x86-64 Programming II
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/99/
Administrivia

- Lab 2 (x86-64) released today, due 2/5
  - Learn to read x86-64 assembly and use GDB
- Lecture readings – due at 11:00am PST
- Submissions that fail the autograder get a ZERO
  - No excuses – make full use of tools & Gradescope’s interface
  - Some leeway was given on Lab 1, do not expect the same leniency moving forward
- hw8 due Wednesday, hw9 due Friday
- Study Guide 1 – due Friday 1/29
Extra Credit

❖ All labs starting with Lab 2 have extra credit portions
  ▪ These are meant to be fun extensions to the labs
❖ Study Guides Task 1 and 2 can also be awarded extra credit, although this will be uncommon

❖ Extra credit points don't affect your lab/guide grades
  ▪ From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
  ▪ Make sure you finish the rest of the lab before attempting any extra credit
Reading Review

❖ Terminology:
   ▪ Address Computation Instruction (lea)
   ▪ Condition codes: Carry Flag (CF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF)
   ▪ Test (test) and compare (cmp) assembly instructions
   ▪ Jump (j*) and set (set*) families of assembly instructions

❖ Questions from the Reading?
Complete Memory Addressing Modes

❖ General:
- $D(Rb, Ri, S)$  $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \times S + D]$
  - **Rb**: Base register (any register)
  - **Ri**: Index register (any register except $\%rsp$)
  - **S**: Scale factor (1, 2, 4, 8) – why these numbers?
  - **D**: Constant displacement value (a.k.a. immediate)

❖ Special cases (see CSPP Figure 3.3 on p.181)
- $D(Rb, Ri)$  $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]$  ($S=1$)
- $(Rb, Ri, S)$  $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \times S]$  ($D=0$)
- $(Rb, Ri)$  $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]$  ($S=1, D=0$)
- $(, Ri, S)$  $\text{Mem}[\text{Reg}[Ri] \times S]$  ($Rb=0, D=0$)
Review Questions

❖ Which of the following x86-64 instructions correctly calculates %rax=9*%rdi?

A. leaq (,%rdi,9), %rax
B. movq (,%rdi,9), %rax
C. leaq (%rdi,%rdi,8), %rax
D. movq (%rdi,%rdi,8), %rax

❖ If %rsi is 0xB0BACAFE 1EE7 F0 0D, what is its value after executing movswl %si, %esi?
Address Computation Instruction

❖ leaq src, dst
  ▪ "lea" stands for load effective address
  ▪ src is address expression (any of the formats we’ve seen)
  ▪ dst is a register
  ▪ Sets dst to the address computed by the src expression (does not go to memory! – it just does math)
  ▪ Example: leaq (%rdx,%rcx,4), %rax

❖ Uses:
  ▪ Computing addresses without a memory reference
    • e.g., translation of p = &x[i];
  ▪ Computing arithmetic expressions of the form x+k*i+d
    • Though k can only be 1, 2, 4, or 8
Example: lea vs. mov

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>0x4</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>0x10</td>
<td>0x108</td>
</tr>
<tr>
<td>0x1</td>
<td>0x100</td>
</tr>
</tbody>
</table>

```assembly
leaq (%rdx, %rcx, 4), %rax
movq (%rdx, %rcx, 4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```
Arithmetic Example

```c
long arith(long x, long y, long z) {
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

### Interesting Instructions
- **leaq**: “address”
- **addq**: computation
- **salq**: shift
- **imulq**: multiplication
  - Only used once!

<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>
</tbody>
</table>
# Arithmetic Example

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>x</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>y</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>z, t4</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>t5</td>
</tr>
</tbody>
</table>

**arith:**

```assembly
leaq   (%rdi,%rsi), %rax  # rax/t1   = x + y
addq  %rdx, %rax          # rax/t2   = t1 + z
leaq  (%rsi,%rsi,2), %rdx # rdx      = 3* y
salq  $4, %rdx            # rdx/t4   = (3* y)*16
leaq  4(%rdi,%rdx), %rcx # rcx/t5   = x + t4 + 4
imulq %rcx, %rax          # rax/rval = t5*t2
ret
```
Control Flow

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

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>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

max:

```asm
???
movq %rdi, %rax
???
???
movq %rsi, %rax
???
ret
```
Control Flow

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

<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>

- **Conditional jump**: if \( x \leq y \) then jump to else
  - `movq %rdi, %rax`
  - `jump to done`

- **Unconditional jump**: else
  - `movq %rsi, %rax`
  - `done:
    - `ret`
Conditionals and Control Flow

