

# x86-64 Programming I

CSE 351 Autumn 2021

## Instructor:

Justin Hsia

## Teaching Assistants:

Allie Pflieger

Anirudh Kumar

Assaf Vayner

Atharva Deodhar

Celeste Zeng

Dominick Ta

Francesca Wang

Hamsa Shankar

Isabella Nguyen

Joy Dang

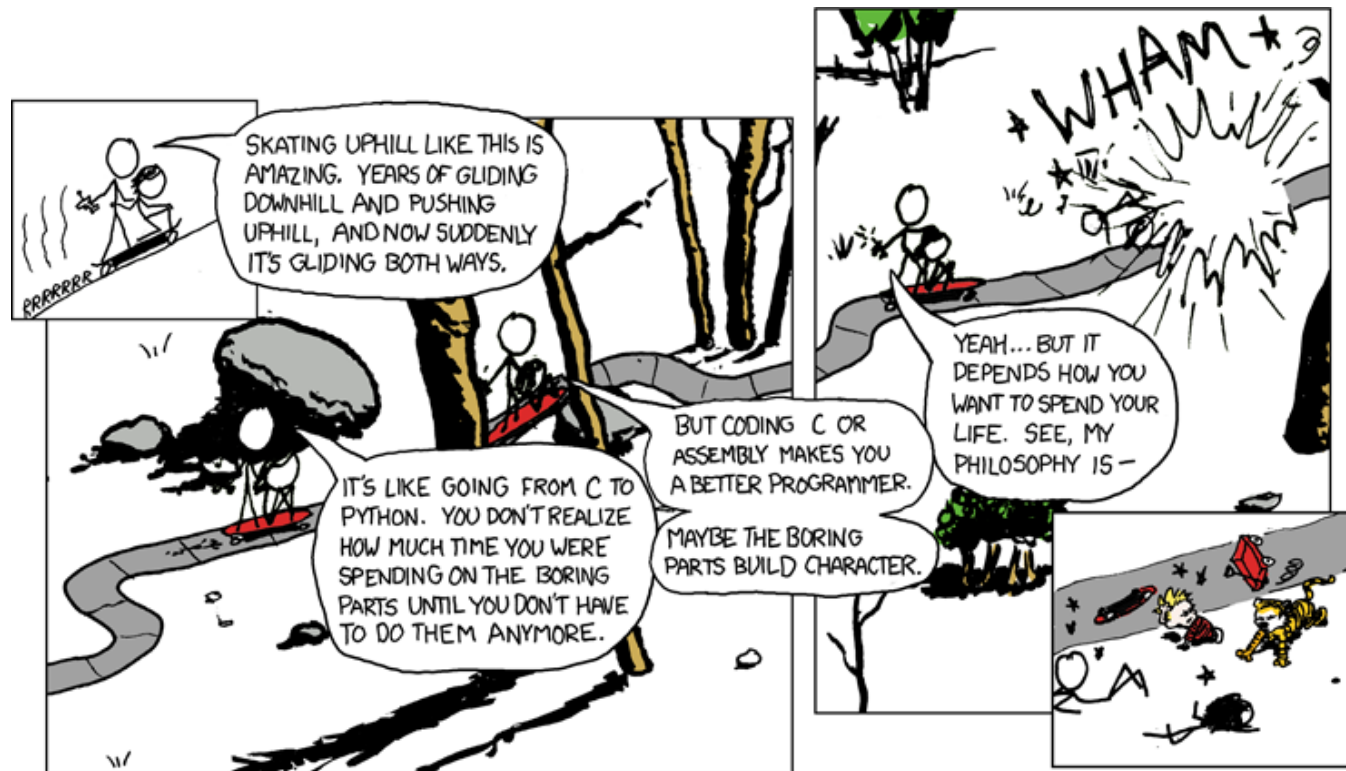
Julia Wang

Maggie Jiang

Monty Nitschke

Morel Fotsing

Sanjana Chintalapati



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

# Relevant Course Information

- ❖ hw7 due Monday, hw8 due Wednesday
  
- ❖ Lab 1b due Monday (10/18) at 11:59 pm
  - 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
  - You have *late day tokens* available

# 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
  
- ❖ Questions from the Reading?

