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: movz/movs (Review)

- 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:

\[
\text{movzbq} \ %al, \ %rbx
\]

\[
\begin{array}{c}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \\
\hline
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF
\end{array}
\]

\%rax

\%rbx
Move extension: \texttt{movz/movs} (Review)

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

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

\texttt{movzSD} / \texttt{movsSD}:
\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}

Example:
\texttt{movsbl} (%rax), %ebx

\begin{itemize}
  \item \texttt{Copy 1 byte from memory into 8-byte register & sign extend it}
\end{itemize}

\textbf{Note:} In x86-64, \textit{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) "flags"
  - Single bit registers:
    - \%rax
    - \%rbx
    - \%rcx
    - \%rdx
    - \%r8
    - \%r9
    - \%r10
    - \%r11
    - \%r12
    - \%r13
    - \%r14
    - \%r15

Registers
\[
\begin{array}{ll}
\%rax & \%r8 \\
\%rbx & \%r9 \\
\%rcx & \%r10 \\
\%rdx & \%r11 \\
\%r8 & \%r12 \\
\%r9 & \%r13 \\
\%r10 & \%r14 \\
\%r11 & \%r15 \\
\end{array}
\]

Program Counter (instruction pointer)
\[
\%rip
\]

Condition Codes
\[
\begin{array}{cccc}
CF & ZF & SF & OF \\
\end{array}
\]

Carry Zero Sign Overflow
Condition Codes (Implicit, RD9)

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

- **CF=1** if carry out from MSB (*unsigned* overflow)
- **ZF=1** if `r==0`
- **SF=1** if `r<0` (if MSB is 1)
- **OF=1** if *signed* overflow
  \[
  (s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r\geq 0)
  \]
- **Not set by lea instruction** (beware!)
Condition Codes (Explicit: Compare, RD9)

- Explicitly set by **Compare** instruction
  - `cmpq src1, src2` like `subq a, b → b - a`
  - `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 (b-a==0)`
    - **SF=1** if `(b-a)<0` (if MSB is 1)
    - **OF=1** if *signed* overflow
      - `(a>0 && b<0 && (b-a)>0) || (a<0 && b>0 && (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` like `andq a, b`
  - `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)
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`?

```
%bl - %al = %bl - ~%al + 1
~%al = ~0x80 = 0x7F
```

```
<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>1</td>
</tr>
<tr>
<td>ZF</td>
<td>0</td>
</tr>
<tr>
<td>SF</td>
<td>0</td>
</tr>
<tr>
<td>OF</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
%bl + (~%al + 1) = 0x80 + 0x7F + 1 = 0x01
0x01 ≠ 0
```

```
%bl + (~%al) < 0
```

```
%bl + (~%al) > 0
```
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 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>
Using Condition Codes: Setting (RD9)

- 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>sete dst</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne dst</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets dst</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns dst</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg dst</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge dst</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl dst</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle dst</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta dst</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb 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

```c
int gt(long x, long y)
{
    return x > y;  // x-y > 0
}
```

---

<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>
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation \((\text{o}_p)\)
  - Conditionals are comparisons against 0

- Come in instruction pairs

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq 5, (p)</code></td>
<td>(\text{add} \text{q} 5, (p))</td>
<td>(d (\text{op}) s \text{ == } 0)</td>
</tr>
<tr>
<td><code>je</code></td>
<td>Equal</td>
<td>(d (\text{op}) s \text{ == } 0)</td>
</tr>
<tr>
<td><code>jne</code></td>
<td>Not equal</td>
<td>(d (\text{op}) s \text{ != } 0)</td>
</tr>
<tr>
<td><code>js</code></td>
<td>Sign (negative)</td>
<td>(d (\text{op}) s &lt; 0)</td>
</tr>
<tr>
<td><code>jns</code></td>
<td>(non-negative)</td>
<td>(d (\text{op}) s \geq 0)</td>
</tr>
<tr>
<td><code>jg</code></td>
<td>Greater</td>
<td>(d (\text{op}) s &gt; 0)</td>
</tr>
<tr>
<td><code>jge</code></td>
<td>Greater or equal</td>
<td>(d (\text{op}) s \geq 0)</td>
</tr>
<tr>
<td><code>jl</code></td>
<td>Less</td>
<td>(d (\text{op}) s &lt; 0)</td>
</tr>
<tr>
<td><code>jle</code></td>
<td>Less or equal</td>
<td>(d (\text{op}) s \leq 0)</td>
</tr>
<tr>
<td><code>ja</code></td>
<td>Above (unsigned &gt;)</td>
<td>(d (\text{op}) s &gt; 0\U)</td>
</tr>
<tr>
<td><code>jb</code></td>
<td>Below (unsigned &lt;)</td>
<td>(d (\text{op}) s &lt; 0\U)</td>
</tr>
</tbody>
</table>
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>cmp $a,b$</th>
<th>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>$b &gt;_U a$</td>
<td>$b&amp;a &gt; 0U$</td>
</tr>
<tr>
<td>\texttt{jb} “Below” (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}: $a_{\text{LSB}} == 0$
- \texttt{jne}: $a_{\text{LSB}} == 1$
# Choosing instructions for conditionals

<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>

## Instructions

- **cmpq $3, %rdi**
  - **jge** T2

## Conditions

- **je** “Equal”
  - `b == a`
  - `b&a == 0`
- **jne** “Not equal”
  - `b != a`
  - `b&a != 0`
- **js** “Sign” (negative)
  - `b-a < 0`
  - `b&a < 0`
- **jns** (non-negative)
  - `b-a >= 0`
  - `b&a >= 0`
- **jg** “Greater”
  - `b > a`
  - `b&a > 0`
- **jge** “Greater or equal”
  - `b >= a`
  - `b&a >= 0`
- **jl** “Less”
  - `b < a`
  - `b&a < 0`
- **jle** “Less or equal”
  - `b <= a`
  - `b&a <= 0`
- **ja** “Above” (unsigned >)
  - `b > a`
  - `b&a > 0`
- **jb** “Below” (unsigned <)
  - `b < a`
  - `b&a < 0`

## Code Example

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

- **T1**: # x < 3: (if)
  - movq $1, %rax
  - ret

- **T2**: # !(x < 3): (else)
  - 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. `cmpq %rsi, %rdi`  \( x-y \)
   `jle .L4`

B. `cmpq %rsi, %rdi`  \( x-y \)
   `jg .L4`

C. `testq %rsi, %rdi`  \( x\&y \)
   `jle .L4`

D. `testq %rsi, %rdi`  \( x\&y \)
   `jg .L4`

E. We’re lost…

absdiff:

\[ \text{\underline{\text{\hspace{2cm} # x > y:}}} \]

\[ \text{\underline{\text{\hspace{2cm}} \text{\hspace{2cm} # x <= y:}}} \]

- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
- \text{\underline{\text{\hspace{2cm} # x <= y:}}} \]
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

```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 (Review)

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 (Review)

**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
loopDone:
```

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

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

x86-64:

```assembly
loopTop:  testq %rax, %rax
          je  loopDone
          do-while loop { <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)
  - \( i \rightarrow \%eax, \ x \rightarrow \%rdi, \ y \rightarrow \%esi \)

```assembly
.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:    movl  $0, \%eax
```

```plaintext
for( int i = 0 ; i < y ; i++ )
```

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