CSE351: Memory, Data, & Addressing I
CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma

Hunter Zahn

John Kaltenbach

Kevin Bi

Sachin Mehta

Suraj Bhat

Thomas Neuman

Waylon Huang

Xi Liu

Yufang Sun

http://xkcd.com/138/

Administrivia

❖ Start-of-Course survey due today at 5pm

▪ Can find staff's mini-bios by clicking our faces on website

❖ Lab 0 due Monday at 5pm

❖ All course materials can be found on the website [schedule](#)

❖ Piazza

▪ Please read `using_piazza.pdf` on Piazza (@6)

▪ Can be risky to rely on e-mails

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
```

OS:

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```
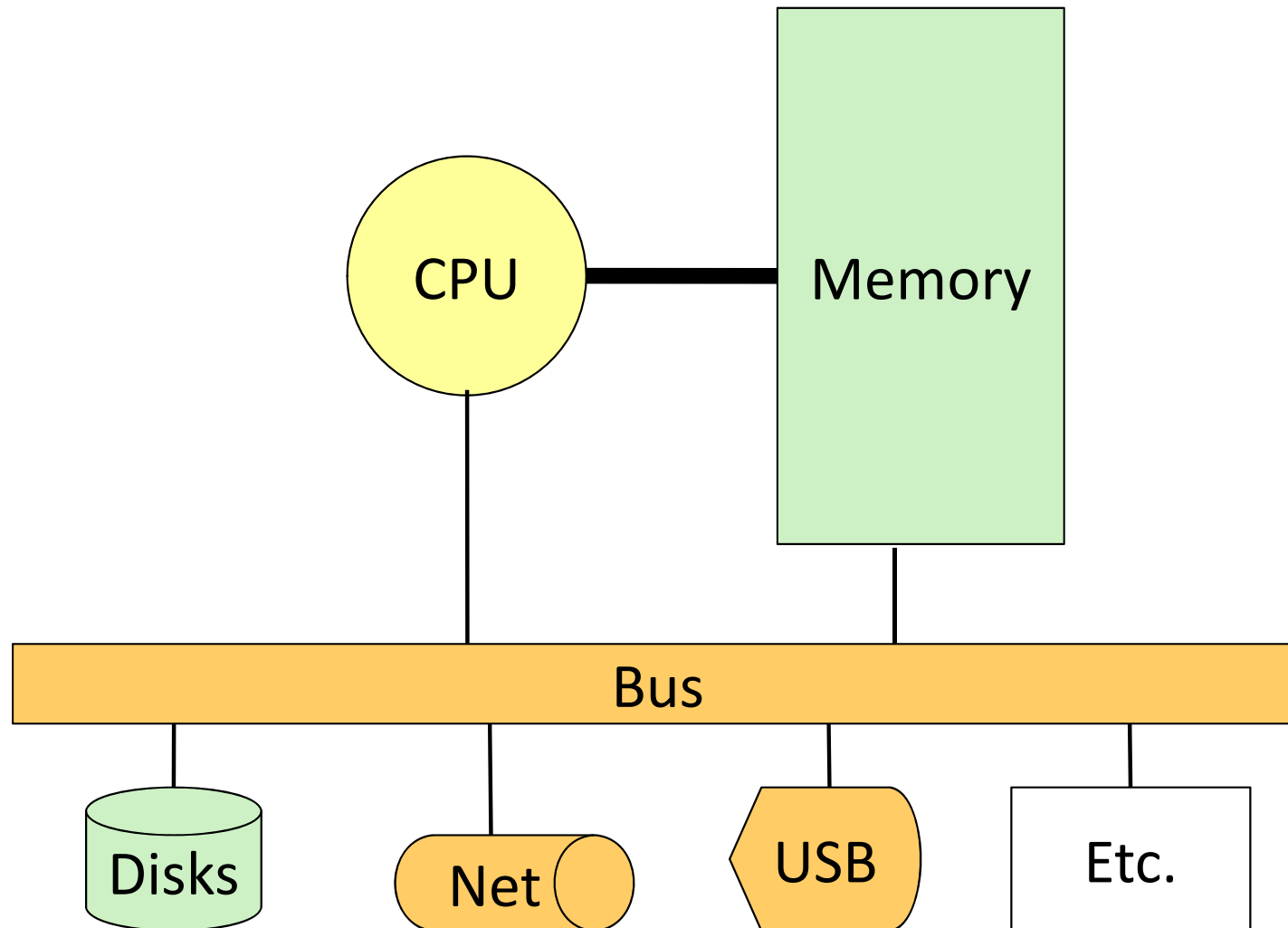
Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
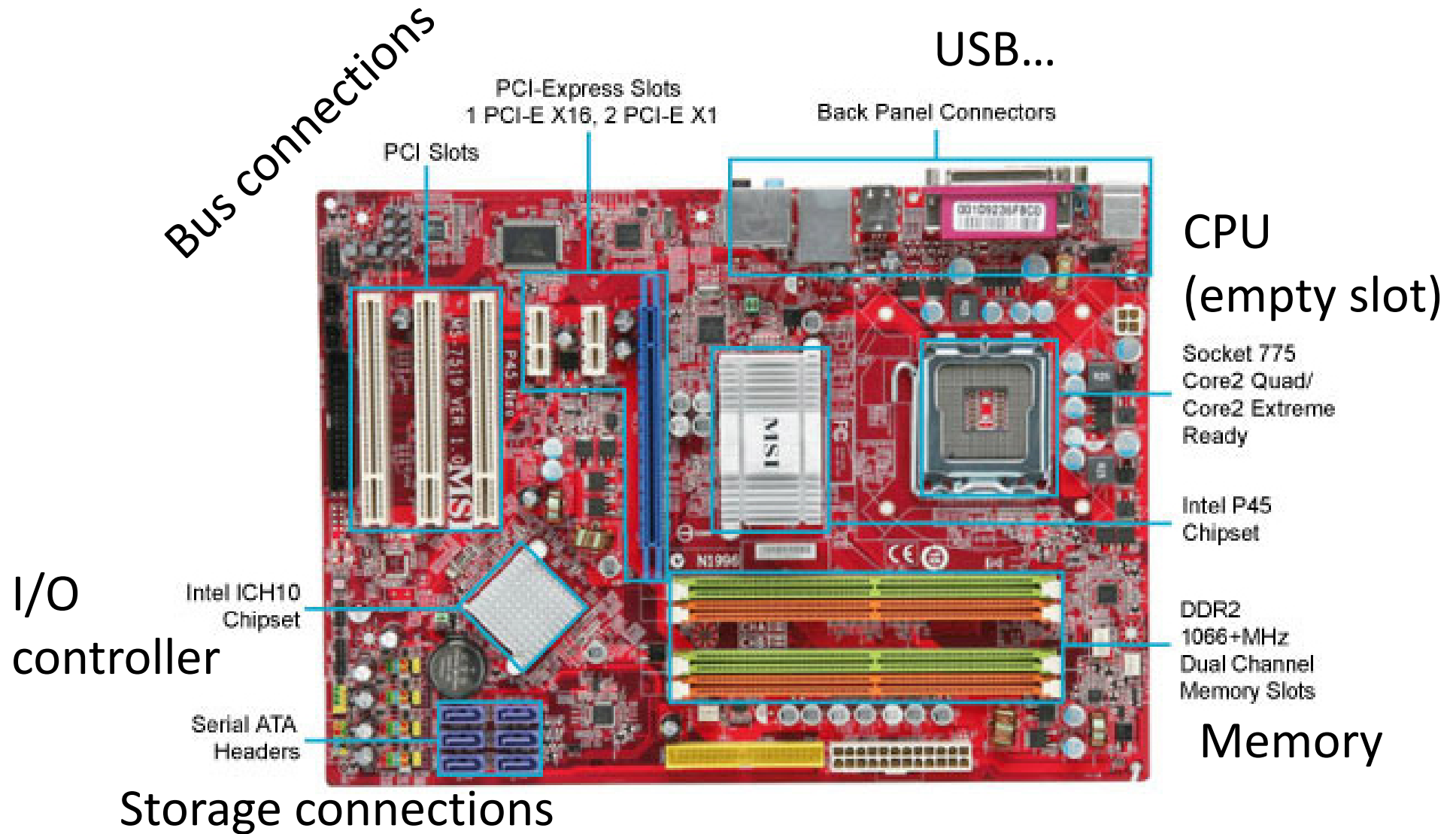Processes
Virtual memory
Memory allocation
Java vs. C

3

Hardware: Logical View

Hardware: Physical View

USB…

Bus connections

