Memory, Data, & Addressing I
CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu

Arjun Narendra

Armin Magness

Assaf Vayner

Carrie Hu

Clare Edmonds
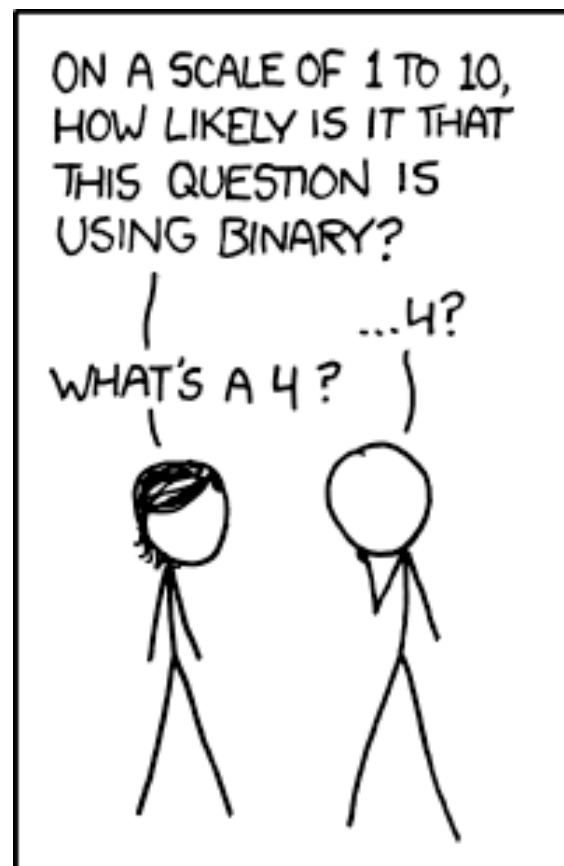
David Dai

Dominick Ta

Effie Zheng

James Froelich

Jenny Peng

Kristina Lansang

Paul Stevans

Renee Ruan

Vincent Xiao

http://xkcd.com/953/
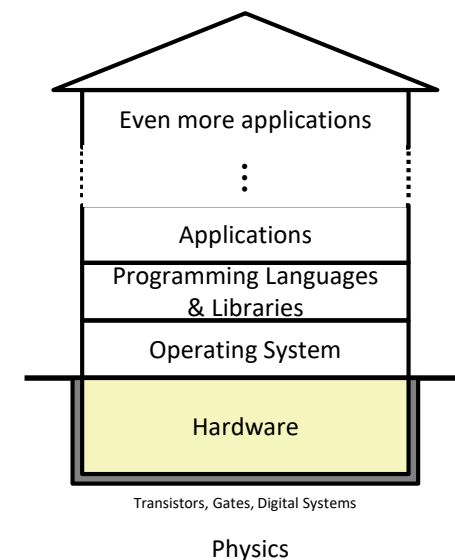
Relevant Course Information

- ❖ Everything not a reading or lecture lesson due @ 11:59 pm
  - Pre-Course Survey and HW0 due tonight
  - HW1 due Monday (10/3)
  - Lab 0 due Monday (10/3)
    - This lab is *exploratory* and looks like a HW; the other labs will look a lot different
- ❖ Ed Discussion etiquette
  - For anything that doesn't involve sensitive information or a solution, post publicly (you can post anonymously!)
  - If you feel like you question has been sufficiently answered, make sure that a response has a checkmark

EPA

- ❖ Encourage class-wide learning!

- ❖ Effort
  - Attending office hours, completing all assignments
  - Keeping up with Ed Discussion activity
- ❖ Participation
  - Making the class more interactive by asking questions in lecture, section, office hours, and on Ed Discussion
  - Lecture question voting
- ❖ Altruism
 - Helping others in section, office hours, and on Ed Discussion

