Memory, Data, & Addressing I

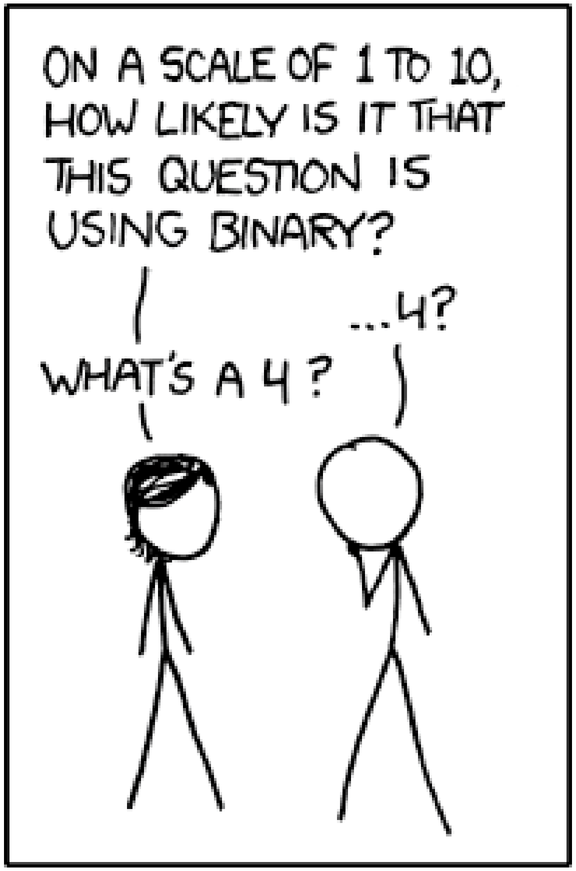
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh


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/

Administrivia

- Pre-Course Survey due tonight @ 11:59 pm
- Lab 0 due Monday (4/08)
- Homework 1 due Wednesday (4/10)

- All course materials can be found on the website schedule

- **Get your machine set up for this class (VM or attu) as soon as possible!**
  - Bring your laptop to section tomorrow if you are having trouble.

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:

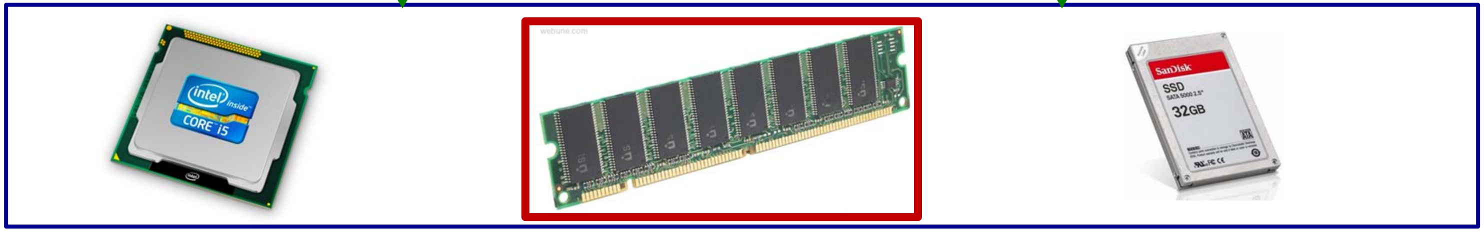
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

Computer system:

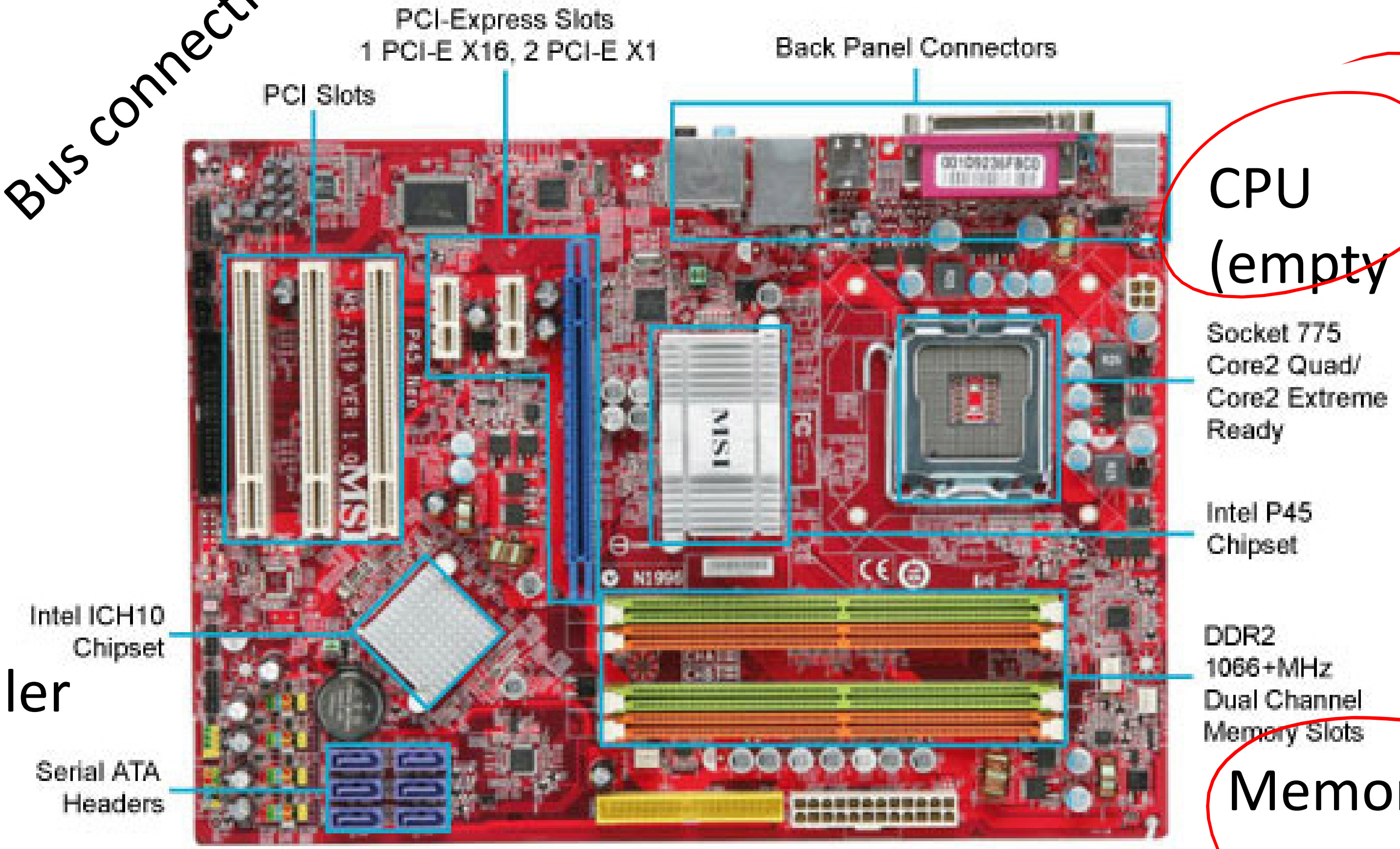


- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Memory, Data, and Addressing

- Hardware - High Level Overview
- Representing information as bits and bytes
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
- Organizing and addressing data in memory
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

