Course Wrap-Up
CSE 351 Autumn 2018

Instructor: **Teaching Assistants:**

Justin Hsia

Akshat Aggarwal	An Wang	Andrew Hu
Brian Dai	Britt Henderson	James Shin
Kevin Bi	Kory Watson	Riley Germundson
Sophie Tian	Teagan Horkan	

https://xkcd.com/1760/
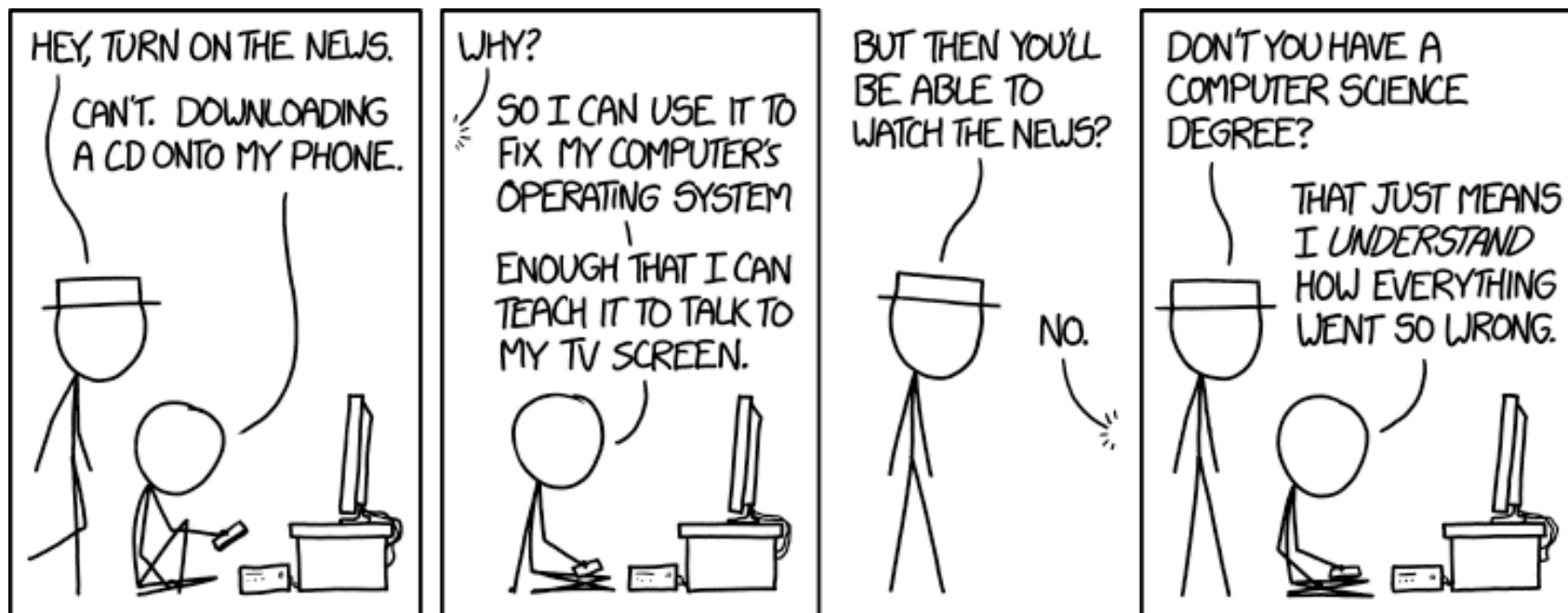
Administrivia

- ❖ Please fill out the **course evaluation**!
  - Evaluations close this Sunday, December 9th at 11:59 pm
    - Not viewable until after grades are submitted
  - We take these seriously and use them to improve our teaching and this class!

- ❖ **Final Exam:** Wed 12/12, 12:30-2:20 pm in KNE 120
  - Review Session: Sun 12/9, 5-7 pm in EEB 105
  - Cumulative (midterm clobber policy applies)
  - TWO double-sided handwritten 8.5×11" cheat sheets
    - Recommended that you reuse or remake your midterm cheat sheet

Today

❖ End-to-end Review

  ▪ What happens after you write your source code?

    • How code becomes a program

    • How your computer executes your code

❖ Victory lap and high-level concepts (🔑 points)

  ▪ More useful for "5 years from now" than "next week's final"

❖ Question time

C: The Low-Level High-Level Language

❖ C is a "hands-off" language that "exposes" more of hardware (especially memory)

  ▪ Weakly-typed language that stresses data as bits

    • Anything can be represented with a number!

  ▪ Unconstrained pointers can hold address of *anything*

    • And no bounds checking – buffer overflow possible!

  ▪ Efficient by leaving everything up to the programmer

  ▪ "C is good for two things: being beautiful and creating catastrophic 0days in memory management." (https://medium.com/message/everything-is-broken-81e5f33a24e1)

C Data Types

- ❖ C Primitive types
  - Fixed sizes and alignments
  - Characters (`char`), Integers (`short`, `int`, `long`), Floating Point (`float`, `double`)
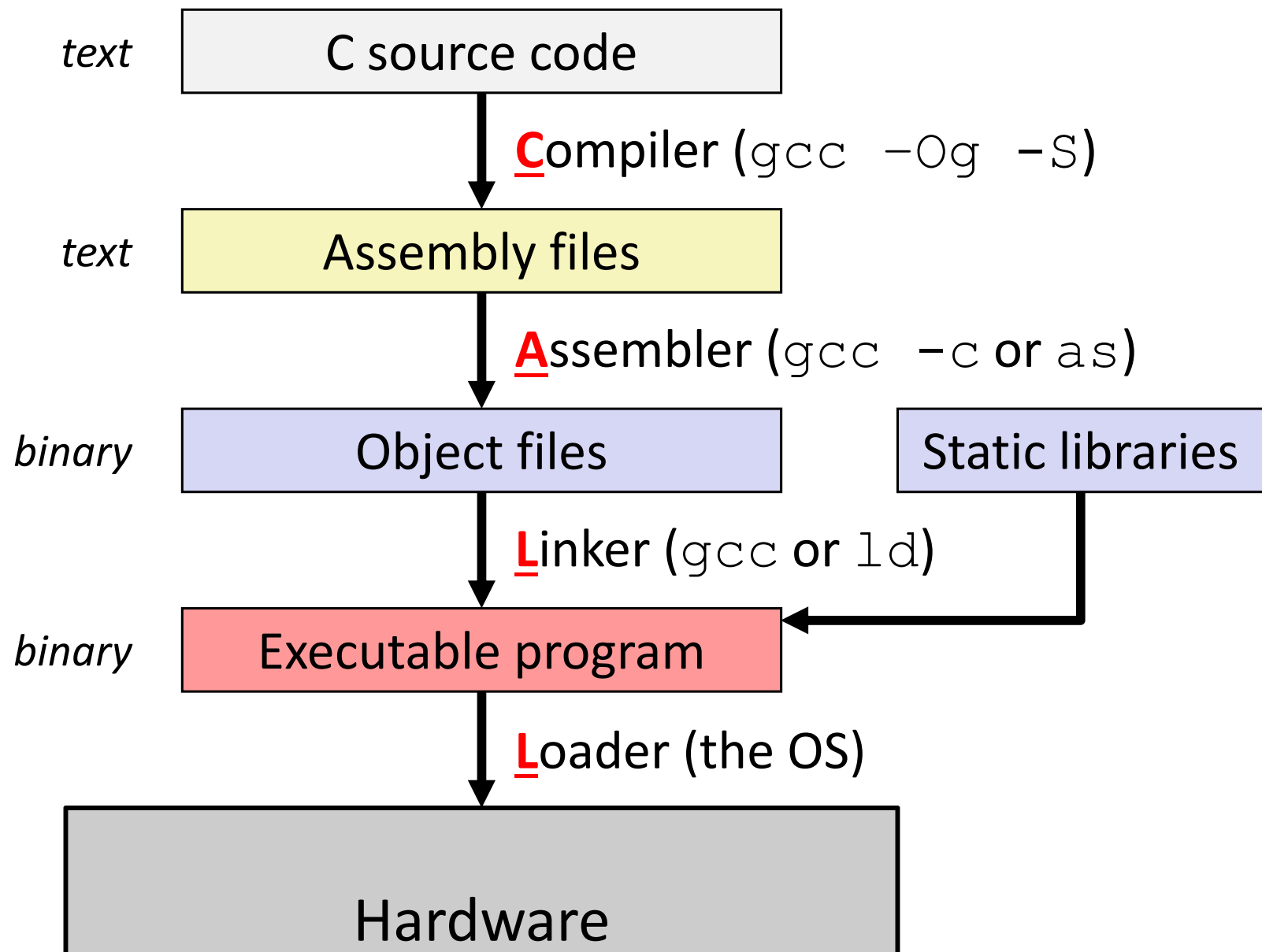
- ❖ C Data Structures
  - Arrays – contiguous chunks of memory
    - Multidimensional arrays = still one continuous chunk, but row-major
    - Multi-level arrays = array of pointers to other arrays
  - Structs – structured group of variables
    - Struct fields are ordered according to declaration order
    - *Internal* **fragmentation:** space between members to satisfy member alignment requirements (aligned for each primitive element)
    - *External* **fragmentation:** space after last member to satisfy overall struct alignment requirement (largest primitive member)

