System Programming

03. B. Memory (ch 2.1)

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

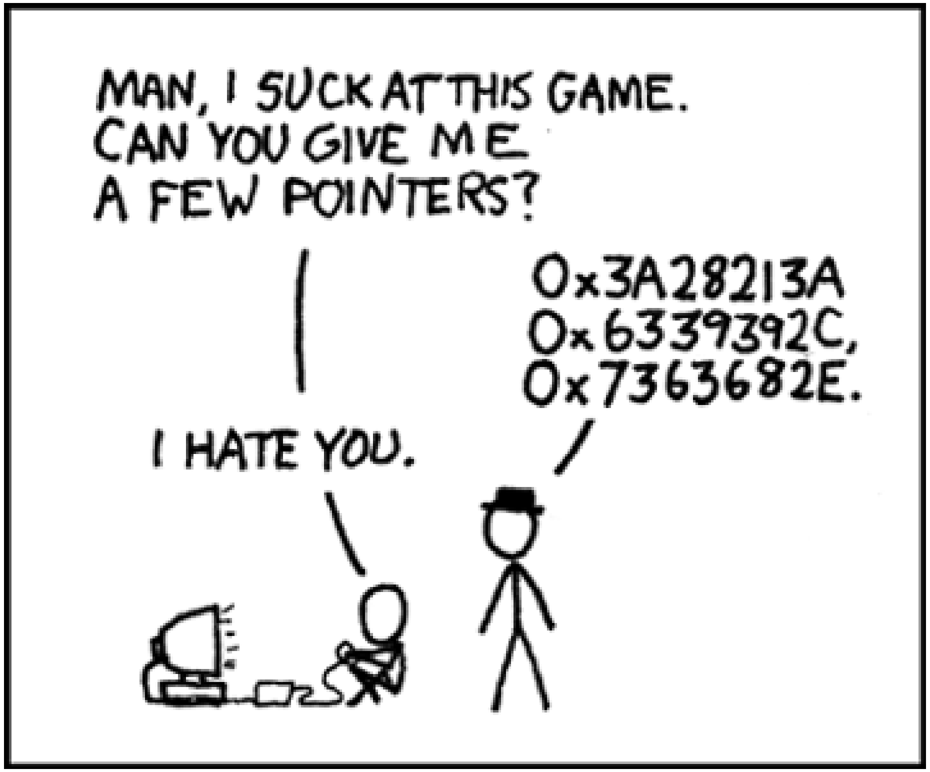
Data Science Lab @ PNU

These slides are based on the slides prepared by Dan Grossman, Ruth Anderson.

datalab
data science laboratory

Pointers

- http://xkcd.com/138/



Roadmap

C:

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

Java:

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

Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

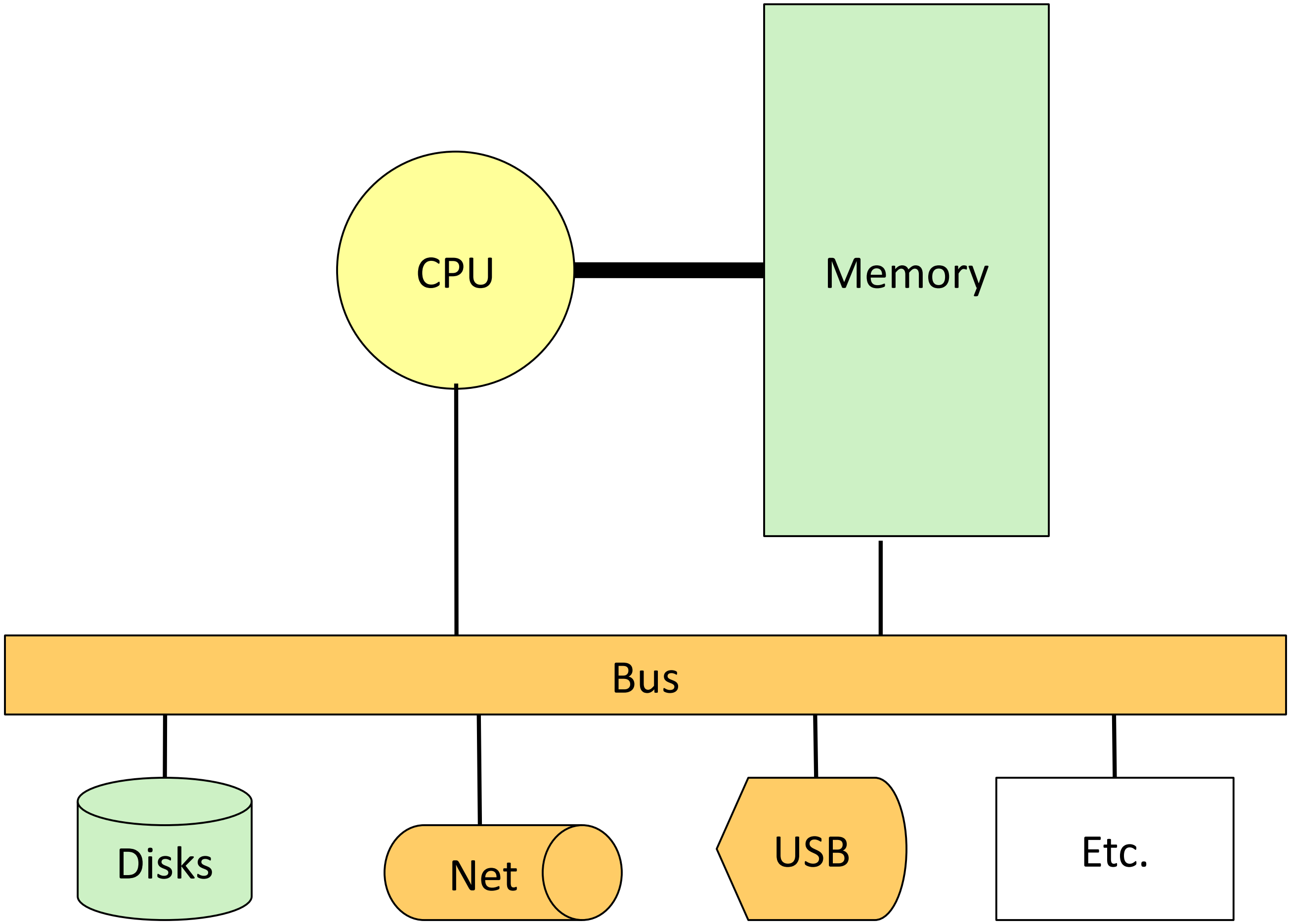
OS:

Windows 8 · Mac

Computer system:

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

Hardware: Logical View

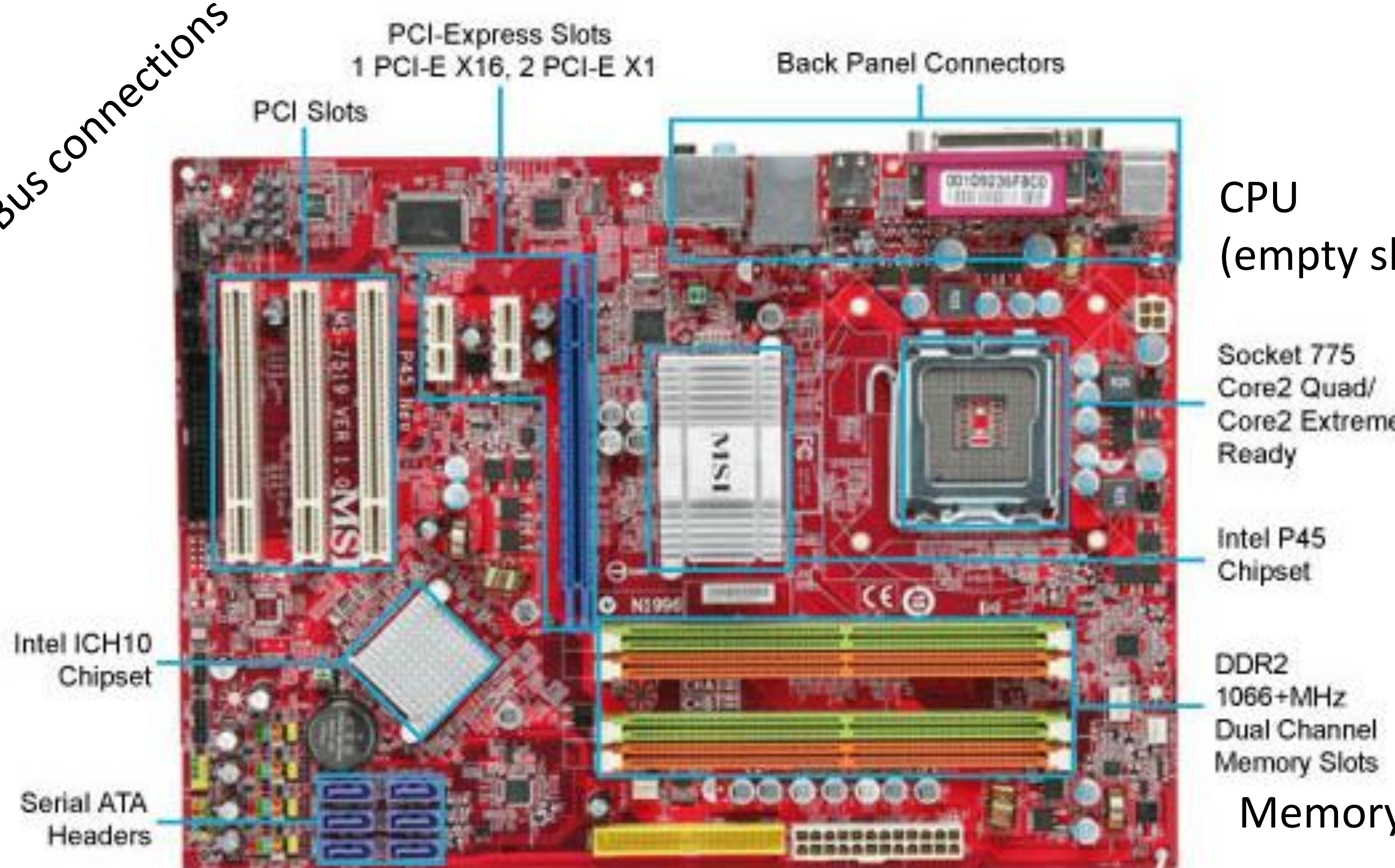


Hardware: Physical View

USB…

Bus connections

CPU
(empty slot)