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**
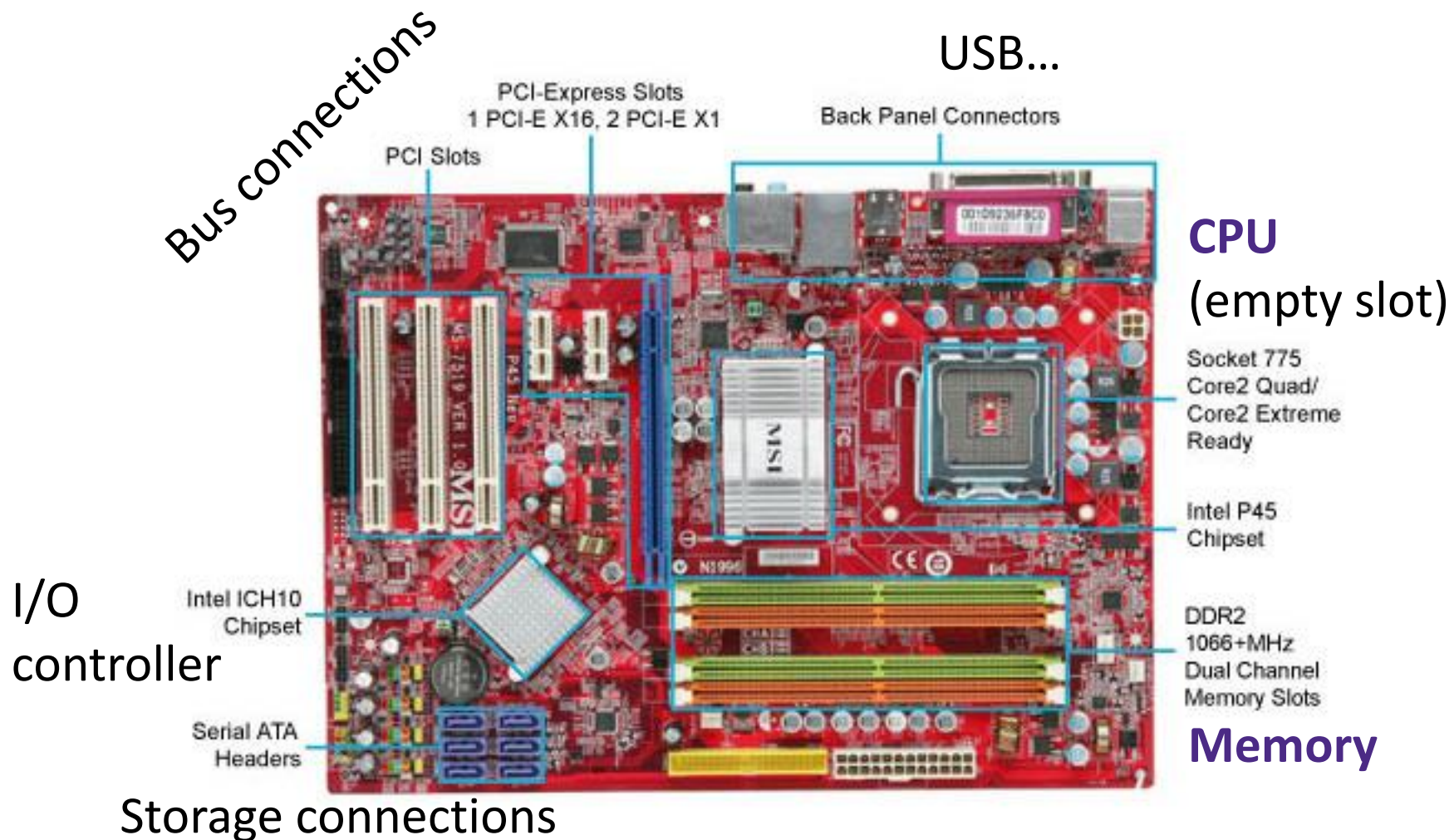
- Caches, Processes, Virtual Memory, Memory Allocation

Even more applications

⋮

Applications

Programming Languages & Libraries

Operating System

Hardware

Transistors, Gates, Digital Systems

Physics

The Hardware/Software Interface

❖ Topic Group 1: **Data**

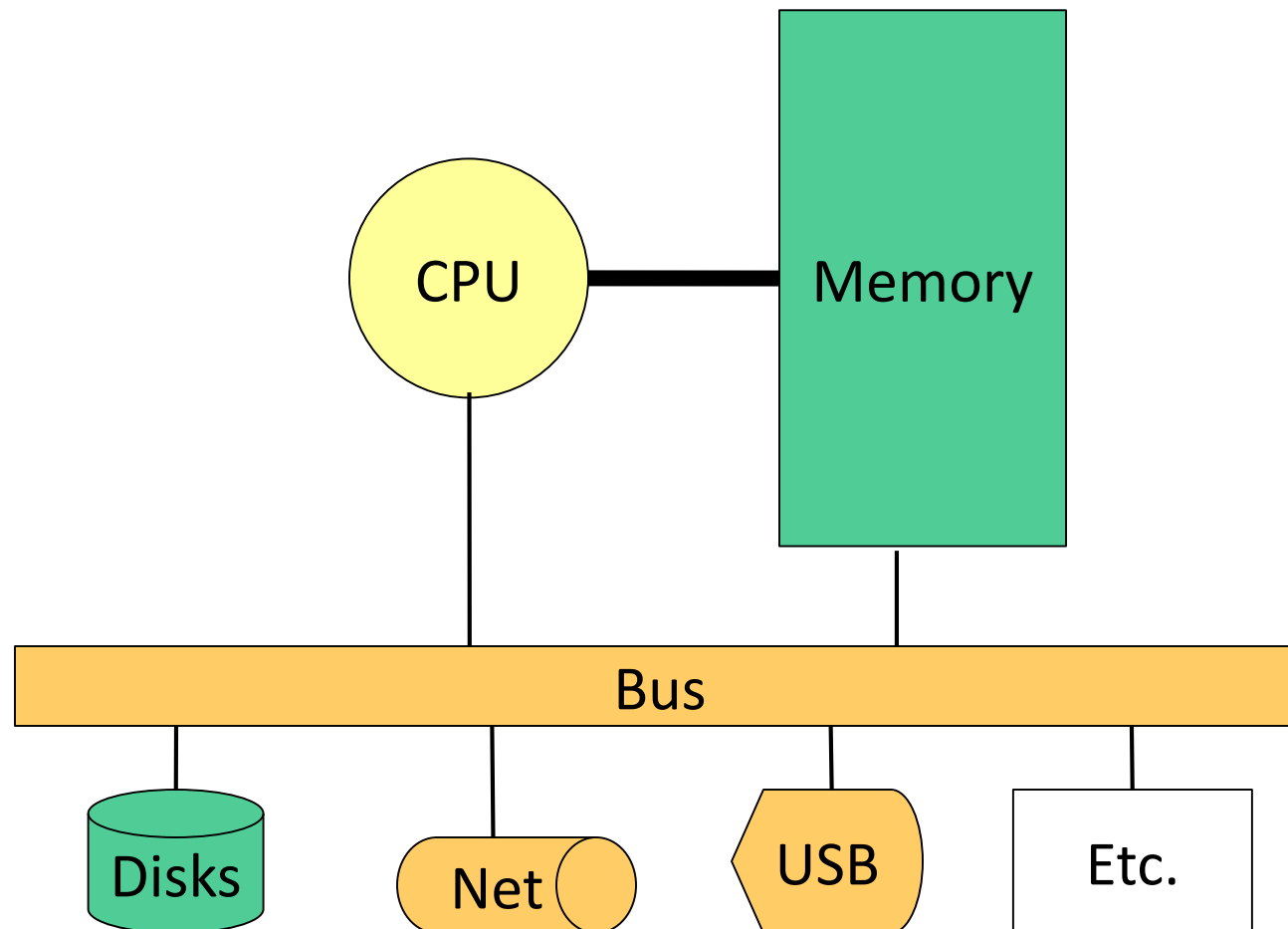- ▪ **Memory, Data**, Integers, Floating Point, Arrays, Structs
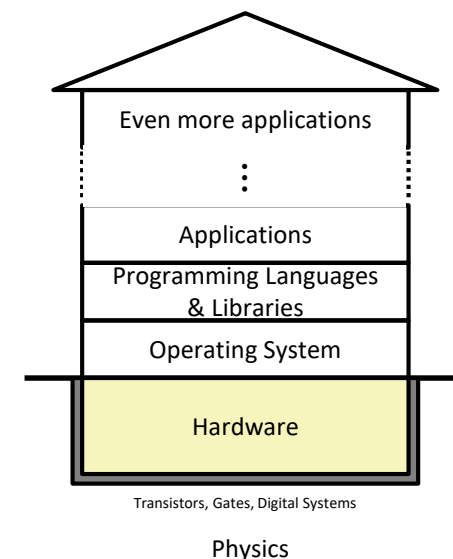
❖ How do we store information for other parts of the house of computing to access?