# Review Questions

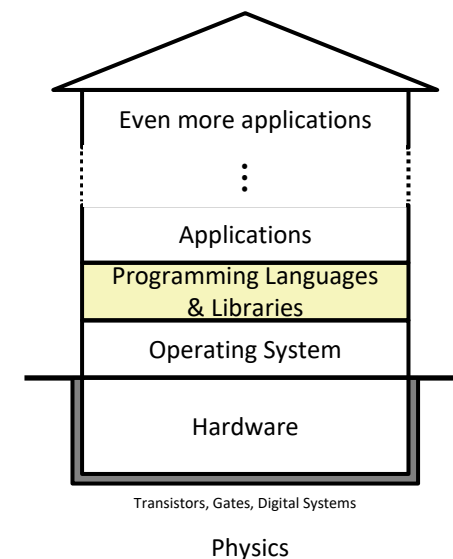
- ❖ Assume that the register `%rax` currently holds the value `0x 01 02 03 04 05 06 07 08`
- ❖ Answer the questions on Ed Lessons about the following instruction (`<instr> <src> <dst>`):

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

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

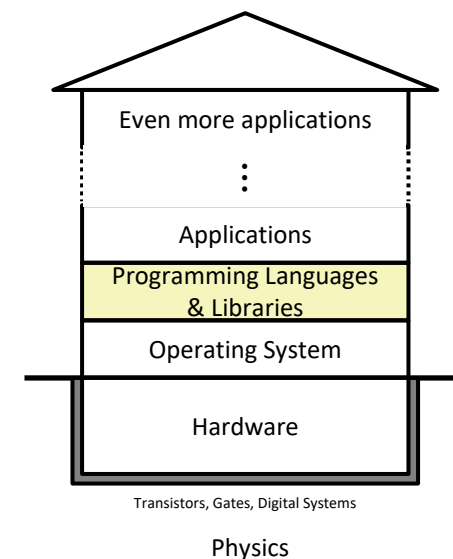
# The Hardware/Software Interface

- ❖ Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, Arrays, Structs
- ❖ Topic Group 2: **Programs**
  - **x86-64 Assembly**, Procedures, Stacks, Executables
- ❖ Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



# The Hardware/Software Interface

- ❖ Topic Group 2: **Programs**
  - **x86-64 Assembly**, Procedures, Stacks, Executables



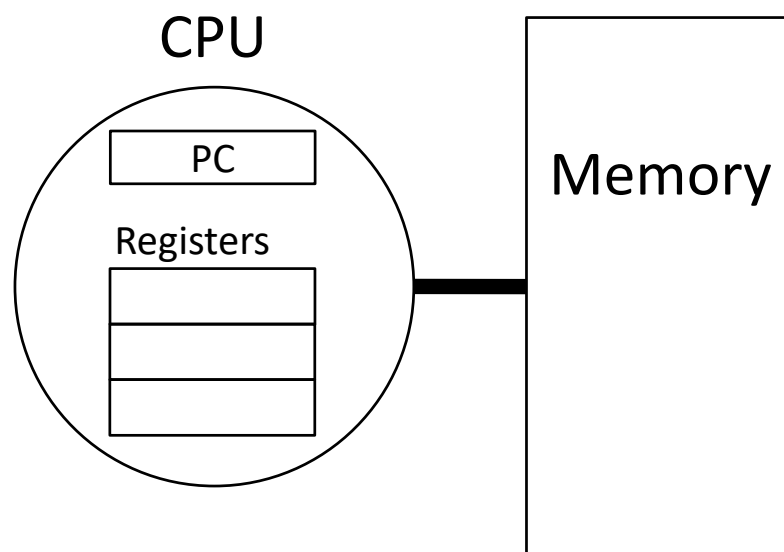
- ❖ How are programs created and executed on a CPU?
  - How does your source code become something that your computer understands?
  - How does the CPU organize and manipulate local data?

