

# x86-64 Programming I

CSE 351 Autumn 2024

**Instructor:**

Ruth Anderson

**Teaching Assistants:**

Alexandra Michael

Connie Chen

Chloe Fong

Chendur Jayavelu

Joshua Tan

Nikolas McNamee

Nahush Shrivatsa

Naama Amiel

Neela Kausik

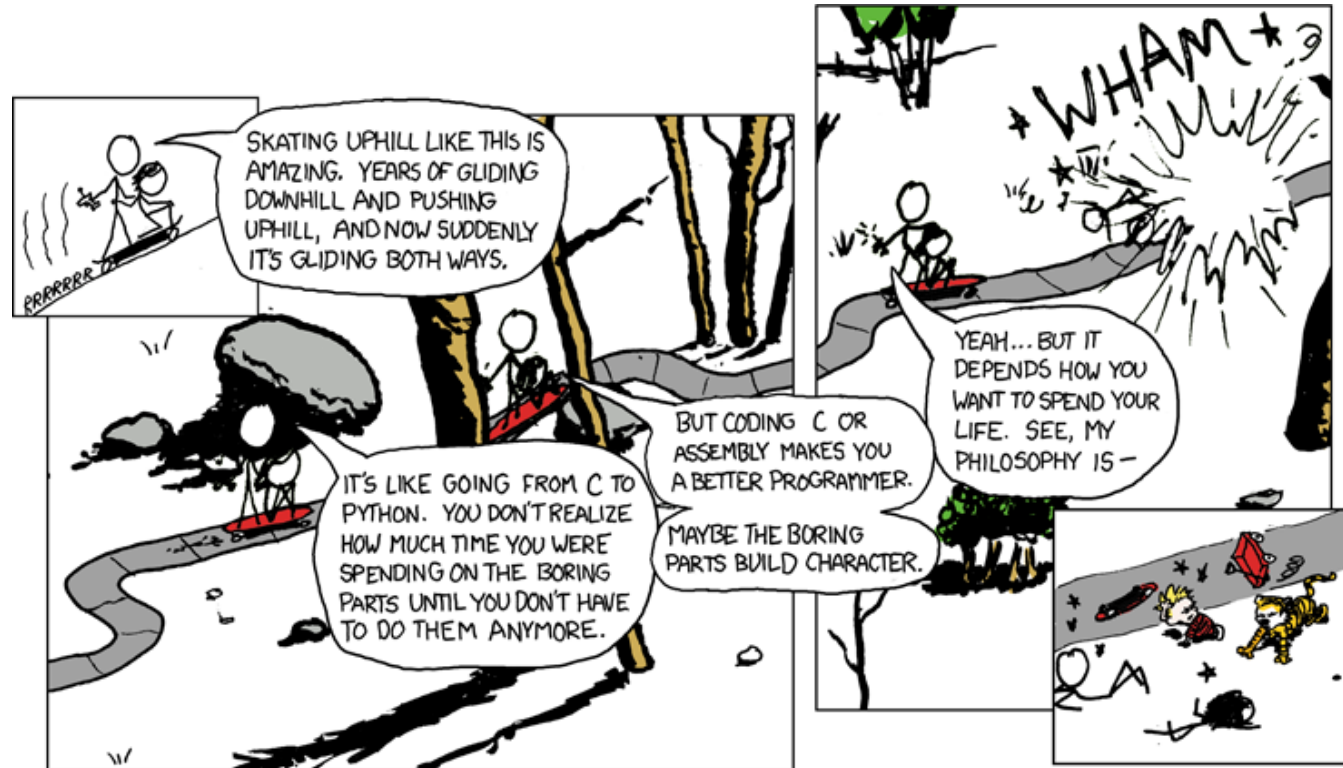
Renee Ruan

Rubee Zhao

Samantha Dreussi

Sean Siddens

Waleed Yagoub



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

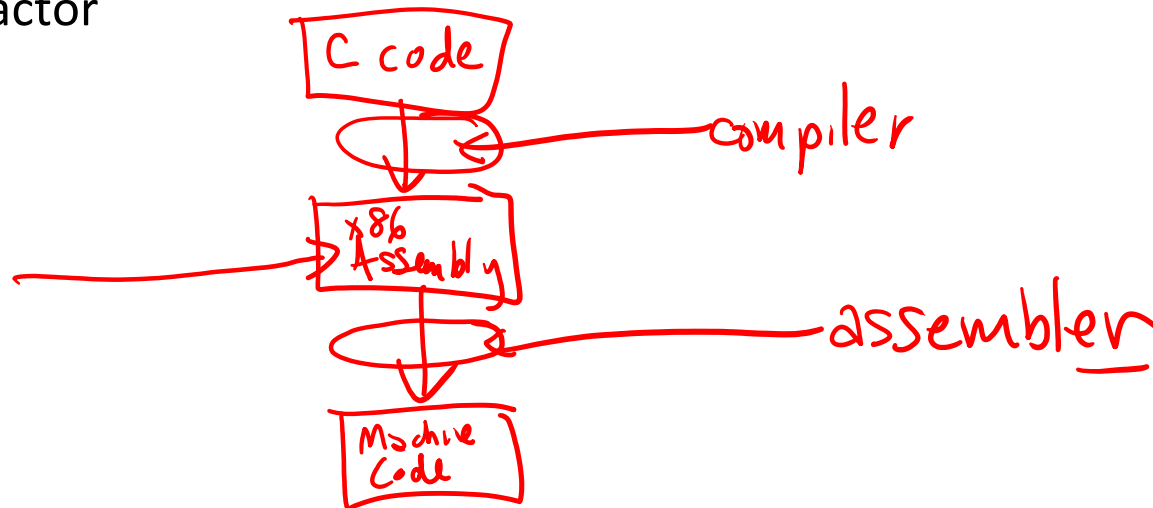
# Relevant Course Information

- ❖ HW5 due tonight, Wednesday (10/09) @ 11:59 pm
- ❖ Lab 1b, due Monday (10/14) @ 11:59pm
  - No major programming restrictions, but should **avoid magic numbers by using C macros (#define)**
  - For debugging, can use provided utility functions `print_binary_short()` and `print_binary_long()`
  - Pay attention to the output of `aisle_test` and `store_test` – failed tests will show you actual vs. expected
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
- ❖ HW6 due Friday (10/11) @ 11:59 pm