Hardware: Physical View



Bus connections

USB…

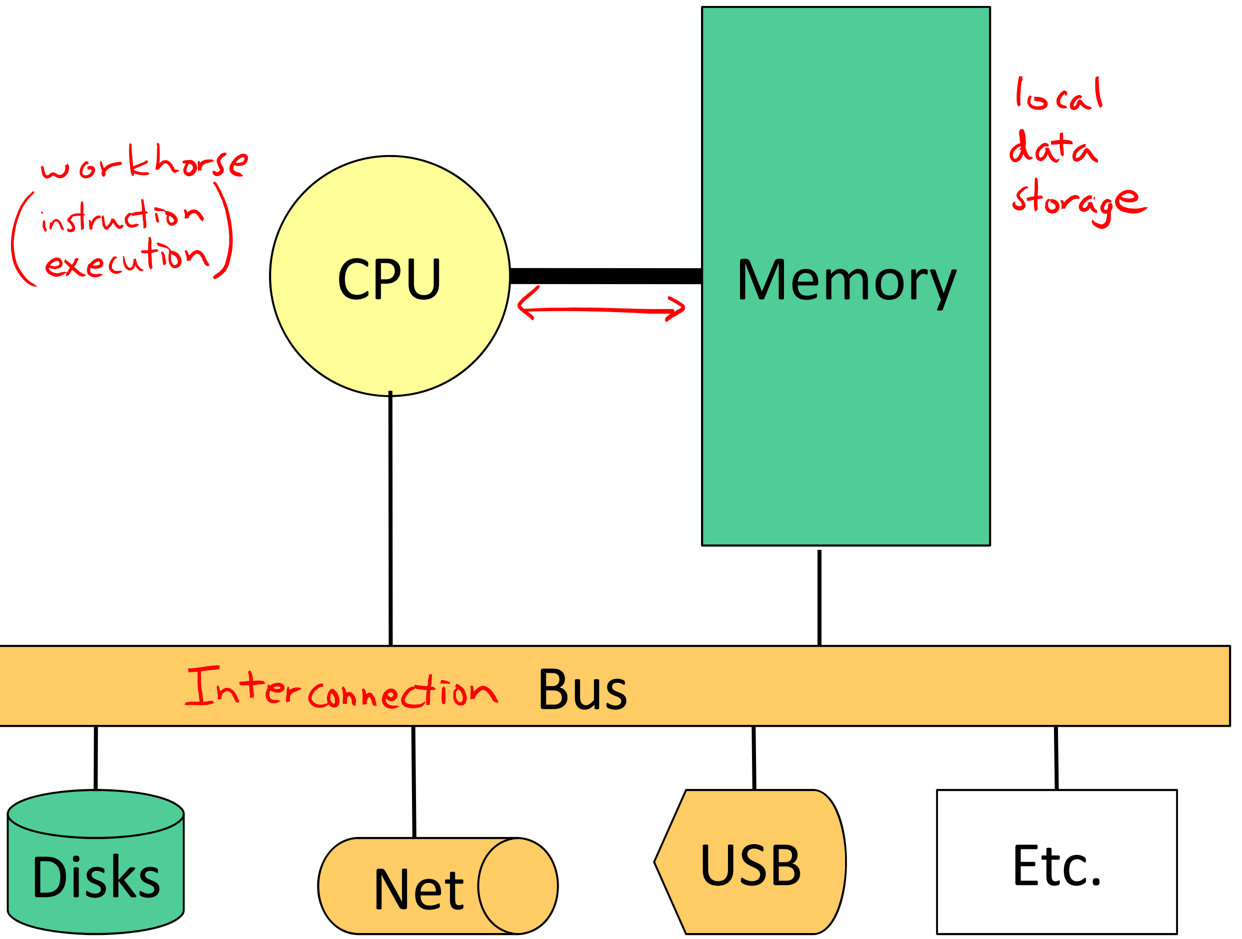
CPU
(empty slot)

I/O
controller

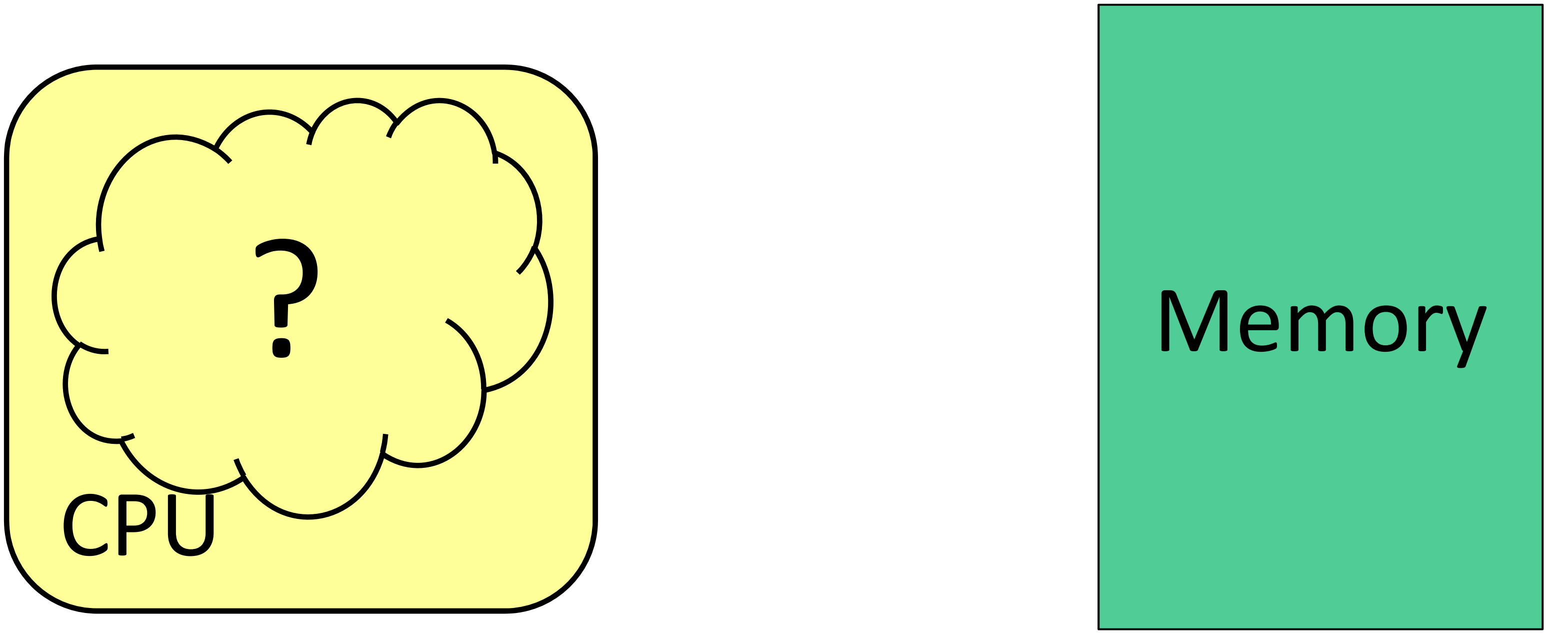
Memory

Storage connections

Hardware: Logical View



Hardware: 351 View (version 0)



