

Memory, Data, & Addressing I

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

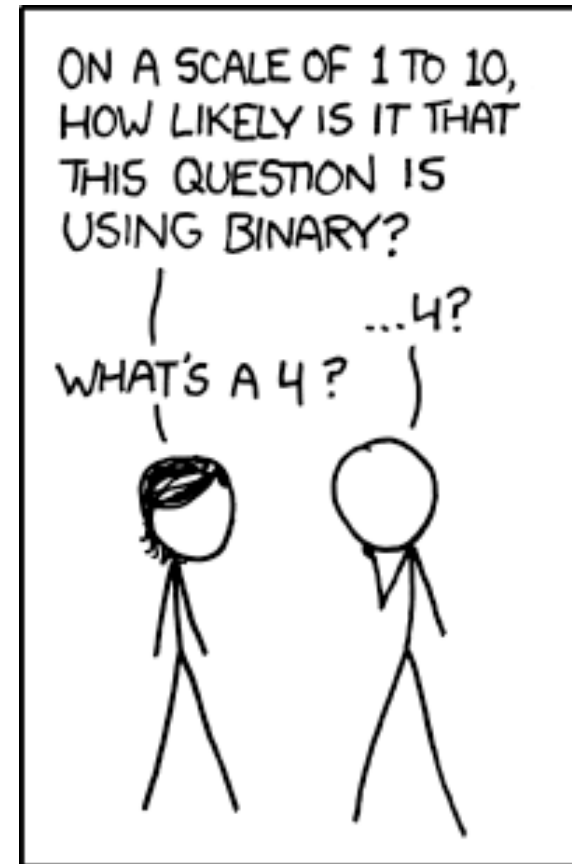
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



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

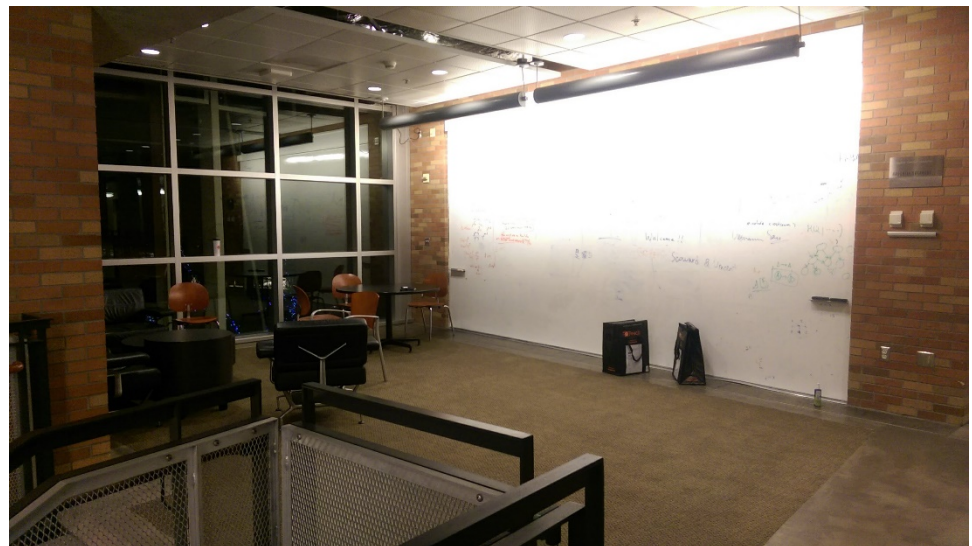
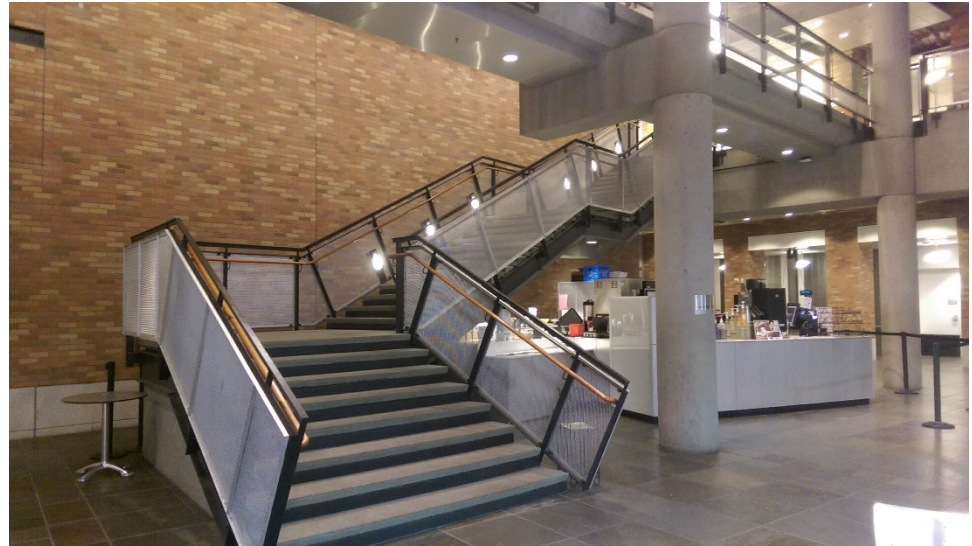
Administrivia

- ❖ Pre-Course Survey due Friday @ 11:59 pm
- ❖ Lab 0 due Monday (1/14)
- ❖ Homework 1 due Wednesday (1/16)
- ❖ All course materials can be found on the website schedule

TA Office Hours

❖ CSE 2nd floor breakout

- Up the stairs in the CSE Atrium (next to the café)
- At the top of that first flight, the open area with the whiteboard wall is the 2nd floor breakout!



