

The Hardware/Software Interface

x86-64 Programming I

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus

Grace Zhou

Jiuyang Lyu

Kurt Gu

Mendel Carroll

Naama Amiel

Rose Maresh

Violet Monserate

Divya Ramu

Jessie Sun

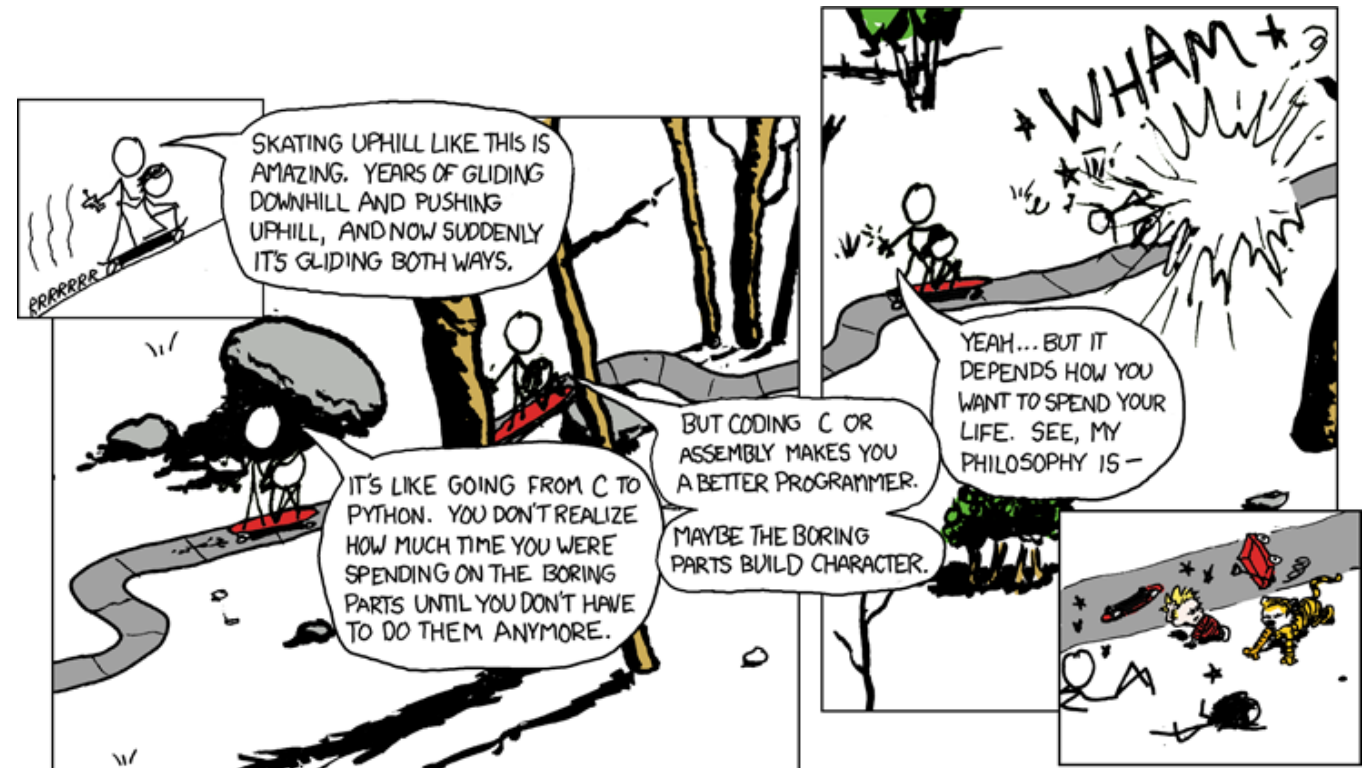
Kanishka Singh

Liander Rainbolt

Ming Yan

Pollux Chen

Soham Bhosale



<http://xkcd.com/409/>

Relevant Course Information

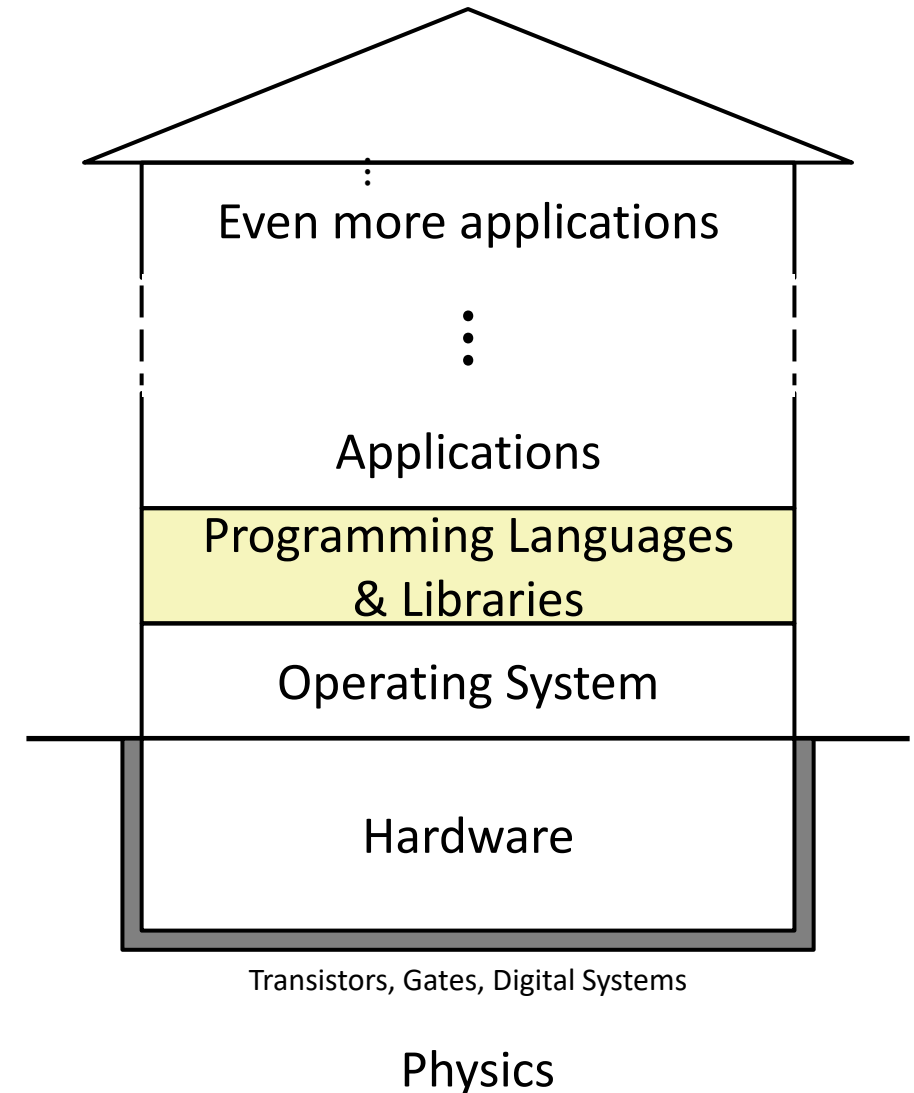
- ❖ HW5 due tonight, HW6 due Friday, HW7 due Monday
- ❖ Lab 1a: last chance to submit is tonight @ 11:59 pm
 - One submission per partnership
 - Make sure you check the Gradescope autograder output!
 - Grades hopefully released by end of Sunday (10/12)
- ❖ Lab 1b due Monday (10/13)
 - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
 - Section tomorrow should help with Lab 1b

Getting Help with 351

- ❖ Lecture recordings, readings, inked slides, section worksheet solutions
- ❖ Attend lectures and office hours
 - Can also chat with other students– help each other learn!
- ❖ Form a study group!
 - Good for everything but labs, which should be done in pairs
 - Communicate regularly, use the class terminology, ask and answer each others' questions, show up to OH together
- ❖ Post on Ed Discussion
- ❖ Request a 1-on-1 meeting
 - Available on a limited basis for special circumstances

House of Computing Check-In

- ❖ Topic Group 2: **Programs**
 - **x86-64 Assembly**, Procedures, Stacks, Executables
- ❖ How are programs created and executed on a CPU?
 - *How does your source code become something that your computer understands?*
 - How does the CPU organize and manipulate local data?

