x86-64 Programming I
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee
Natalie Andreeva
Teagan Horkan

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

- Lab 1b due on Thursday (7/5)
  - Submit `bits.c, lab1reflect.txt`
  - Josie has OH on Thursday 1–3 pm
- Homework 2 due next Wednesday (7/11)
  - On Integers, Floating Point, and x86-64
- No lecture on Wednesday!
- Section Thursday on Floating Point
Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - “Gaps” produced in representable numbers means we can lose precision, unlike ints
    - Some “simple fractions” have no exact representation (e.g. 0.2)
    - “Every operation gets a slightly wrong result”

- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results

- Never test floating point values for equality!
- Careful when converting between ints and floats!
Number Representation Really Matters

- **1991**: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996**: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000**: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038**: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (float division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
C:

```c
#include <stdlib.h> // For malloc and free

struct car {
    int miles;
    int gals;
};

int main() {
    car *c = malloc(sizeof(car));
    c->miles = 100;
    c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
    return 0;
}
```

Java:

```java
public class Car {
    private int miles;
    private int gals;

    public Car() {
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        return (float) miles / gals;
    }
}
```

Assembly language:

```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ... 
    popq %rbp
    ret
```

Machine code:

```
0111010000110000
100011010000011000
1000100111000010
110000011111101000011111
```
What makes programs run fast(er)?
HW Interface Affects Performance

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

Compiler
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

we will be using
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? *instructions are data! binary encoding*

- **Registers**
  - How many registers are there?
  - How wide are they? *word size (64 bits)*

- **Memory**
  - How do you specify a memory location? *an address is a word size different ways to specify/"build" an address*
# 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
- **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**: Big

### Applications
- **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
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, 470

- Are the following part of the architecture?
  - Number of registers? **Yes**
  - How about CPU frequency? **No**
  - Cache size? Memory size? **No - modular**
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 (untyped pointers)

- Floating point data of 4, 8, or 2x8, 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
  - *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)
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.

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
</table>

- **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
     - %reg = Mem[address]
   - **Store** register data into memory
     - Mem[address] = %reg

2) Perform arithmetic operation on register or memory data
   - c = a + b;    z = x << y;    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”

\[
\begin{align*}
%rax & \quad \%rcx \\
\%rdx & \quad \%rbx \\
\%rsi & \quad \%rdi \\
\%rsp & \quad \%rbp \\
%rN & \quad r8 - r15
\end{align*}
\]
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

- \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

<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>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>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_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**
       
\[
\begin{align*}
\text{movq} & \quad (\%rax), \%rdx \\
\text{movq} & \quad %rdx, (\%rbx)
\end{align*}
\]
x86-64 Introduction

- Arithmetic operations
- Memory addressing modes
  - swap example
- Address computation instruction (lea)
Some Arithmetic Operations

- Binary (two-operand) instructions:
  - **Maximum of one memory operand**
  - Beware argument order (**AT&T syntax**)
  - No notion of datatypes
    - Just bits!
    - Only arithmetic vs. logical shifts
  - How do you implement “r3 = r1 + r2”? 
    - **addq** `src, dst`  
      - `dst = dst + src`
    - **subq** `src, dst`  
      - `dst = dst - src`
    - **imulq** `src, dst`  
      - `dst = dst * src`
    - **sarq** `src, dst`  
      - `dst = dst >> src`
    - **shrq** `src, dst`  
      - `dst = dst >> src`
    - **shlq** `src, dst`  
      - `dst = dst << src`
    - **xorq** `src, dst`  
      - `dst = dst ^ src`
    - **andq** `src, dst`  
      - `dst = dst & src`
    - **orq** `src, dst`  
      - `dst = dst | src`

- **Format Computation**

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

- Signed mult

- Arithmetic

- Logical

(same as salq)
Some Arithmetic Operations

- **Unary (one-operand) Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Computation Type</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;
}
```

<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
```

---

*Convention!*
Example of Basic Addressing Modes

```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
```
### Understanding `swap()`

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

#### Registers

- `%rdi` → `xp`
- `%rsi` → `yp`
- `%rax` → `t0`
- `%rdx` → `t1`

#### Memory

#### Register ↔ Variable

- `%rdi` ↔ `xp`
- `%rsi` ↔ `yp`
- `%rax` ↔ `t0`
- `%rdx` ↔ `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>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### `swap:`

```assembly
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><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>
</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

- 123
- 456

### 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</td>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td></td>
<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
```
Understanding `swap()`

**Registers**

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

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>456</code></td>
</tr>
</tbody>
</table>

**Word Address**

- `0x120`
- `0x118`
- `0x110`
- `0x108`
- `0x100`

**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><code>%rdi</code></td>
<td></td>
<td>456</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
<td>0x108</td>
</tr>
</tbody>
</table>

These didn’t change!

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

- x86-64 is a complex instruction set computing (CISC) architecture
- **registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly operands include immediates, registers, and data at specified memory locations