PCI-Express Slots
1 PCI-E X16, 2 PCI-E X1

PCI Slots

Back Panel Connectors

CPU
(empty slot)

Socket 775
Core2 Quad/
Core2 Extreme
Ready

Intel P45
Chipset

Intel ICH10
Chipset

I/O
controller

DDR2
1066+MHz
Dual Channel
Memory Slots

Memory

Serial ATA
Headers

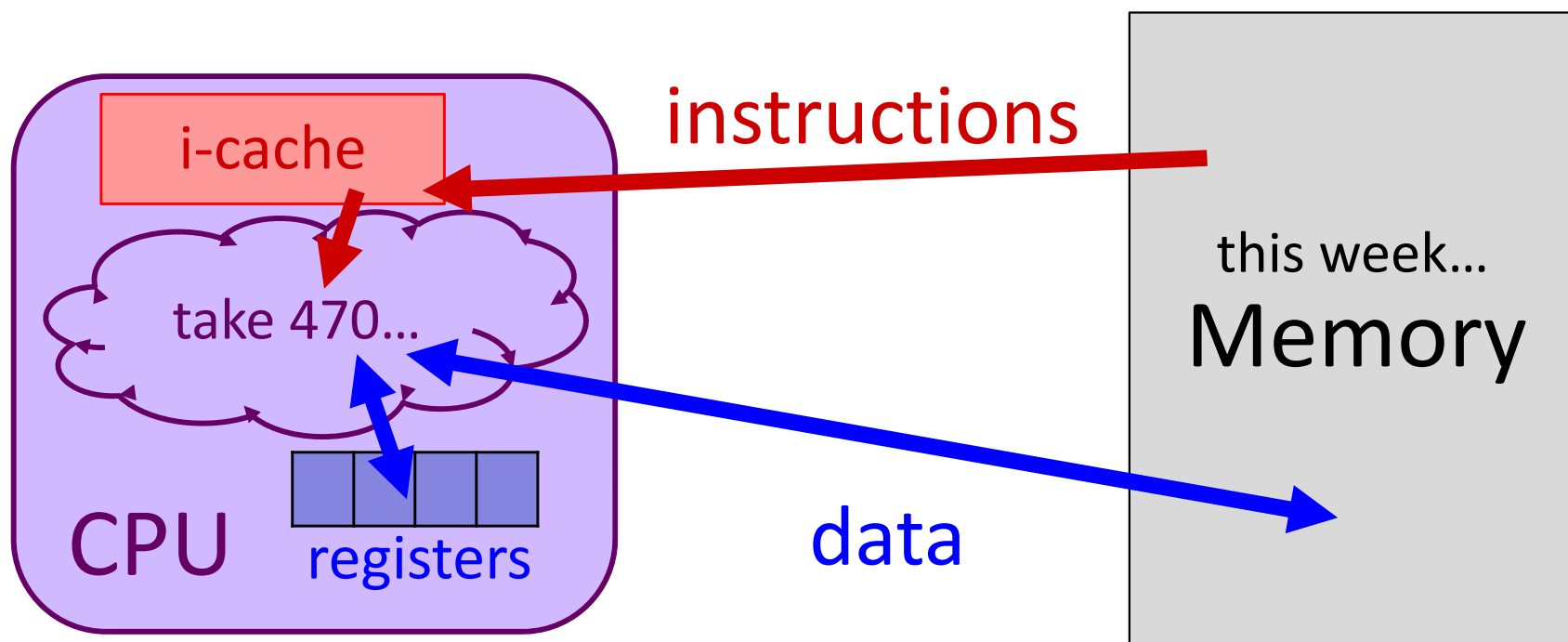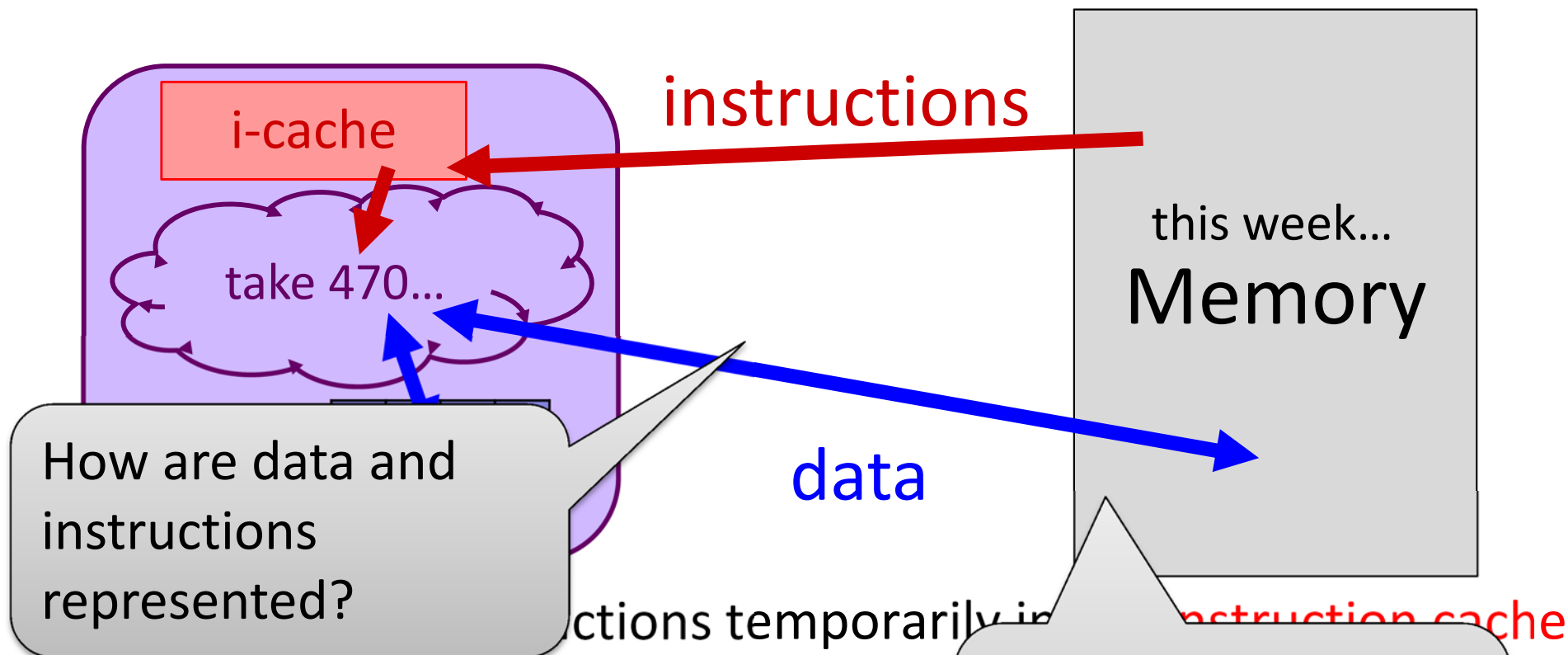Storage connections

Hardware: 351 View (version 0)
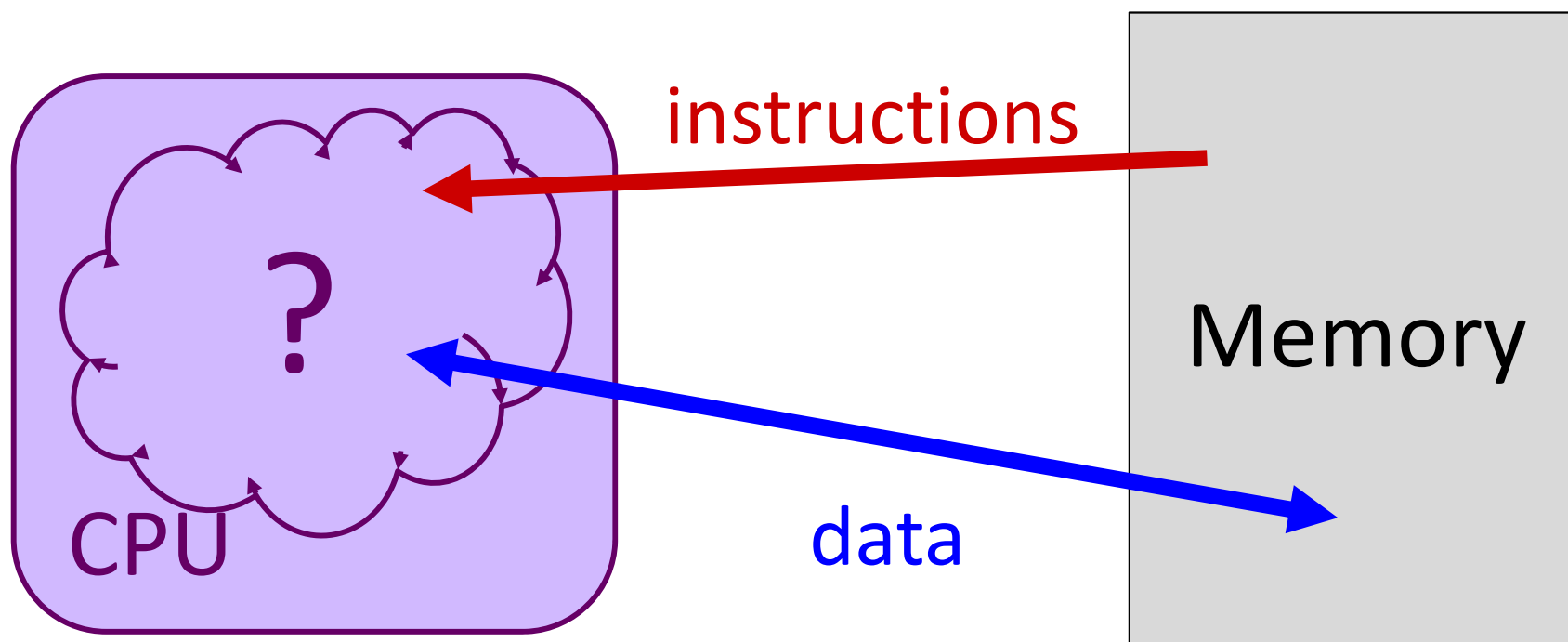
❖ CPU executes instructions; memory stores data

❖ 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)

instructions

i-cache

take 470...

this week...
Memory

data

How are data and instructions represented?

...ctions temporarily in ...nstruction cache

How does a program find its data in memory?

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

Memory, Data, and Addressing

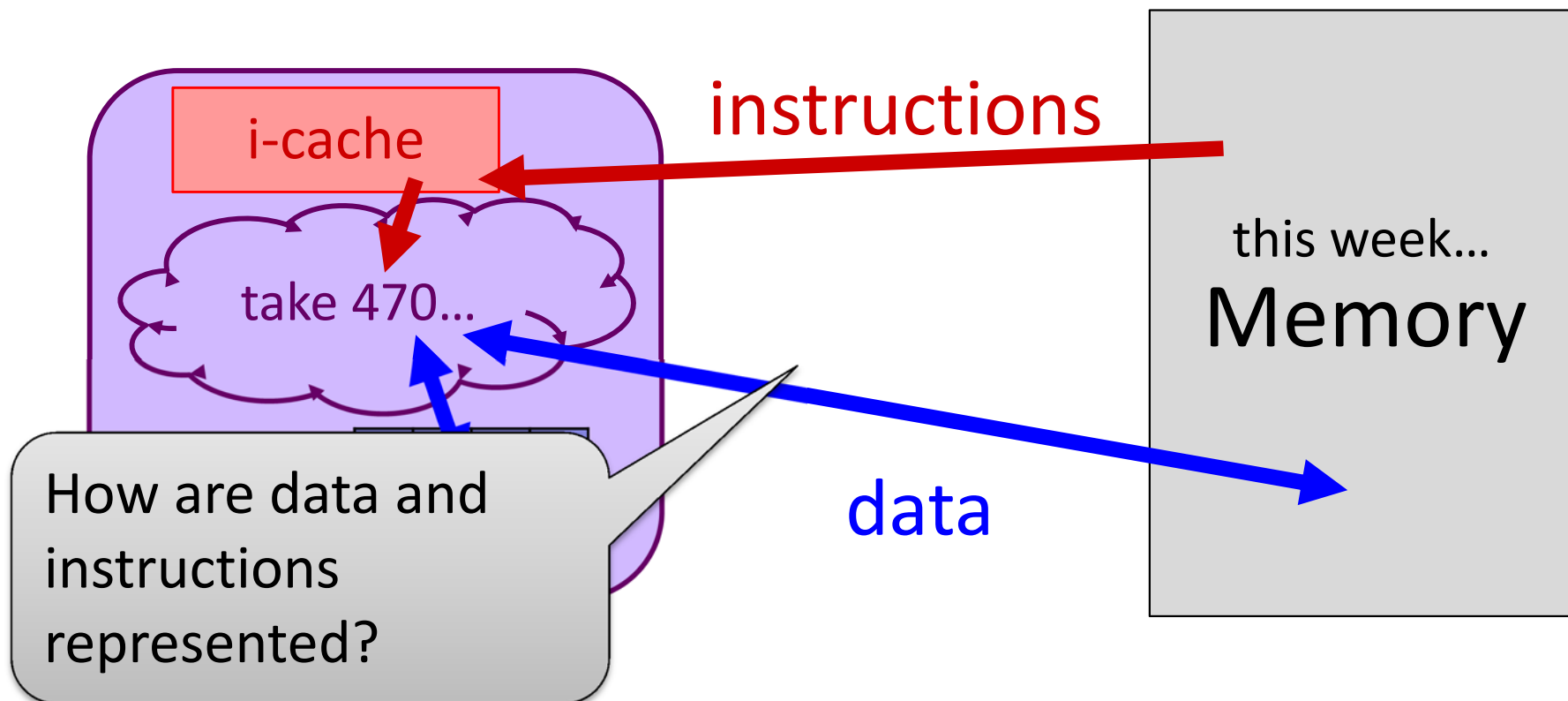- ❖ Representing information as bits and bytes