C and Memory

- ❖ Using C allowed us to examine how we store and access data in memory
  - ▪ Endianness (**only applies to memory**)
    - • Is the first byte (lowest address) the least significant (little endian) or most significant (big endian) of your data?
  - ▪ Array indices and struct fields result in calculating proper addresses to access
- ❖ Consequences of your code:
  - ▪ Affects performance (locality)
  - ▪ Affects security
- ❖ But to understand these effects better, we had to dive deeper…

How Code Becomes a Program

text   C source code

Compiler (`gcc -Og -S`)

text   Assembly files

Assembler (`gcc -c` or `as`)

binary   Object files   Static libraries

Linker (`gcc` or `ld`)

binary   Executable program

Loader (the OS)

Hardware

Instruction Set Architecture

Source code

Different applications
or algorithms

Compiler

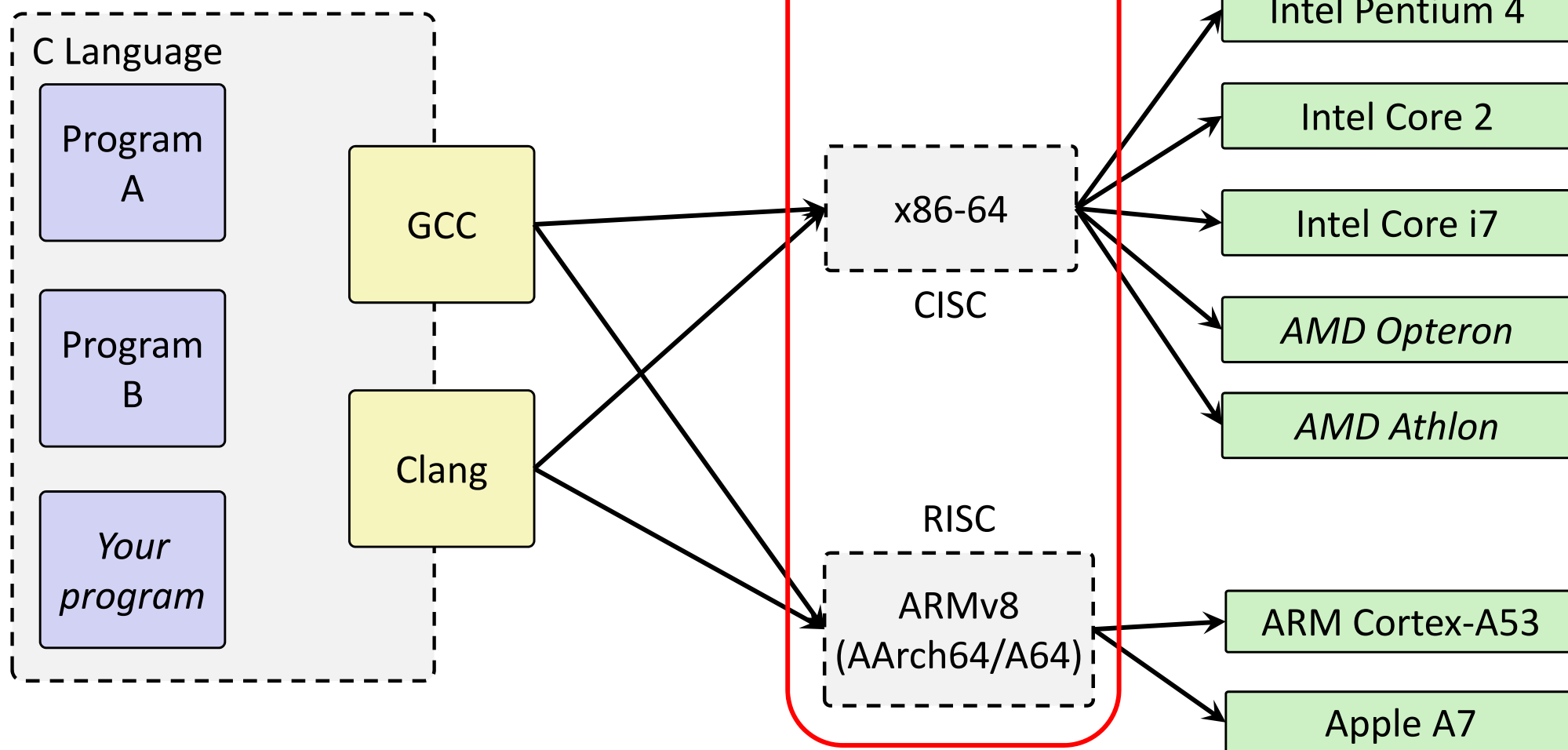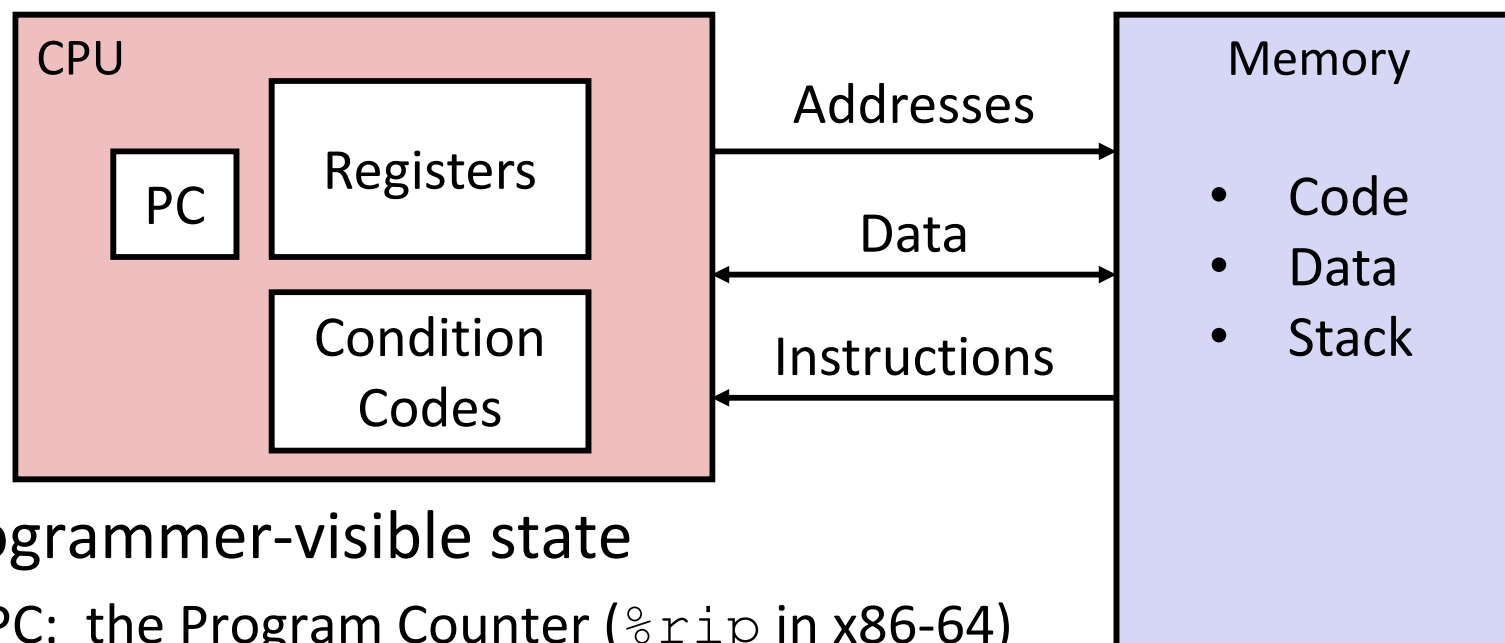Perform optimizations,
generate instructions

