Memory, Data, & Addressing I

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

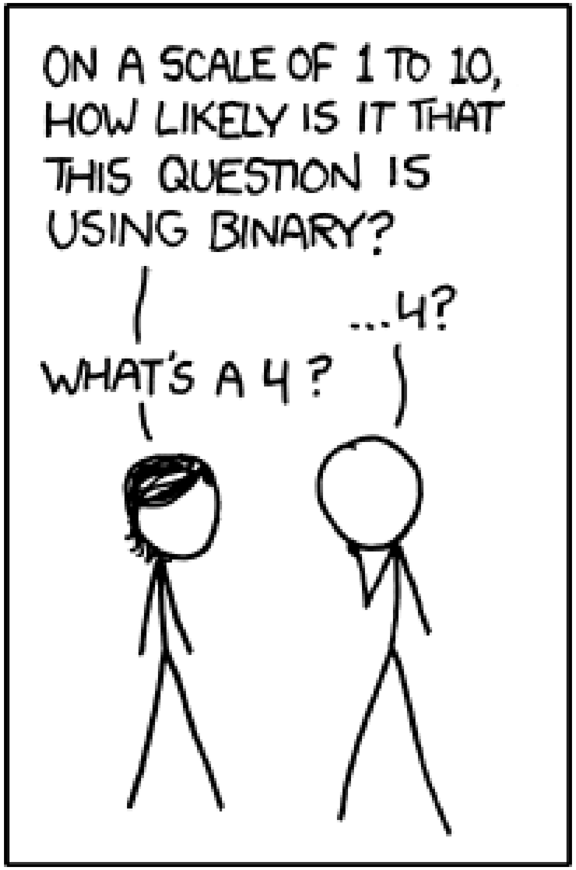
Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja


ON A SCALE OF 1 TO 10, HOW LIKELY IS IT THAT THIS QUESTION IS USING BINARY?
WHAT'S A 4?
...4?

http://xkcd.com/953/

Admin

- Pre-Course Survey and hw0 due tonight @ 11:59 pm
  - Starting Week 2: hw due at 11:00 am (Seattle time)
- hw1 due Friday (1/8) @ 11:59 pm
- hw2 due Monday (1/11) @ 11:00 am
- Lab 0 due Friday (1/8) @ 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 you 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

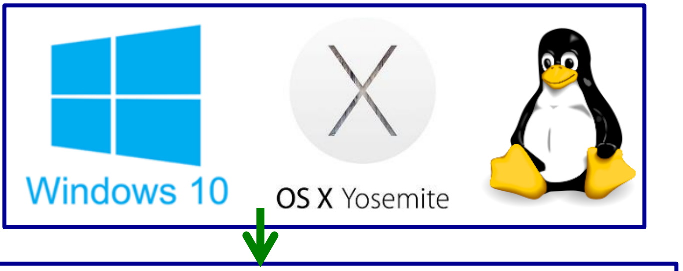
Assembly language:

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

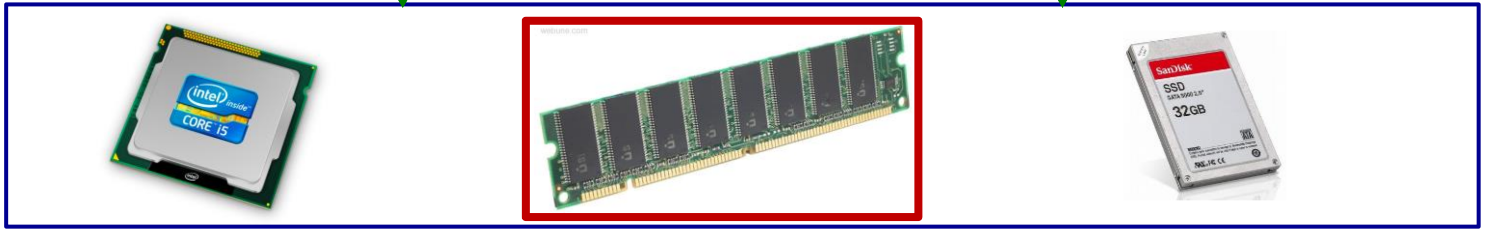
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:

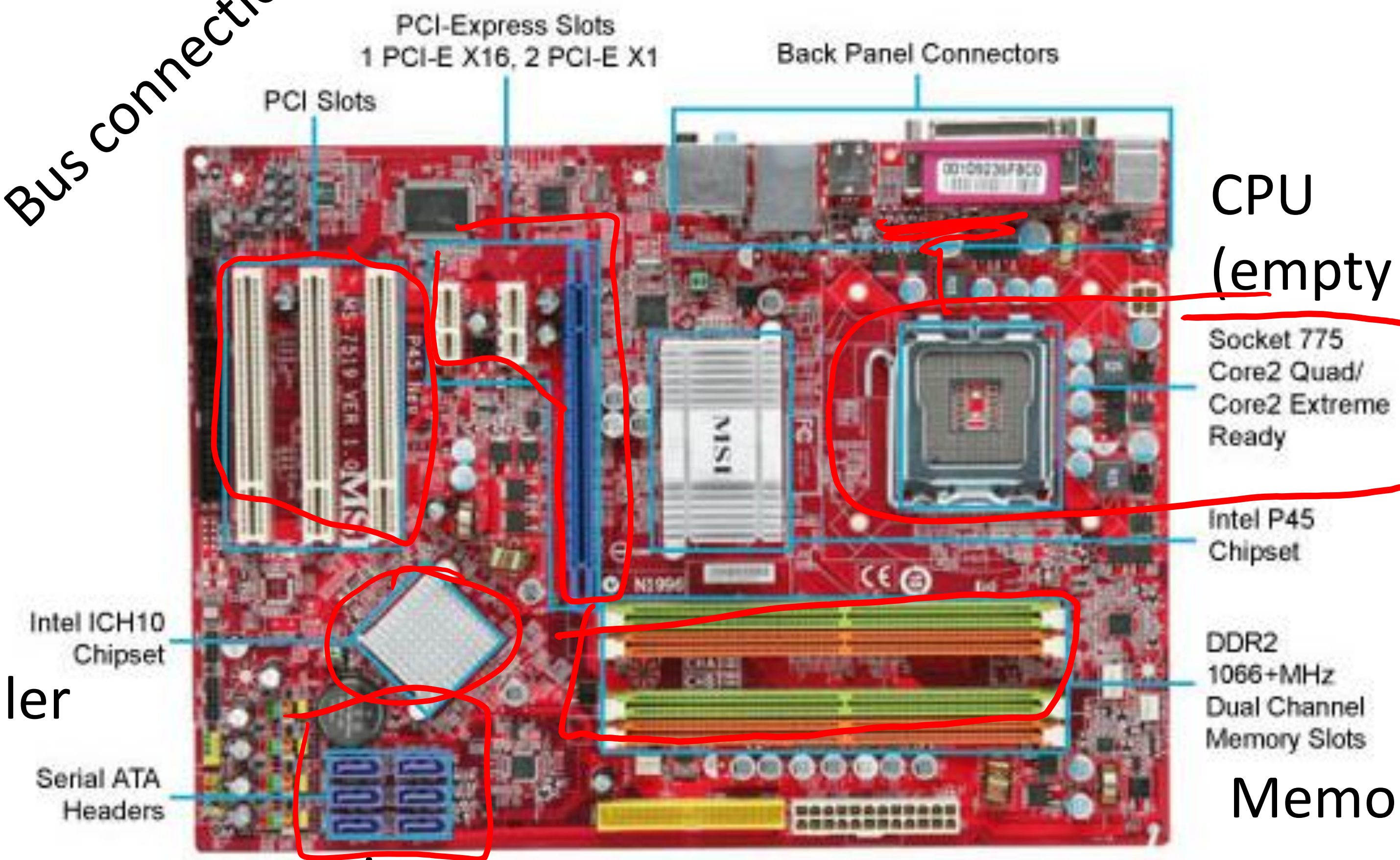


Reading Review

- 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



Bus connections

USB…

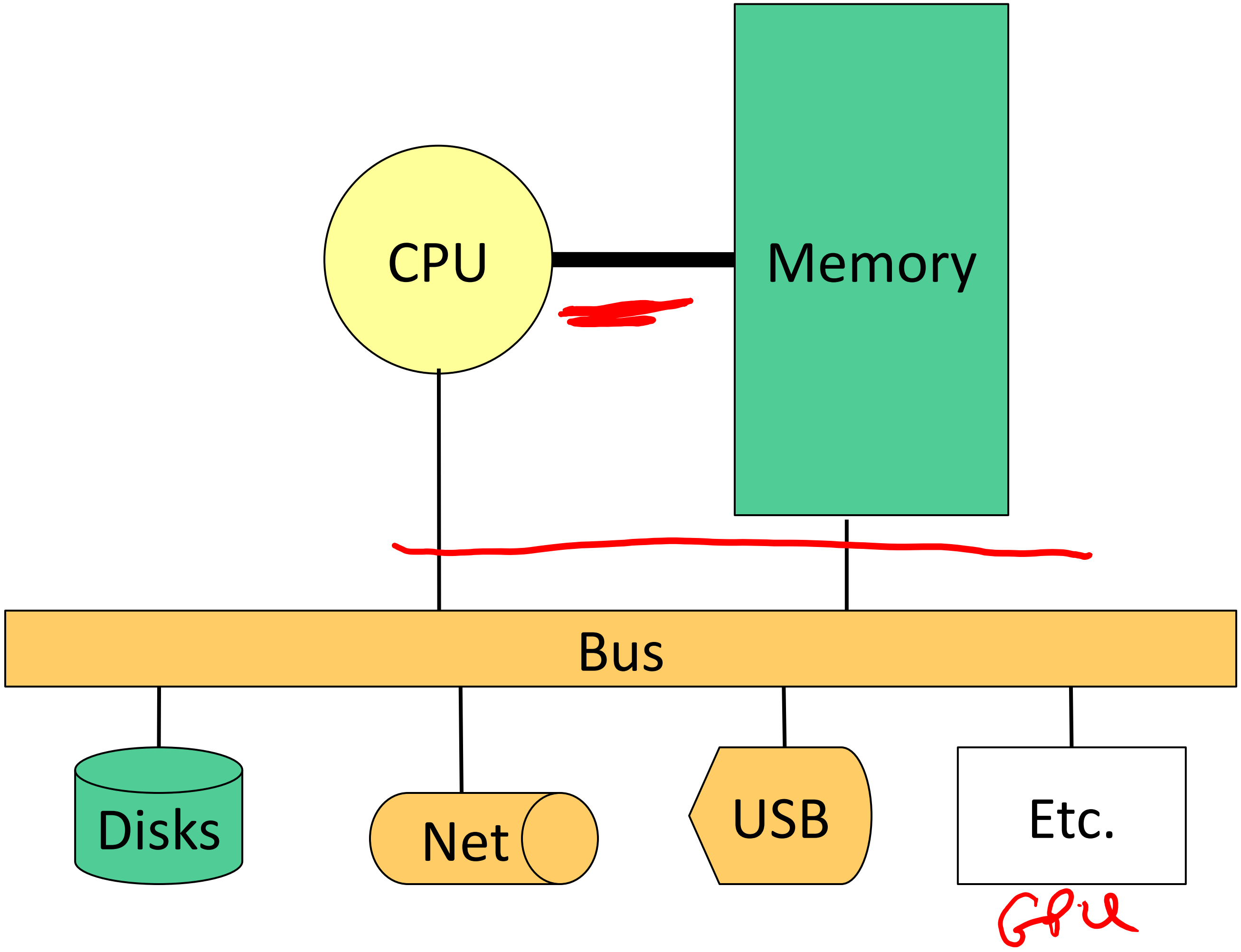
CPU
(empty slot)

