

CSE351: Memory, Data, & Addressing I

CSE 351 Winter 2017



<http://xkcd.com/138/>

Administrivia

- ❖ Start-of-Course survey due tomorrow at 5pm
- ❖ Lab 0 due Monday at 5pm
 - Who tried it?
- ❖ Consider taking CSE391 (System and Software tools)

- ❖ All course materials can be found on the website
 - Calendar link fixed, subscribe to it!
 - Section materials sidebar 
 - Readings: try to do them before class 
 - Slides posted weekend before classes. Ink saved.
 - Book: Sorry, really need 3rd edition.

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();
```

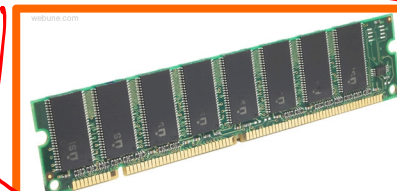
Assembly
language:

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

Machine
code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer
system:



Memory & data

Integers & floats

Machine code & C

x86 assembly

Procedures & stacks

Arrays & structs

Memory & caches

Processes

Virtual memory

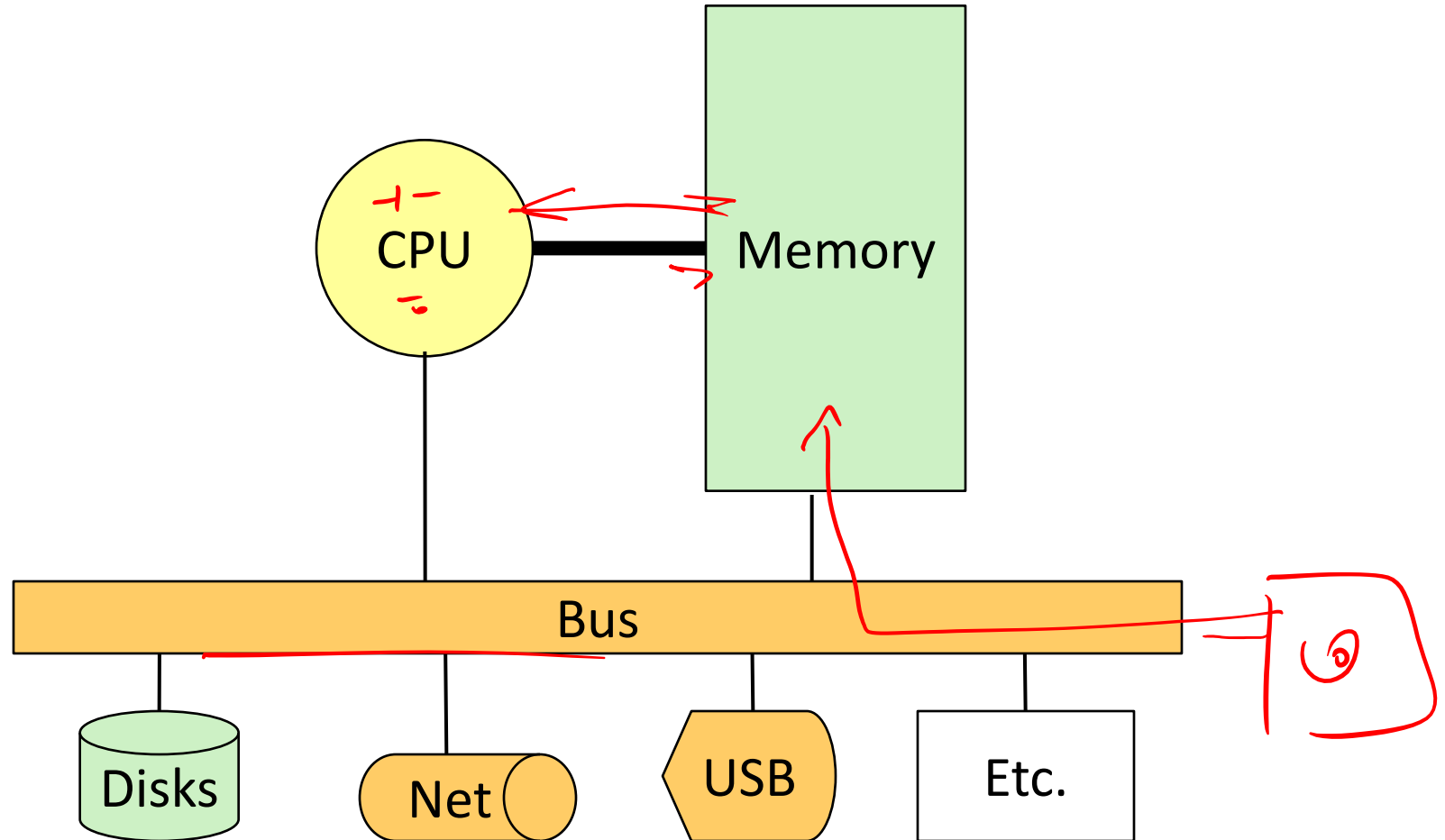
Memory allocation

Java vs. C

