x86-64 Programming I

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

https://tapas.io/episode/753918

Announcements, Reminders

- Lab1a and HW5 due tonight! (8 Apr)
  - HW6 due 10 Apr; HW7 due 15 Apr
- Lab 1b due 15 Apr by 11:59 PM:
  - No major programming restrictions, but 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 your actual vs. expected
  - Can use (up to two) late days to turn in by **17 Apr at 11:59 PM**
- Reminder: 1-on-1 request form on course website!
- Synthesis questions: our goal is to assess learning, not to be pedantic

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 `%rdx` currently holds the value:

`0x 01 02 03 04 05 06 07 08`

- Answer the questions about the following instruction (<instr> <src> <dst>):

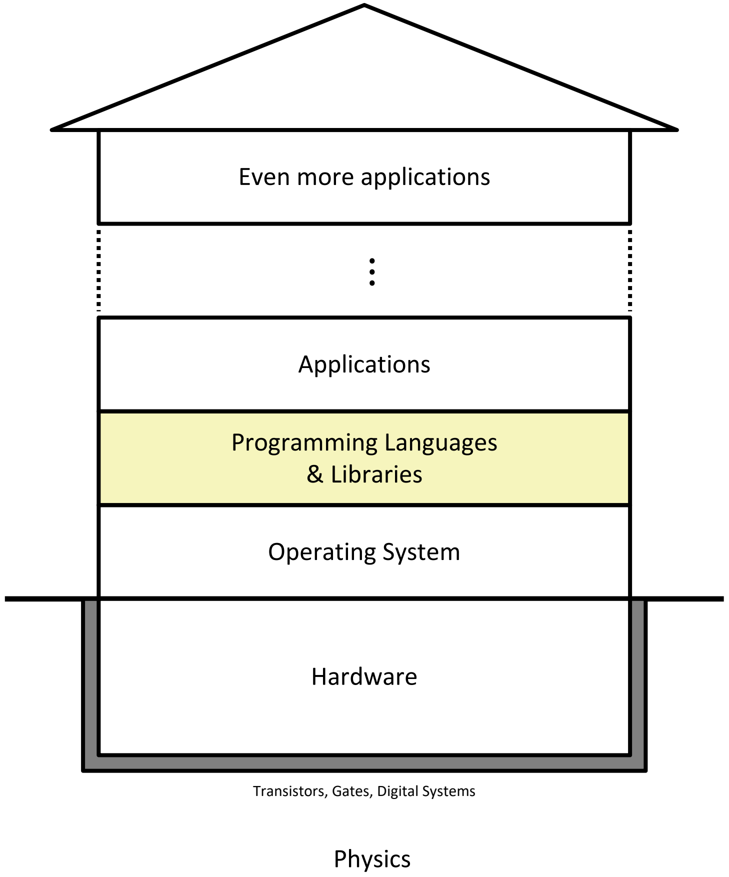
`subq $1, %rdx`

  - Operation type: Arithmetic/Logical
  - Operand types: $1 → immediate %rdx → register
  - Operation width: subq 8 Bytes
  - (extra) Result in `%rdx`:

```
  0x 01 02 03 04 05 06 07 08
- 0x 00 00 00 00 00 00 00 01
----------------------------
  0x 01 02 03 04 05 06 07 07
```