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
```

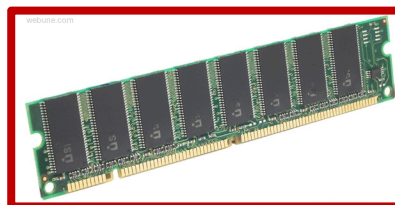
Machine
code:

```
0111010000011000  
100011010000010000000010  
1000100111000010  
110000011111101000011111
```

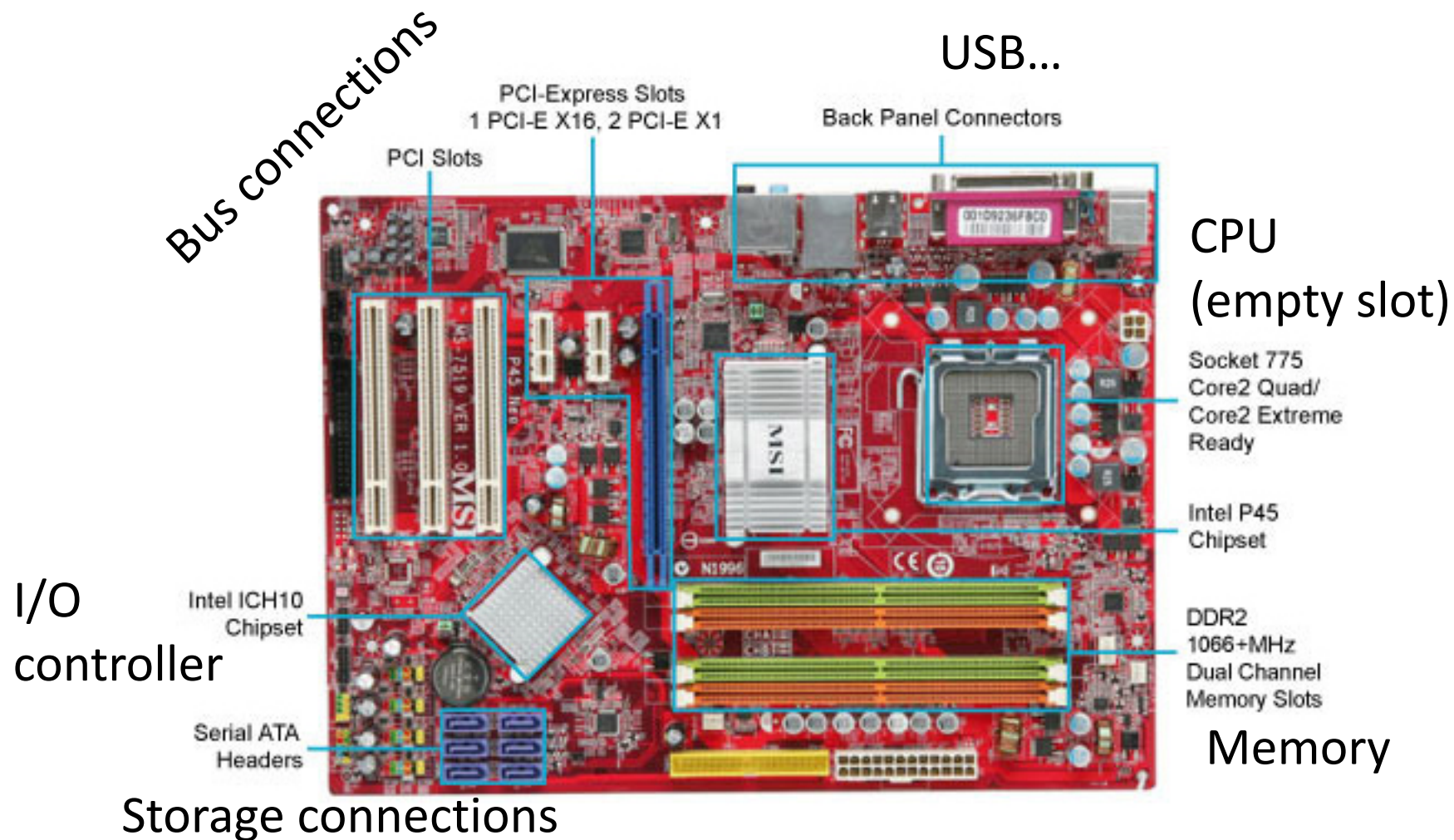
OS:



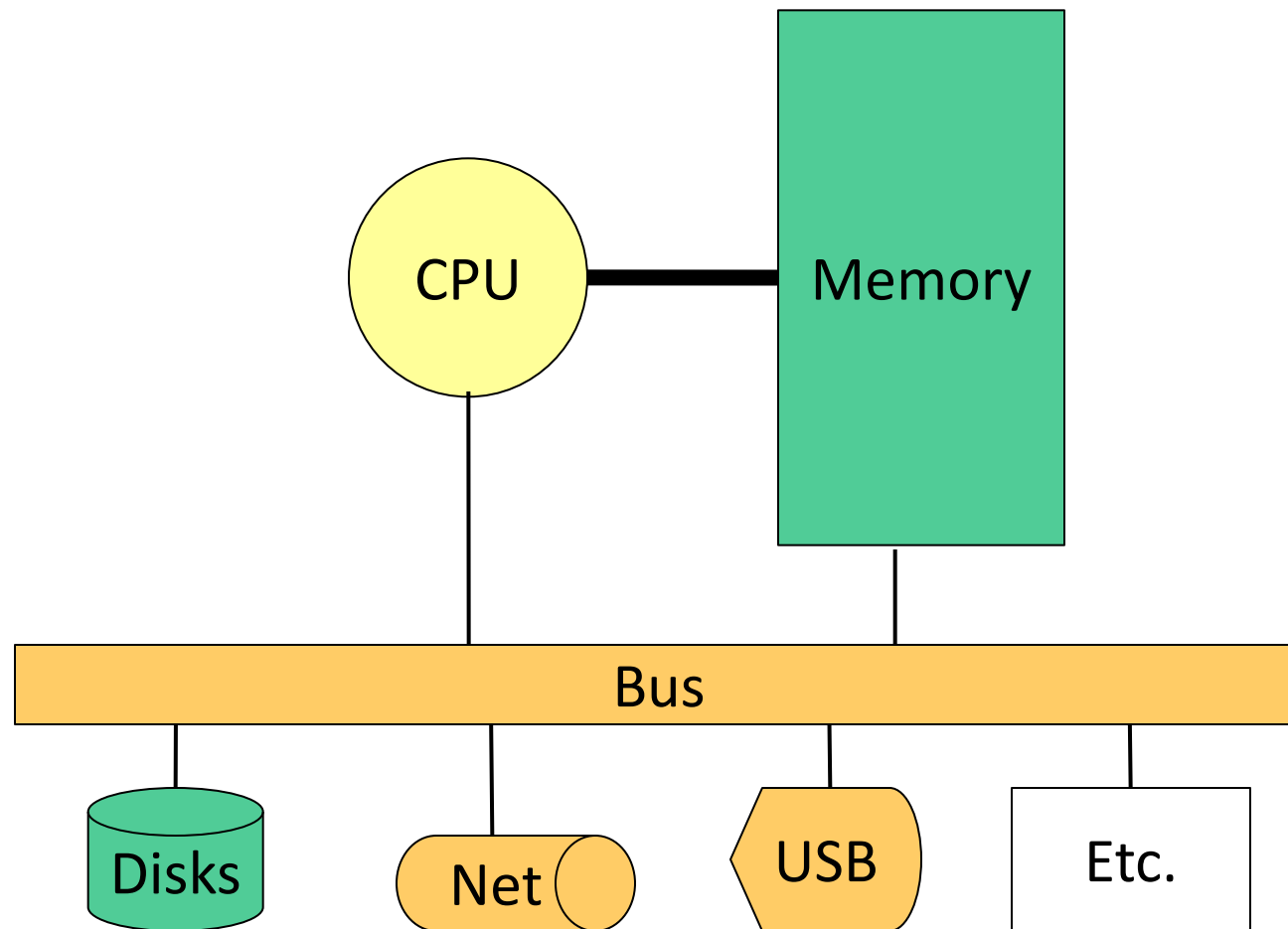
Computer