- ▪ How do we represent data and what limitations exist?
- ▪ What design decisions and priorities went into these encodings?

Hardware: Physical View

Bus connections

USB…

PCI-Express Slots
1 PCI-E X16, 2 PCI-E X1

PCI Slots

Back Panel Connectors

CPU
(empty slot)

Socket 775
Core2 Quad/
Core2 Extreme
Ready

Intel P45
Chipset

I/O controller

Intel ICH10
Chipset

DDR2
1066+MHz
Dual Channel
Memory Slots

Memory

Serial ATA
Headers

Storage connections
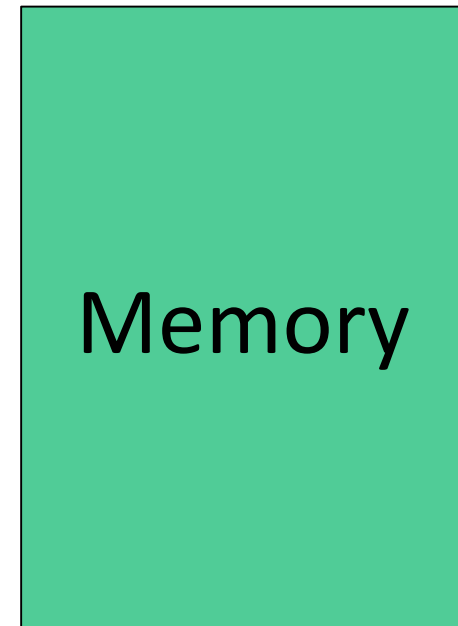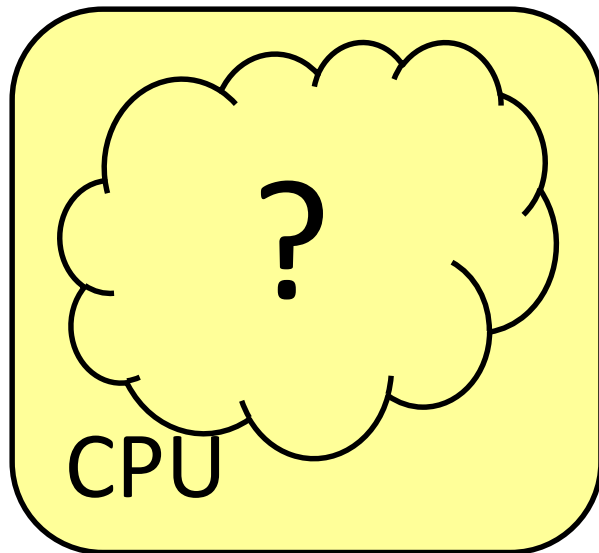
Hardware: Logical View

Hardware: 351 View (version 0)

- ❖ The CPU executes instructions
- ❖ Memory stores data

  Q1: How are data and instructions represented?

- ❖ Binary encoding!
  - ▪ Instructions *are* just data

Aside: Why Base 2?