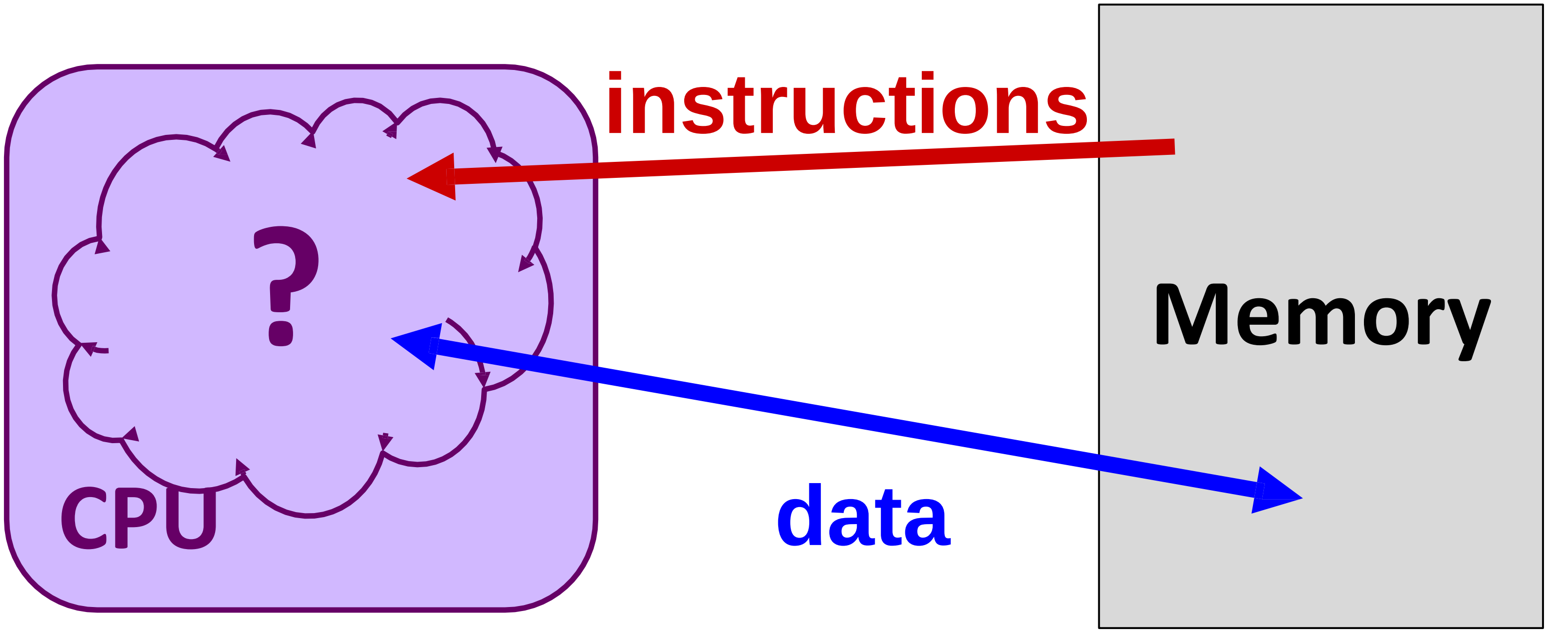


I/O
controller

Memory

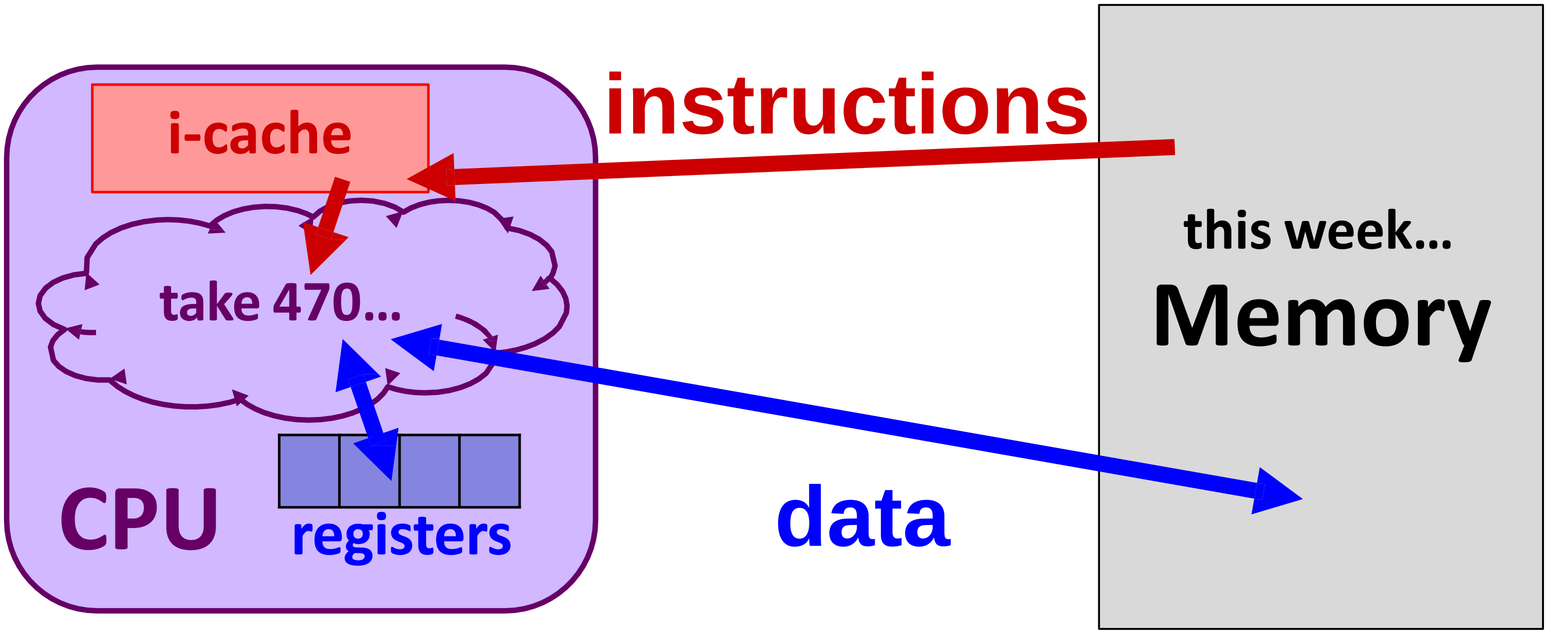
Storage connections

Hardware: View (version 0)



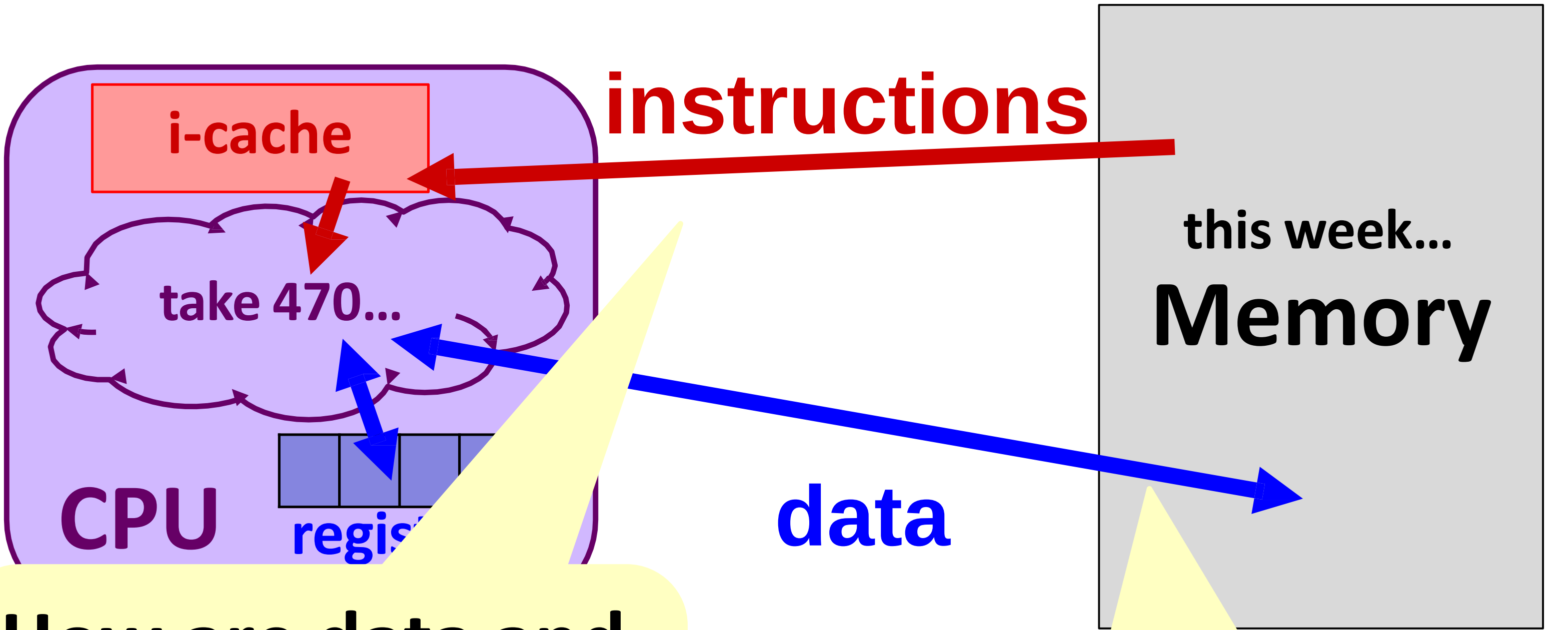
- CPU executes instructions; memory stores data
- To execute an instruction, the CPU must:
  - fetch an instruction;
  - fetch the data used by the instruction; and, finally,
  - execute the instruction on the data…
  - which may result in writing data back to memory.

Hardware: 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 HW-controlled
- **Data movement** is (assembly language) programmer-controlled
- We'll learn about the instructions the CPU executes
  - take computer architecture course to find out how it actually executes them

Hardware: View (version 1)



How are data and instructions represented?

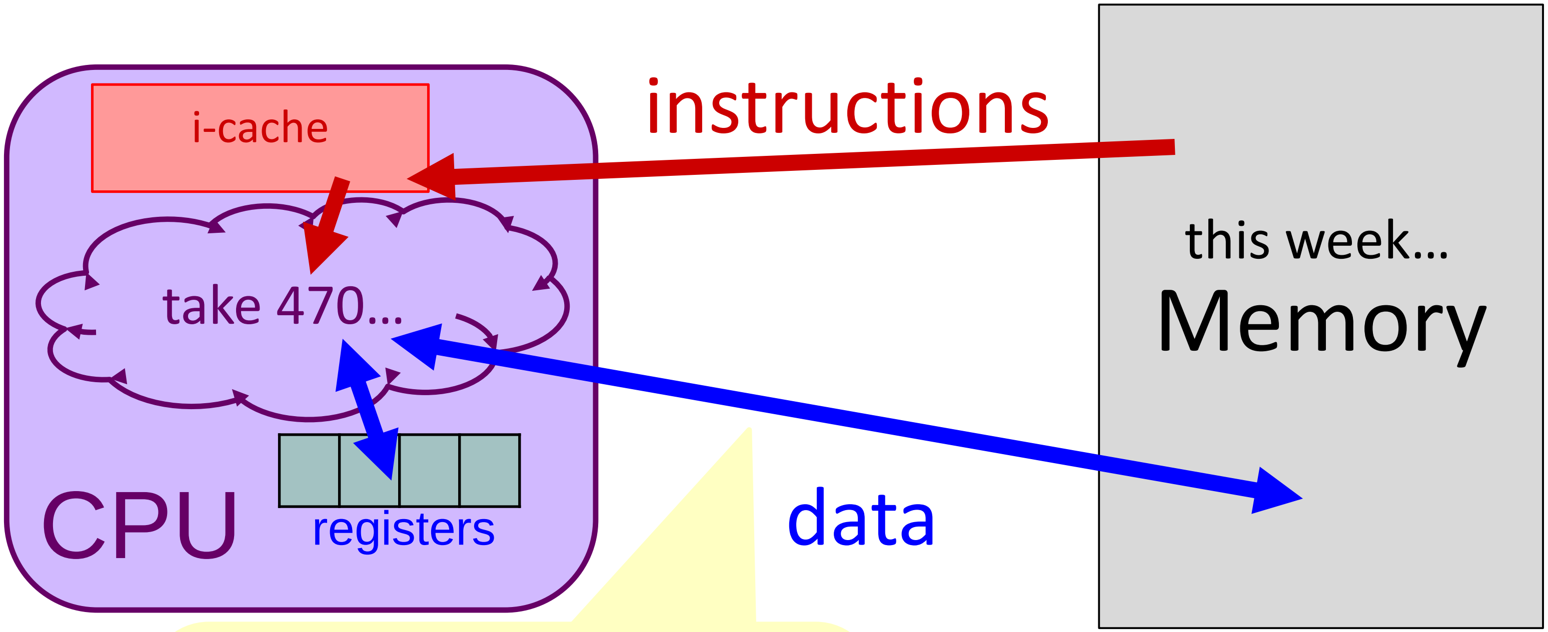
How does a program find its data in memory?

