

# Memory, Data, & Addressing I

CSE 351 Autumn 2020

## Instructor:

Justin Hsia

## Teaching Assistants:

Aman Mohammed

Ami Oka

Callum Walker

Cosmo Wang

Hang Do

Jim Limprasert

Joy Dang

Julia Wang

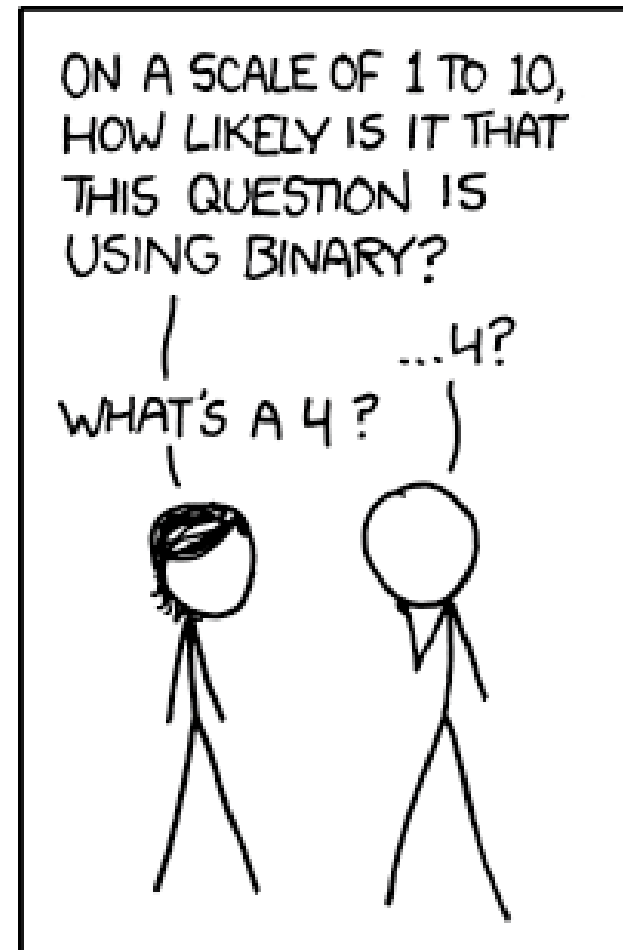
Kaelin Laundry

Kyrie Dowling

Mariam Mayanja

Shawn Stanley

Yan Zhe Ong



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

# Administrivia

- ❖ Pre-Course Survey and hw0 due tonight @ 11:59 pm
  - All other hw due at 11:00 am (Seattle time)
- ❖ hw1 due Monday (10/5)
- ❖ Lab 0 due Monday (10/5) @ 11:59 pm
  - 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 your question has been sufficiently answered, make sure that a response has a checkmark

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Memory & data

- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

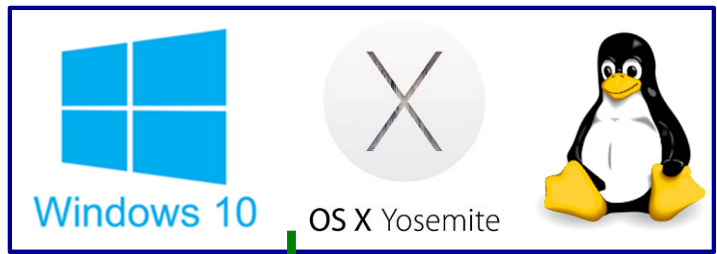
Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

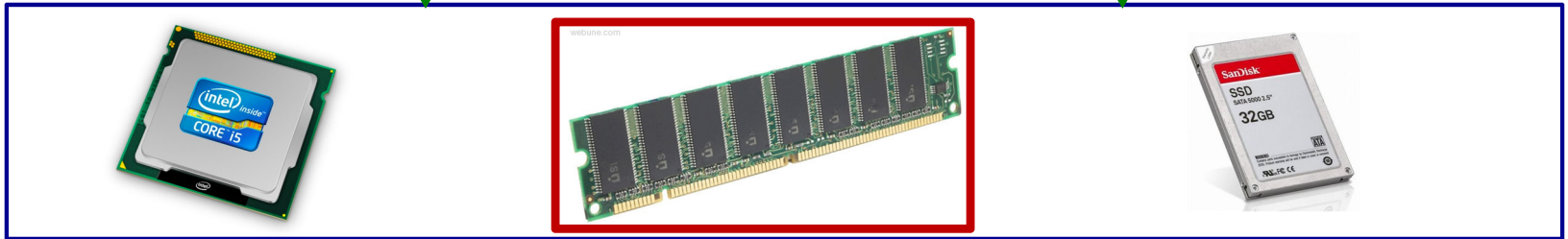
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