❖ **Electronic implementation**

▪ Easy to store with bi-stable elements

▪ Reliably transmitted on noisy and inaccurate wires
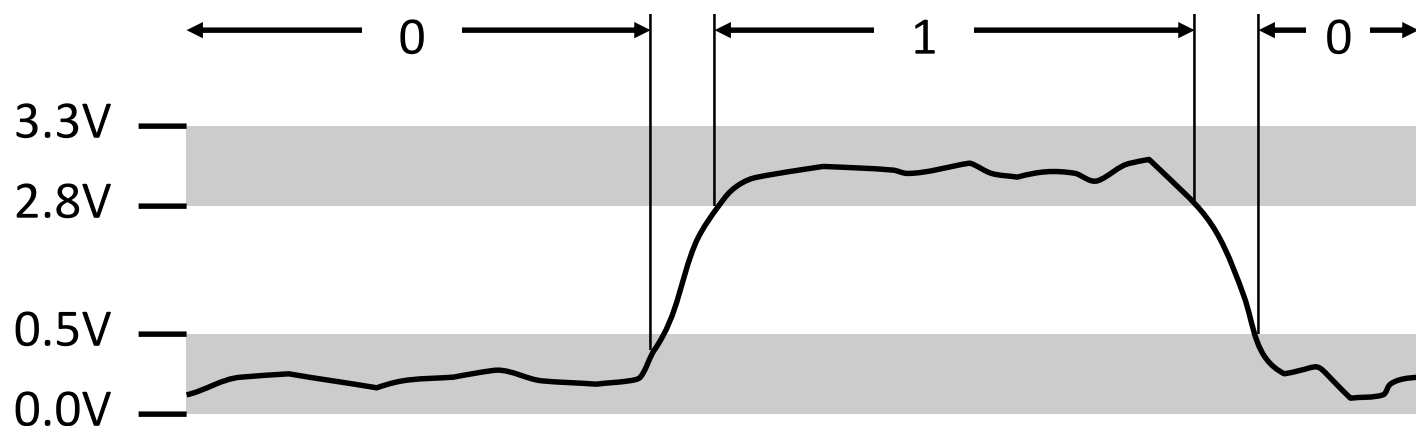
❖ **Other bases possible, but not yet viable:**

▪ DNA data storage (base 4: A, C, G, T) is hot @UW

▪ Quantum computing

Hardware: 351 View (version 0)
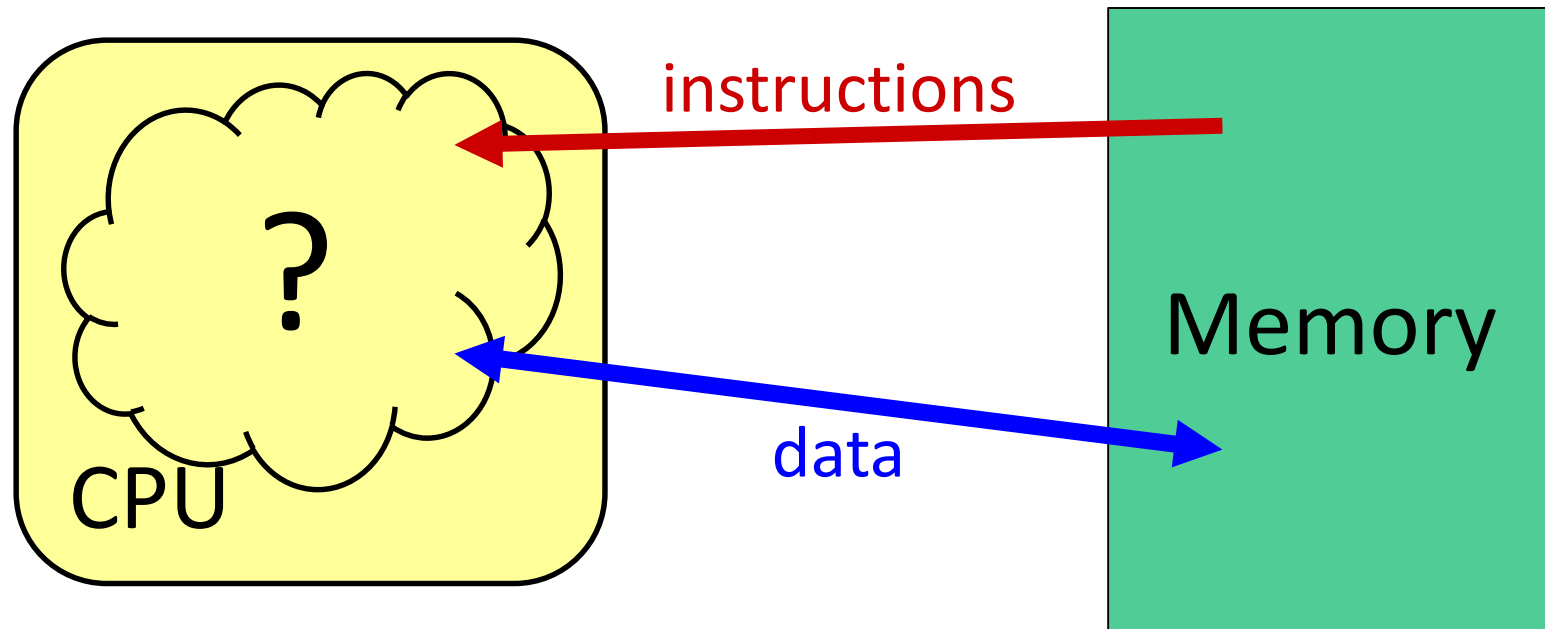
❖ To execute an instruction, the CPU must:
1) Fetch the instruction
2) (if applicable) Fetch data needed by the instruction
3) Perform the specified computation
4) (if applicable) Write the result back to memory

Hardware: 351 View (version 1)

