x86-64 Programming II
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

http://xkcd.com/409/
Administrivia

- Lab 1a due tonight!
  - Pay attention to Gradescope’s feedback!
- Lab 2 (x86-64) coming soon
  - Learn to read x86-64 assembly and use GDB
- Submissions that fail the autograder get a **ZERO**
  - No excuses – make full use of tools & Gradescope’s interface
- Midterm is in two weeks (2/10 during lecture)
  - You will be provided a fresh reference sheet
    - Study and use this NOW so you are comfortable with it when the exam comes around
  - Form study groups and look at past exams!
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: \texttt{lea} vs. \texttt{mov}

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td>%rbx</td>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x10</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x1</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\texttt{leaq} & (\%rdx,\%rcx,4), \%rax \\
\texttt{movq} & (\%rdx,\%rcx,4), \%rbx \\
\texttt{leaq} & (\%rdx), \%rdi \\
\texttt{movq} & (\%rdx), \%rsi
\end{align*}
lea – “It just does math”
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” 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>%rdi</td>
<td>x</td>
</tr>
<tr>
<td>%rsi</td>
<td>y</td>
</tr>
<tr>
<td>%rdx</td>
<td>z, t4</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

**arith:**

```
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
```
Polling Question

- Which of the following x86-64 instructions correctly calculates $\%rax = 9 \times \%rdi$?

- Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. `leaq (,%rdi,9), %rax`
B. `movq (,%rdi,9), %rax`
C. `leaq (%rdi,%rdi,8), %rax`
D. `movq (%rdi,%rdi,8), %rax`
E. We’re lost...
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>

```
max:
    ???
    movq  %rdi, %rax
    ???
    ???
    movq  %rsi, %rax
    ???
    ret
```
Control Flow

```
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>1&lt;sup&gt;st&lt;/sup&gt; argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; 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

Unconditional jump: jump to done

max:
```
    movq %rdi, %rax
    jmp done
```

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

- Program Counter (instruction pointer)
  - Current top of the Stack

- Condition Codes
  - CF
  - ZF
  - SF
  - OF
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!)**
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
  
  \[
  (a>0 \ \&\& \ b<0 \ \&\& \ (b-a)>0) \ \| \ \\
  (a<0 \ \&\& \ b>0 \ \&\& \ (b-a)<0)
  \]
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)

<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>
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>
## Reminder: x86-64 Integer Registers

- **Accessing the low-order byte:**

  | %rax | %al |
  | %rbx | %bl |
  | %rcx | %cl |
  | %rdx | %dl |
  | %rsi | %sil |
  | %rdi | %dil |
  | %rsp | %spl |
  | %rbp | %bpl |
  | %r8  | %r8b |
  | %r9  | %r9b |
  | %r10 | %r10b|
  | %r11 | %r11b|
  | %r12 | %r12b|
  | %r13 | %r13b|
  | %r14 | %r14b|
  | %r15 | %r15b|
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`, `dl`) 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>%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>

```asm
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, dl) 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;
}
```

### 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>
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)

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:
movzbq %al, %rbx

\[
\begin{array}{c}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \leftarrow %rax \\
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF \leftarrow %rbx
\end{array}
\]
Aside: movz and movs

\[
\begin{align*}
\text{movz} & \_\_ \ src, \ regDest & \# \ Move \ with \ \text{zero} \ extension \\
\text{movs} & \_\_ \ src, \ regDest & \# \ Move \ with \ \text{sign} \ extension \\
\end{align*}
\]

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

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

\textbf{Example:}

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

Copy 1 byte from memory into 8-byte register & sign extend it

\texttt{0x00 0x00 0x7F 0xFF 0xC6 0x1F 0xA4 0xE8} \leftarrow \%rax

\texttt{0x00 0x00 0xFF 0xFF 0x80} \leftarrow \%rbx

\[
\begin{align*}
\ldots & 0x?? 0x?? 0x80 0x?? 0x?? 0x?? \ldots \leftarrow \text{MEM}
\end{align*}
\]

\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.
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