x86-64 Programming II
CSE 351 Autumn 2021

Instructor:
Justin Hsia

Teaching Assistants:
Allie Pfleger
Anirudh Kumar
Assaf Vayner
Atharva Deodhar
Celeste Zeng
Dominick Ta
Francesca Wang
Hamsa Shankar
Isabella Nguyen
Joy Dang
Julia Wang
Maggie Jiang
Monty Nitschke
Morel Fotsing
Sanjana Chintalapati

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, 11/3–11/5)
  ▪ 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
Example of Basic Addressing Modes

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

Compiler Explorer: [https://godbolt.org/z/zc4Pcq](https://godbolt.org/z/zc4Pcq)
Understanding `swap()`

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

**Registers**

- `%rdi`
- `%rsi`
- `%rax`
- `%rdx`

**Memory**

- `xp` is stored at `%rdi`
- `yp` is stored at `%rsi`
- `t0` is stored at `%rax`
- `t1` is stored at `%rdx`

**Swap**

- `movq (%rdi), %rax`
- `movq (%rsi), %rdx`
- `movq %rdx, (%rdi)`
- `movq %rax, (%rsi)`
- `ret`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</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></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

### Word Address

- 0x120
- 0x118
- 0x110
- 0x108
- 0x100

### 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`
Complete Memory Addressing Modes

❖ General:

- \( D(R_b, R_i, S) \) \[ Mem[Reg[R_b]+Reg[R_i]*S+D] \]
  - \( R_b \): Base register (any register)
  - \( R_i \): 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)

- \( D(R_b, R_i) \) \[ Mem[Reg[R_b]+Reg[R_i]+D] \] (\( S=1 \))
- \( (R_b, R_i, S) \) \[ Mem[Reg[R_b]+Reg[R_i]*S] \] (\( D=0 \))
- \( (R_b, R_i) \) \[ Mem[Reg[R_b]+Reg[R_i]] \] (\( S=1, D=0 \))
- \( (\hat{R}_i, S) \) \[ Mem[Reg[R_i]*S] \] (\( R_b=0, D=0 \))

\( \hat{R}_b \) so reg name not interpreted as \( R_b \)
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address (8 bytes wide)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x8(%rdx))</td>
<td>(\text{Reg}[%rb]+D = 0xf000 + 0x8)</td>
<td>(0xf008)</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>(\text{Reg}[%rb]+\text{Reg}[%ri]*4)</td>
<td>(0xf100)</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>(\text{Reg}[%ri]*4)</td>
<td>(0xf400)</td>
</tr>
<tr>
<td>(0x80(%rdx,2))</td>
<td>(\text{Reg}[%ri]*2 + 0x80)</td>
<td>(0x1e080)</td>
</tr>
</tbody>
</table>

\(\%rdx\) \(0xf000\)
\(\%rcx\) \(0x0100\)

\[D(\%rb,\%ri, S) \rightarrow \text{Mem}[\text{Reg}[\%rb]+\text{Reg}[\%ri]*S+D]\]

Ignore the memory access for now.

- **Default Values:**
  - \(S = 1\)
  - \(D = 0\)
  - \(\text{Reg}[\%rb] = 0\)
  - \(\text{Reg}[\%ri] = 0\)
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?
Review Questions

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

A. `leaq (,%rdi,8), %rax` invalid syntax
B. `movq (,%rdi,9), %rax` invalid syntax
C. `leaq (%rdi,%rdi,8), %rax` %rax = 9*%rdi
D. `movq (%rdi,%rdi,8), %rax` %rax = Mem[9*%rdi]

❖ If %rsi is 0xB0BACAFAE 1EE7 F0 0D, what is its value after executing `movswl %si, %esi`?

%esi MSB of %esi is a 1

0x 0000 0000 x86-64 rule when destination is 32 bits
FFFF FOOD sign extension
original data source is 2 bytes
destination is 4 bytes

%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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax 0x110</td>
<td>0x400</td>
<td>0x120</td>
</tr>
<tr>
<td>%rbx 0x8</td>
<td>0xF</td>
<td>0x118</td>
</tr>
<tr>
<td>%rcx 0x4</td>
<td>0x8</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx 0x100</td>
<td>0x10</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdi 0x100</td>
<td>0x1</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi 0x1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x100 + 0x4 * 4 = 0x110

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

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;  // replaced by lea & shift
    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>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>

---

**Interesting Instructions**

- **leaq:** “address”
- **addq:** 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;
}
```

```
Register | Use(s)
--- | ---
%rdi    | x
%rsi    | y
%rdx    | z, t4
%rax    | t1, t2, rval
%rcx    | t5

limited registers means they often get reused!
```

```asm
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
```

Comment (AT&T syntax)

$S \in \{1,2,4,8\}$

$S \in \{1,2,4,8\}$
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 # 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\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>

**Conditional jump**
- \textit{if} $x \leq y$ \textit{then jump to} else
- \texttt{movq} %rdi, %rax
- \texttt{jump to} done

**Unconditional jump**
- \texttt{else:}
- \texttt{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

❖ **Control flow in x86 determined by Condition Codes**