The Hardware/Software Interface

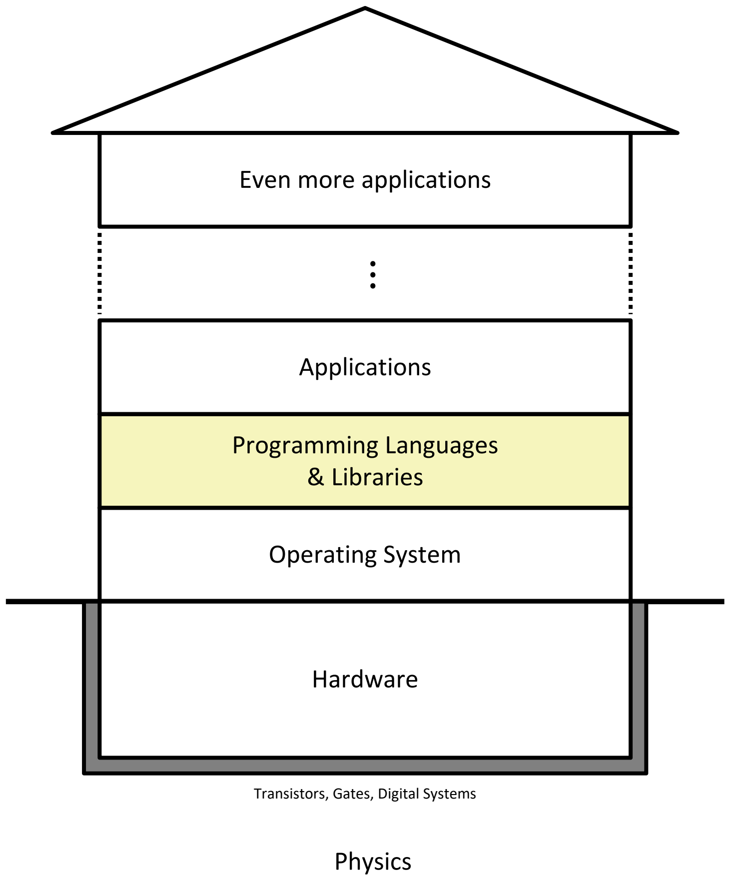
- Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, Arrays, Structs
- Topic Group 2: **Programs**
  - **x86-64 Assembly**, Procedures, Stacks, Executables
- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



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?



But First: Definitions

Instruction Set Architecture

- **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:** Actual implementation of the architecture
  - CSE/EE 469

physical

ISAs are Born: IBM System/360 (1964)

G. M. Amdahl
G. A. Blaauw
F. P. Brooks, Jr.,

Architecture of the IBM System/360

Abstract: The architecture* of the newly announced IBM System/360 features four innovations:

1. An approach to storage which permits and exploits very large capacities, hierarchies of speeds, read-only storage for microprogram control, flexible storage protection, and simple program relocation.

2. An input/output system offering new degrees of concurrent operation, compatible channel operation, data rates approaching 5,000,000 characters/second, integrated design of hardware and software, a new low-cost, multiple-channel package sharing main-frame hardware, new provisions for device status information, and a standard channel interface between central processing unit and input/output devices.

3. A truly general-purpose machine organization offering new supervisory facilities, powerful logical processing operations, and a wide variety of data formats.

!!! This is new!

4. Strict upward and downward machine-language compatibility over a line of six models having a performance range factor of 50.

This paper discusses in detail the objectives of the design and the rationale for the main features of the architecture. Emphasis is given to the problems raised by the need for compatibility among central processing units of various size and by the conflicting demands of commercial, scientific, real-time, and logical information processing. A tabular summary of the architecture is shown in the Appendices.

ISAs are Born: IBM System/360 (1964)

Introduction

The design philosophies of the new general-purpose machine organization for the IBM System/360 are discussed in this paper.† In addition to showing the architecture* of the new family of data processing systems, we point out the various engineering problems encountered in attempts to make the system design compatible, at the program bit level, for large and small models. The compatibility was to extend not only to models of any size but also to their various applications—scientific, commercial, real-time, and so on.

*The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.

physical

† Additional details concerning the architecture, engineering design, programming, and application of the IBM System/360 will appear in a series of articles in the *IBM Systems Journal*.

This definition is perfect.

