x86-64 Programming III
CSE 351 Autumn 2022

Instructor:
Justin Hsia

Teaching Assistants:
Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/1652/
Relevant Course Information

❖ Lab 1a regrade requests open on Gradescope
❖ Lab 1b submissions close tonight
❖ Lab 2 due next Friday (10/28)

❖ Section tomorrow on Assembly
  ▪ Use the midterm reference sheet, bring your laptop!
  ▪ Optional GDB Tutorial slides and Lab 2 phase 1 walkthrough

❖ Midterm (take home, 11/3–11/5)
  ▪ Make notes and use the midterm reference sheet
  ▪ Form study groups and look at past exams!
Move extension: \texttt{movz/movs} (Review)

\begin{itemize}
  \item \texttt{movz} $\_\_\_ \ src,\ regDest$ \hspace{1cm} \# Move with \texttt{zero} extension
  \item \texttt{movs} $\_\_\_ \ src,\ regDest$ \hspace{1cm} \# Move with \texttt{sign} extension
\end{itemize}

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

Example:

\texttt{movzbq} $%al,\ %rbx$ \hspace{2cm}
\begin{tabular}{c}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \hspace{1cm} \%rax \\
0x00 0x00 0x00 0x00 0x00 0x00 0xFF \hspace{1cm} \%rbx
\end{tabular}
Move extension: movz/movs (Review)

- move extension:
  - movz
  - 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 stepi/nexti)
  ▪ Printing out expressions (print – works with regs & vars)
  ▪ Examining memory (x)
x86 Control Flow

❖ Condition codes
❖ Conditional and unconditional branches
❖ Loops
❖ Switches
Processor State (x86-64, partial)

- Information about currently executing program
  - Temporary data (%rax, ...)
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, ...)
  - Status of recent tests (CF, ZF, SF, OF)
    - Single bit registers:

```
%rax  %r8
%rbx  %r9
%rcx  %r10
%rdx  %r11
%rsi  %r12
%rdi  %r13
%rsp  %r14
%rbp  %r15
```

- Program Counter (instruction pointer)
- Condition Codes:
  - CF
  - ZF
  - SF
  - OF

Current top of the Stack
Condition Codes (Implicit, RD9)

- Implicitly set by arithmetic operations
  - (think of it as side effects)
  - Example: \texttt{addq src, dst} $\leftrightarrow$ \texttt{r = d+s}

- \texttt{CF=1} if carry out from MSB (\textit{unsigned} overflow)
- \texttt{ZF=1} if \texttt{r==0}
- \texttt{SF=1} if \texttt{r<0} (if MSB is 1)
- \texttt{OF=1} if \textit{signed} overflow
  \[(s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r>=0)\]

- \textit{Not} set by \texttt{lea} instruction (beware!)
Condition Codes (Explicit: Compare, RD9)

- Explicitly set by Compare instruction
  - `cmpq src1, src2`
  - `cmpq a, b` sets flags based on $b-a$, but doesn’t store

- $CF=1$ if carry out from MSB (good for unsigned comparison)
- $ZF=1$ if $a==b$
- $SF=1$ if $(b-a)<0$ (if MSB is 1)
- $OF=1$ if signed overflow
  
  $$
  (a>0 \land b<0 \land (b-a)>0) \lor \ \\
  (a<0 \land b>0 \land (b-a)<0)
  $$

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>
Condition Codes (Explicit: Test, RD9)

- **Explicitly set by Test instruction**
  - `testq src2, src1`
  - `testq a, b` sets flags based on `a & b`, but doesn’t store
    - Useful to have one of the operands be a *mask*
  - Can’t have carry out (CF) or overflow (OF)
  - **ZF=1** if `a & b == 0`
  - **SF=1** if `a & b < 0` (signed)

**Carry Flag**  **Zero Flag**  **Sign Flag**  **Overflow Flag**
Example Condition Code Setting

❖ Assuming that `%al = 0x80` and `%bl = 0x81`, which flags (CF, ZF, SF, OF) are set when we execute `cmpb %al, %bl`?
Using Condition Codes: Jumping (RD9)

❖ **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 <em>target</em></td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je <em>target</em></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne <em>target</em></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js <em>target</em></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns <em>target</em></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg <em>target</em></td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge <em>target</em></td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl <em>target</em></td>
<td>(SF^OF')</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle <em>target</em></td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja <em>target</em></td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>jb <em>target</em></td>
<td>CF</td>
<td>Below (unsigned “&lt;“)</td>
</tr>
</tbody>
</table>
Using Condition Codes: Setting (RD9)

