

The Hardware/Software Interface

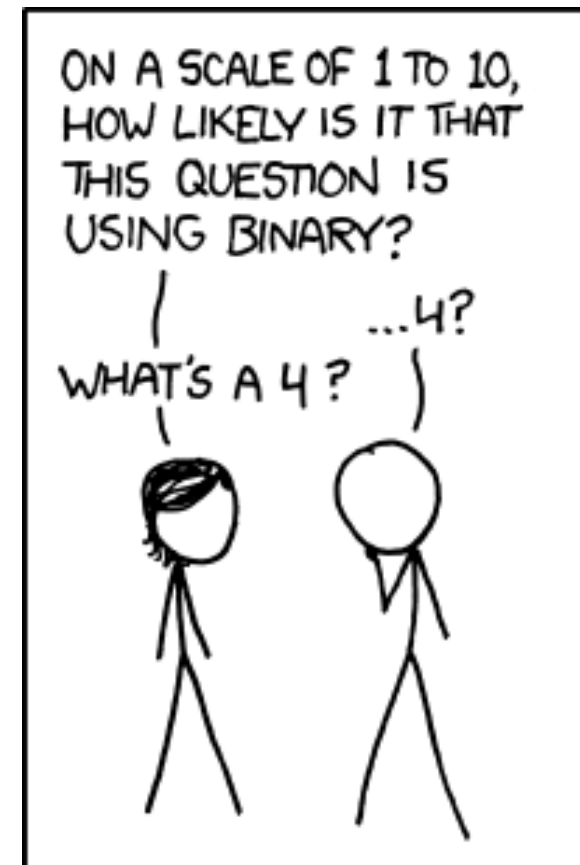
Memory, Data, & Addressing I

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	



<http://xkcd.com/953/>

Relevant Course Information

❖ Upcoming deadlines

- Pre-Course Survey and HW0 due tonight
- HW1 due Monday (9/29) night
- Lab 0 due Monday (9/29) night
 - This lab is *exploratory* and looks like a HW; the other labs will look a lot different
- Reminder: Readings due before every lecture!

❖ Ed Discussion etiquette

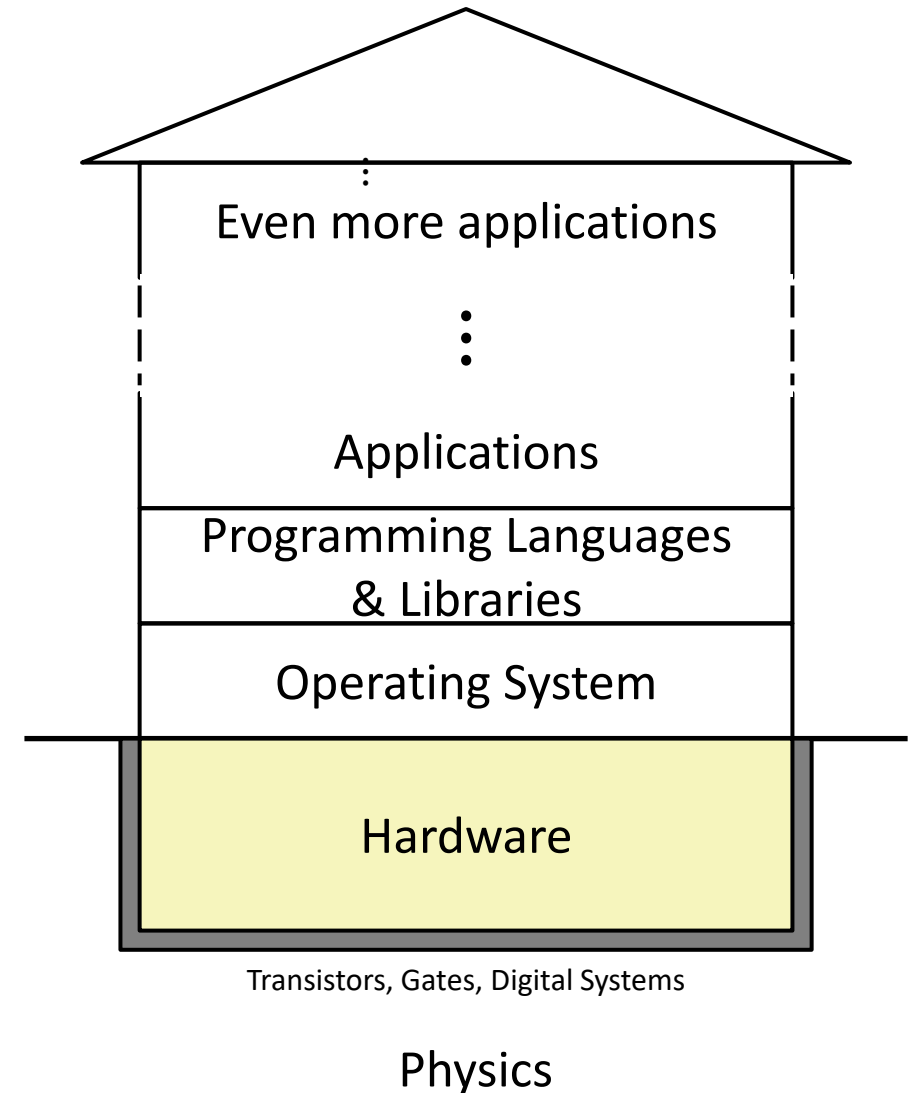
- For anything that doesn't involve sensitive information or a solution, post publicly (you can post anonymously!)
- If you feel like your question has been sufficiently answered, make sure that a response has a checkmark