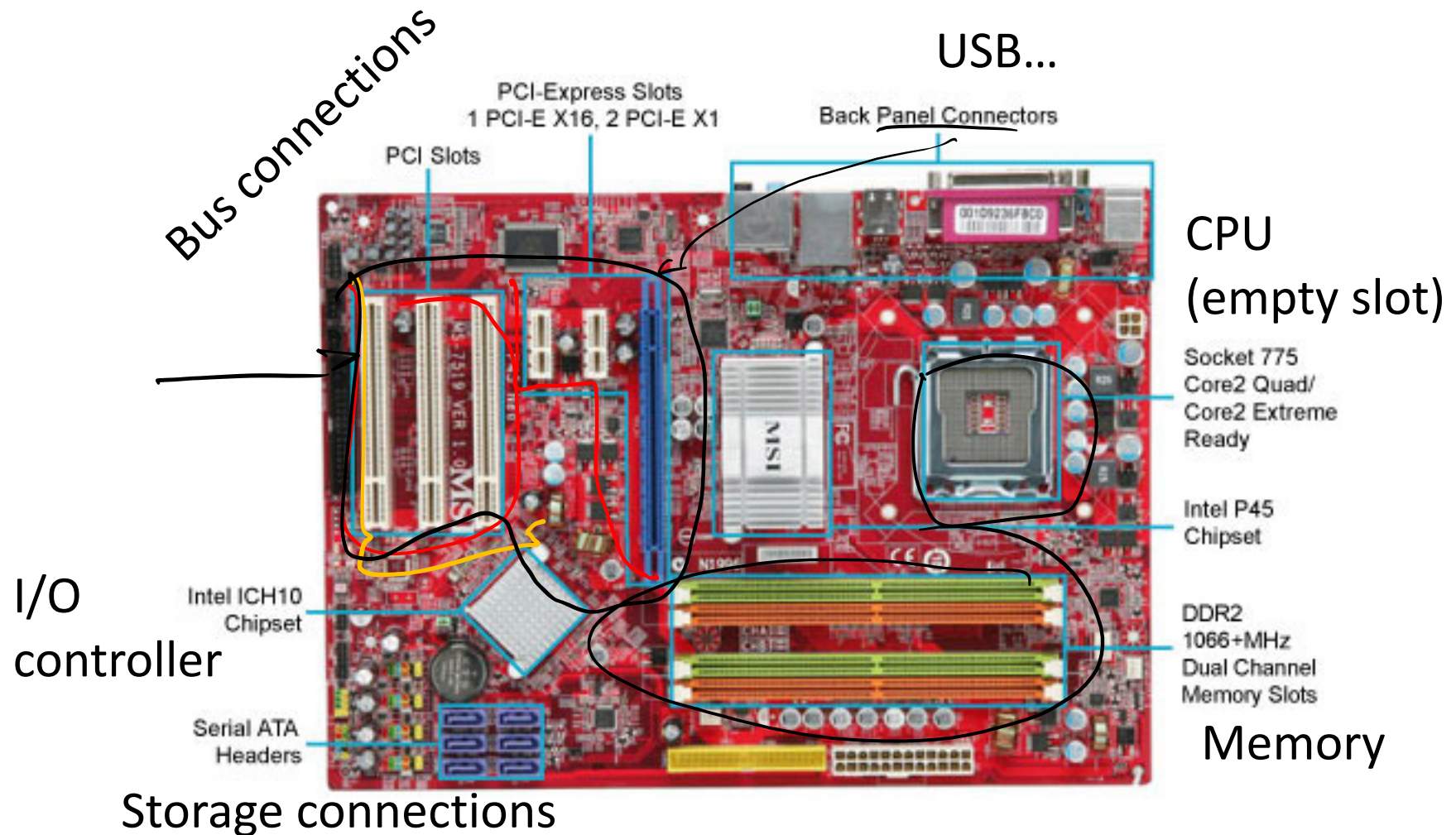
OS:



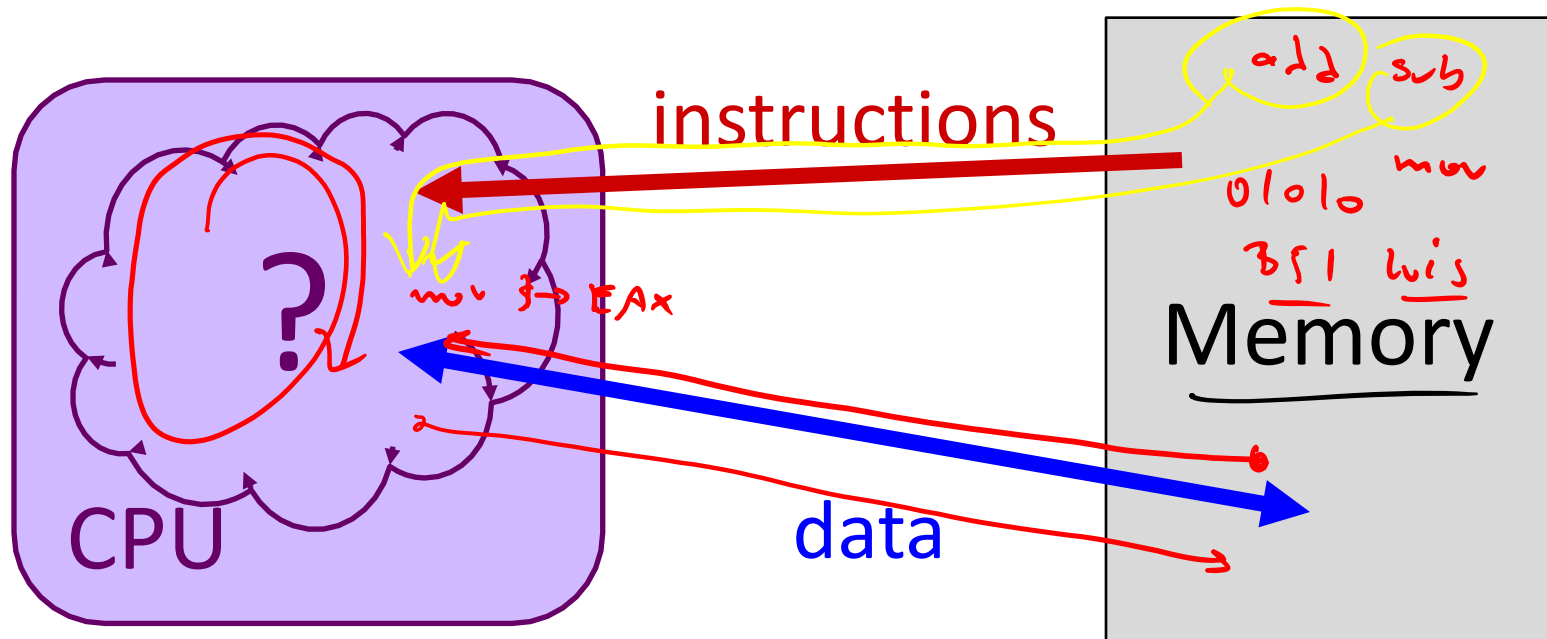
Hardware: Logical View



Hardware: Physical View

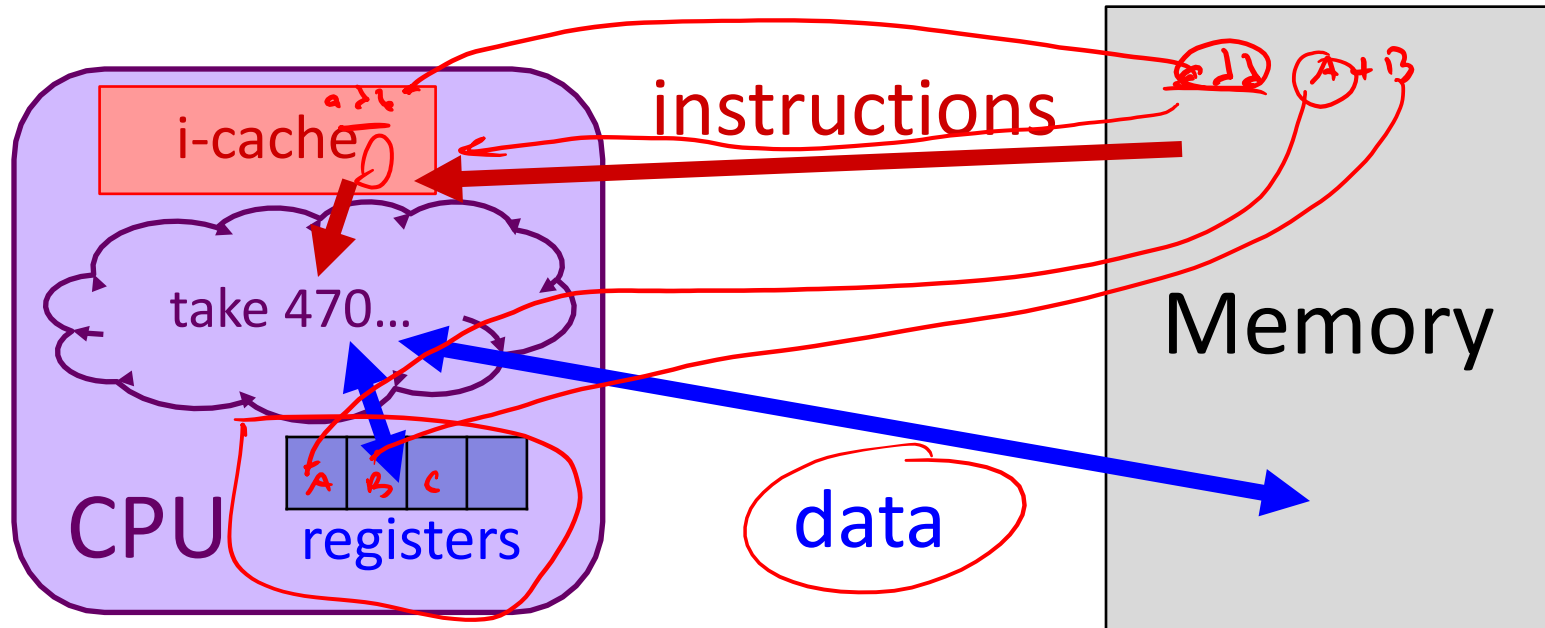


Hardware: 351 View (version 0)



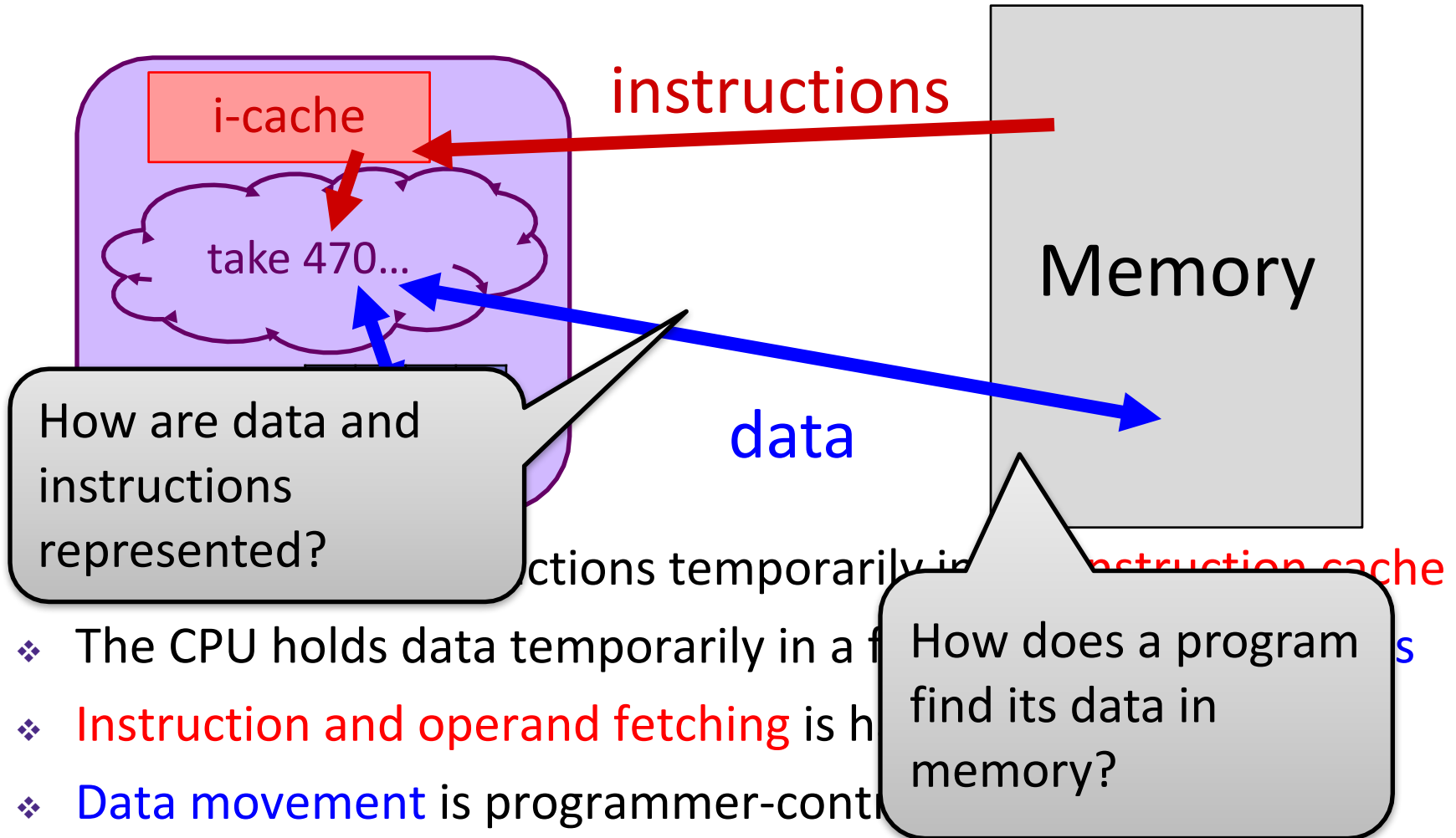
- ❖ CPU executes instructions; memory stores data
- ❖ To execute an instruction, the CPU must:
 - fetch an instruction;
 - fetch the data used by the instruction; and, finally,
 - execute the instruction on the data...
 - which may result in writing data back to memory

Hardware: 351 View (version 1)



- ❖ The CPU holds instructions temporarily in the **instruction cache**
- ❖ The CPU holds data temporarily in a fixed number of **registers**
- ❖ **Instruction and operand fetching** is hardware-controlled
- ❖ **Data movement** is programmer-controlled (in assembly)
- ❖ We'll learn about the instructions the CPU executes – take CSE/EE470 to find out how it actually executes them

Hardware: 351 View (version 1)

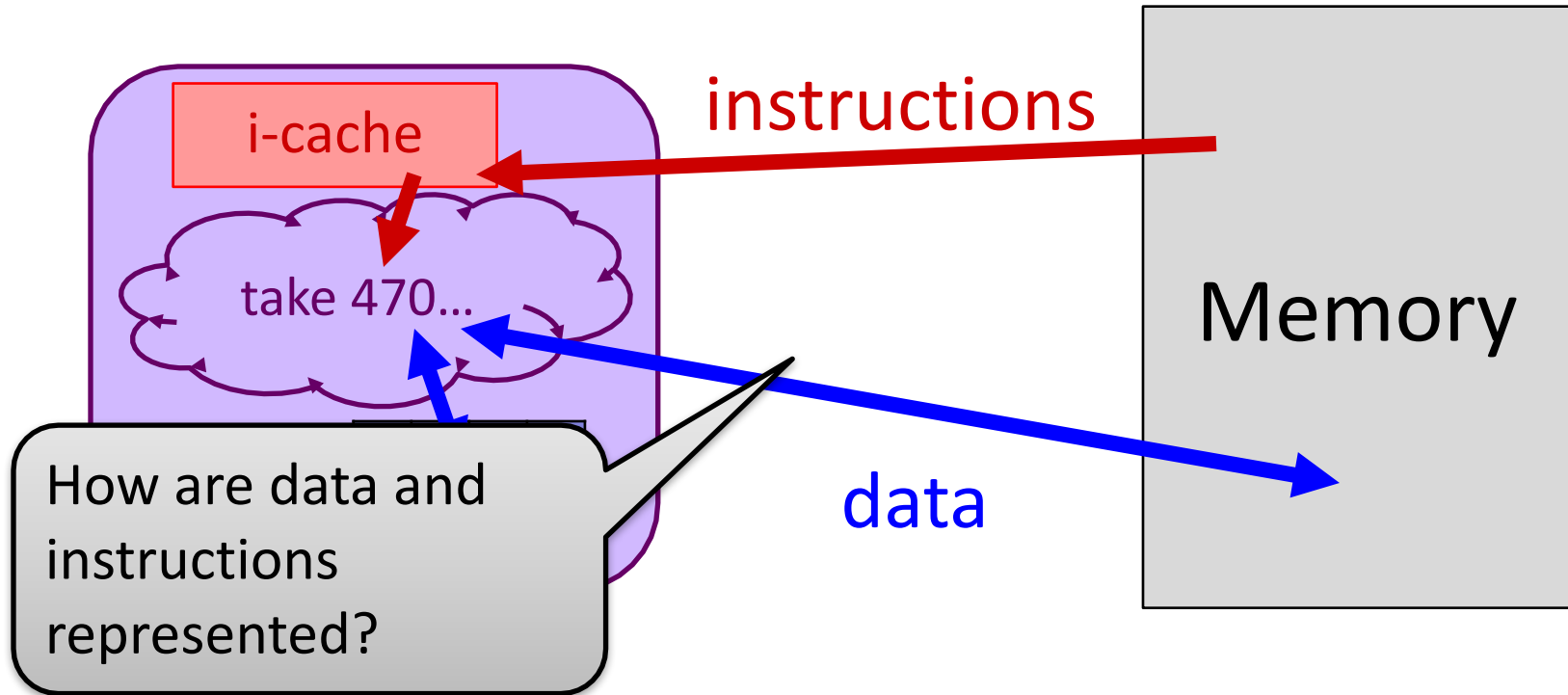