Architecture

Instruction set

Hardware

Different
implementations

C Language

- Program A
- Program B
- *Your program*

GCC

Clang

x86-64

CISC

RISC

ARMv8
(AArch64/A64)

- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- *AMD Opteron*
- *AMD Athlon*
- ARM Cortex-A53
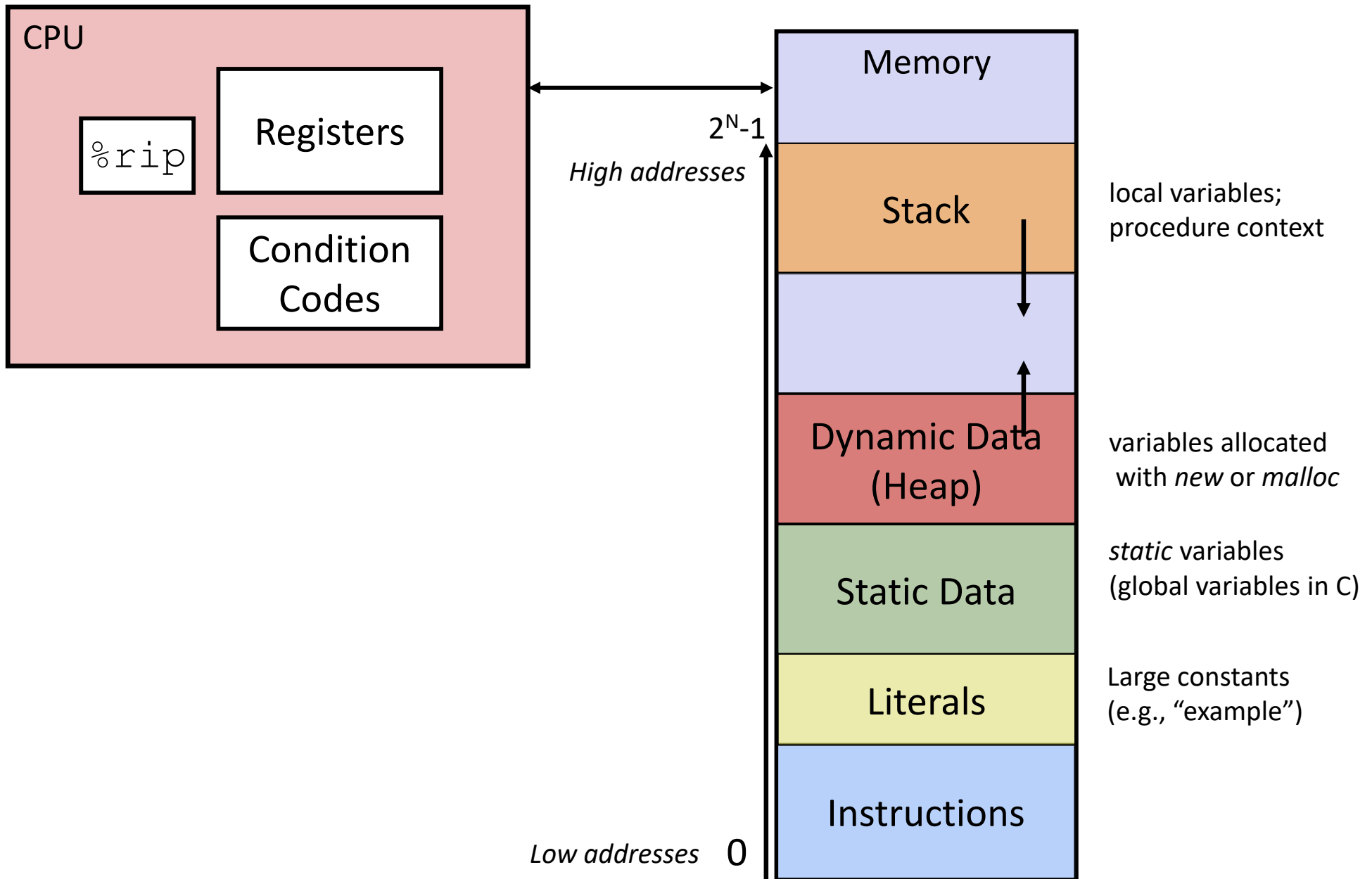- Apple A7

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
  ▪ Huge *virtual* address space
  ▪ *Private, all to yourself...*

Program's View

CPU

`%rip` Registers

Condition Codes

Memory

2^N-1

High addresses

Stack — local variables; procedure context

Dynamic Data (Heap) — variables allocated with *new* or *malloc*

Static Data — *static* variables (global variables in C)

Literals — Large constants (e.g., "example")