I/O
controller

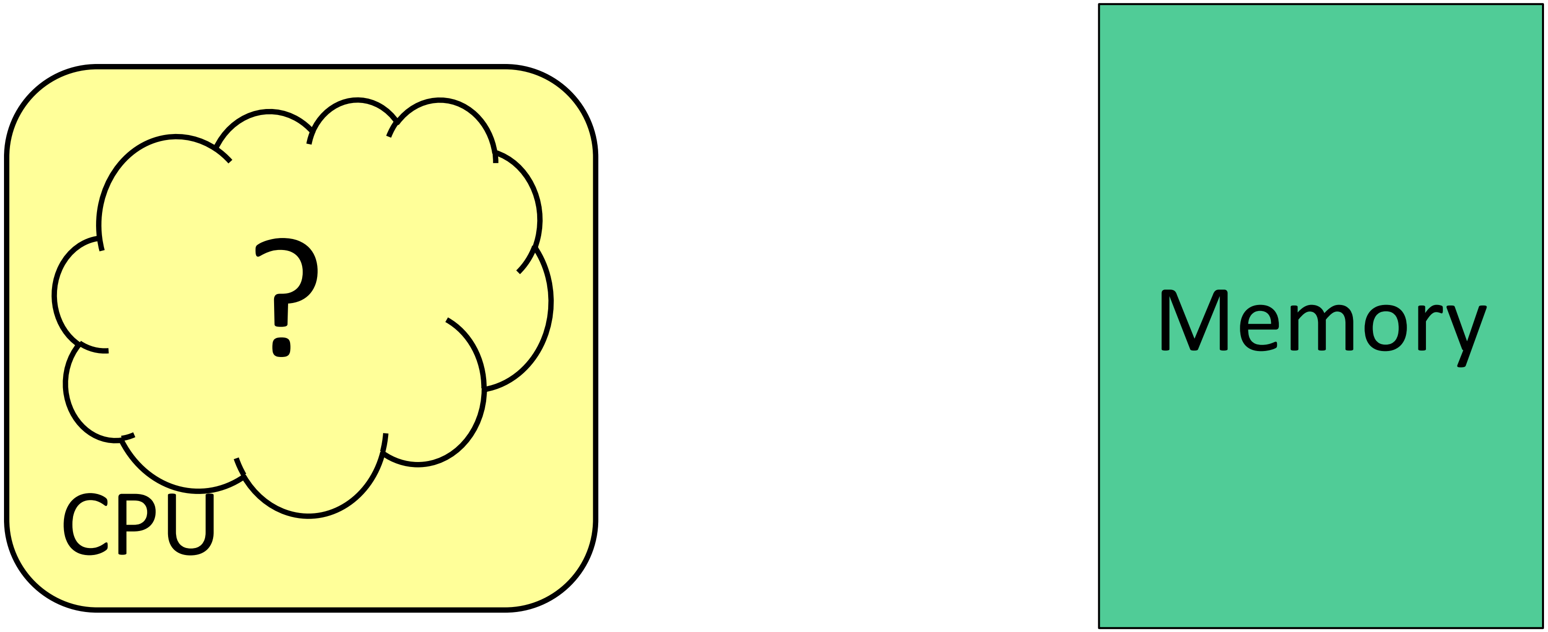
Memory

Storage connections

Hardware: Logical View



Hardware: 351 View (version 0)

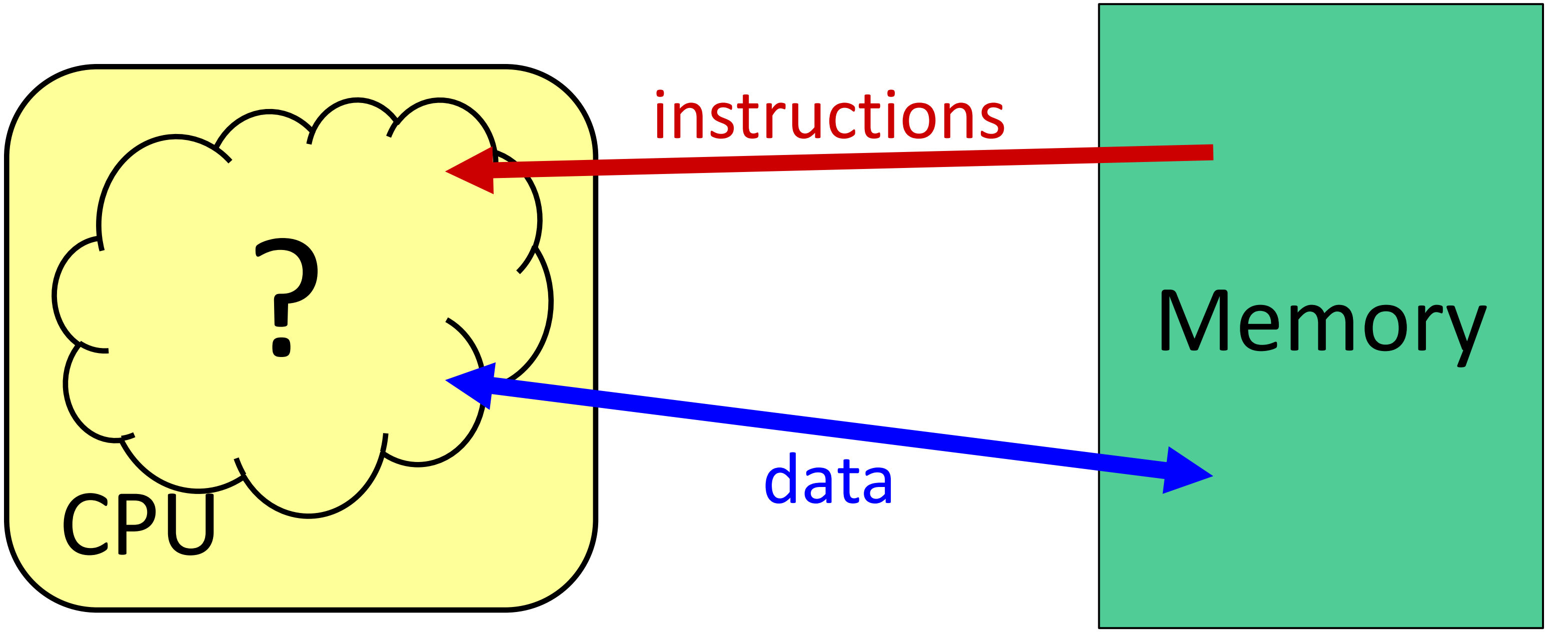


- The CPU executes instructions
- Memory stores data +instructions

How are data and instructions represented?

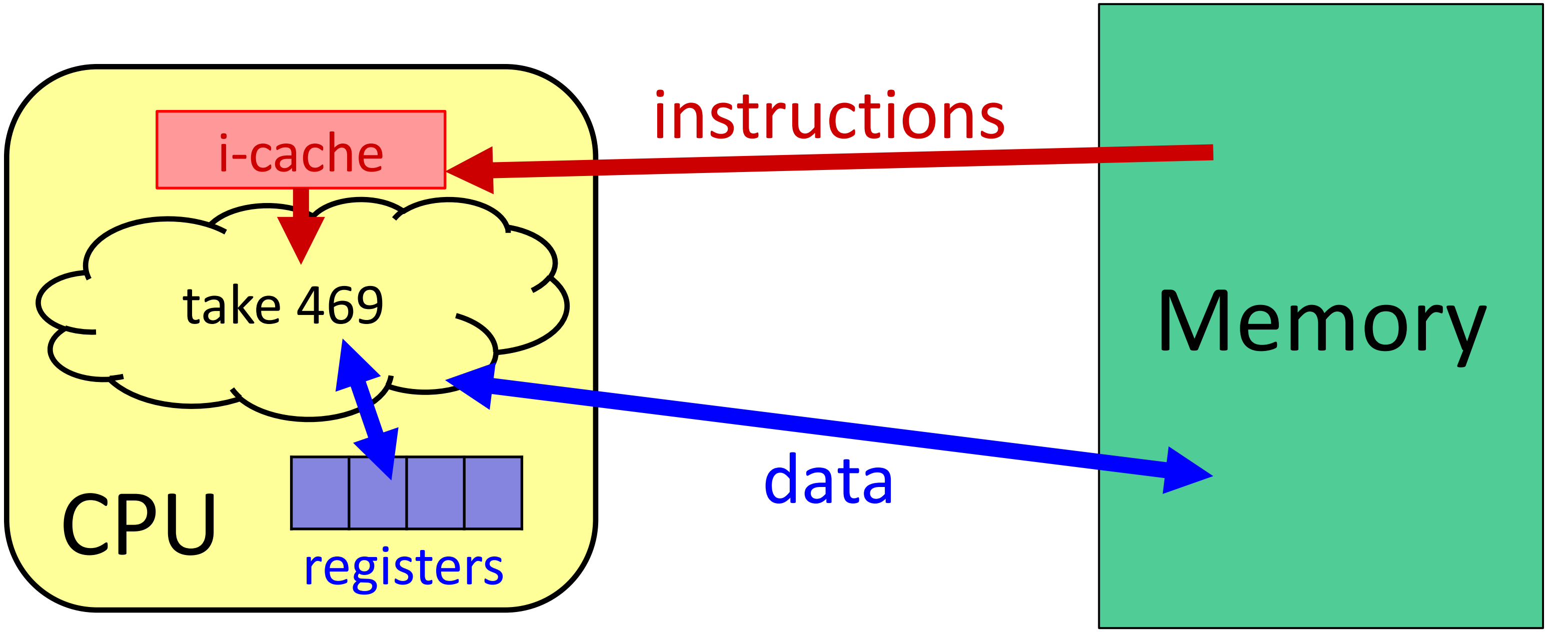
- Binary encoding!
  - Instructions *are* just data