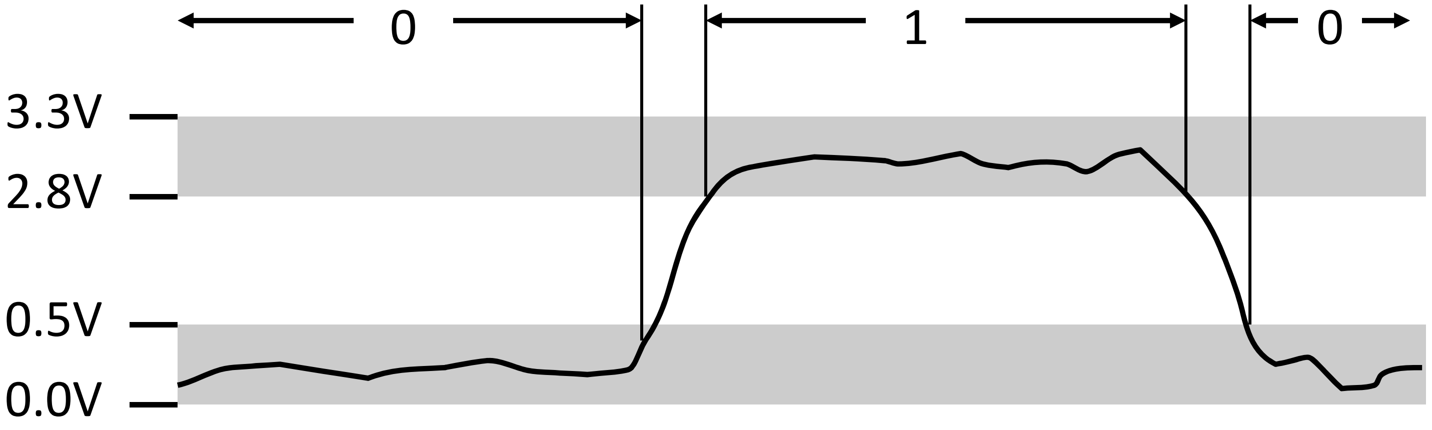
- The CPU executes instructions
- Memory stores data

How are data and instructions represented?

- Binary encoding!
  - Instructions *are* just data (and stored in Memory)

Aside: Why Base 2?

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

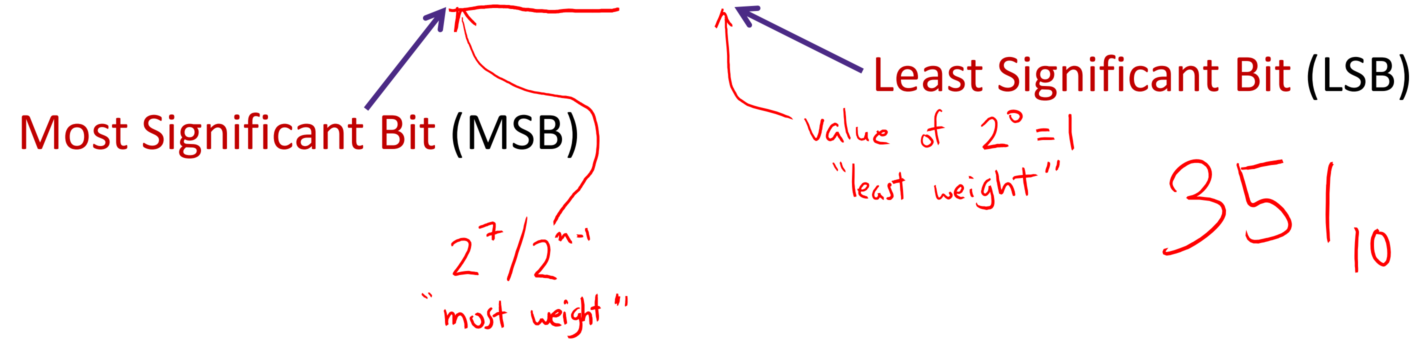


- Other bases possible, but not yet viable:
  - DNA data storage (base 4: A, C, G, T) is a hot topic
  - Quantum computing

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 0b$\overset{\text{padding}}{\underline{00000}}$100



Least Significant Bit (LSB)

value of $2^0 = 1$

“least weight”

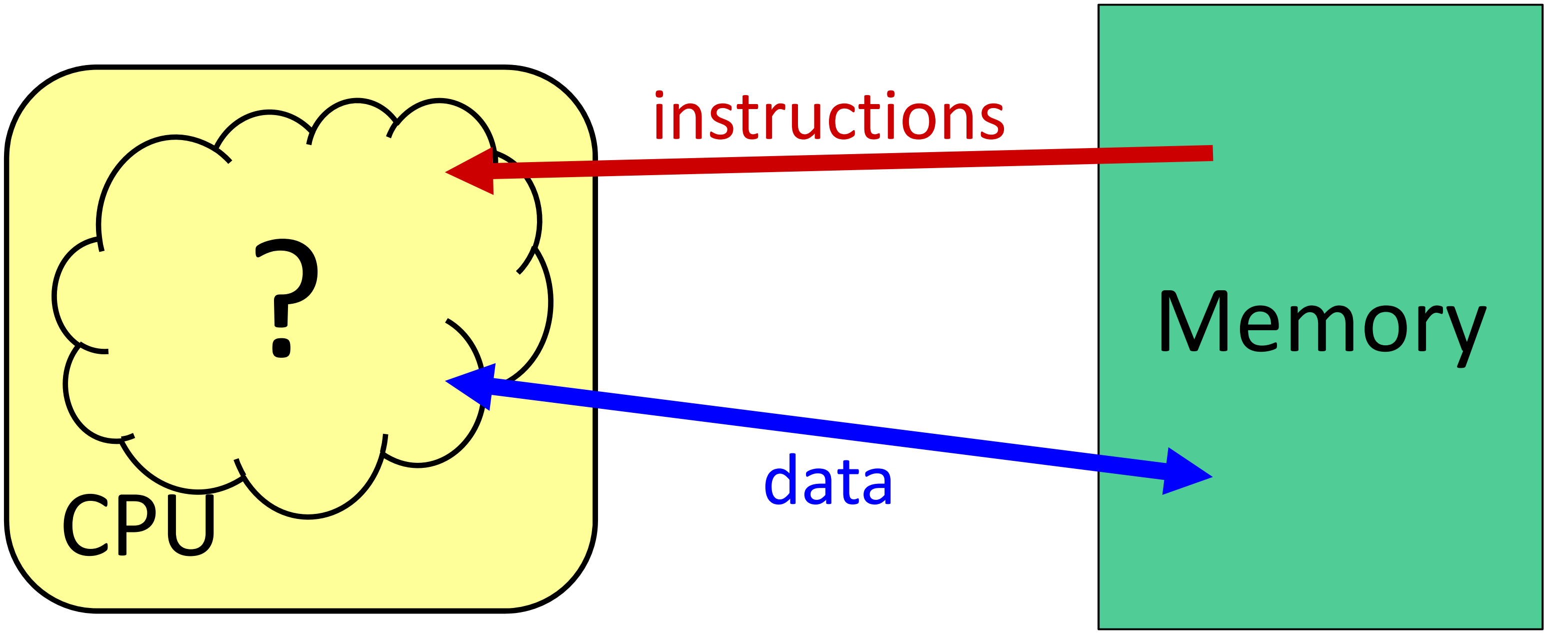
Most Significant Bit (MSB)

$2^7 / 2^{n-1}$

“most weight”

