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
1000100111000010
110000011111101000011111
```

**OS:**

- Windows 10
- OS X Yosemite

**Memory & data**
- Integers & floats

**x86 assembly**
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation

**Java vs. C**
Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

C Language
Program A

Program B

Your program

GCC

Clang

x86-64

ARMv8
(AArch64/A64)

Intel Pentium 4

Intel Core 2

Intel Core i7

AMD Opteron

AMD Athlon

ARM Cortex-A53

Apple A7
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
- **Introduced**: 1978 (16-bit), 1985 (32-bit), 2003 (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\(^1\)
- **Endianness**: Bi (little as default)

## 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

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
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>Bits</th>
<th>Purpose</th>
<th>Name Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>32</td>
<td>accumulate</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%ecx</td>
<td>32</td>
<td>counter</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%edx</td>
<td>32</td>
<td>data</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%ebx</td>
<td>32</td>
<td>base</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%esi</td>
<td>32</td>
<td>source index</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%edi</td>
<td>32</td>
<td>destination index</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%esp</td>
<td>32</td>
<td>stack pointer</td>
<td>mostly obsolete</td>
</tr>
<tr>
<td>%ebp</td>
<td>32</td>
<td>base pointer</td>
<td>mostly obsolete</td>
</tr>
</tbody>
</table>

#### General Purpose

- %eax: 32-bit general purpose register
- %ecx: 32-bit general purpose register
- %edx: 32-bit general purpose register
- %ebx: 32-bit general purpose register
- %esi: 32-bit general purpose register
- %edi: 32-bit general purpose register
- %esp: 32-bit stack pointer
- %ebp: 32-bit base pointer

#### Accumulate

- %ah: Accumulate register
- %al: Accumulate register

#### Counter

- %ch: Counter register
- %cl: Counter register

#### Data

- %dh: Data register
- %dl: Data register

#### Base

- %bh: Base register
- %bl: Base register

#### Source Index

- %si: Source index register

#### Destination Index

- %di: Destination index register

#### 16-bit Virtual Registers (backwards compatibility)

- %ax, %bx, %dx, %bx

#### Name Origin

- %esp, %ebp
Memory vs. Registers

- **Addresses**
  - 0x7FFF024C3DC

- **Big**
  - ~8 GiB

- **Slow**
  - ~50-100 ns

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

- **vs.**
  - **Names**
    - %rdi

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

- **Slow vs. Fast**
  - sub-nanosecond timescale

- **Dynamic vs. Static**
  - fixed number in hardware
Three Basic Kinds of Instructions

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

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, %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 (mov)
- Arithmetic operations
- Memory addressing modes
  - swap example
- Address computation instruction (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”
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?
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 / src</td>
</tr>
</tbody>
</table>

*Format, Computation, and 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>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 = ~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>

**Assembly Code**

```
simple_arith:
    addq %rdi, %rsi
    imulq $3, %rsi
    movq %rsi, %rax
    ret
```

```c
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:
    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;
}
```

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

Registers and Memory Diagram:

- `%rdi`-linked with `xp`
- `%rsi`-linked with `yp`
- `%rax`-linked with `t0`
- `%rdx`-linked with `t1`
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></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>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 `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></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>

### Swap Code

```
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>456</td>
</tr>
</tbody>
</table>

### Word Address

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>

```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>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>

### Code Snippet

```
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>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>

### Word Address

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
</tr>
<tr>
<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
  - \textbf{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
  - \textbf{Example:} \texttt{movq 8(\%rbp), \%rdx}
Complete Memory Addressing Modes

❖ General:

- \(D(Rb, Ri, S)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]\times S+D]\)
  - \(Rb\): 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)

- \(D(Rb, Ri)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]\) \((S=1)\)
- \((Rb, Ri, S)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]\times S]\) \((D=0)\)
- \((Rb, Ri)\) \(\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]\) \((S=1, D=0)\)
- \((, Ri, S)\) \(\text{Mem}[\text{Reg}[Ri]\times S]\) \((Rb=0, D=0)\)
## 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>D(Rb,Ri,S) → Mem[Reg[Rb]+Reg[Ri]*S+D]</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>
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