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:

\[ D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \times S + D] \]

- **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)

\[ D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1) \]
\[ (Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \times S] \quad (D=0) \]
\[ (Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0) \]
\[ (, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] \times S] \quad (Rb=0, D=0) \]
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 \( \text{rax} = \text{rdx} + (\text{rcx} \times 4) \)

- 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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x110</td>
<td>0x400</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x8</td>
<td>0xF</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x4</td>
<td>0x118</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x108</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x1</td>
<td>0x100</td>
</tr>
</tbody>
</table>

8 bytes

leaq (%rdx, %rcx, 4), %rax
movq (%rdx, %rcx, 4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi

0x100 + (0x4 * 4)

0x100
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; // mult.
    long t5 = t3 + t4;
    long rval = t2 * t5; // mult.
    return rval;
}
```

### Interesting Instructions
- **leaq**: "address" computation
- **salq**: shift
- **imulq**: multiplication
  - Only used once!
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;
}
```

<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>

Arithmetic Example:

```
arith:
leaq (%rdi,%rsi), %rax        # rax/t1   = x + y
addq %rdx, %rax               # rax/t2   = t1 + z = x+y+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
```
Review Questions

❖ Which of the following x86-64 instructions correctly calculates \( \%\text{rax} = 9 \times \%\text{rdi} \)?

A. `leaql (,\%rdi,9), \%rax`

B. `movq (,\%rdi,9), \%rax`

C. `leaql (\%rdi,\%rdi,8), \%rax`

D. `movq (\%rdi,\%rdi,8), \%rax`

❖ If \( \%\text{rsi} \) is 0x B0BACAFE 1EE7 F0 0D, what is its value after executing `movswl \%si, \%esi`?

- `\%\text{si}` is sign-extended.
- `2 \times \%\text{rsi}` is calculated as: `2 \times 0x B0BACAFE 1EE7 F0 0D`
- The low 32-bits of `\%\text{rsi}` are set to zero.

0x FFFFFFFF 0D
Control Flow

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

```
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;
}

max:
    if x <= y then jump to else
    movq %rdi, %rax
    jump to done
else:
    movq %rsi, %rax
done:
    ret

<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>
Conditionals and Control Flow

- Conditional branch/\textit{jump}
  - Jump to somewhere else if some \textit{condition} is true, otherwise execute next instruction
- Unconditional branch/\textit{jump}
  - \textit{Always} jump when you get to this instruction

Together, they can implement most control flow constructs in high-level languages:

- \textbf{if} (\textit{condition}) \textbf{then} {...} \textbf{else} {...}
- \textbf{while} (\textit{condition}) {...}
- \textbf{do} {...} \textbf{while} (\textit{condition})
- \textbf{for} (\textit{initialization}; \textit{condition}; \textit{iterative}) {...}
- \textbf{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:
  - %rip
  - CF
  - ZF
  - SF
  - OF

Registers:
- %rax
- %rbx
- %rcx
- %rdx
- %r8
- %r9
- %r10
- %r11
- %r12
- %r13
- %r14
- %r15
- %rsi
- %rdi
- %rspb
- %rbp

Program Counter (instruction pointer)
- %rip

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
  - \((s>0 \land d>0 \land r<0) \lor (s<0 \land d<0 \land r\geq 0)\)

- **Not set by lea instruction (beware!)**
Condition Codes (Explicit Setting: Compare)

- **Explicitly set by Compare instruction**
  - `cmpq  src1, src2  → sub`
  - `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
      \[(a>0 \&\& b<0 \&\& (b-a)>0) || (a<0 \&\& b>0 \&\& (b-a)<0)\]

<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 Setting: Test)

- *Explicitly* set by **Test** instruction
  - `testq src2, src1` ➜ `and` 
  - `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**) ➜ `CF=0` `OF=0`
  - **ZF=1** if `a & b == 0`
  - **SF=1** if `a & b < 0` (signed)

<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>
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`?

\[
\begin{align*}
\text{\%al} &= 0x80 \\
\text{\%bl} &= 0x81 \\
\text{cmp} : \%bl - \%al &= \%bl + \neg \%al + 1 \\
&= 0x81 + 0x80 = 0x2f + 1 \\
&= 0x81 \rightarrow CF = 1
\end{align*}
\]

\[
\begin{align*}
ZF &= \text{bl} - \text{al} \neq \phi \\
ZF &= \phi \\
SF &= 0x81 - 0x80 \\
&= 0x1 \\
SF &= 0 \\
OF &= 0 \\
OF : 0x01 &= 0b00000001 \\
\%bl + (-\%al) &= 0 < 0 > 0 \text{ signs diff.}
\end{align*}
\]
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><code>jmp target</code></td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td><code>je target</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>jne target</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>js target</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td><code>jns target</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>jg target</code></td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>jge target</code></td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>jl target</code></td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>jle target</code></td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td><code>ja target</code></td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td><code>jb target</code></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>sete $dst$</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne $dst$</td>
<td>$\neg ZF$</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>$\neg SF$</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg $dst$</td>
<td>$(SF \land \neg OF) \land \neg ZF$</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge $dst$</td>
<td>$(SF \land \neg OF)$</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl $dst$</td>
<td>$(SF \land \neg OF)$</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>settle $dst$</td>
<td>$(SF \land \neg OF) \lor ZF$</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta $dst$</td>
<td>$\neg CF \land \neg ZF$</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;
}
```

<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>
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>

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

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

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

\texttt{movzSD / 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{movzbq} %al, %rbx

\[
\begin{array}{c}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \leftarrow \%rax \\
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF \leftarrow \%rbx
\end{array}
\]
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**<sub>SD</sub> / **movs**<sub>SD</sub>:
- **S** – size of source (<b>1</b> byte, <b>2</b>)
- **D** – size of dest (<b>2</b> bytes, <b>4</b>, <b>8</b>)

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.
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