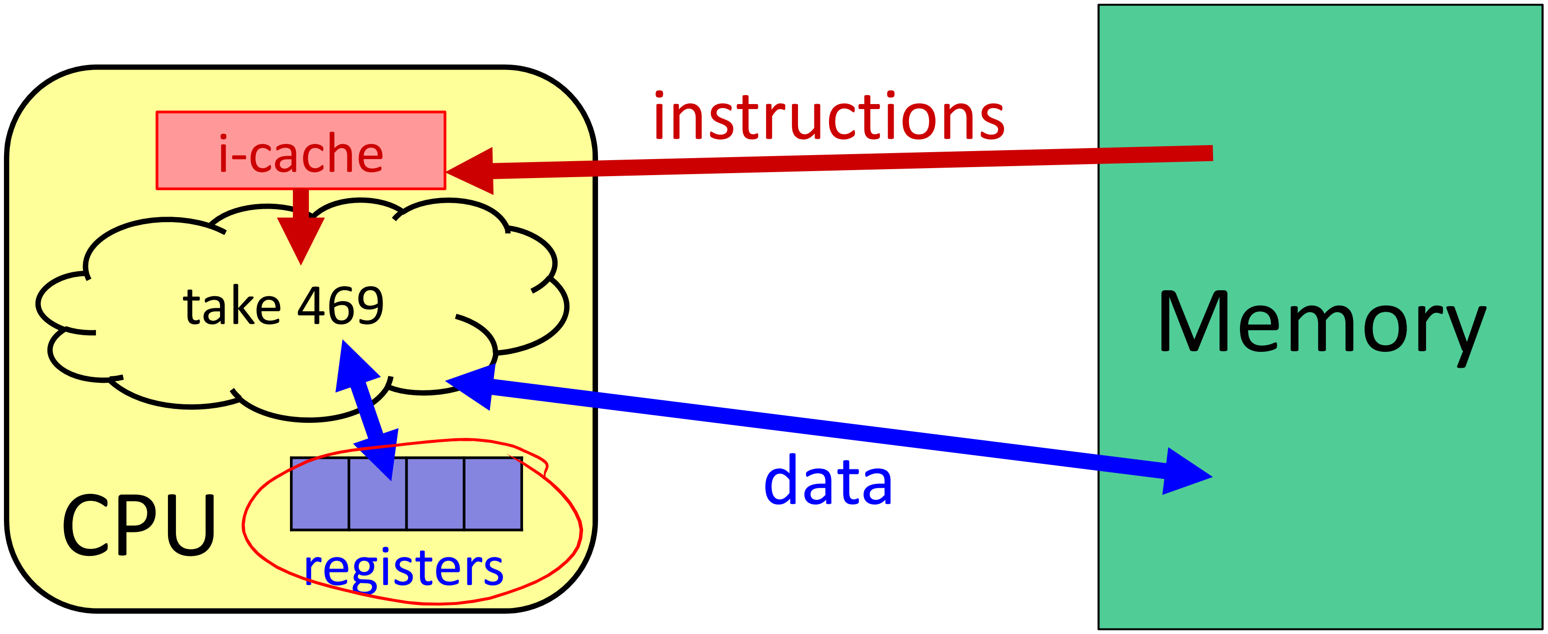
351_{10}

Hardware: 351 View (version 0)



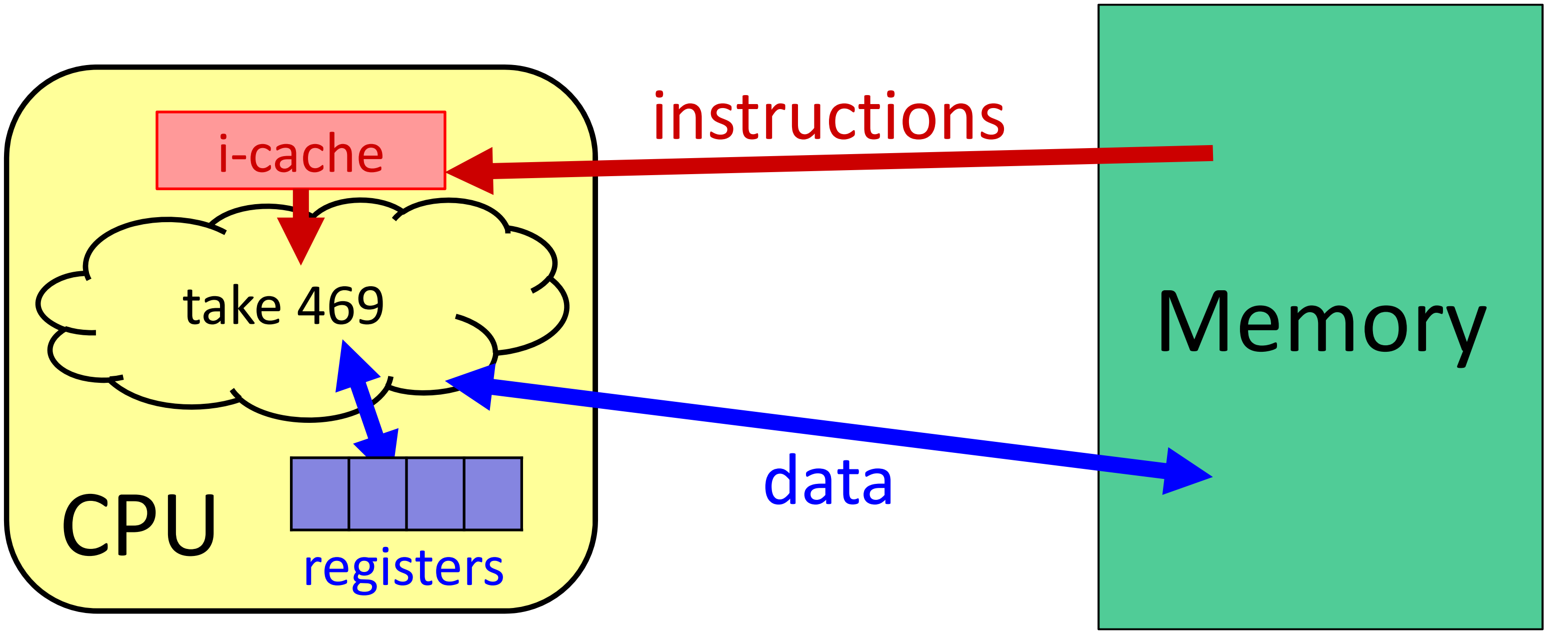
- To execute an instruction, the CPU must:
  1) Fetch the instruction
  2) (if applicable) Fetch data needed by the instruction
  3) Perform the specified computation
  4) (if applicable) Write the result back to memory

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)

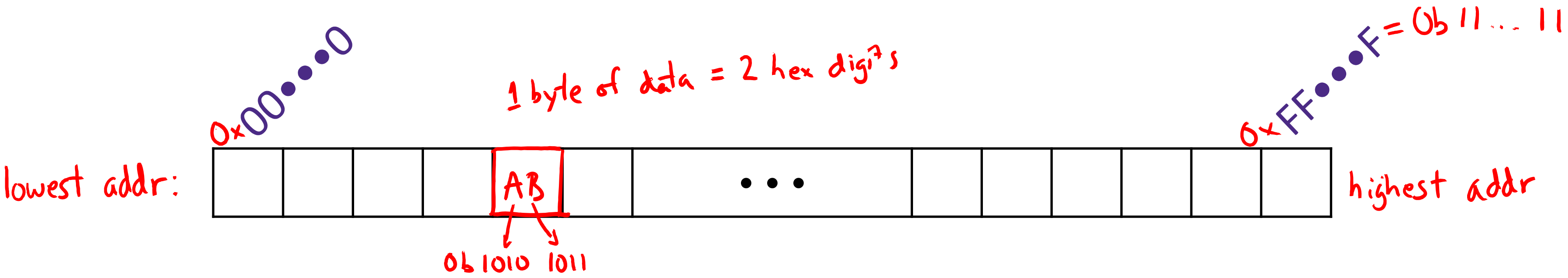
Hardware: 351 View (version 1)



- We will start by learning about Memory

How does a program find its data in memory?