Hardware: 351 View (version 0)



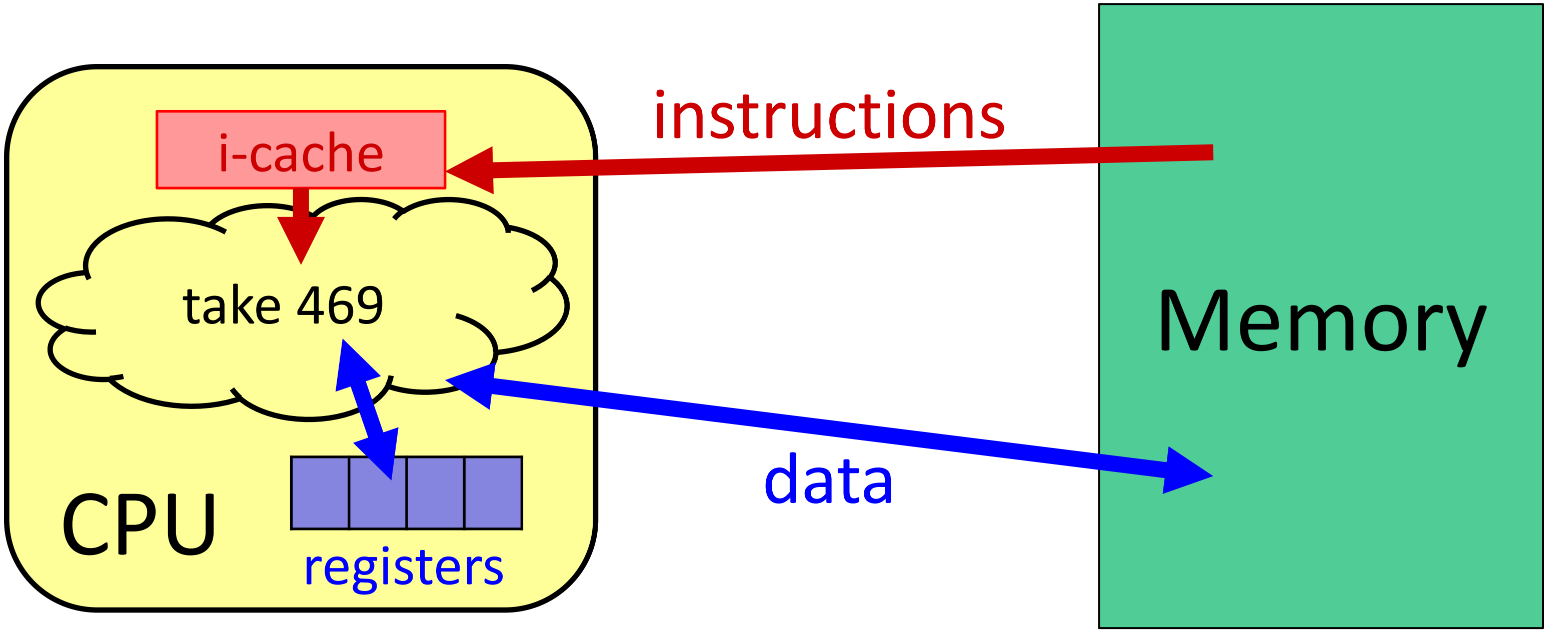
- To execute an instruction, the CPU must:
  1) Fetch the instruction
  2) (if applicable) Fetch data needed by the instruction
  3) Perform the specified computation
  4) (if applicable) Write the result back to memory

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?

- Addresses!
  - Can be stored in *pointers*

Review Questions – Ed Lessons (1.5)

- 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] need encoding!

- We can fetch a piece of data from memory as long as we have its address.

  A. True [✓ B. False]

  need: address ✓
  size ✗

- Which of the following bytes have a most-significant bit (MSB) of 1?

  A. 0x63 [✓ B. 0x90] [✓ C. 0xCA] D. 0xF

  0b01100011 0b10010000 0b11001010 0b0000 1111

  0x0F

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 = 0x04

Least Significant Bit (LSB)

Most Significant Bit (MSB)

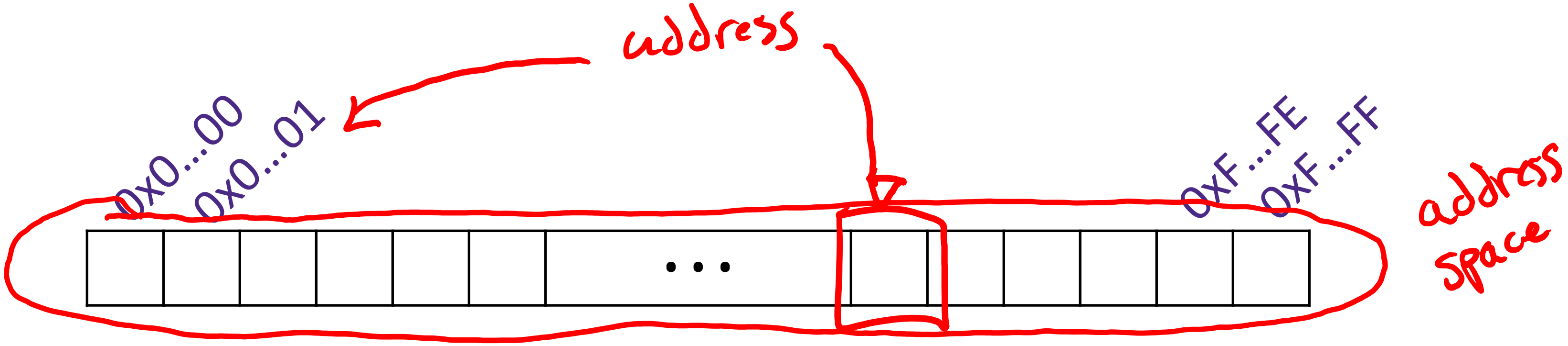
Bits and Bytes and Things

- 1 byte = 8 bits
- n bits can represent up to 2^n things
  - Sometimes (oftentimes?) those “things” are bytes!