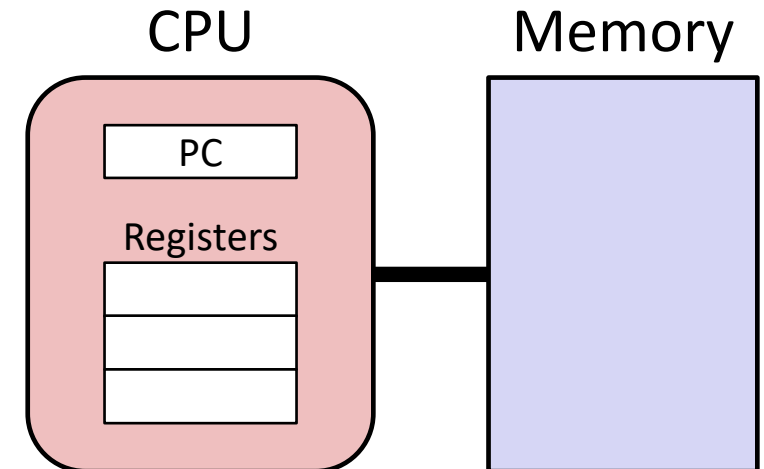


Lecture Outline (1/4)

- ❖ **Instruction Set Architectures (ISAs)**
- ❖ x86-64 Syntax and Instructions
- ❖ x86-64 Operands
- ❖ First Assembly Examples

Instruction Set Architectures (Review)

- ❖ **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
 - What is directly visible to software – the “contract” or “blueprint” between hardware and software
 - The system’s *state* (e.g., program counter, registers, memory)
 - The set of *instructions* the CPU can execute
 - The *effect* that each of these instructions will have on the system state
 - This is separate from the *microarchitecture*, which is the implementation of the architecture
 - Take EE/CSE 469 if interested



Instruction Set Philosophies (Review)

- ❖ *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

Windows desktop/laptops
(Core i3, i5, i7, Ryzen)
[x86-64 Instruction Set](#)



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]

Smartphone-like devices
(iPhone, Android, Raspberry Pi)
Apple products (ca. 2020-)
(Macbook, Mac Mini)
[ARM Instruction Set](#)