# Reading Review

## ❖ Terminology:

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



# Review Questions

operation    src, dst

- ❖ Assume that the register %rdx currently holds the value 0x01 02 03 04 05 06 07 08

07

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

subq <sup>src</sup> \$1, <sup>dst</sup> %rdx ←

- Operation type: Arithmetic & Logical
- Operand types: src: immediate    dst: register
- Operation width: 8 bytes, 64 bits
- (extra) Result in %rdx:

0x 01 02 03 04 05 06 07 07

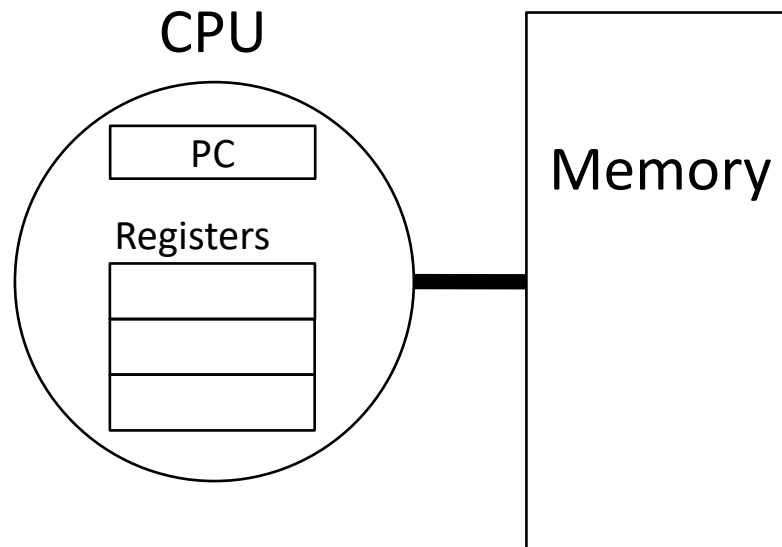
# Definitions

- ❖ **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
  
- ❖ **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469

# Instruction Set Architectures (Review)

## ❖ 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 (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, less power-hungry hardware
- Let software do complicated operations by composing simpler ones
- ARM, RISC-V



