Course Wrap-Up

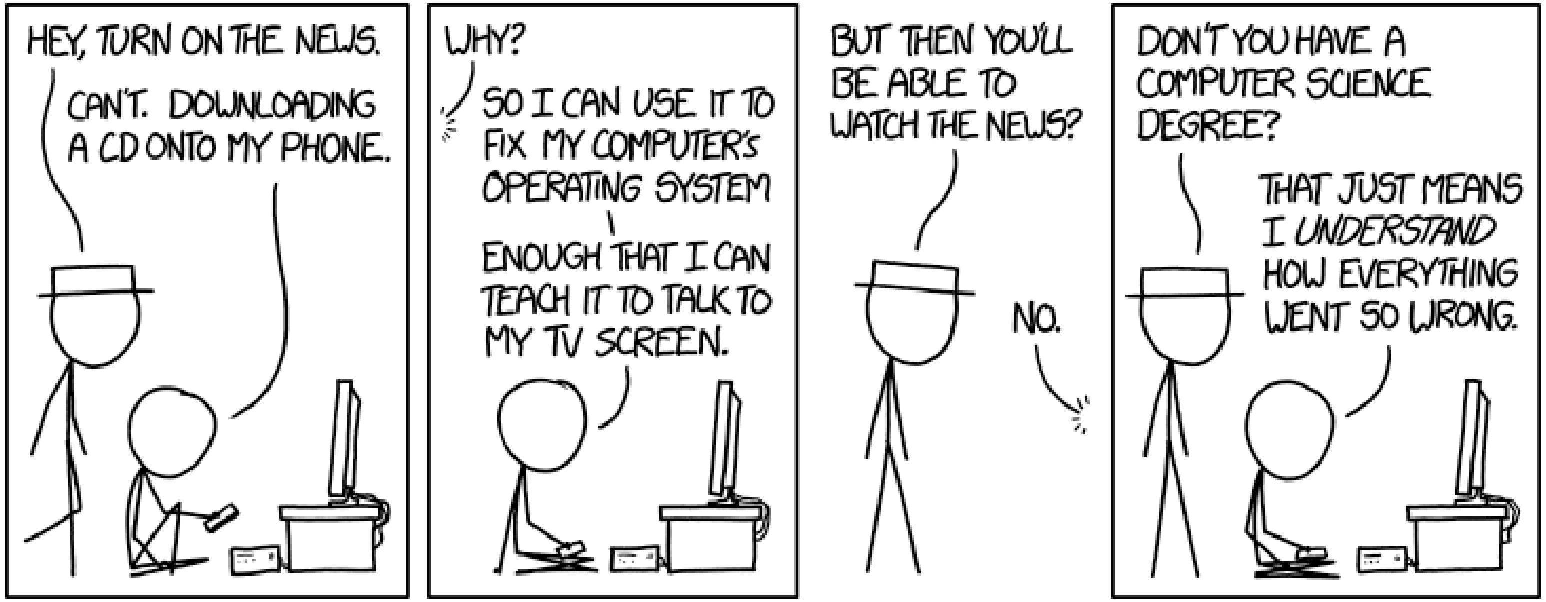
CSE 351 Winter 2024

Instructor:

Justin Hsia

Teaching Assistants:

Adithi Raghavan
Aman Mohammed
Connie Chen
Eyoel Gebre
Jiawei Huang
Malak Zaki
Naama Amiel
Nathan Khuat
Nikolas McNamee
Pedro Amarante
Will Robertson



https://xkcd.com/1760/

Relevant Course Information

- Please fill out the **course evaluation**!
  - Evaluations close Sunday, March 10 at 11:59 pm
    - Not viewable until after grades are submitted
  - See Ed post #572 for links (separate for Lecture and Section)
  - We take these seriously and use them to improve our teaching and this class!

- **Final Exam:** take-home March 11-13
  - Review Session: tonight, 4:30-6:30 pm on Zoom & CSE2 G01
  - Similar structure to Midterm, including Gilligan's Island Rule
  - Final reference sheet on website

End-to-End Review

- What happens after you write your source code?
  - How code becomes a program
  - How your computer executes your code

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 (one row-major chunk) vs. multi-level (array of pointers to other arrays)
  - Structs – structured group of fields 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)