An Address Refers to a Byte of Memory



- Conceptually, memory is a single, large array of bytes, not bits each with a unique *address* (index)
  - Each address is just a number represented in *fixed-length* binary

Hence there is a finite number of possible addresses available: 2^n with n being the number of bits of the address

- 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

$2^8 = 256$

$0, 1, \ldots 255$

- 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?
  - Vote at http://pollev.com/rea

A. 16 bits

[✓ B. 16 bytes]

C. 4 bits

D. 4 bytes

E. We're lost…

an address: 0b _ _ _ _

lowest:	0	0	0	0
highest:	1	1	1	1

4 bits ⟺ represent 2^4 things =16

Here, each address refers to 1 byte of data, so our addr space is $\boxed{\text{16 bytes}}$

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

"words" are just machine instructions stored in memory (32 or 64 bit each). When machine code runs, it uses the addresses of these "words" to find them which is why it is important we choose our address size/width to be compatible with the "word" length. See next slide.

- We have *chosen* to tie word size to address size/width
  - word size = address size = register size
  - word size = w bits → 2^w addresses → 2^w-byte address space

gap between adjacent addresses is the same gap as word size?

- Current x86 systems use **64-bit (8-byte) words**
  - Potential address space: $\mathbf{2^{64}}$ addresses
    2^{64} bytes ≈ $\mathbf{1.8 \times 10^{19}}$ **bytes**
    = 18 billion billion bytes = 18 EB (exabytes)

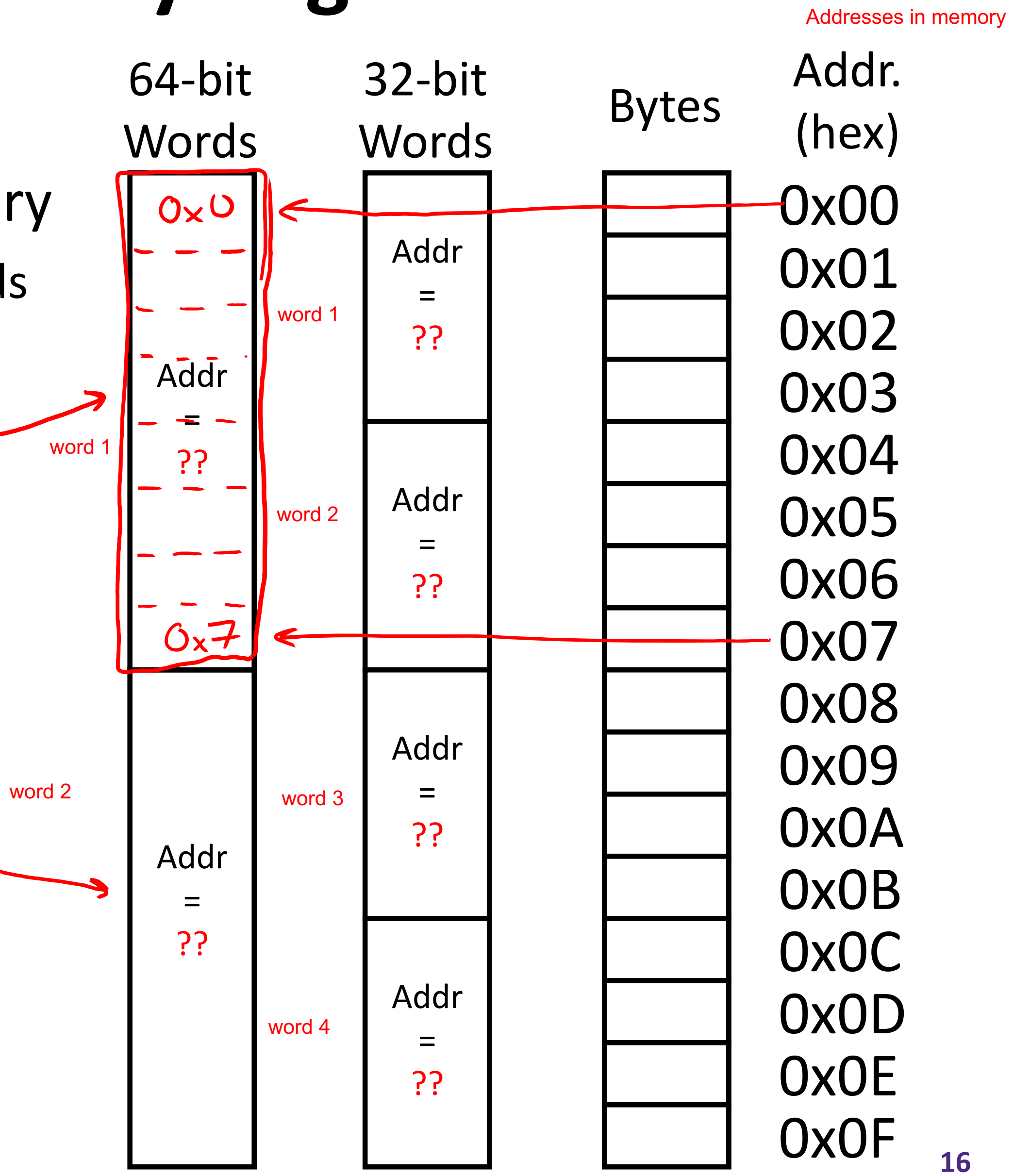
(where these addresses can point to; way more space than we could need)

  - Actual physical address space: **48 bits**

Again 64 bits is more than enough. In actuality we only need a small fraction of this address space.

