x86-64 Programming I
CSE 351 Spring 2021
Instructor: Ruth Anderson

http://xkcd.com/409/
Administrivia

- 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](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 0x0102030405060708.

- Answer the questions on Ed Lessons about the following instruction (\texttt{<instr> <src> <dst>}):
  \[
  \text{xorw} \ -1, \ %ax
  \]

  - Operation type:
  - Operand types:
  - Operation width:
  - Result in %rax:
    \[
    \begin{align*}
    0 \times 07 \ 08 \ \land \ 0 \times \text{FFFF} \Rightarrow \ & %\text{rax}: 0 \times 01 \ 02 \ 03 \ 04 \ 05 \ 06 \ F8 \ F7
    \end{align*}
    \]
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
11000001111101000011111
```

Computer system:

OS:
- Windows 10
- OS X Yosemite

Memory & data
- Integers & floats
- x86 assembly

Procedures & stacks
- Executables
- Arrays & structs
- Processes
- Virtual memory
- Memory allocation

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? 16
  - How wide are they? 64 bits

- **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

<table>
<thead>
<tr>
<th>Designer</th>
<th>Intel, AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>16-bit, 32-bit and 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
</tr>
<tr>
<td>Design</td>
<td>CISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-memory</td>
</tr>
<tr>
<td>Encoding</td>
<td>Variable (1 to 15 bytes)</td>
</tr>
<tr>
<td>Branching</td>
<td>Condition code</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Designer</th>
<th>Arm Holdings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Introduced</td>
<td>1985</td>
</tr>
<tr>
<td>Design</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</td>
<td>AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions; ARMv7 user-space compatibility.</td>
</tr>
<tr>
<td>Branching</td>
<td>Condition code, compare and branch</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little (as default)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Designer</th>
<th>MIPS Technologies, Imagination Technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>64-bit (32 → 64)</td>
</tr>
<tr>
<td>Introduced</td>
<td>1985</td>
</tr>
<tr>
<td>Version</td>
<td>MIPS32/64 Release 6 (2014)</td>
</tr>
<tr>
<td>Design</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</td>
<td>Fixed</td>
</tr>
<tr>
<td>Branching</td>
<td>Compare and branch</td>
</tr>
<tr>
<td>Endianness</td>
<td>Bi</td>
</tr>
</tbody>
</table>

**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

- 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
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

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

<table>
<thead>
<tr>
<th>Register</th>
<th>64-bit Name</th>
<th>32-bit Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td><code>%eax</code></td>
<td></td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td><code>%ebx</code></td>
<td></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td><code>%ecx</code></td>
<td></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td><code>%edx</code></td>
<td></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td><code>%esi</code></td>
<td></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td><code>%edi</code></td>
<td></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td><code>%esp</code></td>
<td></td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><code>%ebp</code></td>
<td></td>
</tr>
<tr>
<td><code>%r8</code></td>
<td><code>%r8d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r9</code></td>
<td><code>%r9d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r10</code></td>
<td><code>%r10d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r11</code></td>
<td><code>%r11d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r12</code></td>
<td><code>%r12d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r13</code></td>
<td><code>%r13d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r14</code></td>
<td><code>%r14d</code></td>
<td></td>
</tr>
<tr>
<td><code>%r15</code></td>
<td><code>%r15d</code></td>
<td></td>
</tr>
</tbody>
</table>
Some 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 vs. Registers

- Addresses
  - 0x7FFFFFFD024C3DC

- Big
  - ~ 8 GiB

- Slow
  - ~50-100 ns

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

vs.

- Names
  - %rdi

- Small
  - (16 \times 8 \text{ B}) = 128 \text{ B}

- Fast
  - sub-nanosecond timescale

vs.

- 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>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>Reg</td>
<td>Mem</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?
    1. **Mem → Reg**
       - movq (%rax), %rdx
    2. **Reg → Mem**
       - 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

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq src, dst</code></td>
<td><code>dst = dst + src</code></td>
</tr>
<tr>
<td><code>subq src, dst</code></td>
<td><code>dst = dst - src</code></td>
</tr>
<tr>
<td><code>imulq src, dst</code></td>
<td><code>dst = dst * src</code></td>
</tr>
<tr>
<td><code>sarq src, dst</code></td>
<td><code>dst = dst &gt;&gt; src</code></td>
</tr>
<tr>
<td><code>shrq src, dst</code></td>
<td><code>dst = dst &gt;&gt; src</code></td>
</tr>
<tr>
<td><code>shlq src, dst</code></td>
<td><code>dst = dst &lt;&lt; src</code></td>
</tr>
<tr>
<td><code>xorq src, dst</code></td>
<td><code>dst = dst ^ src</code></td>
</tr>
<tr>
<td><code>andq src, dst</code></td>
<td><code>dst = dst &amp; src</code></td>
</tr>
<tr>
<td><code>orq src, dst</code></td>
<td>`dst = dst</td>
</tr>
</tbody>
</table>

 Operands: `src` (source), `dst` (destination) - `Imm`, `Reg`, or `Mem`

Memory operand: `Reg` or `Mem`

Signed multiplication: `imulq src, dst`

Arithmetic shifts: `sarq` and `shrq`

Logical shifts: `shlq` (same as `salq`)

Memory operand: `(dst += src)`

Operand size specifier: `[b, w, l, q]`
Practice Question

Which of the following are valid implementations of \( \text{rcx} = \text{rax} + \text{rbx} \)?

- \( \text{addq} \ %\text{rax}, \ %\text{rcx} \)
- \( \text{addq} \ %\text{rbx}, \ %\text{rcx} \)
- \( \text{movq} \ %\text{rax}, \ %\text{rcx} \)
- \( \text{addq} \ %\text{rbx}, \ %\text{rcx} \)
- \( \text{xorq} \ %\text{rax}, \ %\text{rax} \)
- \( \text{addq} \ %\text{rbx}, \ %\text{rcx} \)

\[ \text{rcx} = \ ??? \]
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>

Calling convention:

```
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;
}
```

Compiler Explorer:
[https://godbolt.org/z/zc4Pcq](https://godbolt.org/z/zc4Pcq)
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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

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

Register | Variable
---------|---------
%rdi ⇔ xp
%rsi ⇔ yp
%rax ⇔ t0
%rdx ⇔ t1

%rdi
%rsi
%rax
%rdx
# 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</td>
</tr>
<tr>
<td>%rdx</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>%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>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x100</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>

### 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>456</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 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()`

### 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>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></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>%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>
<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></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:** \(\text{movq } (\text{%rcx}), \text{ %rax}\)

- **Displacement:** \(D(R)\)  \(\text{Mem}[\text{Reg}[R]+D]\)
  - Data in register \(R\) specifies the \textit{start} of some memory region
  - Constant displacement \(D\) specifies the offset from that address
  - **Example:** \(\text{movq } 8(\text{%rbp}), \text{ %rdx}\)
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>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</td>
</tr>
</tbody>
</table>

\[
D(R_b, R_i, S) \rightarrow \text{Mem}[\text{Reg}[R_b] + \text{Reg}[R_i] \times S + D]
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8 (%rdx)</td>
<td></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