- emporarily in the tion cache
- arily in a fixe
- ching is HW
- **Data movement** is (assembly language)
- We'll learn about the instructions the CP
  - take computer architecture course 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:



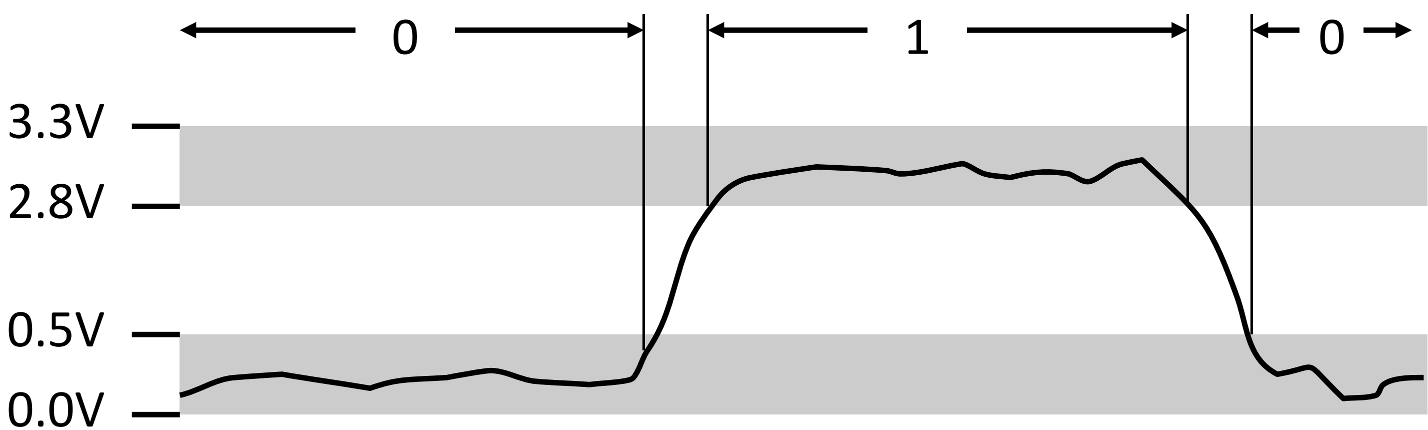
How are data and instructions represented?

Binary Representations

- Everything is bits
- 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 $d_{n-1}d_{n-2} \ldots d_1d_0$
  - 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

- Binary (00000000_2 – 11111111_2)
  - Byte = 8 bits (binary digits)

MSB

LSB

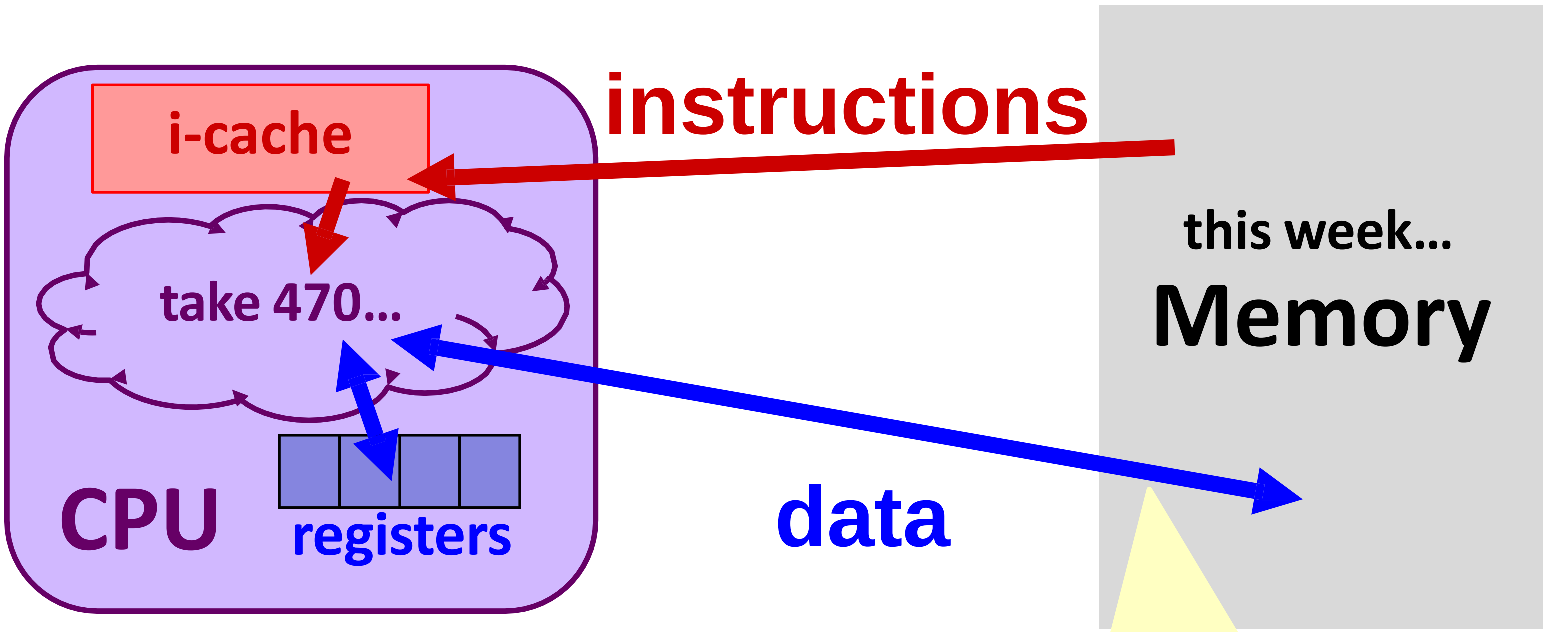
0	0	1	0	1	1	0	1
$0*2^7$	$0*2^6$	$1*2^5$	$0*2^4$	$1*2^3$	$1*2^2$	$0*2^1$	$1*2^0$
		32		8	4		1

$= 45_{10}$

- 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…

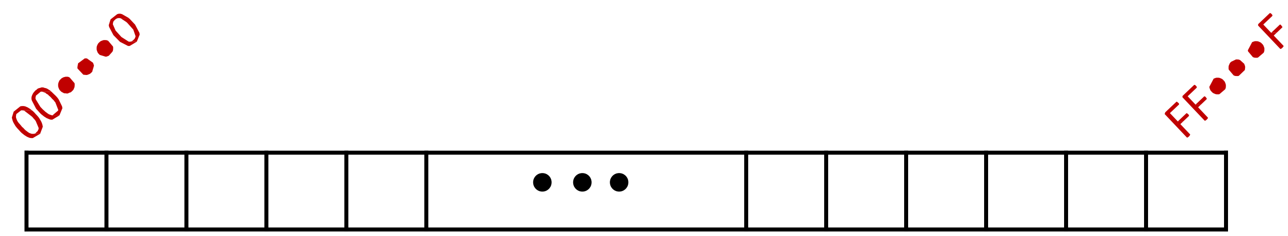
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Question 2:



How does a program find its data in memory?

