Condition Codes: Jumping

❖ j* Instructions

- Jumps to target (an address) based on condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp target</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je target</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne target</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js target</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns target</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg target</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge target</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl target</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle target</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja target</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;“)</td>
</tr>
<tr>
<td>jb target</td>
<td>CF</td>
<td>Below (unsigned “&lt;“)</td>
</tr>
</tbody>
</table>

ZF: (r == 0)
SF: (r < 0), MSB == 1
CF: unsigned overflow
OF: signed overflow
Using Condition Codes: Setting

- **set* Instructions**
  - Set low-order byte of `dst` to 0 or 1 based on condition codes
  - Does not alter remaining 7 bytes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sete dst</code></td>
<td><code>ZF</code></td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>setne dst</code></td>
<td><code>~ZF</code></td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>sets dst</code></td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns dst</code></td>
<td><code>~SF</code></td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>setg dst</code></td>
<td><code>~(SF^OF) &amp; ~ZF</code></td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>setge dst</code></td>
<td><code>~(SF^OF)</code></td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>setl dst</code></td>
<td><code>(SF^OF)</code></td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>setle dst</code></td>
<td>`(SF^OF)</td>
<td>ZF`</td>
</tr>
<tr>
<td><code>seta dst</code></td>
<td><code>~CF&amp;~ZF</code></td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><code>setb dst</code></td>
<td><code>CF</code></td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>

**Condition Codes**

- `ZF`: `r == 0`
- `SF`: `r < 0`, MSB == 1
- `CF`: unsigned overflow
- `OF`: signed overflow
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*

```plaintext
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>je:    *p+5 == 0</code></td>
<td>&quot;Equal&quot;</td>
<td><code>d(op) s == 0</code></td>
</tr>
<tr>
<td><code>jne:  *p+5 != 0</code></td>
<td>&quot;Not equal&quot;</td>
<td><code>d(op) s != 0</code></td>
</tr>
<tr>
<td><code>js:    *p+5 &lt; 0</code></td>
<td>&quot;Sign&quot; (negative)</td>
<td><code>d(op) s &lt; 0</code></td>
</tr>
<tr>
<td><code>jns:   *p+5 &gt;= 0</code></td>
<td>(non-negative)</td>
<td><code>d(op) s &gt;= 0</code></td>
</tr>
<tr>
<td><code>jg:    *p+5 &gt; 0</code></td>
<td>&quot;Greater&quot;</td>
<td><code>d(op) s &gt; 0</code></td>
</tr>
<tr>
<td><code>jl:    *p+5 &lt;= 0</code></td>
<td>&quot;Less or equal&quot;</td>
<td><code>d(op) s &lt;= 0</code></td>
</tr>
<tr>
<td><code>orq a, b</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`je:    b</td>
<td>a == 0`</td>
<td>&quot;Equal&quot;</td>
</tr>
<tr>
<td>`jne:   b</td>
<td>a != 0`</td>
<td>&quot;Not equal&quot;</td>
</tr>
<tr>
<td>`jg:    b</td>
<td>a &gt; 0`</td>
<td>&quot;Greater&quot;</td>
</tr>
<tr>
<td>`jl:    b</td>
<td>a &lt; 0`</td>
<td>&quot;Less or equal&quot;</td>
</tr>
</tbody>
</table>
```