EPA

- ❖ Encourage class-wide learning!
- ❖ Effort
 - Attending support hours, completing all assignments
 - Keeping up with Ed Discussion activity
- ❖ Participation
 - Making the class more interactive by asking questions in lecture, section, support hours, and on Ed Discussion
- ❖ Altruism
 - Helping others in section, support hours, and on Ed Discussion

House of Computing Check-In

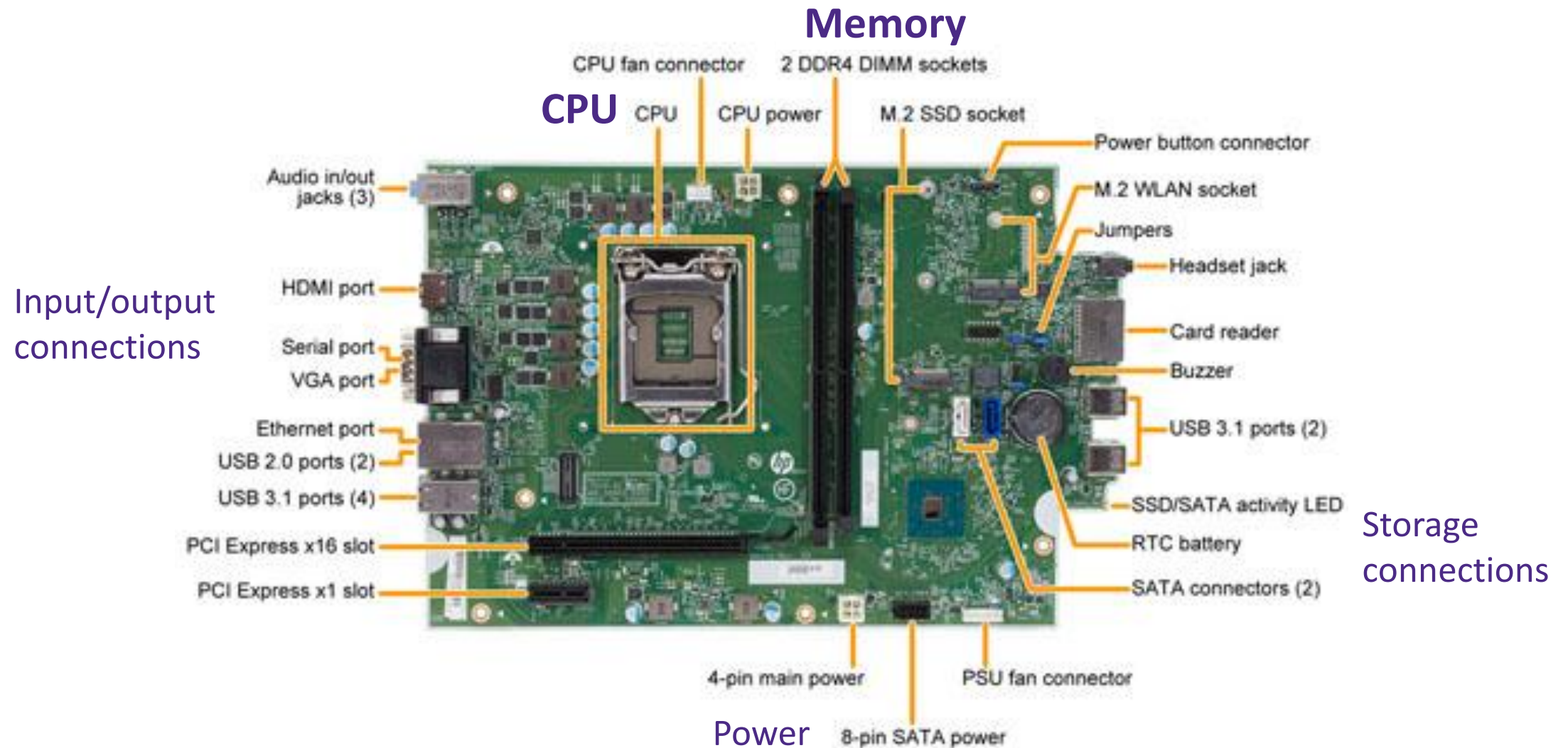
- ❖ Topic Group 1: **Data**
 - **Memory, Data**, Integers, Floating Point, Arrays, Structs
- ❖ How do we store information for other parts of the house of computing to access?
 - How do we represent data and what limitations exist?
 - What design decisions and priorities went into these encodings?