This is extra (non-testable) material
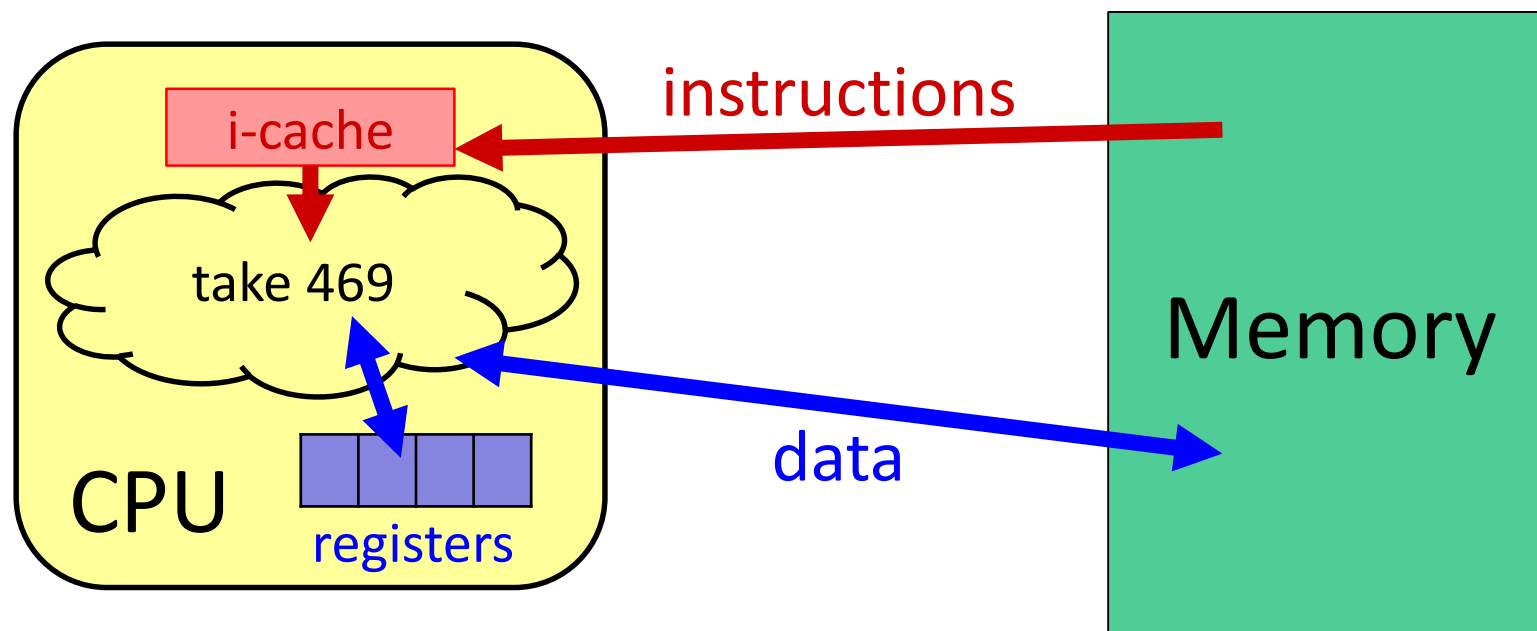
- ❖ More CPU details:
  - ▪ Instructions are held temporarily in the instruction cache
  - ▪ Other data are held temporarily in registers
- ❖ Instruction fetching is hardware-controlled
- ❖ Data movement is programmer-controlled (assembly)

11

Hardware: 351 View (version 1)
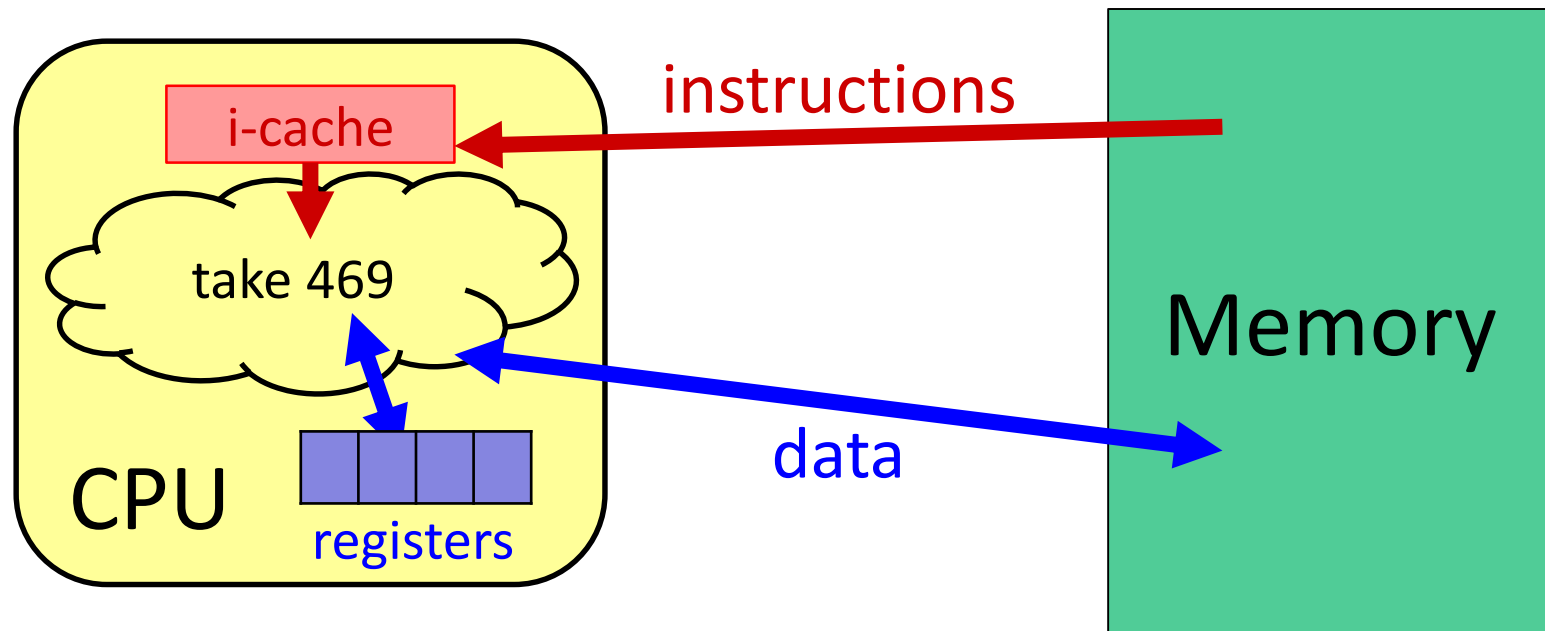
- ❖ We will start by learning about Memory