- ❖ The CPU holds data temporarily in a **data cache**
- ❖ **Instruction and operand fetching** is handled by the CPU
- ❖ **Data movement** is programmer-controlled
- ❖ We'll learn about the instructions the CPU executes – take CSE/EE470 to find out how it actually executes them

Memory, Data, and Addressing

- ❖ Representing information as bits and bytes
- ❖ Organizing and addressing data in memory
- ❖ Manipulating data in memory using C

Question 1:



Binary Representations

❖ Base 2 number representation

- A base 2 digit (0 or 1) is called a *bit*
- Represent 351_{10} as 0000000101011111_2 or 101011111_2

❖ *Why?*

Leading zeros



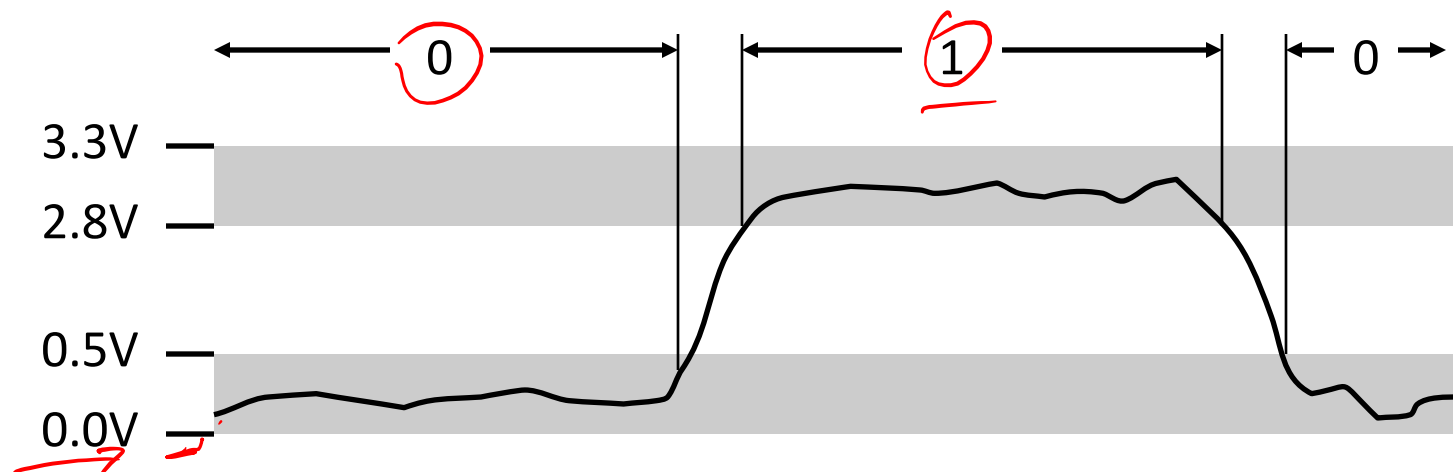
Binary Representations

❖ Base 2 number representation

- A base 2 digit (0 or 1) is called a bit
- Represent 351_{10} as 0000000101011111_2 or 101011111_2
Leading zeros

❖ Electronic implementation

- Easy to store with bi-stable elements
- Reliably transmitted on noisy and inaccurate wires