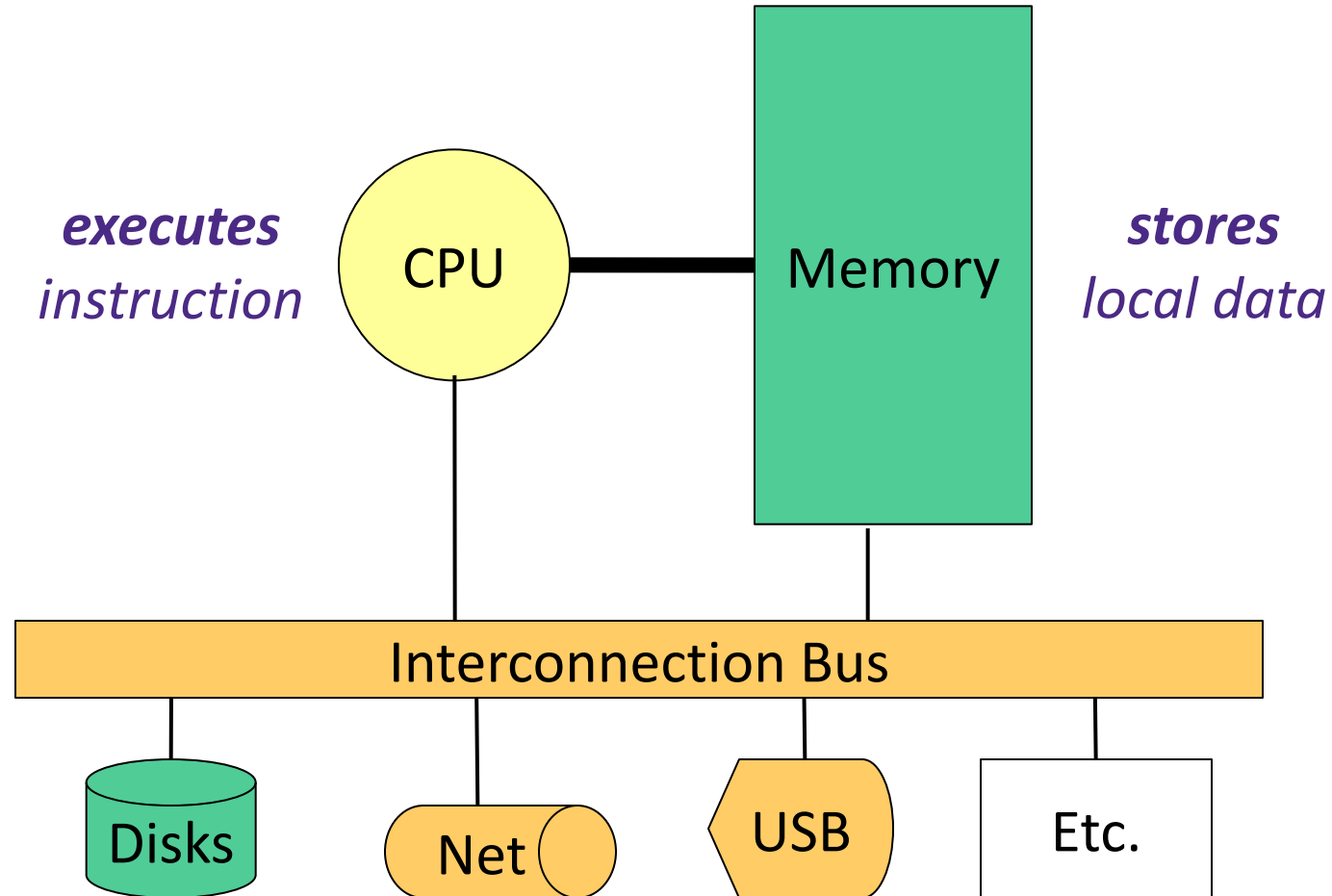
Lecture Outline (1/3)

