Instructor:
Justin Hsia

Teaching Assistants:
Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan

http://www.smbc-comics.com/?id=2999
Adminstrivia

- Lab 1b due tonight at 11:59 pm
  - You have *late day tokens* available
- Homework 2 due next Friday (10/19)
- Lab 2 (x86-64) released on Monday (10/15)
  - Due on 10/26
Non-Compiling Code

- You get a zero on the assignment
  - No excuses – you have access to our grading environment

- Some leeway was given on Lab 1a, do not expect the same leniency moving forward
Writing Assembly Code? In 2018???

- Chances are, you’ll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:
  - Behavior of programs in the presence of bugs
    - When high-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing systems software
    - What are the “states” of processes that the OS must manage
    - Using special units (timers, I/O co-processors, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form
Assembly Programmer’s View

- **Programmer-visible state**
  - **PC**: the Program Counter (\%rip in x86-64)
    - Address of next instruction
  - Named registers
    - Together in “register file”
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)
x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses
- 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, ...)
- 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, ...

Not covered in 351
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* only 16 of them...
### x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>64-bit names</th>
<th>32-bit names</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<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>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<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** : accumulate
- **%ecx** : counter
- **%edx** : data
- **%ebx** : base
- **%esi** : source index
- **%edi** : destination index
- **%esp** : stack pointer
- **%ebp** : base pointer

16-bit virtual registers (backwards compatibility)  
Name Origin (mostly obsolete)
Memory

- Addresses
  - 0x7FFFD024C3DC

- Big
  - ~8 GiB

- Slow
  - ~50-100 ns

- Dynamic
  - Can “grow” as needed while program runs

vs.

Registers

- Names
  - %rdi

- Small
  - (16 x 8 B) = 128 B

- Fast
  - sub-nanosecond timescale

- Static
  - fixed number in hardware
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
Operands types

- **Immediate**: Constant integer data
  - Examples: $0x400$, $-533$ (hexadecimal, decimal)
  - 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”

Take data in %rax, treat as address, pull data at that address
x86-64 Introduction

- Data transfer instruction (\texttt{mov})
- Arithmetic operations
- Memory addressing modes
  - \texttt{swap} example
- Address computation instruction (\texttt{lea})
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>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>Mem</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?
    1. Mem → Reg
    2. Reg → Mem

\[
\begin{align*}
\text{movq} & \hspace{1cm} \text{movq} \\
\%r8, \%rdx & \hspace{1cm} \%r8, \%rdx
\end{align*}
\]
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>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>( dst = dst + src )</td>
<td>imm, reg, or mem</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>( dst = dst - src )</td>
<td>( dst += src )</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>( dst = dst \times src )</td>
<td>signed mult</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>( dst = dst \gg src )</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>( dst = dst \gg src )</td>
<td>Logical</td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>( dst = dst \ll src )</td>
<td>(same as salq)</td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>( dst = dst \oplus src )</td>
<td></td>
</tr>
<tr>
<td>andq src, dst</td>
<td>( dst = dst &amp; src )</td>
<td></td>
</tr>
<tr>
<td>orq src, dst</td>
<td>( dst = dst</td>
<td>src )</td>
</tr>
</tbody>
</table>
Some Arithmetic Operations

- Unary (one-operand) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq</td>
<td>$dst = dst + 1$</td>
<td>increment</td>
</tr>
<tr>
<td>decq</td>
<td>$dst = dst - 1$</td>
<td>decrement</td>
</tr>
<tr>
<td>negq</td>
<td>$dst = -dst$</td>
<td>negate</td>
</tr>
<tr>
<td>notq</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;
    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;`

```
simple_arith:
  addq %rdi, %rsi
  imulq $3, %rsi
  movq %rsi, %rax
  ret # return
```
Example of Basic Addressing Modes

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

```
swap:     src    , dst    (AT&T Syntax)
         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`
- `%rsi`
- `%rax`
- `%rdx`

### Memory

### Register ↔ Variable

- `%rdi` ↔ `xp`
- `%rsi` ↔ `yp`
- `%rax` ↔ `t0`
- `%rdx` ↔ `t1`
Understanding `swap()`

### Registers
- `%rdi`: 0x120
- `%rsi`: 0x100
- `%rax`
- `%rdx`

### Memory
- Word Address:
  - 0x120
  - 0x118
  - 0x110
  - 0x108
  - 0x100
- `123` at 0x120
- `456` at 0x100

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

Comments:
- `# src` (Source)
- `# dst` (Destination)
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 0x120</td>
<td></td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td>123 0x118</td>
<td></td>
</tr>
<tr>
<td>%rax 123</td>
<td>456 0x110</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>456 0x108</td>
<td></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>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()`

### 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>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>

### Code

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

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Registers} & \textbf{Memory} \\
\hline
%rdi & 0x120 \hline
%rsi & 0x100 \hline
%rax & 123 \hline
%rdx & 456 \hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Word} & \textbf{Address} \\
\hline
456 & 0x120 \hline
123 & 0x118 \hline
123 & 0x110 \hline
123 & 0x108 \hline
123 & 0x100 \hline
\end{tabular}
\end{center}

\texttt{swap:}
\begin{verbatim}
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
\end{verbatim}
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:
  - \[ D(Rb, Ri, S) \] \[ \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \cdot S + D] \]
    - \( Rb \): Base register (any register)
    - \( Ri \): Index register (any register except \%rsp)
    - \( S \): Scale factor (1, 2, 4, 8) – why these numbers? data type widths
    - \( D \): 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)
  - \[ (Rb, Ri, S) \] \[ \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \cdot S] \] (D=0)
  - \[ (Rb, Ri) \] \[ \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \] (S=1, D=0)
  - \[ (, Ri, S) \] \[ \text{Mem}[\text{Reg}[Ri] \cdot S] \] (Rb=0, D=0)

\[ \text{ar}[i] \leftrightarrow *(\text{ar} + i) \rightarrow \text{Mem}[\text{ar} + i \cdot \text{size of (data type)}] \]
### Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(Rb, Ri, S)</td>
<td>Mem[Reg[Rb] + Reg[Ri] * S + D]</td>
<td>0xf1e080</td>
</tr>
<tr>
<td>0x8 (%rdx)</td>
<td>Reg[Rb] + D = 0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>Reg[Rb] + Reg[Ri] * 4</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>Reg[Ri] * 2 + 0x80</td>
<td>0xf400</td>
</tr>
<tr>
<td>D(Ri, S)</td>
<td>0xf000 * 2</td>
<td></td>
</tr>
<tr>
<td>0x80 (%rdx, 2)</td>
<td>0xf000 &lt;&lt; 1 = 0x1e000</td>
<td></td>
</tr>
</tbody>
</table>

\[ D = \text{default values:} \quad S = 1 \quad D = 0 \quad \text{Reg}[Rb] = 0 \quad \text{Reg}[Ri] = 0 \]

\[ \text{ignore the memory access for now} \]
Summary

- There are 3 types of operands in x86-64
  - Immediate, Register, Memory

- There are 3 types of instructions in x86-64
  - Data transfer, Arithmetic, Control Flow

- 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