> **Q2:** How does a program find its data in memory?

- ❖ Addresses!
  - Can be stored in *pointers*

Reading Review

❖ Terminology:
 ▪ word size, byte-oriented memory
 ▪ address, address space
 ▪ most-significant bit (MSB), least-significant bit (LSB)
 ▪ big-endian, little-endian
 ▪ pointer

❖ Questions from the Reading?

Review Questions

- ❖ By looking at the bits stored in memory, I can tell what a particular 4 bytes is being used to represent.

  A. True   **B. False**

- ❖ We can fetch a piece of data from memory as long as we have its address.

  A. True   **B. False**

- ❖ Which of the following bytes have a most-significant bit (MSB) of 1?

  A. 0x63   **B. 0x90**   **C. 0xCA**   **D. 0xF**

Fixed-Length Binary (Review)

❖ Because storage is finite in reality, everything is stored as "fixed" length

- Data is moved and manipulated in fixed-length chunks
- Multiple fixed lengths (*e.g.*, 1 byte, 4 bytes, 8 bytes)
- Leading zeros now *must* be included up to "fill out" the fixed length

❖ <u>Example</u>: the "eight-bit" representation of the number 4 is 0b00000100

Least Significant Bit (LSB)

Most Significant Bit (MSB)

Bits and Bytes and Things (Review)

❖ 1 byte = 8 bits

❖ n bits can represent up to 2^n things

 ▪ Sometimes (oftentimes?) those "things" are bytes!

❖ If an addresses are a-bits wide, how many distinct addresses are there?

❖ What does each address refer to?
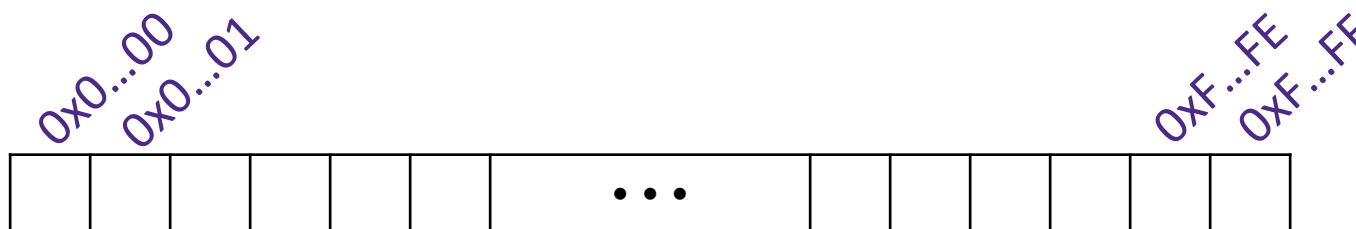
Machine "Words" (Review)

- ❖ Instructions encoded into machine code (0's and 1's)
  - ▪ Historically (still true in some assembly languages), all instructions were exactly the size of a word

- ❖ We have *chosen* to tie word size to address size/width
  - ▪ word size = address size = register size
  - ▪ word size = w bits → 2^w addresses

- ❖ Current x86 systems use **64-bit (8-byte) words**
  - ▪ Potential address space: 2^{64} addresses
    2^{64} bytes ≈ **1.8 x 10^{19} bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - ▪ Actual physical address space: **48 bits**

Data Representations

❖ Sizes of data types (in bytes)

Java Data Type	C Data Type	IA-32 (old)	x86-64
`boolean`	`bool`	1	1
`byte`	`char`	1	1
`char`		2	2
`short`	`short int`	2	2
`int`	`int`	4	4
`float`	`float`	4	4
	`long int`	4	8
`double`	`double`	8	8
`long`	`long long`	8	8
	`long double`	8	16
(reference)	**pointer ***	**4**	**8**

address size = word size

To use "`bool`" in C, you must `#include <stdbool.h>`

Discussion Question

❖ Over time, computers have grown in word size:

Word size	Instruction Set Architecture	First? Intel CPU	Year Introduced
8-bit	??? (Poor & Pyle)	Intel 8008	1972
16-bit	x86	Intel 8086	1978
32-bit	IA-32	Intel 386	1985
64-bit	IA-64	Itanium (Merced)	2001
64-bit	x86-64	Xeon (Nocona)	2004

▪ What do you think were some of the *causes*, *advantages*, and *disadvantages* of this trend?

Address of Multibyte Data (Review)

❖ Addresses still specify locations of <u>bytes</u> in memory, but we can choose to *view* memory as a series of <u>chunks</u> of fixed-sized data instead

  ▪ Addresses of successive chunks differ by data size

  ▪ Which byte's address should we use for each word?