system:



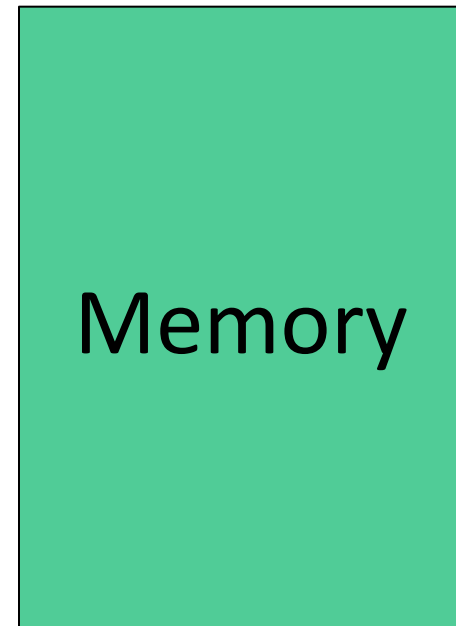
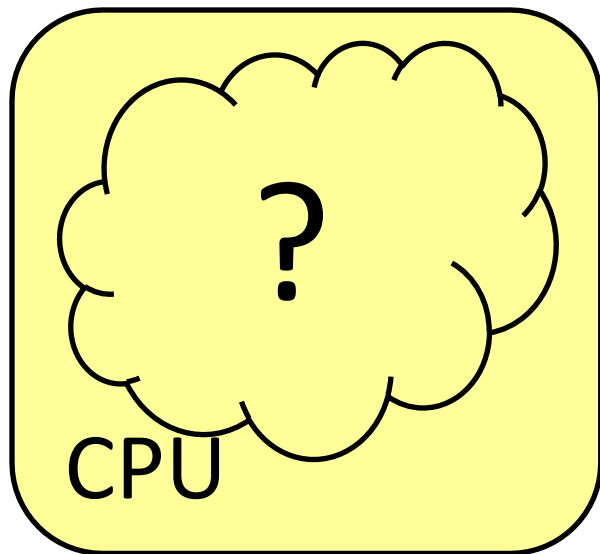
Hardware: Physical View



Hardware: Logical View



Hardware: 351 View (version 0)



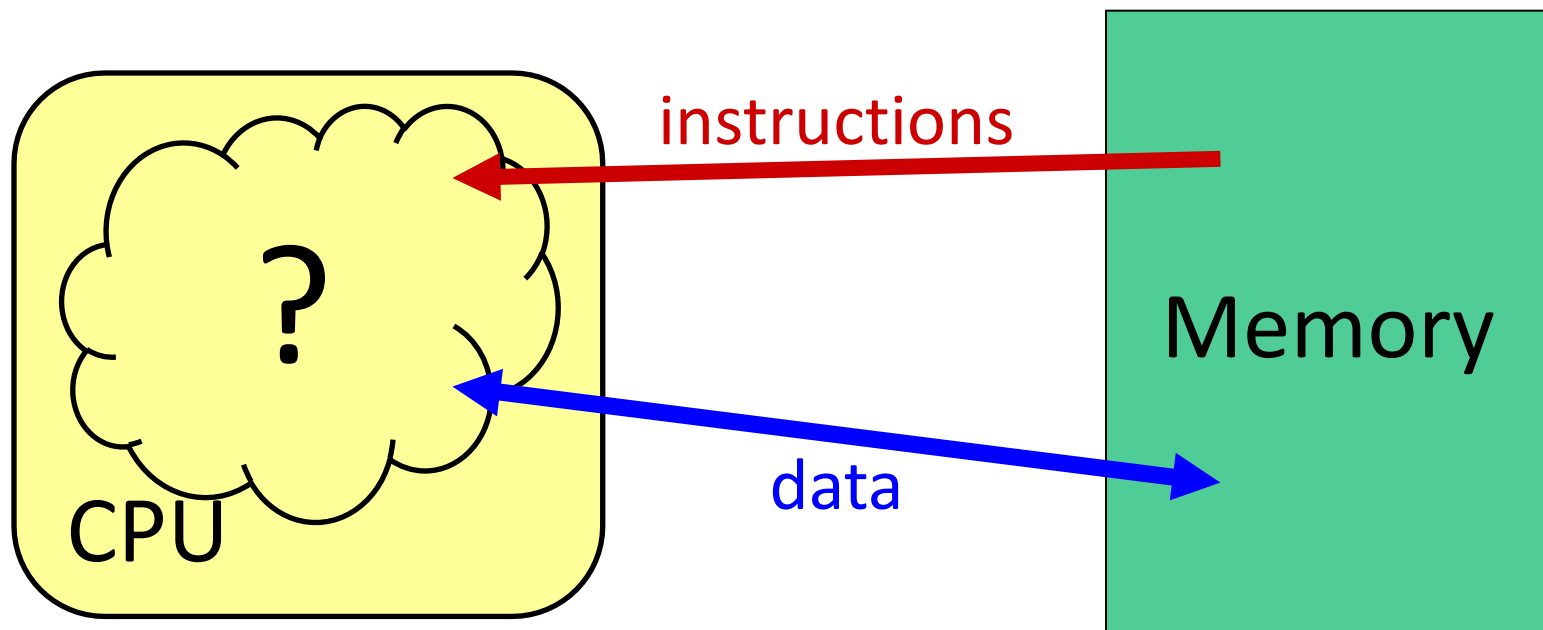
- ❖ The CPU **executes** instructions
- ❖ Memory **stores** data
- ❖ Binary encoding!
