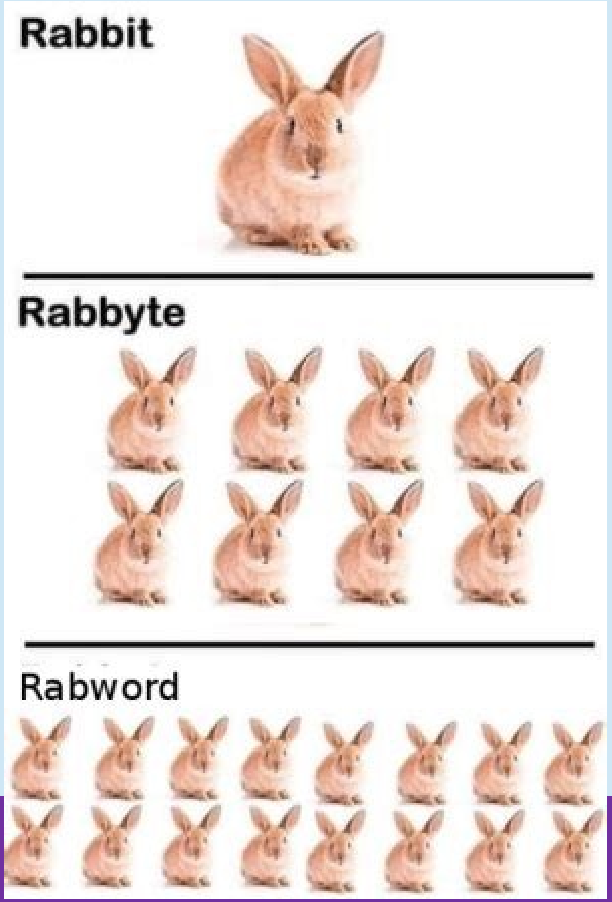
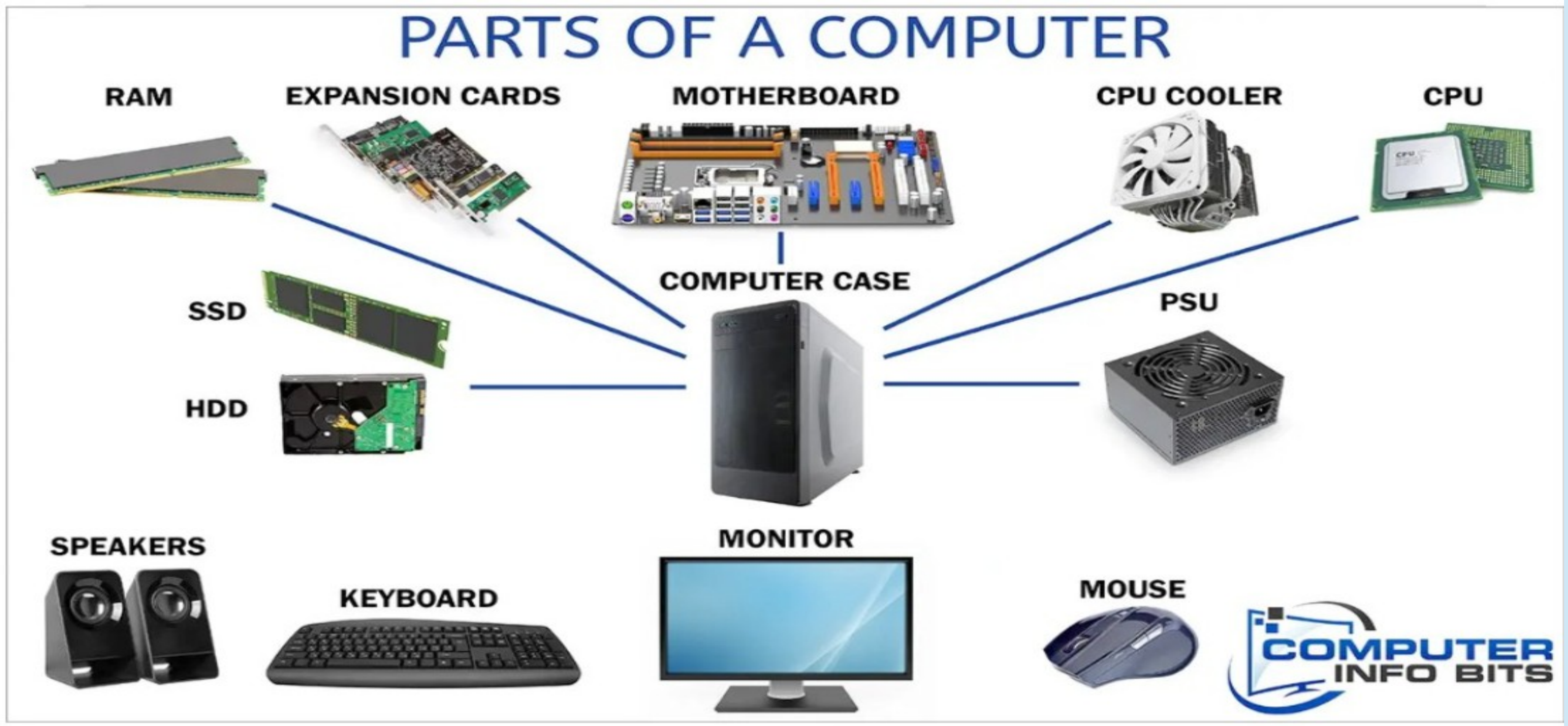