Review: Number Bases

- ❖ **Key terminology:** digit (d) and base (B)
 - In base B , each digit is one of B possible symbols
- ❖ Value of i -th digit is $d \times B^i$ where i starts at 0 and increases from right to left
 - n digit number $d_{n-1} d_{n-2} \dots d_1 d_0$
 - $\text{value} = d_{n-1} \times B^{n-1} + d_{n-2} \times B^{n-2} + \dots + d_1 \times B^1 + d_0 \times B^0$
 - In a *fixed-width* representation, left-most digit is called the **most-significant** and the right-most digit is called the **least-significant**
- ❖ **Notation:** Base is indicated using either a prefix or a subscript

Describing Byte Values

$$2^8 = 256$$

- ❖ Binary ($00000000_2 - 11111111_2$)

- Byte = 8 bits (binary digits)

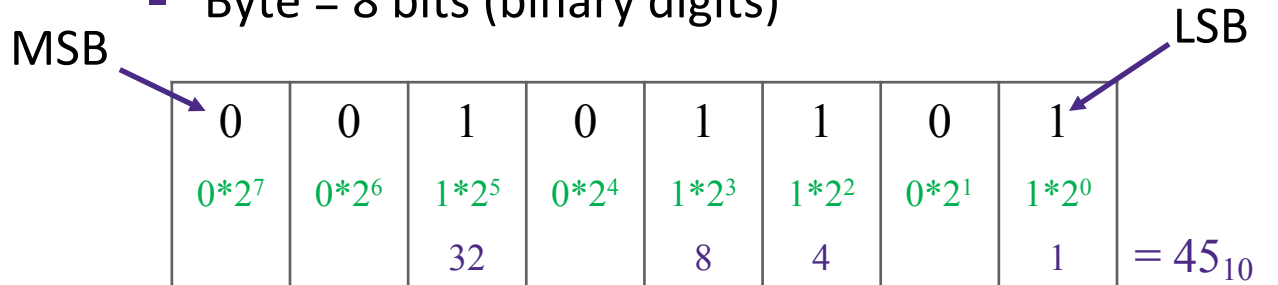
MSB									LSB
	0	0	1	0	1	1	0	1	
	$0 \cdot 2^7$	$0 \cdot 2^6$	$1 \cdot 2^5$	$0 \cdot 2^4$	$1 \cdot 2^3$	$1 \cdot 2^2$	$0 \cdot 2^1$	$1 \cdot 2^0$	
			32		8	4		1	
									$= 45_{10}$

- ❖ Decimal (0_{10} – 255_{10})
- ❖ 10 is not a power of 2 ☹.