1,0 1,0

$\underset{3}{\underline{2}}\ \underset{2}{\underline{2}}\ \underset{1}{\underline{2}}\ \underset{0}{\underline{2}}$ $4=2^4$

- If addresses are a-bits wide, how many distinct addresses are there? 2^a
- What does each address refer to? byte



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 → not x86
- We have *chosen* to tie word size to address size/width
  - word size = address size = register size
  - word size = w bits → 2^w addresses
- Current x86 systems use **64-bit (8-byte) words**
  - Potential address space: $\mathbf{2^{64}}$ addresses
    2^{64} bytes ≈ **1.8 x $\mathbf{10^{19}}$ bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

Data Representations

- Sizes of data types (in bytes)

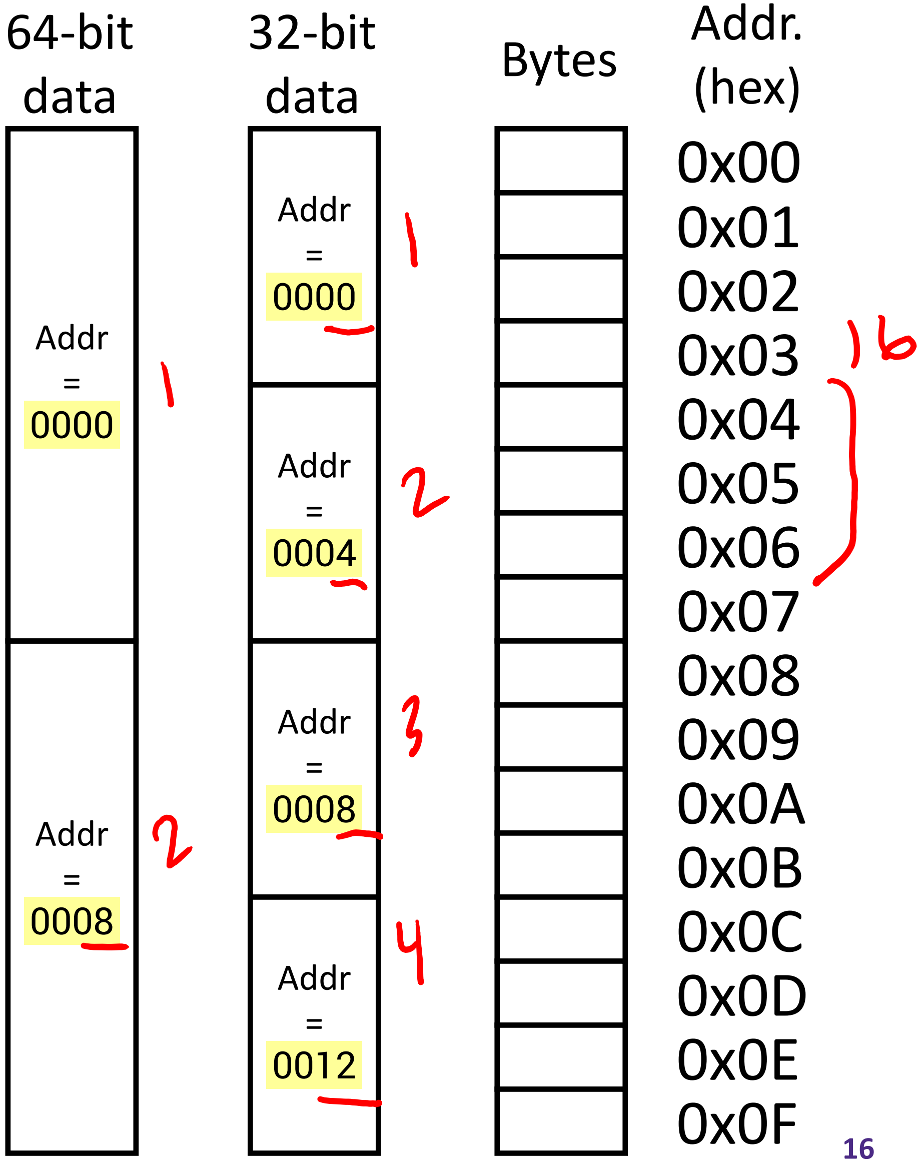
Java Data Type	C Data Type	~~32-bit (old)~~	x86-64
`boolean`	`bool`	~~1~~	1
`byte`	`char`	~~1~~	1
`char`		~~2~~	2
`short`	`short int`	~~2~~	2
`int`	`int`	~~4~~	4
`float`	`float`	~~4~~	4
	`long int`	~~4~~	8
`double`	`double`	~~8~~	8
`long`	`long long`	~~8~~	8
	`long double`	~~8~~	16
`(reference)`	**`pointer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

