x86-64 Programming I
CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou

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

- hw7 due Friday – 11am, hw8 due Monday – 11am
- Lab 1b due Monday (4/20)
  - Submit `bits.c` and `lab1Breflect.txt`

- You must log on with your @uw google account to access!!
  - Google doc for 11:30 Lecture: https://tinyurl.com/351-04-15A

- Week 2 Feedback Survey
  - https://catalyst.uw.edu/webq/survey/rea2000/388285
### Roadmap

**C:**

```c
struct car {
    int miles;
    int gals;
};

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

**Java:**

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

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

    public int getMiles() {
        return miles;
    }
}

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMiles();
```

**Assembly language:**

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

**Machine code:**

```
0111010000011000 1000110100000100
1000100111000010 110000011111101000001111
```

**Computer system:**

- **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

Compiler

GCC

Clang

x86-64

ARMv8 (AArch64/A64)

Hardware

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
- Design: CISC
- Type: Register-memory
- Encoding: Variable (1 to 15 bytes)
- Endianness: Little

**ARM**
- 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: 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 2020???

- 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 ($\%\text{rip}$ 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>Type</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
<td>Name Origin (mostly obsolete)</td>
</tr>
<tr>
<td>%ax</td>
<td>counter</td>
<td></td>
</tr>
<tr>
<td>%ah</td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>%al</td>
<td>data</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%cx</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%ch</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%cl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>source index</td>
<td></td>
</tr>
<tr>
<td>%dx</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%dh</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%dl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>destination index</td>
<td></td>
</tr>
<tr>
<td>%bx</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%bh</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%bl</td>
<td>base</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%si</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
<td></td>
</tr>
<tr>
<td>%sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
<td></td>
</tr>
<tr>
<td>%bp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Memory vs. Registers

- Addresses vs. Names
  - 0x7FFFFD024C3DC 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 vs. Static
  - Can “grow” as needed while program runs vs. 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

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>Mem</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>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>sh1q 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>

Format Computation

- Signed mult
- Arithmetic
- Logical
  - (same as salq)

Operand size specifier
Polling Question [Asm I – a]

- Assume: r3 is in \%rcx, r1 is in \%rax, and r2 is in \%rbx
  which of the following would implement:
  \[ r3 = r1 + r2 \]
  - Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. `addq %rax, %rbx, %rcx`

B. `addq %rcx, %rax, %rbx`

C. `movq %rax, %rcx
addq %rbx, %rcx`

D. `movq (%rbx), %rcx
addq (%rax), %rcx`

E. We’re lost...
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;
}
```

<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
```
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
```
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

Registers:
- %rdi
- %rsi
- %rax
- %rdx

Memory:

Variable:
- xp
- yp
- t0
- t1

Register ➔ Variable:
- %rdi ➔ xp
- %rsi ➔ yp
- %rax ➔ t0
- %rdx ➔ t1

Understanding swap()
Understanding `swap()`

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>0x120 0x118</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>0x110 0x108</td>
</tr>
<tr>
<td>%rdx</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>%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>
</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>
<tr>
<td>123</td>
</tr>
<tr>
<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 \texttt{swap()} \\

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Registers} & \textbf{Memory} \\
\hline
%rdi & 0x120 & 123 & 0x120 \\
%rsi & 0x100 & 0x118 & 0x110 \\
%rax & 123 & 0x108 & 0x100 \\
%rdx & 456 & 456 & \\
\hline
\end{tabular}
\end{center}

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

**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>%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>456</td>
</tr>
<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>

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:** \(\text{movq} 8(\%rbp), \%rdx\)
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) \rightarrow 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