Describing *Byte* Values

❖ Binary ($00000000_2 - 11111111_2$)

- Byte = 8 bits (binary digits)



❖ Decimal ($0_{10} - 255_{10}$)

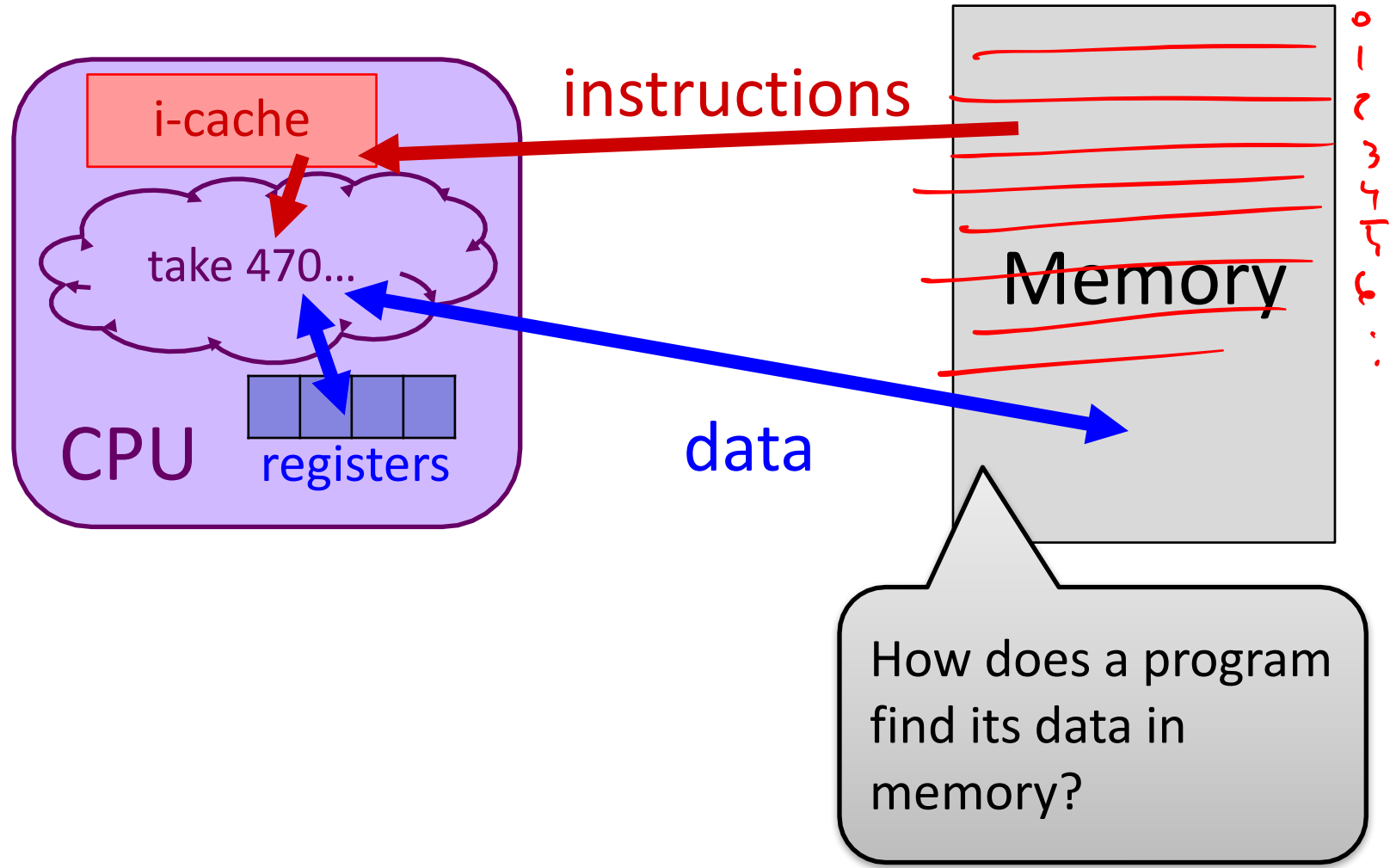
❖ Hexadecimal ($00_{16} - FF_{16}$)

- Byte = 2 hexadecimal (or “hex” or base 16) digits
- Base 16 number representation
- Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Write $FA1D37B_{16}$ in the C language
 - as `0xFA1D37B` or `0xfa1d37b`

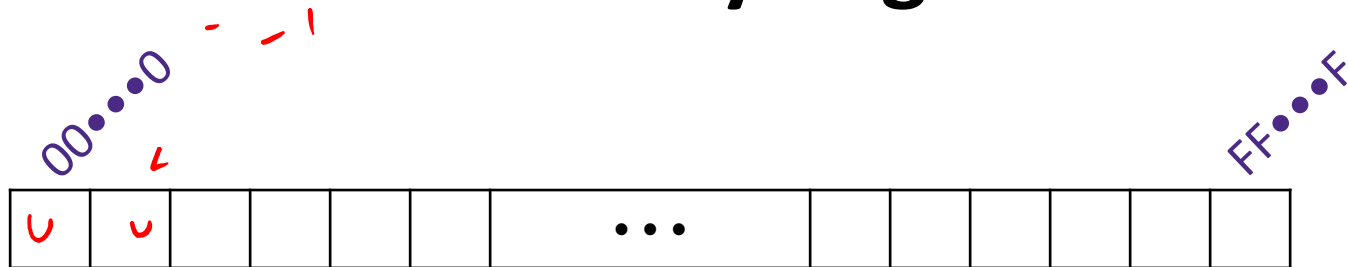
❖ More on specific data types later...

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
<u>F</u>	15	1111

Question 2:



Byte-Oriented Memory Organization



- ❖ Conceptually, memory is a single, large array of bytes, each with a unique address (index)
- ❖ The value of each byte in memory can be read and written
- ❖ Programs refer to bytes in memory by their *addresses*
 - Domain of possible addresses = *address space*
- ❖ But not all values (e.g., 351) fit in a single byte...
 - Store addresses to “remember” where other data is in memory
 - How much memory can we address with 1-byte (8-bit) addresses?
- ❖ Many operations actually use multi-byte values

Machine Words

