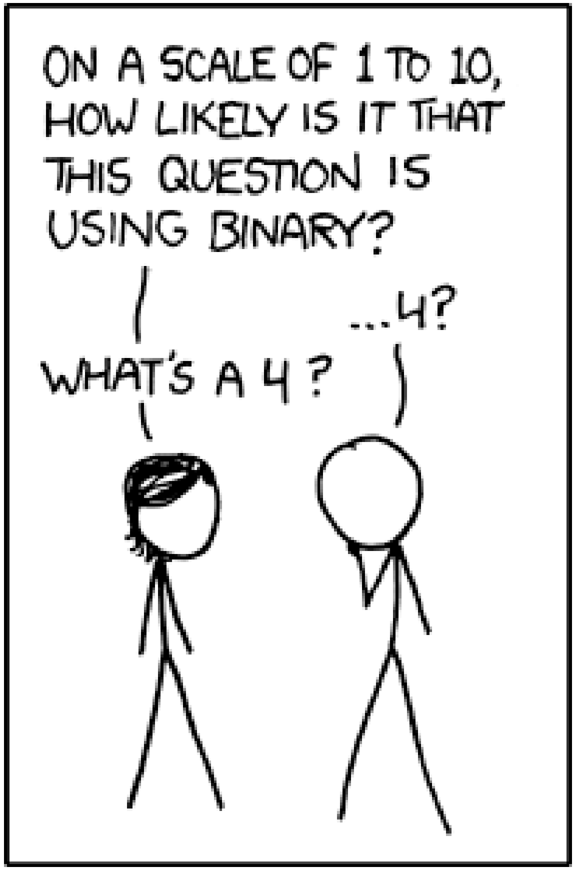
Memory, Data, & Addressing I

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



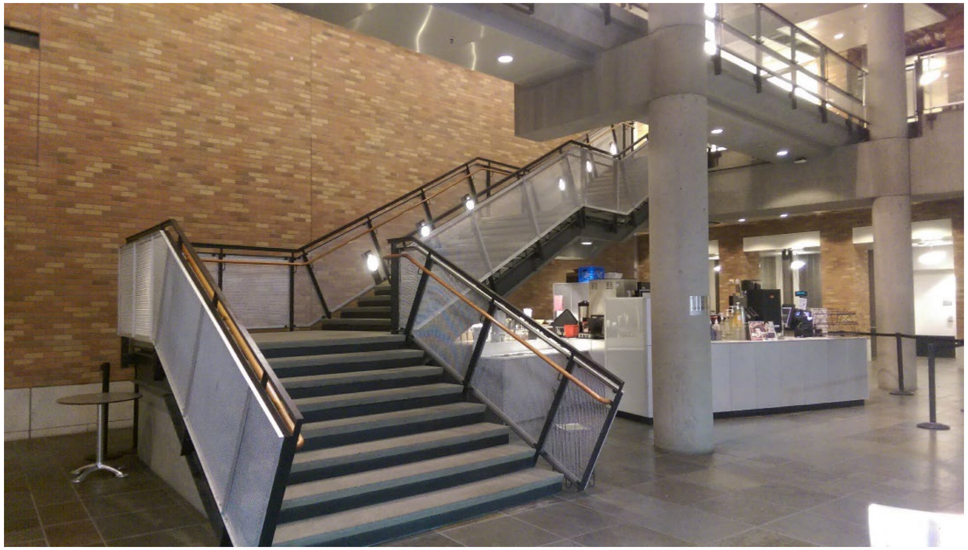
http://xkcd.com/953/

Relevant Course Information

- Pre-Course Survey and HW0 due tonight @ 11:59 pm
- HW1 due Monday (9/30) @ 11:59 pm
- Lab 0 due Monday (9/30) @ 11:59 pm
  - This lab is *exploratory* and looks like a hw; the other labs will look a lot different (involve writing code etc.)
  - Don’t worry if everything in Lab 0 doesn’t make perfect sense right now! We will cover all of these topics in more detail later in the course.
  - Lab 0 is about getting you used to modifying C code and running it to see what the outcome is – a powerful tool for understanding the concepts in this course!
- Readings should be completed by 11am on day of lecture
- Lecture activities should be completed by 11am of NEXT lecture