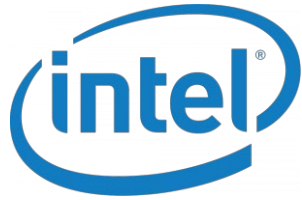
# CISC Example

- ❖ *Complex Instruction Set Computing (CISC):*  
Add more and more elaborate and specialized instructions as needed

## Example: ADDSUBPS

“Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.”

# Mainstream ISAs



## x86

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

PCs and older Macs  
(Core i3, i5, i7, M)  
x86-64 Instruction Set



## ARM

<b>Designer</b>	Arm Holdings
<b>Bits</b>	32-bit, 64-bit
<b>Introduced</b>	1985
<b>Design</b>	RISC
<b>Type</b>	Register-Register
<b>Encoding</b>	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions; ARMv7 <i>user-space</i> compatibility. <sup>[1]</sup>
<b>Branching</b>	Condition code, compare and branch
<b>Endianness</b>	Bi (little as default)

Mobile devices, M1/M2 Macs  
ARM Instruction Set



## RISC-V

<b>Designer</b>	University of California, Berkeley
<b>Bits</b>	32 · 64 · 128
<b>Introduced</b>	2010
<b>Design</b>	RISC
<b>Type</b>	Load-store
<b>Encoding</b>	Variable
<b>Endianness</b>	Little <sup>[1][3]</sup>

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

# Architecture Sits at the Hardware Interface

## Source code

Different applications  
or algorithms

## Compiler

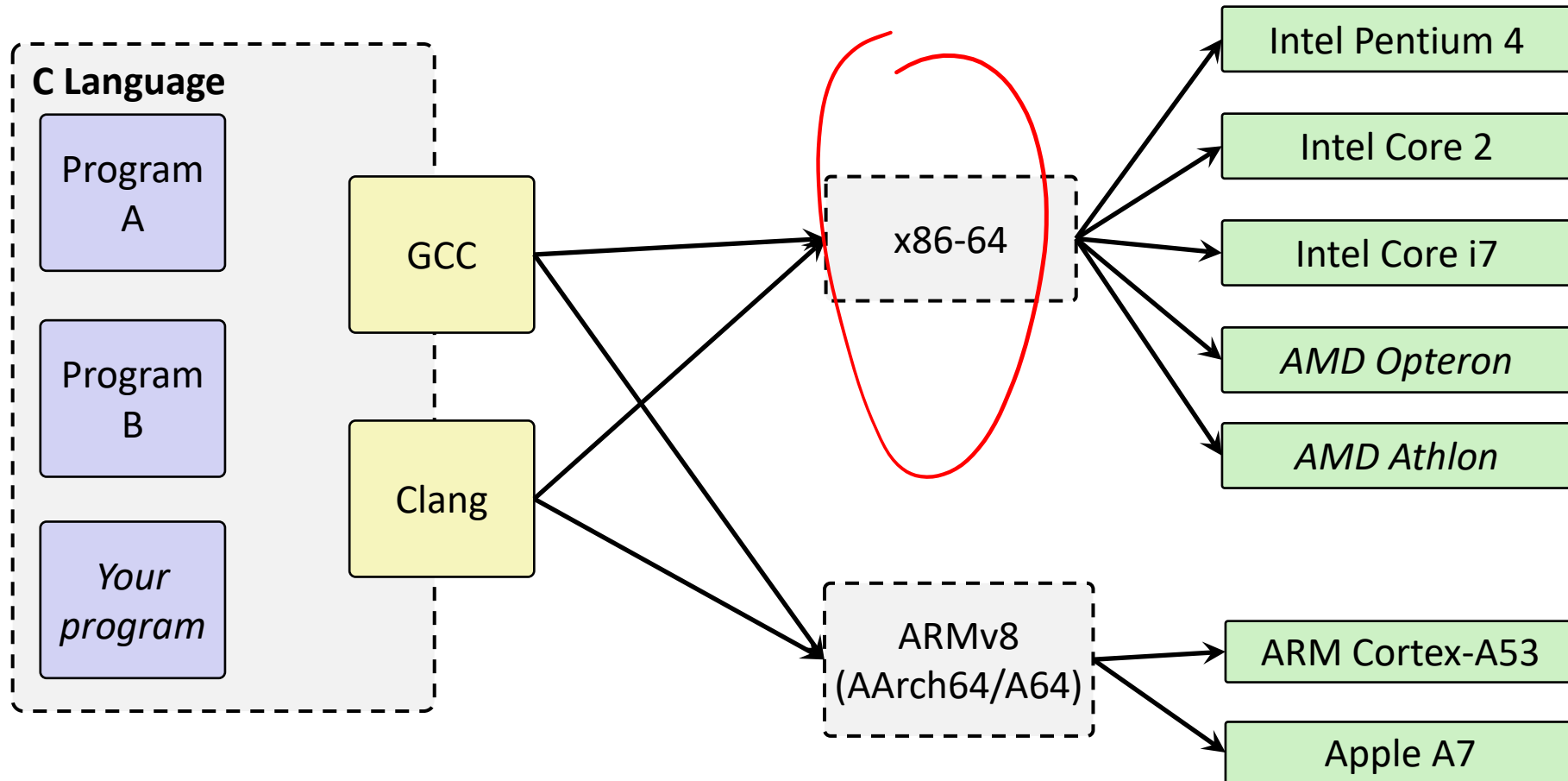
Perform optimizations,  
generate instructions