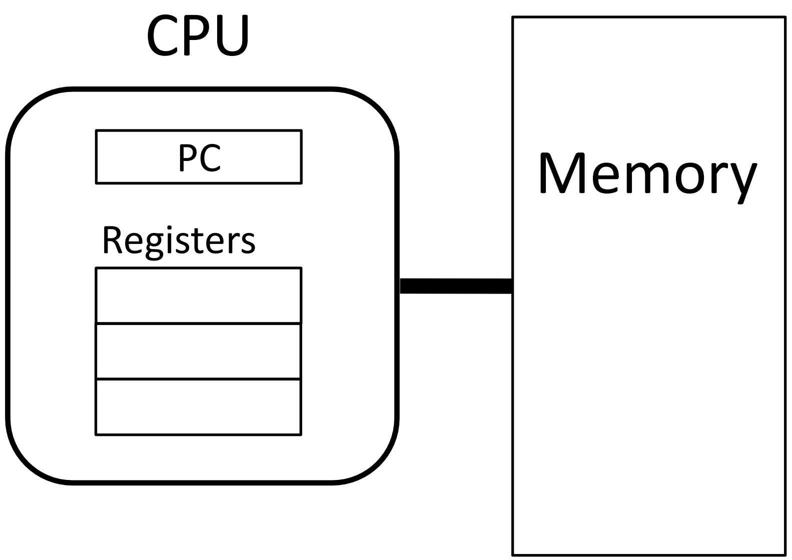
The section that follows describes the objectives of the new system design, i.e., that it serve as a base for new technologies and applications, that it be general-purpose, efficient, and strictly program compatible in all models. The remainder of the paper is devoted to the design problems faced, the alternatives considered, and the decisions made for data format, data and instruction codes, storage assignments, and input/output controls.

Design objectives

The new architecture builds upon but differs from the designs that have gradually evolved since 1950. The evolution of the computer had included, besides major technological improvements, several important systems concepts and developments:

Instruction Set Architectures (Review)

- The ISA defines:
  - The system's **state** *e.g.*, registers, memory, program counter (PC)
  - The **instructions** the CPU can execute
  - The **effect** that each of these instructions will have on the system state



General ISA Design Decisions

(More like what computer architects decided heavily about. ISAs are rarely created these days.)

- 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
  - Coined in 1980, but concept arguably existed before that (IBM 801, Tanenbaum)
  - Easier to build fast, less power-hungry hardware
  - Let software do the complicated operations by composing simpler ones

Seminal Paper: "The Case For A Reduced Instruction Set Computer" Patterson & Ditzel

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
- Ex: 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."

Yeeep, definitely CISC.

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

PCs, some Macs
(Core i3, i5, i7, M)
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]
Branching	Condition code, compare and branch
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
M1/M2 Macs (new!)
ARM Instruction Set

RISC-V

Open Source! Very new!

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

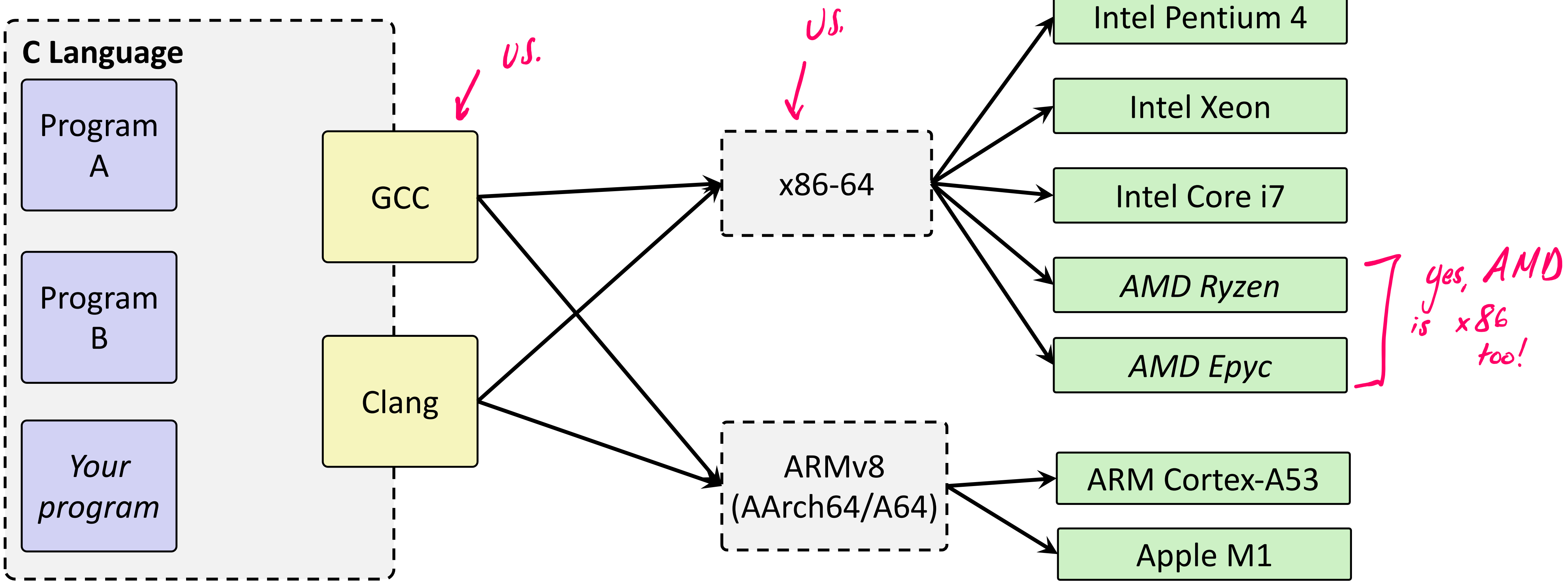
Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