❖ The address of *any* chunk of memory is given by the address of the first byte

  ▪ To specify a chunk of memory, need *both* its **address** and its **size**

64-bit data	32-bit data	Bytes	Addr. (hex)
	Addr = 0000		0x00
			0x01
			0x02
Addr = 0000			0x03
	Addr = 0004		0x04
			0x05
			0x06
			0x07
	Addr = 0008		0x08
			0x09
			0x0A
Addr = 0008			0x0B
	Addr = 0012		0x0C
			0x0D
			0x0E
			0x0F

20

A Picture of Memory (64-bit view)

❖ A "64-bit (8-byte) word-aligned" <u>view</u> of memory:

- In this type of picture, each row is composed of 8 bytes
- Each cell is a byte
- An aligned, 64-bit chunk of data will fit on one row

one word

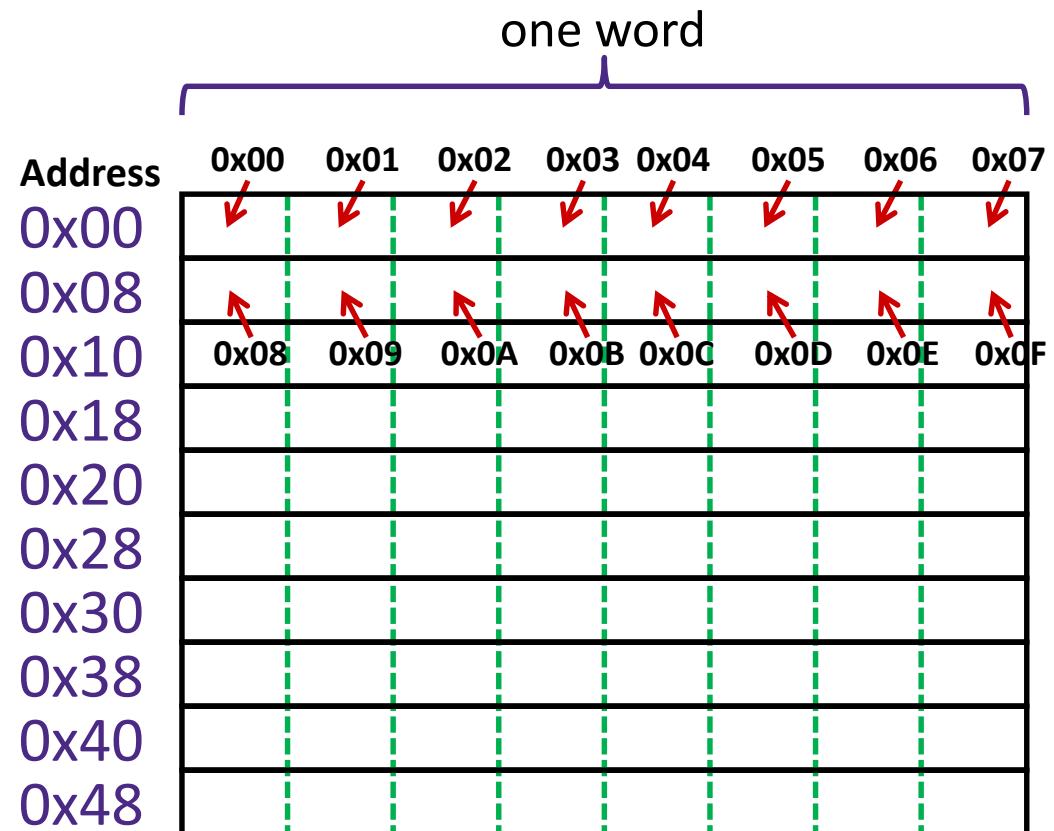| Address	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00								
0x08								
0x10								
0x18								
0x20								
0x28								
0x30								
0x38								
0x40								
0x48								

A Picture of Memory (64-bit view)

❖ A "64-bit (8-byte) word-aligned" <u>view</u> of memory:

▪ In this type of picture, each row is composed of 8 bytes

▪ Each cell is a byte

▪ An aligned, 64-bit chunk of data will fit on one row

one word

Address	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00								
0x08	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0x10								
0x18								
0x20								
0x28								
0x30								
0x38								
0x40								
0x48								

Addresses and Pointers

❖ An *address* refers to a location in memory

❖ A *pointer* is a data object that holds an address

- Address can point to *any* data

❖ Value 504 stored as a word at addr 0x08

- $504_{10} = 1F8_{16}$
  $= 0x\ 00\ ...\ 00\ 01\ F8$

❖ Pointer stored at 0x38 points to address 0x08

Address

0x00							
0x08	00	00	00	00	00	00	01	F8
0x10							
0x18							
0x20							
0x28							
0x30							
0x38	00	00	00	00	00	00	00	08
0x40							
0x48							