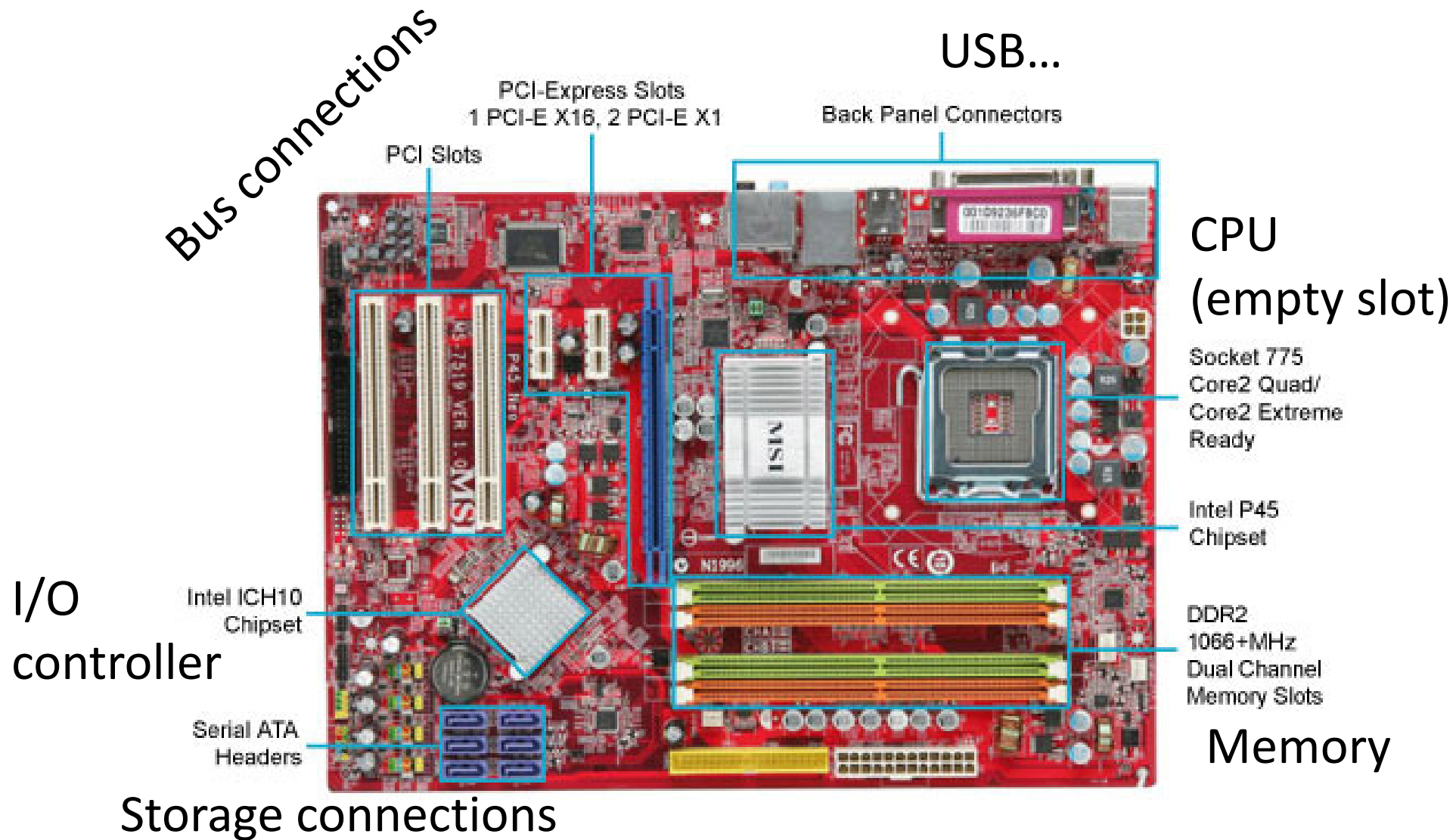
Computer system:



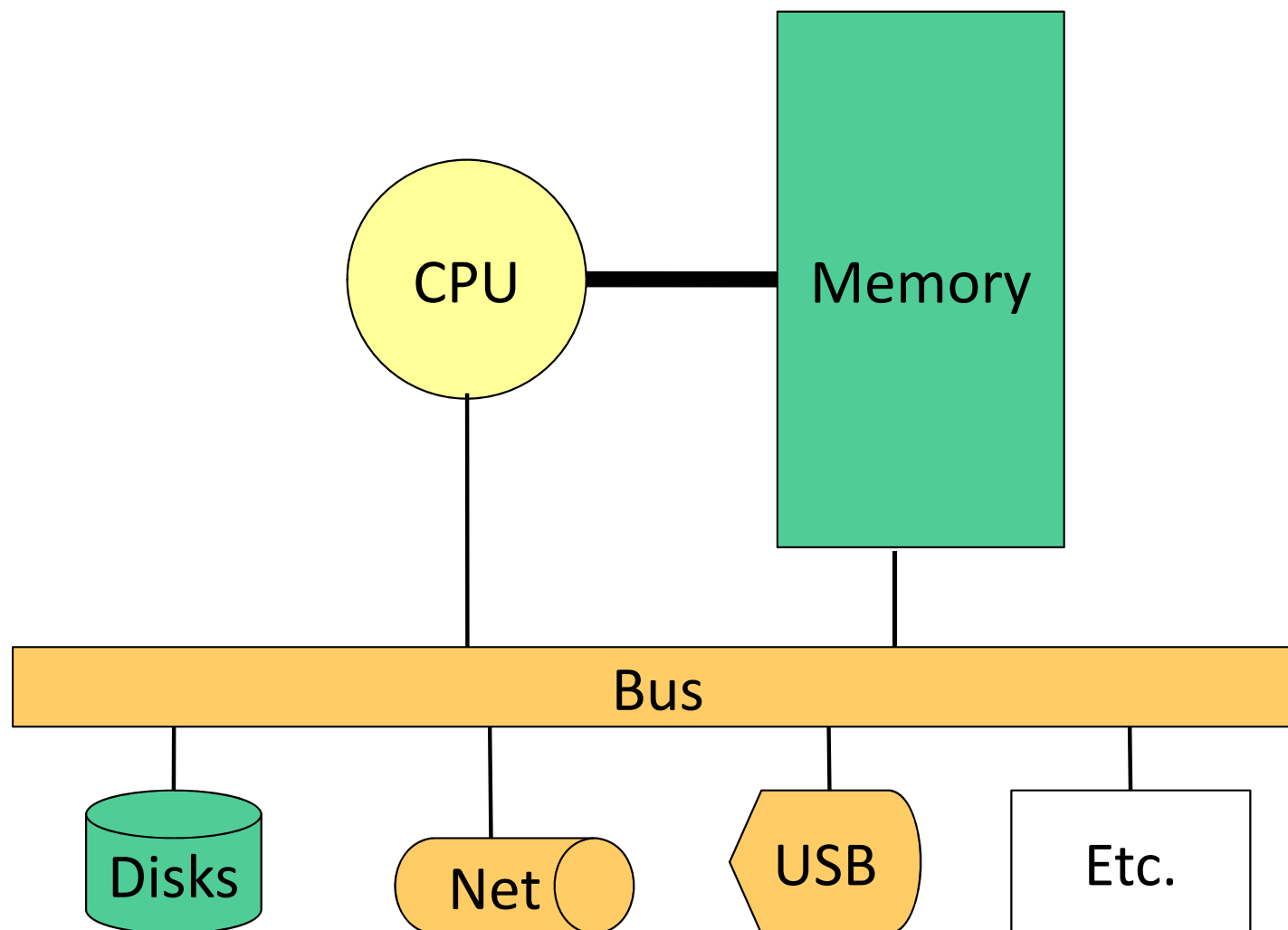
# Reading Review

- ❖ 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?