 - Instructions *are* just data

How are data
and instructions
represented?

Binary Encoding Additional Details

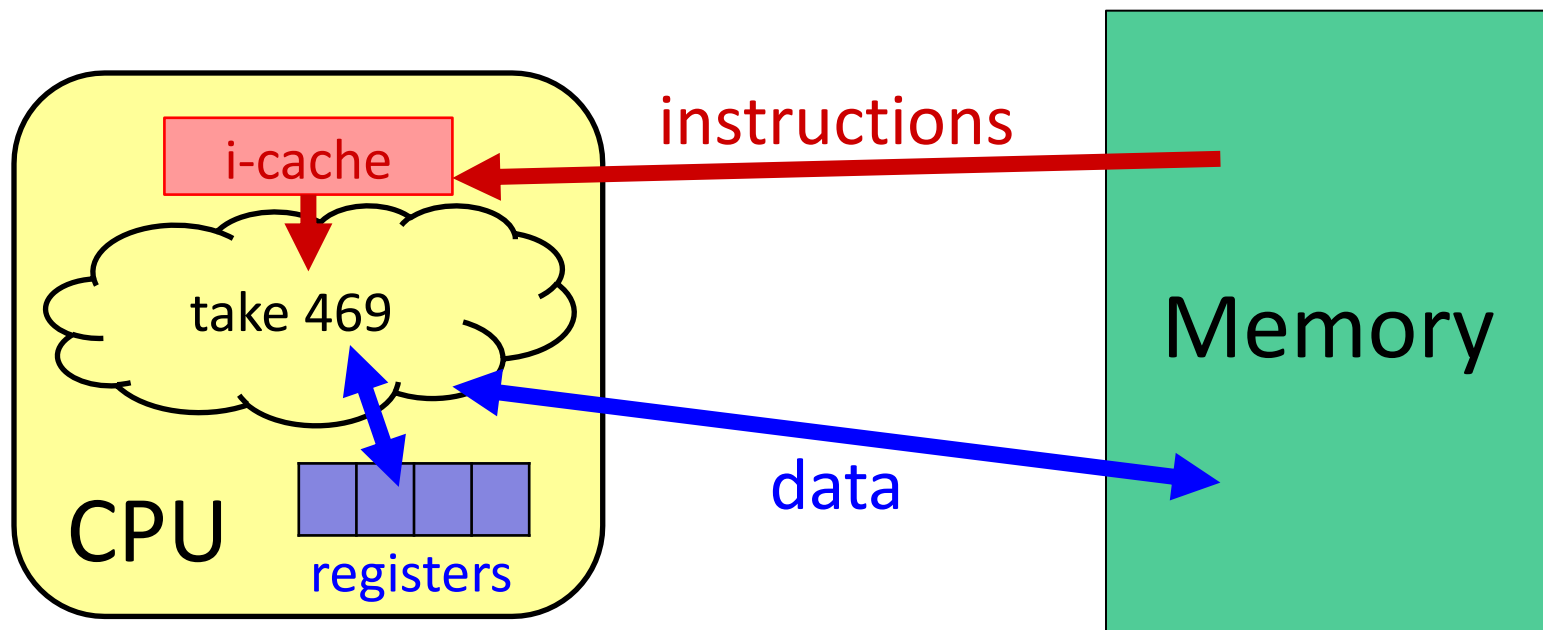
- ❖ Because storage is finite in reality, everything is stored as “fixed” length
 - Data is moved and manipulated in fixed-length chunks
 - Multiple fixed lengths (*e.g.* 1 byte, 4 bytes, 8 bytes)
 - Leading zeros now *must* be included up to “fill out” the fixed length
- ❖ Example: the “eight-bit” representation of the number 4 is 0b00000100
 - Most Significant Bit (MSB)
 - Least Significant Bit (LSB)

Hardware: 351 View (version 0)



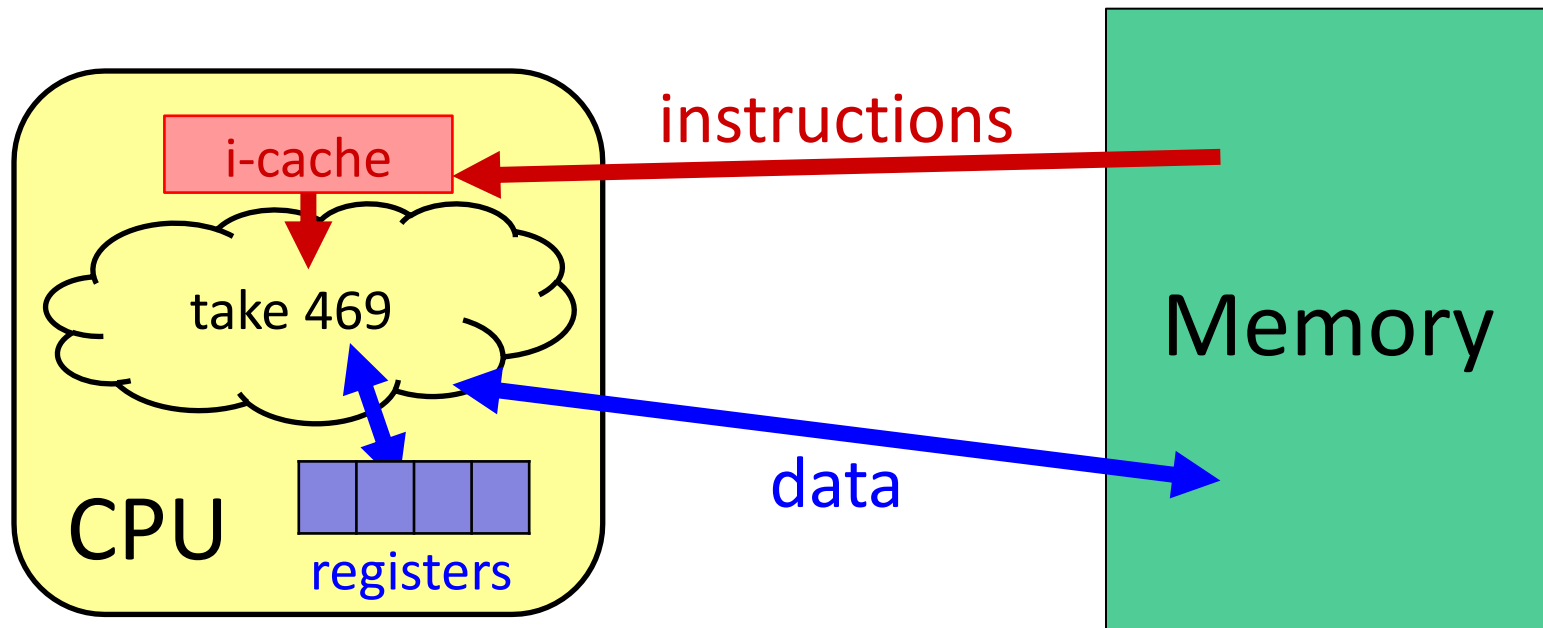
- ❖ To execute an instruction, the CPU must:
 - 1) Fetch the instruction
 - 2) (if applicable) Fetch data needed by the instruction
 - 3) Perform the specified computation
 - 4) (if applicable) Write the result back to memory

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)

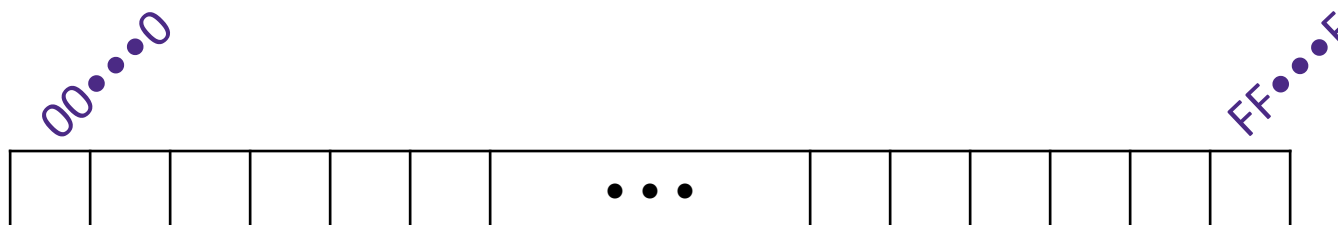
Hardware: 351 View (version 1)



- ❖ We will start by learning about Memory

How does a program find its data in memory?

Byte-Oriented Memory Organization



- ❖ 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

Peer Instruction Question

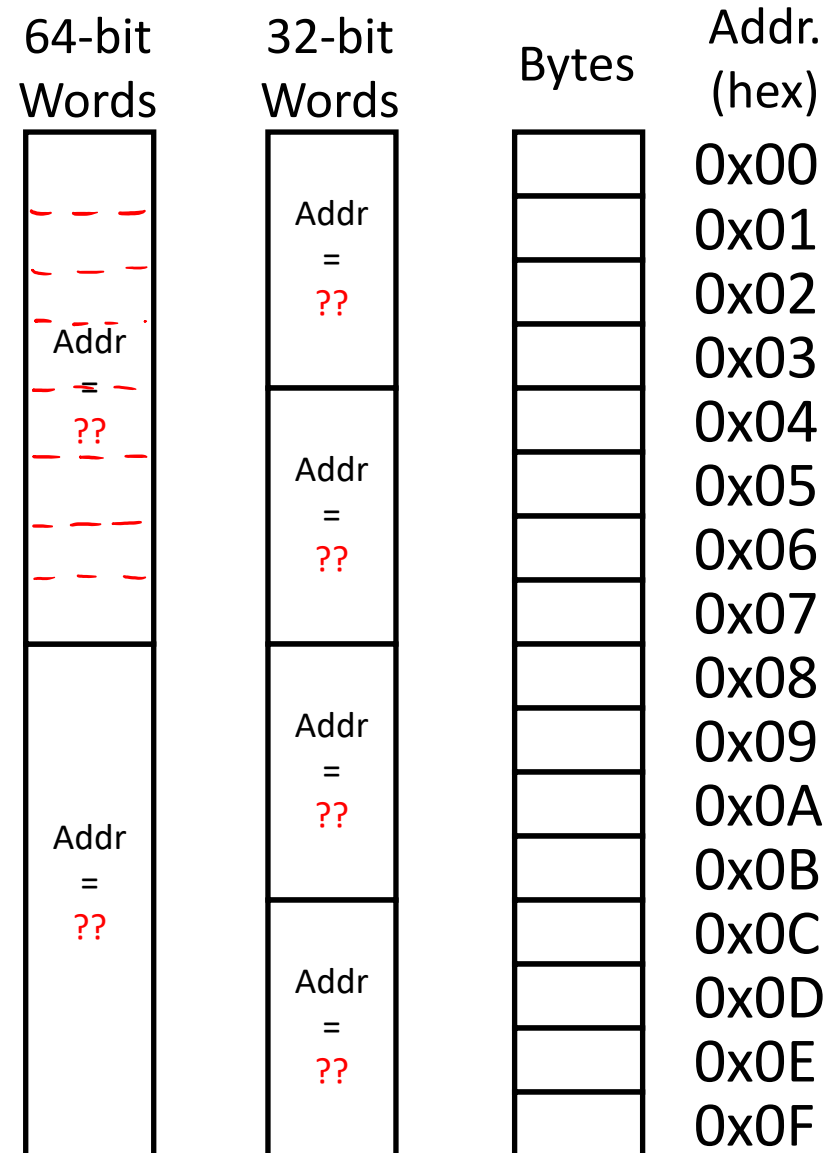
- ❖ If we choose to use 4-bit addresses, how big is our address space?