In-Person Office Hours

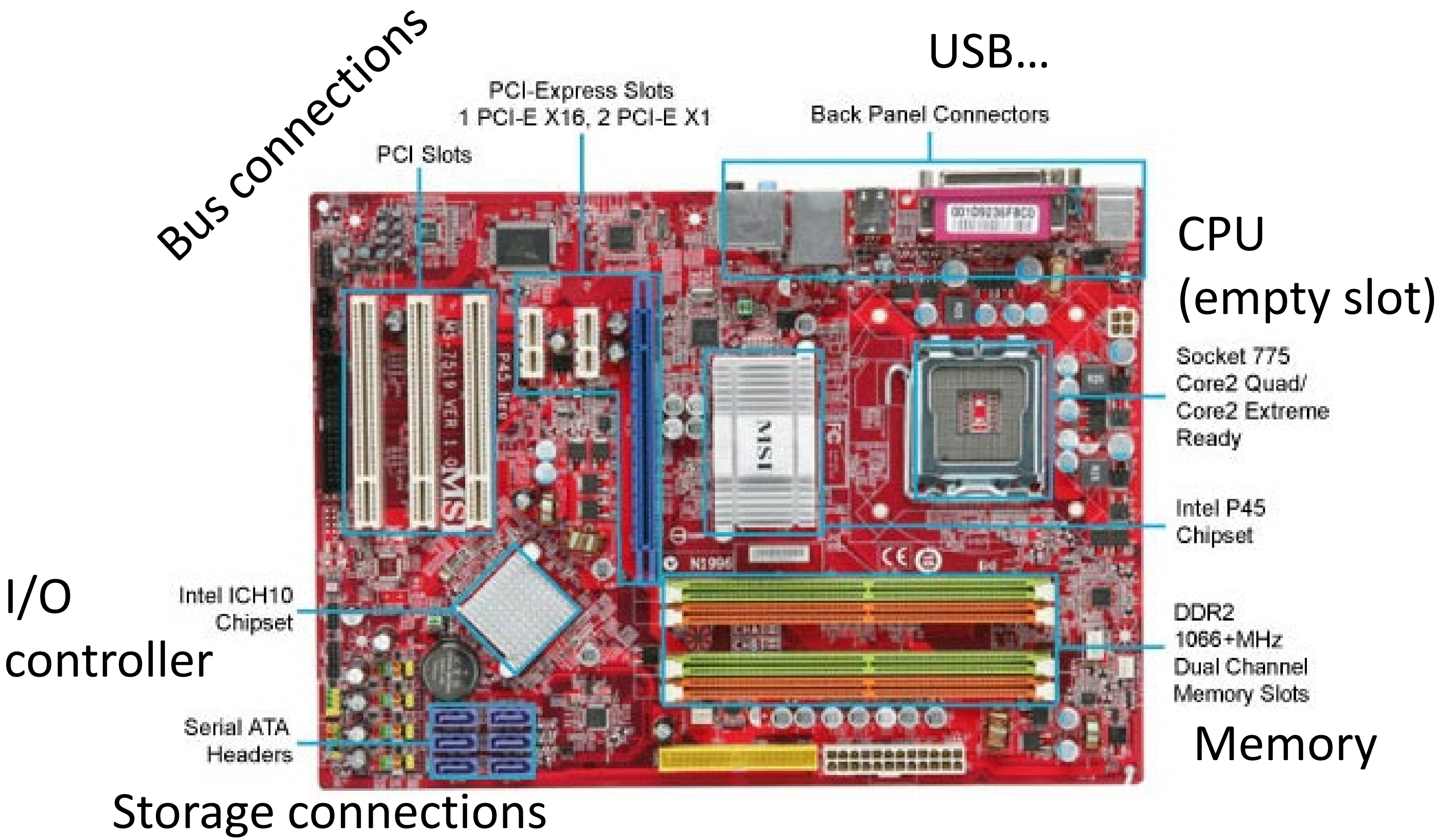
- Many are in the CSE/Allen Center breakouts
  - Up the stairs in the CSE Atrium (Allen Center, not Gates)
  - 2nd, 3rd , 4th, 5th floors
  - At the top of the stairs, the open area with the whiteboard wall is a breakout!