- ❖ **Memory and Addresses**
- ❖ Data in Memory
- ❖ Data Basics in Programming

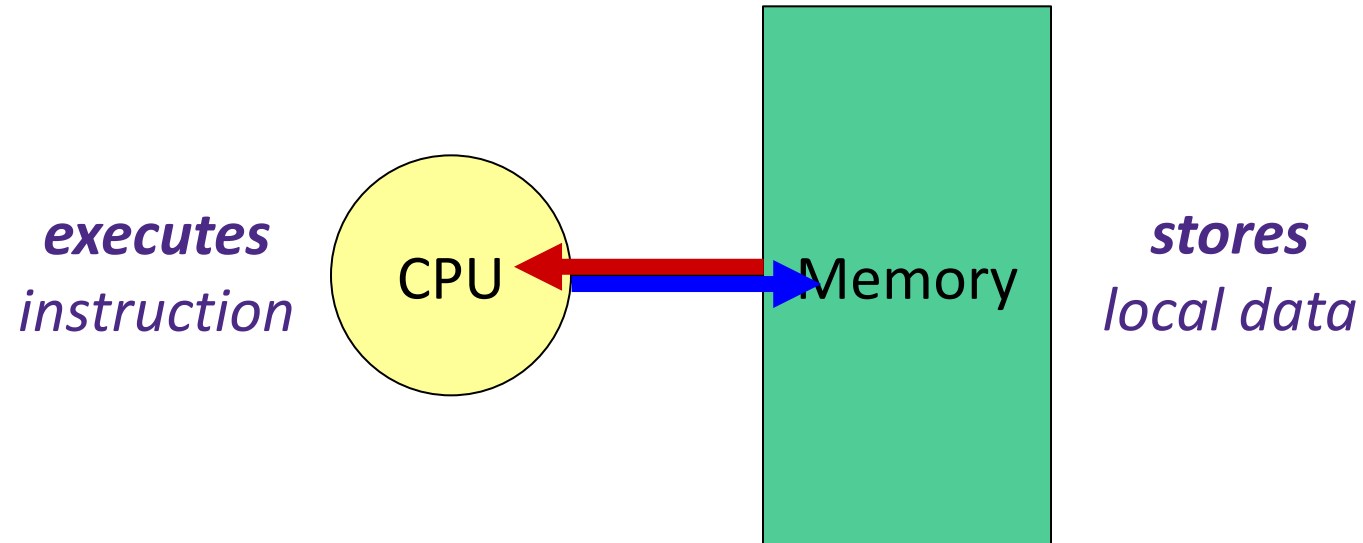
Hardware: Physical View



Hardware: Logical View

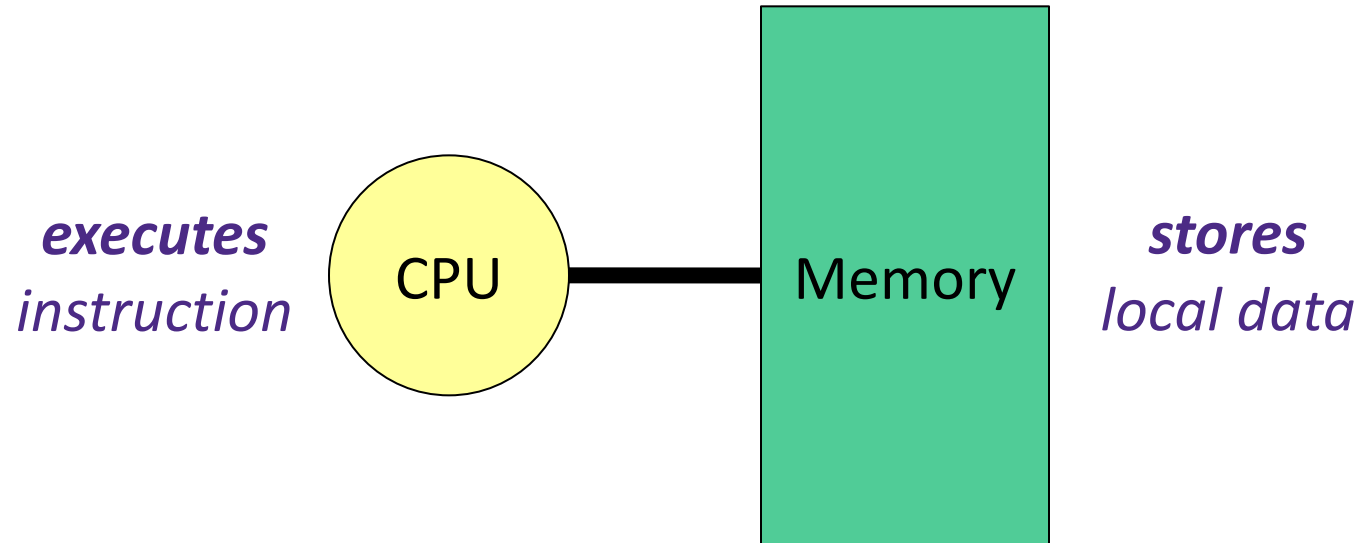


CPU and Memory



- ❖ The CPU communicates with memory frequently
 - **Fetches** (loads) data upon request from memory
 - **Writes** (stores) data to memory

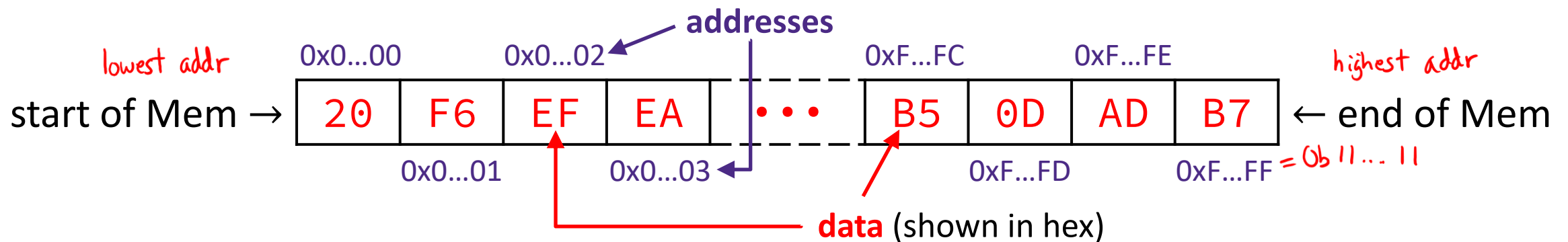
Memory Operation Basics



- ❖ What does data look like? *binary representation*
 - It turns out that instructions *are* data, too, and encoded in “machine code”
- ❖ How do we find or specify data in memory? *addresses*
 - Programs have built-in ways to track addresses