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**



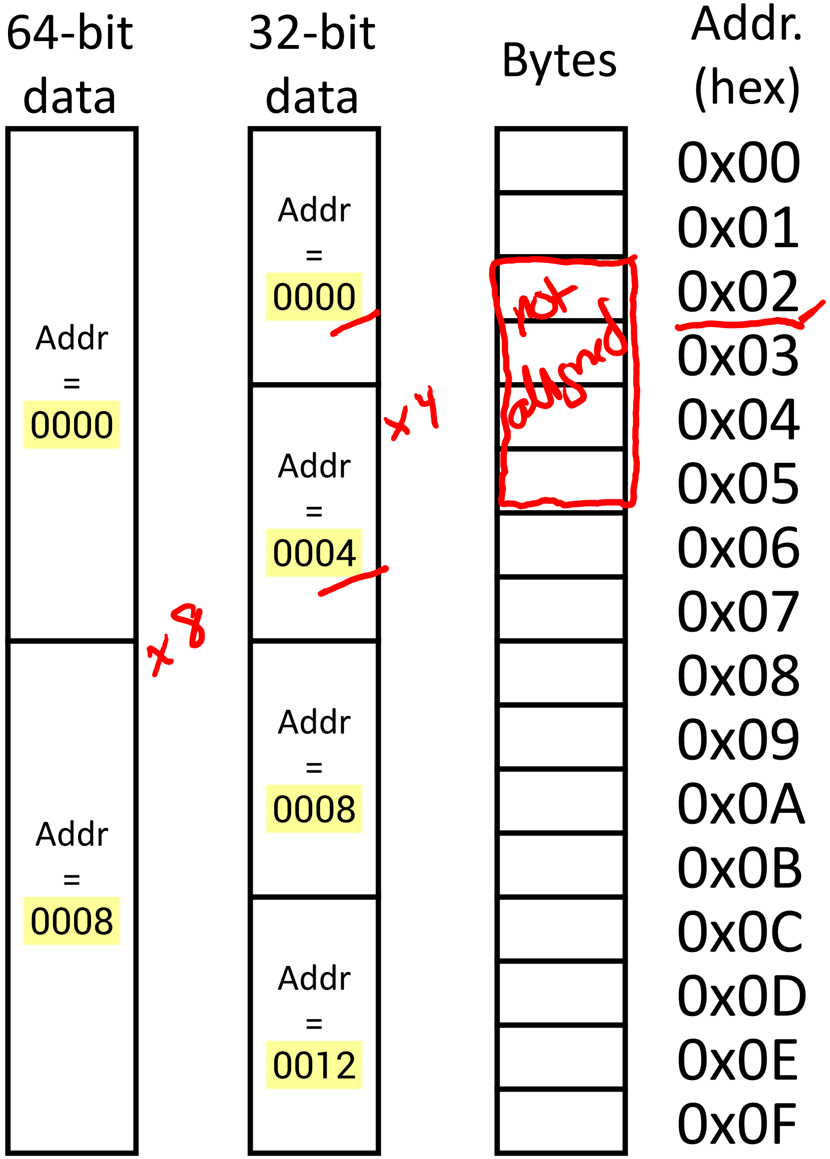
Alignment

- 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 % k = 0

↑

aligment



64-bit data

Addr = 0000

Addr = 0008

32-bit data

Addr = 0000

Addr = 0004

Addr = 0008

Addr = 0012

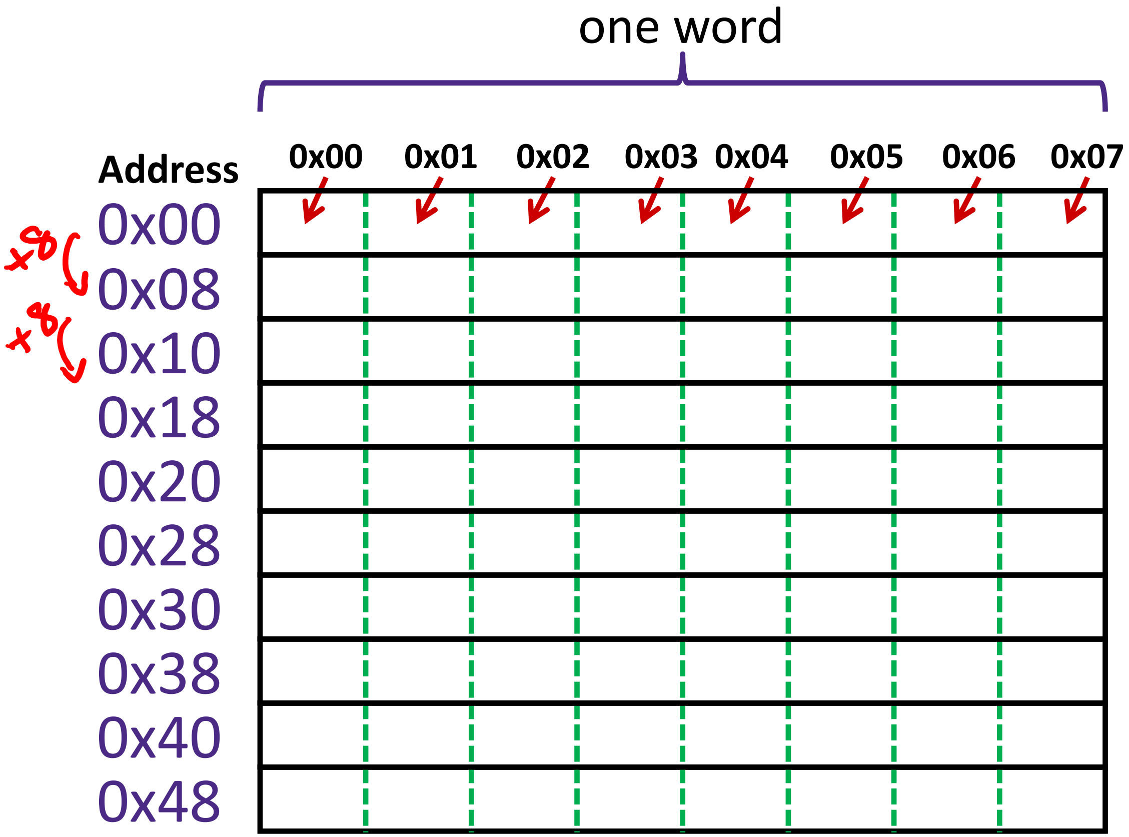
Bytes

Addr. (hex)