- ❖ Organizing and addressing data in memory
- ❖ Manipulating data in memory using C

Question 1:

instructions

i-cache

take 470…

this week…
Memory

How are data and instructions represented?

data

Binary Representations

❖ Base 2 number representation
- A base 2 digit (0 or 1) is called a *bit*
- Represent 351_{10} as 000000101011111_2 or 101011111_2

  Leading zeros

❖ Electronic implementation
- Easy to store with bi-stable elements
- Reliably transmitted on noisy and inaccurate wires
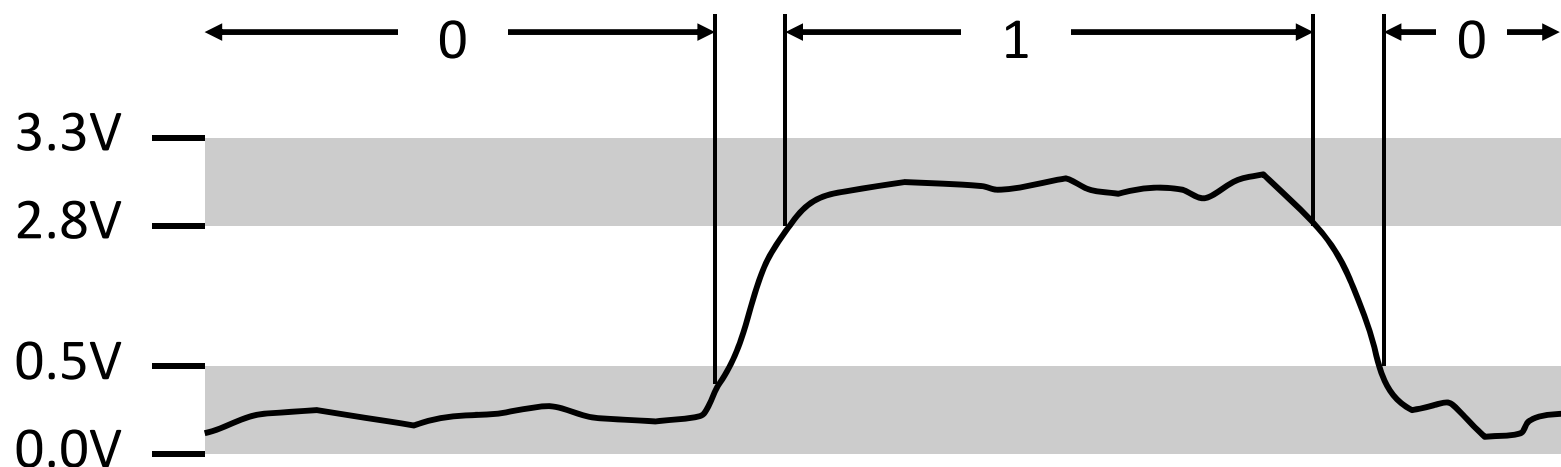
Review: Number Bases

❖ **Key terminology:** digit (*d*) and base (*B*)

- In base B, each digit is one of B possible <u>symbols</u>

❖ Value of i-th digit is $d \times B^i$ where i starts at 0 and increases from right to left

- *n* digit number $d_{n-1} d_{n-2} \ldots d_1 d_0$

- value = $d_{n-1} \times B^{n-1} + d_{n-2} \times B^{n-2} + \ldots + d_1 \times B^1 + d_0 \times B^0$

