x86-64 Programming I

CSE 351 Spring 2021

Instructor:
Ruth Anderson

Teaching Assistants:
Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Neil Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu

http://xkcd.com/409/
Admininstrivia

- hw6 due TONIGHT (4/14) @ 11:59 pm
- Lab 1a closes TONIGHT (4/12) @ 11:59 pm
  - Submit POINTER.c and lab1Areflect.txt
  - Make sure you check the Gradescope autograder output!
  - Can use late day tokens to submit up until Wed 11:59 pm
- Lab 1b, due 4/19
  - Submit aisle_manager.c, store_client.c, and lab1Breflect.txt
- Questions Docs: Use @uw google account to access!!
  - https://tinyurl.com/CSE351-21sp-Questions
Reading Review

- Terminology:
  - Instruction Set Architecture (ISA): CISC vs. RISC
  - Instructions: data transfer, arithmetic/logical, control flow
    - Size specifiers: b, w, l, q
  - Operands: immediates, registers, memory
    - Memory operand: displacement, base register, index register, scale factor
Review Questions

- Assume that the register `%rax` currently holds the value `0x 01 02 03 04 05 06 07 08`

- Answer the questions on Ed Lessons about the following instruction (`<instr> <src> <dst>`):

  ```
xorw $-1, %ax
  ```

  - Operation type:
  - Operand types:
  - Operation width:
  - Result in `%rax`: 
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
```

Computer system:

```
Windows 10

OS X Yosemite
```

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory & caches
Java vs. C
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
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?
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
# 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)
- **Branching**: Condition code
- **Endianness**: Little

## ARM
- **Designer**: Arm Holdings
- **Bits**: 32-bit, 64-bit
- **Introduced**: 1985
- **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]
- **Branching**: Condition code, compare and branch
- **Endianness**: Bi

## MIPS
- **Designer**: MIPS Technologies, Imagination Technologies
- **Bits**: 64-bit (32 → 64)
- **Introduced**: 1985
- **Version**: MIPS32/64 Release 6 (2014)
- **Design**: RISC
- **Type**: Register-Register
- **Encoding**: Fixed
- **Branching**: Compare and branch
- **Endianness**: Bi

## 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**
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
Writing Assembly Code? In 2021???

- 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>%eax</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>%ebx</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>%ecx</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>%edx</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>%esi</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>%edi</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>%esp</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>%ebp</td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>%r8d</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>%r9d</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>%r10d</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>%r11d</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>%r12d</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>%r13d</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>%r14d</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>%r15d</td>
</tr>
</tbody>
</table>

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

General Purpose

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

- **Addresses**
  - 0x7FFFD024C3DC
- **Big**
  - ~8 GiB
- **Slow**
  - ~50-100 ns
- **Dynamic**
  - Can “grow” as needed while program runs

- **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
   - \( \%\text{reg} = \text{Mem}[\text{address}] \)
   - **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!
Instruction Sizes and Operands

- **Size specifiers**
  - b = 1-byte “byte”, w = 2-byte “word”, l = 4-byte “long word”, q = 8-byte “quad word”
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names

- **Operand types**
  - *Immediate*: Constant integer data ($)
  - *Register*: 1 of 16 integer registers (%)
  - *Memory*: Consecutive bytes of memory at a computed address (())
x86-64 Introduction

- Data transfer instruction (mov)
- Arithmetic operations
- Memory addressing modes
  - swap example
Moving Data

- General form: `mov_ source, destination`
  - Really more of a “copy” than a “move”
  - Like all instructions, missing letter (_,) is the size specifier
  - Lots of these in typical code
Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>movq $-147, (%rax)</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>movq %rax, %rdx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</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

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Operand size specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>dst = dst + src</td>
<td>(dst += src)</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>dst = dst – src</td>
<td>signed mult</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>dst = dst * src</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td>Logical</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
<td>(same as salq)</td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>dst = dst &lt;&lt; src</td>
<td></td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>dst = dst ^ src</td>
<td></td>
</tr>
<tr>
<td>andq src, dst</td>
<td>dst = dst &amp; src</td>
<td></td>
</tr>
<tr>
<td>orq src, dst</td>
<td>dst = dst</td>
<td>src</td>
</tr>
</tbody>
</table>
Practice Question

- Which of the following are valid implementations of $rcx = rax + rbx$?

  - addq %rax, %rcx
    - addq %rbx, %rcx
  - movq $0, %rcx
    - movq %rax, %rcx
    - addq %rbx, %rcx
  - xorq %rax, %rax
    - addq %rax, %rcx
    - addq %rbx, %rcx
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>

- `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;
}
```

Compiler Explorer: [https://godbolt.org/z/zc4Pcq](https://godbolt.org/z/zc4Pcq)
Understanding \texttt{swap()}\\

\begin{verbatim}
void swap(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
\end{verbatim}

\begin{tabular}{|c|}
\hline
\textbf{Register} & \textbf{Variable} \\
\hline
%rdi & xp \\
%rsi & yp \\
%rax & t0 \\
%rdx & t1 \\
\hline
\end{tabular}
## 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></td>
</tr>
<tr>
<td>%rdx</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:

```
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></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:

```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>
<th>123</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>456</td>
</tr>
</tbody>
</table>

### Swap Function

```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()`

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>0x120, 0x118</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
<td>0x110, 0x108</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
<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
```
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>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
</tr>
<tr>
<td>123</td>
</tr>
</tbody>
</table>

### Word Address

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

### Code

```assembly
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:** `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:** `movq 8(%rbp), %rdx`
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)
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0x8(%rdx)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(%rdx,%rcx)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(%rdx,%rcx,4)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0x80(,%rdx,2)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D(Rb,Ri,S) → 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