Memory, Data, & Addressing I
CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Announcements, Reminders

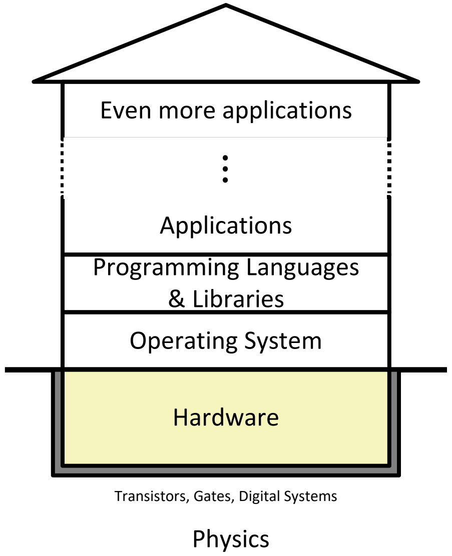
- Everything not a reading or lecture lesson due @ 11:59:00 PM
  - e.g. LC1 and RD2 were due today at 11:00 AM
  - Pre-Course Survey (Canvas) and HW0 due tonight
  - HW1 due Friday (3/29) by 11:59 PM
  - Lab 0 due Monday (4/01) by 11:59 PM
    - This lab is *exploratory* and looks more like a HW; the other labs will look a lot different!
- Labs: Partners allowed! One lab submission between both students.
- 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; make sure your post is in Question form!

EPA

- **Encourage class-wide learning!**
- Effort
  - Attending office hours, completing all assignments
  - Keeping up with Ed Discussion activity
- Participation
  - Making the class more interactive by asking questions in lecture, section, office hours, and on Ed Discussion
  - Lecture question voting
- Altruism
  - Helping others in section, office hours, and on Ed Discussion

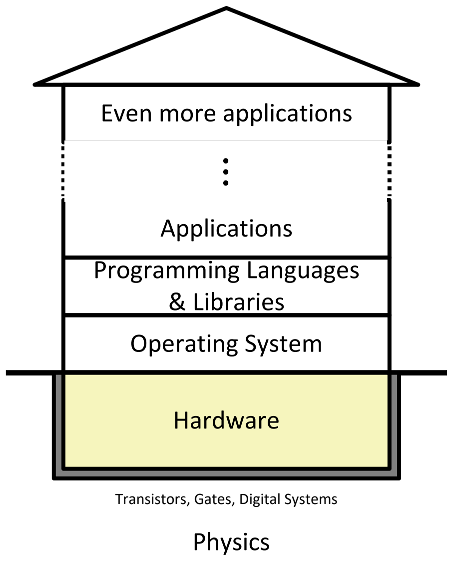
The Hardware/Software Interface

- Topic Group 1: **Data**
  - **Memory, Data**, Integers, Floating Point, Arrays, Structs
- Topic Group 2: **Programs**
  - x86-64 Assembly, Procedures, Stacks, Executables
- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation



The Hardware/Software Interface

- Topic Group 1: **Data**
  - **Memory, Data**, Integers, Floating Point, Arrays, Structs



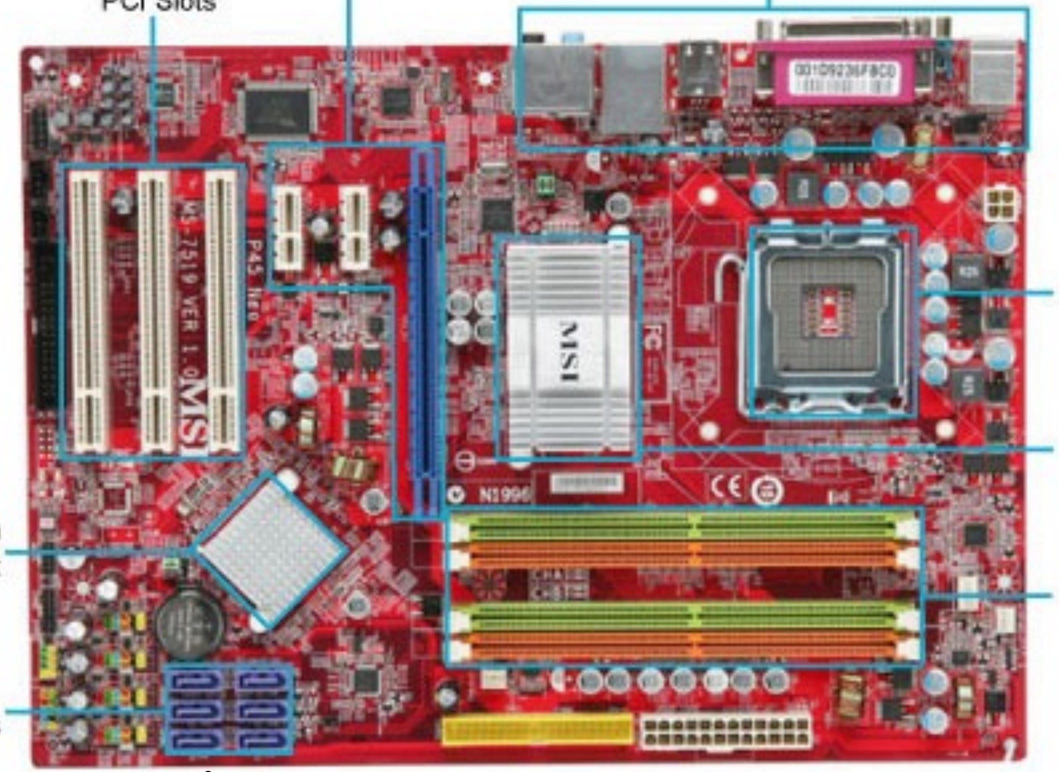
- Topic Question: How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings? → Helps understand thought process!

Hardware: Physical View

ew...