orq a, b
```
Choosing instructions for conditionals

- **Reminder:** `cmp` is like `sub`, `test` is like `and`

  - Result is not stored anywhere

<table>
<thead>
<tr>
<th></th>
<th><code>cmp a,b</code></th>
<th><code>test a,b</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code></td>
<td>“Equal”</td>
<td><code>b == a</code></td>
</tr>
<tr>
<td><code>jne</code></td>
<td>“Not equal”</td>
<td><code>b != a</code></td>
</tr>
<tr>
<td><code>js</code></td>
<td>“Sign” (negative)</td>
<td><code>b-a &lt; 0</code></td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td><code>b-a &gt;=0</code></td>
</tr>
<tr>
<td><code>jg</code></td>
<td>“Greater”</td>
<td><code>b &gt; a</code></td>
</tr>
<tr>
<td><code>jge</code></td>
<td>“Greater or equal”</td>
<td><code>b &gt;= a</code></td>
</tr>
<tr>
<td><code>jl</code></td>
<td>“Less”</td>
<td><code>b &lt; a</code></td>
</tr>
<tr>
<td><code>jle</code></td>
<td>“Less or equal”</td>
<td><code>b &lt;= a</code></td>
</tr>
<tr>
<td><code>ja</code></td>
<td>“Above” (unsigned &gt;)</td>
<td><code>b &gt; U a</code></td>
</tr>
<tr>
<td><code>jb</code></td>
<td>“Below” (unsigned &lt;)</td>
<td><code>b &lt; U a</code></td>
</tr>
</tbody>
</table>

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

testq a, a
  je:  a == 0
  jne: a != 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>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; u a</td>
<td>b&amp;a &gt; 0u</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; u a</td>
<td>b&amp;a &lt; 0u</td>
</tr>
</tbody>
</table>

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>

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

ret

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

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

<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. \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...

```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 “Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne “Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js “Sign” (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 “Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge “Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl “Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle “Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja “Above” (unsigned &gt;)</td>
<td>b &gt; U a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb “Below” (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;
}

```assembly
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/Tfrv33
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:

```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

**While Loop:**

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

x86-64:
```c
loopTop:  testq  %rax, %rax
          je     loopDone
          jmp   loopTop
loopDone:
```

**Do-while Loop:**

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

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

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

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

x86-64:
```c
loopTop:  testq  %rax, %rax
          je     loopDone
          jmp   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)

- \( i \rightarrow \%eax, \ x \rightarrow \%rdi, \ y \rightarrow \%esi \)

```
  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:
```
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

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

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];
go to target;
```
Jump Table Structure

C code:

```c
switch (x) {
    case 1: <code> break;
    case 2: <code>
    case 3: <code> break;
    case 5:
    case 6: <code> break;
    case 7: <code> break;
    default: <code>
}
```

Use the jump table when $x \leq 7$:

```c
if (x <= 7)
    target = JTab[x];
    goto target;
else
    goto default;
```
Switch Statement Example

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

Note compiler chose to not initialize `w`

```
switch_ex:
    movq  %rdx, %rcx
    cmpq $7, %rdi        # x:7
    ja   .L9             # default
    jmp  *.L4(,%rdi,8)   # jump table
```

Jump above – unsigned > catches negative default cases

Take a look!
https://godbolt.org/z/Y9Kerb
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 .L9   # x = 0
        .quad .L8   # x = 1
        .quad .L7   # x = 2
        .quad .L10  # x = 3
        .quad .L9   # x = 4
        .quad .L5   # x = 5
        .quad .L5   # x = 6
        .quad .L3   # x = 7
]
```

```
switch_ex:
    movq %rdx, %rcx
    cmpq $7, %rdi     # x:7
    ja .L9           # default
    jmp * .L4(%rdi,8) # jump table
```

*Indirect jump*
Assembly Setup Explanation

❖ Table Structure
  ▪ Each target requires 8 bytes (address)
  ▪ Base address at .L4

❖ Direct jump: jmp .L9
  ▪ Jump target is denoted by label .L9

❖ 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 ≤ x ≤ 7

Jump table

```
.section .rodata
.align 8
.L4:
  .quad .L9  # x = 0
  .quad .L8  # x = 1
  .quad .L7  # x = 2
  .quad .L10 # x = 3
  .quad .L9  # x = 4
  .quad .L5  # x = 5
  .quad .L5  # x = 6
  .quad .L3  # x = 7
```
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 stepi/nexti)
  ▪ Printing out expressions (print – works with regs & vars)
  ▪ Examining memory (x)