## Architecture

Instruction set

## Hardware

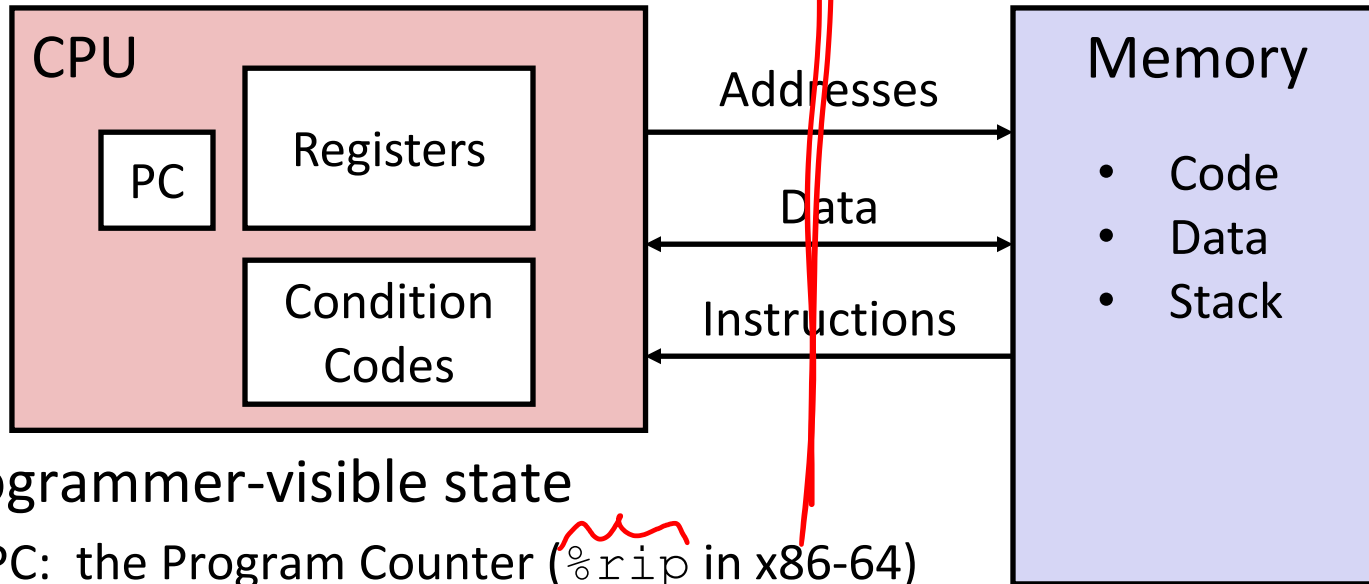
Different  
implementations



# Writing Assembly Code? In 2024???

- ❖ 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)