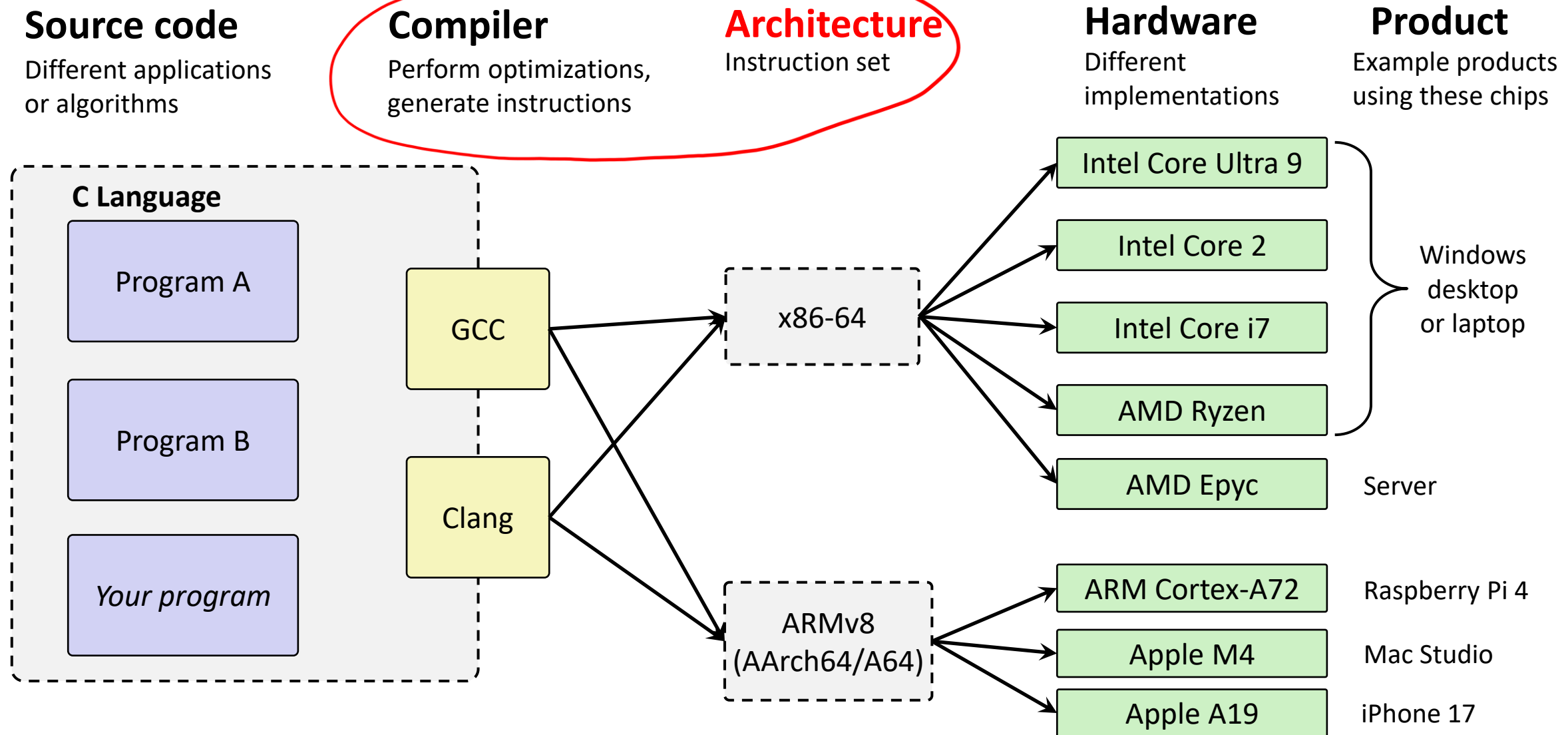


RISC-V

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load-store
Encoding	Variable
Endianness	Little ^{[1][3]}

Mostly research
(some traction in embedded)
[RISC-V Instruction Set](#)

Architecture Sits at the Hardware Interface (1/2)



Architecture Sits at the Hardware Interface (2/2)

Source code

Different applications
or algorithms

Compiler

Perform optimizations,
generate instructions

Architecture

Instruction set

Hardware

Different
implementations

```
long mult2(long m1, long m2);  
  
void multstore(long x, long y, long* d) {  
    long t = mult2(x, y);  
    *d = t;  
}
```

```
multstore:  
    pushq %rbx  
    movq %rdx, %rbx  
    call mult2  
    movq %rax, (%rbx)  
    popq %rbx  
    ret
```

Source Code (hex):

```
53  
48 89 d3  
e8 00 00 00 00  
48 89 03  
5b  
c3
```

GCC

Lecture Outline (2/4)

- ❖ Instruction Set Architectures (ISAs)
- ❖ **x86-64 Syntax and Instructions**
- ❖ x86-64 Operands
- ❖ First Assembly Examples

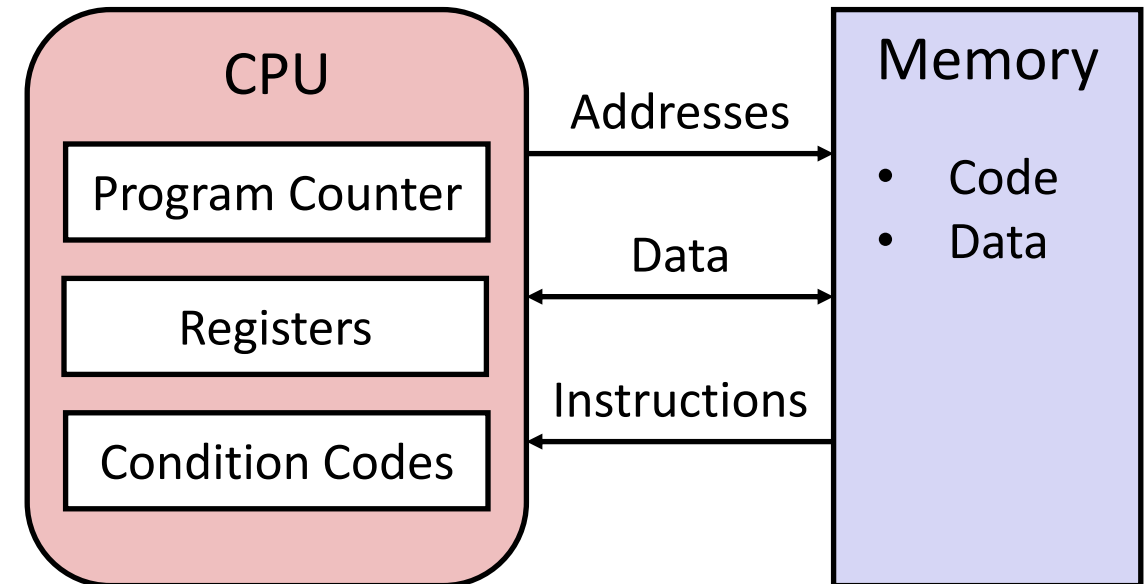
Writing Assembly Code? Who Does That???