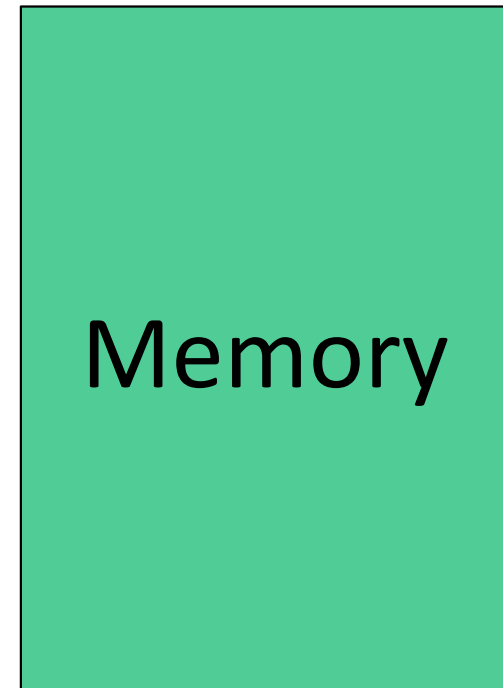
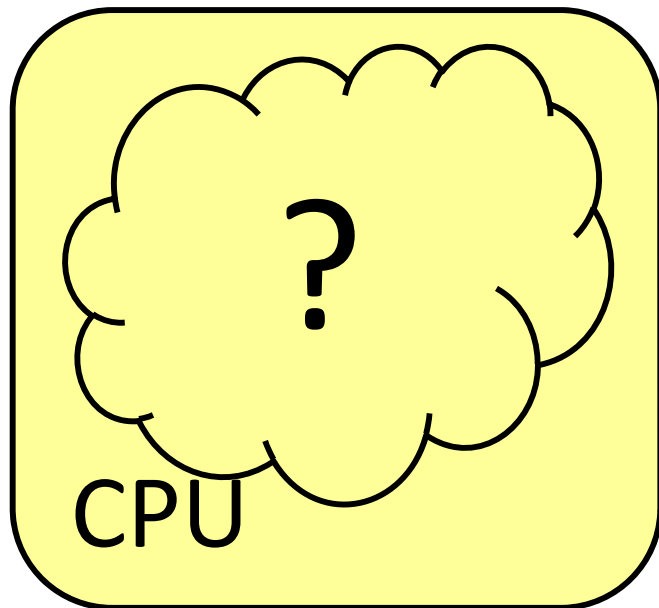
# Hardware: Physical View



# Hardware: Logical View



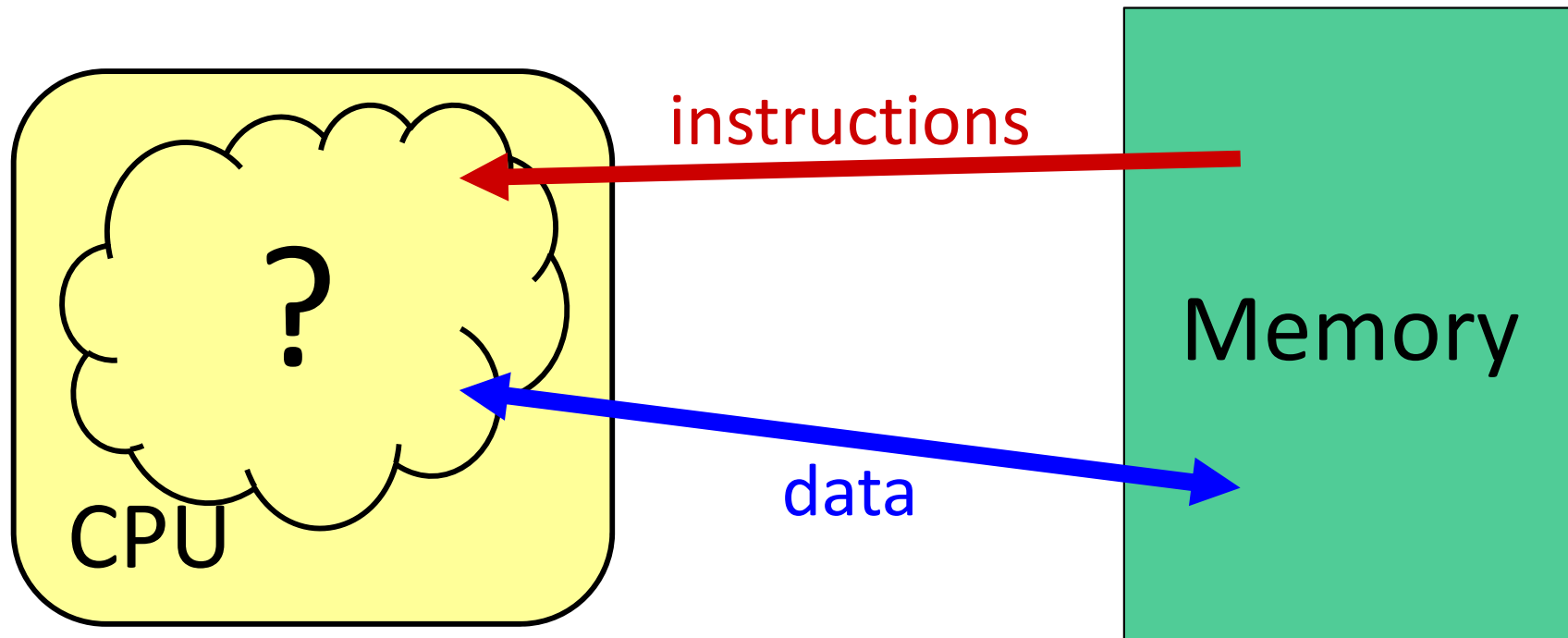
# Hardware: 351 View (version 0)



- ❖ The CPU **executes** instructions
- ❖ Memory **stores** data
- ❖ Binary encoding!
  - Instructions *are* just data

How are data  
and instructions  
represented?