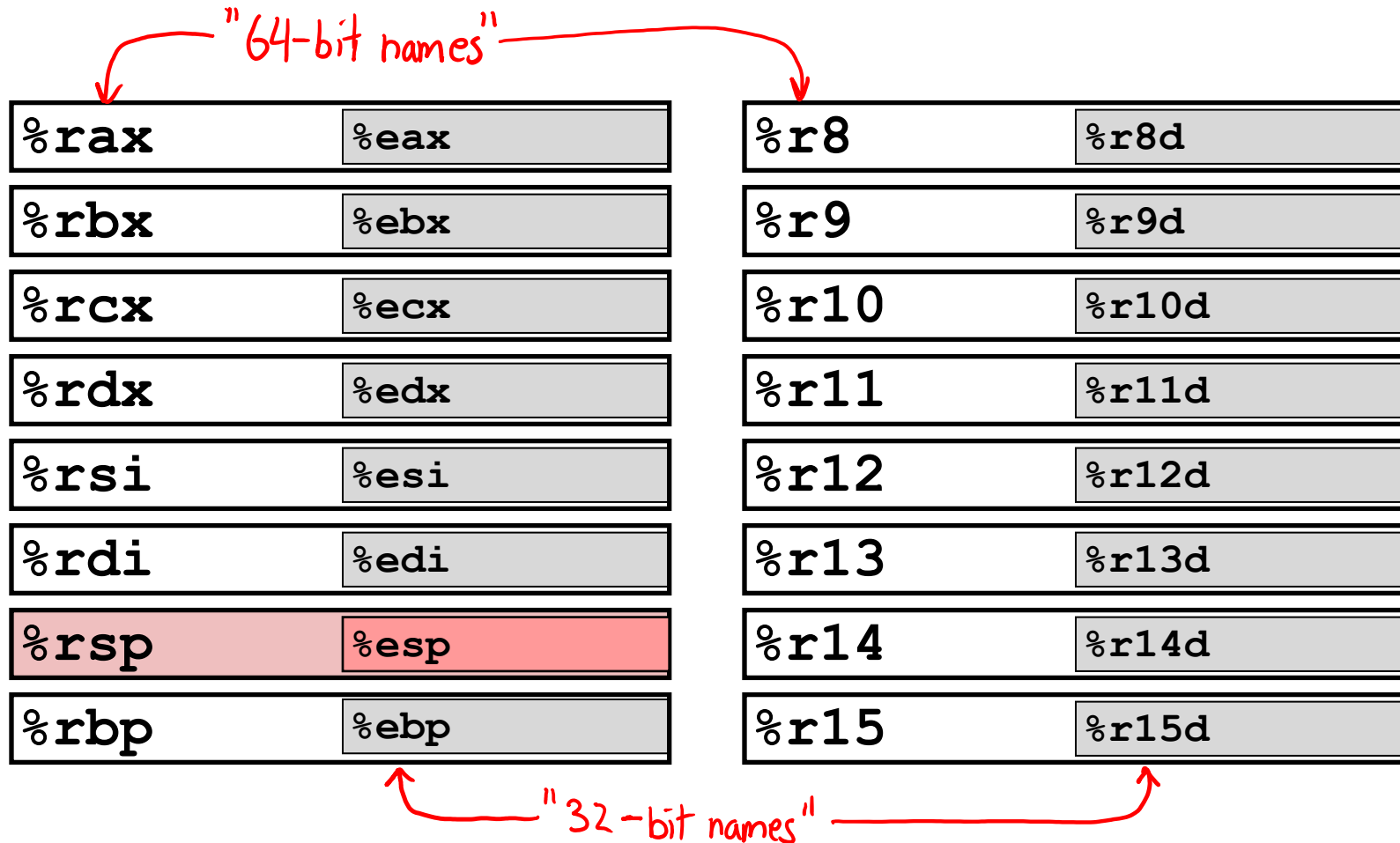
# x86-64 Assembly “Data Types”

- ❖ 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, ...)
- } Not covered  
in 351
- ❖ 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!!
- operation src, dst