 - *i.e.* How much space can we “refer to” using our addresses?
- A. 16 bits
- B. 16 bytes
- C. 4 bits
- D. 4 bytes
- E. We're lost...

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 **1.8×10^{19} bytes**
= 18 billion billion bytes = 18 EB (exabytes)
 - Actual physical address space: **48 bits**

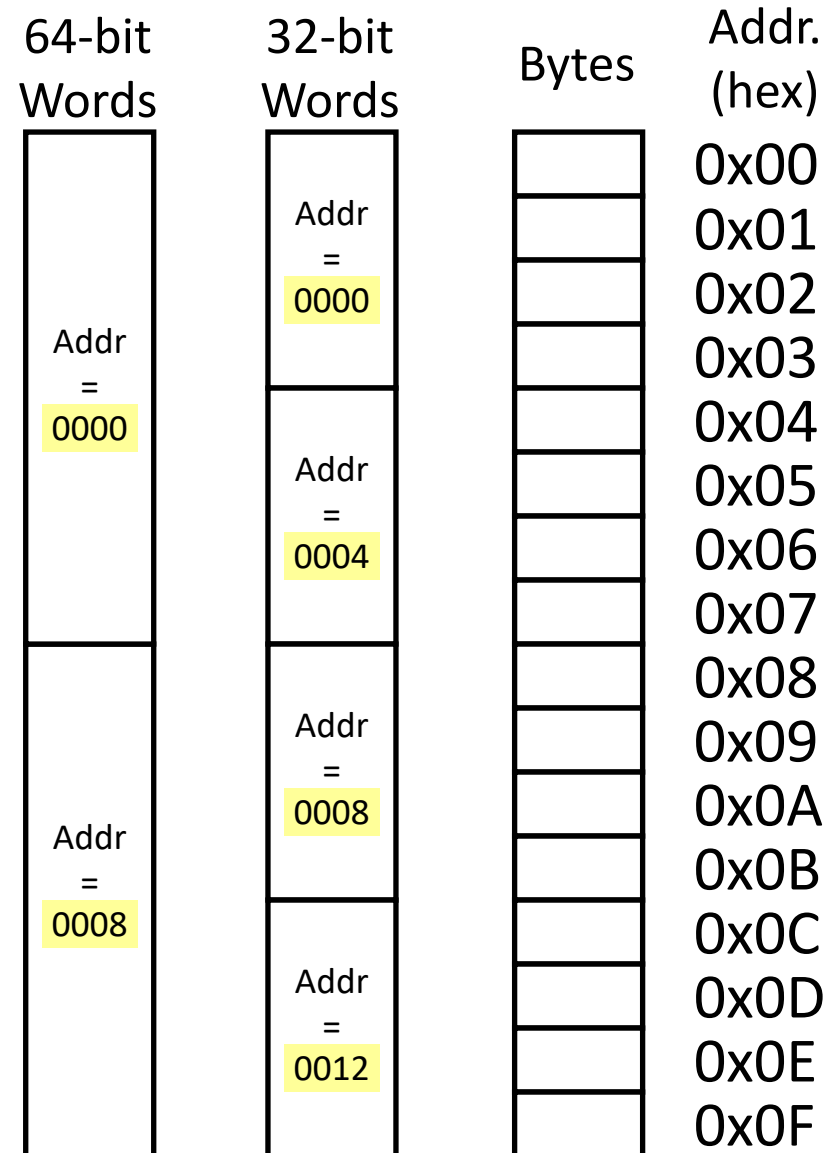
Word-Oriented Memory Organization

- ❖ Addresses still specify locations of *bytes* in memory
 - Addresses of successive words differ by word size (in bytes): *e.g.* 4 (32-bit) or 8 (64-bit)
 - Address of word 0, 1, ... 10?