Executed on ... via



Transform C programs to "very elementary instructions" executable by hardware

Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

```
long mult2(long, long);

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

→ GCC →

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

→

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

↕

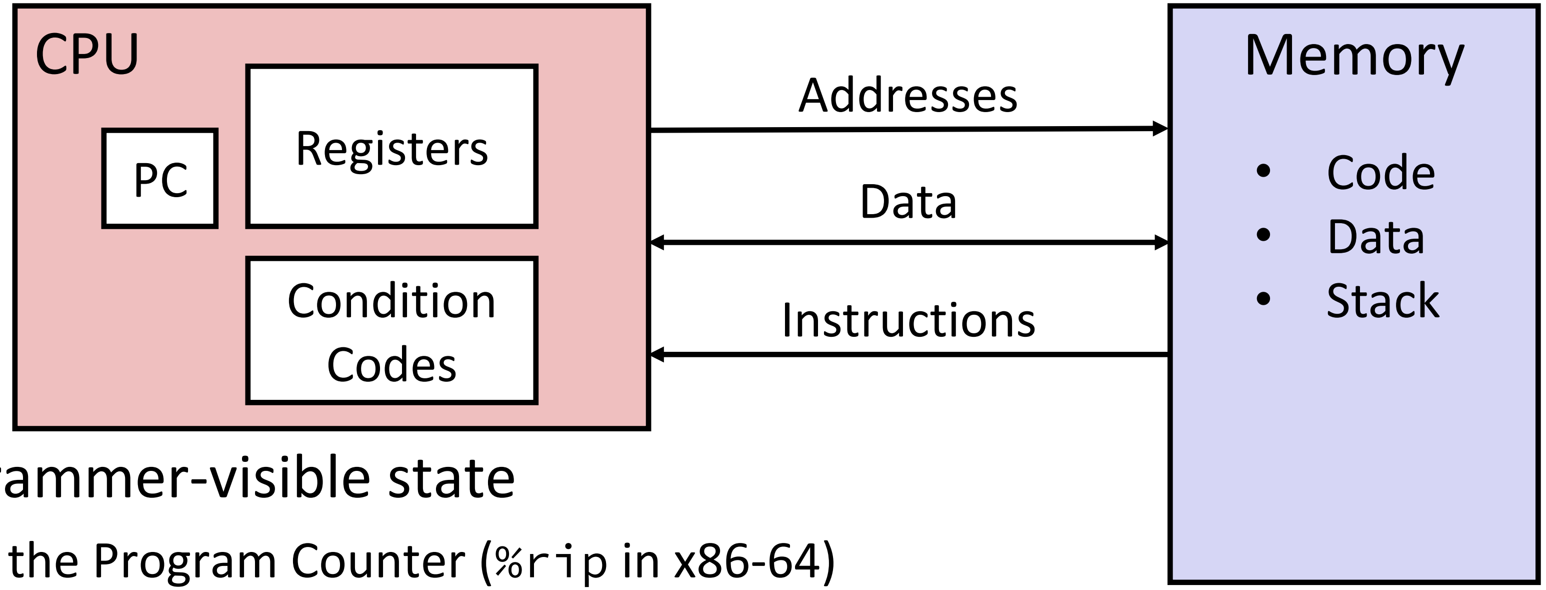
```
Binary:
0101 0011
0100 1000 1000 1001 1101 0011
1110 1000 0000 0000 0000 0000 0000 0000 0000 0000
0100 1000 1000 1001 0000 0011
0101 1011
1100 0011
```

See Section 3.2.2 in CSPP for more details…

Writing Assembly Code? Elba, are you serious?!

- 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 (very unlikely though…)
    - 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; how to find out what it's doing?

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 (b, w, l, q)
  - Data values
  - Addresses

8 bit 16 bit 32 bit 64 bit