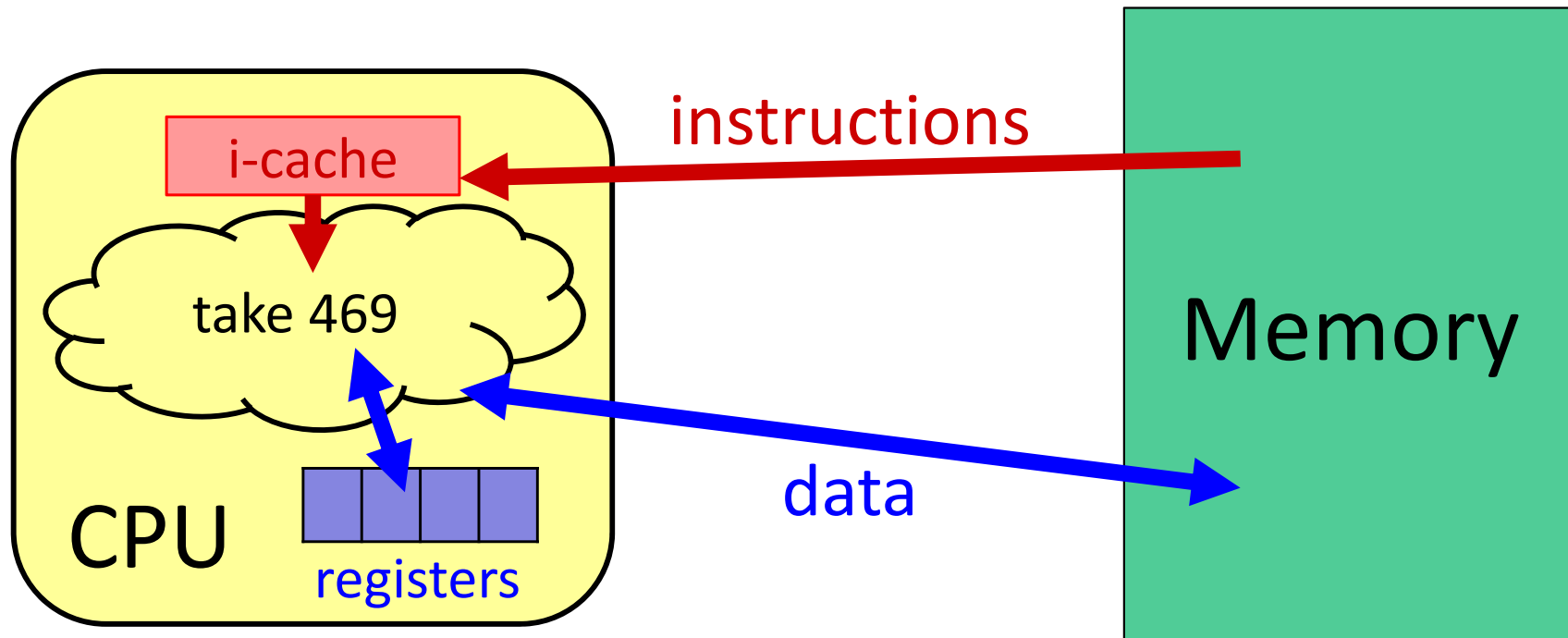
# Hardware: 351 View (version 0)



- ❖ 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

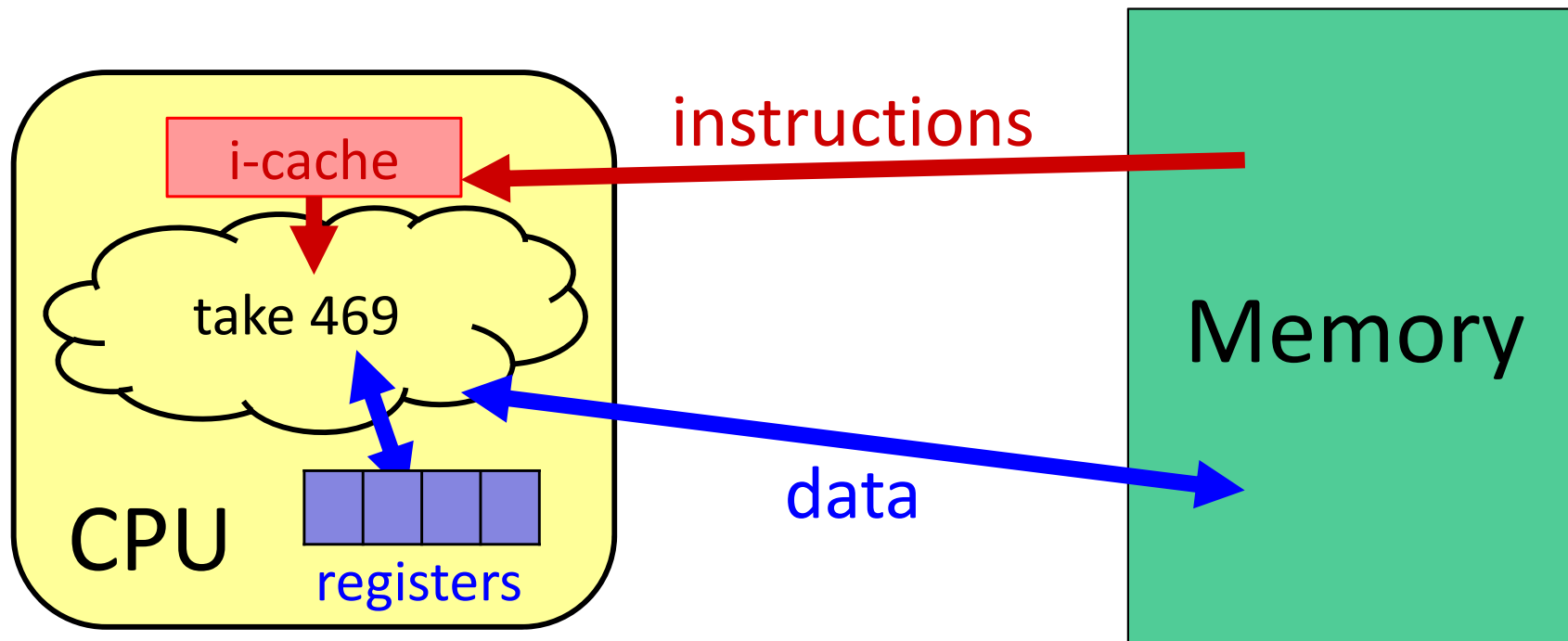


# Hardware: 351 View (version 1)



- ❖ 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

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

How does a program find its data in memory?

# Review Questions

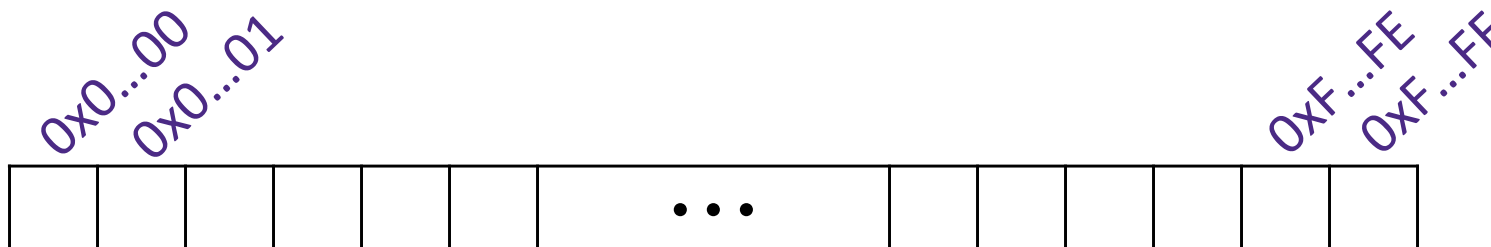
- ❖ By looking at the bits stored in memory, I can tell what a particular 4 bytes is being used to represent.  
**A. True**      **B. False**
- ❖ We can fetch a piece of data from memory as long as we have its address.  
**A. True**      **B. False**
- ❖ Which of the following bytes have a most-significant bit (MSB) of 1?  
**A. 0x63**      **B. 0x90**      **C. 0xCA**      **D. 0xF**