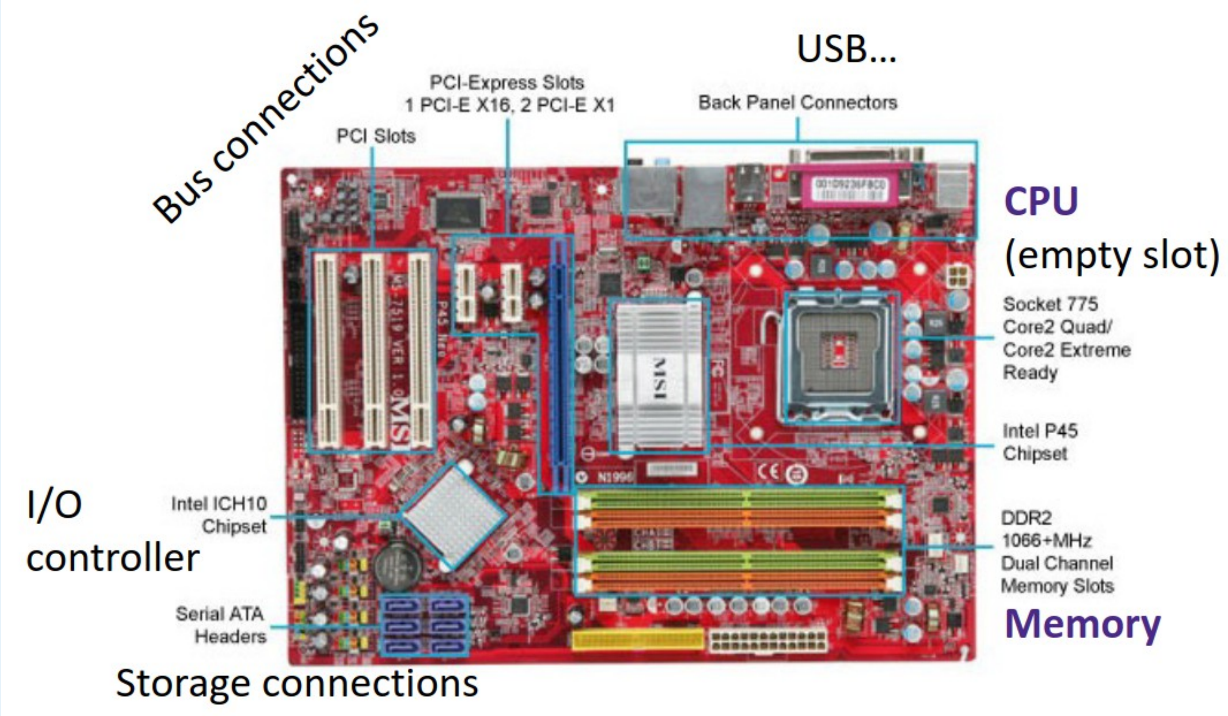
Memory, Data, & Addressing I