0x00
0x01
0x02
0x03
0x04
0x05
0x06
0x07
0x08
0x09
0x0A
0x0B
0x0C
0x0D
0x0E
0x0F

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



one word

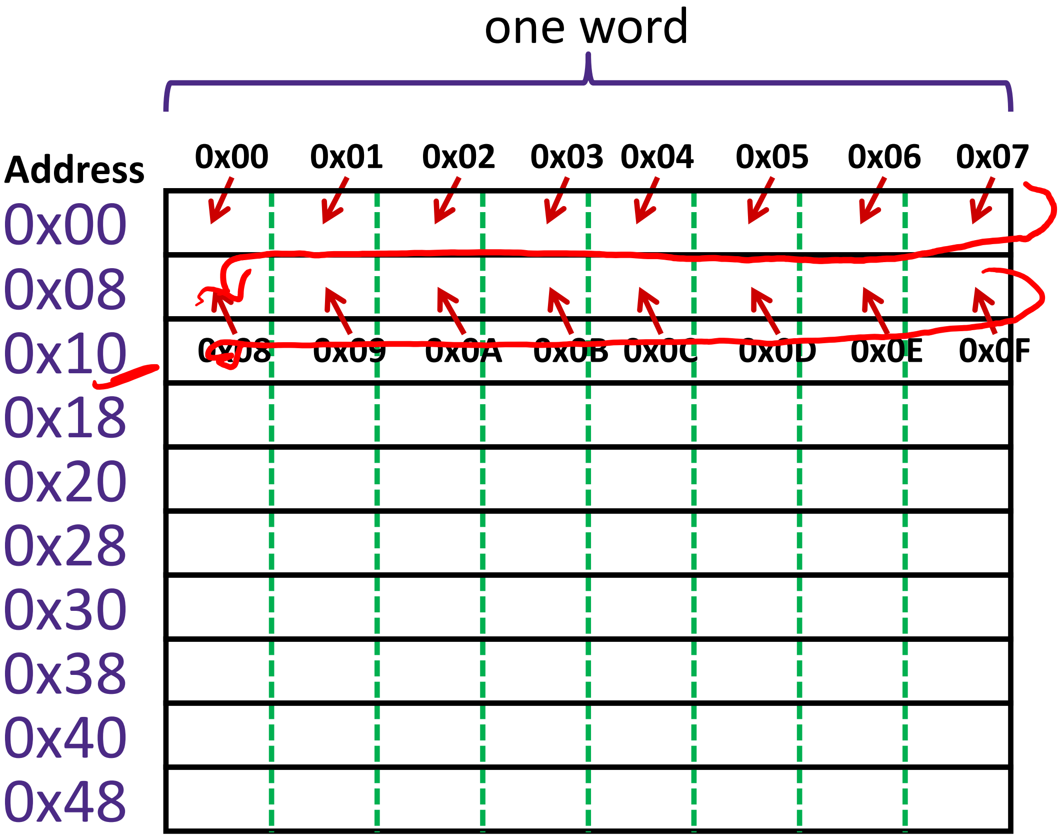
Address

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

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

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



one word

Address

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

0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F

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

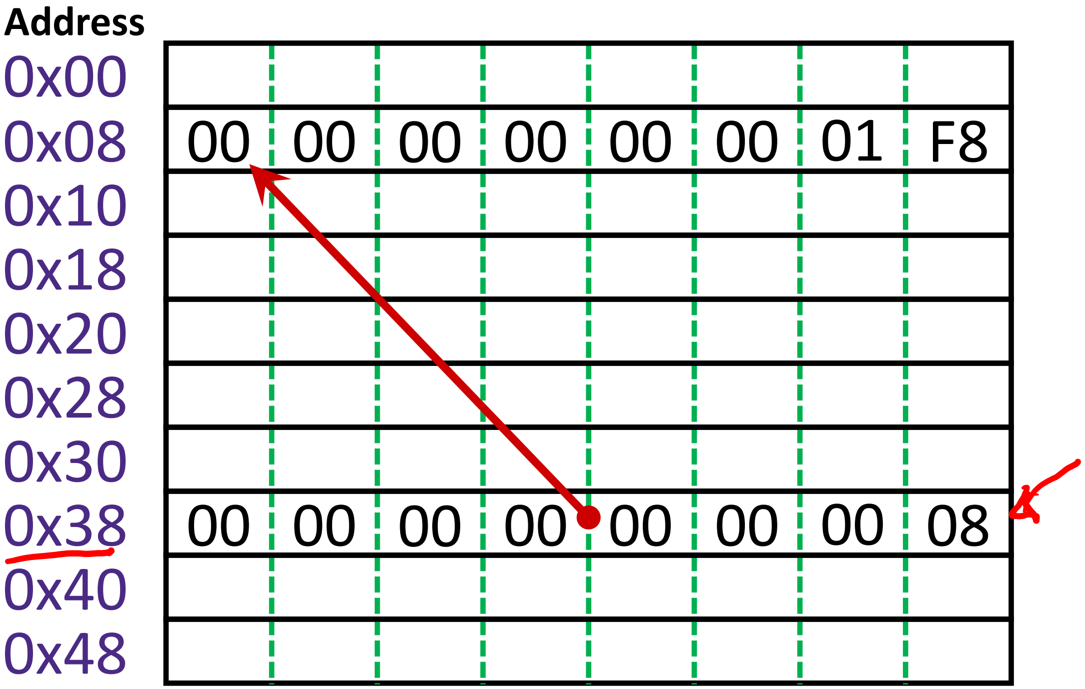
Addresses and Pointers

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

big-endian

address space
$a \rightarrow 2^a$

- 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



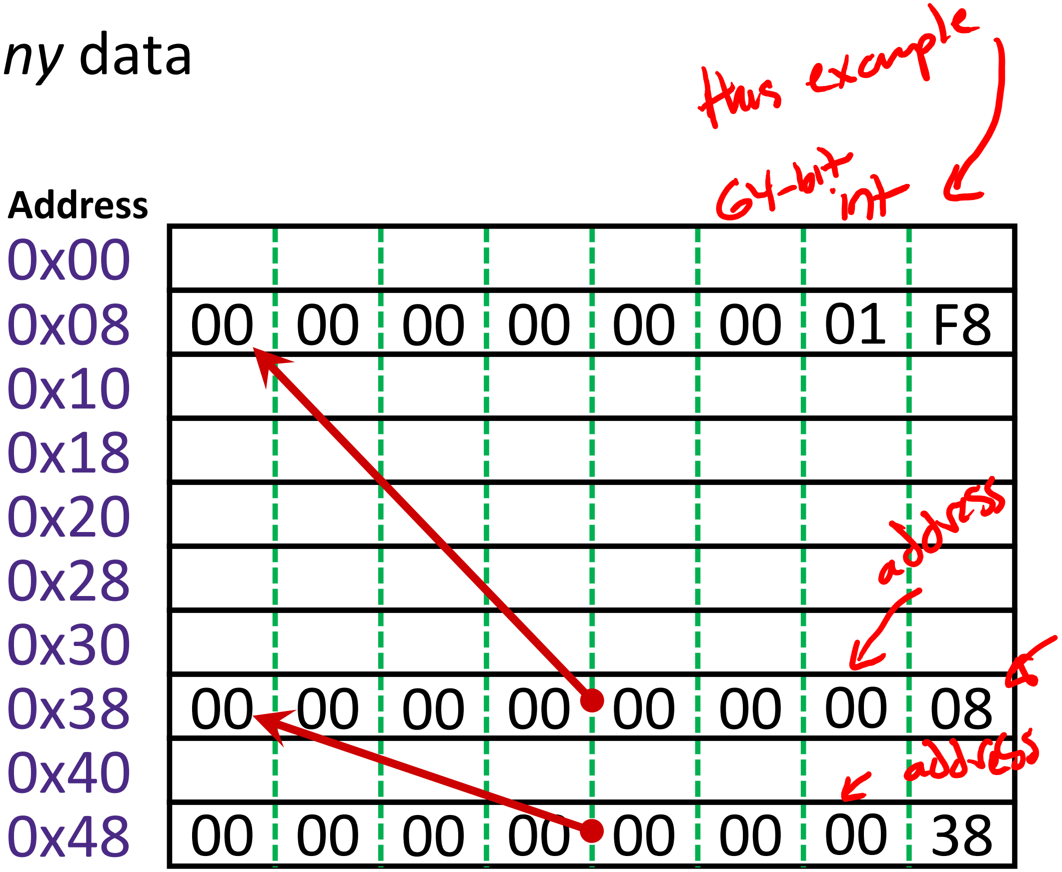
Address								
0x00								
0x08	00	00	00	00	00	00	01	F8
0x10								
0x18								
0x20								
0x28								
0x30								
0x38	00	00	00	00	00	00	00	08
0x40								
0x48								