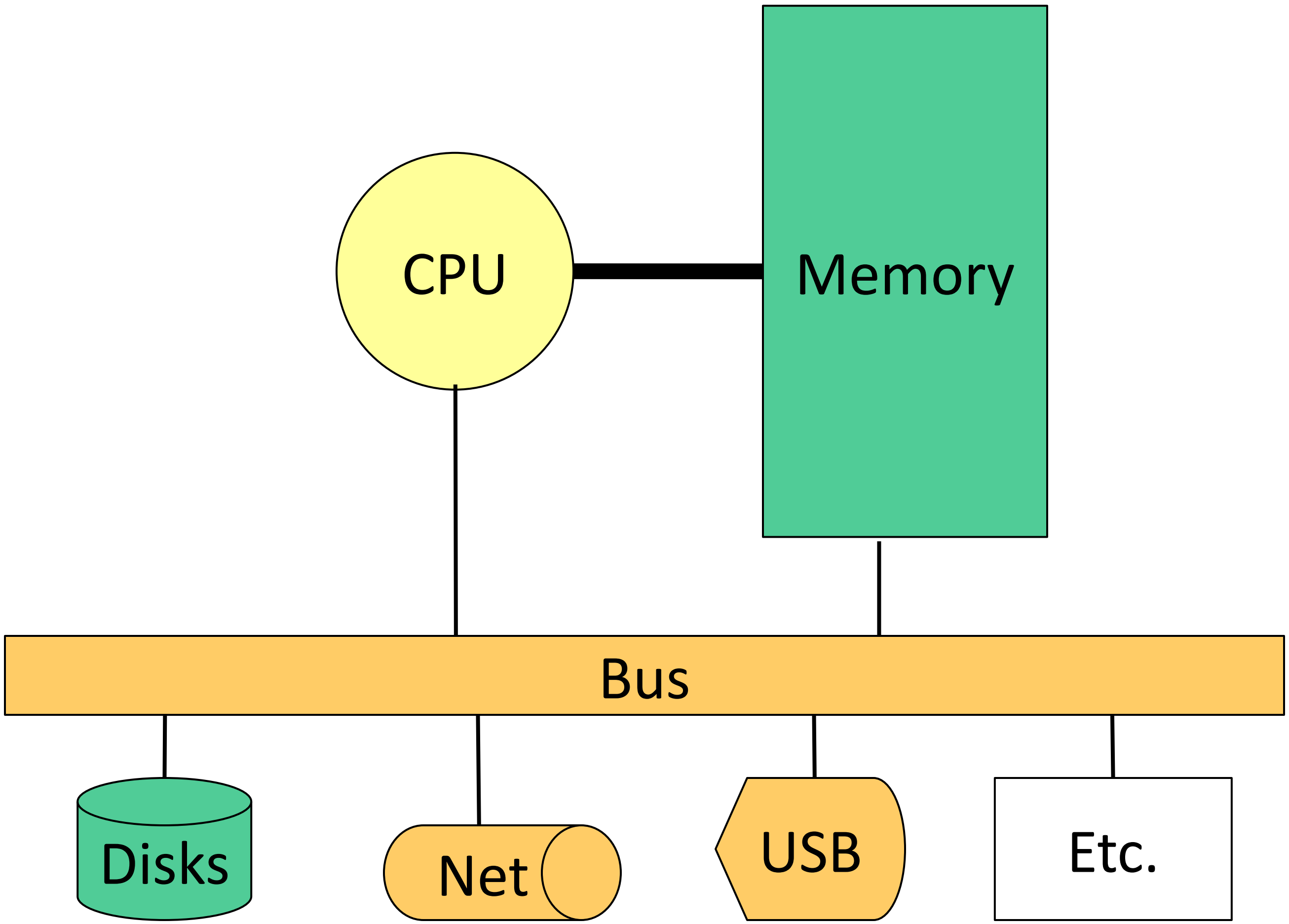
Memory, Data, and Addressing

- Hardware - High Level Overview
- Representing information as bits and bytes
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
- Organizing and addressing data in memory
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

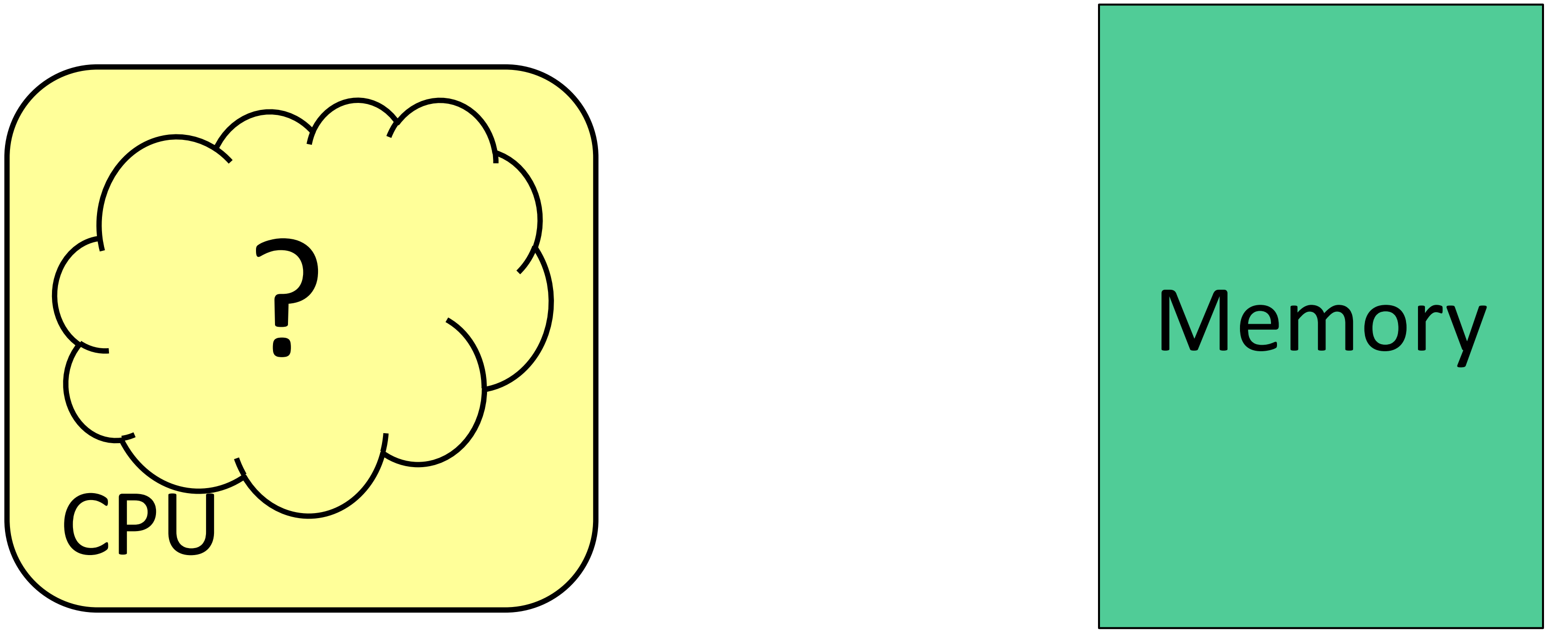
Hardware: Physical View



Hardware: Logical View



Hardware: 351 View (version 0)