Byte-Oriented Memory Organization



- Conceptually, memory is a single, large array of bytes, each with an 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
  - Until recently, most machines used **32-bit (4-byte) words**
    - Potential address space: 2^{32} addresses

      2^{32} bytes $\approx$ **4×10^9 bytes** = 4 billion bytes = **4GB**
    - Became too small for memory-intensive applications
  - 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**

Aside: Units and Prefixes

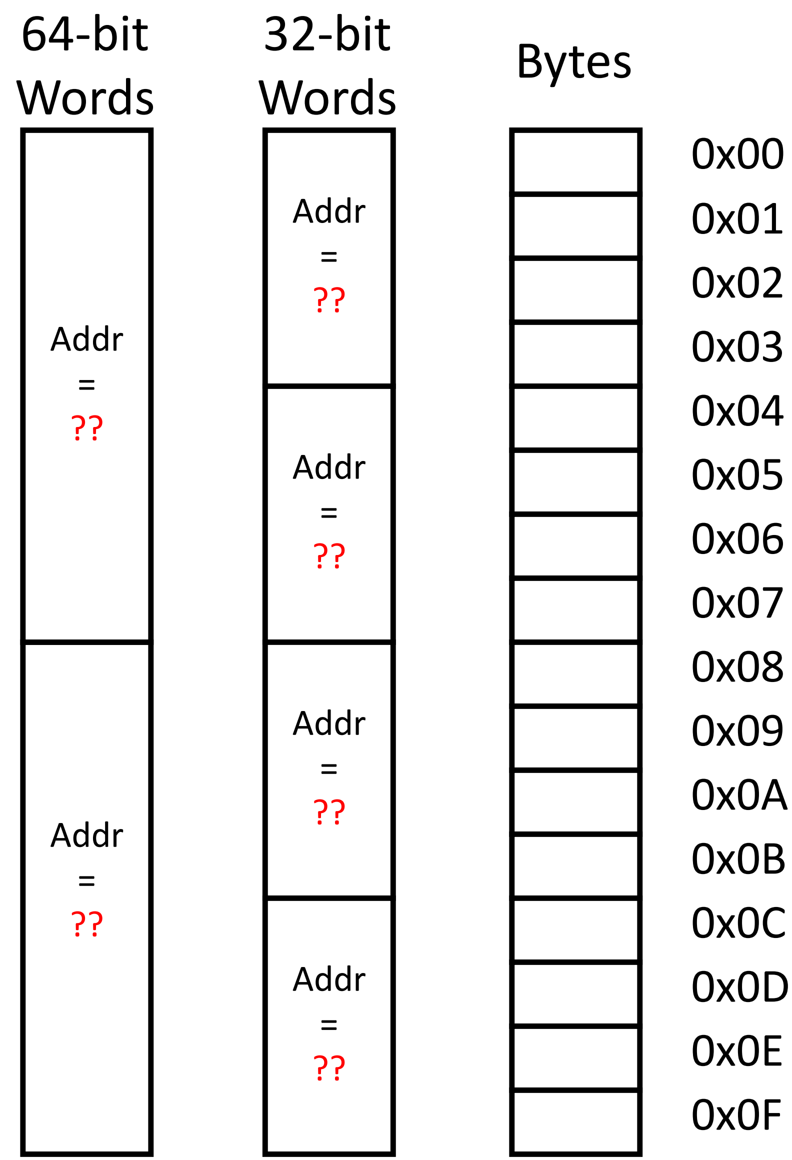
- Here focusing on large numbers (exponents > 0)
- Note that $10^3 \approx 2^{10}$
- SI prefixes are ambiguous if base 10 or 2
- IEC prefixes are unambiguously base 2

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

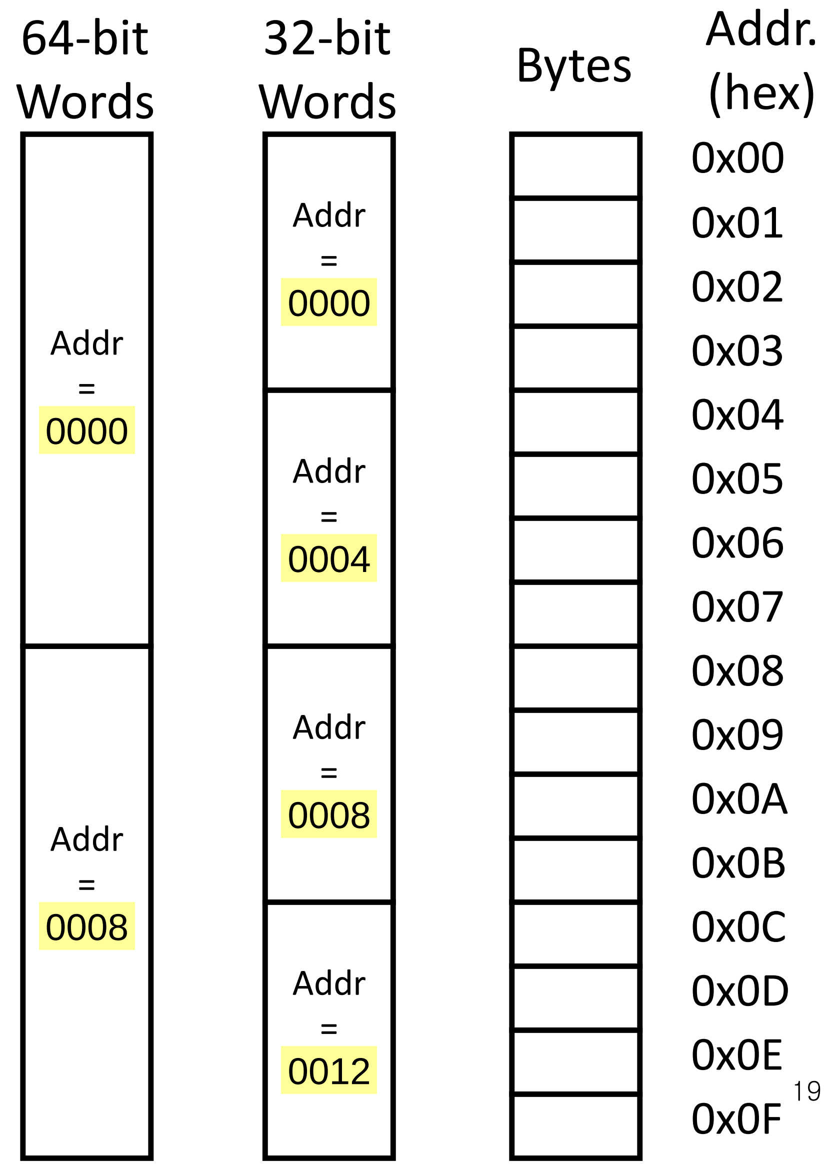
Word-Oriented Memory Organization

- 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?