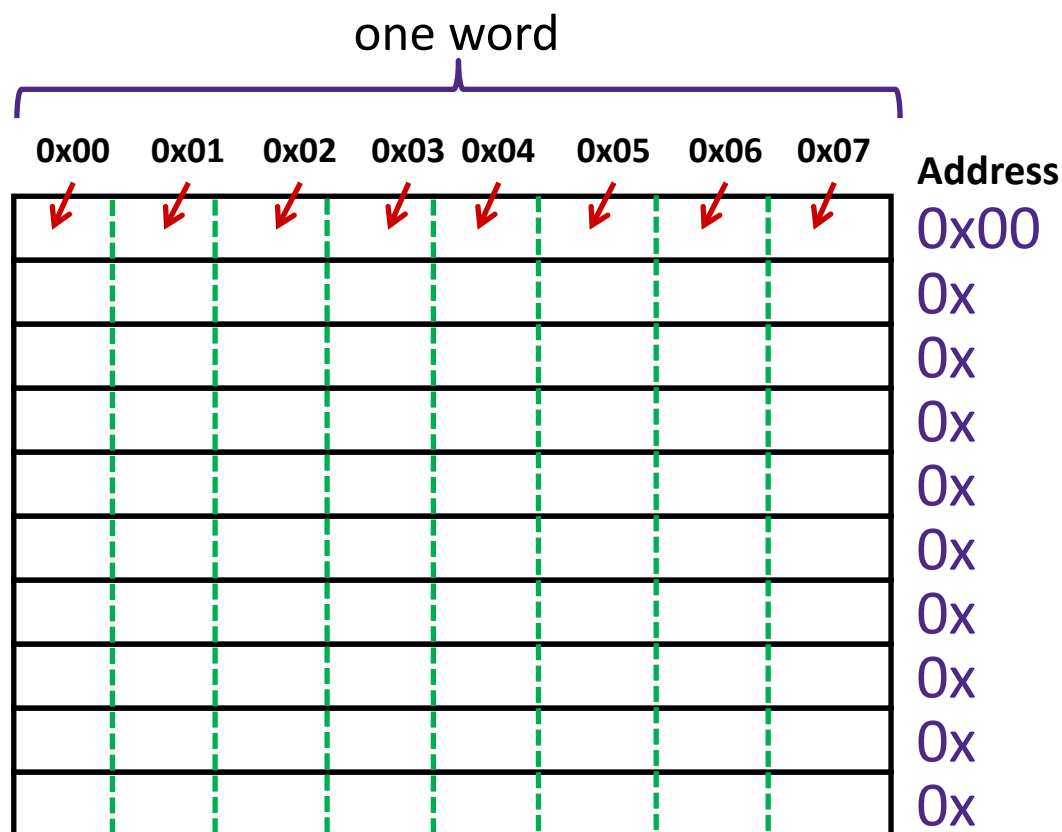
Word-Oriented Memory Organization

- ❖ Addresses still specify locations of *bytes* in memory
 - Addresses of successive words differ by word size (in bytes): *e.g.* 4 (32-bit) or 8 (64-bit)
 - Address of word 0, 1, ... 10?
- ❖ **Address of word**
 - = address of *first* byte in word
 - The address of *any* chunk of memory is given by the address of the first byte
 - ***Alignment***



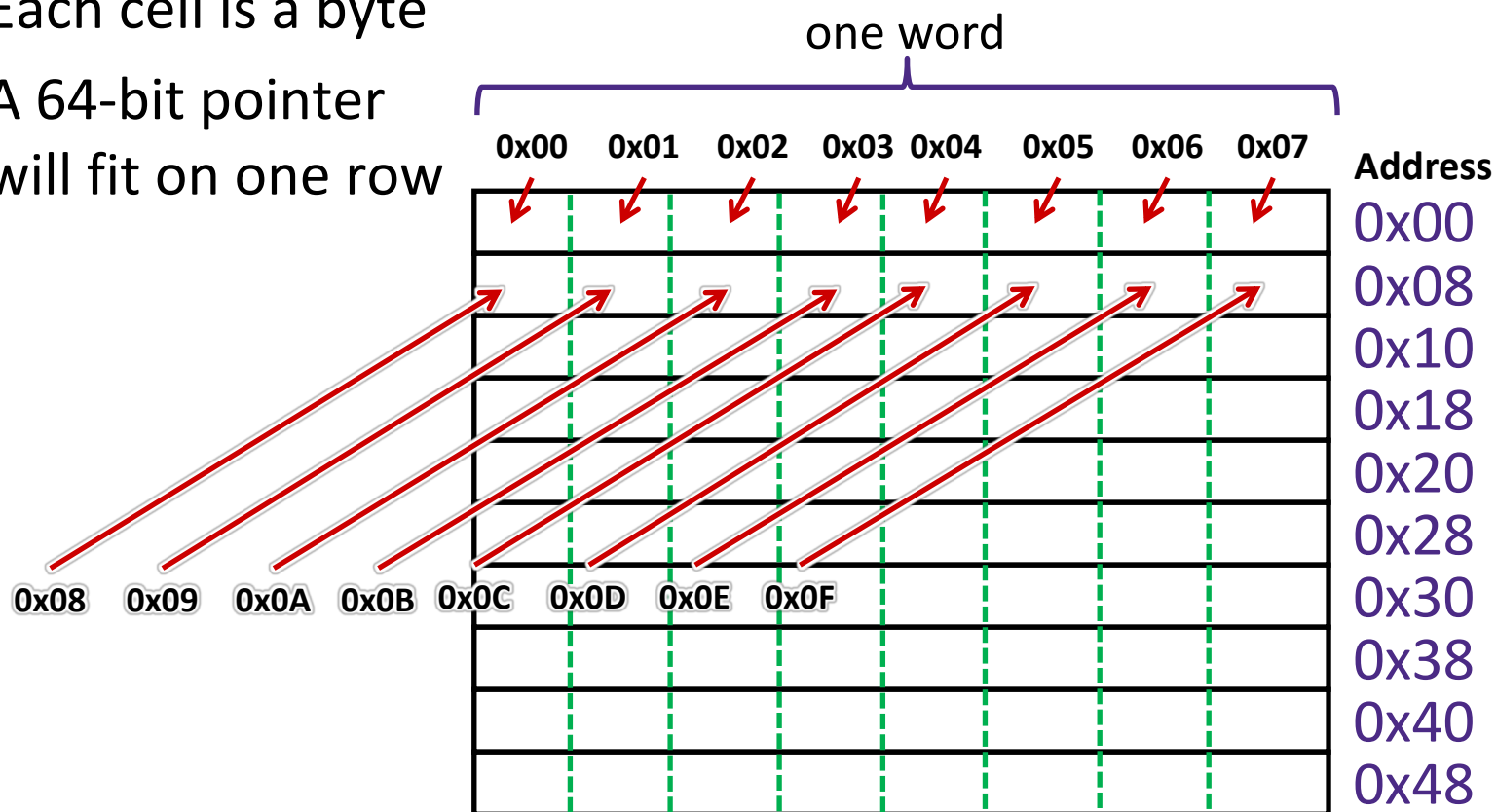
A Picture of Memory (64-bit view)

- ❖ 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
 - A 64-bit pointer 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
 - A 64-bit pointer will fit on one row



Addresses and Pointers

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

big-endian

- ❖ An *address* is 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**

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* is 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

The diagram illustrates a memory layout with 16 rows, each representing a 64-bit address. The addresses are listed on the right: 0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38, 0x40, and 0x48. The data in the rows is as follows: 0x08 contains '00 00 00 00 00 00 01 F8', 0x38 contains '00 00 00 00 00 00 00 08', and 0x48 contains '00 00 00 00 00 00 00 38'. Red arrows indicate that the pointer stored at address 0x48 points to address 0x38, and the pointer stored at address 0x38 points to address 0x08.