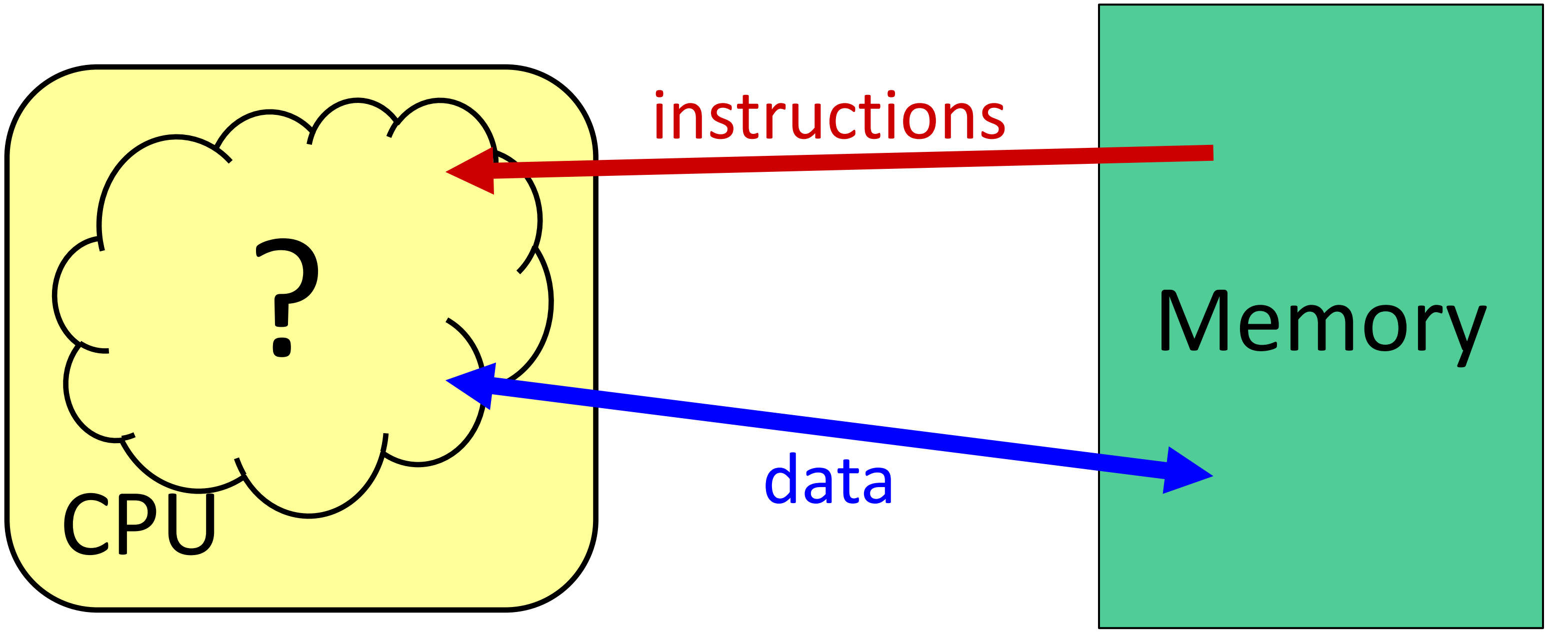


- The CPU executes instructions
- Memory stores data

> How are data and instructions represented?

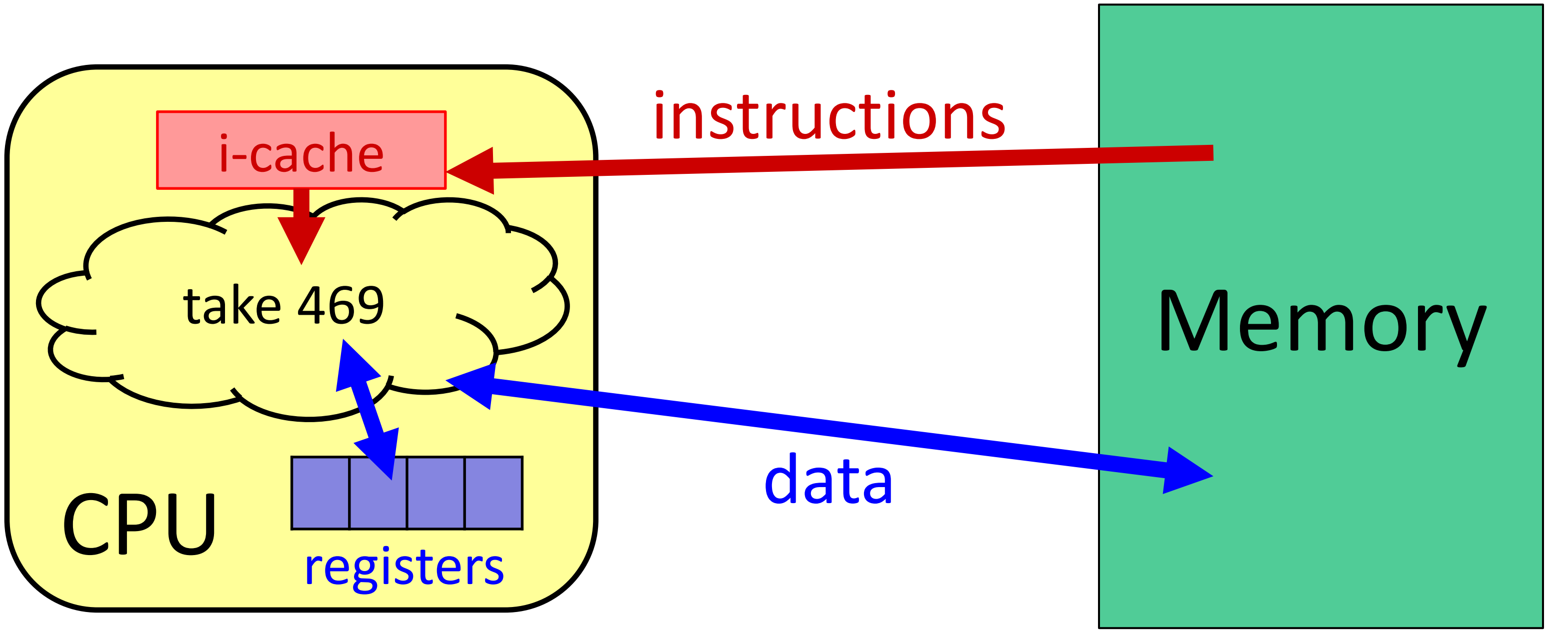
- Binary encoding!
  - Instructions *are* just data