- ❖ Chances are, you'll never write a program in assembly, but assembly is the key to understanding the machine-level execution model:
 - Catching bugs where the high-level language model breaks down
 - Implementing systems software
 - What are the “states” of processes that the OS must manage
 - Using special units (*e.g.*, timers, I/O co-processors) inside processor!
 - Fighting malicious software since distributed software is in binary form
 - Fine-tuning program performance (not relevant anymore)
 - Tweaking optimizations that may or may not have been done by the compiler
 - Identifying sources of program inefficiency

Assembly Programmer's View

❖ Programmer-visible state

- Program Counter (%rip in x86-64)
 - Address of next instruction
- General purpose (named) registers
 - Heavily used locations for data manipulation
- Condition codes
 - Store status information about most recent arithmetic operation and used for conditional branching
- Memory
 - Byte-addressable array containing code and user data



x86-64 Assembly Data

- ❖ Data is moved and manipulated in fixed-length chunks with treatment determined by instruction
- ✳ Integral data (*e.g.*, integers, addresses) will use integer operations
 - *e.g.*, `addq %rax, %rbx`
- Floating point data uses separate hardware and these instructions are *extensions* to x86; **not covered in 351**
 - *e.g.*, `addss %xmm0, %xmm1`
- No aggregate types such as arrays or structures, just contiguously allocated bytes in memory
 - Interesting consequences for implementing higher-level language data structures like objects

x86-64 Syntax Note

❖ AT&T syntax ✓

- Destination operand comes last
- Memory specified like:
-0x30(%rcx,%rax,8)
- Comments start with '#'
- Example:
pushq %rbx
movq %rdx, %rbx
call mult2
movq %rax, (%rbx)
popq %rbx
ret

❖ Intel syntax ✗

- Destination operand comes first
- Memory specified like:
[rcx+rax*8-0x30]
- Comments start with ';'
- Example:
push rbx
mov rbx, rdx
call mult2
mov QWORD PTR [rbx], rax
pop rbx
ret

Make sure that you know which one you're reading!

x86-64 Instructions and Sizes (Review)

- ❖ Common formats: instruction name followed by 1-2 operands, separated by commas
 - `instr op` # e.g., `"negq %rsi"` negates the value in `%rsi`
 - `instr src, dst` # e.g., `"addq %rdi, %rax"` does `%rax = %rax+%rdi`
- ❖ Size specifier suffixes
 - **b** = 1-byte “byte”
 - **w** = 2-byte “word”
 - **l** = 4-byte “long word”
 - **q** = 8-byte “quad word”

Due to backward-compatible support for 8086 programs (16-bit machines from 1978!), “word” means 16 bits = 2 bytes in x86 instruction names... 🤔 🤔 🤔

Instruction Types (Review)

❖ Three instruction types:

1) Transfer data between memory and register

- *Load* (`%reg = Mem[address]`) and *store* (`Mem[address] = %reg`)

2) Perform arithmetic/logical operation on register or memory data

- *e.g.*, `c = a + b`; `z = x << y`; `i = h & g`;

3) Control flow: what instruction to execute next

- Unconditional jumps and conditional branches

❖ Moving data: `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

Instructions: Arithmetic & Logical Operations

❖ Unary (one-operand) Instructions:

Format	Computation	
incq <i>dst</i>	$dst = dst + 1$	increment
decq <i>dst</i>	$dst = dst - 1$	decrement
negq <i>dst</i>	$dst = -dst$	negate
notq <i>dst</i>	$dst = \sim dst$	bitwise complement

❖ Binary (two-operand) Instructions:

- Beware argument order!
- No distinction between signed and unsigned
 - Only arithmetic vs. logical shifts

$incq\ x$ $\# x_1 = x_0 + 1$
 $addq\ x, x$ $\# x_2 = x_1 + x_1$
 $x_{new} = 2 * x_{old} + 2$

Format	Computation	
addq <i>src, dst</i>	$dst = dst + src$	(<i>dst += src</i>)
subq <i>src, dst</i>	$dst = dst - src$	
imulq <i>src, dst</i>	$dst = dst * src$	signed mult
sarq <i>src, dst</i>	$dst = dst \gg src$	Arithmetic
shrq <i>src, dst</i>	$dst = dst \gg src$	Logical
shlq <i>src, dst</i>	$dst = dst \ll src$	(same as <code>salq</code>)
xorq <i>src, dst</i>	$dst = dst \wedge src$	
andq <i>src, dst</i>	$dst = dst \& src$	
orq <i>src, dst</i>	$dst = dst src$	

Polling Questions (1/2)

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


 `addq %rax, %rcx`
`addq %rbx, %rcx`
 $rcx = rcx + rax + rbx$

Diagram: Red arrows show `%rax` being added to `%rcx` (labeled `+=`) and `%rbx` being added to `%rcx` (labeled `+=`).