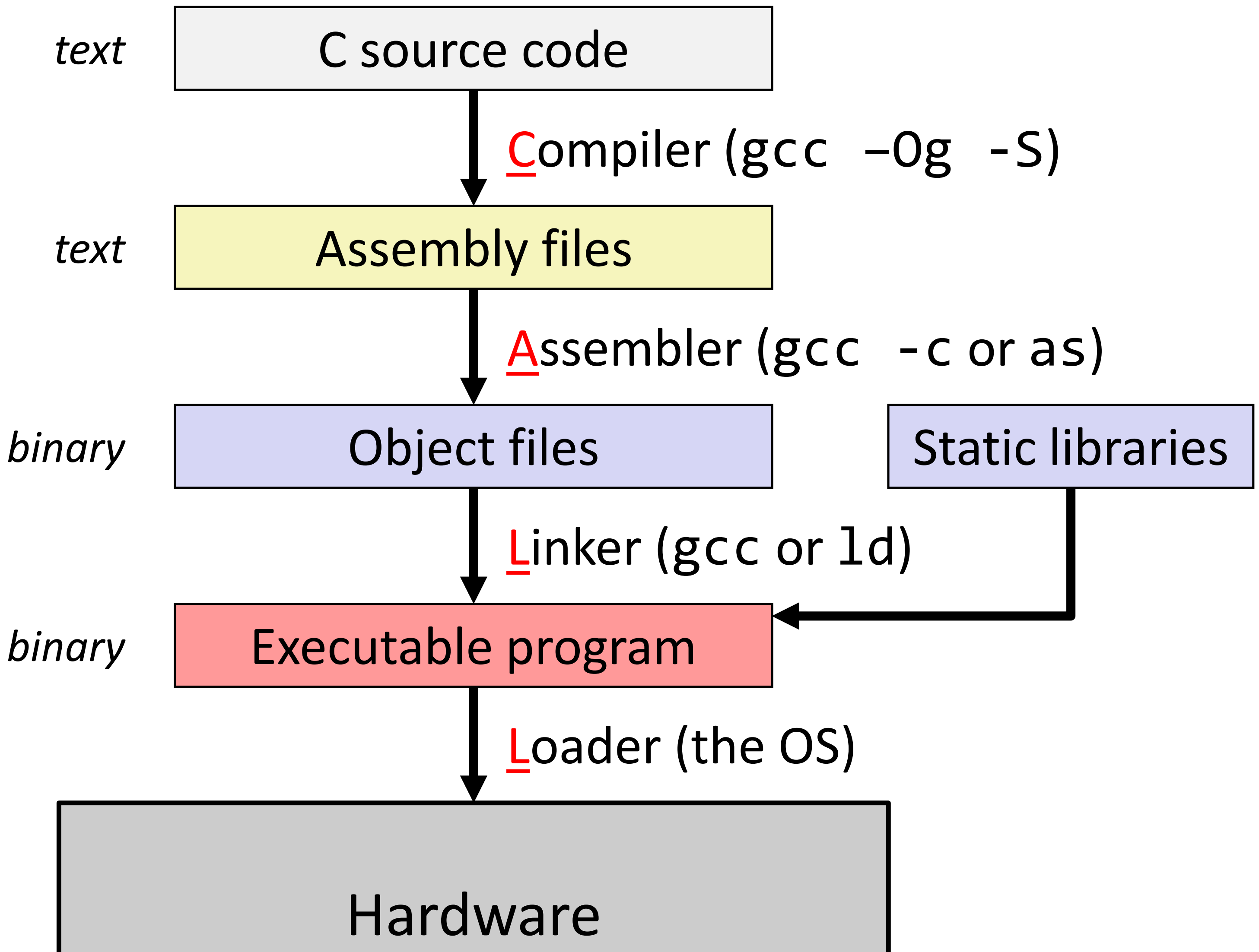
C and Memory

- 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



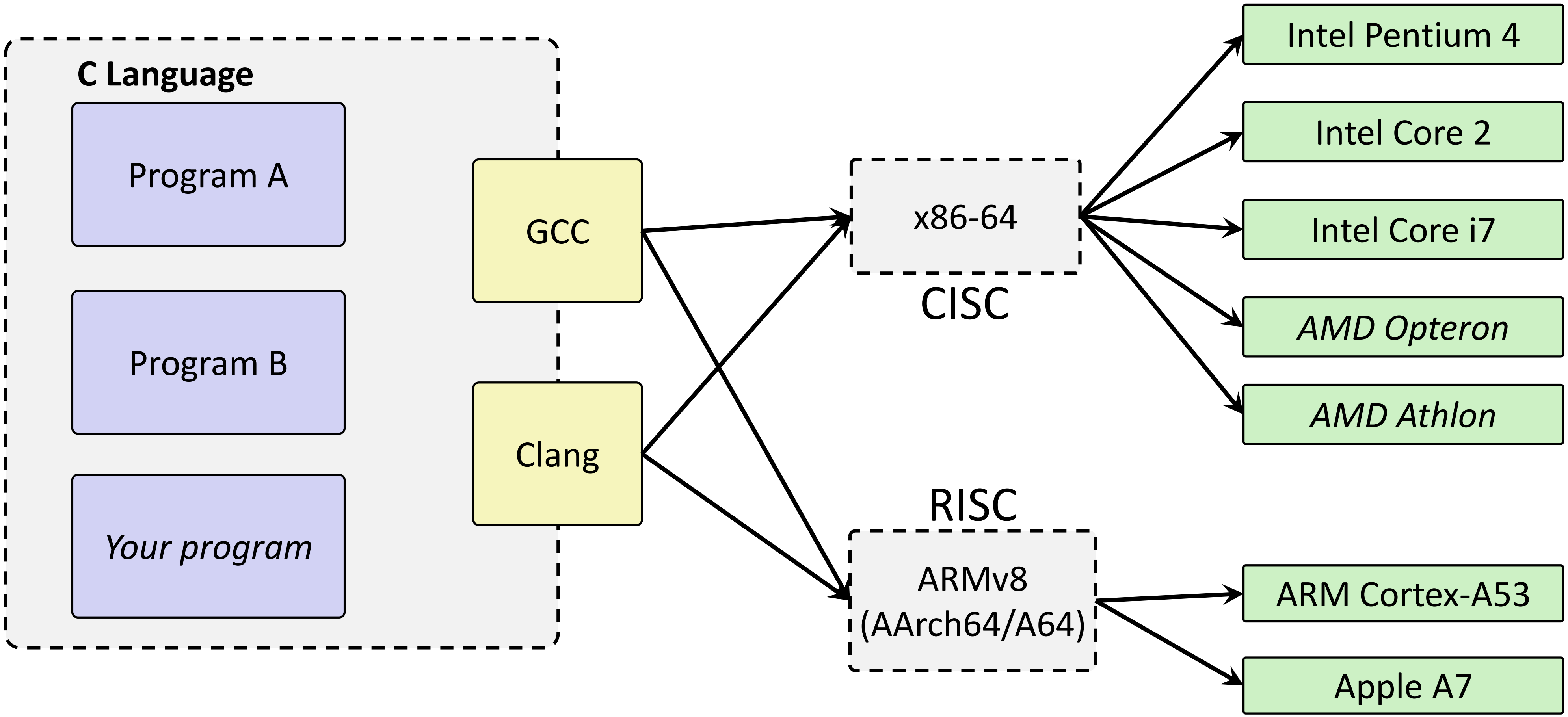
Instruction Set Architecture

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

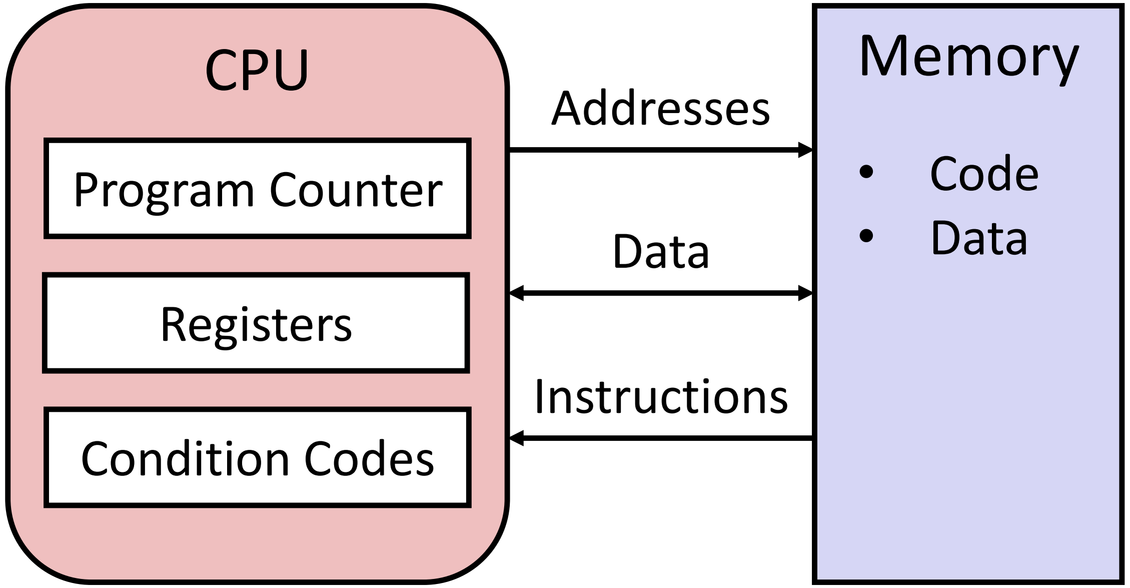