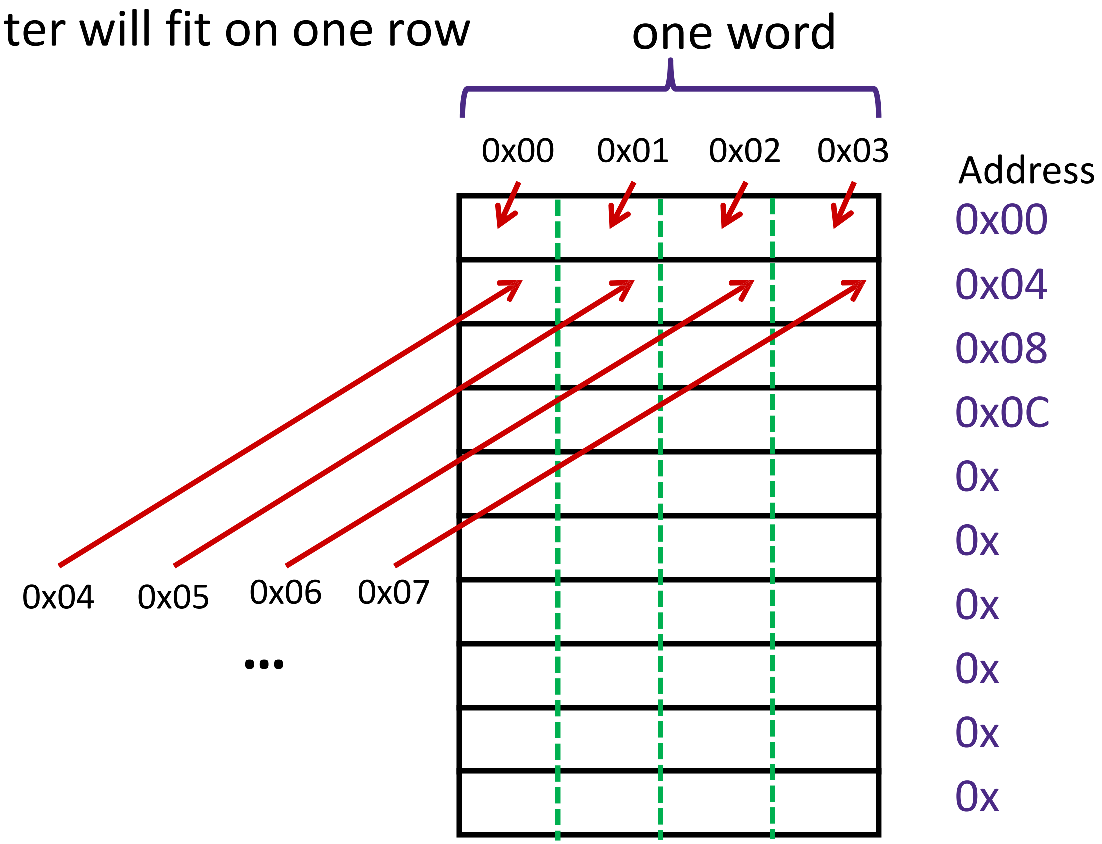
Word-Oriented Memory Organization

- 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?



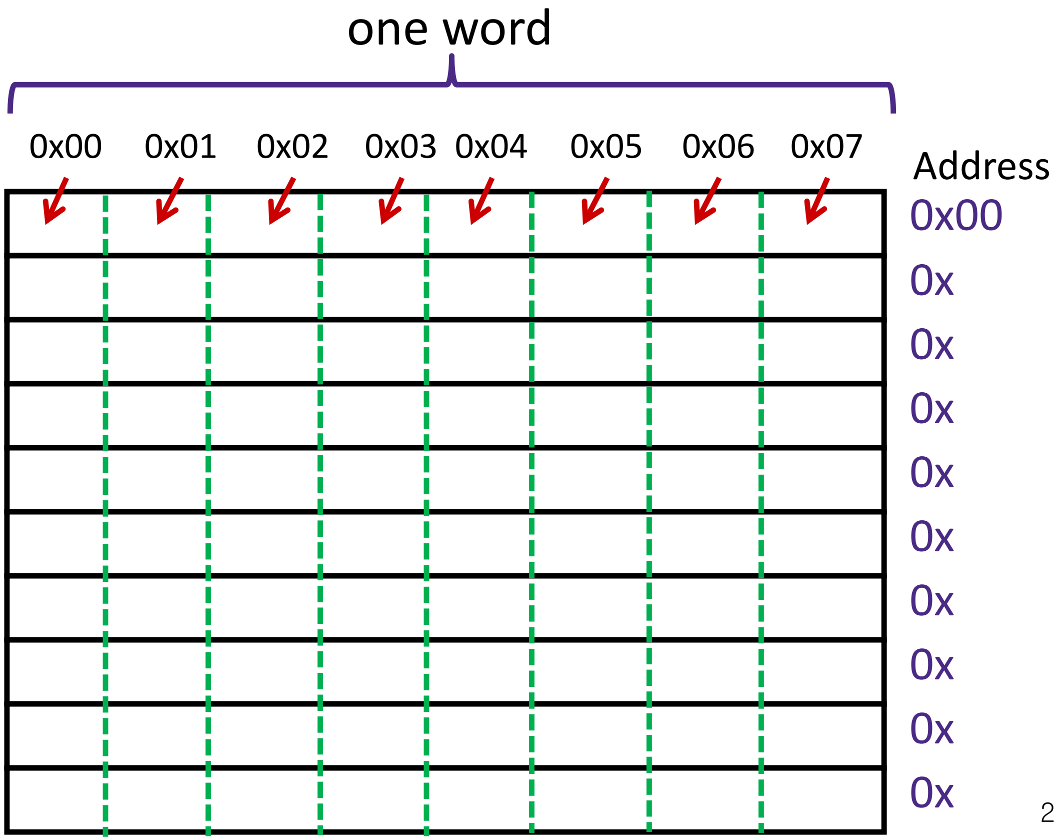
A Picture of Memory (32-bit view)

- **A "32-bit (4-byte) word-aligned" view of memory:**
  - In this type of picture, each row is composed of 4 bytes
  - Each cell is a byte
  - A 32-bit pointer will fit on one row