 `movq %rax, %rcx`
`addq %rbx, %rcx`
 $rcx = rax + rbx$

Diagram: Red arrows show `%rax` being moved to `%rcx` (labeled `=`) and `%rbx` being added to `%rcx` (labeled `+=`).


 `movq $0, %rcx`
`addq %rbx, %rcx`
`addq %rax, %rcx`
 $rcx = 0 + rbx + rax$

Diagram: Red arrows show `$0` being moved to `%rcx` (labeled `=`), `%rbx` being added to `%rcx` (labeled `+=`), and `%rax` being added to `%rcx` (labeled `+=`).


 `xorq %rax, %rax` ($rax = 0$)
`addq %rax, %rcx`
`addq %rbx, %rcx`
 $rcx = rcx + 0 + rbx$

Diagram: Red arrows show `%rax` being XORed with `%rax` (labeled `^=`), `%rax` being added to `%rcx` (labeled `+=`), and `%rbx` being added to `%rcx` (labeled `+=`).

Lecture Outline (3/4)

- ❖ Instruction Set Architectures (ISAs)
- ❖ x86-64 Syntax and Instructions
- ❖ **x86-64 Operands**
- ❖ First Assembly Examples

Operand Types (Review)

- ❖ **Immediate (\$):** Constant integer data
 - e.g., \$1
- ❖ **Register (%):** The name of any of the 16 general-purpose integer registers
 - e.g., %rax
- ❖ **Memory (()):** A specified address that is usually dereferenced
 - e.g., (%rax)

Operand Type: Immediate

- ❖ Conceptually similar to literals in code
- ❖ Can be specified in decimal or hex
 - *e.g.*, \$0xFF
 - Decimals can be specified as positive (*e.g.*, \$351) or negative (*e.g.*, \$-1)
- ❖ **Cannot be used as the destination operand in a binary instruction!**
 - Not a valid location

Operand Type: Register (Review)

- ❖ A register is a location in the CPU that stores a small amount of data (a word size) that can be accessed very quickly
 - A fixed number of them (only 16 general purpose in x86-64)
 - Registers have *names*, not *addresses*
 - Registers are at the heart of assembly programming

Register Widths (Review)

- ❖ x86-64 general purpose integer registers (and sub-registers):

<u>%rax</u>	<u>%eax</u>	%ax	%al	%r8	%r8d	%r8w	%r8b
<u>%rbx</u>	<u>%ebx</u>	%bx	%bl	%r9	%r9d	%r9w	%r9b
<u>%rcx</u>	<u>%ecx</u>	%cx	%cl	%r10	%r10d	%r10w	%r10b
<u>%rdx</u>	<u>%edx</u>	%dx	%dl	%r11	%r11d	%r11w	%r11b
<u>%rsi</u>	<u>%esi</u>	%si	%sil	%r12	%r12d	%r12w	%r12b
<u>%rdi</u>	<u>%edi</u>	%di	%dil	%r13	%r13d	%r13w	%r13b
<u>%rsp</u>	<u>%esp</u>	%sp	%spl	%r14	%r14d	%r14w	%r14b
<u>%rbp</u>	<u>%ebp</u>	%bp	%bpl	%r15	%r15d	%r15w	%r15b
8 bytes	4 bytes	2 bytes	1 byte	8 bytes	4 bytes	2 bytes	1 byte

- Names for smaller divisions refer to least significant bytes
 - When used as a destination, leaves upper bytes untouched EXCEPT for 32-bit register destinations, which zero out the upper 4 bytes
- Make sure to use the correct register name for desired data width!

Operand Type: Memory

- ❖ A way to specify an address in memory
 - *e.g.*, (`%rax`), assuming that an address is currently stored in `%rax`
 - By default, instructions will dereference the specified address
 - Size of data is inferred from instruction size
- ❖ Memory is large, but extremely slow to access
 - 2^{64} -byte address space in a 64-bit machine
 - 2-3 orders of magnitude slower than register
- ❖ You cannot have both operands be Memory type
 - Design decision for performance and encoding reasons

Operand Combination Examples

- ❖ Actual effect will depend on specifics of the two-operand instruction used
 - Immediate is like a literal, Register is like a variable, Memory is like a pointer