# 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?

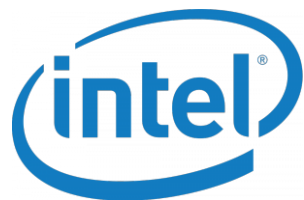
## ❖ Memory

- 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 hardware
  - Let software do the complicated operations by composing simpler ones

# 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

Macbooks & PCs  
(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 user-space compatibility. <sup>[1]</sup>

**Branching** Condition code, compare and branch

**Endianness** Bi (little as default)

Smartphone-like devices  
(iPhone, iPad, Raspberry Pi)  
[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

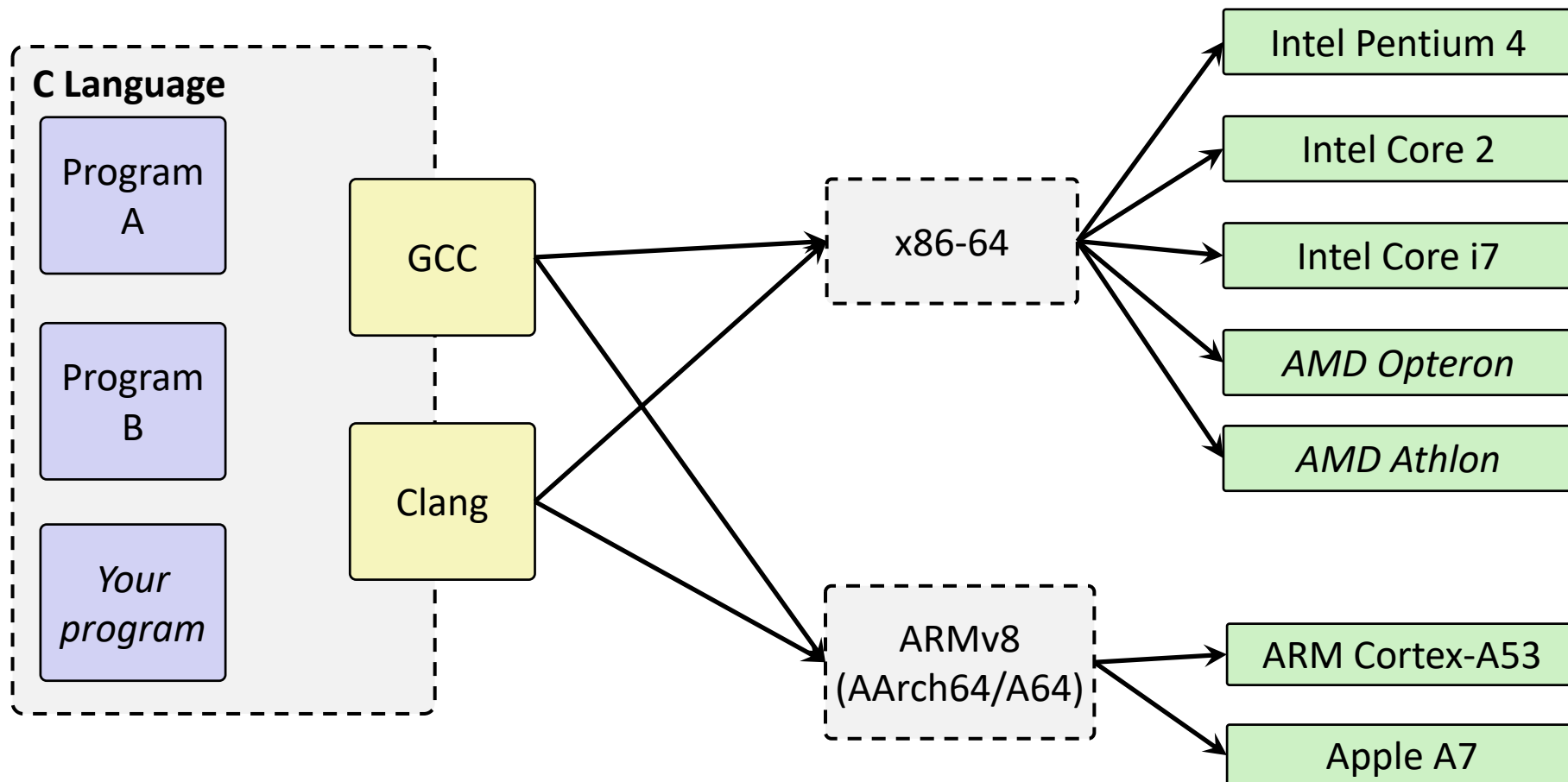
Perform optimizations, generate instructions