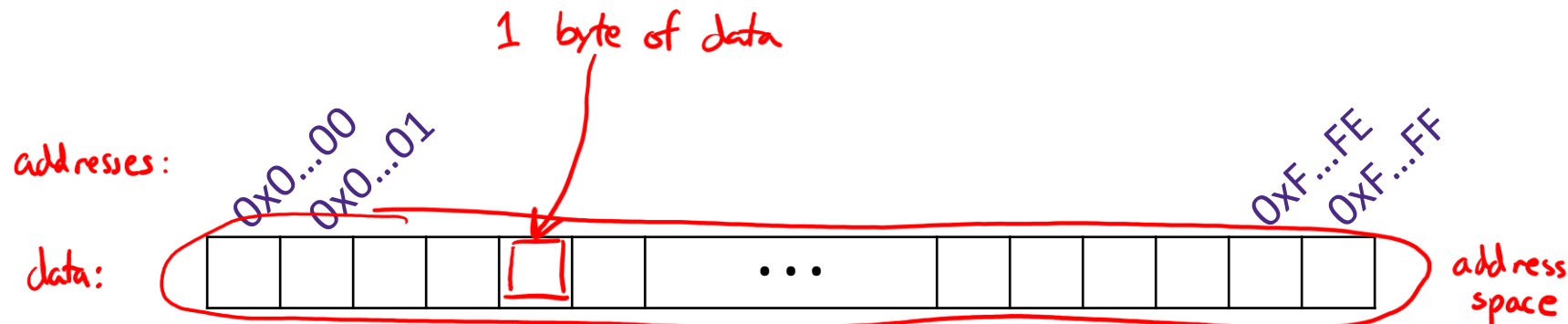
Addresses (Review)

- ❖ Conceptually, memory is a single, large array of bytes (*i.e.*, **byte-oriented**)
- ❖ Programs refer to bytes in memory by their unique **addresses** (indices)
 - We number addresses in increasing order starting from 0
 - By convention, address size = word size (*fixed-length*)
 - Domain of possible addresses = **address space**



Bits and Bytes and Things

- ❖ 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?

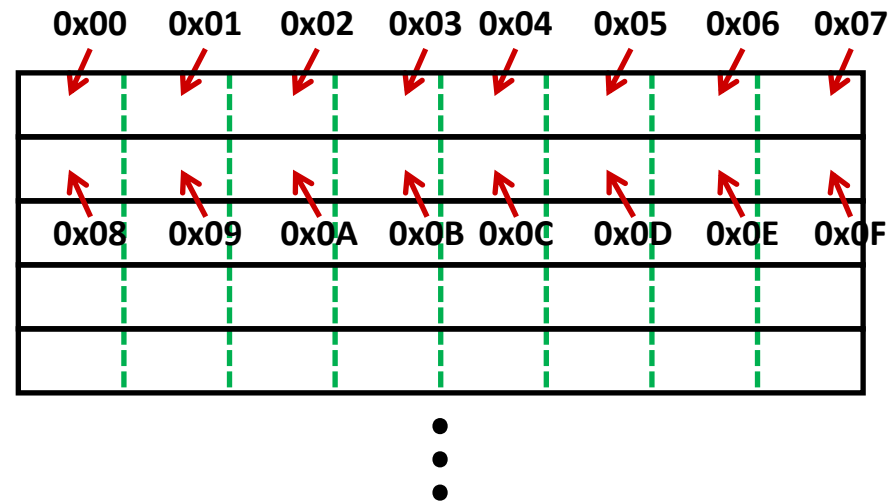


Modern System Details

- ❖ Current x86-64 systems use **64-bit (8-byte) words** (“64-bit machines”)
 - Potential address space: 2^{64} addresses
 2^{64} bytes $\approx$ **1.8×10^{19} bytes**
= 18 billion billion bytes = 18 EB (exabytes)
 - Actual physical address space: **48 bits**
 - This is sufficient space for now and allows for some operating system tricks
 - Example address: 0x 7f fc 3d d5 06 94
- ❖ There’s a lot more to this story... stay tuned for virtual memory!

Visualizing Memory

- ❖ We will regularly depict memory as a two-dimensional array
 - Each cell is a byte
 - Addresses increase from left-to-right and then top-to-bottom
 - Row width will most commonly be chosen to the word size (8 bytes here)



Lecture Outline (2/3)

- ❖ Memory and Addresses
- ❖ **Data in Memory**
- ❖ Data Basics in Programming

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 1-byte representation of 4 is 0b00000100
-
- value of 2^7
“most weight”
- padding
- value of 2^0
“least weight”
- Most Significant Bit (MSB)
- Least Significant Bit (LSB)

Address of Multibyte Data (Review)

- ❖ Data that span multiple bytes can be thought of as “chunks” of memory

- Example: 351 = 0b1 0101 1111, stored as 0x015F
- Each individual byte has a unique address – how should we refer to the chunk’s address/location?