- 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

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
F	15	1111

❖ Binary (00000000_2 – 11111111_2)

  ▪ Byte = 8 bits (binary digits)

MSB   LSB

0	0	1	0	1	1	0	1
$0*2^7$	$0*2^6$	$1*2^5$	$0*2^4$	$1*2^3$	$1*2^2$	$0*2^1$	$1*2^0$
		32		8	4		1

$= 45_{10}$
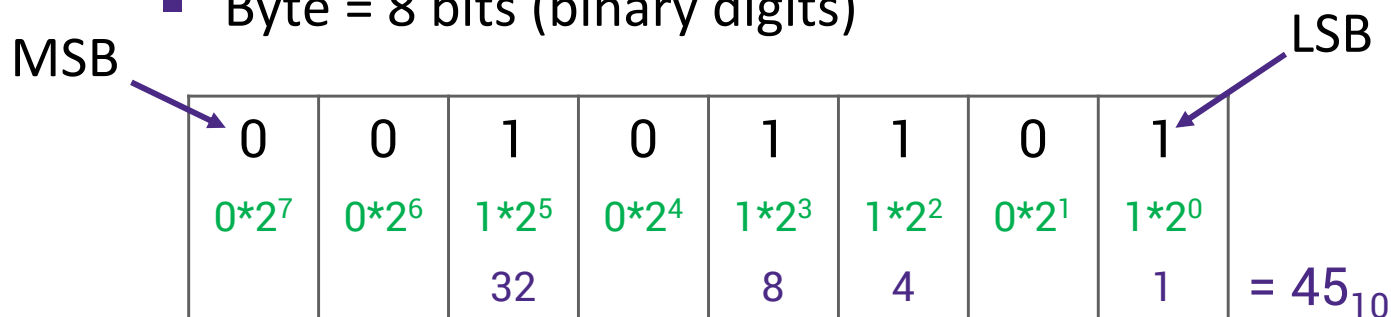
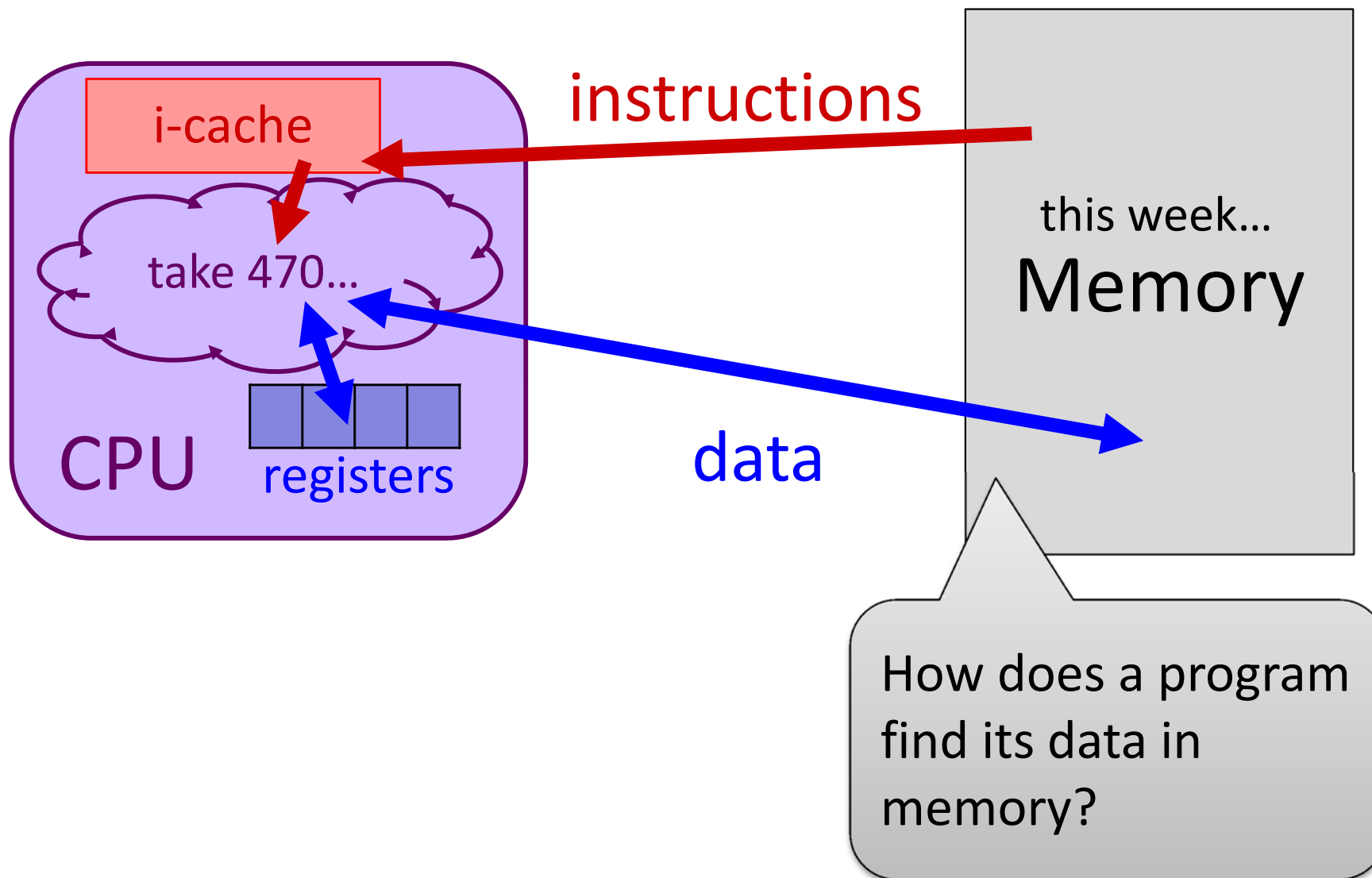❖ Decimal (0_{10} – 255_{10})

❖ 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…

Question 2:

Byte-Oriented Memory Organization
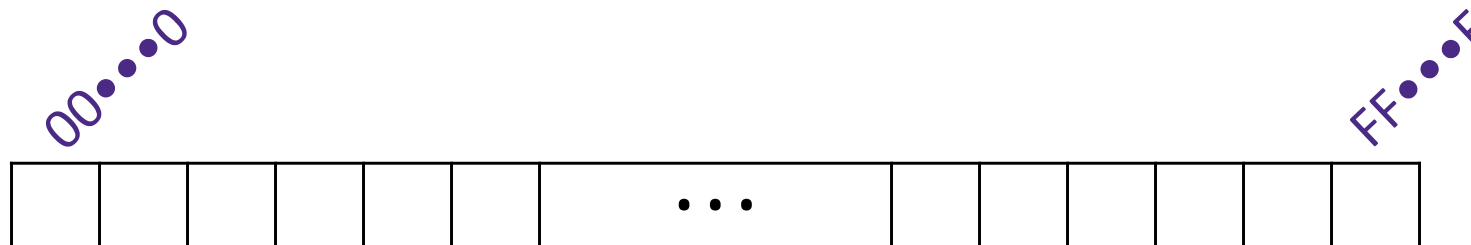
❖ Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)

❖ 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 → 2^w addresses

❖ Current x86 systems use **64-bit (8-byte) words**

  ▪ Potential address space: 2^{64} addresses
    2^{64} bytes ≈ **1.8 x 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
10^3	Kilo-	K	2^{10}	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?