Bus connections

USB…



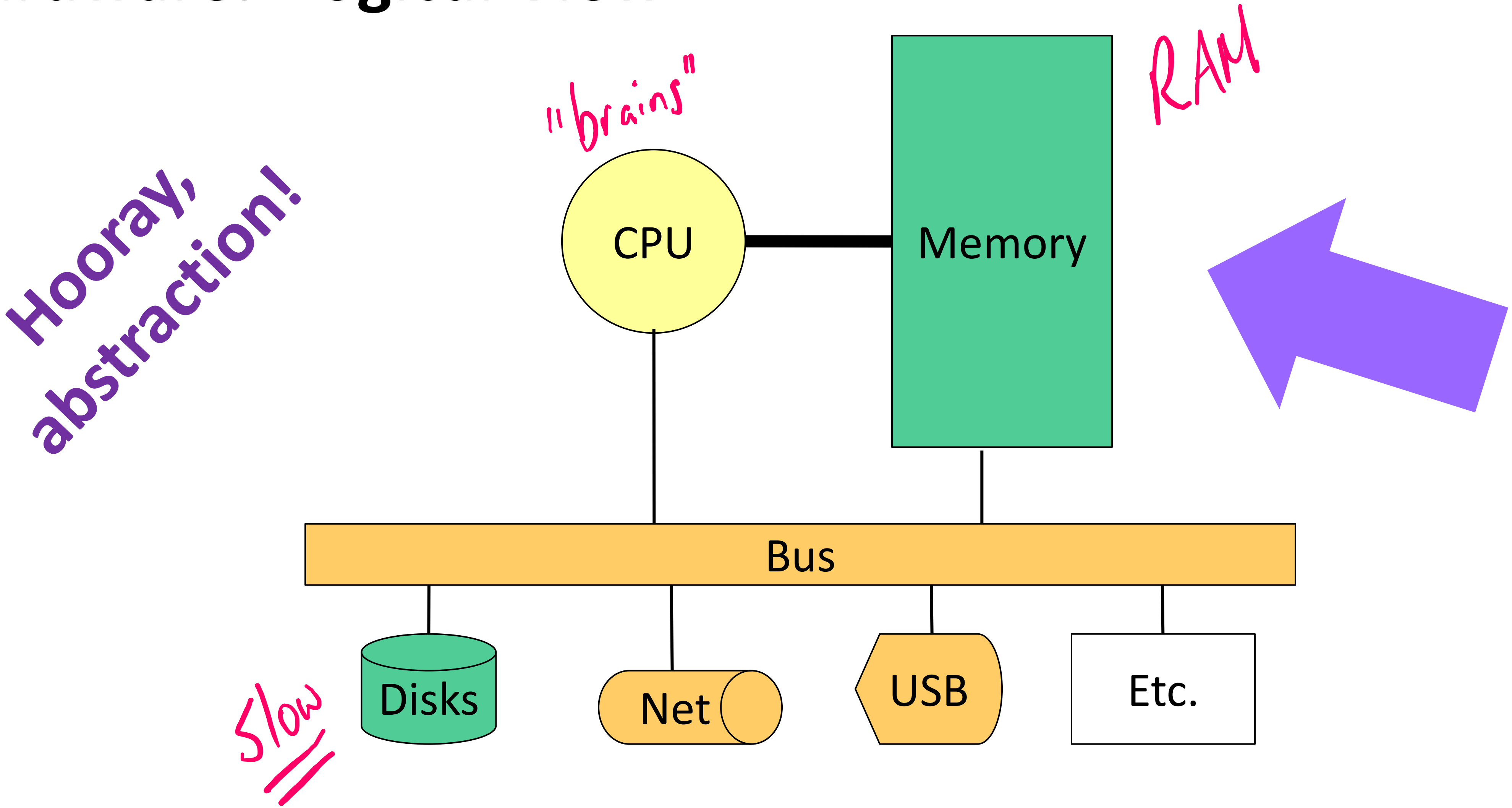
CPU
(empty slot)