## Architecture

Instruction set

## Hardware

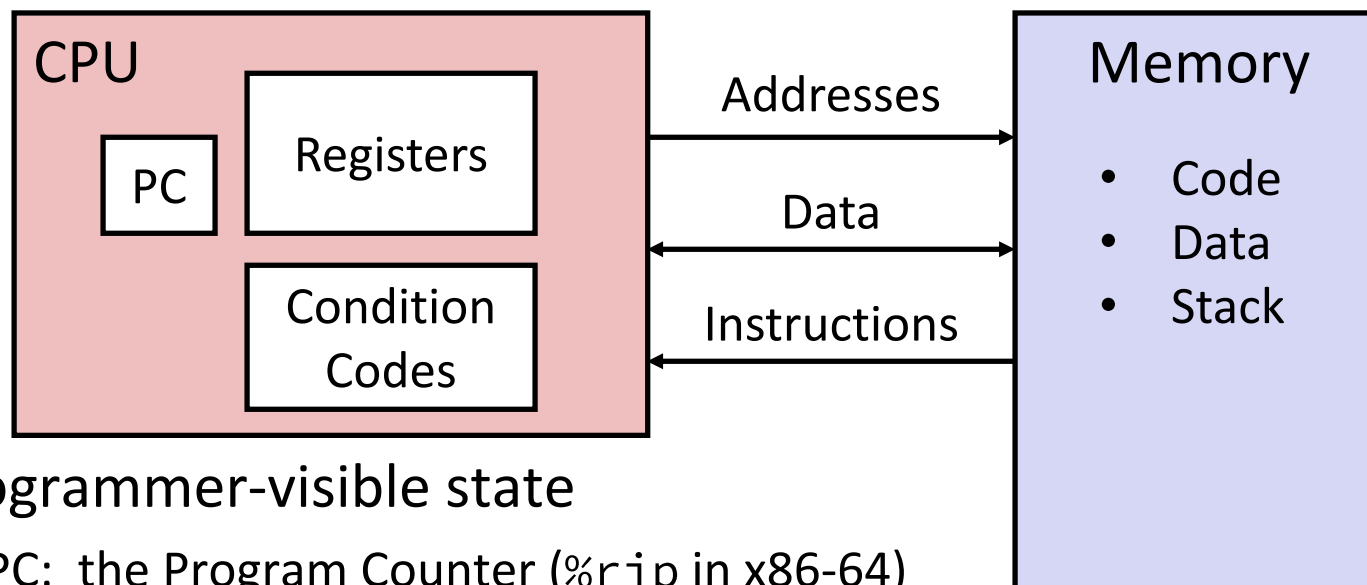
Different implementations



# Writing Assembly Code? In 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

# Instruction Types (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”,  
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 general-purpose integer registers (%)
- **Memory:** Consecutive bytes of memory at a computed address (( ))