Architecture
Instruction set

Hardware
Different implementations

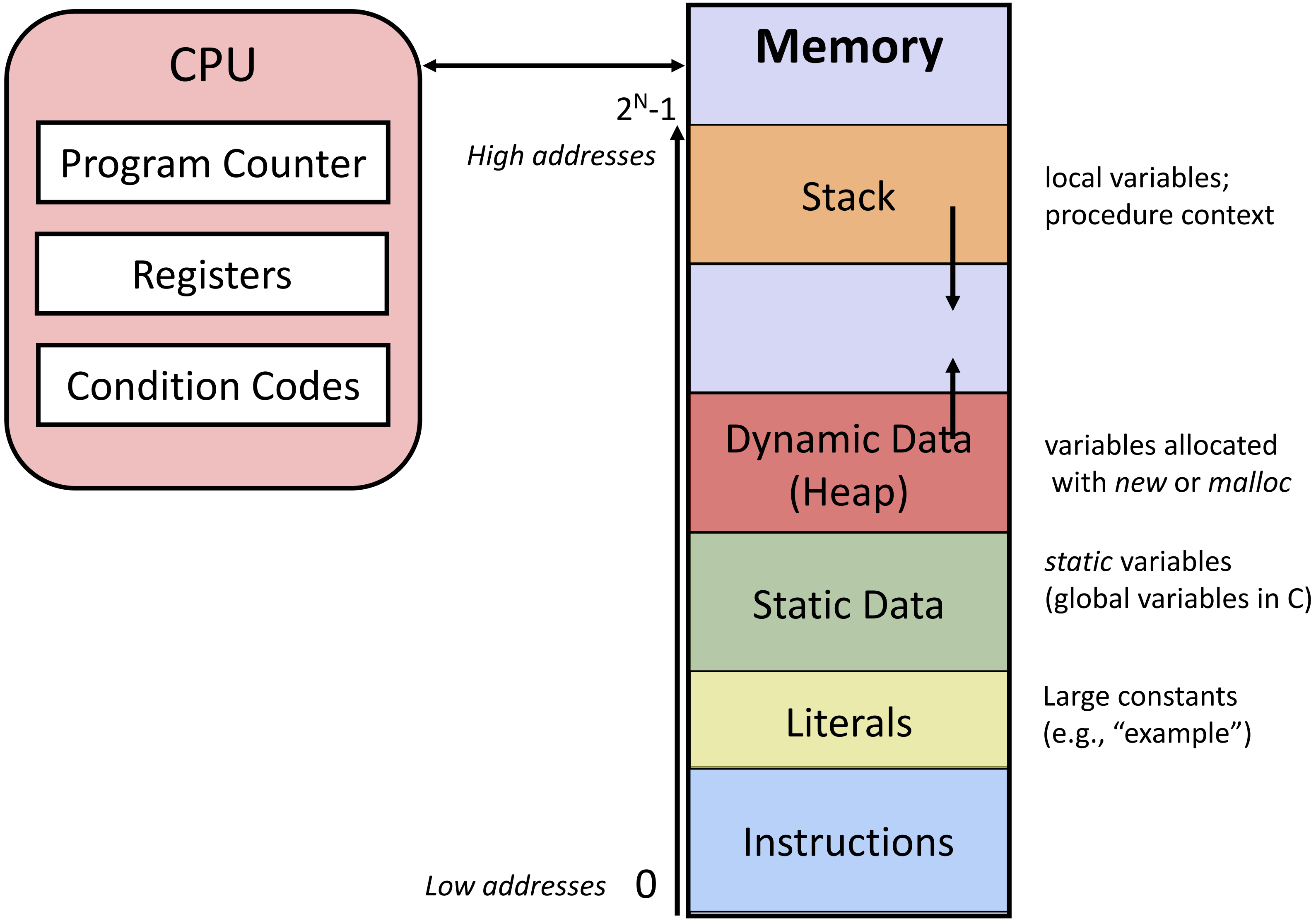


Assembly Programmer's View

- 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
    - Huge *virtual* address space that is *private*