64-bit Words	32-bit Words	Bytes	Addr. (hex)
Addr = ??	Addr = ??		0x00
			0x01
			0x02
			0x03
	Addr = ??		0x04
			0x05
			0x06
			0x07
Addr = ??	Addr = ??		0x08
			0x09
			0x0A
			0x0B
	Addr = ??		0x0C
			0x0D
			0x0E
			0x0F

18
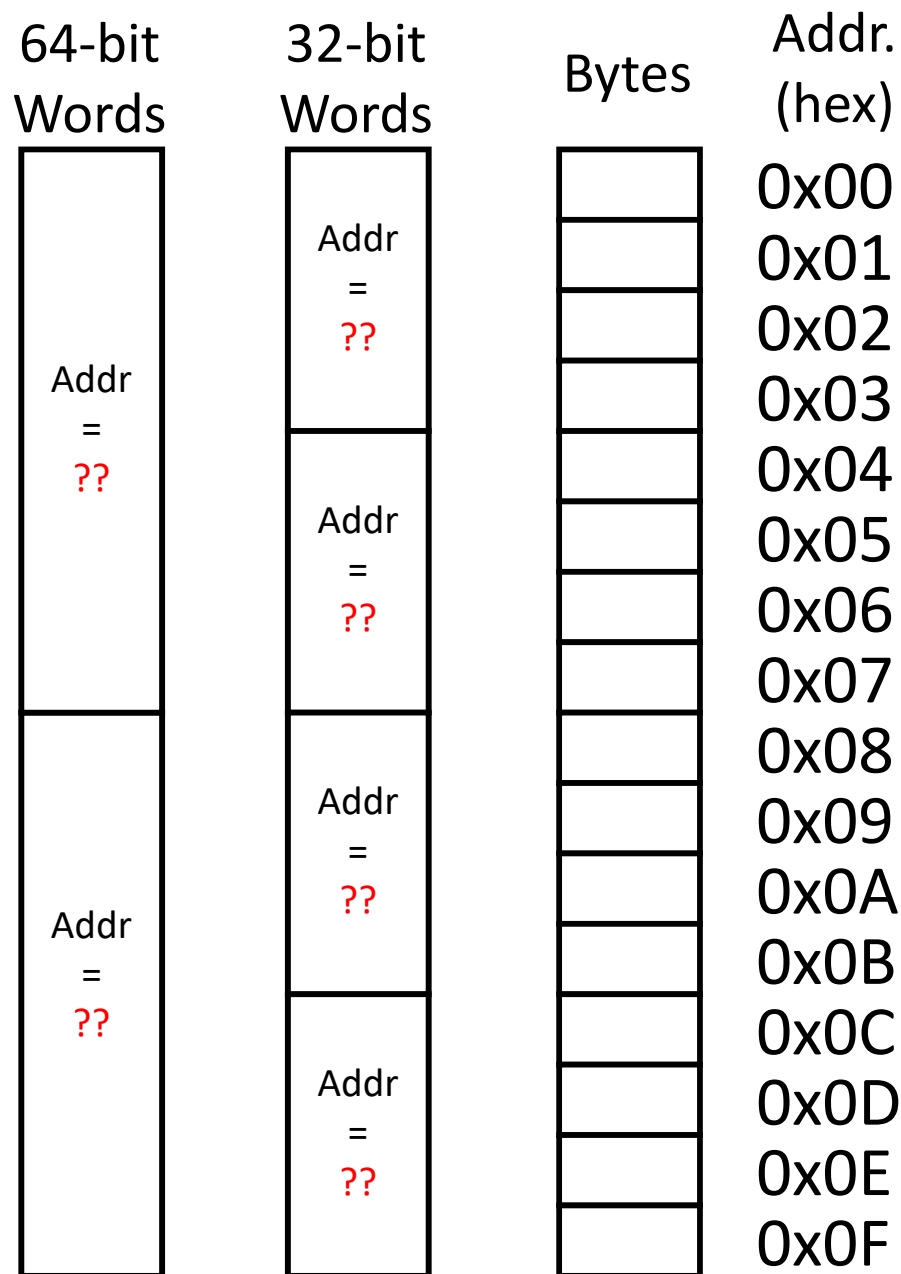
Word-Oriented Memory Organization

❖ 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***
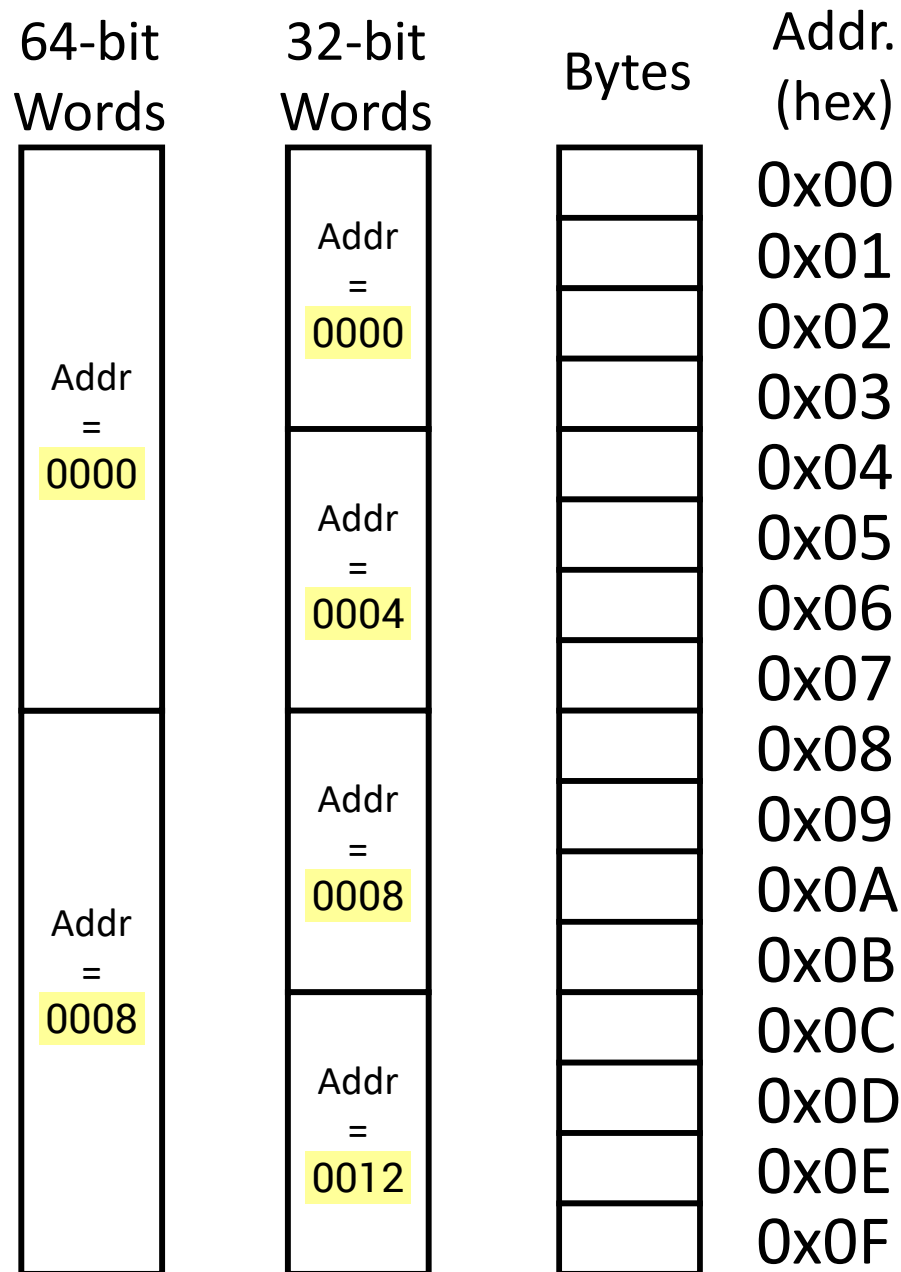
64-bit Words	32-bit Words	Bytes	Addr. (hex)
	Addr = 0000		0x00
			0x01
Addr = 0000			0x02
			0x03
	Addr = 0004		0x04
			0x05
			0x06
			0x07
	Addr = 0008		0x08
			0x09
Addr = 0008			0x0A
			0x0B
	Addr = 0012		0x0C
			0x0D
			0x0E
			0x0F

19

A Picture of Memory (64-bit view)

❖ A "64-bit (8-byte) word-aligned" <u>view</u> 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

one word

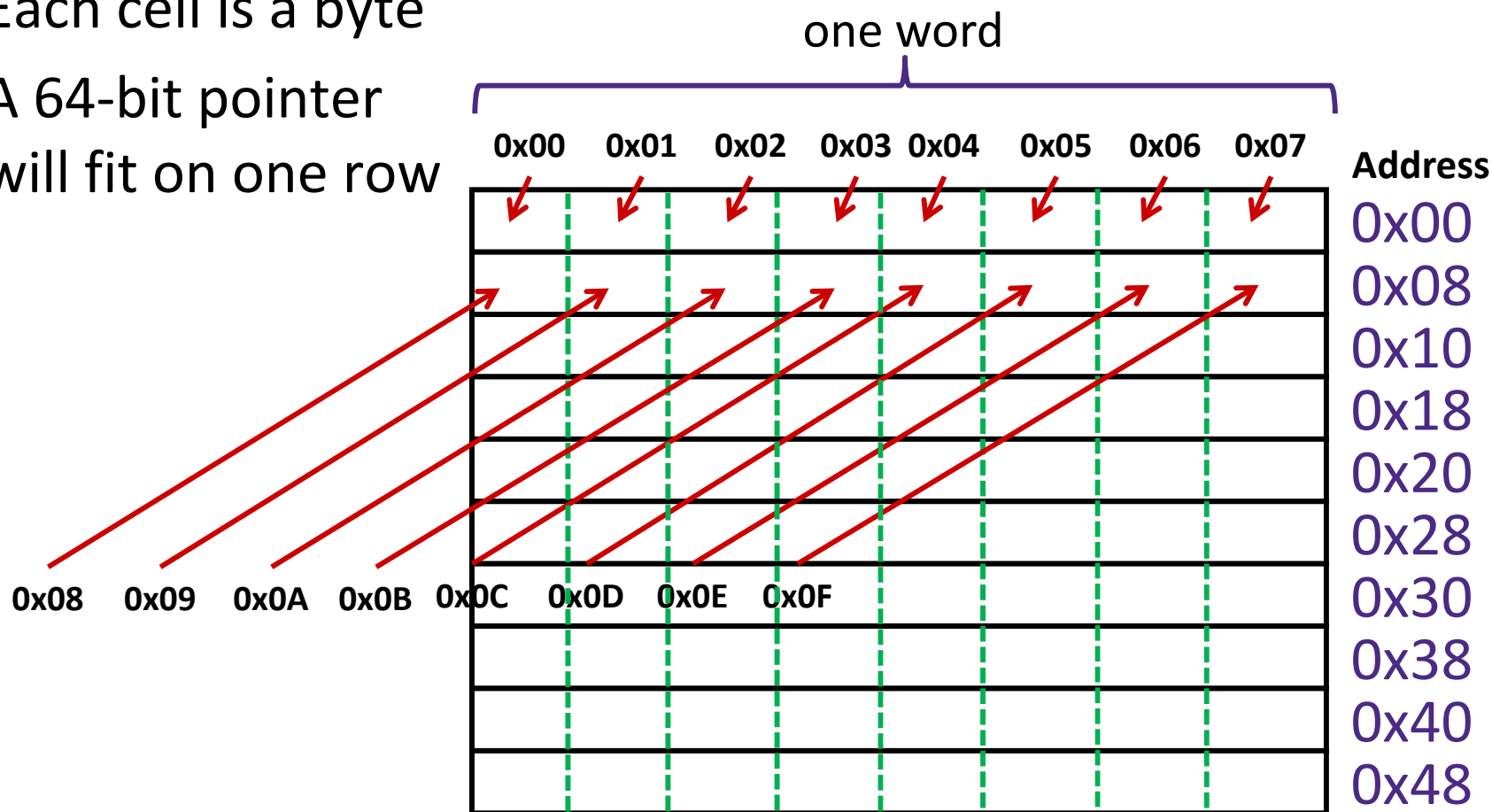| 0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	Address
								0x00
								0x
								0x