# What is a Register? (Review)

- ❖ 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 only 16 of them...*

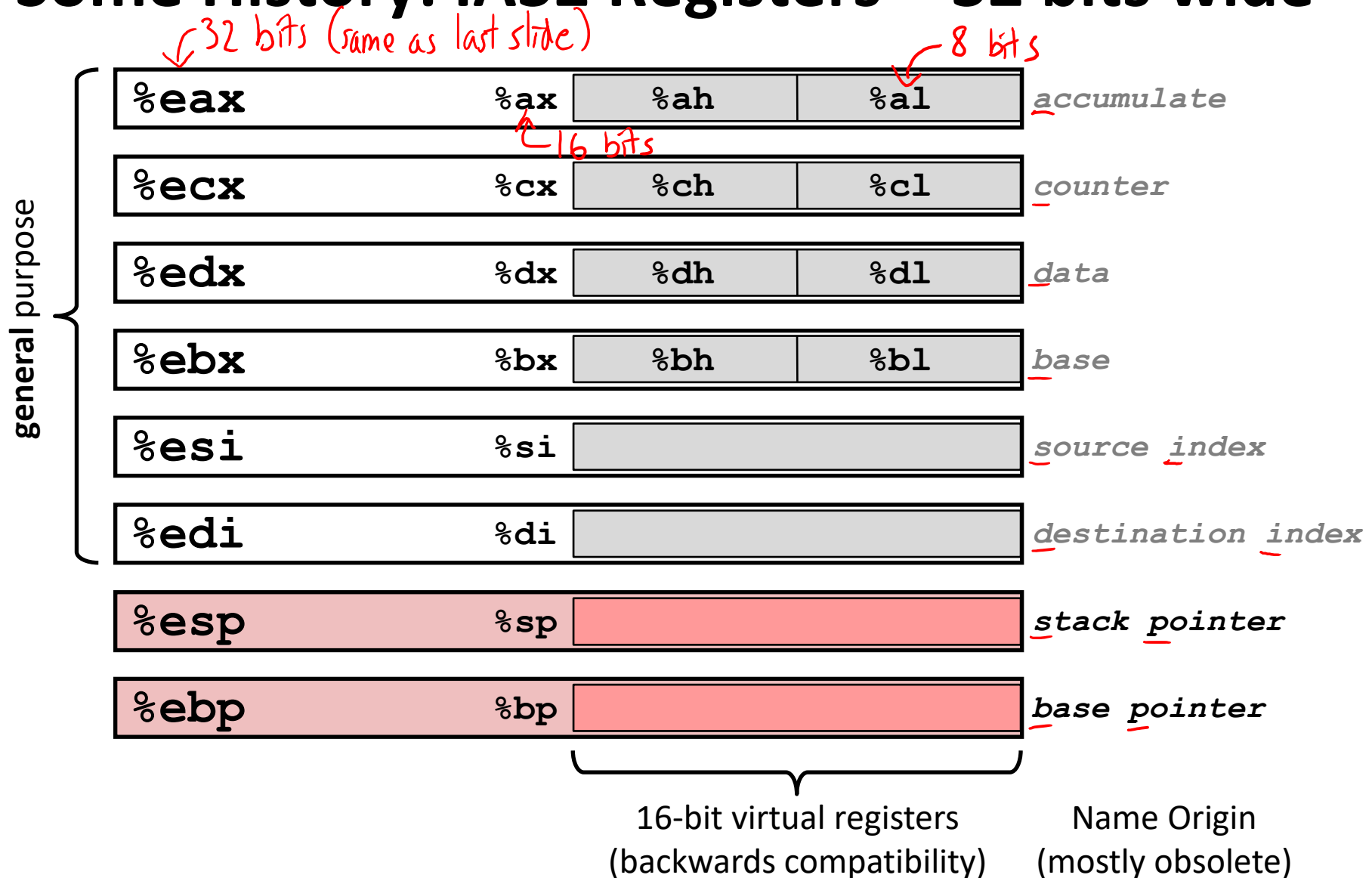
# x86-64 Integer Registers – 64 bits wide