Word-Oriented Memory Organization

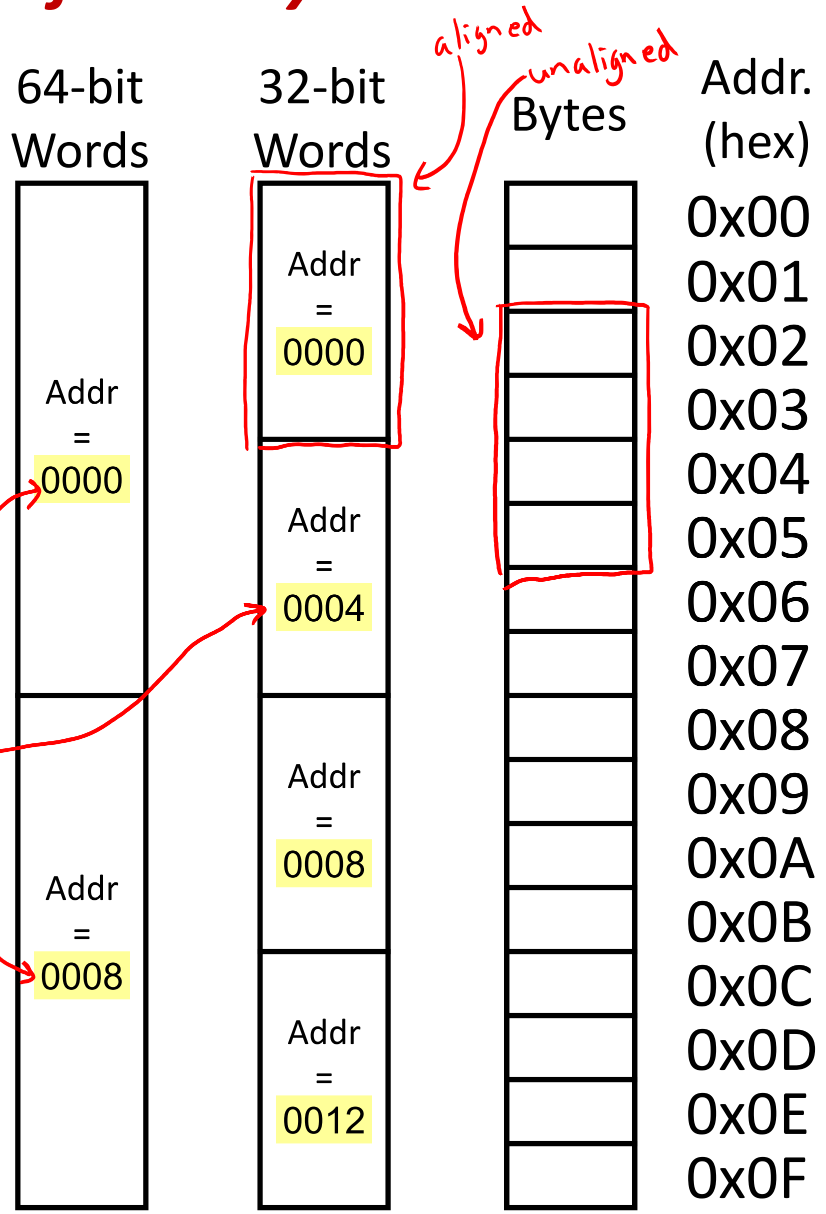
- Addresses still specify locations of *bytes* in memory
  - Addresses of successive words differ by word size (in bytes): *e.g.* 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, … 10?



Address of a Word = Address of First Byte in the Word

- 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***

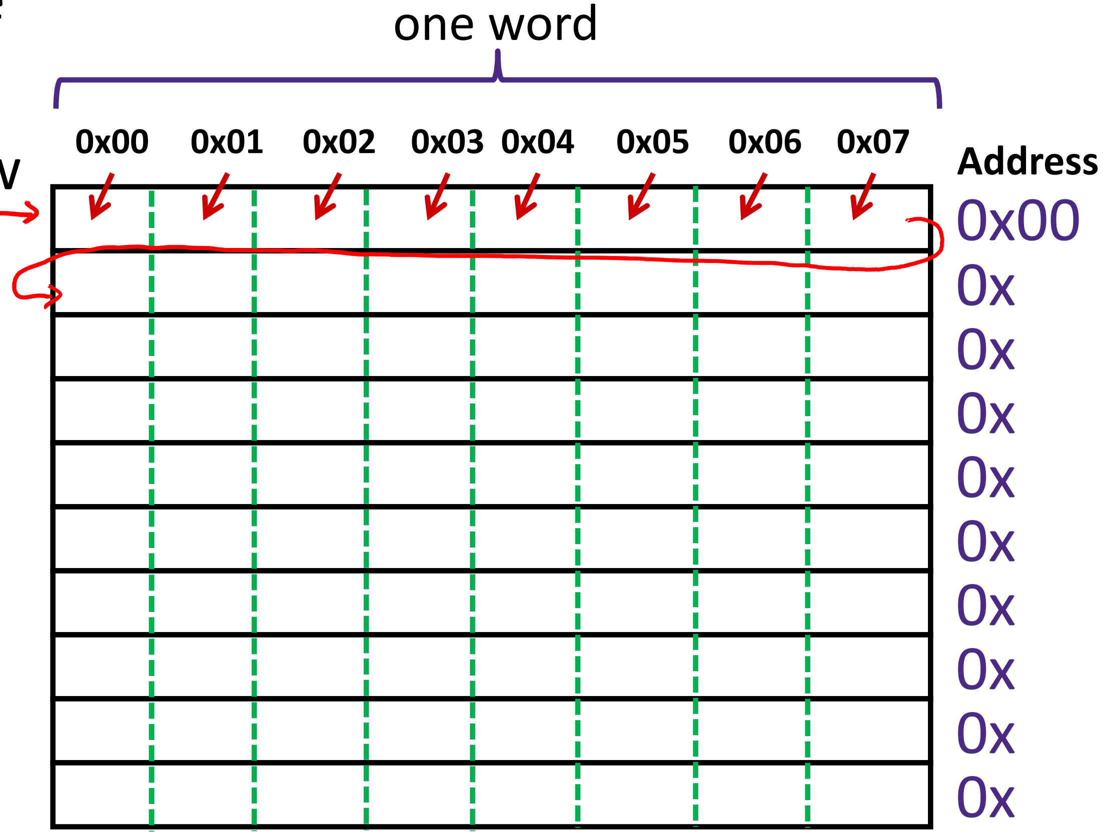
Alignment just means that the word addresses are just multiples of the word size



A Picture of Memory (64-bit word view)

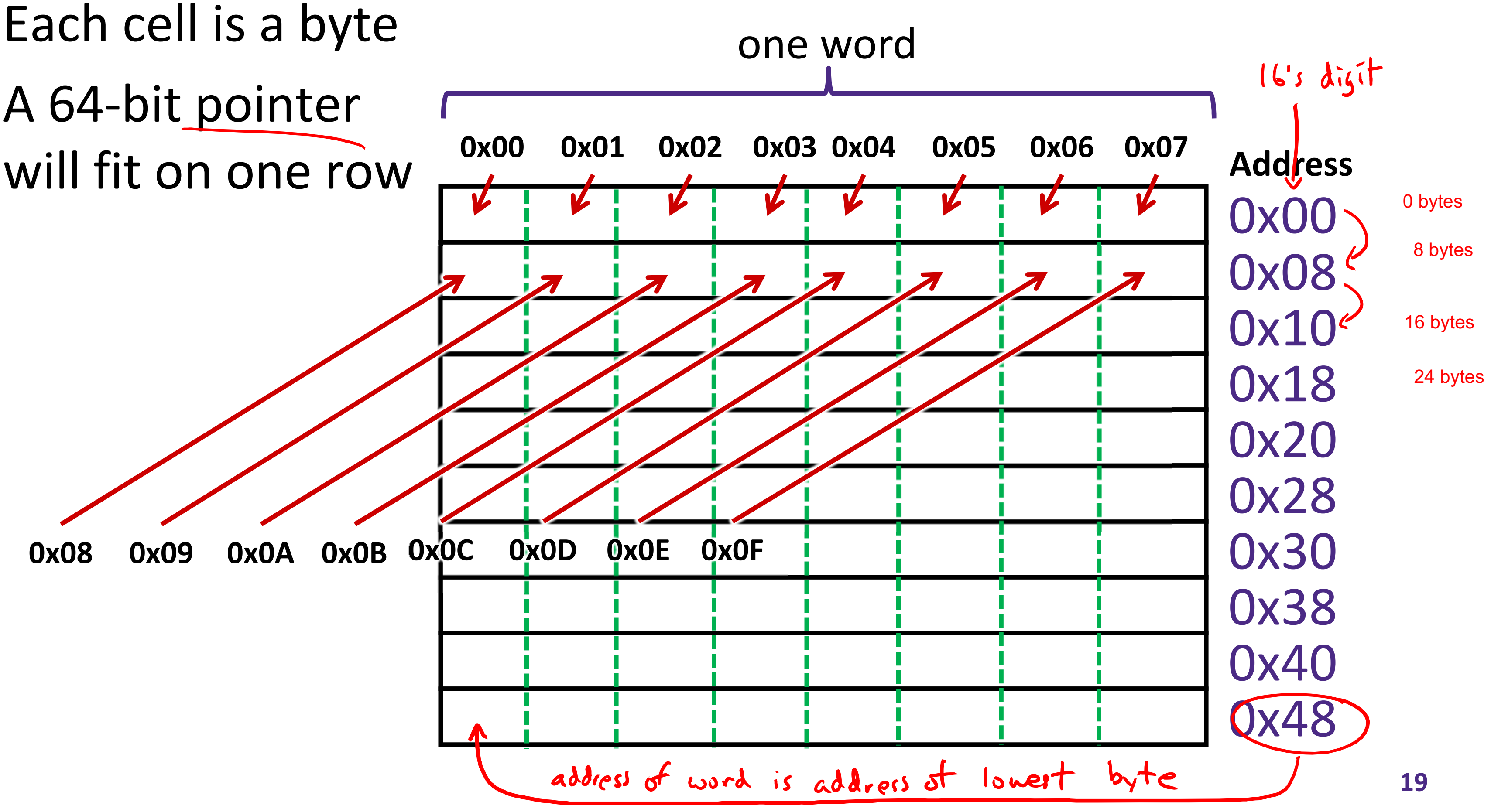
- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row