Hardware: 351 View (version 0)



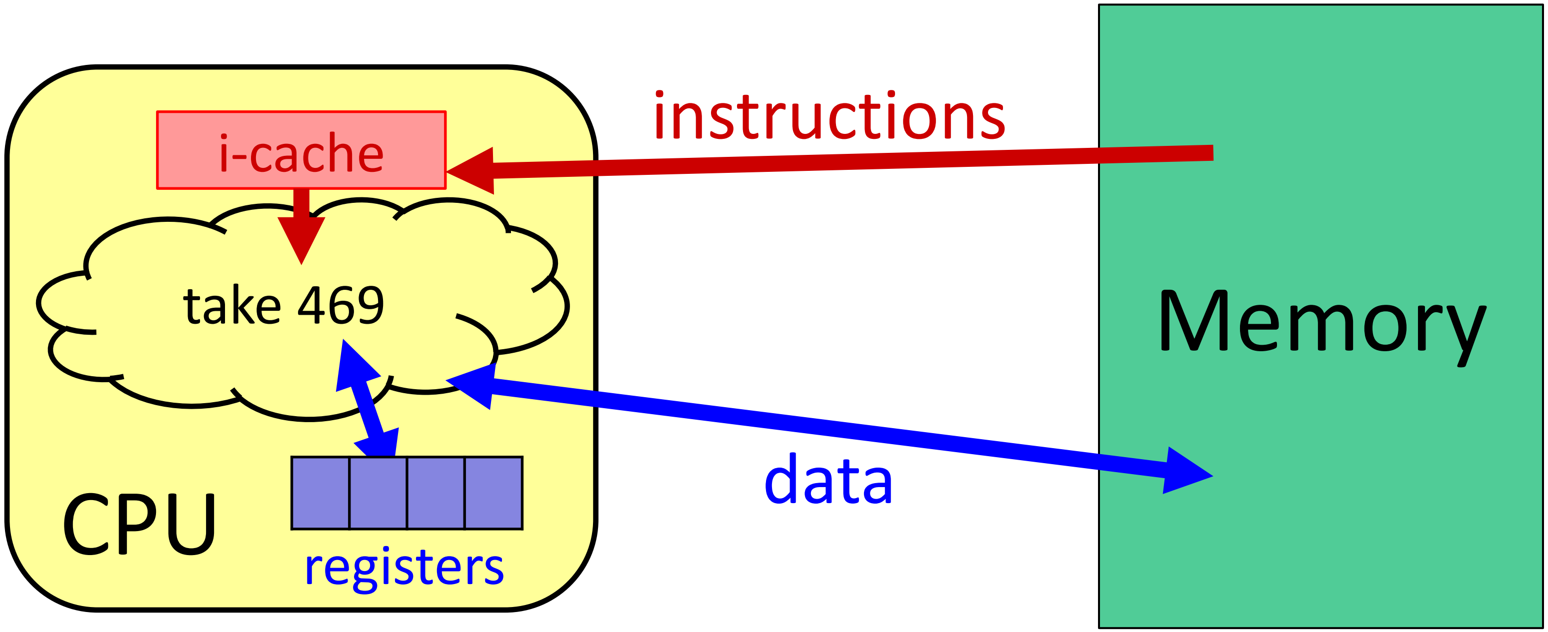
- To execute an instruction, the CPU must:
  1) Fetch the instruction
  2) (if applicable) Fetch data needed by the instruction
  3) Perform the specified computation
  4) (if applicable) Write the result back to memory

Hardware: 351 View (version 1)

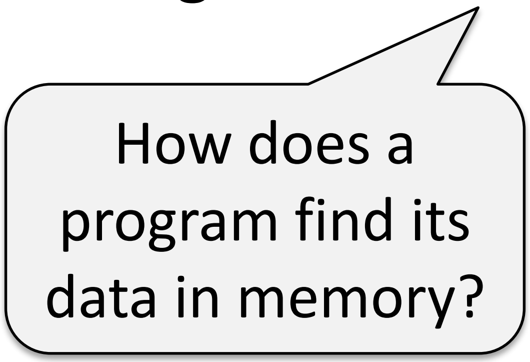


- More CPU details:
  - Instructions are held temporarily in the instruction cache
  - Other data are held temporarily in registers
- Instruction fetching is hardware-controlled
- Data movement is programmer-controlled (assembly)

Hardware: 351 View (version 1)



- We will start by learning about Memory



Review Questions

- By looking at the bits stored in memory, I can tell what a particular 16 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 or its known size.
  - A. True
  - B. False
- Which of the following bytes have a most-significant bit (MSB) of 1?
  - A. 0x3F
  - B. 0xA0
  - C. 0xCA
  - D. 0xDD

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

- Example: the "eight-bit" representation of the number 4 is 0b00000100

Least Significant Bit (LSB)

Most Significant Bit (MSB)