Instructions

Low addresses 0

Program's View

❖ **Instructions**

- ▪ **Data movement**
  - mov, movz, movz
  - push, pop
- ▪ **Arithmetic**
  - add, sub, imul
- ▪ **Control flow**
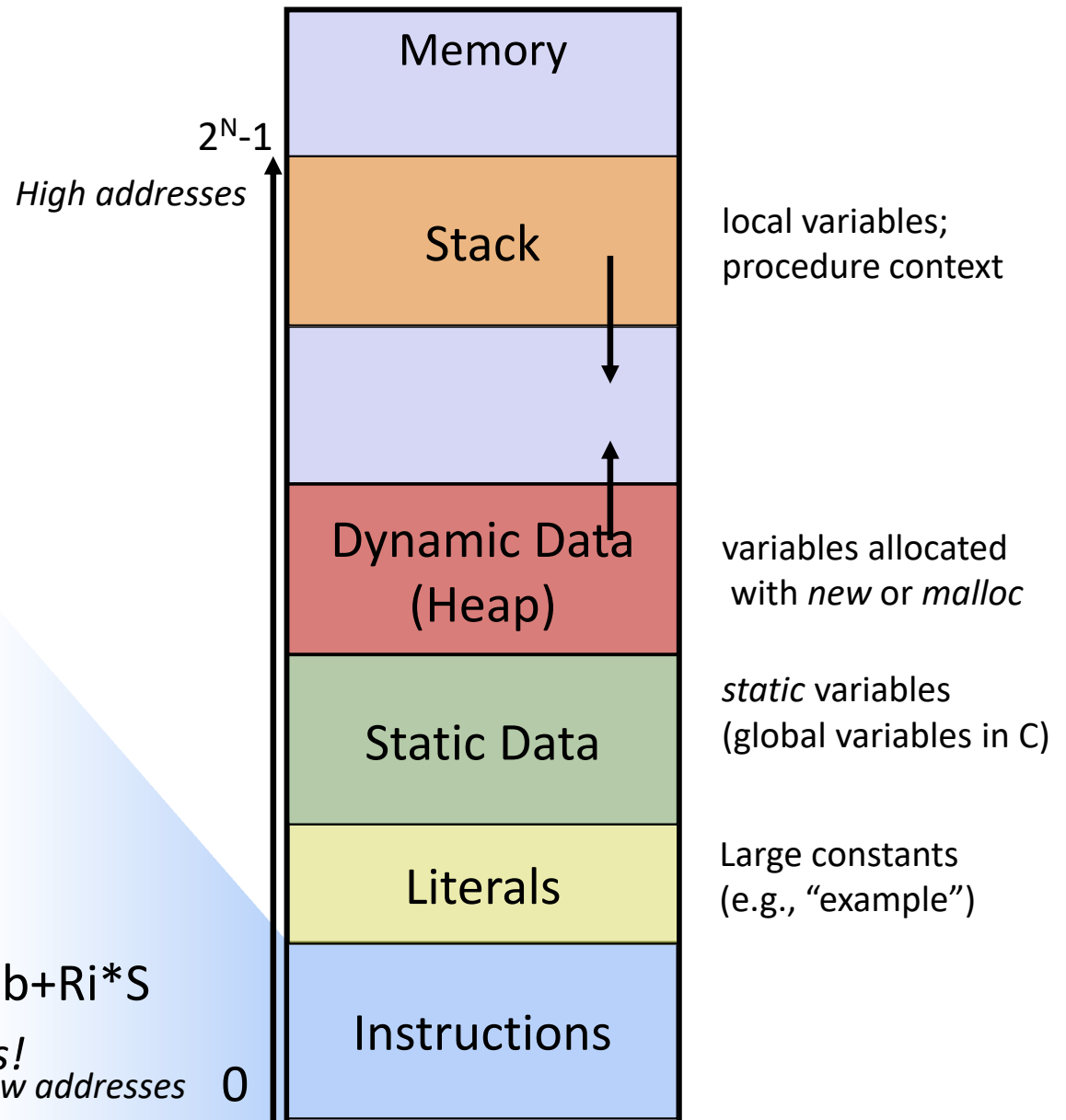 - cmp, test
  - jmp, je, jgt, ...
  - call, ret
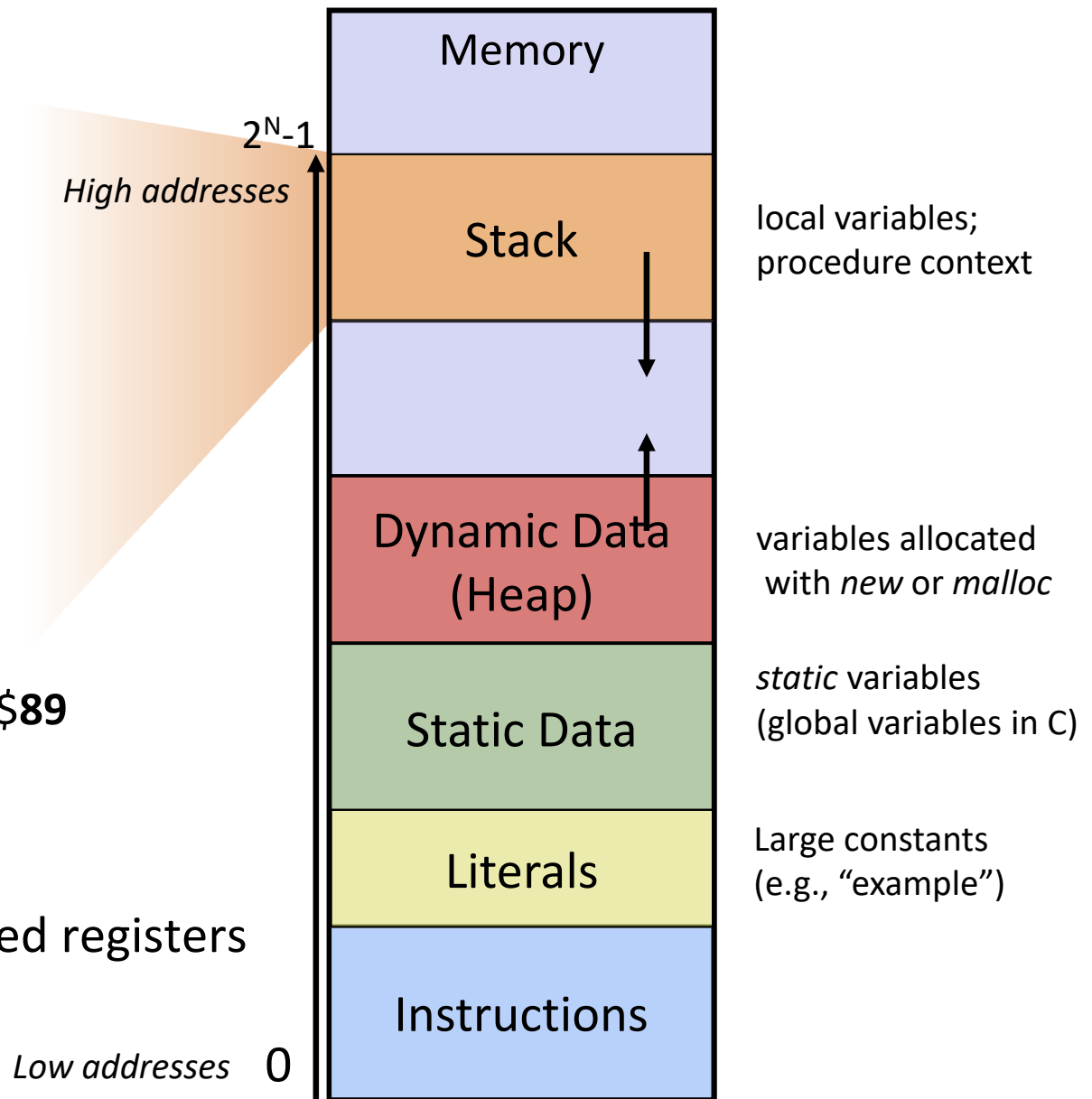
❖ **Operand types**

- ▪ Literal: $8
- ▪ Register: %rdi, %al
- ▪ Memory: D(Rb,Ri,S) = D+Rb+Ri*S
  - lea: *not a memory access!*