I/O
controller

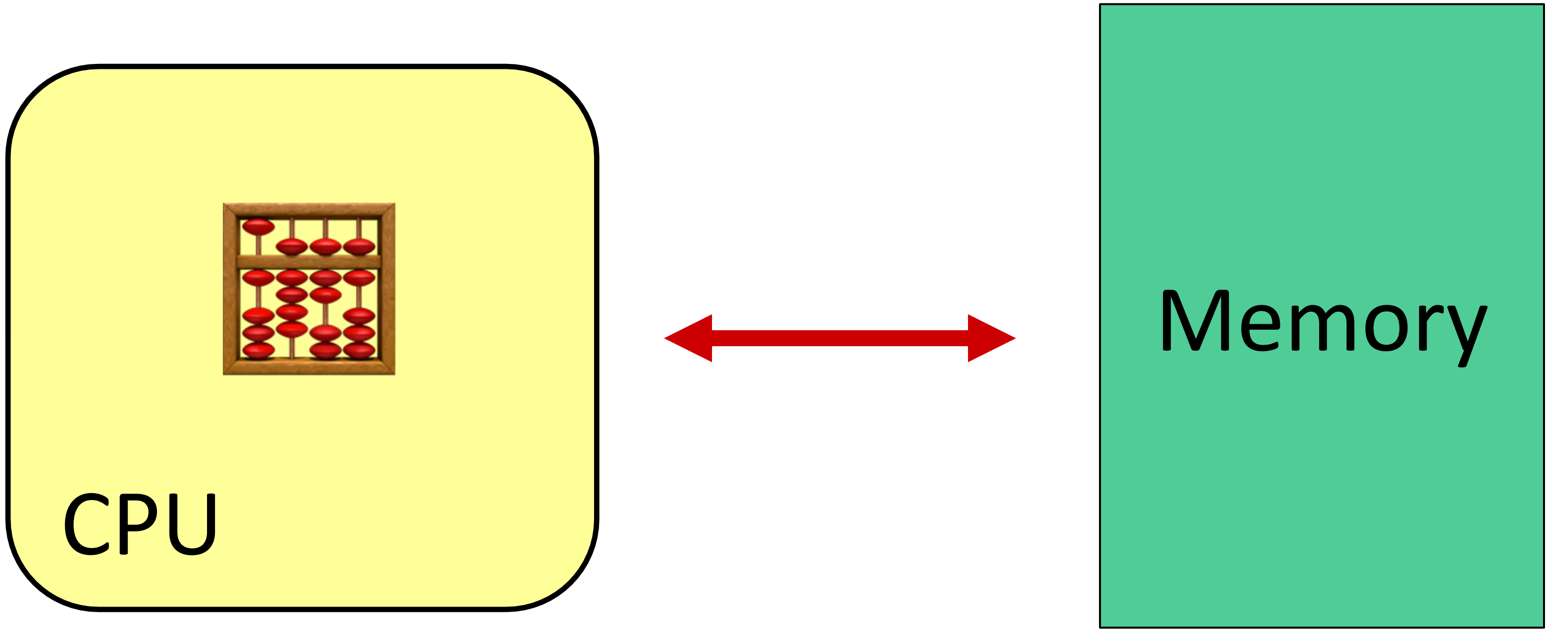
Memory

Storage connections

Hardware: Logical View



Hardware: 351 View (version 0)



- The CPU **executes** instructions
- Memory **stores** data

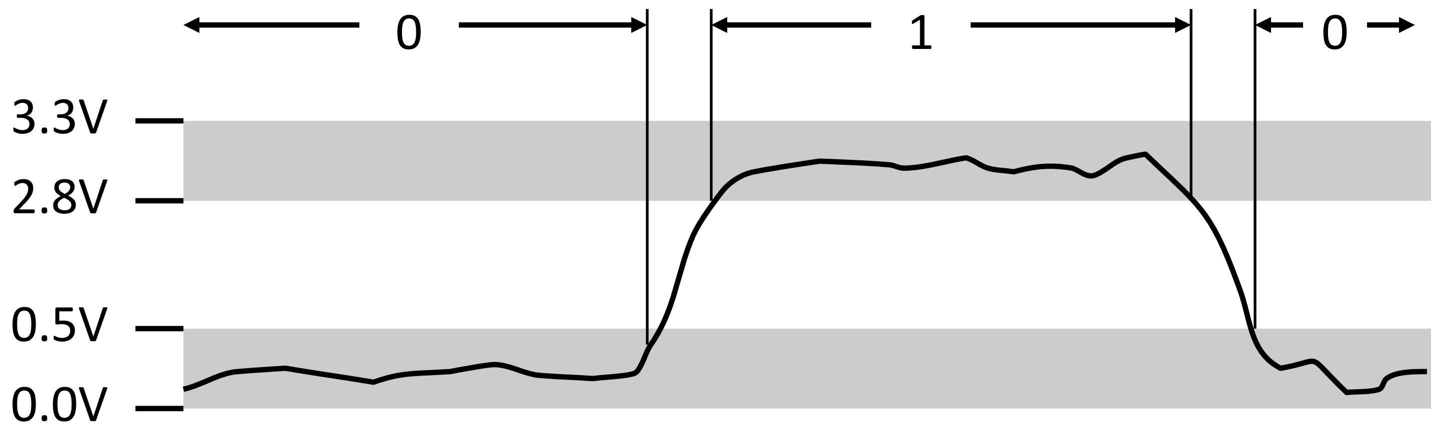
Q1: How are data and instructions represented?

Binary encoding!

Instructions *are* just data; also stored in memory!

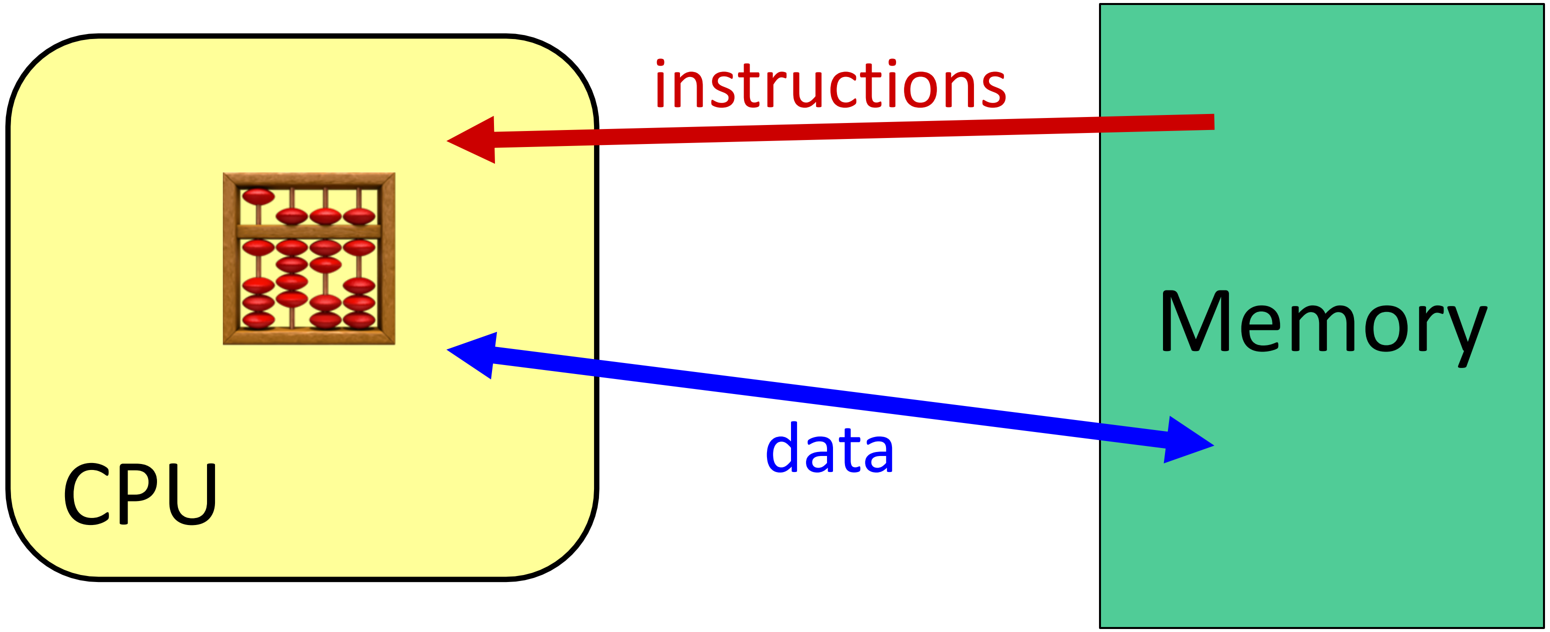
Aside: Why Base 2?