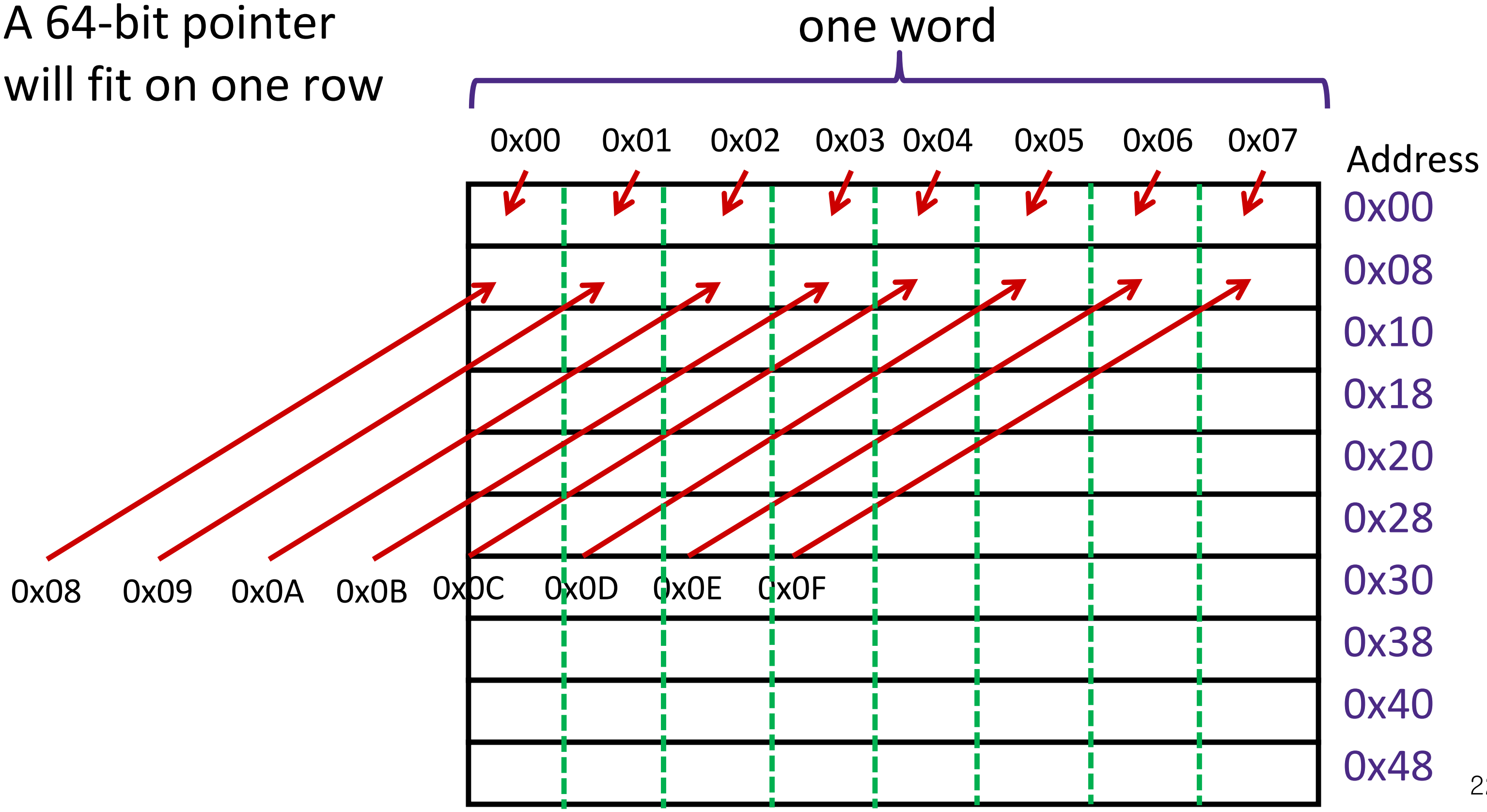
A Picture of Memory (64-bit view)

- 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



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 (1)

32-bit example
(pointers are 32-bits wide)

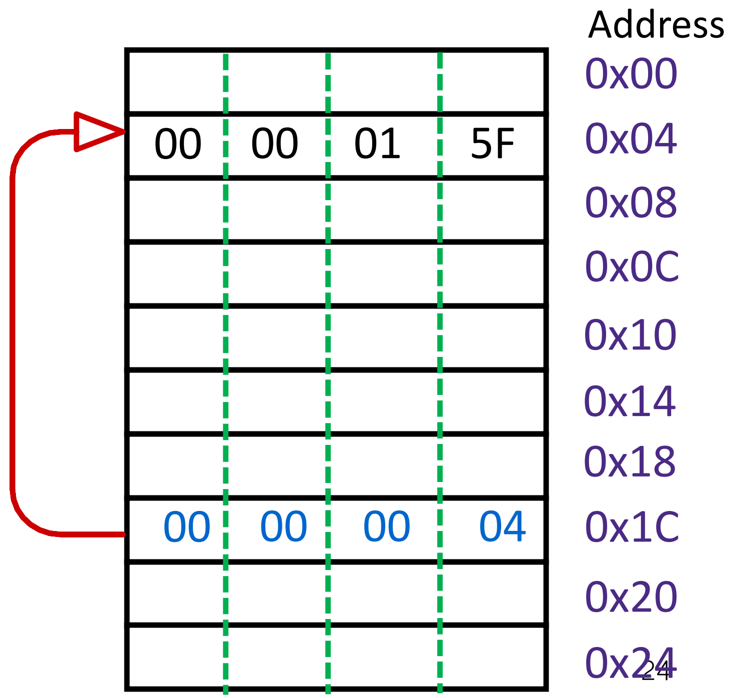
- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address 0x04
    - $351_{10} = 15F_{16} = \text{0x 00 00 01 5F}$

				Address
				0x00
00	00	01	5F	0x04
				0x08
				0x0C
				0x10
				0x14
				0x18
				0x1C
				0x20
				0x24

Addresses and Pointers (2)

32-bit example
(pointers are 32-bits wide)

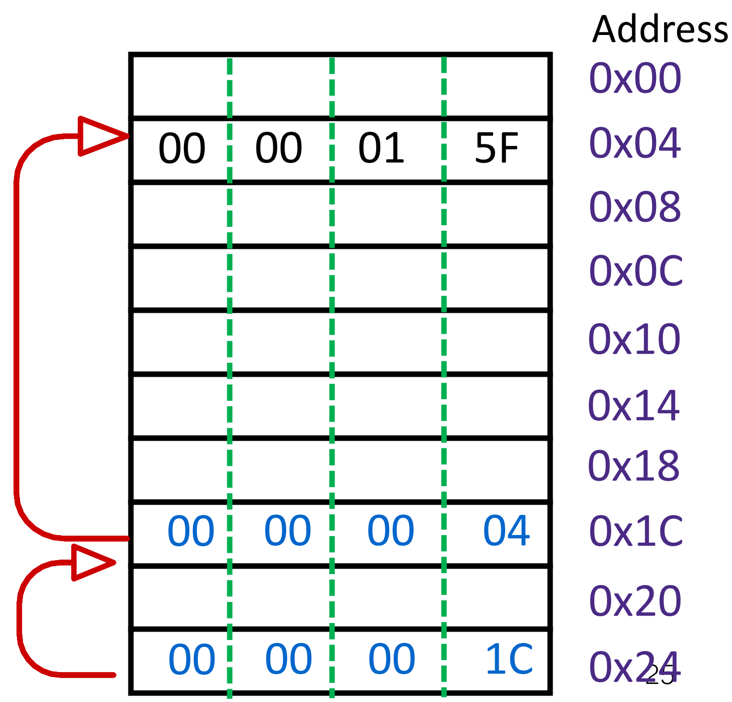
- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address 0x04
  - $351_{10} = 15F_{16} = \text{0x } 00\ 00\ 01\ 5F$
- **A pointer stored at address 0x1C points to address 0x04**



Addresses and Pointers (3)

32-bit example
(pointers are 32-bits wide)