Binary Encoding

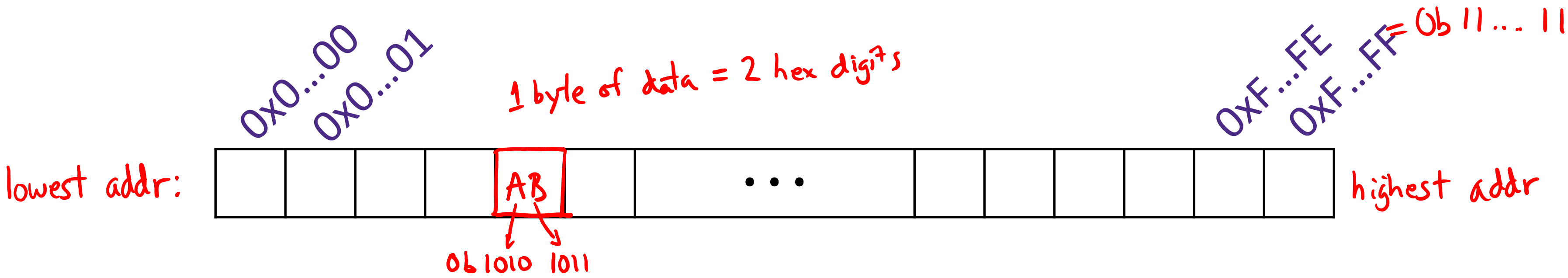
1 bit 0, 1

2 bits 00, 01, 10, 11

- With N binary digits, how many "things" can you represent?
  - Need N binary digits to represent n things, where $2^N \geq n$
  - Example: 5 binary digits for alphabet because $2^5 = 32 > 26$

- A binary digit is known as a bit
- A group of 4 bits (1 hex digit) is called a nibble
- A group of 8 bits (2 hex digits) is called a byte
  - 1 bit → 2 things, 1 nibble → 16 things, 1 byte → 256 things

An Address Refers to a Byte of Memory



- Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)
  - Each address is just a number represented in *fixed-length* binary

- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
  - We can store addresses as data to "remember" where other data is in memory

- But not all values fit in a single byte… (*e.g.* 351)
  - Many operations actually use multi-byte values

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: $\mathbf{2^{64}}$ addresses
    2^{64} bytes ≈ **1.8 x $\mathbf{10^{19}}$ 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	32-bit (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>`

Questions about Multibyte Data

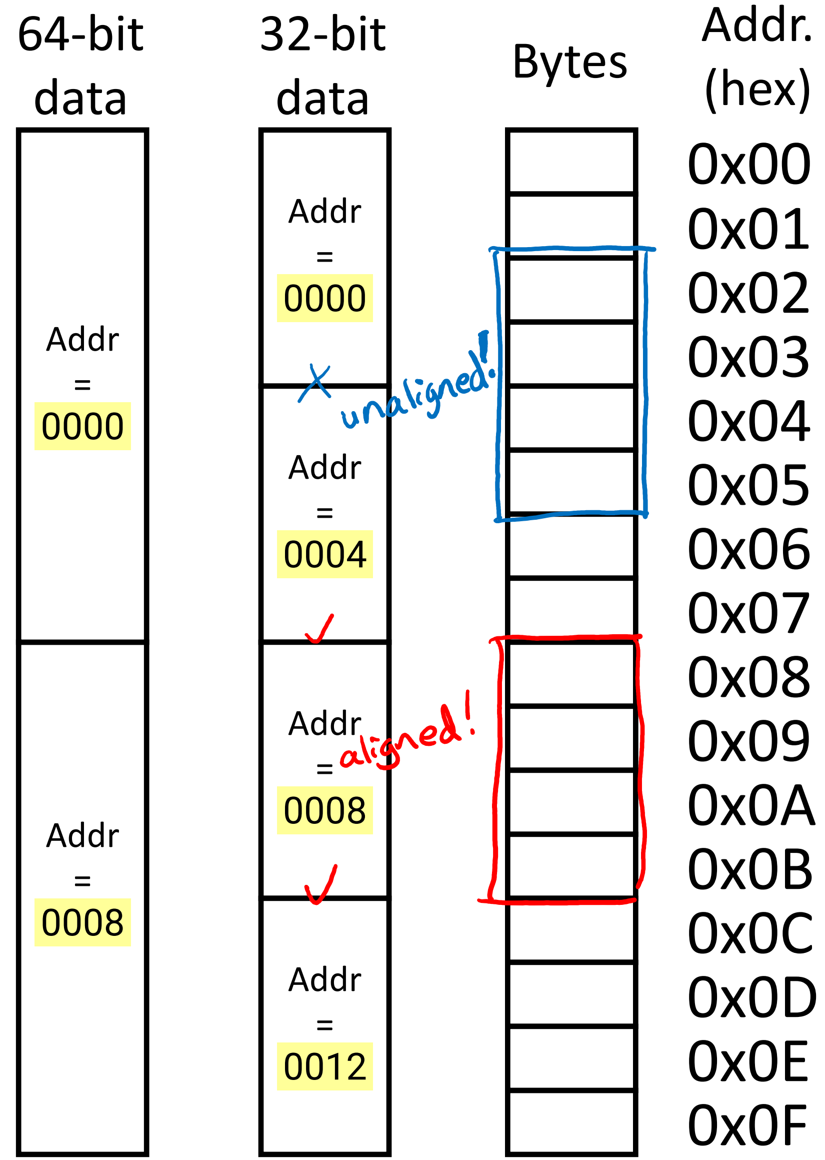
- 1) What do we use as the address of this data object?
- 2) Are there any rules about where you can place multibyte data in memory?