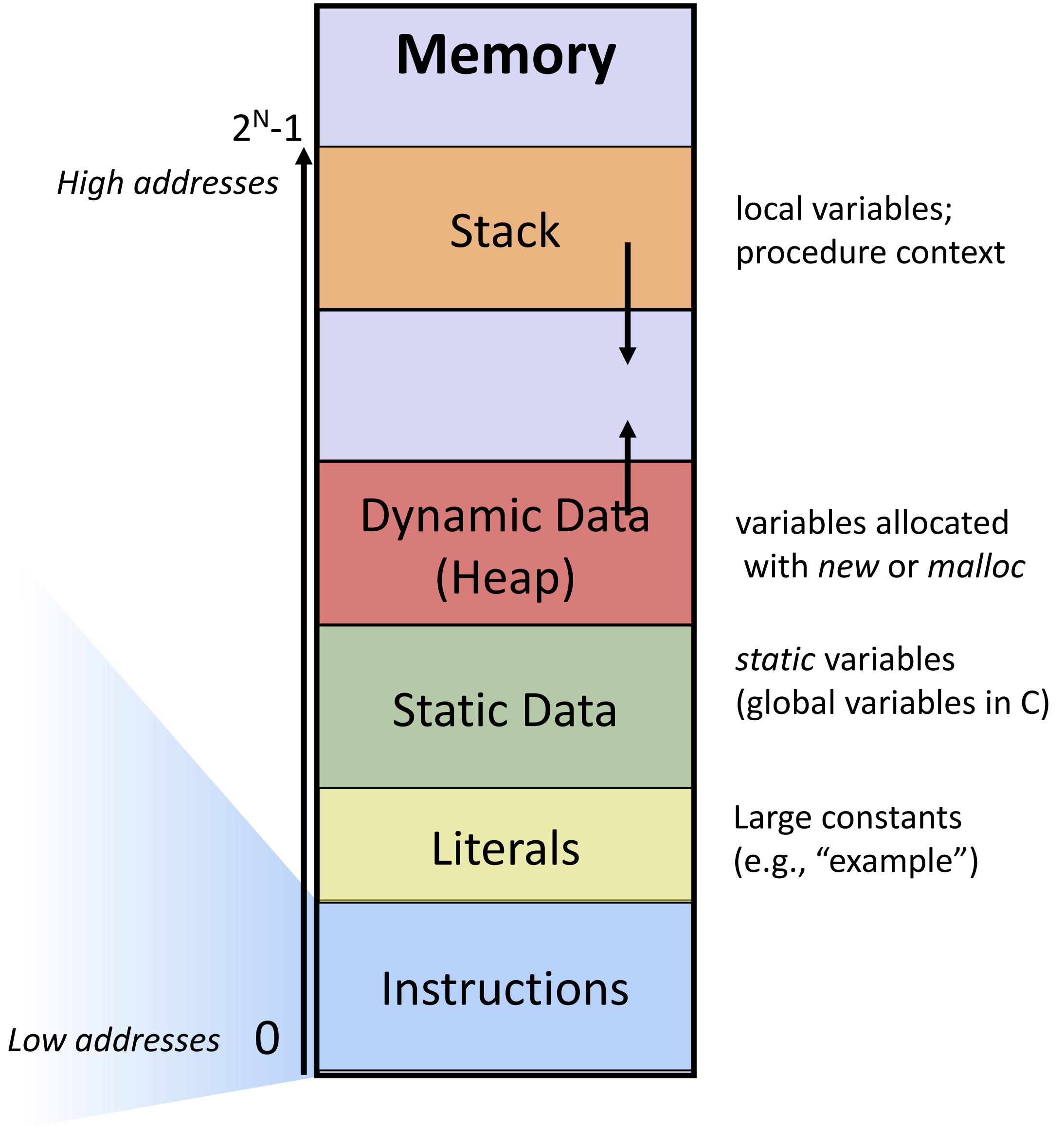


Program's View



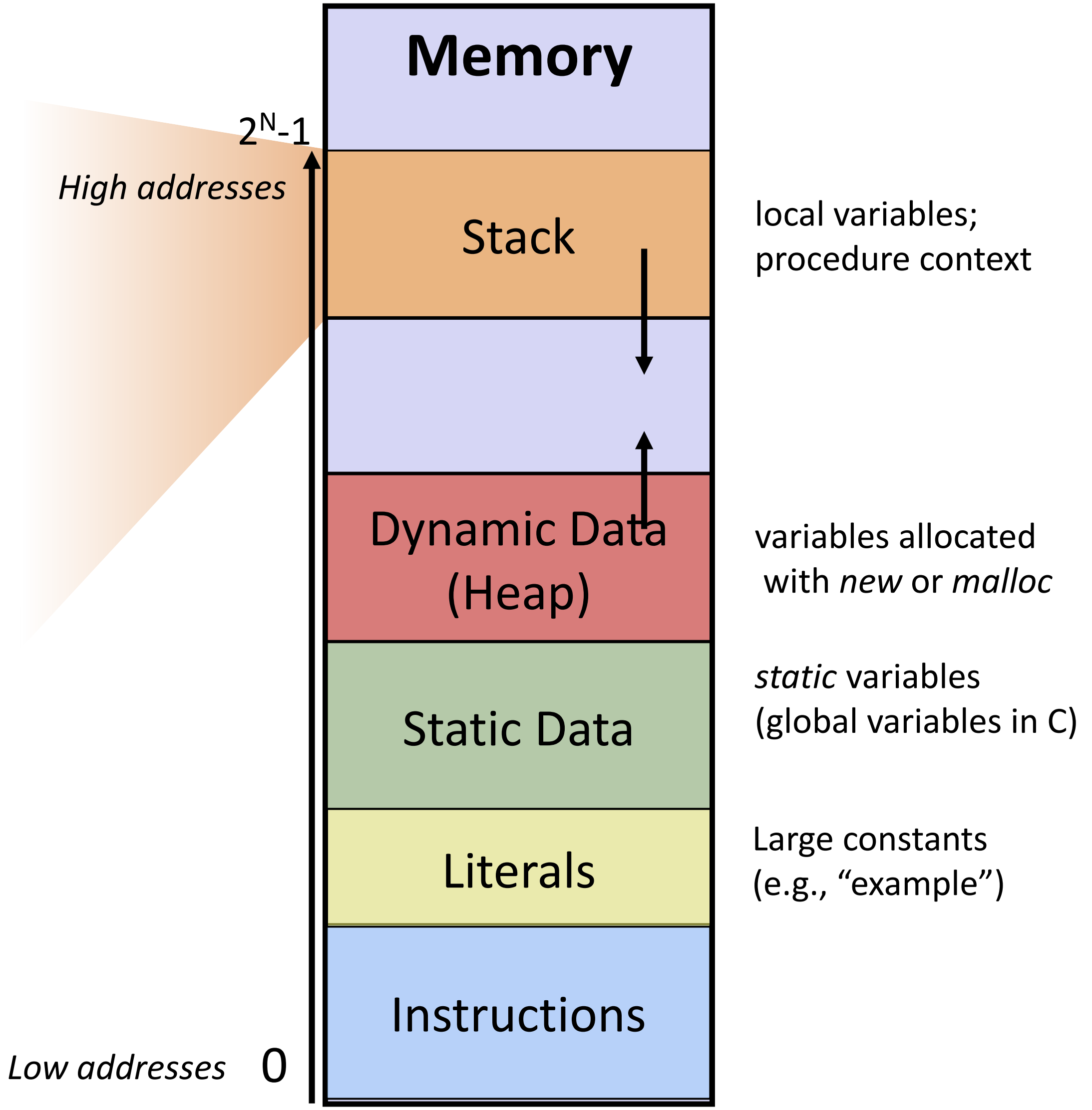
Program's View

- 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!*



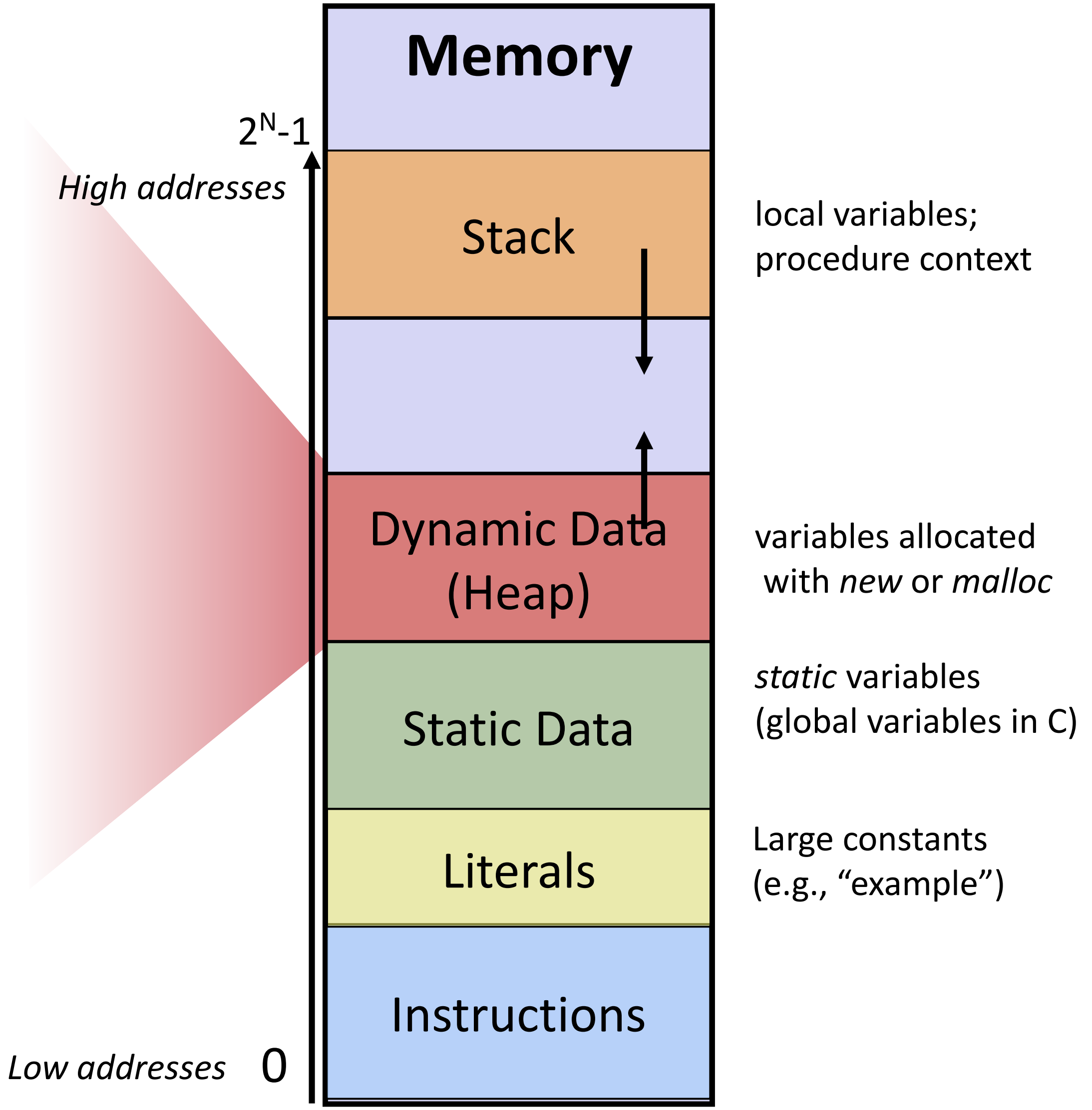
