x86-64 Programming I
CSE 351 Spring 2019

Instructor:
Ruth Anderson

Teaching Assistants:
Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

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

- Lab 1b due Monday (4/22)
  - Submit `bits.c` and `lab1Breflect.txt`

- Homework 2 due Wednesday (4/24)
  - On Integers, Floating Point, and x86-64

- Lab 2 (x86-64) coming soon, due Wednesday (5/01)
Non-Compiling Code

- You get a zero on the assignment
  - No excuses – you have access to our grading environment
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
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, ...
  - Must know which you’re reading

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Origin</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah %al</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch %cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh %dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh %bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source_index</td>
<td>destination index</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td>base pointer</td>
</tr>
</tbody>
</table>

- **General Purpose Registers**: %eax, %ecx, %edx, %ebx, %esi, %edi
- **16-bit Virtual Registers**: %ecx, %edx, %ebx
- **32-bit (Same as Last Slide)**: %eax
- **8 bits**: %ah %al, %ch %cl, %dh %dl

**Backwards Compatibility**

- **Name Origin** (mostly obsolete)
- **16-bit virtual registers** (backwards compatibility)
Memory vs. Registers

- **Addresses**
  - vs. **Names**
    - 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**
  - Can “grow” as needed vs. **Static**
    - while program runs 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
     - \( \text{Mem}[\text{address}] = \% \text{reg} \)

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

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
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”

```
%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp (stack pointer)
%rbp
%r8 - %r15
```
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:** \texttt{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”

- \texttt{movw\ src, dst}
  - Move 2-byte “word”

- \texttt{movl\ src, dst}
  - Move 4-byte “long word”

- \texttt{movq\ src, dst}
  - Move 8-byte “quad word”
**movq** Operand Combinations

\[ \text{movq } \text{src, dst} \]

<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>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>*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. Mem → Reg  
       movq (%rax), %rdx  
    2. Reg → Mem  
       movq %rdx, (%rbx)
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>

Other ways to set to 0:
- `subq %rcx, %rcx`
- `andq $0, %rcx`
- `xorq %rax, %rcx`
- `imulq $0, %rcx`

Imm, Reg, or Mem

Operand size specifier (b, w, l, q)
Some Arithmetic Operations

- Unary (one-operand) Instructions:

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

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>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Convention!

```

```asm
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:
    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**

<table>
<thead>
<tr>
<th>Register</th>
<th>Memory 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></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>

**swap:**

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

**Comment**
Understanding `swap()`

### Registers

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

```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>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<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></th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<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{tabular}{|c|c|}
\hline
\textbf{Registers} & \textbf{Memory} \\
\hline
%rdi & 0x120 & 456 \\
%rsi & 0x100 & 456 \\
%rax & 123 \\
%rdx & 456 \\
\hline
\end{tabular} \\

\textbf{Word}
\textbf{Address}
0x120
0x110
0x108
0x100

\texttt{swap:}
\begin{quote}
\texttt{movq (}%rdi\texttt{), }%rax \quad \# \ t0 = \*xp \\
\texttt{movq (}%rsi\texttt{), }%rdx \quad \# \ t1 = \*yp \\
\texttt{movq }%rdx, (\%rdi) \quad \# \ \*xp = \ t1 \\
\texttt{movq }%rax, (\%rsi) \quad \# \ \*yp = \ t0 \\
\texttt{ret}
\end{quote}
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></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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</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>123</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:** \((R)\) \(\text{Mem}[\text{Reg}[R]]\)
  - 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:** \(\text{movq} \ 8(\%rbp), \ %rdx\)

\(\text{rbp} + 8\)
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)
  - \( D(Rb, Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \) \( (S=1) \)
  - \( (Rb, Ri, S) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]*S] \) \( (D=0) \)
  - \( (Rb, Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \) \( (S=1, D=0) \)
  - \( (Ri, S) \) \( \text{Mem}[\text{Reg}[Ri]*S] \) \( (Rb=0, D=0) \)

\(^c\text{so reg name not interpreted as Rb}\)
## Address Computation Examples

### Expression Address Computation Address

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address (8 bytes wide)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D(R_b, R_i, S) )</td>
<td>( \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] \times S + D] )</td>
<td>( \text{Reg}[R_b] + D = 0xf000 + 0x8 ) ( 0xf008 )</td>
</tr>
<tr>
<td>( 0x8 (%rdx) )</td>
<td></td>
<td>( 0x80 )</td>
</tr>
<tr>
<td>( (%rdx, %rcx) )</td>
<td></td>
<td>( 0x100 )</td>
</tr>
<tr>
<td>( (%rdx, %rcx, 4) )</td>
<td></td>
<td>( 0x400 )</td>
</tr>
<tr>
<td>( 0x80 (, %rdx, 2) )</td>
<td></td>
<td>( 0x1e080 )</td>
</tr>
</tbody>
</table>

- \( S = 1 \)
- \( D = 0 \)
- \( \text{Reg}[R_b] = 0 \)
- \( \text{Reg}[R_i] = 0 \)

\[ (\text{if not specified}) \]

(default values)

ignore the memory access for now

\( 0xf000 \times 2 \)
\( 0xf000 << 1 = 0x1e000 \)

\[ 1111 \ 0000 \]
\[ 1110 \ 0000 \ldots 0 \]
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