beginning of memory



A Picture of Memory (64-bit word 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

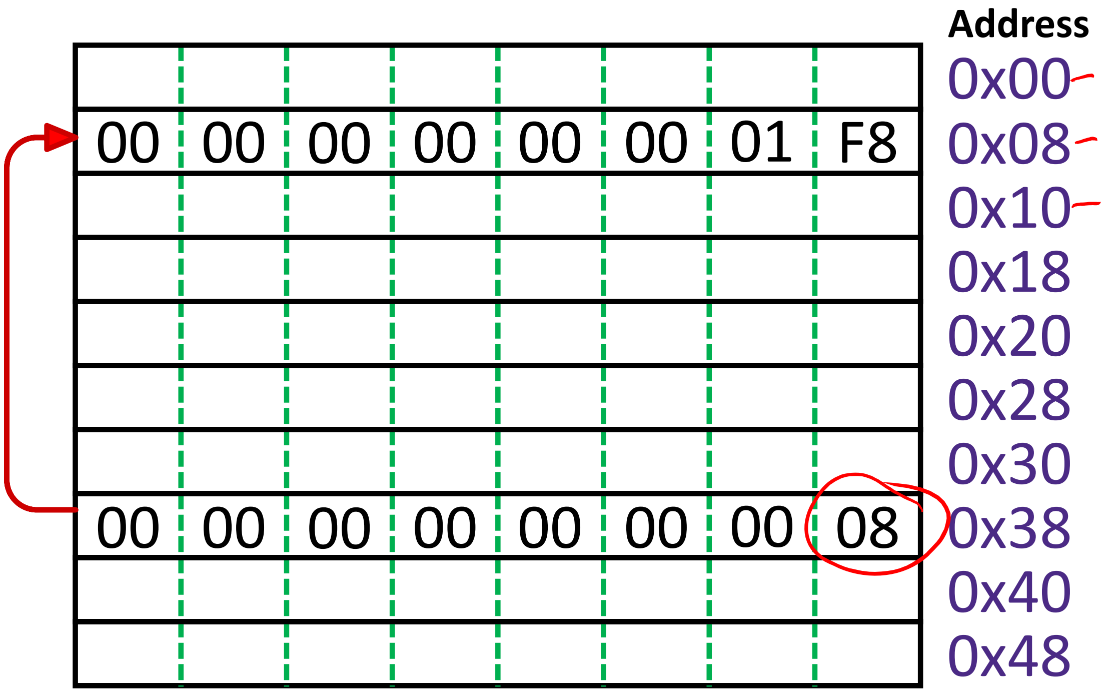


Addresses and Pointers

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

big-endian

- An *address* 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
00	00	00	00	00	00	01	F8	0x08
								0x10
								0x18
								0x20
								0x28
								0x30
00	00	00	00	00	00	00	08	0x38
								0x40
								0x48

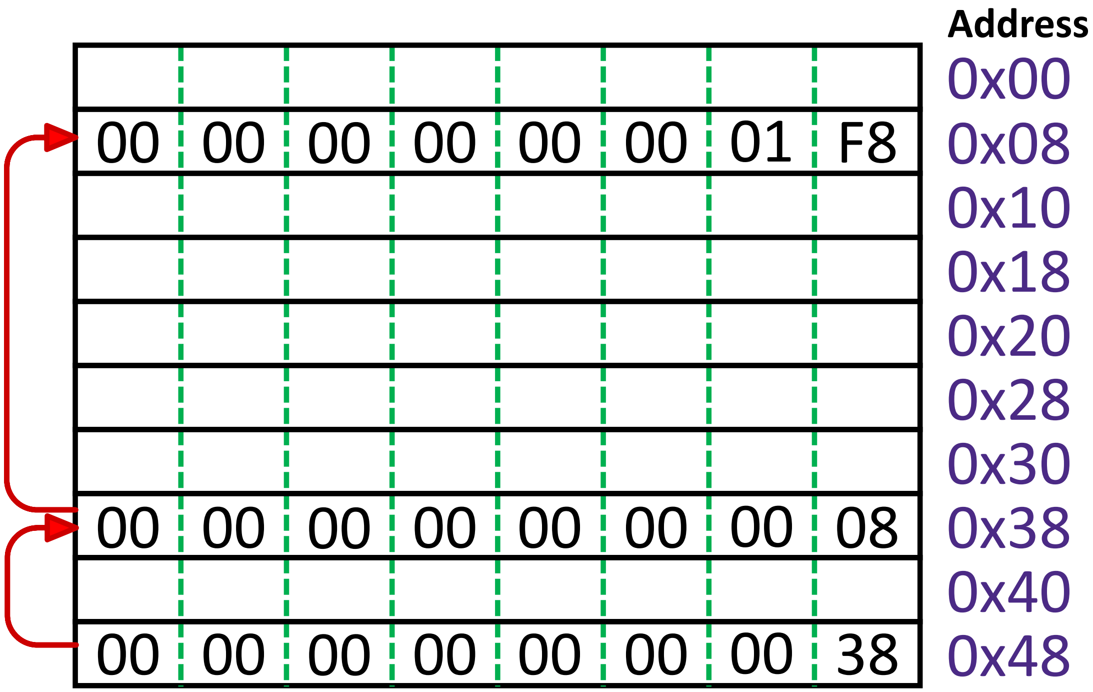
Addresses and Pointers

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

the hardware doesn't know!



								Address
								0x00
00	00	00	00	00	00	01	F8	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

There is nothing in the hex representation that shows what type the bytes are referring to in memory?