Program's View

- 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

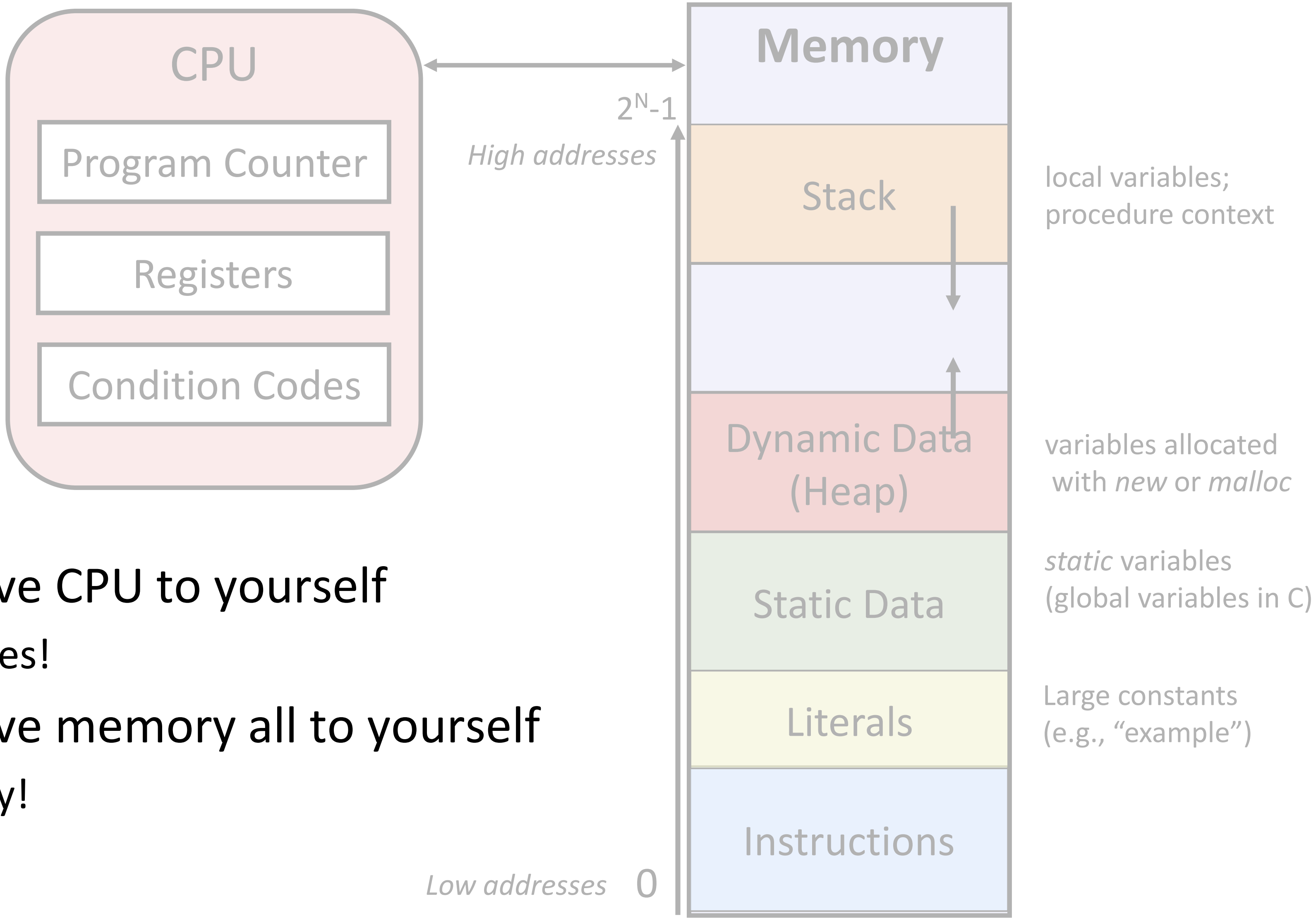


Program's View

- 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*



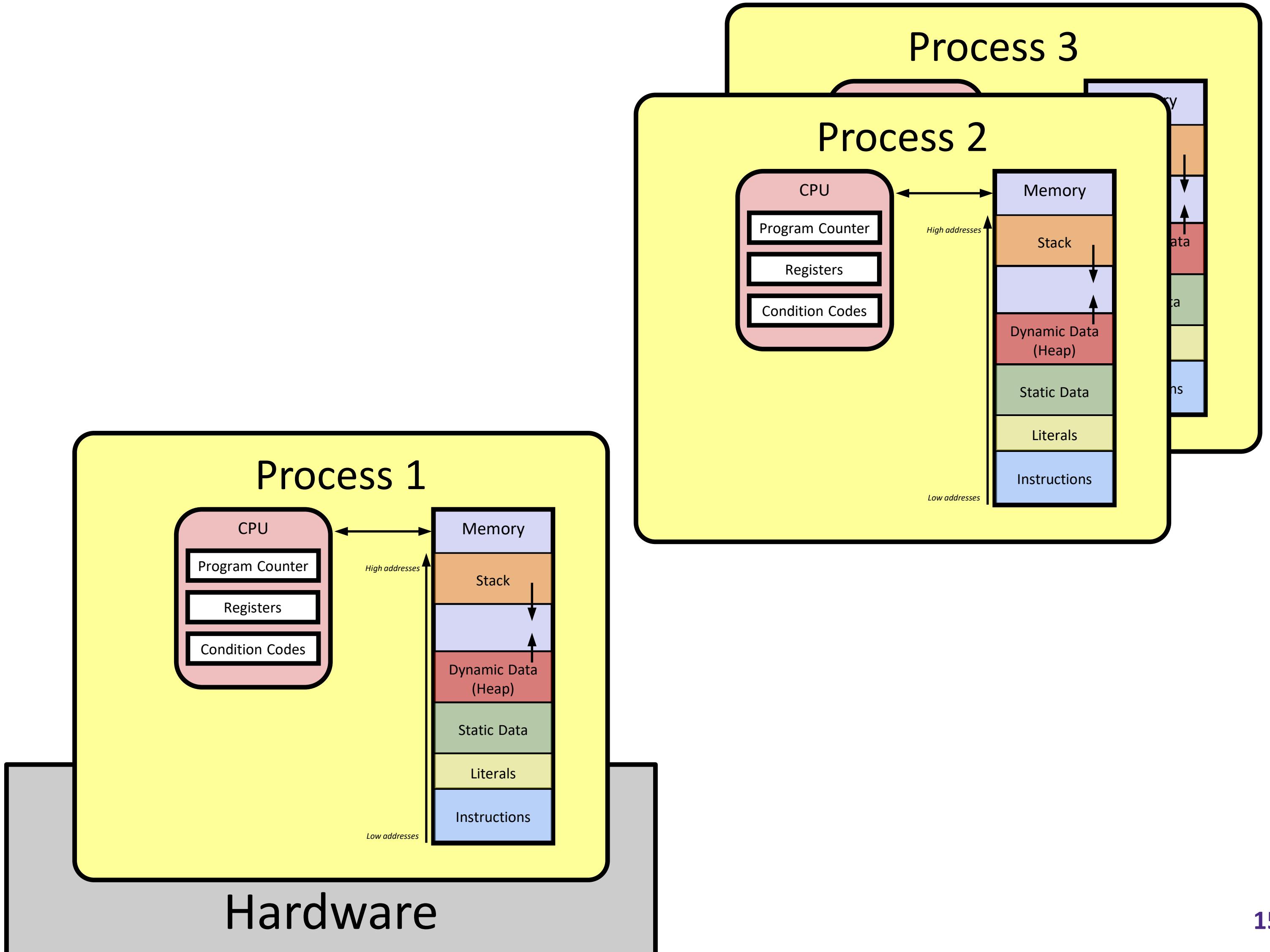