Address of Multibyte Data

- 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**

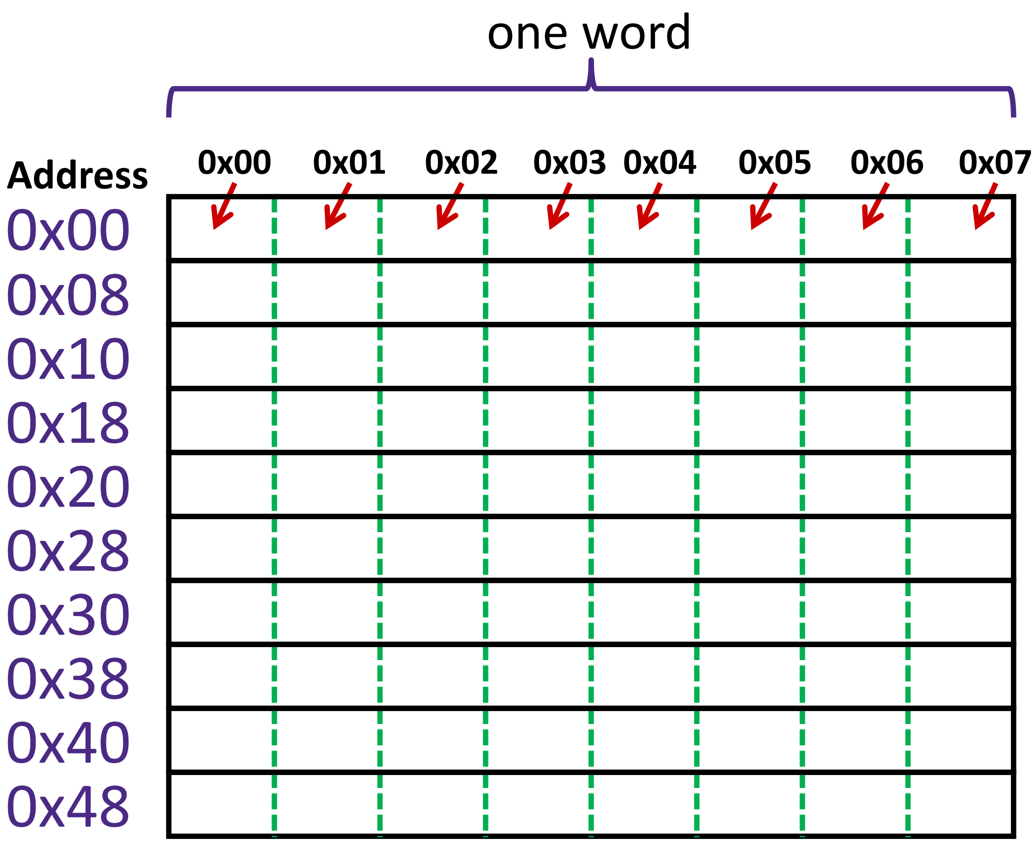
Alignment of Multibyte Data

- The address of a chunk of memory is considered aligned if its address is a multiple of its size
  - View memory as a series of consecutive chunks of this particular size and see if your chunk doesn't cross a boundary



A Picture of Memory (64-bit view)

- A "64-bit (8-byte) word-aligned" view 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



A Picture of Memory (64-bit view)

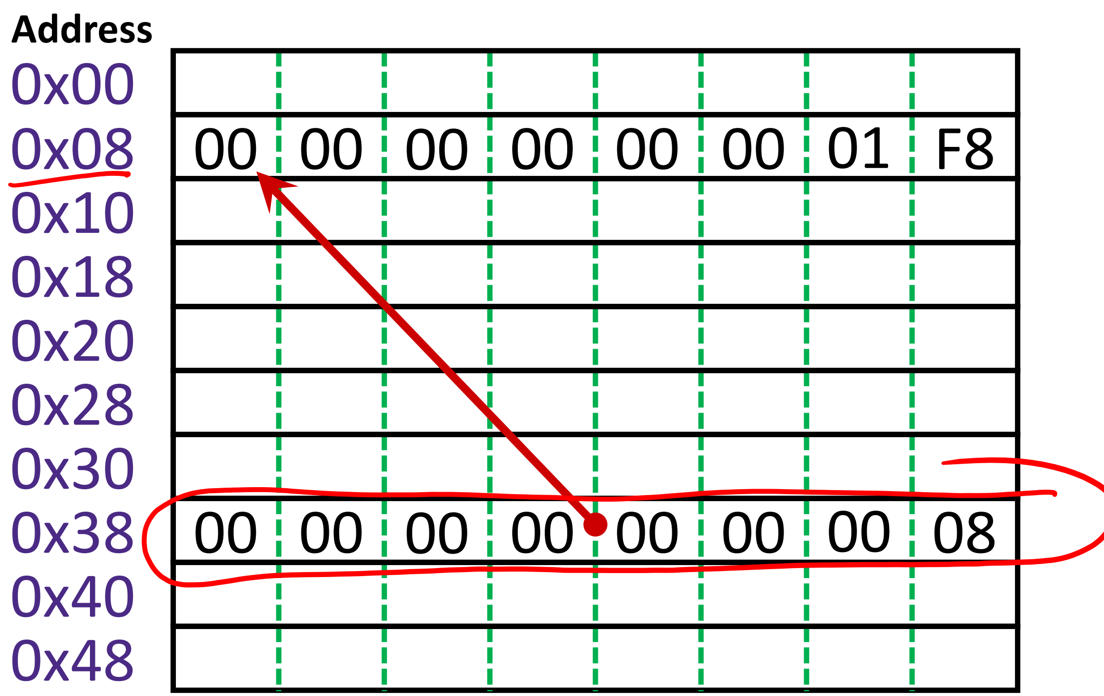
- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - An aligned, 64-bit chunk of data will fit on one row

Addresses and Pointers (step 1)