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



# Some History: IA32 Registers – 32 bits wide



# Memory

## ❖ Addresses

- `0x7FFFD024C3DC`

## ❖ Big

- `~ 8 GiB`

## ❖ Slow

- `~50-100 ns`

## ❖ Dynamic

- Can “grow” as needed while program runs

# vs. Registers

*on the chip*

## vs. Names

`%rdi`

## vs. Small

(16 x 8 B) = 128 B

## vs. Fast

sub-nanosecond timescale

## vs. Static

fixed number in hardware

# Three Basic Kinds of Instructions (Review)

## 1) Transfer data between memory and register

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

**Remember:** Memory is indexed just like an array of bytes!

## 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

# Instruction Sizes and Operands (Review)

## ❖ Size specifiers

- $b$  = 1-byte “byte”,  $w$  = 2-byte “word”,  
 $\ell$  = 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

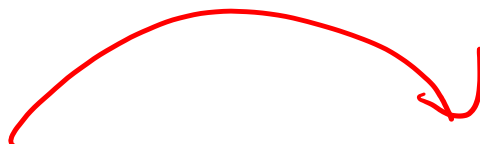
## ❖ Operand types

- **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: `movl` source, destination
  - Really more of a “copy” than a “move”
  - Like all instructions, missing letter (l) is the size specifier
  - Lots of these in typical code

# Operand Combinations

x86      C  
 Imm  $\leftrightarrow$  Constant  
 Reg  $\leftrightarrow$  Variable  
 Mem  $\leftrightarrow$  dereferencing  
**C Analog** a pointer

	Source	Dest	Src, Dest	
movq	Imm	Reg	movq \$0x4, %rax	var_a = 0x4;
		Mem	movq \$-147, (%rax)	*p_a = -147;
	Reg	Reg	movq %rax, %rdx	var_d = var_a;
		Mem	movq %rax, (%rdx)	*p_d = var_a;
	Mem	Reg	movq (%rax), %rdx	var_d = *p_a;

❖ *Cannot do memory-memory transfer with a single instruction*

■ How would you do it?

① Mem  $\rightarrow$  Reg

② Reg  $\rightarrow$  Mem

movq (%rax), %rdx

movq %rdx, (%rbx)

# Some Arithmetic Operations

*src + dst  
cannot both  
be mem*

## ❖ Binary (two-operand) Instructions: *Imm, Reg, or Mem*

**Maximum of one  
memory operand**

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

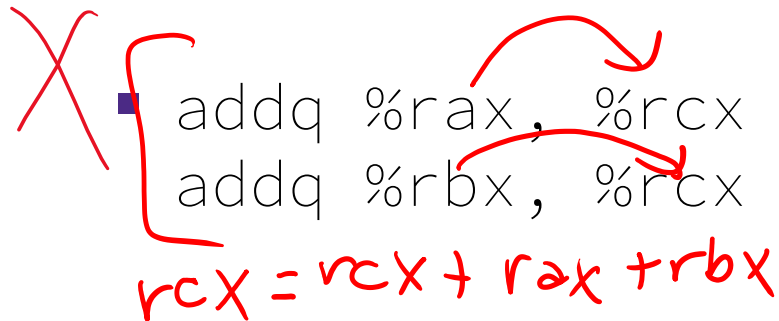
Format	Computation	
<code>addq src, dst</code>	$dst = dst + src$	(dst <u>+=</u> src)
<code>subq src, dst</code>	$dst = \underline{dst - src}$	
<code>imulq src, dst</code>	$dst = dst * src$	signed mult
<code>sarq src, dst</code>	$dst = dst \gg src$	Arithmetic
<code>shrq src, dst</code>	$dst = dst \gg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

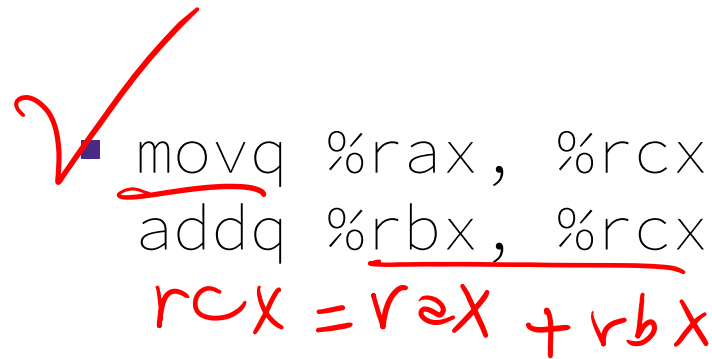
*operation* → *operand size specifier (b, w, l, q)*