Imm→Reg: `addq $-42, %rax` *# like var_rax += -42;*


Reg→Reg: `subl %eax, %edx` *# like var_edx -= var_eax;*

Mem→Reg: `xorq (%rbx), %rax` *# like var_rax ^= *ptr_rbx;*

Imm→Mem: `movq $0x3, (%rbx)` *# like *ptr_rbx = 3;*

Reg→Mem: `orw %ax, (%rbx)` *# like *ptr_rbx |= var_ax;*

Polling Questions (2/2)

- ❖ 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>`):

exclusive or (^) → `xorw $-1, %ax`

logical operation

source: immediate, destination: register

2 bytes ("word")

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

$$\begin{array}{r} 0x\ 07\ 08 \\ \wedge\ 0x\ FF\ FF \\ \hline 0x\ F8\ F7 \end{array} \Rightarrow \%rax: \boxed{0x\ 01\ 02\ 03\ 04\ 05\ 06\ F8\ F7}$$

Lecture Outline (4/4)

- ❖ Instruction Set Architectures (ISAs)
- ❖ x86-64 Syntax and Instructions
- ❖ x86-64 Operands
- ❖ **First Assembly Examples**

Example: Basic Arithmetic

```
long arith(long x, long y) {  
    return 3*(x+y);  
}
```

arbitrary! (for now...)

Variable	Register
x	%rdi
y	%rsi
return value	%rax

```
long arith_mod(long x, long y) {  
    long t1 = x + y;  
    long t2 = t1 * 3;  
    return t2; temporaries  
}
```

```
y += x;  
y *= 3;  
long r = y;  
return r;
```

*must return
in %rax*

instr src , dst

```
arith:  
    addq    %rdi, %rsi  
    imulq   $3, %rsi  
    movq    %rsi, %rax  
    ret    # return
```

Example: Using Memory (1/2)

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

Compiler Explorer:

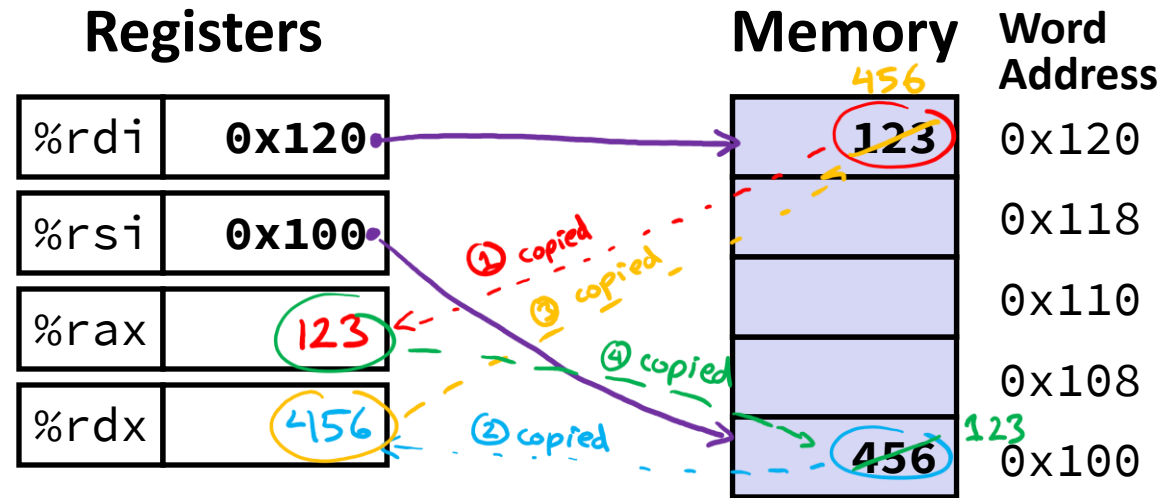
<https://godbolt.org/z/vjzxr5xb8>

memory operands *register operands*

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

<u>Register</u>		<u>Variable</u>
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Example: Using Memory (2/2)



swap:

```
① movq    (%rdi), %rax    # t0 = *xp
② movq    (%rsi), %rdx    # t1 = *yp
③ movq    %rdx, (%rdi)    # *xp = t1
④ movq    %rax, (%rsi)    # *yp = t0
ret
```