# Binary Encoding Additional Details

- ❖ 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
  - Most Significant Bit (MSB)
  - Least Significant Bit (LSB)

# 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?



# Machine “Words”

- ❖ 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  $\rightarrow 2^w$  addresses
- ❖ Current x86 systems use **64-bit (8-byte) words**
  - Potential address space:  $2^{64}$  addresses  
 $2^{64}$  bytes  $\approx$  **1.8 x 10<sup>19</sup> bytes**  
= 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

# Data Representations

## ❖ Sizes of data types (in bytes)

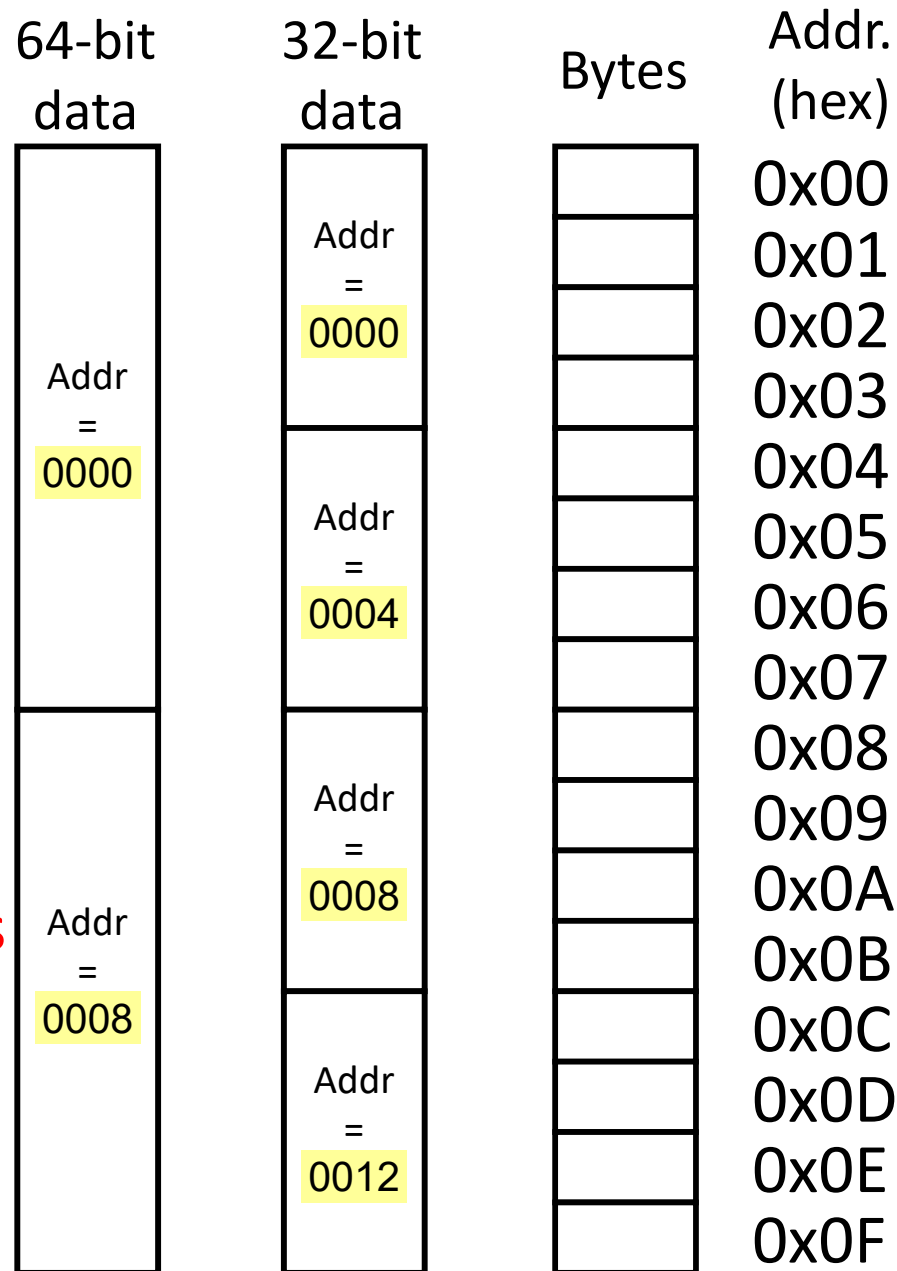
Java Data Type	C Data Type	32-bit (old)	x86-64
boolean	bool	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
<b>(reference)</b>	<b>pointer *</b>	<b>4</b>	<b>8</b>