								0x
								0x
								0x
								0x
								0x
								0x
								0x

A Picture of Memory (64-bit view)

❖ A "64-bit (8-byte) word-aligned" <u>view</u> 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

one word

0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07

Address
0x00
0x08
0x10
0x18
0x20
0x28
0x30
0x38
0x40
0x48

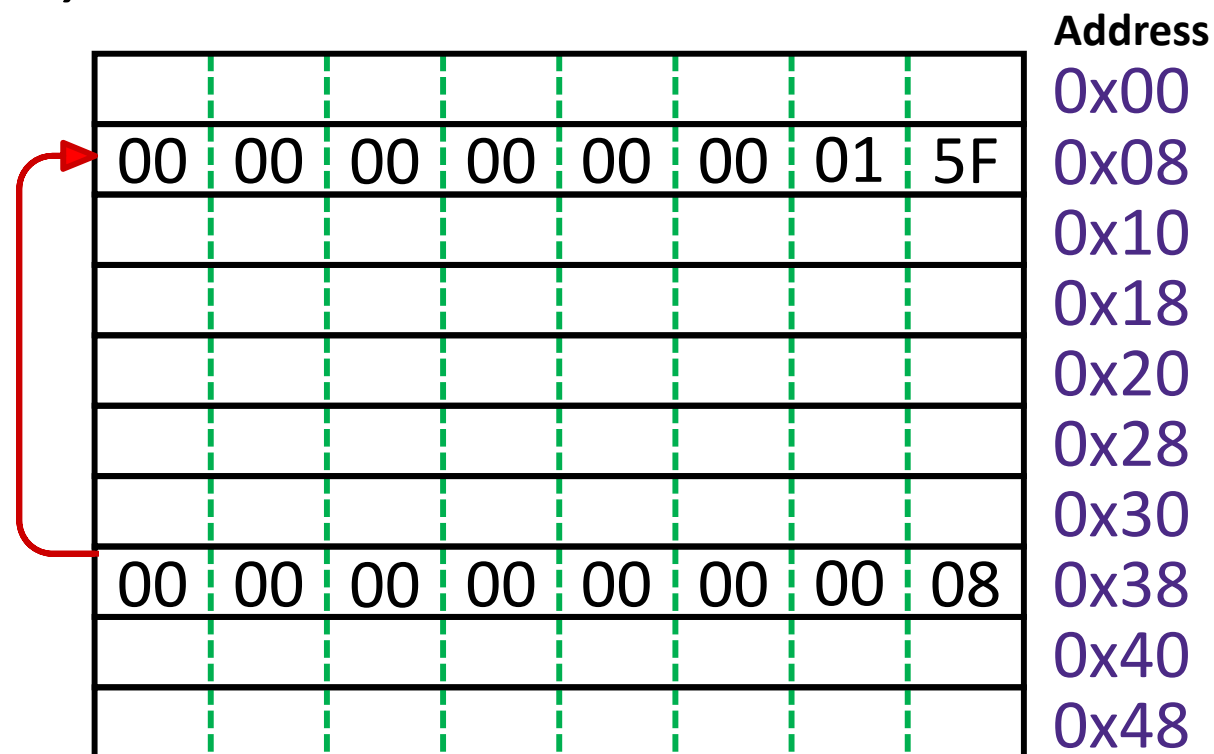0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
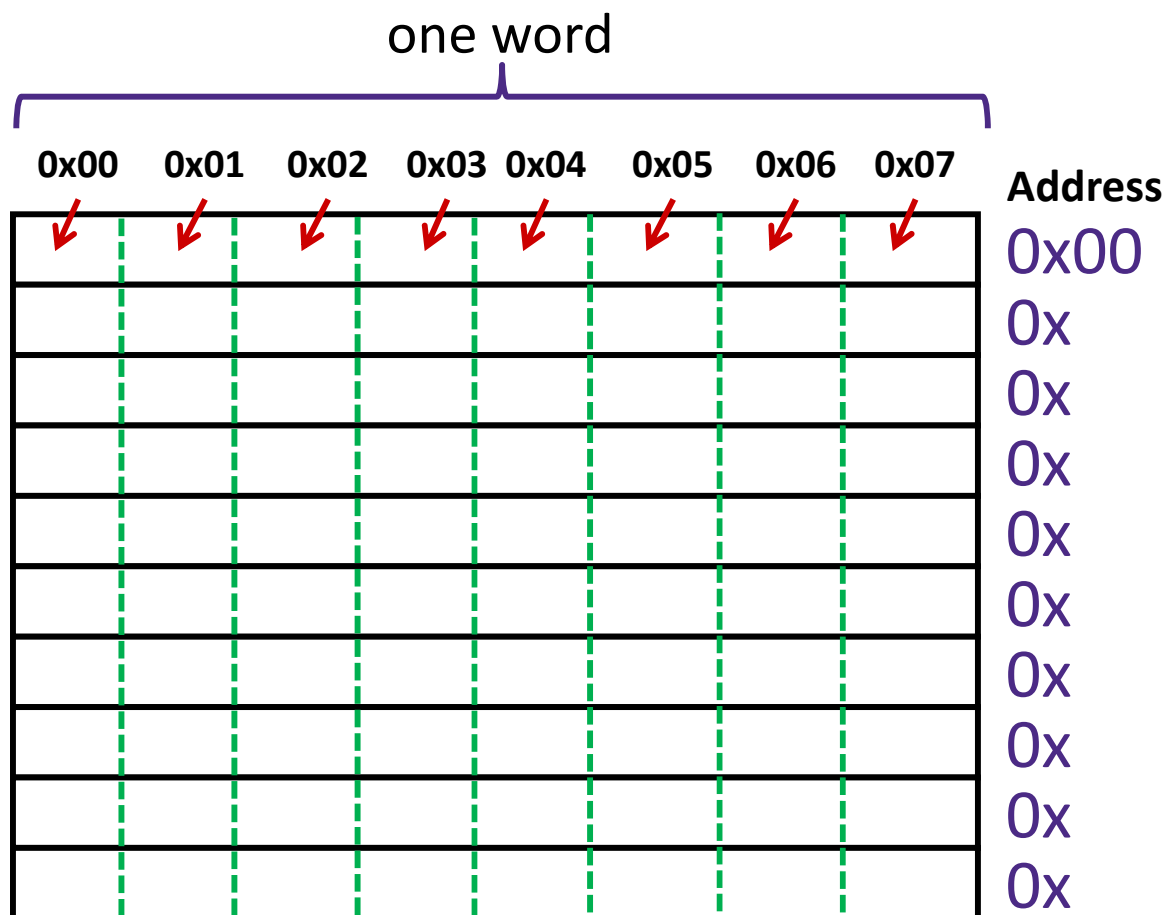
Addresses and Pointers

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
00	00	00	00	00	00	01	5F	0x08
								0x10
								0x18
								0x20
								0x28
								0x30
00	00	00	00	00	00	00	08	0x38
								0x40
								0x48

22

Addresses and Pointers

- ❖ 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
00	00	00	00	00	00	01	5F	0x08
								0x10
								0x18
								0x20
								0x28
								0x30
00	00	00	00	00	00	00	08	0x38
								0x40