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



- Other bases possible, but not yet viable:
  - Ternary has existed (Setun, 1958)
  - DNA data storage (base 4: A, C, G, T) here at UW
  - Quantum computing

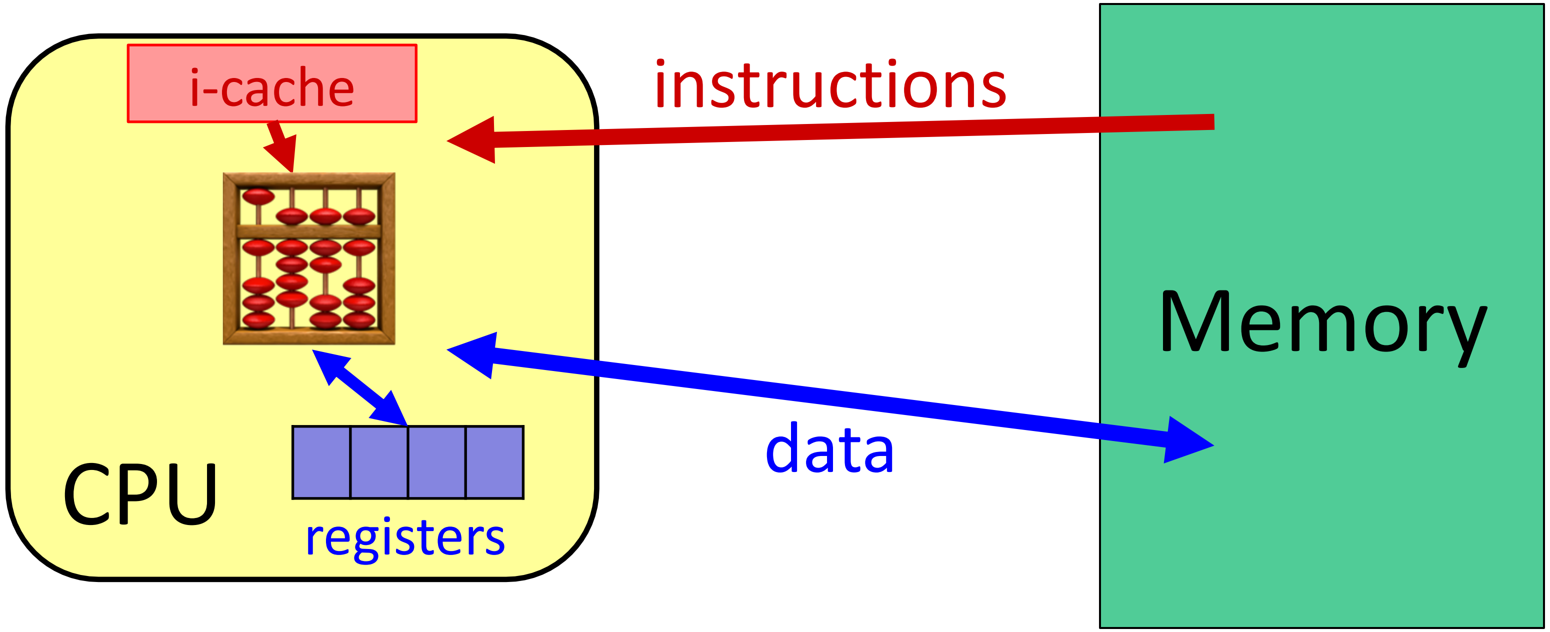
Hardware: 351 View (version 0)