Addresses and Pointers

64-bit example
(pointers are 64-bits wide)

big-endian

❖ An *address* refers to a location in memory

❖ A *pointer* is a data object that holds an address
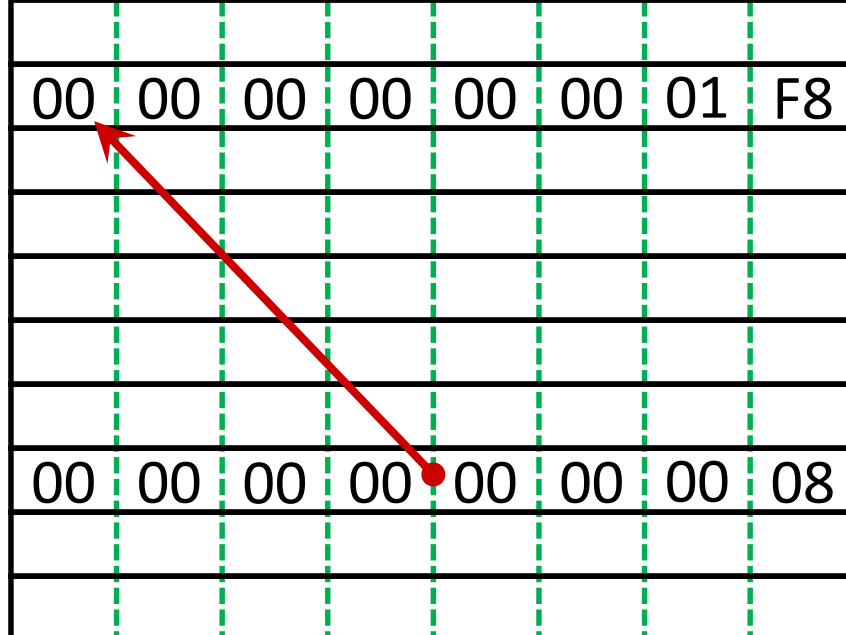
  ▪ Address can point to *any* data

❖ Pointer stored at 0x48 points to address 0x38

  ▪ Pointer to a pointer!

❖ Is the data stored at 0x08 a pointer?
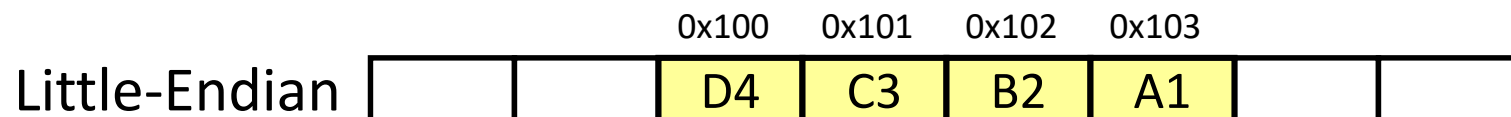
  ▪ Could be, depending on how you use it

Address

0x00							
0x08	00	00	00	00	00	00	01	F8
0x10							
0x18							
0x20							
0x28							
0x30							
0x38	00	00	00	00	00	00	00	08
0x40							
0x48	00	00	00	00	00	00	00	38

24

Byte Ordering (Review)

❖ How should bytes within a word be ordered *in memory?*

  ▪ Want to keep consecutive bytes in consecutive addresses

  ▪ **Example:** store the 4-byte (32-bit) `int`:
    `0x A1 B2 C3 D4`

❖ By convention, ordering of bytes called *endianness*

  ▪ The two options are big-endian and little-endian

    • In which address does the least significant *byte* go?

    • Based on *Gulliver's Travels*: tribes cut eggs on different sides (big, little)

Byte Ordering

❖ Big-endian (SPARC, z/Architecture)

  ▪ Least significant byte has highest address

❖ Little-endian (x86, x86-64)

  ▪ Least significant byte has lowest address

❖ Bi-endian (ARM, PowerPC)

  ▪ Endianness can be specified as big or little

❖ **Example:** 4-byte data 0xA1B2C3D4 at address 0x100

		0x100	0x101	0x102	0x103		
Big-Endian		A1	B2	C3	D4		

		0x100	0x101	0x102	0x103		
Little-Endian		D4	C3	B2	A1		
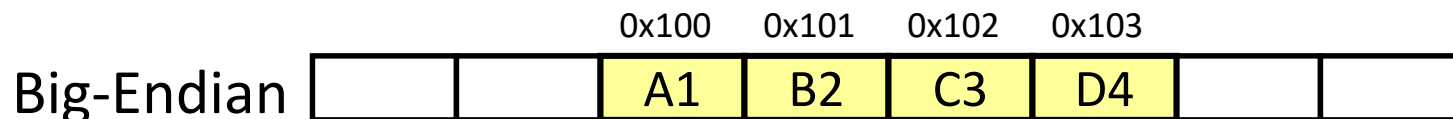
Polling Question

❖ We store the value `0x 01 02 03 04` as a **word** at address 0x100 in a big-endian, 64-bit machine

❖ What is the **byte of data** stored at address 0x104?

   ▪ Vote in Ed Lessons

   A. 0x04

   B. 0x40

   C. 0x01

   D. 0x10

   E. We're lost…

Endianness

❖ *Endianness only applies to memory storage*

❖ Often programmer can ignore endianness because it is handled for you

  ▪ Bytes wired into correct place when reading or storing from memory (hardware)

  ▪ Compiler and assembler generate correct behavior (software)

❖ Endianness still shows up:

  ▪ Logical issues: accessing different amount of data than how you stored it (*e.g.,* store `int`, access byte as a `char`)

  ▪ Need to know exact values to debug memory errors

  ▪ Manual translation to and from machine code (in 351)

Summary

- ❖ Memory is a long, *byte-addressed* array
  - ▪ Word size bounds the size of the *address space* and memory
  - ▪ Different data types use different number of bytes
  - ▪ Address of chunk of memory given by address of lowest byte in chunk
- ❖ Pointers are data objects that hold addresses
  - ▪ Type of pointer determines size of thing being pointed at, which could be another pointer
- ❖ Endianness determines memory storage order for multi-byte data
  - ▪ Least significant byte in lowest (little-endian) or highest (big-endian) address of memory chunk