Memory

2^N-1

High addresses

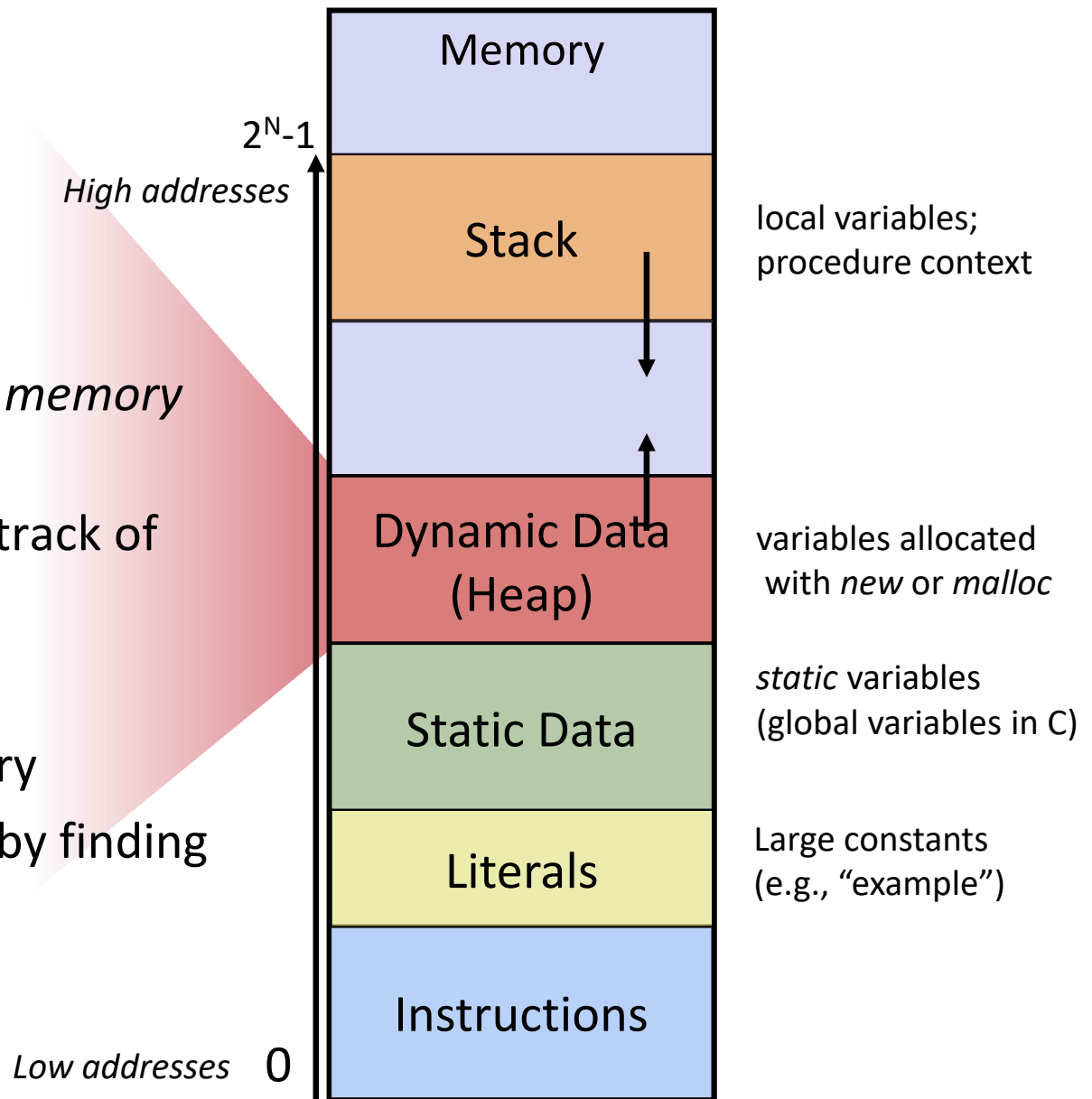| Stack | local variables; procedure context |
Dynamic Data (Heap)	variables allocated with *new* or *malloc*
Static Data	*static* variables (global variables in C)
Literals	Large constants (e.g., "example")
Instructions	

Low addresses   0

Program's View

- ❖ Procedures
  - ▪ Essential abstraction
  - ▪ Recursion…
- ❖ Stack discipline
  - ▪ Stack frame per call
  - ▪ Local variables
- ❖ Calling convention
  - ▪ How to pass arguments
    - • **Di**ane's **Si**lk **D**ress **C**osts $**89**
  - ▪ How to return data
  - ▪ Return address
  - ▪ Caller-saved / callee-saved registers

2^N-1

High addresses

Memory
Stack
Dynamic Data (Heap)
Static Data
Literals
Instructions

local variables; procedure context

variables allocated with *new* or *malloc*

static variables (global variables in C)

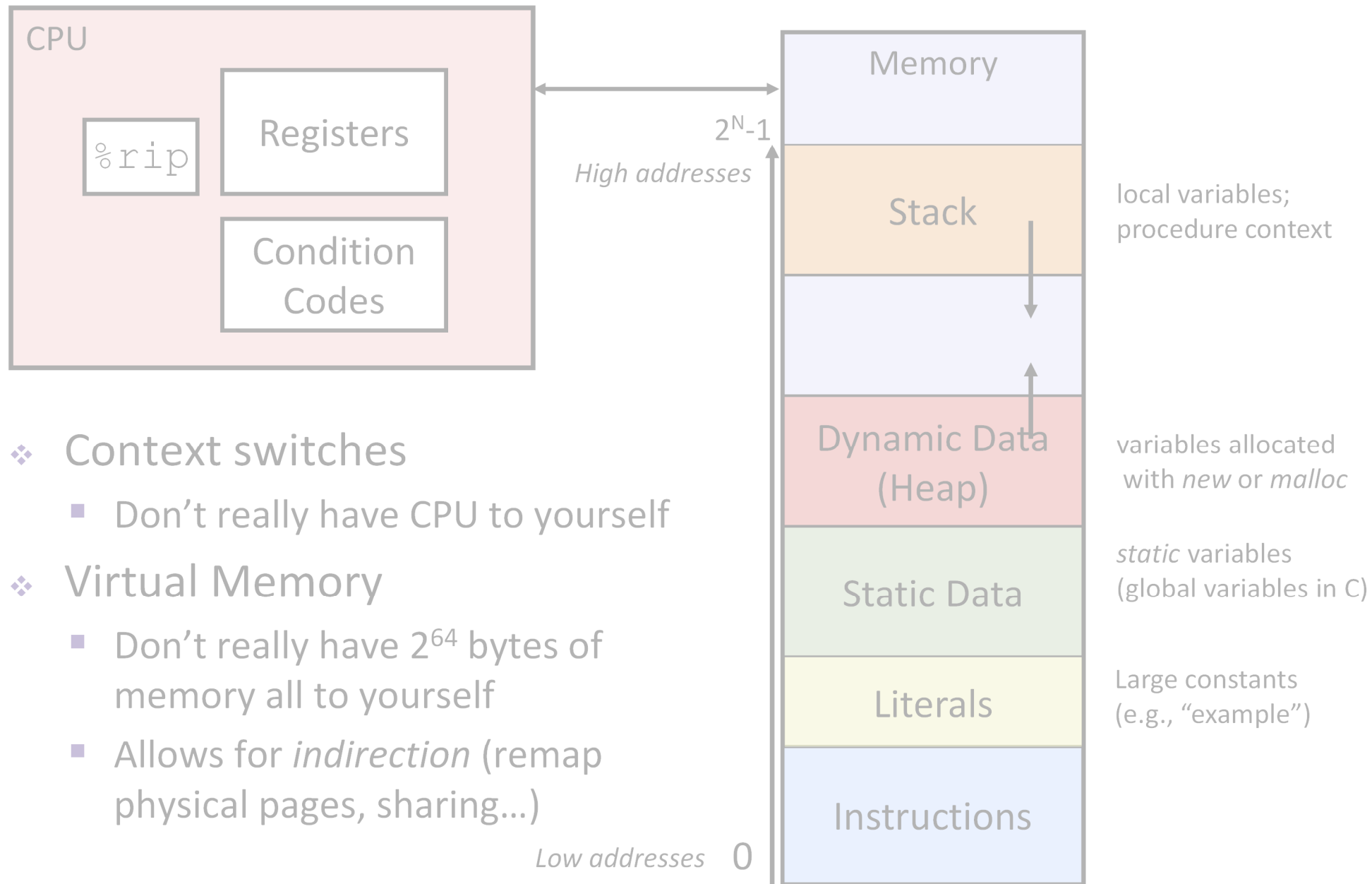Large constants (e.g., "example")
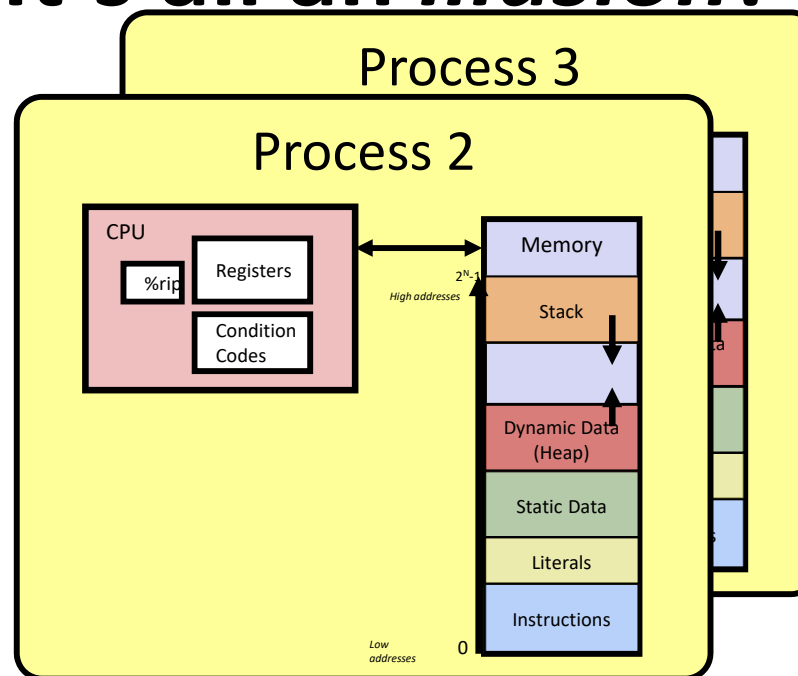
Low addresses 0