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

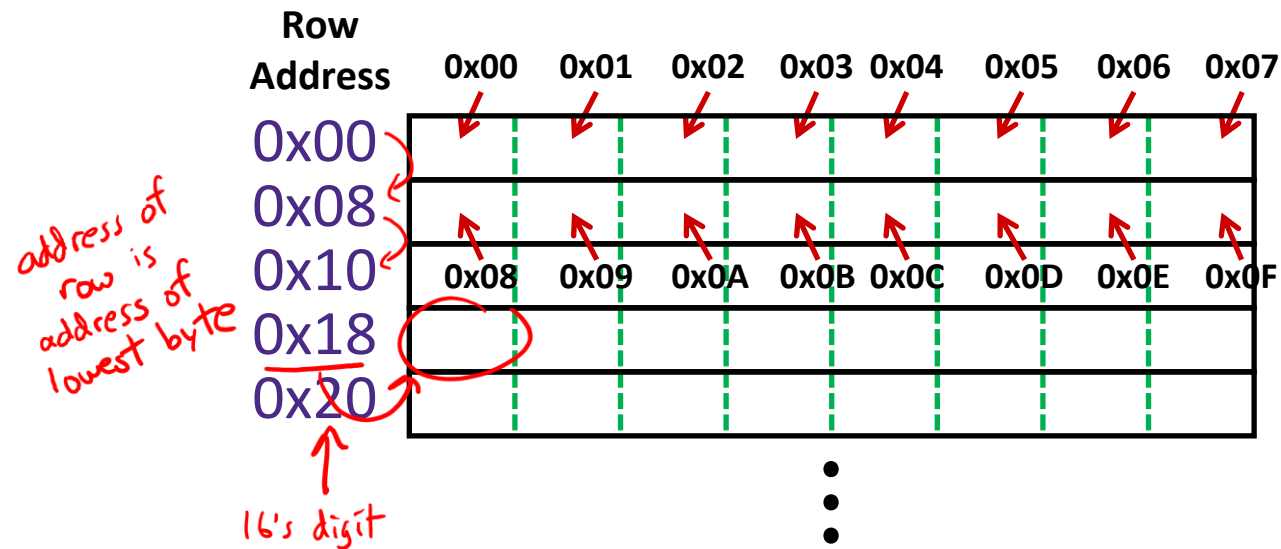
- To specify a chunk of memory, need *both* its **address** and its **size**

Bytes	Addr. (hex)
	0x00
	0x01
	0x02
	0x03
	0x04
	0x05
	0x06
	0x07
	0x08
	0x09
	0x0A
	0x0B
	0x0C

address of 0x2

Visualizing Memory (Revisited)

- ❖ We will regularly depict memory as a two-dimensional array
 - Addresses increase from left-to-right and then top-to-bottom
 - Row width will most commonly be chosen to the word size (8 bytes here)
 - Row address is given by the lowest address in the row



Polling Questions (1/2)

- ❖ 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

many possible encoding schemes

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

A. True

B. False

need: ① address ✓
② data size X

- ❖ Which of the following bytes have a most-significant bit (MSB) of 1? → 8 bits = 2 hex digits

0b 0110 0011
A. 0x63

0b 1001 0000
B. 0x90

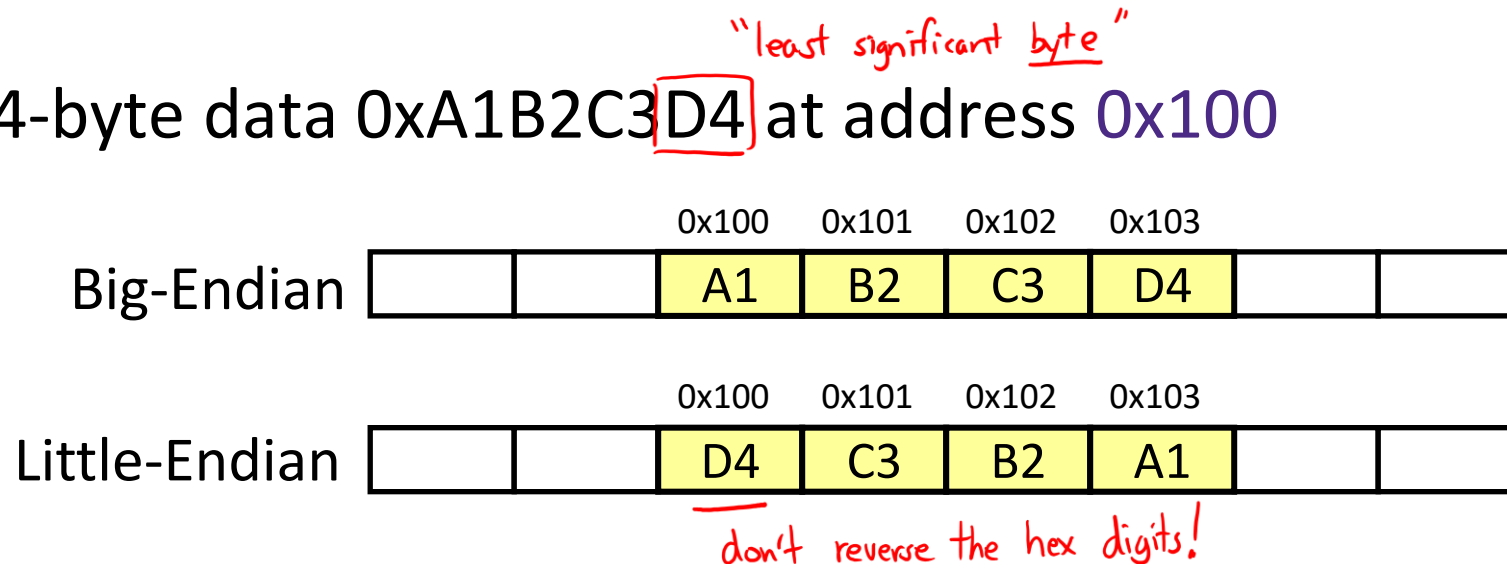
0b 1100 1010
C. 0xCA

0b 0000 1111
D. ~~0xF~~ 0x0F

Byte Ordering (Review)