- ❖ Word size = address size = register size
- ❖ Word size bounds the size of the *address space* and memory
 - 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) = 16 EiB (exbibytes)
 - Actual physical address space: **48 bits**



Aside: Units and Prefixes

- ❖ Here focusing on large numbers (exponents > 0)
- ❖ Note that $10^3 \approx 2^{10}$
- ❖ SI prefixes are ambiguous if base 10 or 2
- ❖ IEC prefixes are unambiguously base 2

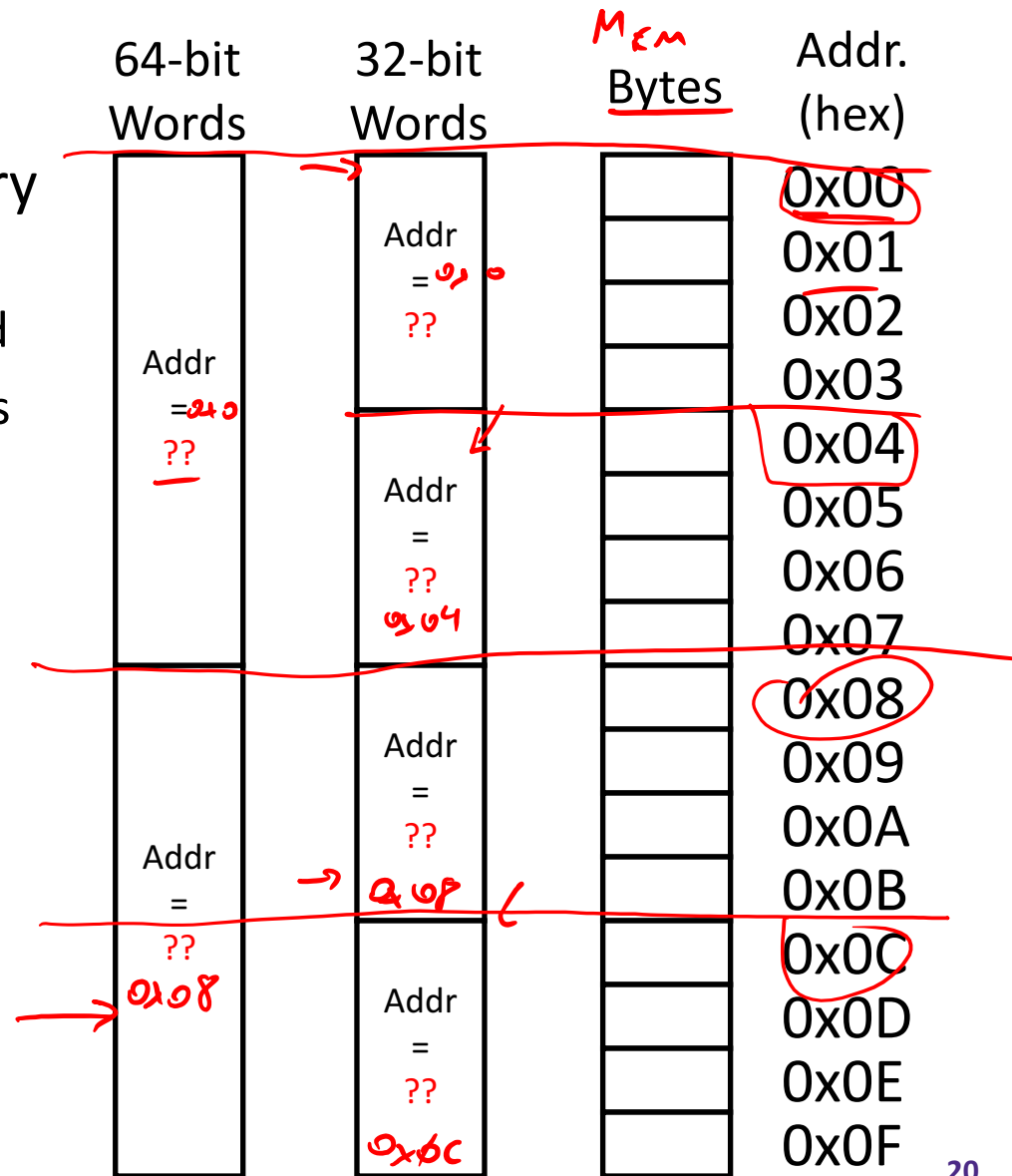
SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
<u>10^3</u>	<u>Kilo-</u>	K	<u>2^{10}</u>	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

Word-Oriented Memory Organization

- ❖ Addresses specify locations of bytes in memory

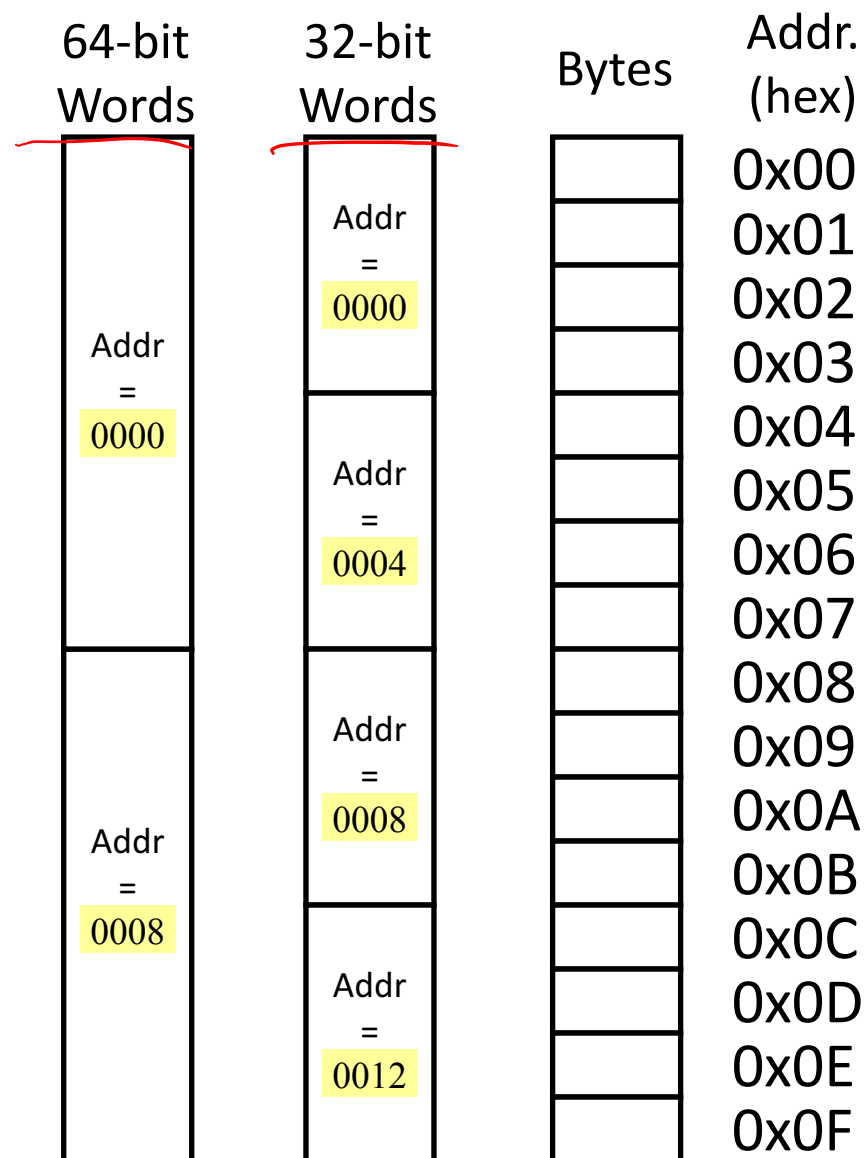
- Address of word
= address of first byte in word
- Addresses of successive words differ by word size (in bytes):
e.g., 4 (32-bit) or 8 (64-bit)
- Address of word 0, 1, ... 10?



Word-Oriented Memory Organization

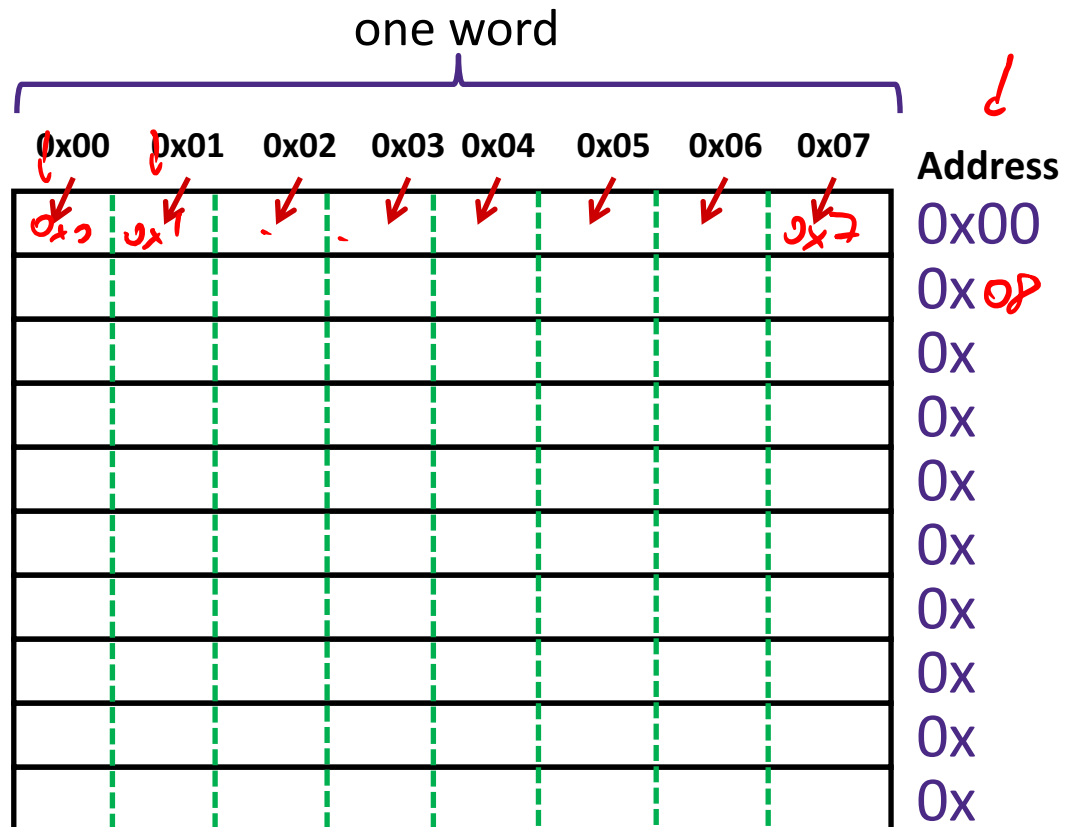
- ❖ Addresses still specify locations of *bytes* in memory

- Address of word
= address of first byte in word
- 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?