12

Program's View

❖ Heap data
  ▪ Variable size
  ▪ Variable lifetime

❖ Allocator
  ▪ Balance *throughput* and *memory utilization*
  ▪ Data structures to keep track of free blocks

❖ Garbage collection
  ▪ Must always free memory
  ▪ Garbage collectors help by finding anything *reachable*
 ▪ Failing to free results in *memory leaks*

Memory

2^N-1

High addresses

Stack — local variables; procedure context

Dynamic Data (Heap) — variables allocated with *new* or *malloc*

Static Data — *static* variables (global variables in C)

Literals — Large constants (e.g., "example")

Instructions

Low addresses 0

But remember... it's all an *illusion*! 😮

CPU

%rip

Registers

Condition Codes

Memory

2^N-1
High addresses

Stack — local variables; procedure context

Dynamic Data (Heap) — variables allocated with *new* or *malloc*

Static Data — *static* variables (global variables in C)

Literals — Large constants (e.g., "example")

Instructions

Low addresses 0

❖ Context switches
   ▪ Don't really have CPU to yourself
❖ Virtual Memory
   ▪ Don't really have 2^{64} bytes of memory all to yourself
   ▪ Allows for *indirection* (remap physical pages, sharing...)

But remember… it's all an *illusion*! 😮

- ❖ `fork`
  - ▪ Creates copy of the process
- ❖ `execv`
  - ▪ Replace with new program
- ❖ `wait`
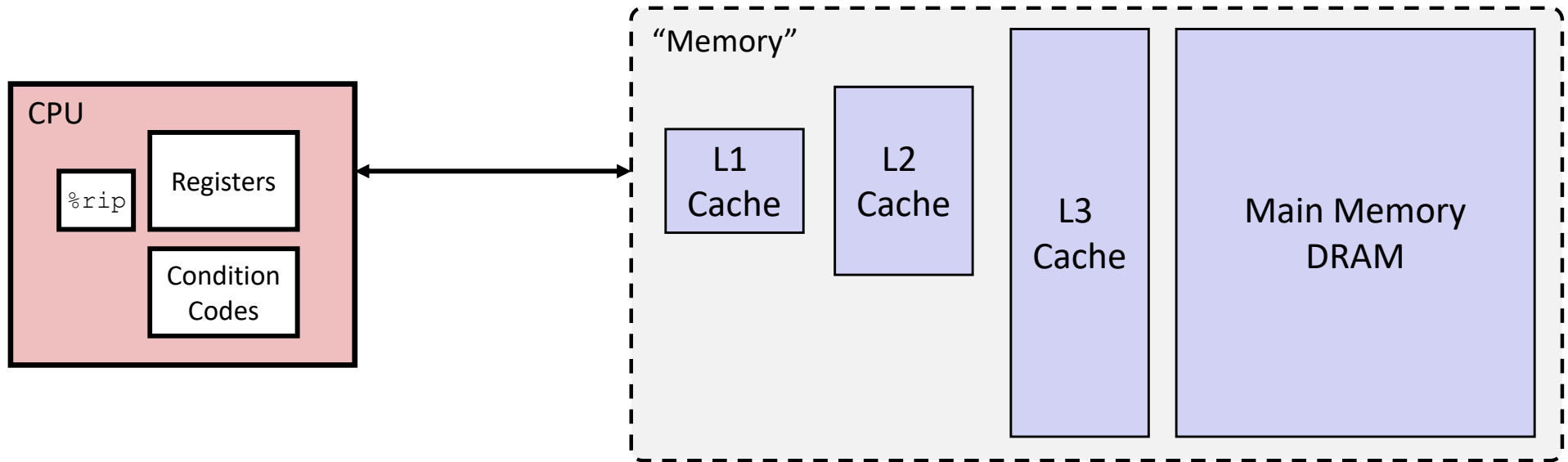 - ▪ Wait for child to die (to *reap* it and prevent *zombies*)