- **set* Instructions**
  - Set low-order byte of \texttt{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>sete \texttt{dst}</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne \texttt{dst}</td>
<td>\sim ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets \texttt{dst}</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns \texttt{dst}</td>
<td>\sim SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg \texttt{dst}</td>
<td>\sim(SF^OF) &amp; \sim ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge \texttt{dst}</td>
<td>\sim(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl \texttt{dst}</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle \texttt{dst}</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta \texttt{dst}</td>
<td>\sim CF &amp; \sim ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb \texttt{dst}</td>
<td>CF</td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **set* Instructions**
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g., %al) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

```
int gt(long x, long y)
{
    return 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>

```
cmpq  %rsi, %rdi  #
setg  %al    #
movzbl %al, %eax  #
ret
```
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>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td>( *p+5 == 0 )</td>
<td>( d (op) s == 0 )</td>
</tr>
<tr>
<td><code>je:</code></td>
<td>( *p+5 == 0 )</td>
<td>( d (op) s == 0 )</td>
</tr>
<tr>
<td><code>jne:</code></td>
<td>( *p+5 != 0 )</td>
<td>( d (op) s != 0 )</td>
</tr>
<tr>
<td><code>jg:</code></td>
<td>( *p+5 &gt; 0 )</td>
<td>( d (op) s &lt; 0 )</td>
</tr>
<tr>
<td><code>jl:</code></td>
<td>( *p+5 &lt; 0 )</td>
<td>( d (op) s &gt; 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orq a, b</code></td>
<td>( b</td>
<td>a == 0 )</td>
</tr>
<tr>
<td><code>je:</code></td>
<td>( b</td>
<td>a == 0 )</td>
</tr>
<tr>
<td><code>jne:</code></td>
<td>( b</td>
<td>a != 0 )</td>
</tr>
<tr>
<td><code>jg:</code></td>
<td>( b</td>
<td>a &gt; 0 )</td>
</tr>
<tr>
<td><code>jl:</code></td>
<td>( b</td>
<td>a &lt; 0 )</td>
</tr>
</tbody>
</table>

\( op \): operation result

\( d \): destination register

\( s \): source register

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

### `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></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 &gt; u a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>b &lt; u a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

Register | Use(s)  
%rdi    | argument x  
%rsi    | argument y  
%rax    | return value

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

```c
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>1\textsuperscript{st} argument ((x))</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument ((y))</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

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

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

absdiff:

```asm
.L4:    # x \leq y:
    movq \%rsi, %rax
    subq %rdi, %rax
    ret
```

```asm
.L4:    # x > y:
    movq \%rdi, %rax
    subq %rsi, %rax
    ret
```
Reading Review

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

❖ Questions from the Reading?
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

```
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 (Review)

C/Java code:

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

Assembly code:

```asm
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 (Review)

**While Loop:**

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

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

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

C:
```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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>movl $0, %eax</td>
</tr>
<tr>
<td>2</td>
<td>.L2: cmpl %esi, %eax</td>
</tr>
<tr>
<td>3</td>
<td>jge  .L4</td>
</tr>
<tr>
<td>4</td>
<td>movslq %eax, %rdx</td>
</tr>
<tr>
<td>5</td>
<td>leaq (%rdi,%rdx,4), %rcx</td>
</tr>
<tr>
<td>6</td>
<td>movl (%rcx), %edx</td>
</tr>
<tr>
<td>7</td>
<td>addl $1, %edx</td>
</tr>
<tr>
<td>8</td>
<td>movl %edx, (%rcx)</td>
</tr>
<tr>
<td>9</td>
<td>addl $1, %eax</td>
</tr>
<tr>
<td>10</td>
<td>jmp  .L2</td>
</tr>
<tr>
<td>11</td>
<td>.L4:</td>
</tr>
</tbody>
</table>

\[ \text{for}(________;________;________) \]
Summary

❖ Control flow in x86 determined by Condition Codes
  ▪ Showed Carry, Zero, Sign, and Overflow, though others exist
  ▪ Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  ▪ Set instructions read out flag values
  ▪ Jump instructions use flag values to determine next instruction to execute
  ▪ Most control flow constructs (e.g., if-else, for-loop, while-loop) can be implemented in assembly using combinations of conditional and unconditional jumps