- Alignment



A Picture of Memory (64-bit view)

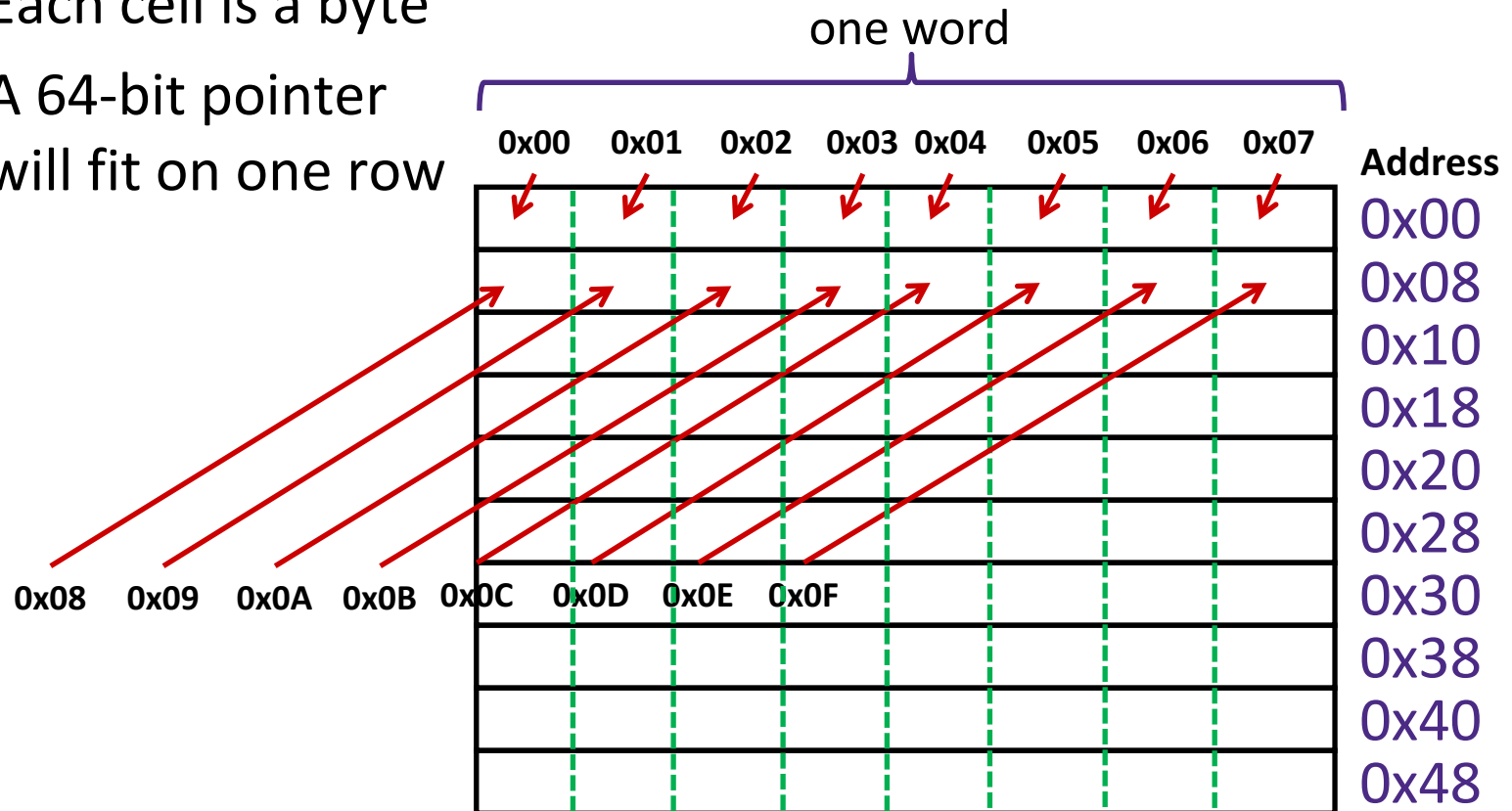
- ❖ A “64-bit (8-byte) word-aligned” view of memory:
 - In this type of picture, each row is composed of 8 bytes
 - Each cell is a byte
 - A 64-bit pointer will fit on one row



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

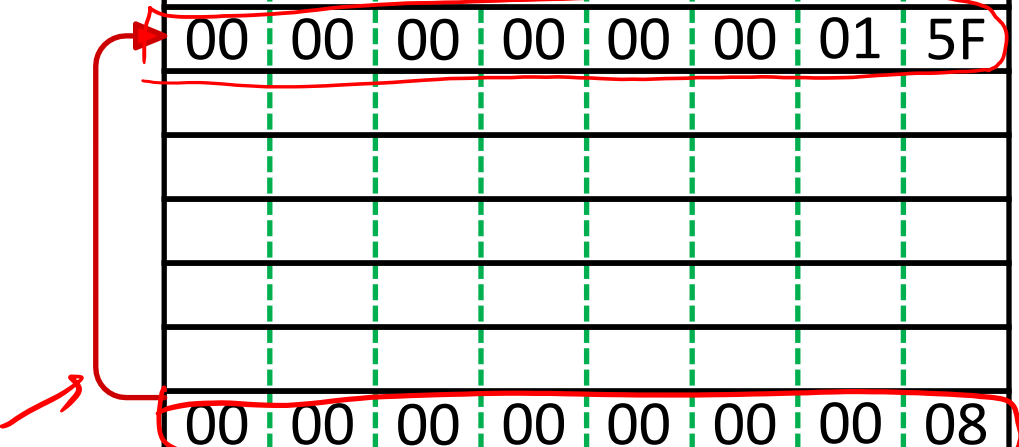


Addresses and Pointers

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

- ❖ An *address* is a location in memory
- ❖ A *pointer* is a data object that holds an address
 - Address can point to *any* data
- ❖ Value 351 stored at address 0x08
 - $351_{10} = 15F_{16}$
= 0x 00 00 01 5F
- ❖ Pointer stored at 0x38 points to address 0x08

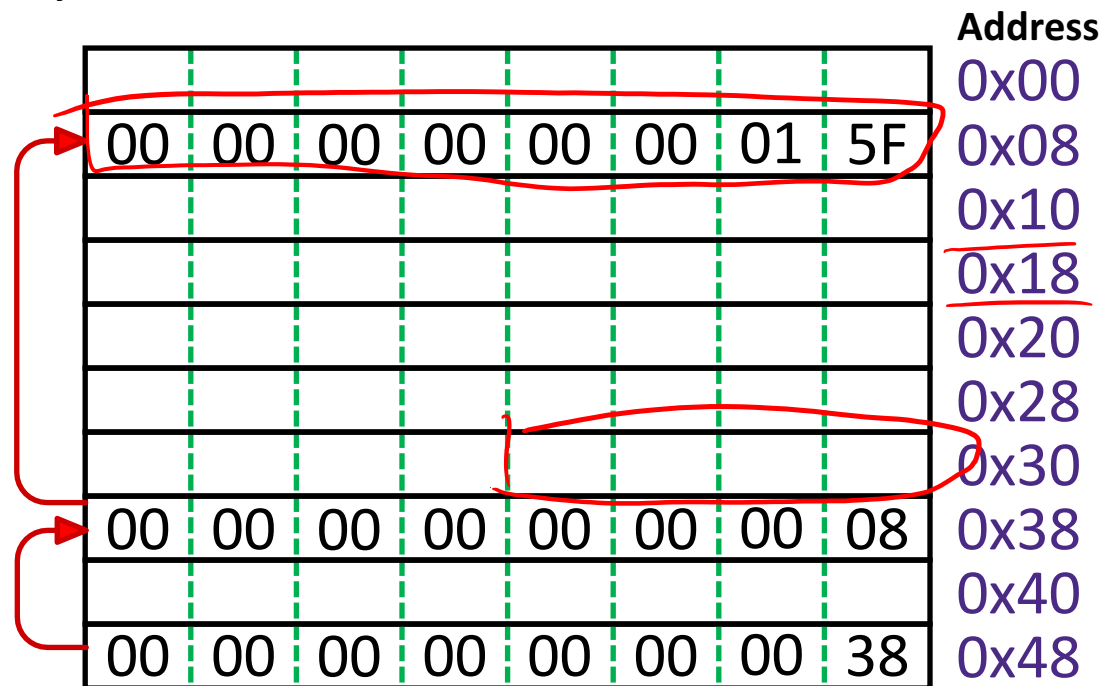
Address	
0x00	
<u>0x08</u>	00 00 00 00 00 00 01 5F
0x10	
0x18	
0x20	
0x28	
0x30	
<u>0x38</u>	00 00 00 00 00 00 00 08
0x40	
0x48	



Addresses and Pointers

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

- ❖ 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



Data Representations

❖ Sizes of data types (in bytes)

8086

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

address size = word size

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

More on Memory Alignment in x86-64

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

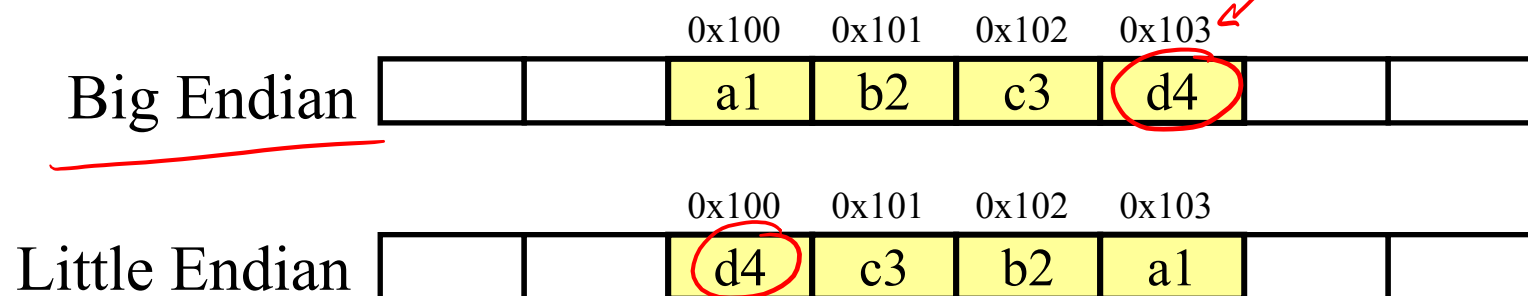
Byte Ordering

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

Byte Ordering

Handwritten notes:
 \$ 1,000,000,000,001
 MS (Most Significant)
 LS (Least Significant)

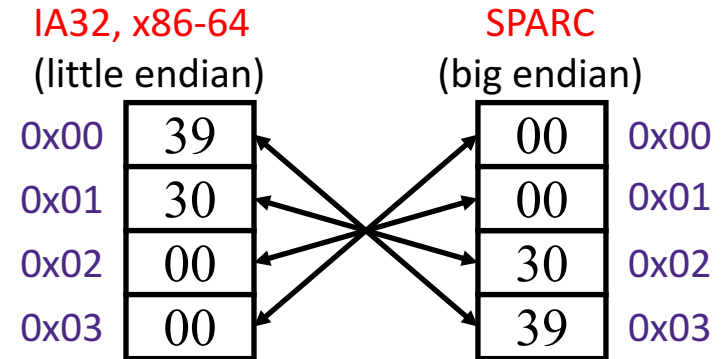
- ❖ Big-endian (SPARC, z/Architecture)
 - Least significant byte has highest address
- ❖ Little-endian (x86, x86-64)
 - Least significant byte has lowest address
- ❖ Bi-endian (ARM, PowerPC)
 - Endianness can be specified as big or little
- ❖ **Example:** 4-byte data 0xa1b2c3d4 at address 0x100



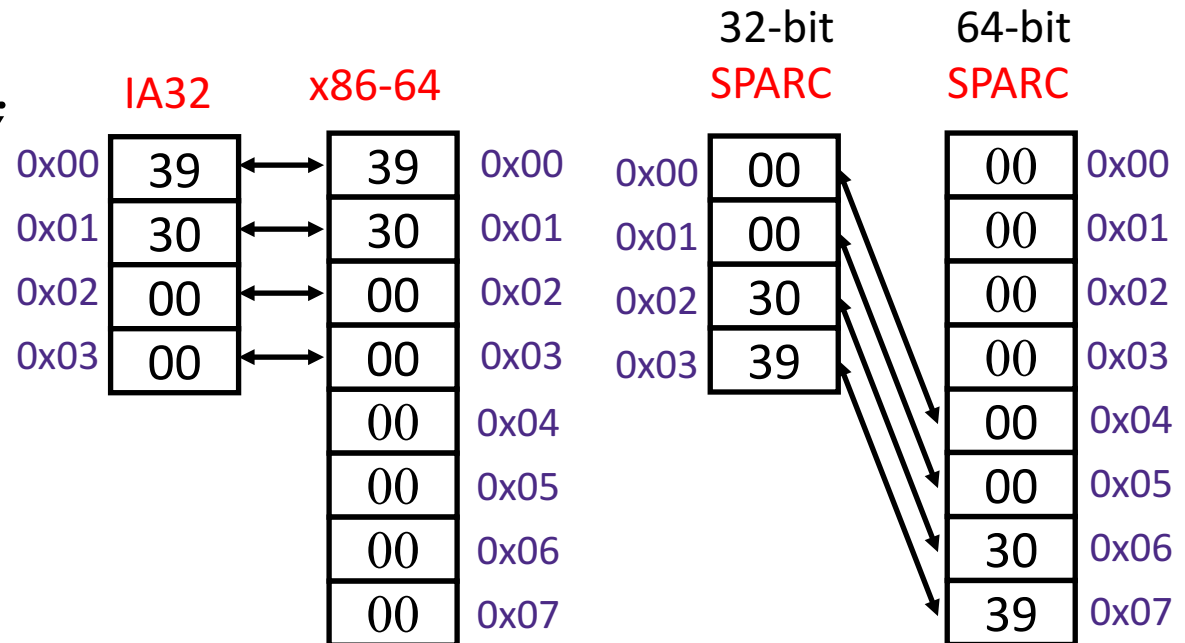
Byte Ordering Examples