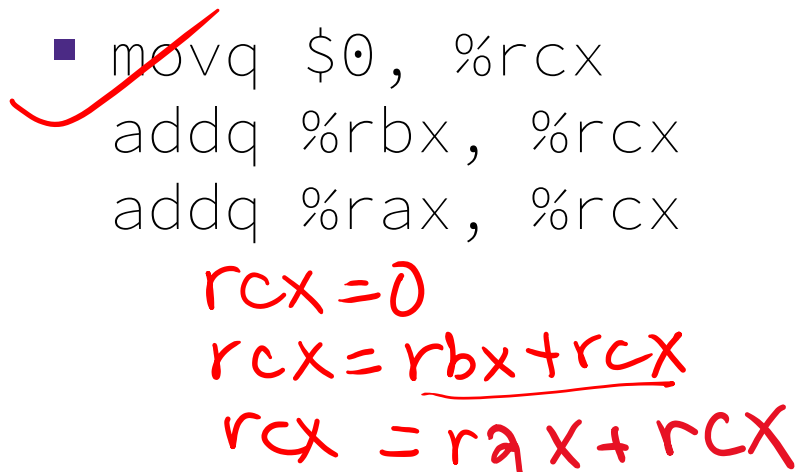


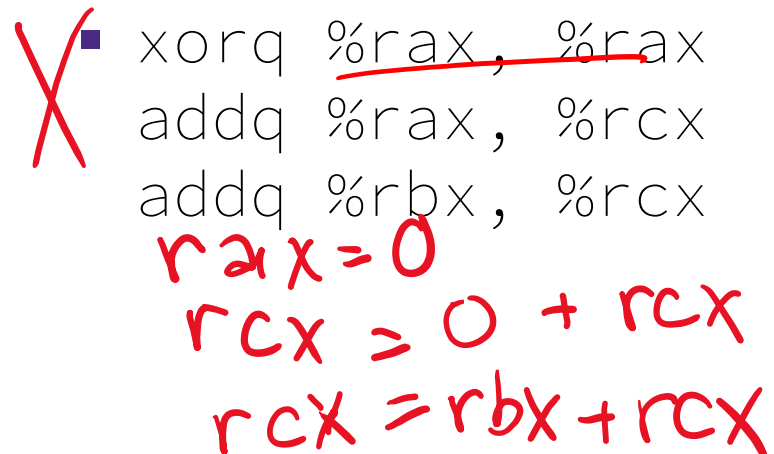
# Practice Question

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

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

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

✓   
■ `movq $0, %rcx`  
`addq %rbx, %rcx`  
`addq %rax, %rcx`  
 $rcx = 0$   
 $rcx = \underline{rbx + rcx}$   
 $rcx = rax + rcx$

X   
■ `xorq %rax, %rax`  
`addq %rax, %rcx`  
`addq %rbx, %rcx`  
 $rax = 0$   
 $rcx = 0 + rcx$   
 $rcx = rbx + rcx$

# Arithmetic Example

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

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rax	return value

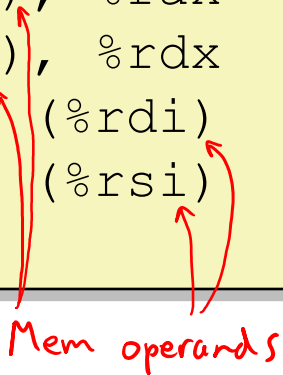
```
y += x;
y *= 3;
long r = y;
return r;
```

```
simple_arith:
    addq    %rdi, %rsi
    imulq   $3, %rsi
    movq    %rsi, %rax
    ret
```

# Example of Basic Addressing Modes

```
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
```



Mem operands

Compiler Explorer:

<https://godbolt.org/z/zc4Pcq>

# 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