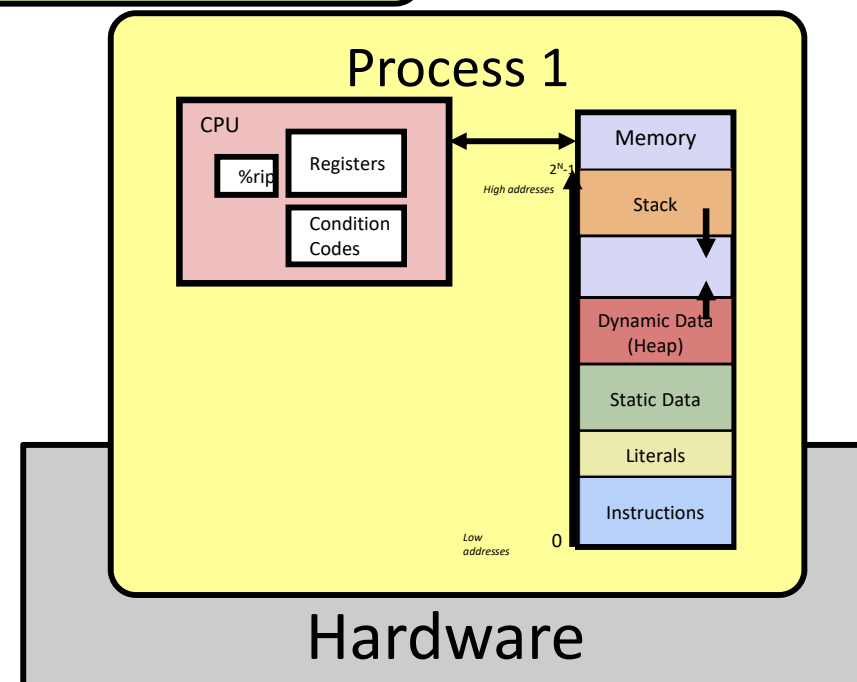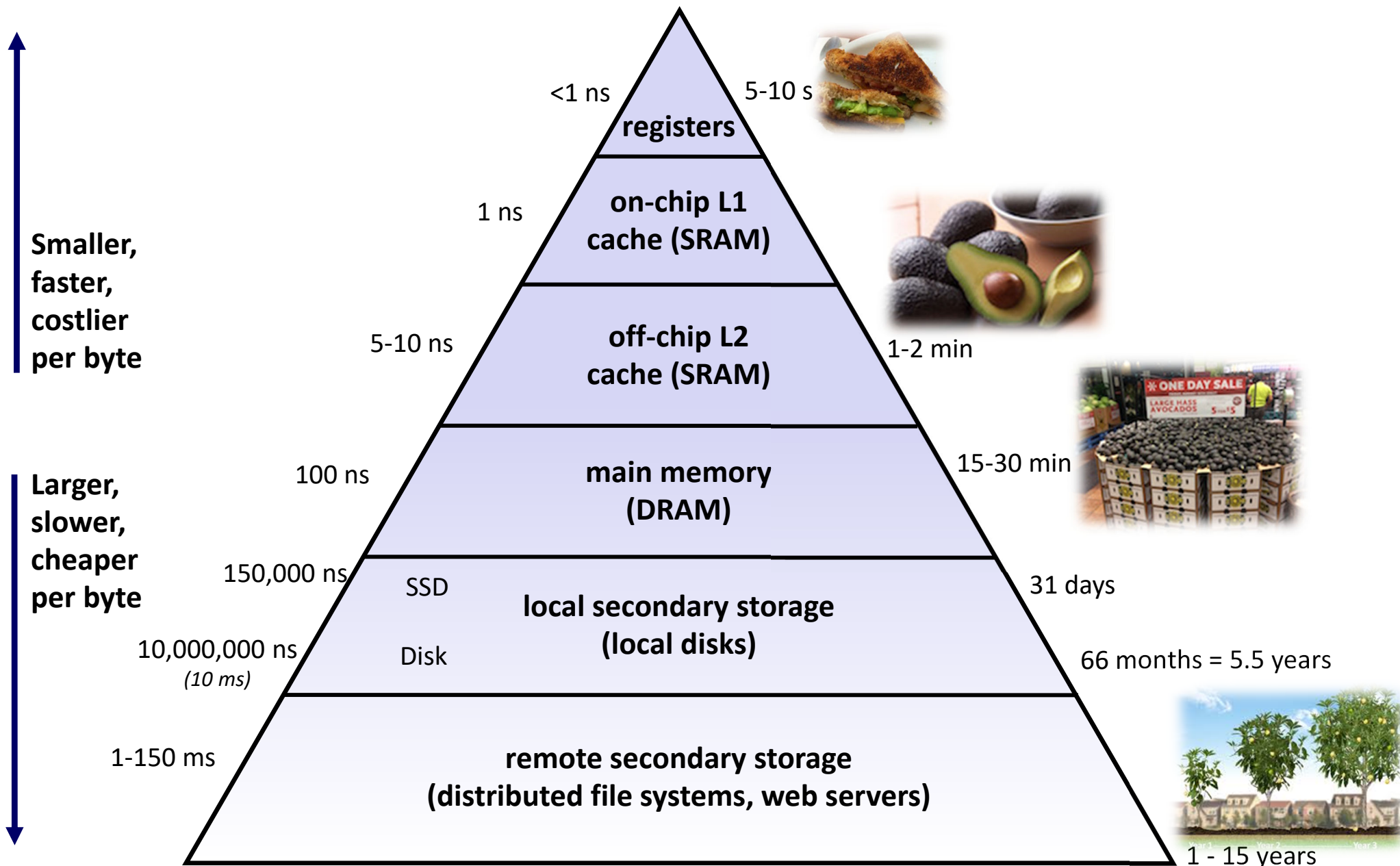
Virtual Memory
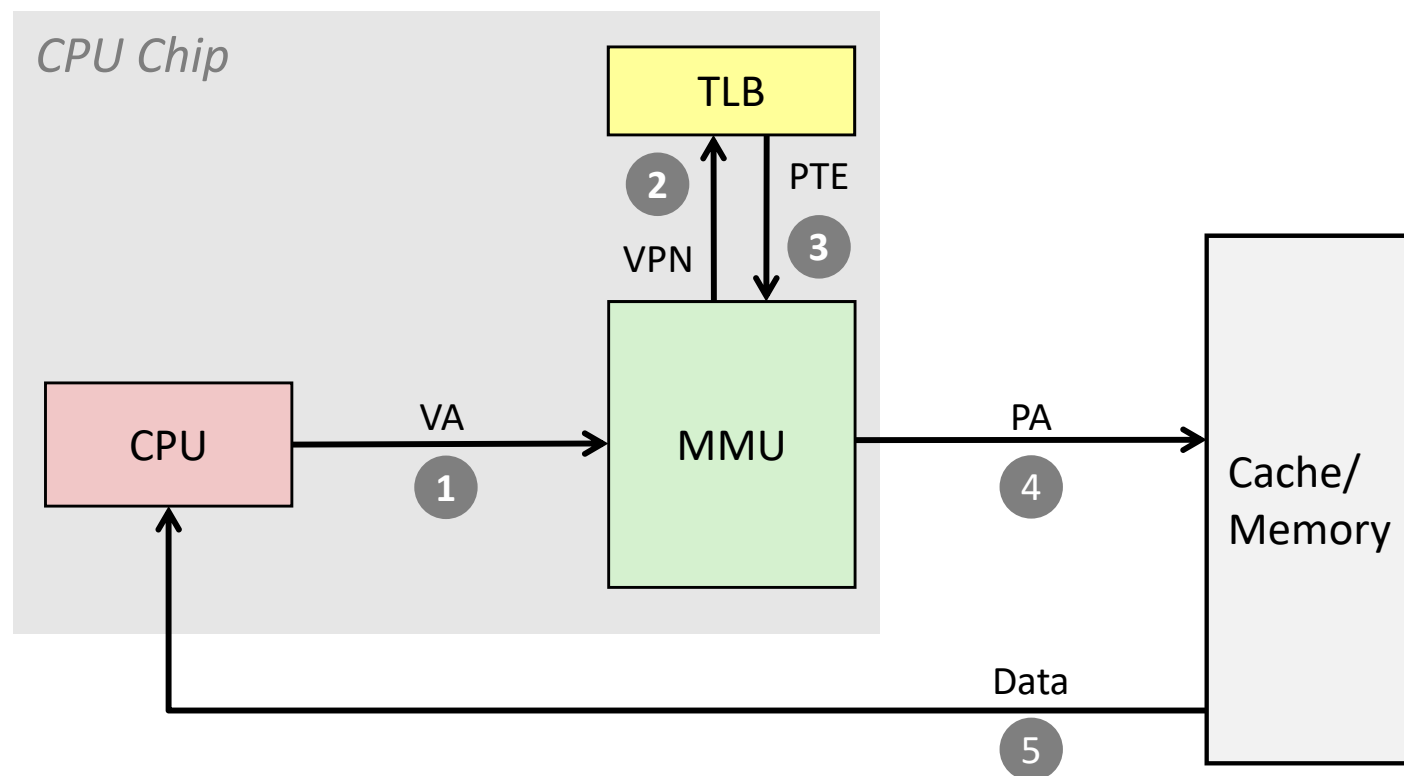
❖ Address Translation
  ▪ Every memory access must first be converted from virtual to physical
  ▪ *Indirection:* just change the address mapping when switching processes
  ▪ Luckily, TLB (and page size) makes it pretty fast

But Memory is Also a Lie! 😲

CPU

%rip   Registers

Condition Codes

"Memory"

L1 Cache

L2 Cache

L3 Cache

Main Memory DRAM

❖ *Illusion* of one flat array of bytes
  ▪ But *caches* invisibly make accesses to physical addresses faster!
❖ Caches
  ▪ **Associativity** tradeoff with miss rate and access time
  ▪ **Block size** tradeoff with spatial and temporal locality
  ▪ **Cache size** tradeoff with miss rate and cost

Memory Hierarchy

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

<1 ns   **registers**   5-10 s

1 ns   **on-chip L1 cache (SRAM)**

5-10 ns   **off-chip L2 cache (SRAM)**   1-2 min

100 ns   **main memory (DRAM)**   15-30 min

150,000 ns   SSD   **local secondary storage (local disks)**   31 days

10,000,000 ns *(10 ms)*   Disk   66 months = 5.5 years

1-150 ms   **remote secondary storage (distributed file systems, web servers)**   1 - 15 years

Victory Lap

- ❖ A victory lap is an extra trip around the track
  - By the exhausted victors (that's us) ☺