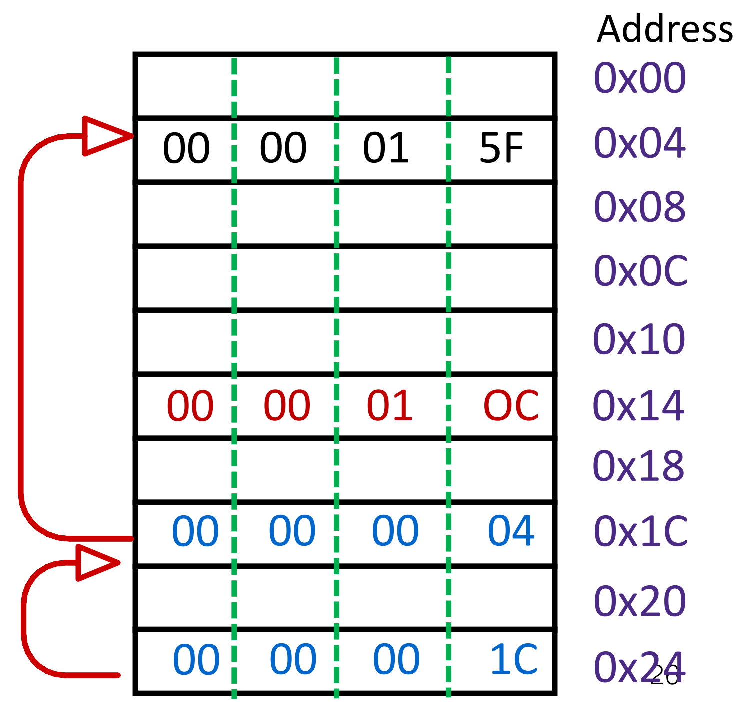
- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address 0x04
  - $351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F$
- A pointer stored at address 0x1C points to address 0x04
- **A pointer to a pointer is stored at address 0x24**



Addresses and Pointers (4)

32-bit example
(pointers are 32-bits wide)

- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address 0x04
    - $351_{10} = 15F_{16} = \text{0x00 00 01 5F}$
- A pointer stored at address 0x1C points to address 0x04
- A pointer to a pointer is stored at address 0x24
- **The value 12 is stored at address** 0x14
    - Is it a pointer?
        - Could be, depending on how you use it



Addresses and Pointers (5)

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

- **A 64-bit (8-byte) word-aligned view of memory**
- Value 351 stored at address 0x08
  - $351_{10} = 15F_{16}$ = 0x 00 00 01 5F
- Pointer stored at 0x38 points to address 0x08

(note hex addresses)

								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

Addresses and Pointers (6)

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

- **A 64-bit (8-byte) word-aligned view of memory**
- Value 351 stored at address 0x08
  - $351_{10} = 15F_{16} =$ 0x 00 00 01 5F
- 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

(note <u>hex</u> addresses)

								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

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

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

More about alignment later in the course

Byte Ordering

- How should bytes within a word be ordered in memory?
- Example:
  - **Store the 4-byte (32-bit) word: 0xa1 b2 c3 d4**
    - In what order will the bytes be stored?

- 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

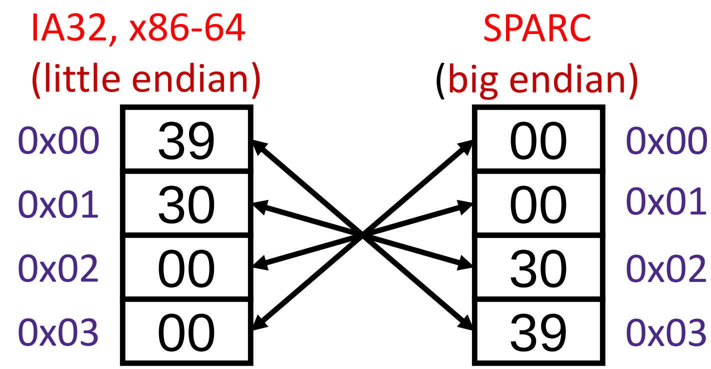
- **Big-Endian** (SPARC, z/Architecture, The Internet)
  - Least significant byte has highest address
- **Little-Endian** (x86, x86-64)
  - Least significant byte has lowest address
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little
- **Example:** 4-byte data 0xa1b2c3d4 at address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

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

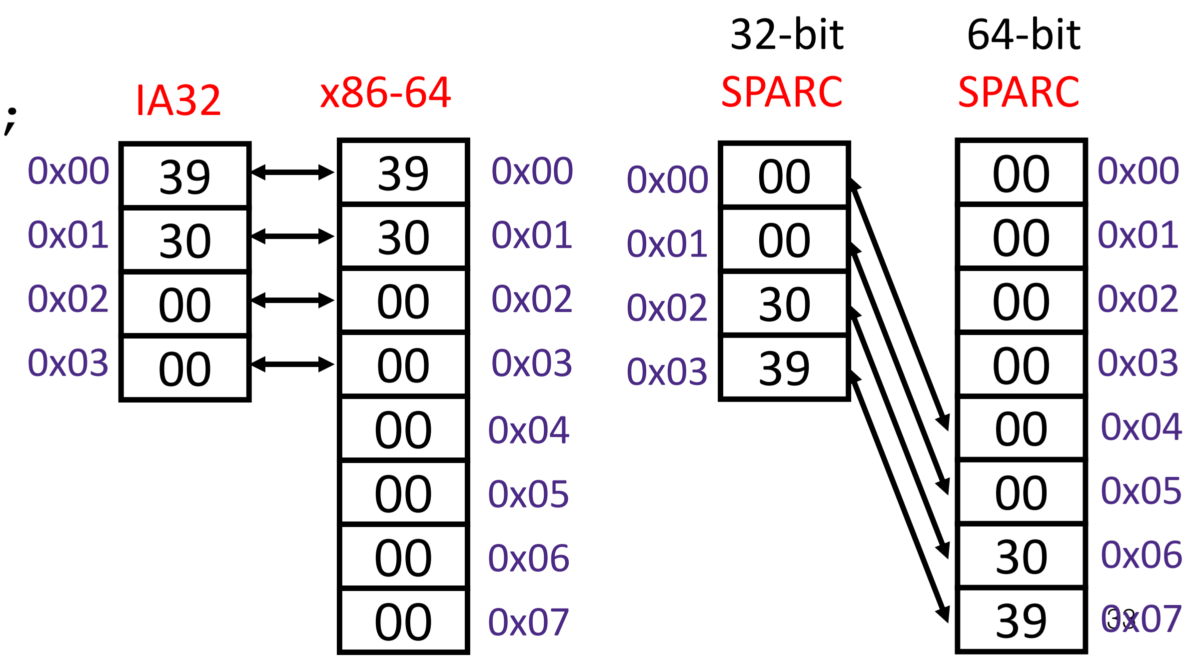
Byte Ordering Examples

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

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

- 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)*

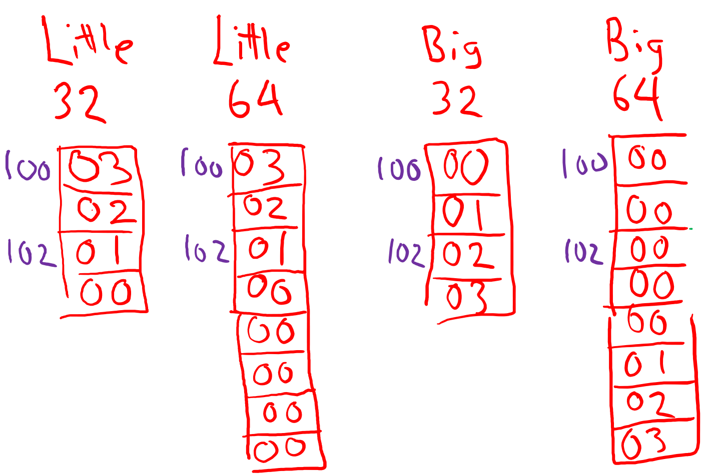
Address	Instruction Code	Assembly Rendition
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx

Deciphering numbers

- Value: `0x12ab`
- Pad to 32 bits: `0x000012ab`
- Split into bytes: `00 00 12 ab`
- Reverse (little-endian): `ab 12 00 00`

Question:

- We store the value `0x 00 01 02 03` as a long int (size of long int = size of pointer) 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

Q&A