00	00	00	00	00	00	00	38	0x48

23
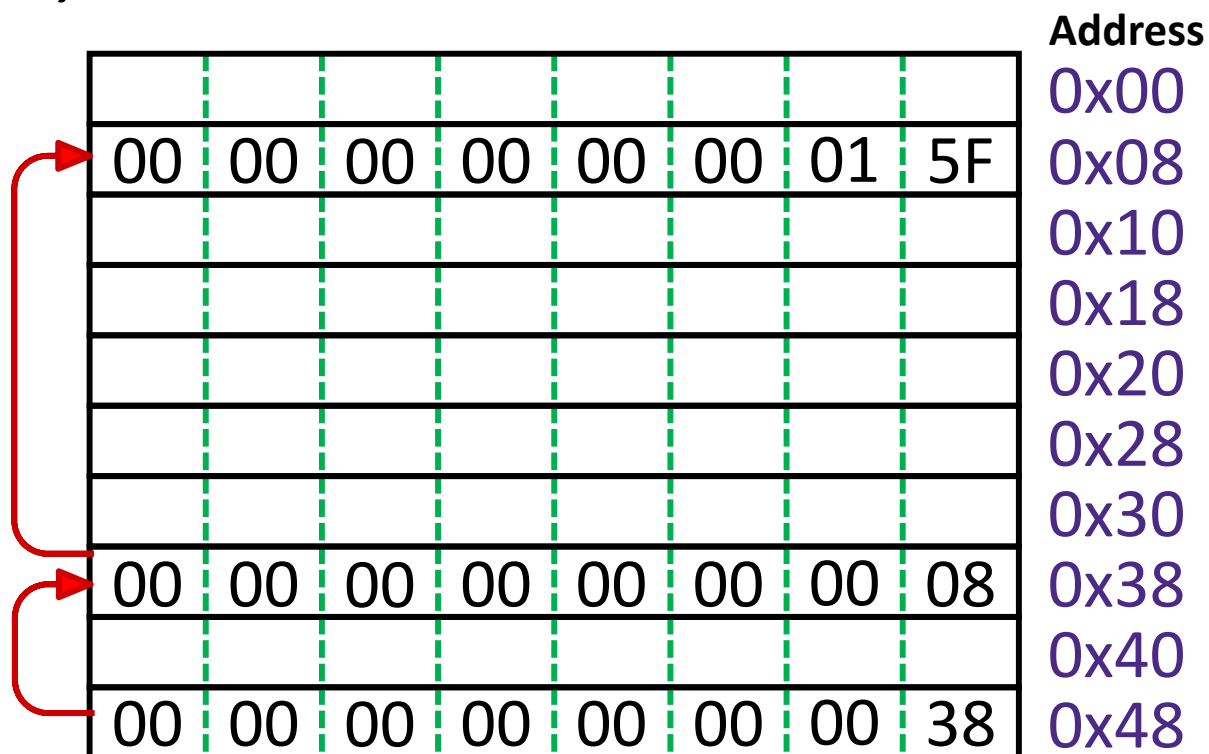
Data Representations

❖ 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>

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
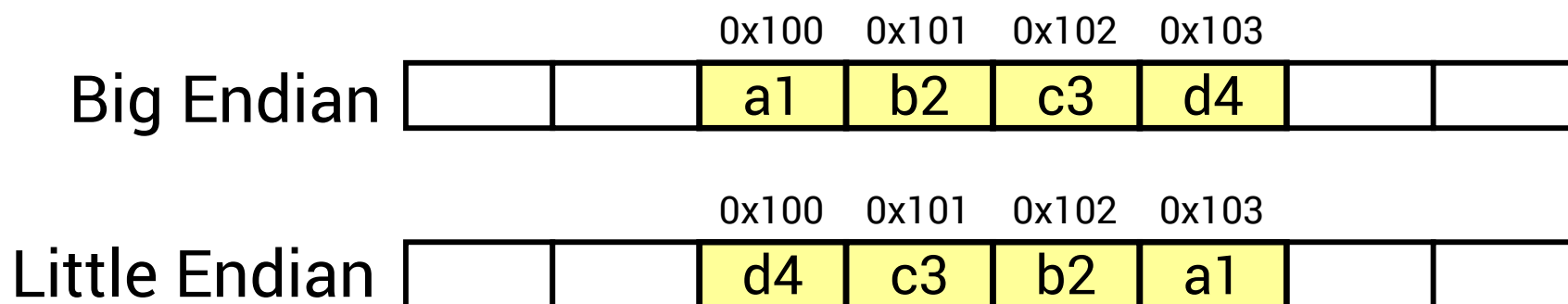 - More about alignment later in the course

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)
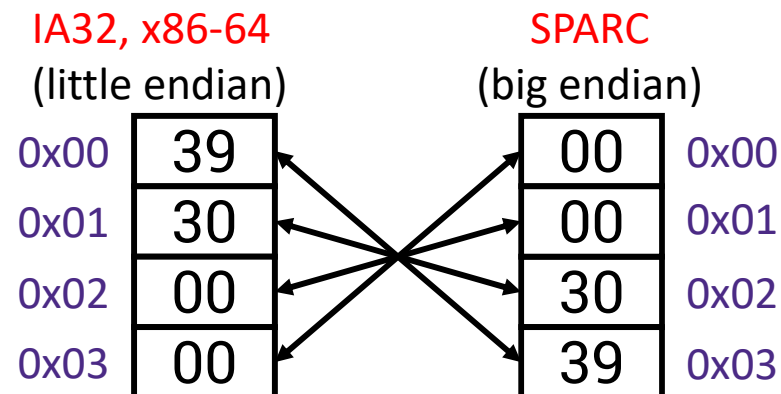
Byte Ordering

- ❖ 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

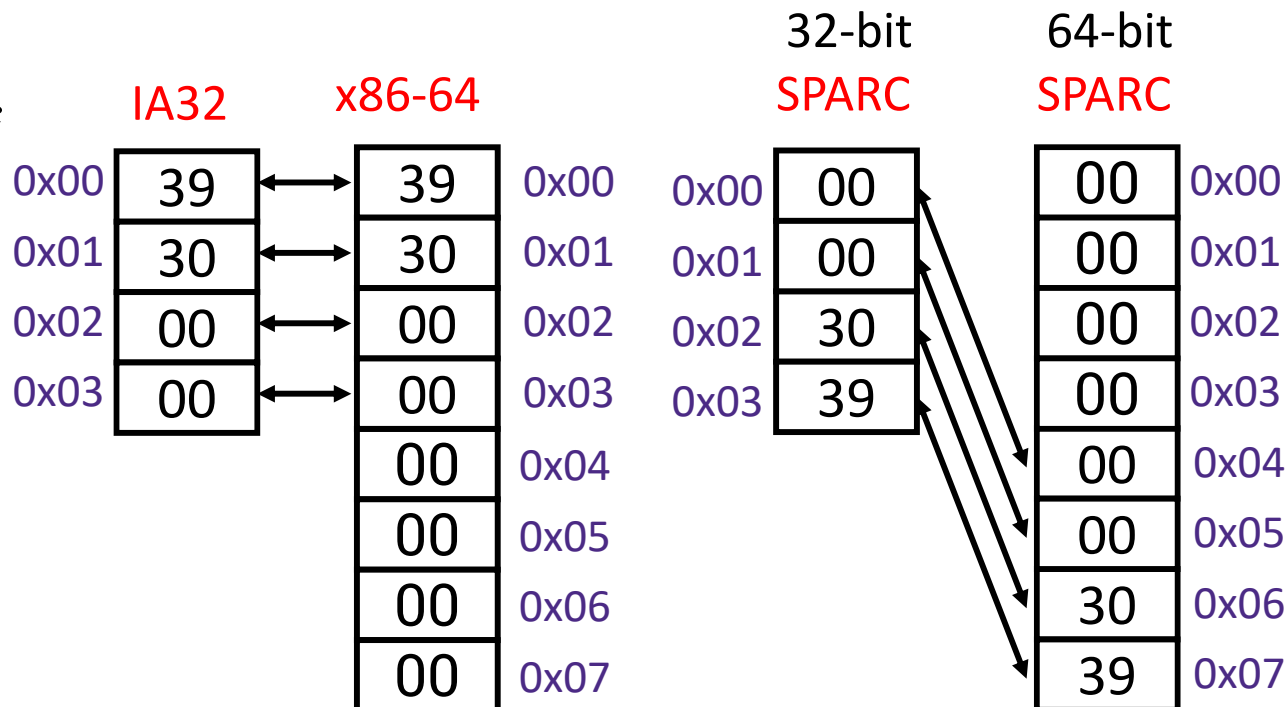|		0x100	0x101	0x102	0x103		
Big Endian		a1	b2	c3	d4		

		0x100	0x101	0x102	0x103		
Little Endian		d4	c3	b2	a1		

27

Byte Ordering Examples

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

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

IA32, x86-64
(little endian)

SPARC
(big endian)

0x00	39			00	0x00
0x01	30			00	0x01
0x02	00			30	0x02
0x03	00			39	0x03

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