But remember… it's all an *illusion*!



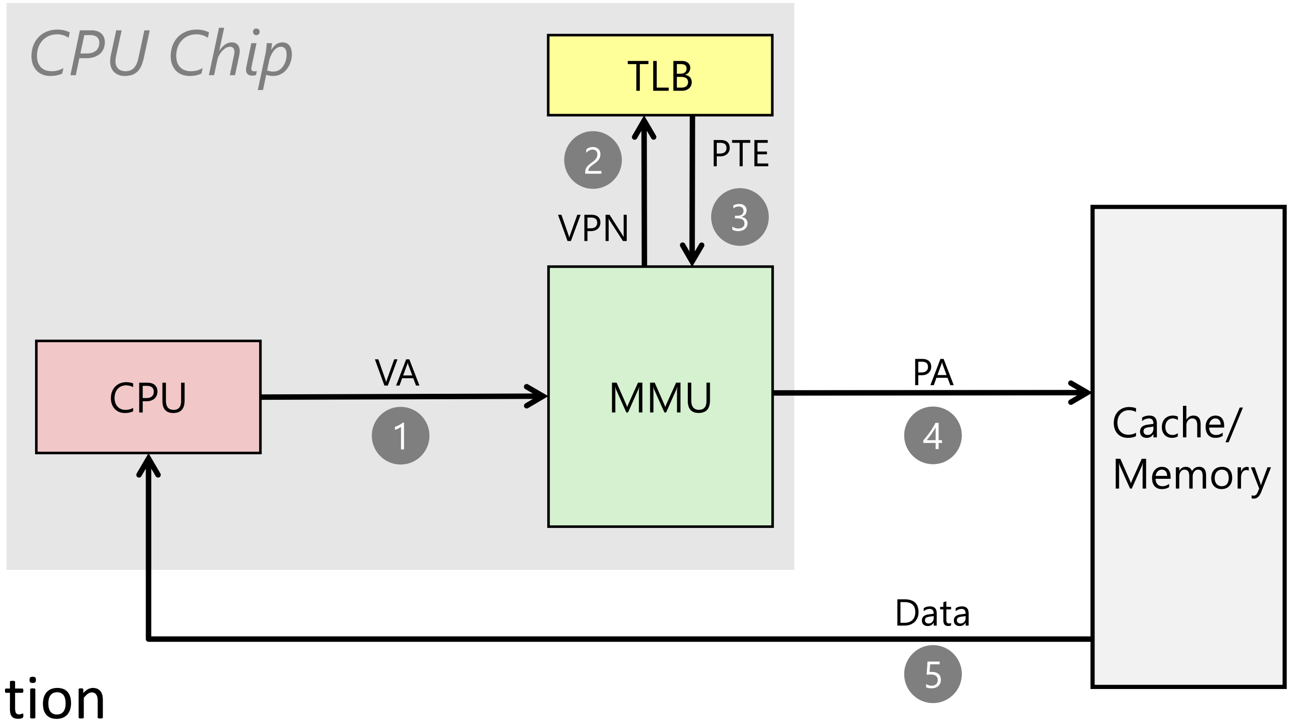
- Don't really have CPU to yourself
  - Context switches!
- Don't really have memory all to yourself
  - Virtual Memory!