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	8	8
	long double	8	16
(reference)	pointer *	4	8

address size = word size

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

Memory Alignment

- ❖ **Aligned:** Primitive object of K bytes must have an address that is a multiple of K
 - More about alignment later in the course

K	Type
1	char
2	short
4	int, float
8	long, double, pointers

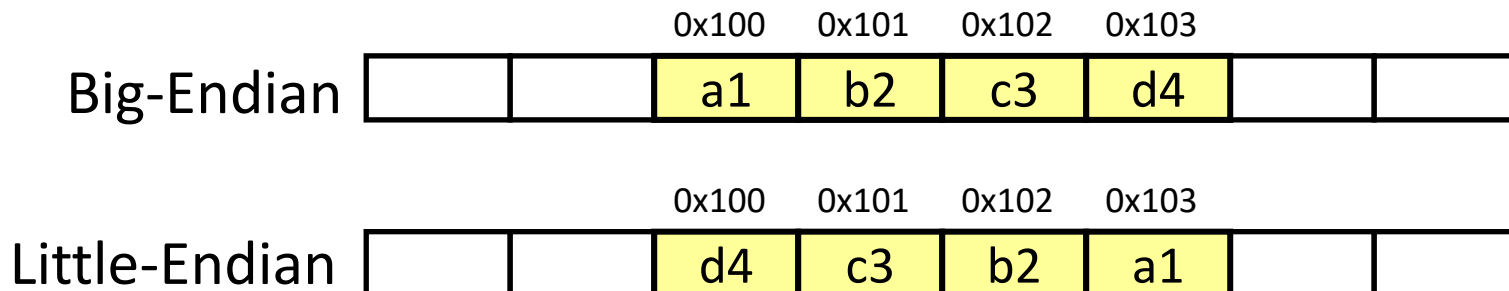
- ❖ For good memory system performance, Intel (x86) recommends data be aligned
 - However the x86-64 hardware will work correctly otherwise
 - Design choice: x86-64 instructions are *variable* bytes long

Byte Ordering

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

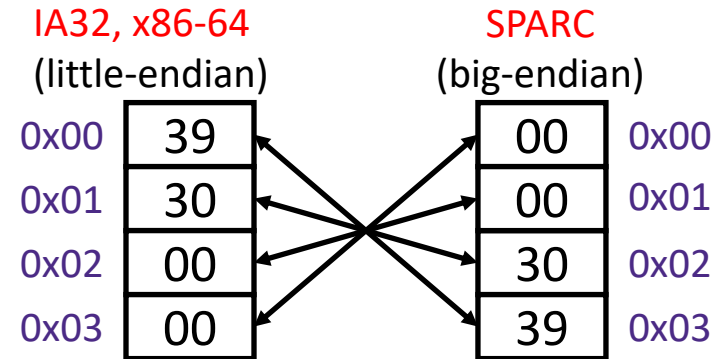
- ❖ Big-endian (SPARC, z/Architecture)
 - Least significant byte has highest address
- ❖ Little-endian (x86, x86-64)
 - Least significant byte has lowest address
- ❖ Bi-endian (ARM, PowerPC)
 - Endianness can be specified as big or little
- ❖ **Example:** 4-byte data 0xa1b2c3d4 at address 0x100



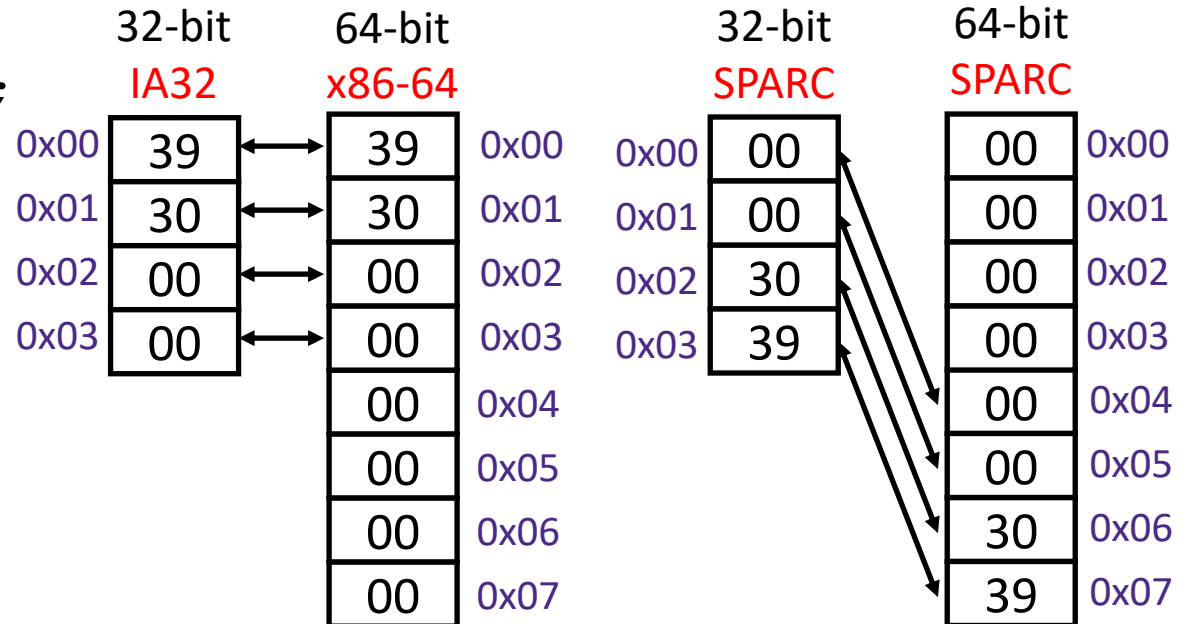
Byte Ordering Examples

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)

Peer Instruction 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**?
- 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