- 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

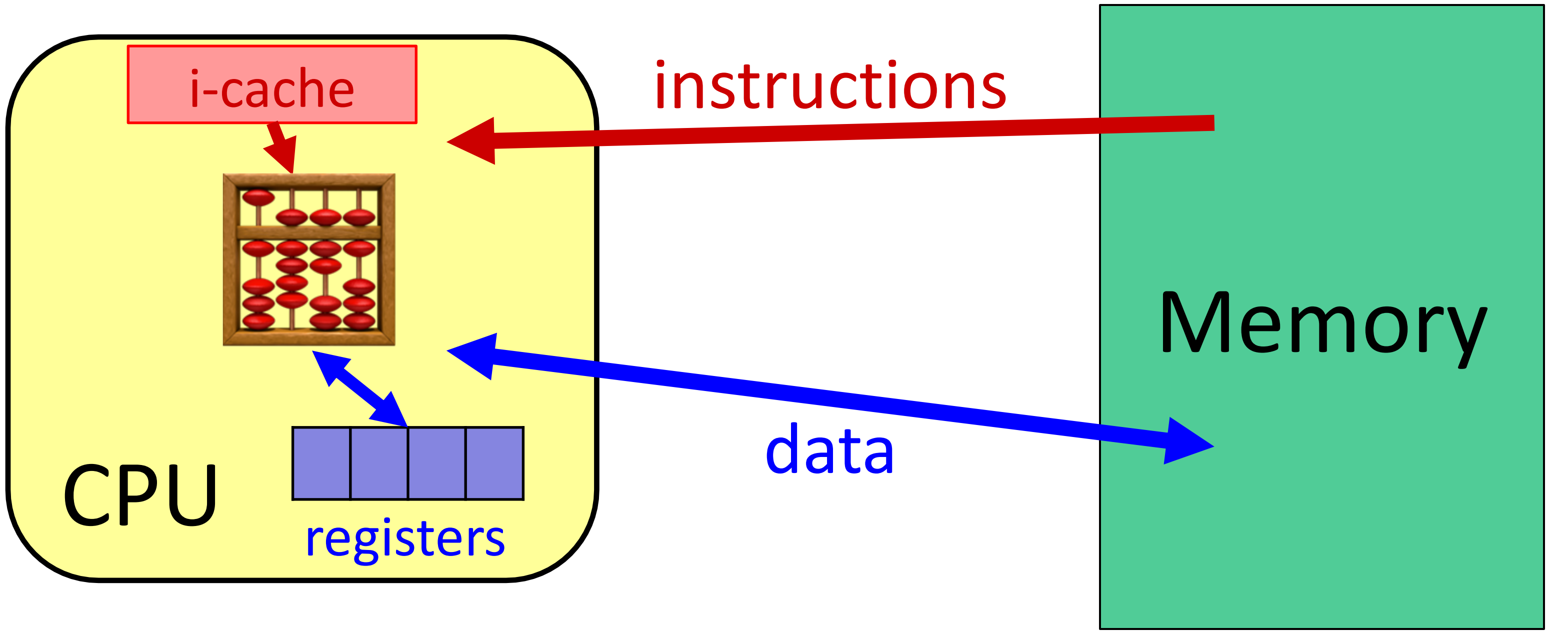
Hardware: 351 View (version 1)

This is extra (non-testable) material



- More CPU details:
  - Instructions are held temporarily in the instruction cache (i.e. Harvard Architecture)
  - Other data are held temporarily in registers ← Very fast, but few.
- Instruction fetching is hardware-controlled (My research! 👩🏻‍💻)
- Data movement is programmer-controlled (assembly)

Hardware: 351 View (version 1)



- We will start by learning about Memory

Q2: How does a program find its data in memory?

Addresses!

Can be stored in *pointers*

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?

Review Questions

- By looking at the bits stored in memory, I can tell what a particular 16 bytes is being used to represent.

  A. True B. False

  Encodings differ!

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

  A. True B. False

  Need both

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

  A. 0x3F B. 0xA0 C. 0xCA D. 0xD

  0b 1010 0000 0b 1100 1010 0b 0000 1101

Base Comparison

Slide from last time for reference!

- Why does all of this matter?
  - *Humans* think about numbers in **base 10**, but *computers* “think” about numbers in **base 2**
  - Binary encoding is what allows computers to do all of the amazing things that they do!
- You should have this table memorized by the end of the class
  - Might as well start now!

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Fixed-Length Binary (Review)

- 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

Most Significant Bit (MSB)   Least Significant Bit (LSB)

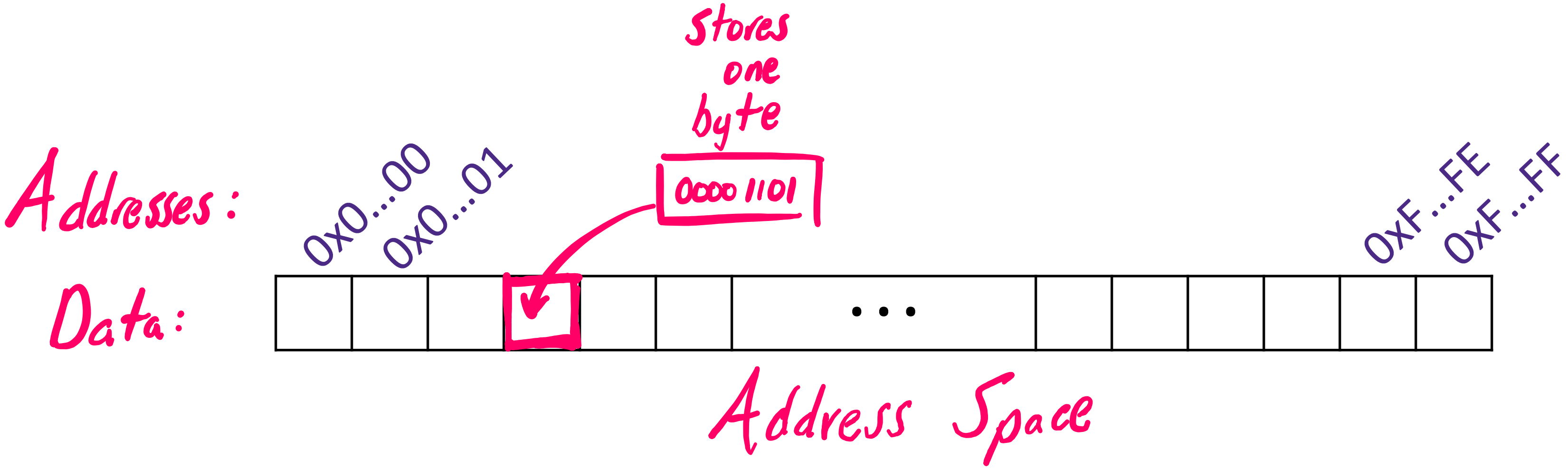
Bits and Bytes and Things (Review)

if $n=2$: 00, 01, 10, 11 — $2^n = 4$

- 1 byte = 8 bits
- n bits can represent up to 2^n things
  - Sometimes (oftentimes?) those "things" are bytes!
- If addresses are a-bits wide, how many distinct addresses are there?

∴ 2^a addresses

- What does each address refer to?



2^a bytes total

2^{a+3} bits total

Machine "Words" (Review)

- 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, no deviation. (Now, there exists much deviation...)
- 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 ≈ $\mathbf{1.8 \times 10^{19}}$ **bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

remember this when we do virtual memory...

Still quite big!