- ❖ How should bytes within a word be ordered *in memory*?
 - Want to keep consecutive bytes in consecutive addresses
 - By convention, ordering of bytes called *endianness* – in which address does the least significant *byte* go?
 - **Big-endian** means least significant byte has highest address
 - **Little-endian** means least significant byte has lowest address

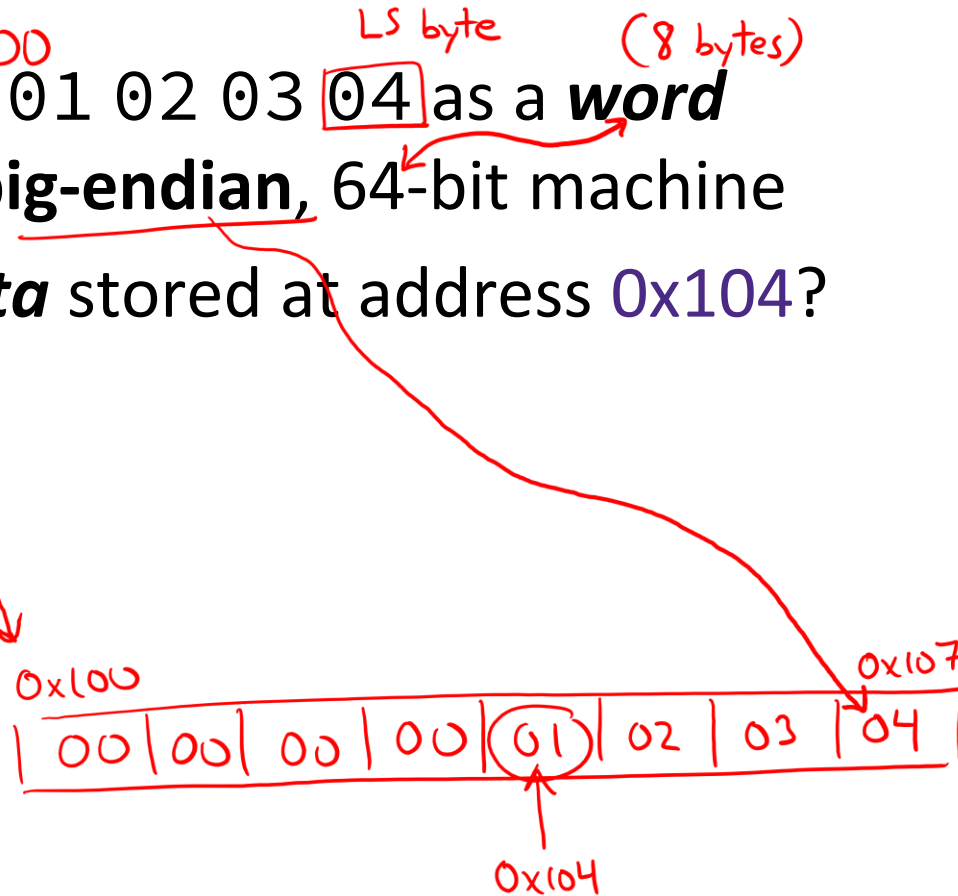
- ❖ **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



Polling Questions (2/2)

- ❖ 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 $0x104$?

- A. $0x04$
- B. $0x40$
- C. $0x01$
- D. $0x10$
- E. We're lost...



Endianness Notes

- ❖ Endianness is a property of the architecture
 - We are using x86-64, which is little-endian
- ❖ *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 (C/C++)
 - Need to know exact values to debug memory errors (common)

Lecture Outline (3/3)

- ❖ Memory and Addresses
- ❖ Data in Memory
- ❖ **Data Basics in Programming**

Data Types and Sizes (Review)

- ❖ Variables stored in memory treated as “chunks”
 - C data type sizes vary somewhat by architecture (*e.g.*, IA-32 vs. x86-64)

C Data Type	Java “Equivalent”	Size in bytes (x86-64)
char	byte	1
short	short	2
int	int	4
long		8
long long	long	8
float	float	4
double	double	8
long double		16

Variables (Review)

- ❖ Variables stored in memory treated as “chunks”
 - A variable name is an alias for a *location* that contains its data/value
 - *Declaring* a variable allocates space for it (e.g., `int x;`)
 - *Initializing* a variable also assigns an initial value to that space (e.g., `int x = 3;`)
- ❖ Programming language differences
 - In Java, variable declaration implicitly performs initialization
 - In C, declaration does not perform initialization (initially “**mystery data**”)

C Data Type	x86-64 Size
char	1B
short	2B
int	4B
long	8B
long long	8B
float	4B
double	8B
long double	16B

Alignment (Review)

- ❖ Variables stored in memory treated as “chunks”
 - A variable name is an alias for a *location* that contains its data/value
 - *Declaring* a variable allocates space for it (e.g., `int x;`)
 - *Initializing* a variable also assigns an initial value to that space (e.g., `int x = 3;`)
- ❖ Alignment
 - A variable is considered **aligned** if its address is a multiple of its size
 - Not always required, but common