address size = word size

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

# Address of Multibyte Data

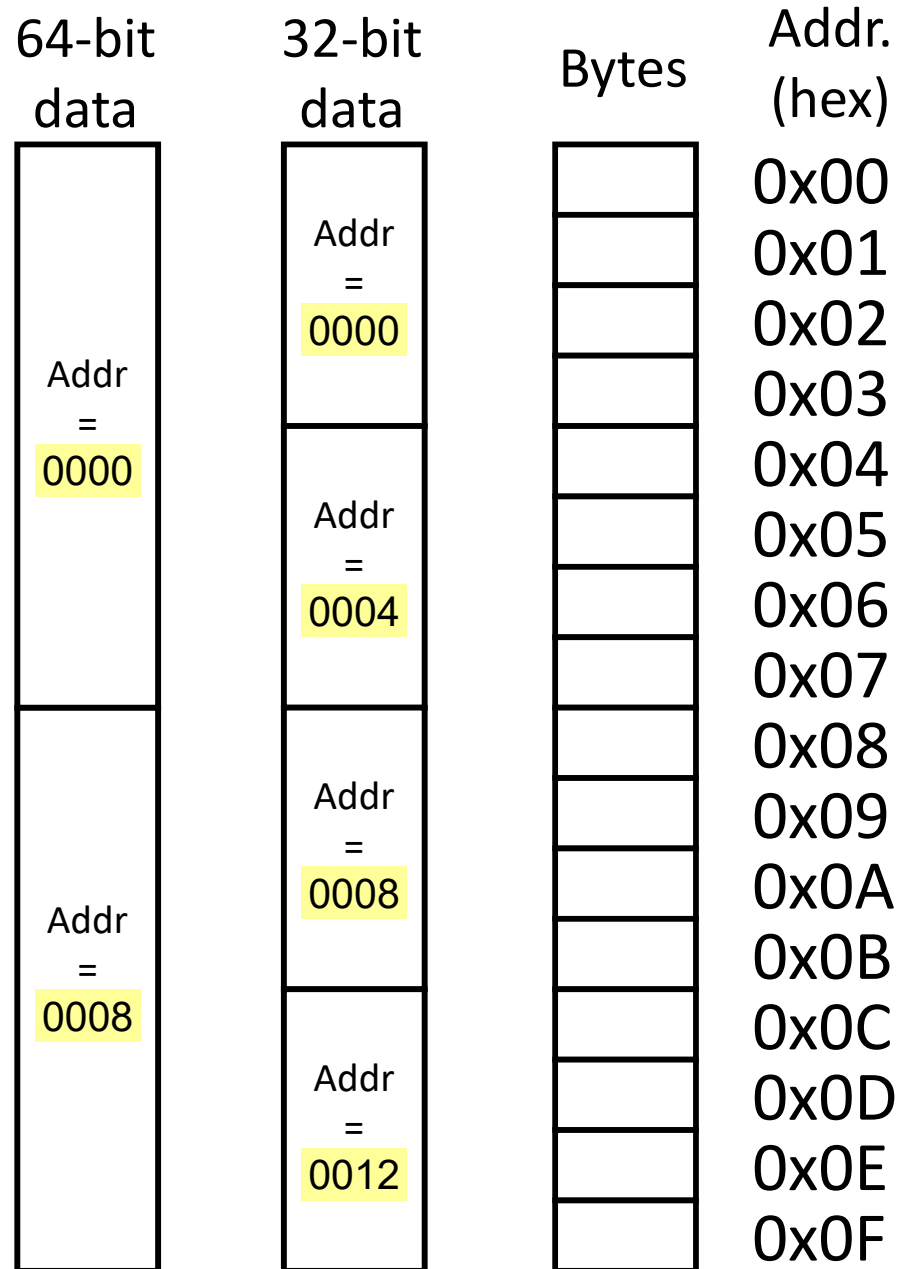
- ❖ Addresses still specify locations of bytes in memory, but we can choose to *view* memory as a series of chunks of fixed-sized data instead
  - Addresses of successive chunks differ by data size
  - Which byte's address should we use for each word?
- ❖ **The address of *any* chunk of memory is given by the address of the first byte**
  - To specify a chunk of memory, need *both* its **address** and its **size**





