x86-64 Programming II
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/99/
Relevant Course Information

- Lab submissions that fail the autograder get a ZERO
  - No excuses – make full use of tools & Gradescope’s interface
  - Leeway on Lab 1a won’t be given moving forward

- Lab 2 (x86-64) released today
  - Learn to trace x86-64 assembly and use GDB

- Midterm is in two weeks (take home, 11/3–5)
  - Open book; make notes and use midterm reference sheet
  - Individual, but discussion allowed via “Gilligan’s Island Rule”
  - Mix of “traditional” and design/reflection questions
    - Form study groups and look at past exams!
Extra Credit

❖ All labs starting with Lab 2 have extra credit portions
  ▪ These are meant to be fun extensions to the labs

❖ Extra credit points don't affect your lab 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?
Memory Addressing Modes (Review)

❖ General:

- **D(Rb,Ri,S)**  \( \text{Mem[Reg[Rb] + Reg[Ri] * S + D]} \)
  - **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)**  \( \text{Mem[Reg[Rb] + Reg[Ri] + D]} \)  \((S=1)\)
- **(Rb,Ri,S)**  \( \text{Mem[Reg[Rb] + Reg[Ri] * S]} \)  \((D=0)\)
- **(Rb,Ri)**  \( \text{Mem[Reg[Rb] + Reg[Ri]} \)  \((S=1, D=0)\)
- **(,Ri,S)**  \( \text{Mem[Reg[Ri] * S]} \)  \((Rb=0, D=0)\)

\[ \text{ar[i]} \leftrightarrow *(ar+i) \rightarrow \text{Mem[ar+i*size of (data type)]} \]

\[ \text{so reg name not interpreted as Rb} \]
Address Computation Instruction (Review)

❖ **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 \( x + k \times i + d \)
  - Though \( k \) can only be 1, 2, 4, or 8
Review Questions

❖ If `%rdx = 0xf000` and `%rcx = 0x100`, what addresses are dereferenced by the following memory operands?

- `(,%rdx,%rcx)`
- `0x80 (,%rdx,2)`

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

A. `leaq (,%rdi,8), %rax`

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

C. `leaq (%rdi,%rdi,8), %rax`

D. `movq (%rdi,%rdi,8), %rax`
Example: Basic Arithmetic

```c
long simple_arith(long x, long y) {
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

```
Register | Use(s)
---------|---------
%rdi     | 1\textsuperscript{st} argument (x)
%rsi     | 2\textsuperscript{nd} argument (y)
%rax     | return value
```

### Example Instruction Set

- `addq  %rdi, %rsi`
- `imulq  $3, %rsi`
- `movq   %rsi, %rax`
- `ret    # return`

### Register Use

- `%rdi`: 1\textsuperscript{st} argument (x)
- `%rsi`: 2\textsuperscript{nd} argument (y)
- `%rax`: return value

### Notes

- Arbitrary! (for now...)
Example: Using Memory

```c
void swap(long* xp, long* yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Compiler Explorer: https://godbolt.org/z/c9M9fMefa
Example: Using Memory

### Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### Word Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

swap:

1. `movq (%rdi), %rax` # t0 = *xp
2. `movq (%rsi), %rdx` # t1 = *yp
3. `movq %rdx, (%rdi)` # *xp = t1
4. `movq %rax, (%rsi)` # *yp = t0

`ret`
**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>0x100</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
<td>0x110</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x1</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Example instructions:

- `leaq (%rdx, %rcx, 4), %rax` → 0x110 (addr)
- `movq (%rdx, %rcx, 4), %rbx` → 0x8 (data)
- `leaq (%rdx), %rdi` → 0x100 (addr)
- `movq (%rdx), %rsi` → 0x1 (data)

Note: `%rax` and `%rsi` are used as destination registers in the example.
Example: lea Arithmetic

```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
- **lea**: “address” computation
- **salq**: shift
- **imulq**: multiplication

- Only used once!
Example: lea Arithmetic

```
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>

limited registers means they often get reused!

```
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
```

Comment (AT&T syntax)
## 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>

### x86-64 Assembly Code

```
max:
    ???
    movq %rdi, %rax # if case
    ???
    ???
    movq %rsi, %rax # else case
    ???
    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>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 \( \text{done} \)
- if \( x > y \) jump to \( \text{done} \)

**Unconditional jump**
- 

---

max: if TRUE
- movq %rdi, %rax
- jump to done
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 {...}
Summary

❖ **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations

❖ **Load effective address** (`lea`) instruction used to compute addresses and perform basic arithmetic
  - *Doesn’t* dereference the source memory operand, unlike all other instructions!

❖ Control flow in x86 determined by Condition Codes