Data Representations

- Sizes of data types (in bytes)

Java Data Type	C Data Type	IA-32 (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

Why? Because pointers store addresses!

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

Discussion Question

- Over time, computers have grown in word size:

Word size	Instruction Set Architecture	First? Intel CPU	Year Introduced
8-bit	??? (Poor & Pyle)	Intel 8008	1972
16-bit	x86	Intel 8086	1978
32-bit	IA-32	Intel 386	1985
64-bit	IA-64	Itanium (Merced)	2001
64-bit	x86-64	Xeon (Nocona)	2004

  - What do you think were some of the *causes*, *advantages*, and *disadvantages* of this trend?

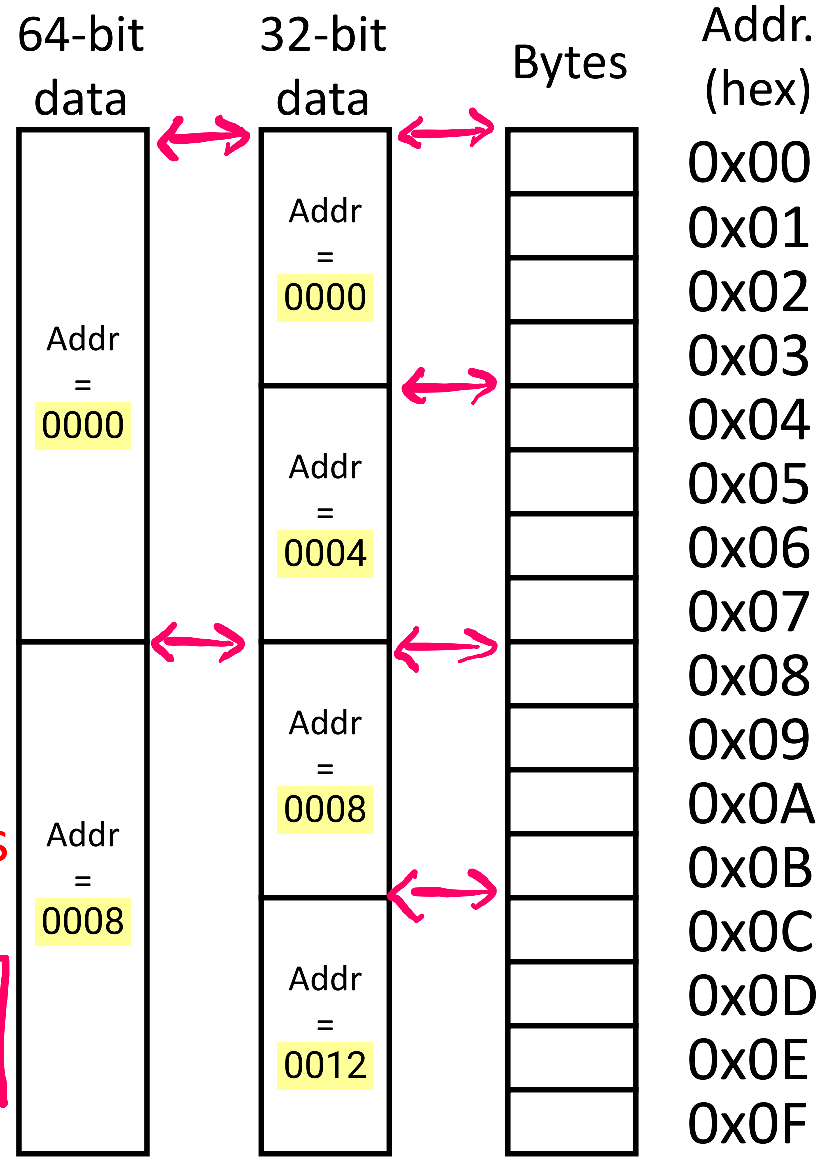
Causes: Dennard Scaling, Moore's Law, innovation

Advantages: Cheaper memory, more compute capabilities

Disadvantages: Environment :(, Sparse data, data overload

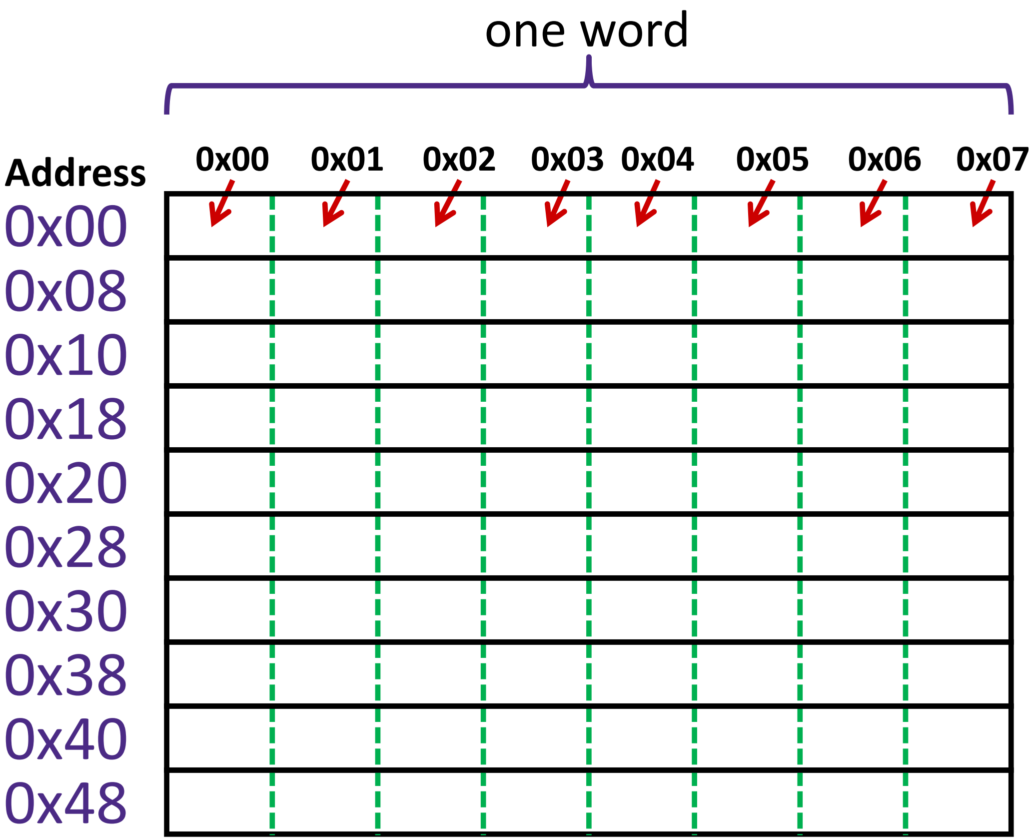
Address of Multibyte Data (Review)