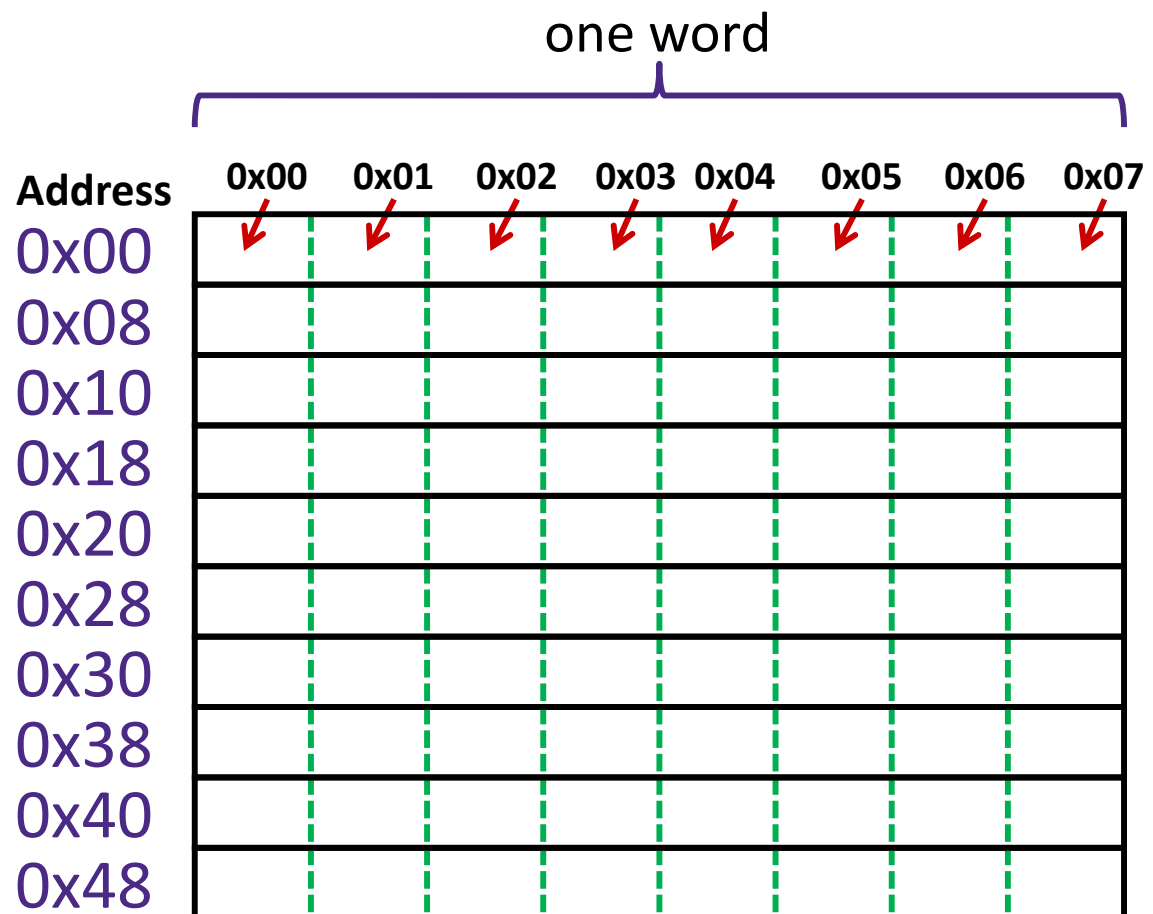
# Alignment

- ❖ 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



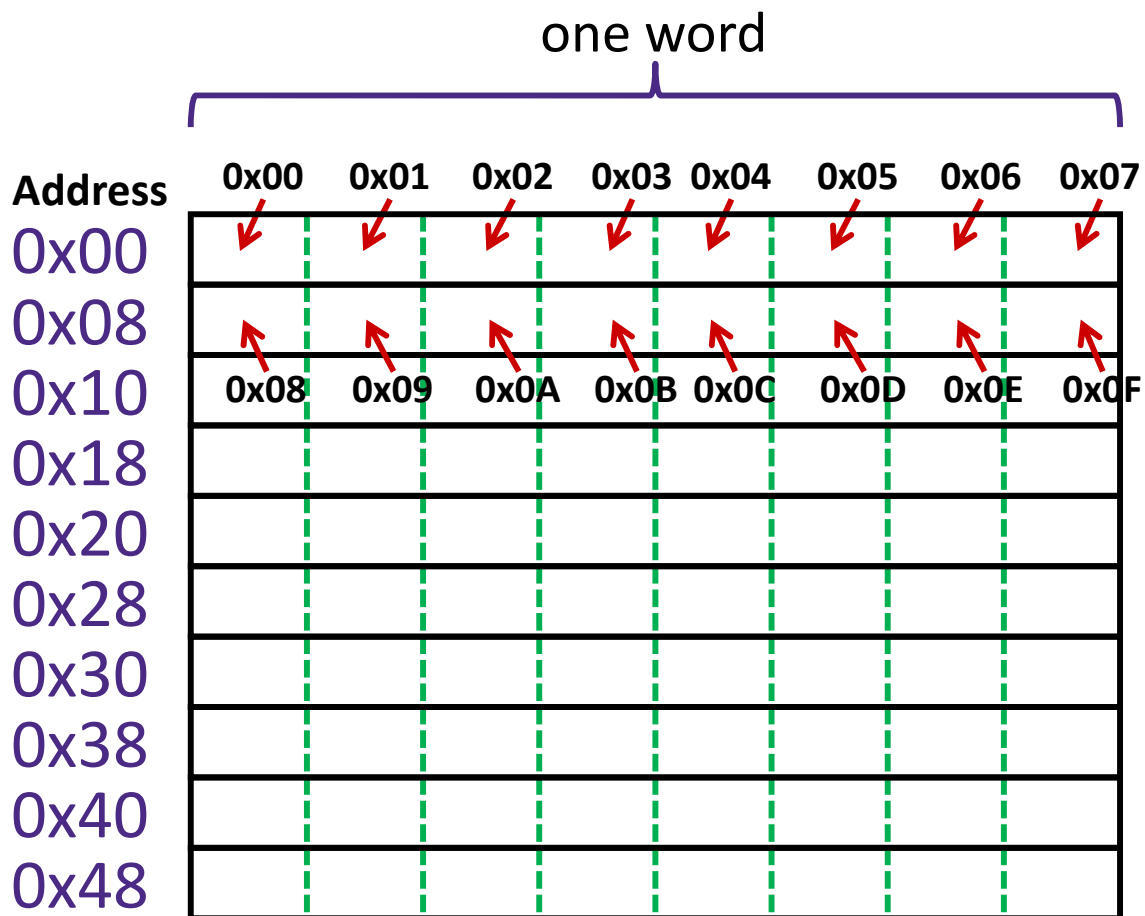
# A Picture of Memory (64-bit view)

- ❖ 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

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

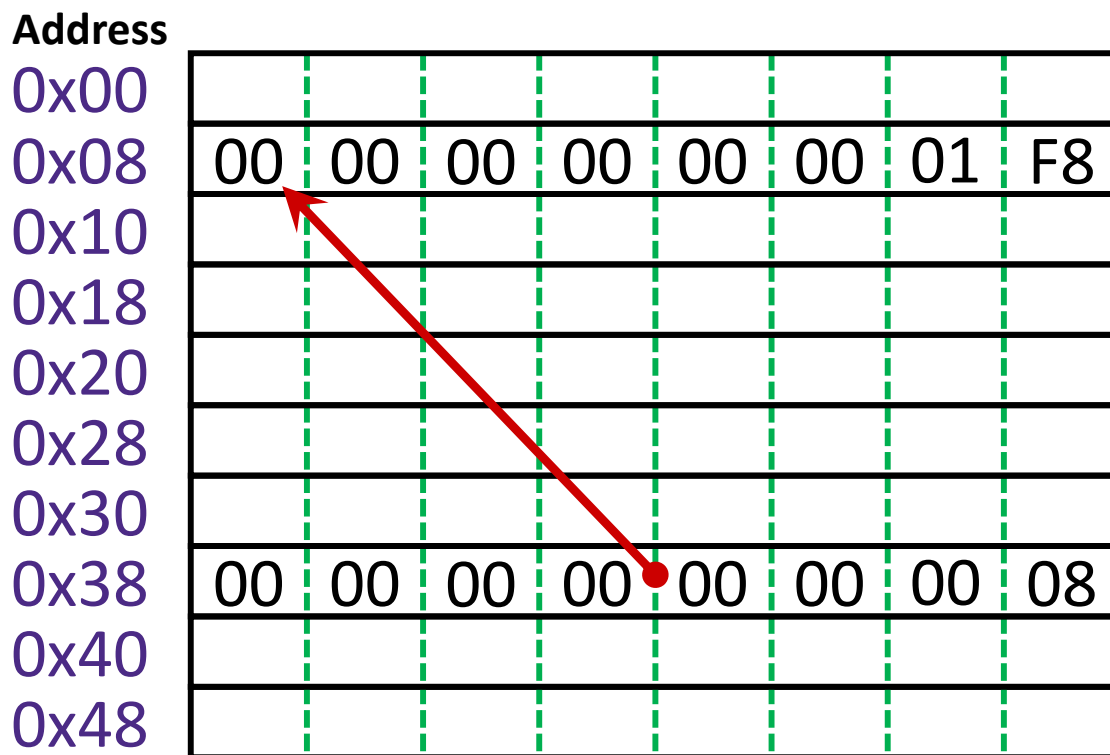
big-endian

- ❖ An *address* refers to a location in memory
- ❖ A *pointer* is a data object that holds an address
  - Address can point to *any* data

❖ Value 504 stored at address **0x08**

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

❖ Pointer stored at **0x38** points to address **0x08**



# 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

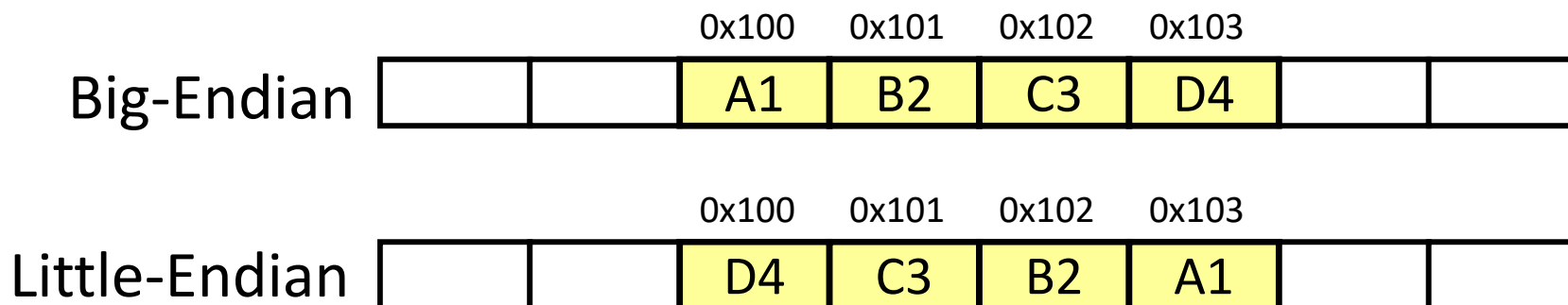
0x00								
0x08	00	00	00	00	00	00	01	F8
0x10								
0x18								
0x20								
0x28								
0x30								
0x38	00	00	00	00	00	00	00	08
0x40								
0x48	00	00	00	00	00	00	00	38