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)

Endianness

- ❖ 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`)
 - When running down memory errors, need to know exact values
 - Manual translation to and from machine code (in 351)

Reading Byte-Reversed Listings

32-bit example

- ❖ Disassembly
 - Take binary machine code and generate an assembly code version
 - Does the reverse of the assembler
- ❖ Example instruction in memory
 - add value 0x12ab to register 'ebx' (*a special location in the CPU*)

Address	Instruction Code	Assembly Rendition
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx

Handwritten annotations: Red circles around 'ab' in the instruction code and '12ab' in the assembly rendition. Red slashes under '81 c3' and 'ab 12' in the instruction code. Red text 'MSB' and 'LSB' with arrows pointing to the '12' and 'ab' parts of the value respectively.

Deciphering numbers

Reading Byte-Reversed Listings

32-bit example

- ❖ Disassembly
 - Take binary machine code and generate an assembly code version
 - Does the reverse of the assembler
- ❖ Example instruction in memory
 - add value 0x12ab to register 'ebx' (*a special location in the CPU*)

Address	Instruction Code	Assembly Rendition
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx

Deciphering numbers

- Value:
- Pad to 32 bits:
- Split into bytes:
- Reverse (little-endian):

0x12ab
 0x000012ab
 00 00 12 ab
 ab 12 00 00

Question:

- ❖ We store the value $0x\ 00\ 01\ 02\ 03$ as a **word** at address $0x100$ and then get back $0x00$ when we read a **byte** at address $0x102$
- ❖ What machine setup are we using?

- (A) 32-bit, big-endian
- (B) 32-bit, little-endian
- (C) 64-bit, big-endian
- (D) 64-bit, little-endian

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
- ❖ IEC prefixes refer to powers of 2^{10}
- ❖ Pointers are data objects that holds addresses
- ❖ Endianness determines storage order for multi-byte objects