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

- `fork`
  - Creates copy of the process
- `execv`
  - Replace with new program
- `wait`
  - Wait for child to terminate (to *reap* it and prevent *zombies*)

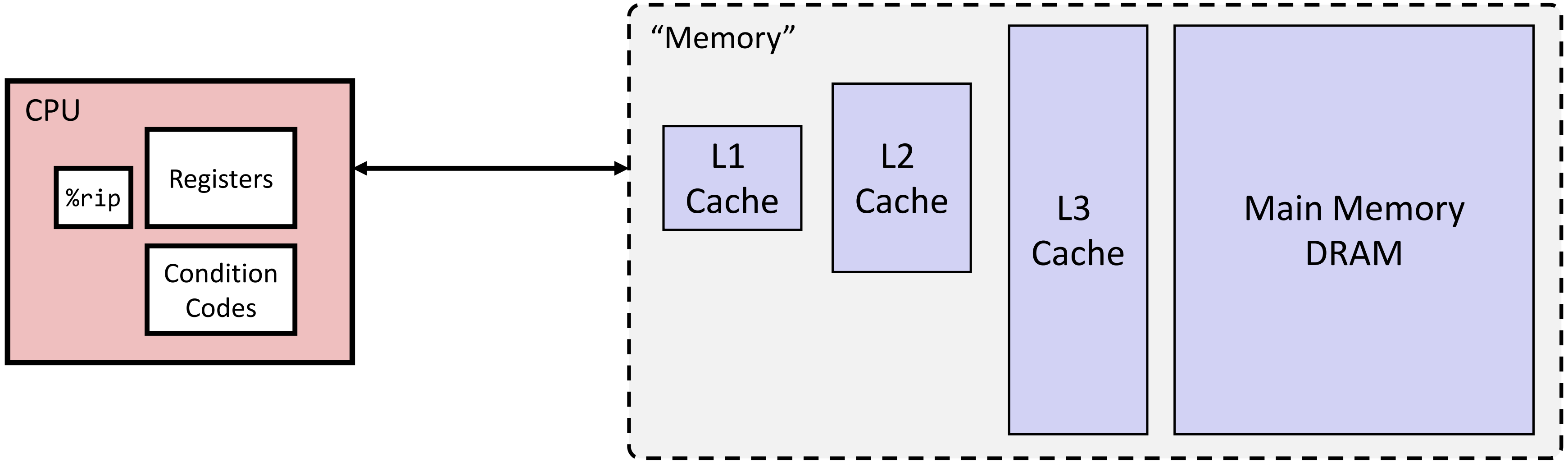


Virtual Memory



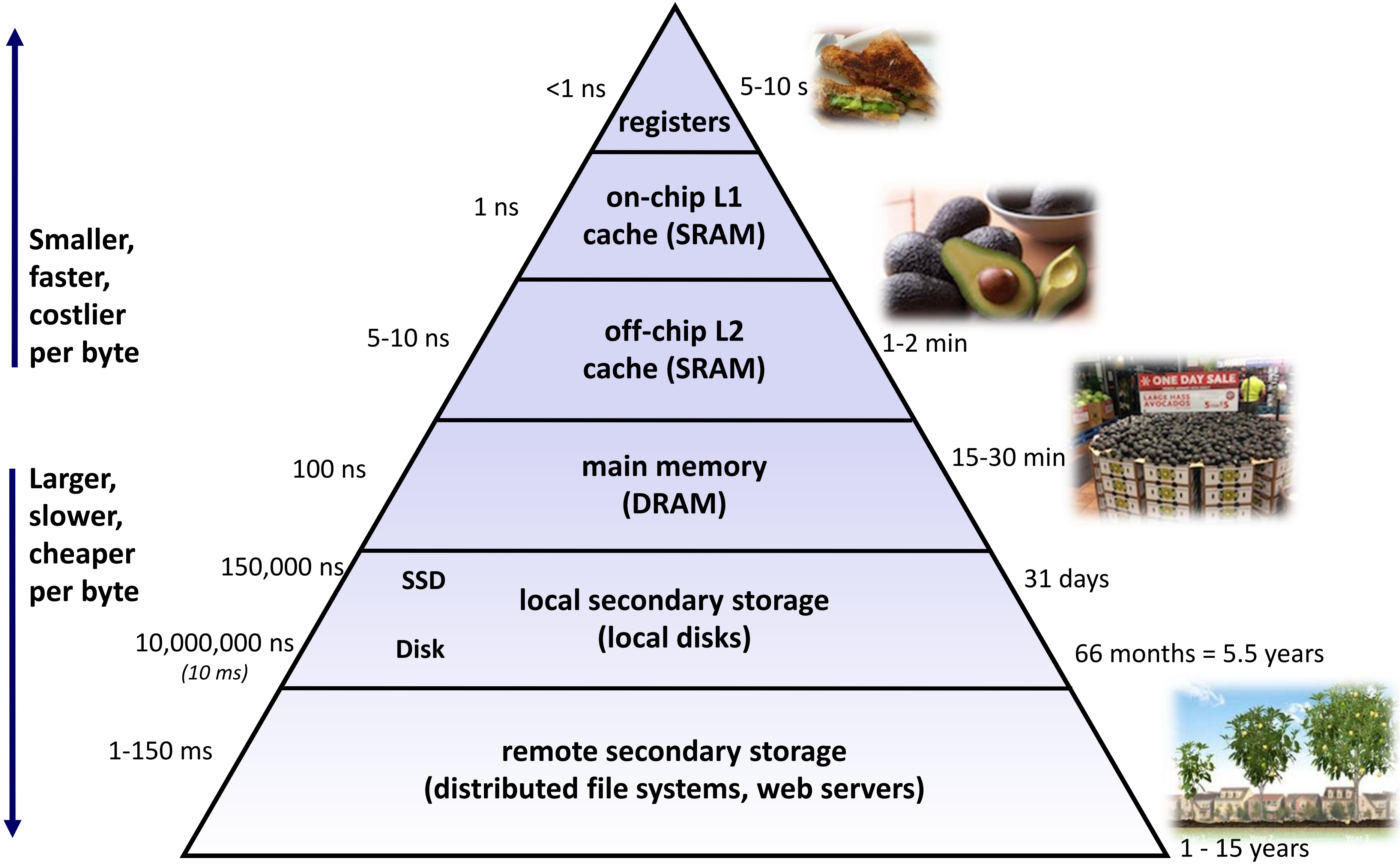
- Address Translation
  - Every memory access must first be converted from virtual to physical
  - *Indirection:* just change the address mapping when switching processes
  - Luckily, TLB (and page size) makes it pretty fast

But Memory is Also a Lie! 😮



- *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



Victory Lap

- High-level concepts (🔑 points) and course goals
  - Put everything into perspective
  - More useful for 5 years from now than the final

Big Theme 1: Abstractions and Interfaces

- Computing is about abstractions
  - (but we can't forget reality)
- What are the abstractions that we use?
- What do you 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/Encoding