# Byte Ordering

- ❖ 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

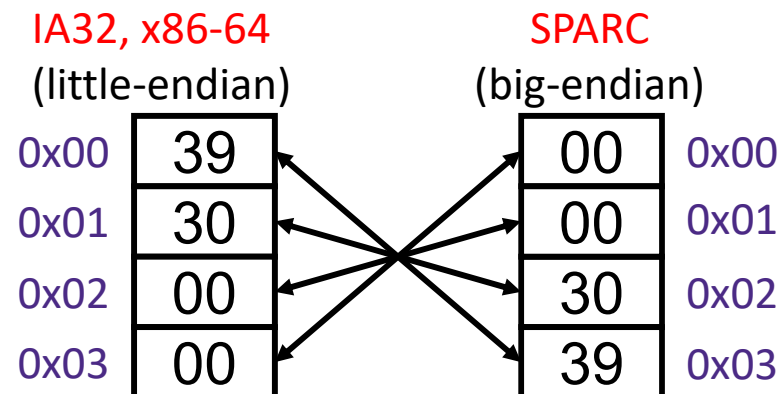
- ❖ Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- ❖ Little-endian (x86, x86-64)
  - Least significant byte has lowest address
- ❖ Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little
- ❖ **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



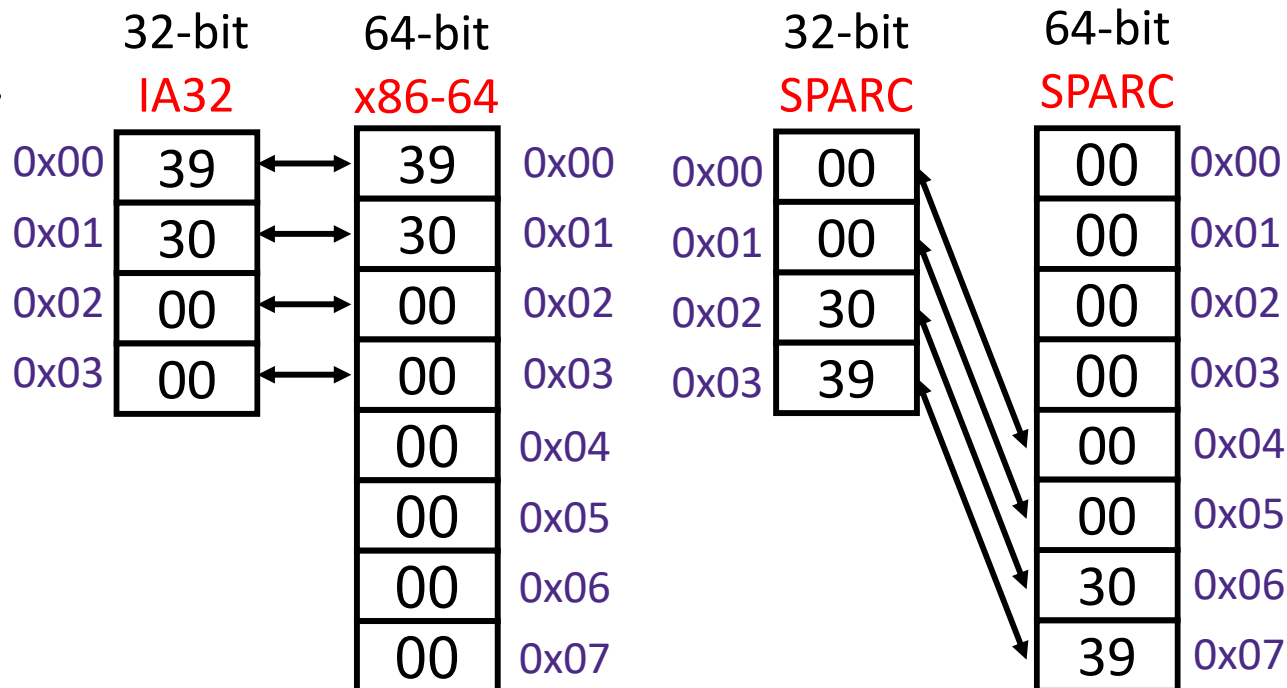
# Byte Ordering Examples

Decimal:	12345
Binary:	0011 0000 0011 1001
Hex:	3 0 3 9

```
int x = 12345;
// or x = 0x3039;
```



```
long int y = 12345;
// or y = 0x3039;
```



(A long int is the size of a word)



# Polling Question

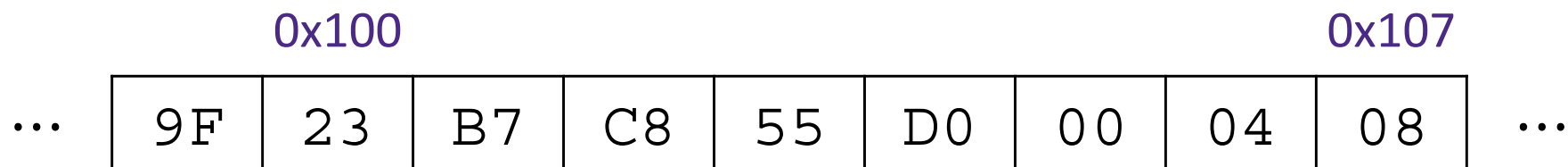
- ❖ We store the value `0x 01 02 03 04` as a ***word*** at address `0x100` in a big-endian, 64-bit machine
  - ❖ What is the ***byte of data*** stored at address `0x104`?
    - Vote in Ed Lessons
- 
- A. **0x04**
  - B. **0x40**
  - C. **0x01**
  - D. **0x10**
  - E. **We're lost...**

# 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)

# Challenge Question

- ❖ Assume the state of memory is as shown below for a little-endian machine.



- ❖ If we (1) *read* the value of an `int` at address 0x102, (2) add 8 to it, and then (3) store the new value as an `int` at address 0x104, which of the following addresses retain their original value?

- A. 0x102    B. 0x104    C. 0x105    D. 0x107

# 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