C Data Type	x86-64 Size
char	1B
short	2B
int	4B
long	8B
long long	8B
float	4B
double	8B
long double	16B

Assignment Example (1/4)

- ❖ Syntax: left-hand side (LHS) = right-hand side (RHS);
- ❖ Effect: store *value* of RHS into the *location* given by LHS

- ❖ Example: Little-endian, current state of memory

- **int** x, y;

- Assume x is at address 0x04, y is at 0x18

	0x00	0x01	0x02	0x03	
0x00	A7	00	32	00	
0x04	00	01	29	F3	x
0x08	DE	AD	BE	EF	
0x0C	FA	CE	CA	FE	
0x10	26	00	00	00	
0x14	00	00	10	00	
0x18	01	00	00	00	y
0x1C	FF	00	F4	96	
0x20	EE	EE	EE	EE	
0x24	00	00	00	00	

Assignment Example (2/4)

- ❖ Syntax: left-hand side (LHS) = right-hand side (RHS);
- ❖ Effect: store *value* of RHS into the *location* given by LHS

- ❖ Example: Little-endian, *partial* state of memory

- **int** x, y;

- Assume x is at address 0x04, y is at 0x18

- x = 0; ^{pad} 0x 00 00 00 00
↑ int (4 bytes)

	0x00	0x01	0x02	0x03	
0x00					
0x04	00	00	00	00	x
0x08					
0x0C					
0x10					
0x14					
0x18	01	00	00	00	y
0x1C					
0x20					
0x24					

Assignment Example (3/4)

- ❖ Syntax: left-hand side (LHS) = right-hand side (RHS);
- ❖ Effect: store *value* of RHS into the *location* given by LHS

- ❖ Example: Little-endian, *partial* state of memory

- **int** x, y;

- Assume x is at address 0x04, y is at 0x18

- x = 0;

- y = 0x3CD02700;

least significant
byte

	0x00	0x01	0x02	0x03	
0x00					
0x04	00	00	00	00	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20					
0x24					

Assignment Example (4/4)

- ❖ Syntax: left-hand side (LHS) = right-hand side (RHS);
- ❖ Effect: store *value* of RHS into the *location* given by LHS

- ❖ Example: Little-endian, *partial* state of memory

- **int** x, y;

- Assume x is at address 0x04, y is at 0x18

- x = 0;

- y = 0x3CD02700;

- x = y + 3; 0x3CD02703

- Get value at y, add 3, store in x

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20					
0x24					

Homework Setup (If Time)

- ❖ Assume that a snippet of memory is shown below (in hex), starting with the byte at address 0x08 on a little-endian machine:

↑ least significant byte at lowest addr

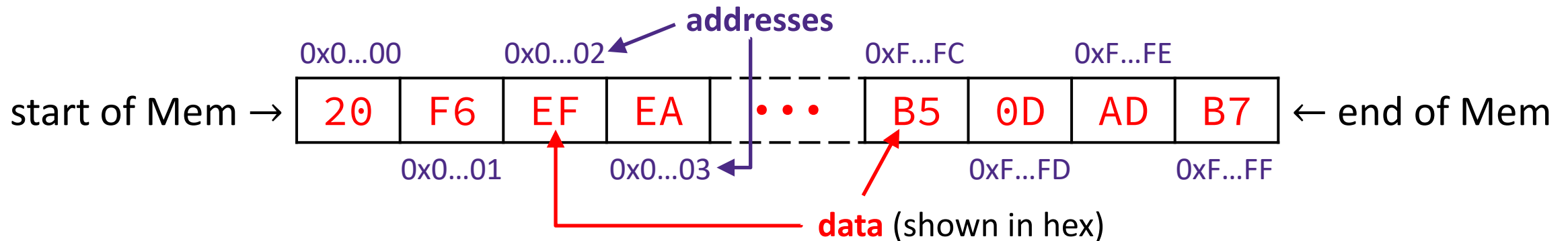
addr:	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
data:	A2	D0	4F	C4	A0	0C	F7	27

- ❖ What is the value of the int stored at address 0x0C?
4 bytes lowest byte

| 0x 27 F7 0CA0 |

Summary (1/2)

- ❖ Memory is a long, *byte-addressed* array
 - Word size bounds the size of the *address space* and memory
 - Address of a chunk of memory given by the address of the lowest byte in chunk
- ❖ Endianness determines memory storage order for multi-byte data
 - Least significant byte in lowest (little-endian) or highest (big-endian) address of memory chunk



Summary (2/2)

❖ Programming Data

- Variable declaration allocates space for data type size
- Assignment results in value being put in memory location

C Data Type	x86-64 Size
char	1B
short	2B
int	4B
long	8B
long long	8B
float	4B
double	8B
long double	16B

	0x00	0x01	0x02	0x03	
0x00	A7	00	32	00	x
0x04	00	01	29	F3	
0x08	DE	AD	BE	EF	
0x0C	FA	CE	CA	FE	
0x10	26	00	00	00	y
0x14	00	00	10	00	
0x18	01	00	00	00	
0x1C	FF	00	F4	96	
0x20	EE	EE	EE	EE	
0x24	00	00	00	00	