- 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

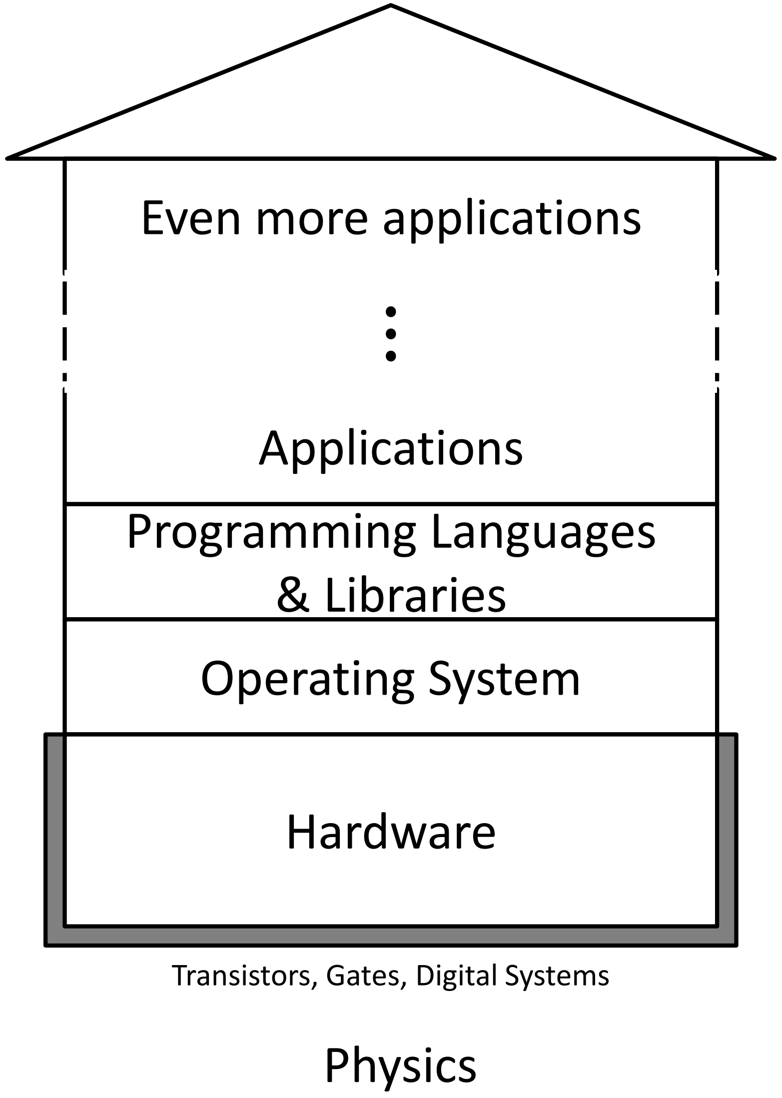
- We know Java as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We encountered C language, assembly language, and machine code (for the x86 family of CPU architectures)

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”?

Big Theme 2: Design Values

- Design choices are a combination of goals and context
  - Based on history and the society of the times
    - Usually assumptions about normativity or "common case"
  - Imbued with the values of the creators (and/or those with power)
    - Think critically about what you are told & sold!
- Nothing is future-proof
  - The House of Computing needs remodeling!
    - Built on the values of efficiency, profit, and militarism
  - Need to reexamine your heading and vision periodically
    - Check your metrics and definition of success



Course Perspective

- CSE351 will make you a more informed 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? (vote in Ed Lessons)

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

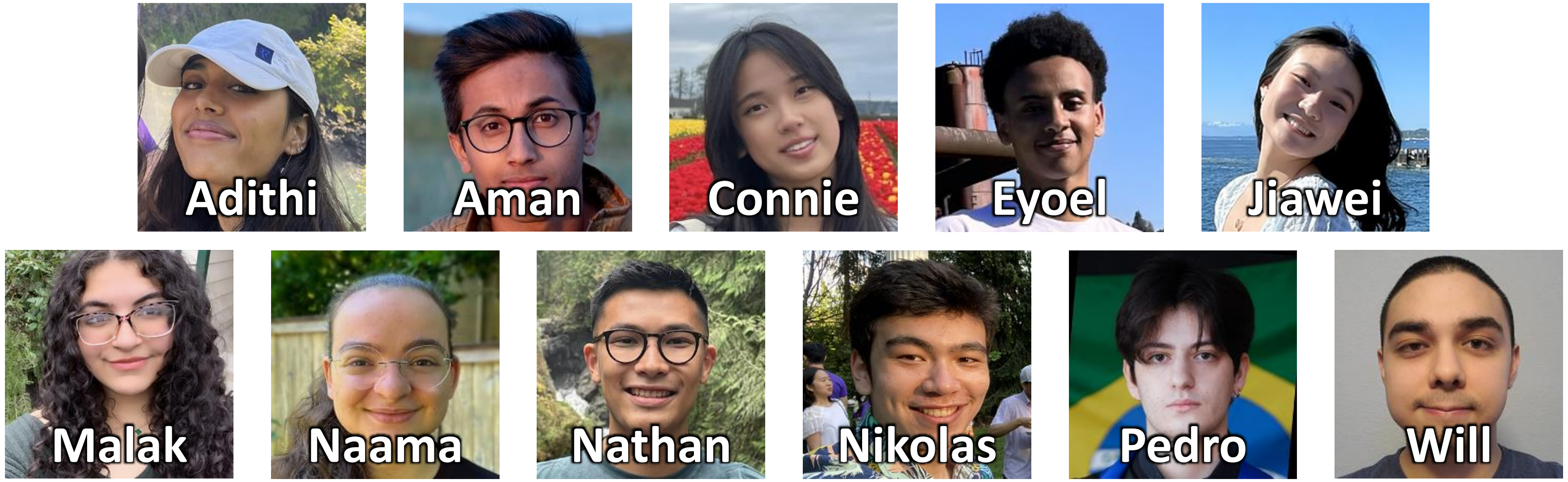
http://xkcd.com/676/

Courses: What's Next?

- Staying near the hardware/software interface:
  - **CSE369/EE271:** Digital Design – basic hardware design using FPGAs
  - **CSE474/EE474:** Embedded Systems – software design for microcontrollers
- Systems software
  - **CSE341/CSE413:** Programming Languages
  - **CSE332/CSE373:** Data Structures and Parallelism
  - **CSE333/CSE374:** 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)
  - **CSE484:** Computer Security (pre-reqs: 332, 351)

Thanks for a great quarter!

- Huge thanks to your awesome TAs!



- Don’t be a stranger!
  - If interested, I’m teaching CSE369 (Sp24), and EE/CSE371 (Sp24)
  - If you TA, I co-lead CSE General TA Training
  - I sometimes attend CSE590E: CS Education research seminar

Ask Me Anything


That's all Folks!