- 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**



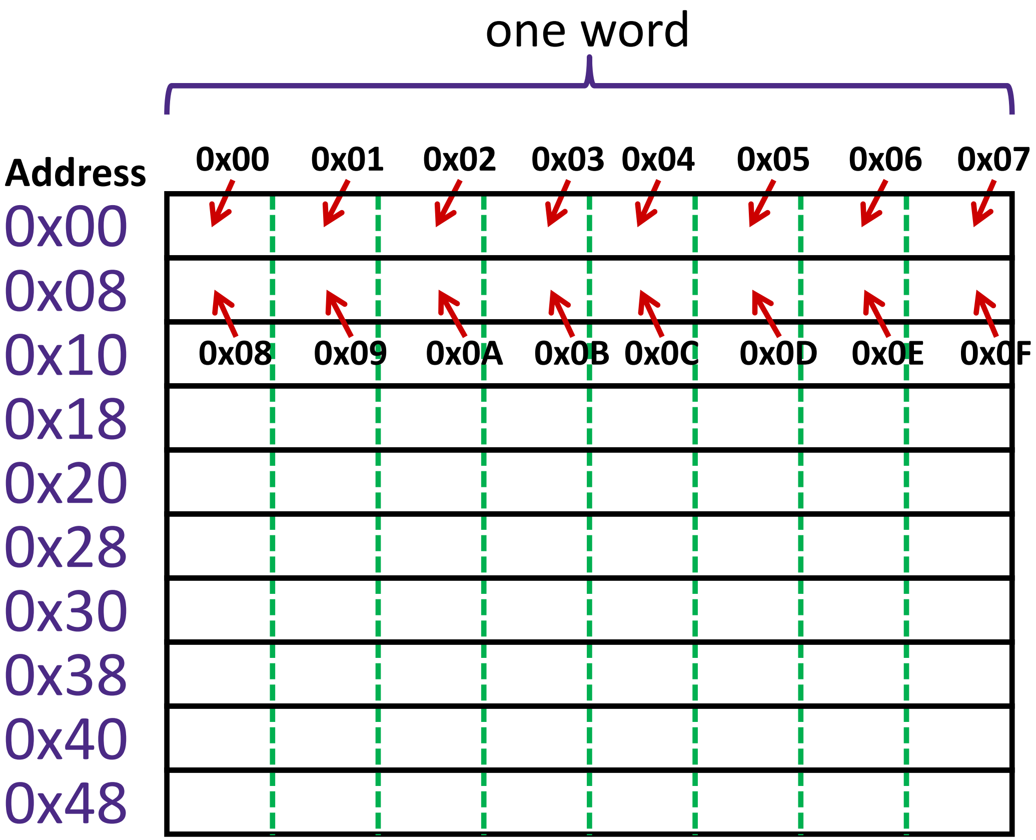
A Picture of Memory (64-bit view)

- 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



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



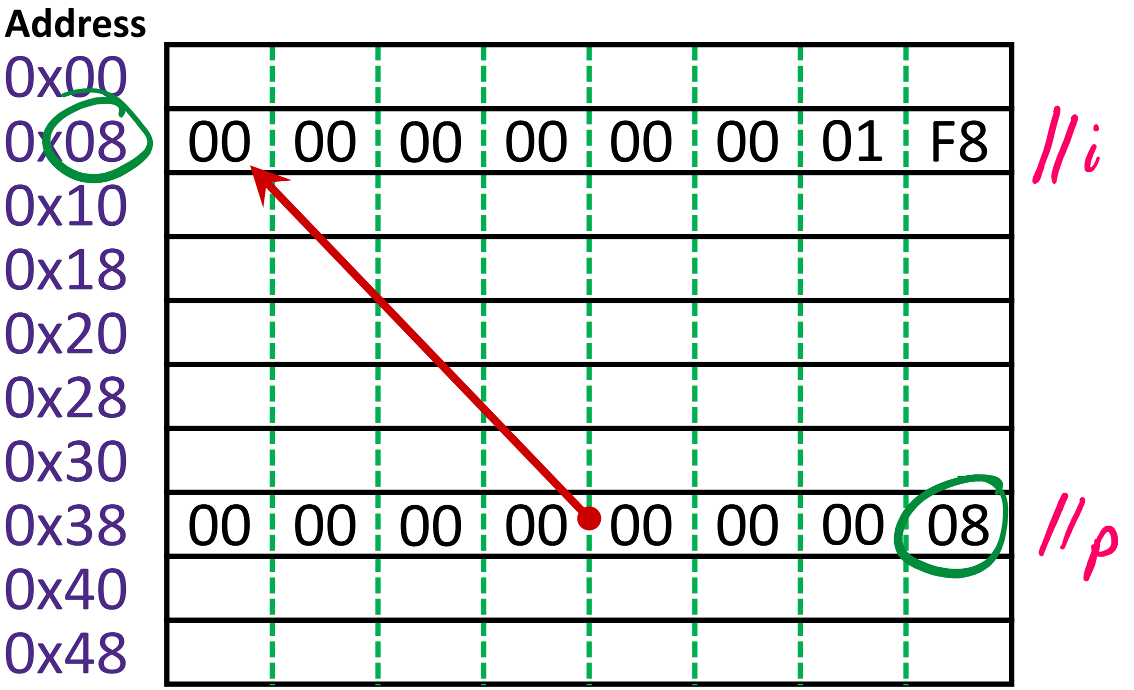
Addresses and Pointers

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 as a word at addr 0x08
  - $504_{10} = 1F8_{16}$
    = 0x 00 … 00 01 F8
- Pointer stored at 0x38 points to address 0x08

int i = 504;
int *p = &i;