- Conditional branch/jump
  - Jump to somewhere else if some condition is true, otherwise execute next instruction

- Unconditional branch/jump
  - Always jump when you get to this instruction

- Together, they can implement most control flow constructs in high-level languages:
  - if (condition) then {...} else {...}
  - while (condition) {...}
  - do {...} while (condition)
  - for (initialization; condition; iterative) {...}
  - switch {...}
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:

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

- Program Counter (instruction pointer)
- Condition Codes

Current top of the Stack
Condition Codes (**Implicit Setting**)

- *Implicitly* set by *arithmetic* operations
  - (think of it as side effects)
  - **Example**: `addq src, dst ← 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:
    
    \[
    (s>0 \: \&\: \& \: d>0 \: \&\: \& \: r<0) \: | \: | \: (s<0 \: \&\: \& \: d<0 \: \&\: \& \: r>=0)
    \]
  
    - *Not set by lea instruction (beware!)*

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
<th>ZF</th>
<th>Zero Flag</th>
<th>SF</th>
<th>Sign Flag</th>
<th>OF</th>
<th>Overflow Flag</th>
</tr>
</thead>
</table>


Condition Codes (Explicit Setting: Compare)

- **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
  - $(a>0 \&\& b<0 \&\& (b-a)>0) \text{ or } (a<0 \&\& b>0 \&\& (b-a)<0)$

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>Zero Flag</th>
<th>Sign Flag</th>
<th>Overflow Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>ZF</td>
<td>SF</td>
<td>OF</td>
</tr>
</tbody>
</table>

### Condition Codes

- **CF (Carry Flag)**: Set if carry out from the most significant bit (good for unsigned comparison).
- **ZF (Zero Flag)**: Set if the result is zero.
- **SF (Sign Flag)**: Set if the result is negative.
- **OF (Overflow Flag)**: Set if there is a signed overflow.

**Example Usage**

- `cmpq src1, src2`: Compares `src1` and `src2` and sets the condition codes.
- `cmpq a, b`: Compares `a` and `b`, setting flags without storing the result.

### Flags in Assembly

- **CF**: Carry flag
- **ZF**: Zero flag
- **SF**: Sign flag
- **OF**: Overflow flag
Condition Codes (Explicit Setting: Test)

- *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)
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

❖ 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

- **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>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>setne dst</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>sets dst</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns dst</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>setg dst</code></td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>setge dst</code></td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>setl dst</code></td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>setle dst</code></td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td><code>seta dst</code></td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><code>setb dst</code></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><code>%rdi</code></td>
<td>1st argument (x)</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>return value</td>
</tr>
</tbody>
</table>

```
cmpq %rsi, %rdi      #
setg %al            #
movzbl %al, %eax    #
ret                  #
```
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;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>1st argument (x)</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>return value</td>
</tr>
</tbody>
</table>

```assembly
cmpq   %rsi, %rdi   # Compare x:y
setg   %al          # Set when >
movzbl %al, %eax   # Zero rest of %rax
ret
```
Aside: \texttt{movz} and \texttt{movs}

\texttt{movz} \_\_ \texttt{src}, \texttt{regDest} \quad \# \textit{Move with zero extension}
\texttt{movs} \_\_ \texttt{src}, \texttt{regDest} \quad \# \textit{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 \texttt{zero} (\texttt{movz}) or \texttt{sign bit} (\texttt{movs})

\texttt{movzSD} / \texttt{movsSD}:  
\textit{S} – size of source (\texttt{b} = 1 byte, \texttt{w} = 2)
\textit{D} – size of dest (\texttt{w} = 2 bytes, \texttt{l} = 4, \texttt{q} = 8)

Example:
\texttt{movz}bq \%al, \%rbx  
\begin{tabular}{c}
0x?? & 0x?? & 0x?? & 0x?? & 0x?? & 0x?? & 0x?? & 0xFF \hline
0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0xFF
\end{tabular} \hline
\%rax \quad  \%rbx
Aside: movz and movs

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

- 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 \textbf{zero} (\texttt{movz}) or \textbf{sign bit} (\texttt{movs})

\[
\text{movz}SD \ / \ \text{movs}SD:
\]
\[
S \quad \text{– size of source (} b = 1 \ \text{byte, } w = 2) \\
D \quad \text{– size of dest (} w = 2 \ \text{bytes, } l = 4, \ q = 8)
\]

Example:
\[
\text{movsbl (} \%rax\text{), } \%ebx
\]

\text{Copy 1 byte from memory into 8-byte register \& sign extend it}
Summary

❖ Control flow in x86 determined by status of 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