Data Representations

- Sizes of data types (in bytes)

32bit-word

64 bit word

Java Data Type	C Data Type	32-bit (old)	x86-64
boolean	bool	1	1 bytes
byte	char	1	1 byte
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

See slide 17 for a visual

  - 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` "least significant byte"

    most significant byte

    each byte will have a different address

- 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)

Big endian puts the most significant byte first (in the lowest address) while little endian is just the reverse: storing the least significant first

No reason for using one over the other, but important to know machines have different conventions (e.g. receiving a binary final from another machine)

Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64) Intel
  - 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		

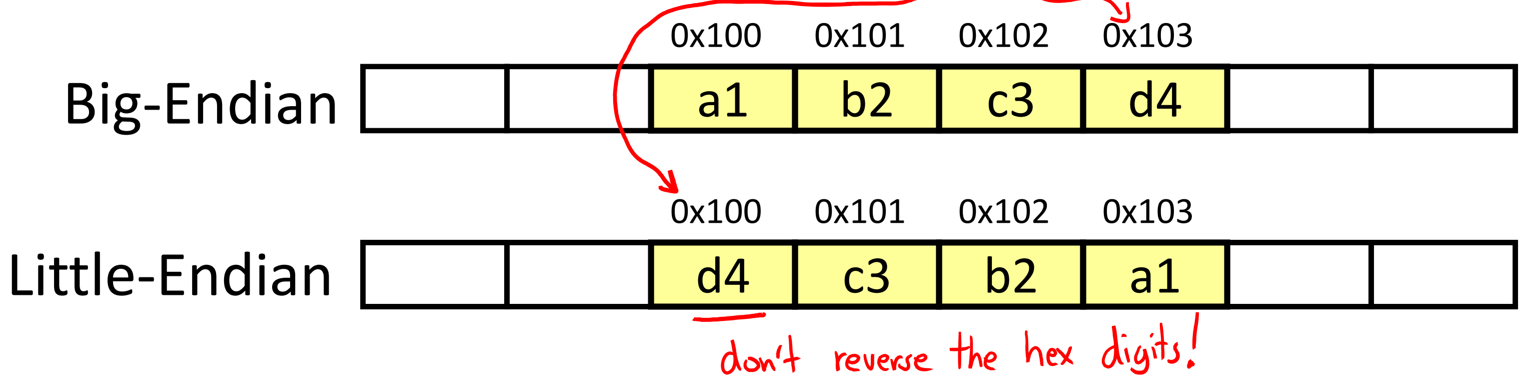
			0x100	0x101	0x102	0x103		
Little-Endian			d4	c3	b2	a1		

Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64) this class
  - 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

LS byte



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

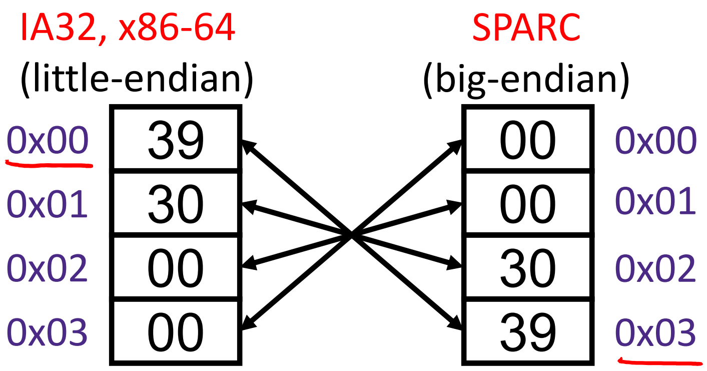
don't reverse the hex digits!

Byte Ordering Examples

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

4 bytes

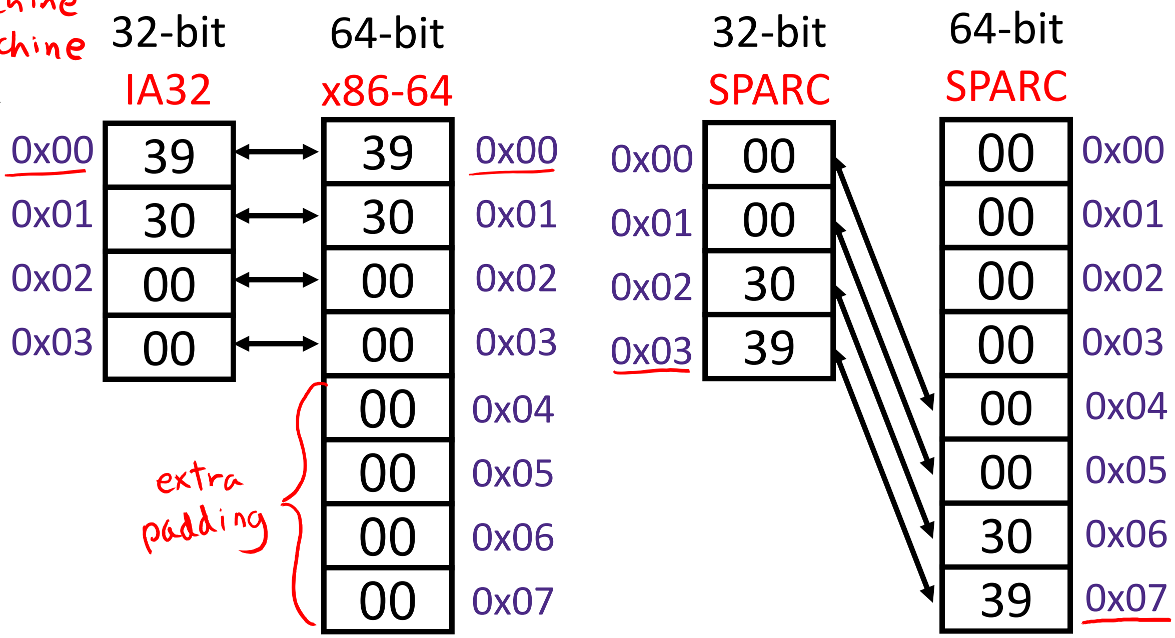
```
int x = 12345;
// or x = 0x3039;
```



4 bytes on 32-bit machine
8 bytes on 64-bit machine

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

(A `long int` is the size of a word)



Peer Instruction Question:

00 00 00 00

- We store the value 0x 01 02 03 04 as a ***word*** at address 0x100 in a big-endian, 64-bit machine 8 bytes per word
- What is the ***byte of data*** stored at address 0x104?
  - Vote at http://pollev.com/rea

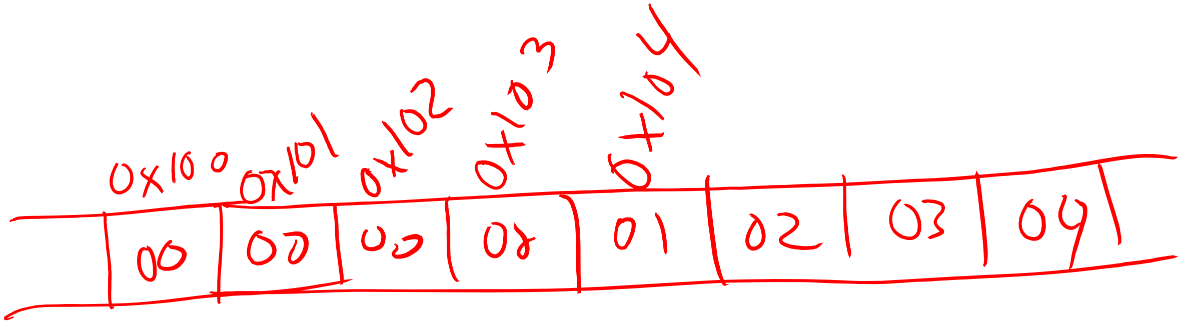
A. 0x04

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