Addresses and Pointers

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

big-endian

- An *address* 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 → 0x | D4 | C3 | B2 | A1 |

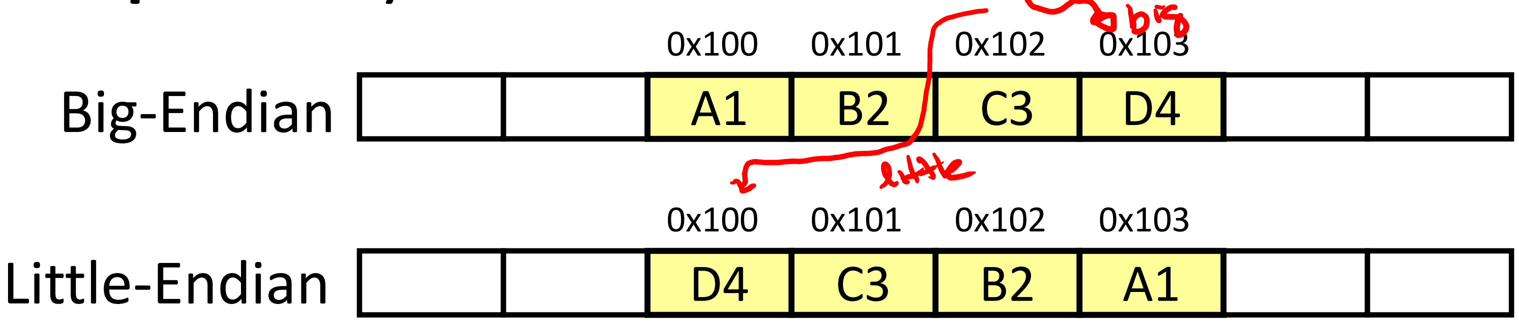
- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian
    - In which address does the least significant *byte* go?
    - Based on *Gulliver's Travels*: tribes cut eggs on different sides (big, little)

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

MSB LSB

- **Example:** 4-byte data 0xA1B2C3D4 at address 0x100



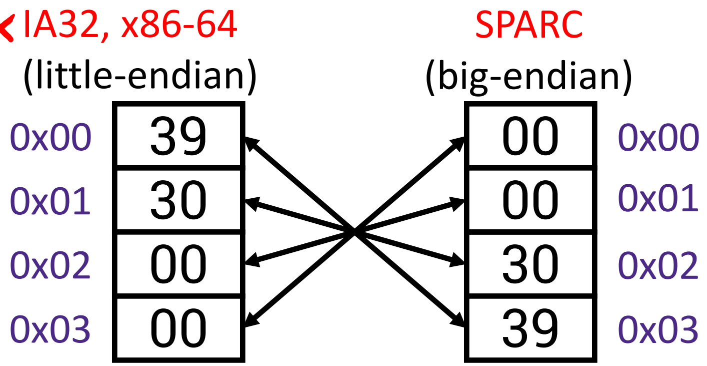
			0x100	0x101	0x102	0x103		
Big-Endian			A1	B2	C3	D4		
			0x100	0x101	0x102	0x103		
Little-Endian			D4	C3	B2	A1		

Byte Ordering Examples

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

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

int = 32-bit

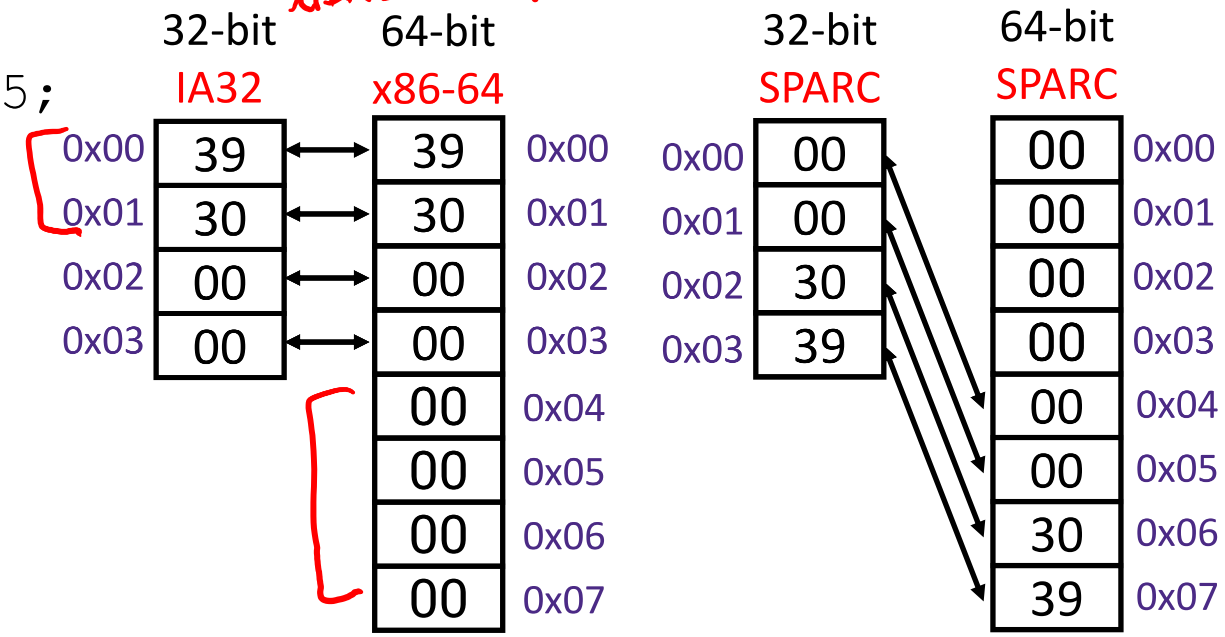


	IA32, x86-64 (little-endian)	SPARC (big-endian)	
0x00	39	00	0x00
0x01	30	00	0x01
0x02	00	30	0x02
0x03	00	39	0x03

64-bit

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

little endian



	32-bit IA32	64-bit x86-64		32-bit SPARC	64-bit SPARC	
0x00	39	39	0x00	00	00	0x00
0x01	30	30	0x01	00	00	0x01
0x02	00	00	0x02	30	00	0x02
0x03	00	00	0x03	39	00	0x03
		00	0x04		00	0x04
		00	0x05		00	0x05
		00	0x06		30	0x06
		00	0x07		39	0x07

(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

word = 8-bytes

0x100				0x104			0x107
00	00	00	00	01	02	03	04

0x 00 00 0000 01 02 03 04

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.

		0x100							0x107	
…	9F	23	B7	C8	55	D0	00	04	08	…

→ 4-bytes

- 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

  $w = 2^w$

  - 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