(A long int is
the size of a word)

IA32   x86-64

32-bit
SPARC

64-bit
SPARC

0x00	39	39	0x00	0x00	00	00	0x00	
0x01	30	30	0x01	0x01	00	00	0x01	
0x02	00	00	0x02	0x02	30	00	0x02	
0x03	00	00	0x03	0x03	39	00	0x03	
		00	0x04			00	0x04	
		00	0x05			00	0x05	
		00	0x06			30	0x06	
		00	0x07			39	0x07	

UNIVERSITY *of* WASHINGTON

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

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: 0x12ab
  ▪ Pad to 32 bits: 0x000012ab
  ▪ Split into bytes: 00 00 12 ab
  ▪ Reverse (little-endian):   ab 12 00 00
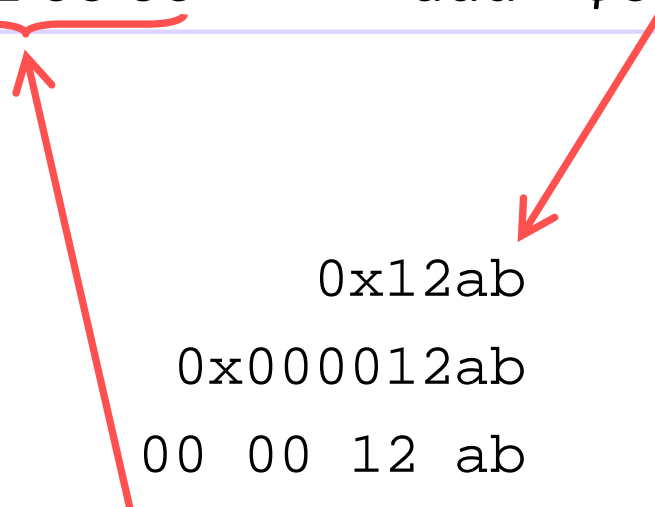
Peer Instruction Question:

❖ 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?

   ▪ Vote at http://PollEv.com/justinh
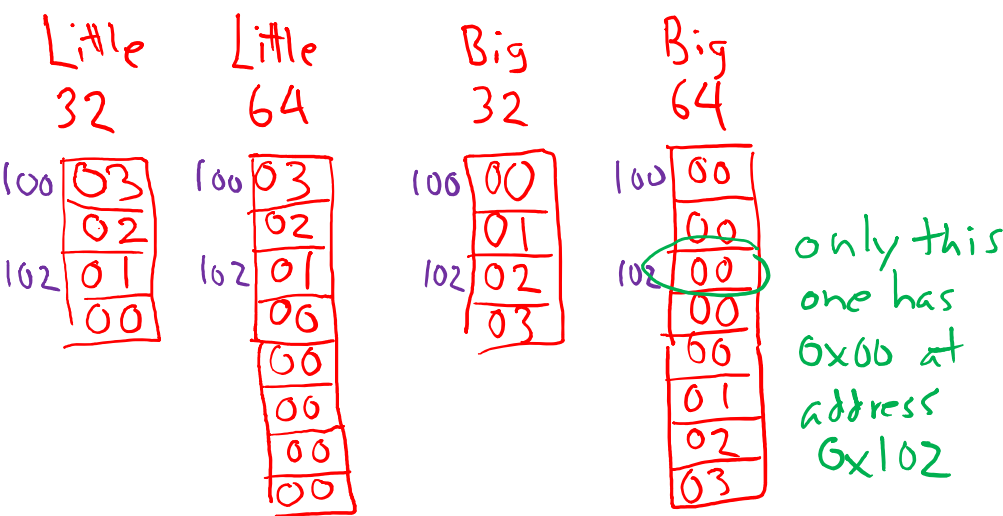
(A) **32-bit, big-endian**
(B) **32-bit, little-endian**
(C) **64-bit, big-endian**
(D) **64-bit, little-endian**

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