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

```
int i = 504;
int *p = &i;
int **q = &p;
```

Address									
0x00									
0x08	00	00	00	00	00	00	01	F8	// i
0x10									
0x18									
0x20									
0x28									
0x30									
0x38	00	00	00	00	00	00	00	08	// p
0x40									
0x48	00	00	00	00	00	00	00	38	// q

64-bit example

Ad

```
[elba@attu1 ~]$ tail pointer_example.c
int main(int argc, char* argv[]) {

    int i = 504;    // i
    int *p = &i;    // p
    int **q = &p;   // q

    printf("i = %i\np = %p\nq = %p\n", i, p, q);

    return 0;
}
[elba@attu1 ~]$ ./pointer_example
i = 504
p = 0x7ffd048b97e4
q = 0x7ffd048b97d8
[elba@attu1 ~]$
```

Code equivalent of the previous Slide !!

Execution of Code above!

Byte Ordering (Review)

- 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`

- 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?
    - Historical: Based on *Gulliver's Travels*—tribes cut their eggs on different sides (big, little)
    - Language aside: how we write languages differs too!

زعفران

azafrán

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		

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

padding!

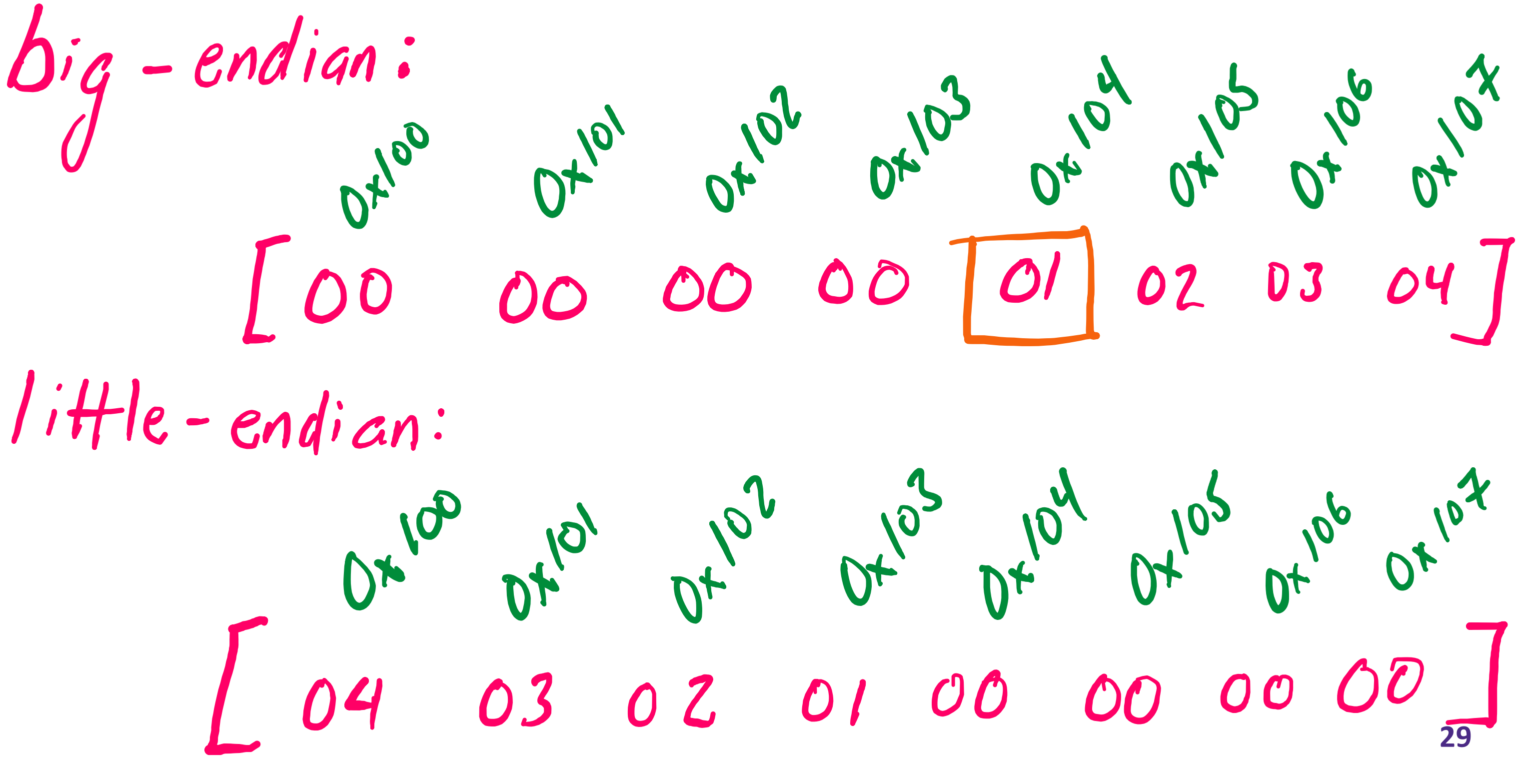
A. **0x04**

B. **0x40**

C. **0x01**

D. **0x10**

E. **We're lost…**



Endianness

btw…

- *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
- Pointers are data objects that hold addresses
  - Type of pointer determines size of thing being pointed at, which could be another pointer
- Endianness determines memory storage order for multi-byte data
  - Least significant byte in lowest (little-endian) or highest (big-endian) address of memory chunk