# 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

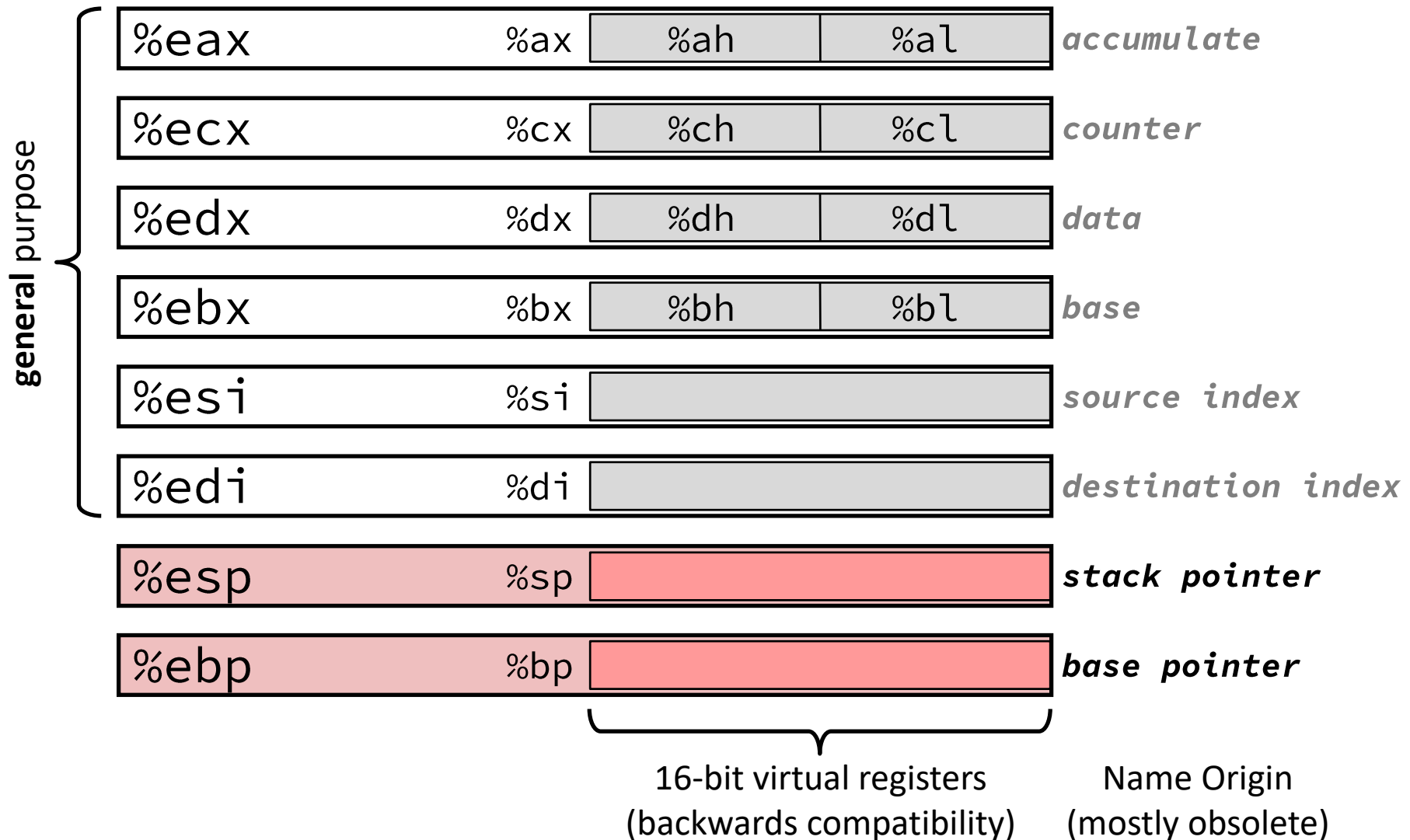
# x86-64 Integer Registers – 64 bits wide

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- 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

- vs. Names
  - %rdi
- vs. Small
  - (16 x 8 B) = 128 B
- vs. Fast
  - sub-nanosecond timescale
- vs. Static
  - fixed number in hardware

# 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

	Source	Dest	Src, Dest	C Analog
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?

# 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

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

↑ operand size specifier



# Practice Question

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

- `addq %rax, %rcx`  
`addq %rbx, %rcx`

- `movq %rax, %rcx`  
`addq %rbx, %rcx`

- `movq $0, %rcx`  
`addq %rbx, %rcx`  
`addq %rax, %rcx`

- `xorq %rax, %rax`  
`addq %rax, %rcx`  
`addq %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
```

# 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