x86-64 Programming I
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

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

- hw7 due Monday, hw8 due Wednesday
- Lab 1b due Monday (1/27) at 11:59 pm
  - You have lab late days available
Roadmap

C:

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

Java:

```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
100010011100010
11000001111101000001111
```
Architecture Sits at the Hardware Interface

**Source code**
- Different applications or algorithms

**Compiler**
- Perform optimizations, generate instructions

**Architecture**
- Instruction set

**Hardware**
- Different implementations

- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- AMD Opteron
- AMD Athlon
- ARM Cortex-A53
- Apple A7

- C Language
  - Program A
  - Program B
  - Your program

- GCC
- Clang

- x86-64
- ARMv8 (AArch64/A64)

"we will be using"
Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469
Instruction Set Architectures

- The ISA defines:
  - The system’s state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state
Instruction Set Philosophies

- **Complex Instruction Set Computing (CISC):** Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

- **Reduced Instruction Set Computing (RISC):** Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
General ISA Design Decisions

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded?

- Registers
  - How many registers are there?
  - How wide are they?

- Memory
  - How do you specify a memory location?
Mainstream ISAs

**x86**
- Designer: Intel, AMD
- Bits: 16-bit, 32-bit and 64-bit
- Design: CISC
- Type: Register-memory
- Encoding: Variable (1 to 15 bytes)
- Endianness: Little

**ARM architectures**
- Designer: ARM Holdings
- Bits: 32-bit, 64-bit
- Introduced: 1985; 31 years ago
- Design: RISC
- Type: Register-Register
- Encoding: AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility
- Endianness: Little

**MIPS**
- Designer: MIPS Technologies, Inc.
- Bits: 64-bit (32–64)
- Introduced: 1981; 35 years ago
- Design: RISC
- Type: Register-Register
- Encoding: Fixed
- Endianness: Bi

Macbooks & PCs
(Core i3, i5, i7, M)
x86-64 Instruction Set

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

Digital home & networking equipment
(Blu-ray, PlayStation 2)
MIPS Instruction Set
Writing Assembly Code? In 2019???

- 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, ...
  - Must know which you’re reading
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)

<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>
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)

- **%esi** (mostly obsolete)
- **%edi** (mostly obsolete)
- **%esp** (mostly obsolete)
- **%ebp** (mostly obsolete)

32 bits (same as last slide)  |  8 bits
## 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>~ 8 GiB</td>
<td>vs.</td>
<td>Small</td>
</tr>
<tr>
<td>~50-100 ns</td>
<td>vs.</td>
<td>Fast</td>
</tr>
<tr>
<td>Can “grow” as needed while program runs</td>
<td>vs.</td>
<td>Static</td>
</tr>
</tbody>
</table>

### Addresses
- 0x7FFFD024C3DC

### Big
- ~8 GiB

### Slow
- ~50-100 ns

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

### vs.
- Names
- Small
- Fast
- Static

### Notes
- (16 x 8 B) = 128 B
- sub-nanosecond timescale
- 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”
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_<width specifier> 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”
Operand Combinations

\[ \text{movq} \begin{cases} \text{Reg} & \text{movq} 0x4, \%rax \\ \text{Mem} & \text{movq} -147, (\%rax) \\ \text{Reg} & \text{movq} \%rax, \%rdx \\ \text{Mem} & \text{movq} \%rax, (\%rdx) \\ \text{Mem} & \text{movq} (\%rax), \%rdx \end{cases} \]

\[ \begin{align*} \text{var}_a &= 0x4; \\
\text{*p}_a &= -147; \\
\text{var}_d &= \text{var}_a; \\
\text{*p}_d &= \text{var}_a; \\
\text{var}_d &= \text{*p}_a; \end{align*} \]

\[ \text{\textbullet\textbullet\textbullet} \text{ Cannot do memory-memory transfer with a single instruction} \]

- How would you do it?

  1. Mem → Reg
     \[ \text{movq} (\%rax), \%rdx \]
  2. Reg → Mem
     \[ \text{movq} \%rdx, (\%rbx) \]
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>
Some Arithmetic Operations

- Unary (one-operand) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq $dst$</td>
<td>$dst = dst + 1$</td>
</tr>
<tr>
<td>decq $dst$</td>
<td>$dst = dst - 1$</td>
</tr>
<tr>
<td>negq $dst$</td>
<td>$dst = -dst$</td>
</tr>
<tr>
<td>notq $dst$</td>
<td>$dst = \sim dst$</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)

- `%rdi` 1st argument (x)
- `%rsi` 2nd argument (y)
- `%rax` return value

```
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;
}
```

`swap:`

```asm
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`<br>
- `%rsi`<br>
- `%rax`<br>
- `%rdx`

**Memory**

swap:

- `movq (%rdi), %rax`
- `movq (%rsi), %rdx`
- `movq %rdx, (%rdi)`
- `movq %rax, (%rsi)`
- `ret`

**Register ↔ Variable**

- `%rdi ↔ xp`
- `%rsi ↔ yp`
- `%rax ↔ t0`
- `%rdx ↔ t1`
# 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></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</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
```

Comment
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></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Location</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:

```assembly
    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></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()`

**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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x100</td>
<td>456</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></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><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>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 Code**

```
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:** \texttt{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:** \texttt{movq 8(\%rbp), \%rdx}
Complete Memory Addressing Modes

\[
\text{ar} \left[ (ar + i) \cdot S + D \right] = \text{ar} + i \cdot \text{size(type)}
\]

- **General:**
  - \( D(Rb, Ri, S) \) \( \xrightarrow{\text{Mem[Reg[Rb]+Reg[Ri]*S+D]}} \)
    - \( Rb \): 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)
  - \( D(Rb, Ri) \) \( \xrightarrow{\text{Mem[Reg[Rb]+Reg[Ri]+D]}} \) (\( S=1 \))
  - \( (Rb, Ri, S) \) \( \xrightarrow{\text{Mem[Reg[Rb]+Reg[Ri]*S]}} \) (\( D=0 \))
  - \( (Rb, Ri) \) \( \xrightarrow{\text{Mem[Reg[Rb]+Reg[Ri]]}} \) (\( S=1, D=0 \))
  - \( (,Ri,S) \) \( \xrightarrow{\text{Mem[Reg[Ri]*S]}} \) (\( Rb=0, D=0 \))
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>0xf008 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0xf000 + 0x0100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>0xf000 + 0x0400</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx, 2)</td>
<td>6x1e08 0</td>
<td></td>
</tr>
</tbody>
</table>

\[
D(Rb, Ri, S) \rightarrow \text{Mem[Reg[Rb]+Reg[Ri]*S+D]}
\]
Summary

- x86-64 is a complex instruction set computing (CISC) architecture
  - 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