x86-64 Programming I

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

Homework 2 due Wednesday (1/24)
- x86-64 part extended until Monday, January 29
- Lab 2 (x86-64) released today (1/22)
  - Due on Friday, February 2 @ 11:59 pm
- Don’t forget to read the book!
  - Ch. 3 will be very helpful!
Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - Maximum of one memory operand
  - Beware argument order!
  - No distinction between signed and unsigned
  - Only arithmetic vs. logical shifts
  - How do you implement “r3 = r1 + r2”?

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>dst = dst + src</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>dst = dst - src</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>dst = dst * src</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>dst = dst &gt;&gt;&gt; src</td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>dst = dst ^ src</td>
</tr>
<tr>
<td>andq src, dst</td>
<td>dst = dst &amp; src</td>
</tr>
<tr>
<td>orq src, dst</td>
<td>dst = dst</td>
</tr>
</tbody>
</table>

  - Arithmetic Example

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

  - Example of Basic Addressing Modes

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

  - Understanding `swap()`

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

  - Understanding `swap()`

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x110</td>
<td>0x110</td>
</tr>
<tr>
<td>%rax</td>
<td>0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x100</td>
<td>0x100</td>
</tr>
</tbody>
</table>

swap:

```
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```

Memory Addressing Modes: Basic

- **Indirect**: (R)  
  Mem[Reg[R]]
  - Data in register R specifies the memory address
  - Like pointer dereference in C
  - Example: movq (%rcx), %rax

- **Displacement**: D(R)  
  Mem[Reg[R]+D]
  - Data in register R specifies the start of some memory region
  - Constant displacement D specifies the offset from that address
  - Example: movq 8(%rbp), %rdx

Complete Memory Addressing Modes

- **General**:  
  D(Rb,Ri,S)  
  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)  
    Mem[Reg[Rb]+Reg[Ri]*S]
    (S=1)
  - (Rb,Ri)  
    Mem[Reg[Rb]+Reg[Ri]*S]
    (D=0)
  - (Rb,Ri)  
    Mem[Reg[Rb]+Reg[Ri]]
    (S=1, D=0)
  - (Rb,Ri)  
    Mem[Reg[Rb]*S]
    (Rb=0, D=0)
**Address Computation Examples**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>D(Rb,Ri,S) → Mem[Reg[8b]+Reg[Ri]*S+D]</td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>(rdx,rcx)</td>
<td></td>
</tr>
<tr>
<td>(rdx,%rcx,4)</td>
<td>0x80,(rdx,2)</td>
<td></td>
</tr>
</tbody>
</table>

**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 \( p = 4\times[1]; \)
  - 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 Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x120</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x118</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

**Arithmetic Example**

- long arith:
  - \( t1 = x + y; \)
  - \( t2 = x + t1; \)
  - \( t3 = x + t4; \)
  - \( t4 = y + d8; \)
  - \( t5 = t3 + t4; \)
  - \( tval = t2 * t5; \)
  - return \( tval; \)

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

**Peer Instruction Question**

- 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
  - E. We’re lost...
Summary

❖ **Operands**: Assembly operands include *immediates*, *registers*, and *data at a specified memory location*.

❖ **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

❖ *lea* is address calculation instruction
  - Does NOT actually go to memory
  - Used to compute addresses or some arithmetic expressions