x86 Programming I
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/409/
Administrivia

- Lab 1 due today at 5pm
  - You have *late days* available
- Lab 2 (x86 assembly) released next Tuesday (10/18)
- Homework 1 due next Friday (10/21)
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:
get_mpg:
pushq   %rbp
movq    %rsp, %rbp...
popq    %rbp
ret

Machine code:
0111010000011000
1000110100000100000000101000100111000010110000011111101000011111

Computer system:

OS:
Windows 8
Mac

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
x86 Topics for Today

- Registers
- Move instructions and operands
- Arithmetic operations
- Memory addressing modes
- swap example
What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with `%` (e.g., `%rsi`)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86
### x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>RAX</th>
<th>EAX</th>
<th>R8</th>
<th>R8D</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX</td>
<td>EBX</td>
<td>R9</td>
<td>R9D</td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
<td>R10</td>
<td>R10D</td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
<td>R11</td>
<td>R11D</td>
</tr>
<tr>
<td>RSI</td>
<td>ESI</td>
<td>R12</td>
<td>R12D</td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
<td>R13</td>
<td>R13D</td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
<td>R14</td>
<td>R14D</td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
<td>R15</td>
<td>R15D</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)
Some History: IA32 Registers – 32 bits wide

- **%eax** (%ax %ah %al): accumulate
- **%ecx** (%cx %ch %cl): counter
- **%edx** (%dx %dh %dl): data
- **%ebx** (%bx %bh %bl): base
- **%esi** (%si): source index
- **%edi** (%di): destination index
- **%esp** (%sp): stack pointer
- **%ebp** (%bp): base pointer

**General Purpose:**

16-bit virtual registers (backwards compatibility)

Name Origin (mostly obsolete)
x86-64 Assembly Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. %xmm1, %ymm2)
  - Come from extensions to x86 (SSE, AVX, ...)
  - Probably won’t have time to get into these 😞
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, ...
  - “Intel”: used by Intel documentation, Intel tools, ...
  - Must know which you’re reading
Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - **Load** data from memory into register
     - \( %\text{reg} = \text{Mem}[\text{address}] \)
   - **Store** register data into memory
     - \( \text{Mem}[\text{address}] = %\text{reg} \)

2) Perform arithmetic operation on register or memory data
   - \( c = a + b; \quad z = x \ll y; \quad i = h \& g; \)

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

Remember: Memory is indexed just like an array of bytes!
Operand types

- **Immediate**: Constant integer data
  - Examples: $0x400$, $-533$
  - Like C literal, but prefixed with `'$'
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other “address modes”
Moving Data

- **General form:** `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte “byte”

- `movw src, dst`
  - Move 2-byte “word”

- `movl src, dst`
  - Move 4-byte “long word”

- `movq src, dst`
  - Move 8-byte “quad word”
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

- **Cannot do memory-memory transfer with a single instruction**
  - **How would you do it?**
Memory vs. Registers

- **Addresses vs. Names**
  - Example: 0x7FFFD024C3DC vs. %rdi

- **Big vs. Small**
  - ~8 GiB vs. (16 x 8 B) = 128 B

- **Slow vs. Fast**
  - ~50-100 ns vs. sub-nanosecond timescale

- **Dynamic vs. Static**
  - Can “grow” as needed while program runs vs. fixed number in hardware
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; src</td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>dst = dst &lt;&lt; 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
Logical
(same as salq)

(unsigned mult)

Maximum of one memory operand

Operand size specifier
Some Arithmetic Operations

- **Unary (one-operand) Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incq dst</code></td>
<td><code>dst = dst + 1</code></td>
<td>increment</td>
</tr>
<tr>
<td><code>decq dst</code></td>
<td><code>dst = dst - 1</code></td>
<td>decrement</td>
</tr>
<tr>
<td><code>negq dst</code></td>
<td><code>dst = -dst</code></td>
<td>negate</td>
</tr>
<tr>
<td><code>notq dst</code></td>
<td><code>dst = ~dst</code></td>
<td>bitwise complement</td>
</tr>
</tbody>
</table>

- See CSPP Section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`
Arithmetic Example

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

<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>

```
y += x;
y *= 3;
long r = y;
return r;
```
Example of Basic Addressing Modes

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

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

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

**Registers**
- `%rdi` ↔ `xp`
- `%rsi` ↔ `yp`
- `%rax` ↔ `t0`
- `%rdx` ↔ `t1`

**Memory**

**Register** | **Variable**
--- | ---
%rdi | xp
%rsi | yp
%rax | t0
%rdx | t1
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></td>
</tr>
<tr>
<td>%rdx</td>
<td></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>456</td>
</tr>
</tbody>
</table>

### Code

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>456</td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td>0x108</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
```
Understanding `swap()`

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi 0x120</td>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax 123</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx 456</td>
<td>456</td>
<td>0x108</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi 0x120</td>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax 123</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx 456</td>
<td></td>
<td>0x108</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
```
# Understanding `swap()`

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi 0x120</td>
<td>456</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax 123</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx 456</td>
<td>123</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

```assembly
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:** \((R)\) \(\text{Mem}[\text{Reg}[R]]\)
  - Data in register \(R\) specifies the memory address
  - Like pointer dereference in C
  - **Example:** \text{movq} (\%rcx), \%rax

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

- **General:**
  - \(D(Rb, Ri, S)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]*S+D]\)
    - \(Rb\): 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)
  - \(D(Rb, Ri)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]\) \((S=1)\)
  - \((Rb, Ri, S)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]*S]\) \((D=0)\)
  - \((Rb, Ri)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]\) \((S=1, D=0)\)
  - \((\, Ri, S)\) \(\text{Mem}[\text{Reg}[Ri]*S]\) \((Rb=0, D=0)\)
## 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></td>
<td></td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80 (,%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
D(R_b, R_i, S) \rightarrow \text{Mem}[(\text{Reg}[R_b] + \text{Reg}[R_i] \times S + D)]
\]
Peer Instruction Question

Which of the following statements is TRUE?

- Vote at http://PollEv.com/justinh

(A) The program counter (%rip) is a register that we manually manipulate

(B) There is only one way to compile a C program into assembly

(C) Mem to Mem (src to dst) is the only disallowed operand combination

(D) We can compute an address without using any registers
Summary

- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly instructions have rigid form
  - Operands include immediates, registers, and data at specified memory locations
  - Many instruction variants based on size of data
- **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