Homework Setup Question

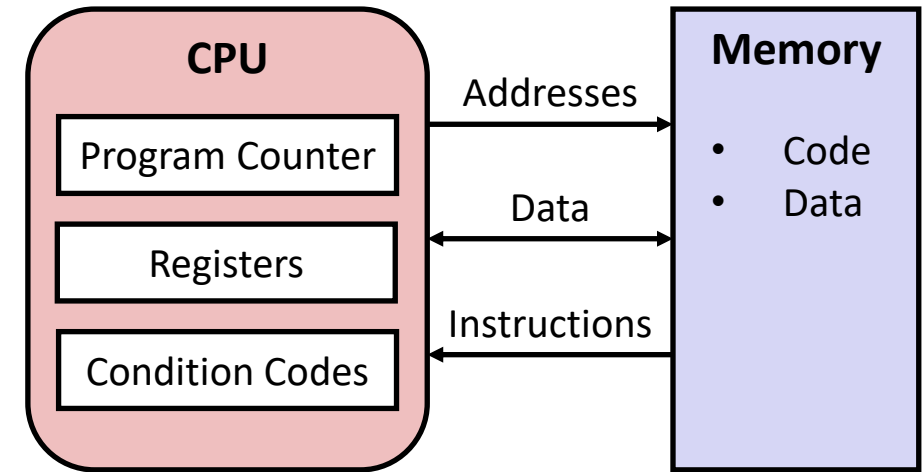
- ❖ Do the following operand types have an implied size?
 - An immediate operand is a literal/constant (e.g., \$3)
No, could be 0x03, 0x0003, 0x00000003, etc.
 - A register operand is the value stored in a register (e.g., %rdx)
Yes, look-up in register table (%rdx is 8 bytes wide)
 - A memory operand represents an address in memory (e.g., (%rsi))
*No, address just gives us the starting point of the data
(the address itself is a word size, though)*

Summary (1/2)

❖ Assembly programmer-visible state:

❖ x86-64 is a *complex instruction set computing* (CISC) architecture

- x86-64 integer instruction common forms: **instr op** and **instr src, dst**
 - Fixed width specified by size suffix: b (1 byte), w (2 bytes), l (4 bytes), or q (8 bytes)
- Instruction types:
 - *Data transfer* (e.g., **movq** (%rsi), %rdx)
 - *Arithmetic* (e.g., **imulq** \$3, %rsi)
 - *Control Flow* (e.g., **ret**)



Summary (2/2)

- ❖ x86-64 is a *complex instruction set computing* (CISC) architecture
 - x86-64 integer instruction common forms: **instr op** and **instr src, dst**
 - Fixed width specified by size suffix: b (1 byte), w (2 bytes), l (4 bytes), or q (8 bytes)
 - Operand types:
 - Immediate (**\$**) is a literal (e.g., `imulq $3, %rsi`)
 - Register (**%**) is a general-purpose integer register or sub-register (e.g., `movq (%rsi), %rdx`)
 - Memory (**()**) is a way to express an address (e.g., `movq (%rsi), %rdx`)

<u>%rax</u>	<u>%eax</u>	<u>%ax</u>	<u>%al</u>	<u>%r8</u>	<u>%r8d</u>	<u>%r8w</u>	<u>%r8b</u>
<u>%rbx</u>	<u>%ebx</u>	<u>%bx</u>	<u>%bl</u>	<u>%r9</u>	<u>%r9d</u>	<u>%r9w</u>	<u>%r9b</u>
<u>%rcx</u>	<u>%ecx</u>	<u>%cx</u>	<u>%cl</u>	<u>%r10</u>	<u>%r10d</u>	<u>%r10w</u>	<u>%r10b</u>
<u>%rdx</u>	<u>%edx</u>	<u>%dx</u>	<u>%dl</u>	<u>%r11</u>	<u>%r11d</u>	<u>%r11w</u>	<u>%r11b</u>
<u>%rsi</u>	<u>%esi</u>	<u>%si</u>	<u>%sil</u>	<u>%r12</u>	<u>%r12d</u>	<u>%r12w</u>	<u>%r12b</u>
<u>%rdi</u>	<u>%edi</u>	<u>%di</u>	<u>%dil</u>	<u>%r13</u>	<u>%r13d</u>	<u>%r13w</u>	<u>%r13b</u>
<u>%rsp</u>	<u>%esp</u>	<u>%sp</u>	<u>%spl</u>	<u>%r14</u>	<u>%r14d</u>	<u>%r14w</u>	<u>%r14b</u>
<u>%rbp</u>	<u>%ebp</u>	<u>%bp</u>	<u>%bpl</u>	<u>%r15</u>	<u>%r15d</u>	<u>%r15w</u>	<u>%r15b</u>
8 bytes	4 bytes	2 bytes	1 byte	8 bytes	4 bytes	2 bytes	1 byte