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* type of data
- Value 504 stored at address 0x08
  - $504_{10} = 1F8_{16}$
    $= 0x\ 00 \ldots 00\ 01\ F8$
- Pointer stored at 0x38 points to address 0x08

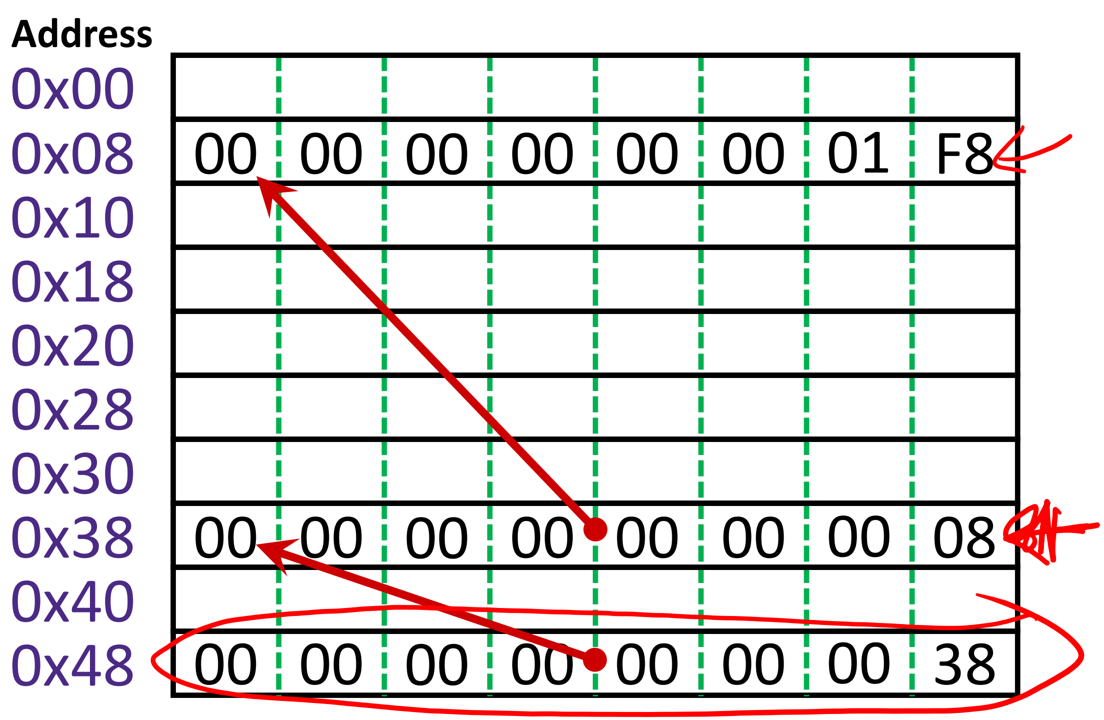


Addresses and Pointers (step 2)

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* type of 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



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`

Most Significant Byte   Least Significant Byte

- 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

Big-Endian

		0x100	0x101	0x102	0x103		
		A1	B2	C3	D4		

Little-Endian

		0x100	0x101	0x102	0x103		
		D4	C3	B2	A1		

Byte Ordering (solution)

- 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		
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

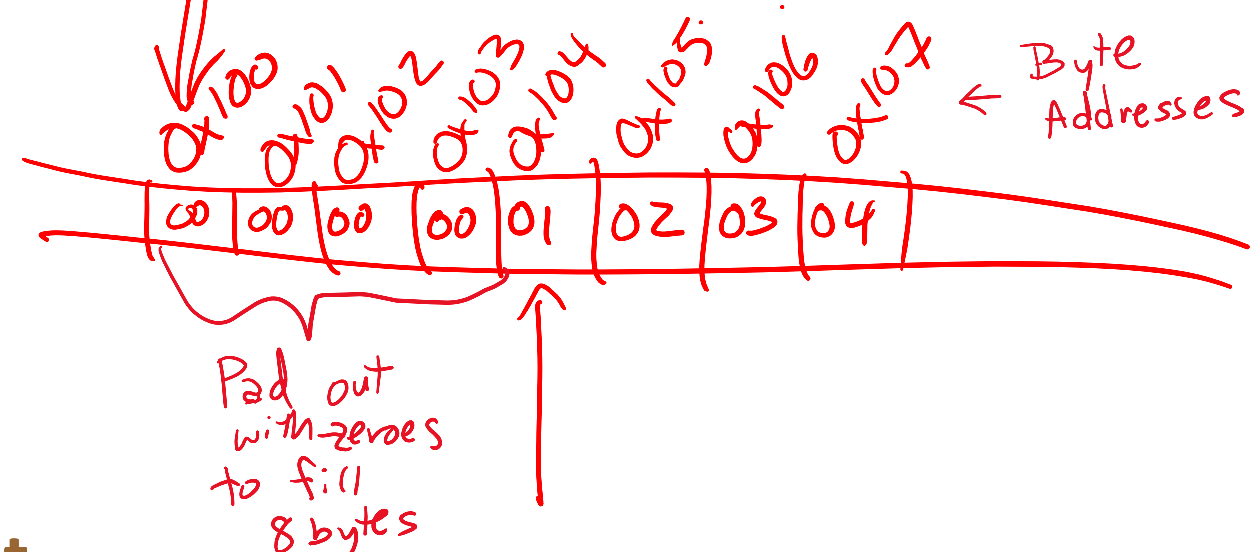
A. 0x04

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
  - Object of K bytes is *aligned* if it has an address that is a multiple of K
- Pointers are data objects that hold addresses
- Endianness determines memory storage order for multi-byte data