- 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—Must know which you're reading!
  - "AT&T": used by our course, slides, textbook, gnu tools, … AKA Bell Labs
  - "Intel": used by Intel documentation, Intel tools, …

Us.

🚨 **They have switched operand orders!** 🚨

Instruction Types (Review)

1) Perform arithmetic operation on register or memory data
   - `c = a + b;`   `z = x << y;`   `i = h & g;`
2) Transfer data between memory and registers
   - **Load** data from memory into register
     - `%reg` = Mem[address] ← We signify this using parentheses
   - **Store** register data into memory
     - Mem[address] = `%reg`
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 (Review)

- Instruction operand size specifiers
  - `b` = 1-byte "byte", `w` = 2-byte "word", `l` = 4-byte "long word", `q` = 8-byte "quad word"

History Note: Due to backwards-compatible support for 8086 programs (Yes, 16-bit machines from 1978…), "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* x86-64

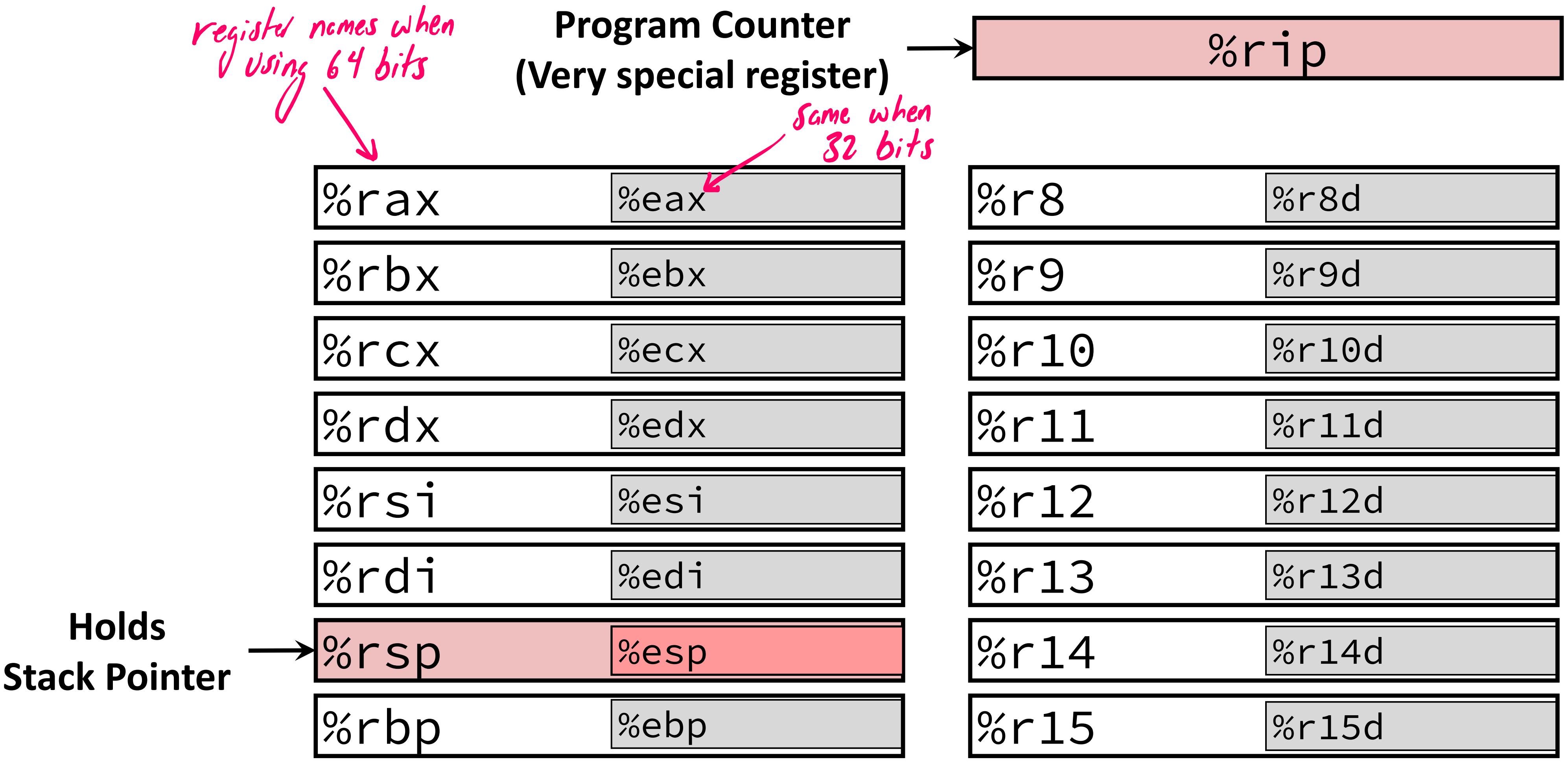
x86-64 only has 16 compared to Armv8's 31 registers!

Memory vs. Registers

Carlos Sainz, my beloved...

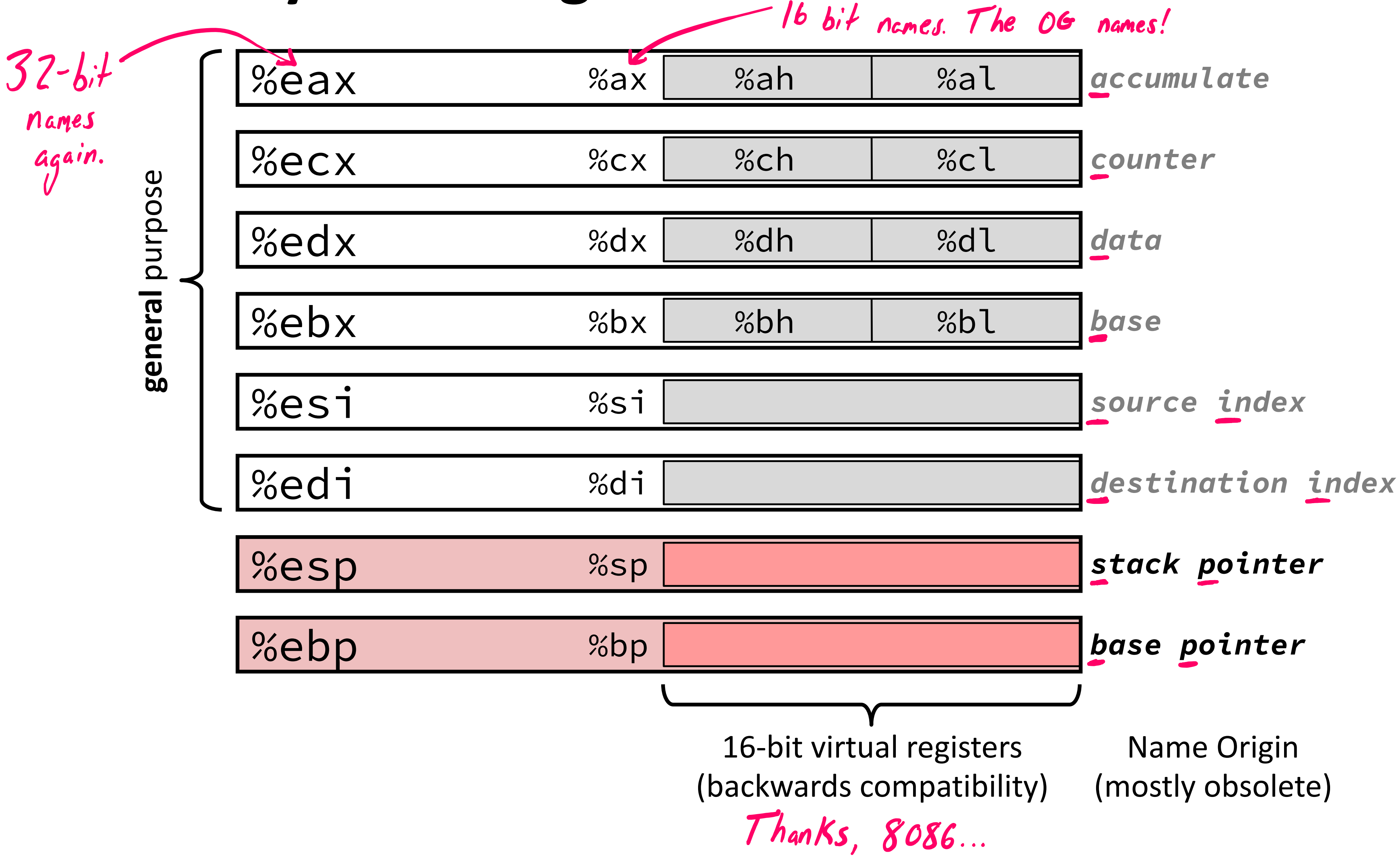
Memory	Registers
Addresses	**Names**
▪ `0x7FFFD024C3DC`	`%rdi`
Big	**Small**
▪ ~ 16 GiB	(16 x 8 B) = 128 B So few!!!
Super **Slow**	**Fast**
▪ ~50-100 ns	sub-nanosecond timescale thus, <1ns
Dynamic	**Static**
▪ Can “grow” as needed while program runs	fixed number in hardware

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



Moving Data

- General form: `mov_ source, destination`
  - Really more of a "copy" than a "move"
  - Like all instructions, missing letter above (_) is the size specifier e.g. `movq`, `movw`
  - Lots of these in typical code

Operand Combinations

x86 | C
immediate ⟷ constant
Register ⟷ variable
Mem ⟷ dereferenced pointer

	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;`

Note: Cannot do memory-memory transfer with a single instruction

- How would you do it?

e.g. Want to do: movq (%rax), (%rbx) ← Can't!

Do this instead:
1. Memory → Register movq (%rax), %rdx
2. Register → Memory movq %rdx, (%rbx)

Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - Beware argument order!
    - `src` and `dst` can be immediate, register,
    - or memory operands
    - Results always stored in `dst`

Maximum of one memory operand!

  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts

imm, reg, or Mem

oppr Format	Computation	always:
`addq src, dst`	*dst = dst + src*	(*dst += src*)
`subq src, dst`	*dst = dst – src*	
`imulq src, dst`	*dst = dst * src*	signed mult
`sarq src, dst`	*dst = dst >> src*	**A**rithmetic
`shrq src, dst`	*dst = dst >> src*	**L**ogical
`shlq src, dst`	*dst = dst << src*	(same as `salq`)
`xorq src, dst`	*dst = dst ^ src*	
`andq src, dst`	*dst = dst & src*	
`orq src, dst`	*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
    ```

Why? Next time!

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