- ❖ Review course goals
  - Put everything in perspective
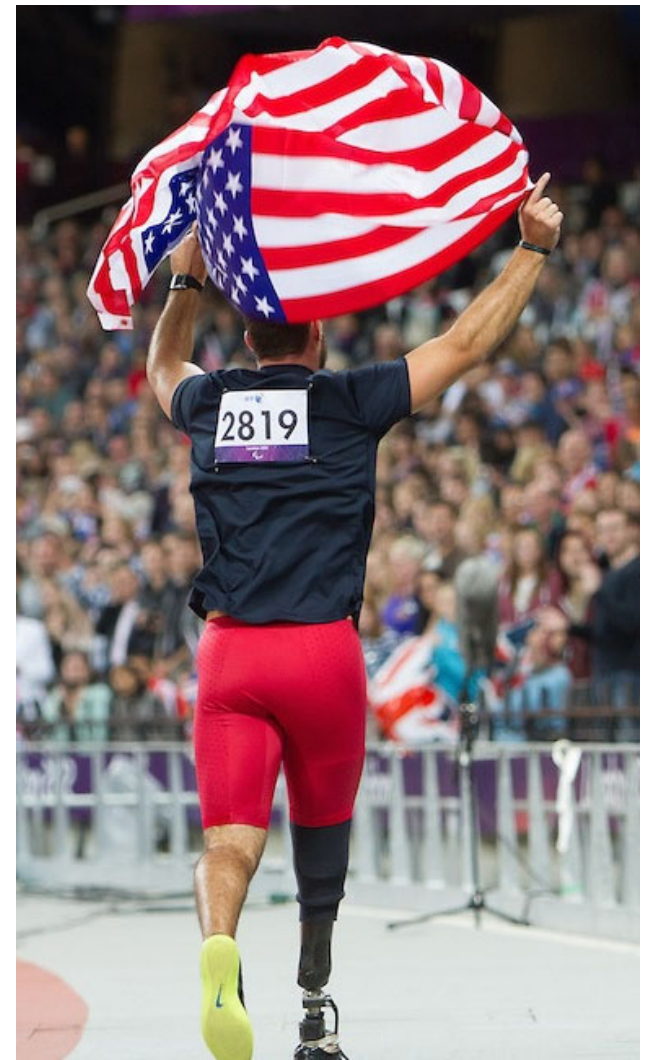
Big Theme: Abstractions and Interfaces

- ❖ Computing is about abstractions
  - ▪ (but we can't forget reality)
- ❖ What are the abstractions that we use?
- ❖ What do <u>you</u> need to know about them?
  - ▪ When do they break down and you have to peek under the hood?
  - ▪ What bugs can they cause and how do you find them?
- ❖ How does the hardware relate to the software?
  - ▪ Become a better programmer and begin to understand the important concepts that have evolved in building ever more complex computer systems

Little Theme 1: Representation

❖ **All digital systems represent everything as 0s and 1s**
  - The 0 and 1 are really two different voltage ranges in the wires
  - Or magnetic positions on a disc, or hole depths on a DVD, or even *DNA*…

❖ **"Everything" includes:**
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects stored away in memory

❖ **Encodings are stored throughout a computer system**
  - In registers, caches, memories, disks, etc.

❖ **They all need addresses (a way to locate)**
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed

Little Theme 2: Translation

- ❖ There is a big gap between how we think about programs and data and the 0s and 1s of computers
  - Need languages to describe what we mean
  - These languages need to be translated one level at a time

- ❖ We know Java as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We encountered C language, assembly language, and machine code (for the x86 family of CPU architectures)

Little Theme 3: Control Flow

❖ How do computers orchestrate everything they are doing?

❖ Within one program:

- How do we implement if/else, loops, switches?

- What do we have to keep track of when we call a procedure, and then another, and then another, and so on?

- How do we know what to do upon "return"?

❖ Across programs and operating systems:

- Multiple user programs

- Operating system has to orchestrate them all
  - Each gets a share of computing cycles
  - They may need to share system resources (memory, I/O, disks)

- Yielding and taking control of the processor
  - Voluntary or "by force"?

Course Perspective

- ❖ CSE351 will make you a better programmer
  - ▪ Purpose is to show how software really works
  - ▪ Understanding the underlying system makes you more effective
    - • Better debugging
    - • Better basis for evaluating performance
    - • How multiple activities work in concert (e.g., OS and user programs)
  - ▪ Not just a course for hardware enthusiasts!
    - • What **every** CSE major needs to know (plus many more details)
    - • See many **patterns** that come up over and over in computing (like caching)
  - ▪ "Stuff everybody learns and uses and forgets not knowing"
- ❖ CSE351 presents a world-view that will empower you
  - ▪ The intellectual and software tools to understand the trillions+ of 1s and 0s that are "flying around" when your program runs

Can You Now Explain These to a Friend?

❖ Which of the following did you actually find the most interesting to learn about? (http://PollEv.com/justinh)

a) What is a GFLOP and why is it used in computer benchmarks?

b) How and why does running many programs for a long time eat into your memory (RAM)?

c) What is stack overflow and how does it happen?

d) Why does your computer slow down when you run out of *disk* space?

e) What was the flaw behind the original Internet worm and the Heartbleed bug?

f) What is the meaning behind the different CPU specifications? (e.g. # of cores, # and size of cache, supported memory types)

The Very First Comic of the Quarter

http://xkcd.com/676/

Courses: What's Next?

❖ Staying near the hardware/software interface:

- **EE271/CSE369:** Digital Design – basic hardware design using FPGAs
- **EE/CSE474:** Embedded Systems – software design for microcontrollers

❖ Systems software

- **CSE341:** Programming Languages
- **CSE332:** Data Structures and Parallelism
- **CSE333:** Systems Programming – building well-structured systems in C/C++

❖ Looking ahead

- **CSE401:** Compilers (pre-reqs: 332)
- **CSE451:** Operating Systems (pre-reqs: 332, 333)
- **CSE461:** Networks (pre-reqs: 332, 333)

Thanks for a great quarter!

❖ Huge thanks to your awesome TAs!
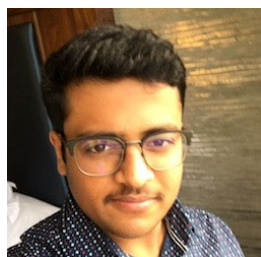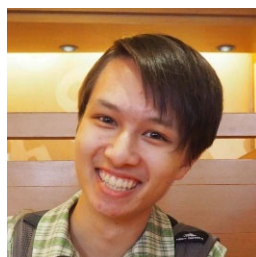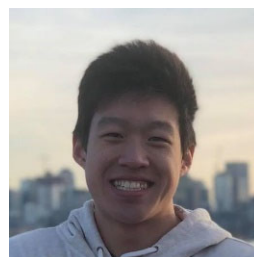
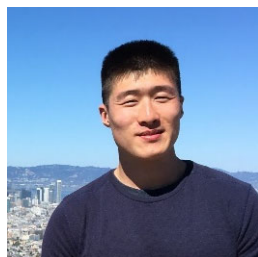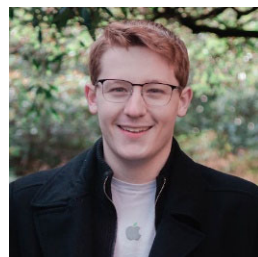Akshat   An   Andrew   Brian   Britt   James
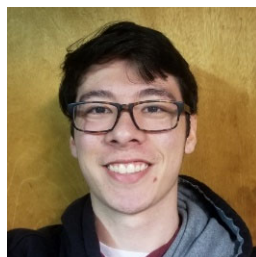
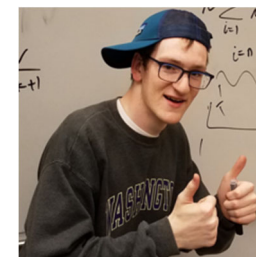Kevin   Kory   Riley   Sophie   Teagan

❖ Don't be a stranger!

▪ If interested, I'm teaching CSE369 (Wi19, Sp19) and CSE333 (Sp19)

Ask Me Anything