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
100011010000010000000010
1000100111000010
110000011111101000001111

Computer system:

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>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<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

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
   - $c = a + b; \quad z = x << y; \quad i = h \& g;$

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`, `%rdx`
  - 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

- \texttt{movb src, dst} | Move 1-byte “byte” 8 bits
- \texttt{movw src, dst} | Move 2-byte “word” 16 bits
- \texttt{movl src, dst} | Move 4-byte “long word” 32 bits
- \texttt{movq src, dst} | Move 8-byte “quad word” 64 bits
### 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>Imm</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>Reg</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>*p_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?
    - 1. movq (%rax), %rdx
    - 2. movq %rdx, (%rbx)
### Memory vs. Registers

<table>
<thead>
<tr>
<th>Memory</th>
<th>vs.</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses</td>
<td>vs.</td>
<td>Names</td>
</tr>
<tr>
<td>0x7FFFD024C3DC</td>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>Big</td>
<td>vs.</td>
<td>Small</td>
</tr>
<tr>
<td>~ 8 GiB</td>
<td>(16 x 8 B) = 128 B</td>
<td></td>
</tr>
<tr>
<td>Slow</td>
<td>vs.</td>
<td>Fast</td>
</tr>
<tr>
<td>~50-100 ns</td>
<td>sub-nanosecond timescale</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>vs.</td>
<td>Static</td>
</tr>
<tr>
<td>Can “grow” as needed while program runs</td>
<td>fixed number in hardware</td>
<td></td>
</tr>
</tbody>
</table>
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</td>
<td>dst = dst + src</td>
</tr>
<tr>
<td>subq</td>
<td>dst = dst - src</td>
</tr>
<tr>
<td>imulq</td>
<td>dst = dst * src</td>
</tr>
<tr>
<td>sarq</td>
<td>dst = dst &gt;&gt; src</td>
</tr>
<tr>
<td>shrq</td>
<td>dst = dst &gt;&gt; src</td>
</tr>
<tr>
<td>shlq</td>
<td>dst = dst &lt;&lt; src</td>
</tr>
<tr>
<td>xorq</td>
<td>dst = dst ^ src</td>
</tr>
<tr>
<td>andq</td>
<td>dst = dst &amp; src</td>
</tr>
<tr>
<td>orq</td>
<td>dst = dst / src</td>
</tr>
</tbody>
</table>

Operand size specifier: (dst += src)
Signed mult: (same as salq)
Arithmetic: Logical
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>$dst = dst + 1$</td>
<td>increment</td>
</tr>
<tr>
<td><code>decq dst</code></td>
<td>$dst = dst - 1$</td>
<td>decrement</td>
</tr>
<tr>
<td><code>negq dst</code></td>
<td>$dst = -dst$</td>
<td>negate</td>
</tr>
<tr>
<td><code>notq dst</code></td>
<td>$dst = \sim dst$</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;  \( y = x + y \)
    long t2 = t1 * 3;  \( y = y \times 3 \)
    return t2;
}
```

### Register Use(s)

<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(r)</td>
</tr>
</tbody>
</table>

By convention:

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

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

```
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` to `%rax`
- `%rsi` to `%rdx`
- `%rax` to `%rdx`

**Memory**
- `xp` to `%rdi`
- `yp` to `%rsi`
- `t0` to `%rax`
- `t1` to `%rdx`

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

### Registers

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

### `swap`

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

Comment in x86:

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

Initial values:
**Understanding \( \text{swap}(\) \)**

### Registers

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

### Memory

<table>
<thead>
<tr>
<th>Word 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()`

### Registers

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

### Memory

<table>
<thead>
<tr>
<th>Word 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>

### `swap`

```
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>0x120</td>
<td>456</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>456</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
<td>456</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
<td>456</td>
</tr>
</tbody>
</table>

swap:

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

### Registers

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

### Memory Content

- Word Address: 456
  - 123

### `swap:`

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

- **Displacement:** \(D(R)\) \(\text{Mem[Reg[R]+D]}\)
  - 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

Pointer Arithmetic: \( ar[i] \leftrightarrow *(ar+i) \leftrightarrow Mem[ar+i \times \text{sizeof}(\cdot)] \)

- **General:**
  - \( D(Rb, Ri, S) \) \( Mem[Reg[Rb]+Reg[Ri] \times S+D] \)
    - \( Rb \): 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)
  - \( D(Rb, Ri) \) \( Mem[Reg[Rb]+Reg[Ri]+D] \) (\( S=1 \))
  - \((Rb, Ri, S)\) \( Mem[Reg[Rb]+Reg[Ri] \times S] \) (\( D=0 \))
  - \((Rb, Ri)\) \( Mem[Reg[Rb]+Reg[Ri]] \) (\( S=1, D=0 \))
  - \((, Ri, S)\) \( Mem[Reg[Ri] \times S] \) (\( Rb=0, D=0 \))
## 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>0xf000 + 0x1 + 0x8</td>
<td></td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0xf000 + 0x0100 * 1 + 0</td>
<td></td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>0xf000 + 0x0100 * 4 + 0</td>
<td></td>
</tr>
<tr>
<td>0x80 (%rdx, 2)</td>
<td>0 + 0xf000*2 + 0x80</td>
<td></td>
</tr>
</tbody>
</table>

If omitted:
- \( D = 0 \)
- \( \text{Reg}[Rb] = 0 \)
- \( \text{Reg}[Ri] = 0 \)
- \( S = 1 \)

\[ \text{Mem}[	ext{Reg}[Rb] + \text{Reg}[Ri] \times S + D] \]
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>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>0xf000 + 0x100*4</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%rdx, 2)</td>
<td>0xf000*2 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Peer Instruction Question

- Which of the following statements is TRUE?
  
  **(A) The program counter (%rip) is a register that we manually manipulate**
  - *not 1 of 16 available, want %rip handled automatically*

  **(B) There is only one way to compile a C program into assembly**
  - *absolutely not!*

  **(C) Mem to Mem (src to dst) is the only disallowed operand combination**
  - *available operand types are Imm, Reg, Mem. can't have Imm as dst.*

  **(D) We can compute an address without using any registers**
  - $D(Rb,Ri,S) \rightarrow$ just omit $Rb$ and $Ri$
  - Example: $\$4()$ accesses address 4
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