Computer Parts

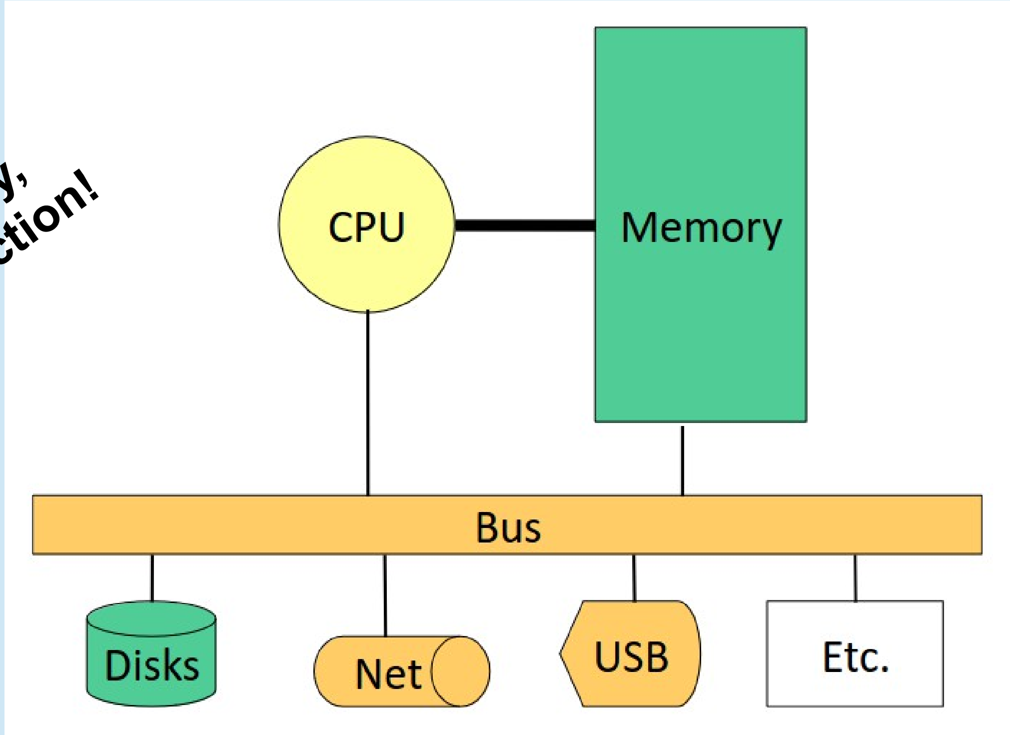


Hardware: Physical View

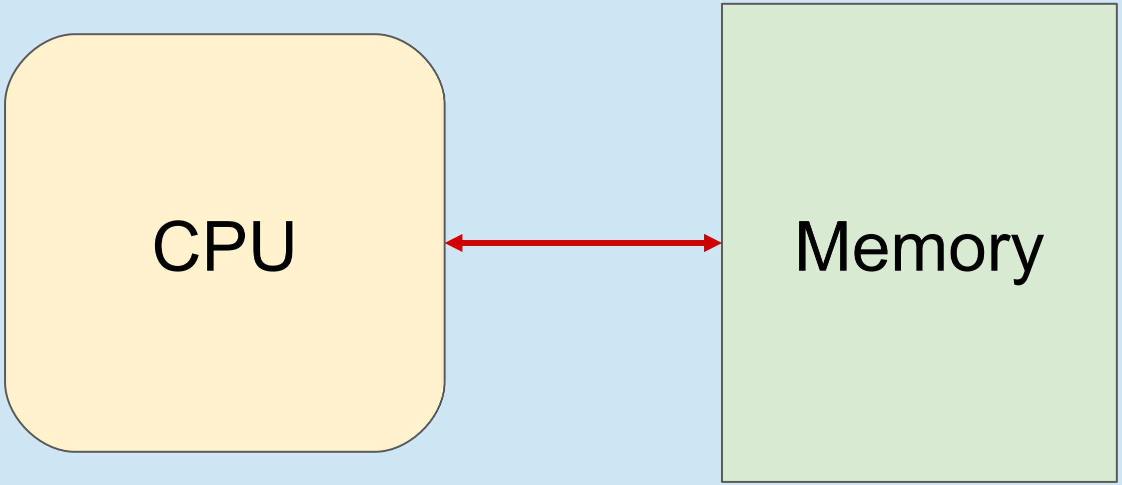


Hardware: Logical View

Hooray, abstraction!

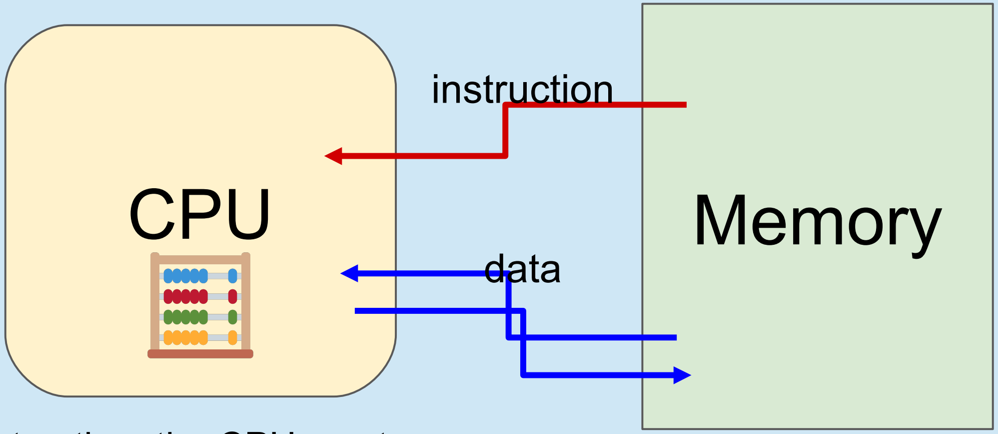


Hardware:



- The **CPU** executes instructions
- **Memory** is where data (including instructions) is stored
- How is data encoded? ***Binary encoding!***

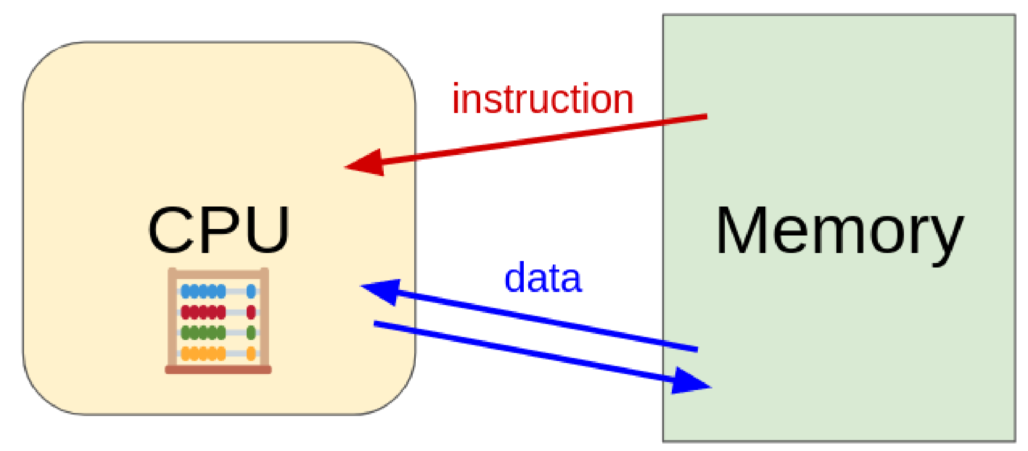
Hardware:



- To execute an instruction, the CPU must:

1. Fetch instruction
2. (if applicable) Fetch data needed for the instruction
3. Perform the computation
4. (if applicable) Write the result back to memory

Hardware: Today's focus



- We'll start by focusing on memory
    a. How does the CPU know where to find its data?
    b. How are common data types encoded?
    c. How can we use C to manipulate data?

Review Questions (pt 1)

1. By looking at the bits stored in memory, I can tell what a particular 4 bytes is used to represent.

   A) True **B)** False

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

   A) True **B)** False

Review Questions (pt 2)

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

A) 0x63 **B)** 0x90 **C)** 0xCA **D)** 0xD

4. Consider the following declared variables:

```
int x = 351;
```

How much space in memory does variable p take up (on a 64-bit machine)?

A) 1 byte **B)** 2 bytes **C)** 4 bytes **D)** 8 bytes

Fixed-length Binary

- Because storage is finite, 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

<u>Example</u>: the “eight-bit” (1-byte) representation of the number 4 is 0b00000100

Most significant bit (MSB)

Least significant bit (LSB)

Bits and Bytes and Things

- **Useful Fact:** *n* bits can represent up to 2^n things
  - If we want to represent *x* things, we need *n* bits, where $2^n >= x$

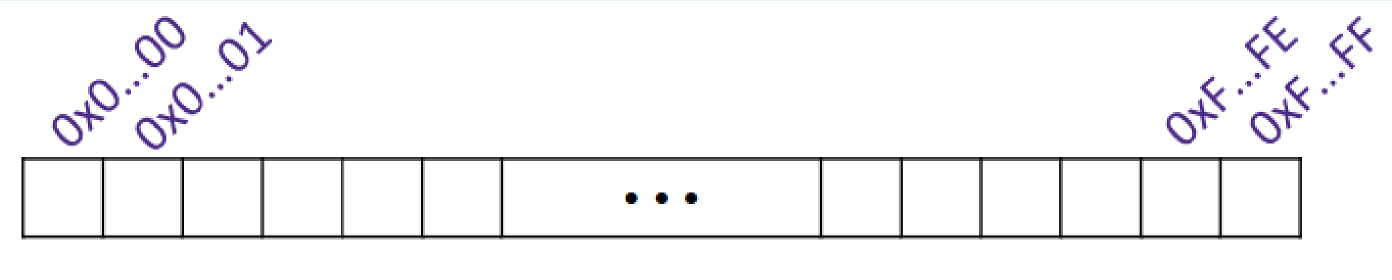
    <u>Example</u>: how many bits would we need to represent the letters a-z?

    26 letters, so we need 5 bits ($2^5 = 32 > 26$)

- Sometimes (oftentimes?) the “things” we are representing are ***bytes***!

Addresses in Memory

- You can think of memory as a single, large array of bytes, each with a unique **address** (index)
  - Addresses are fixed-width
  - Amount of addressable memory = **address space**
- If addresses are *a* bits long, how many addresses are there?
  - So how big is the address space?



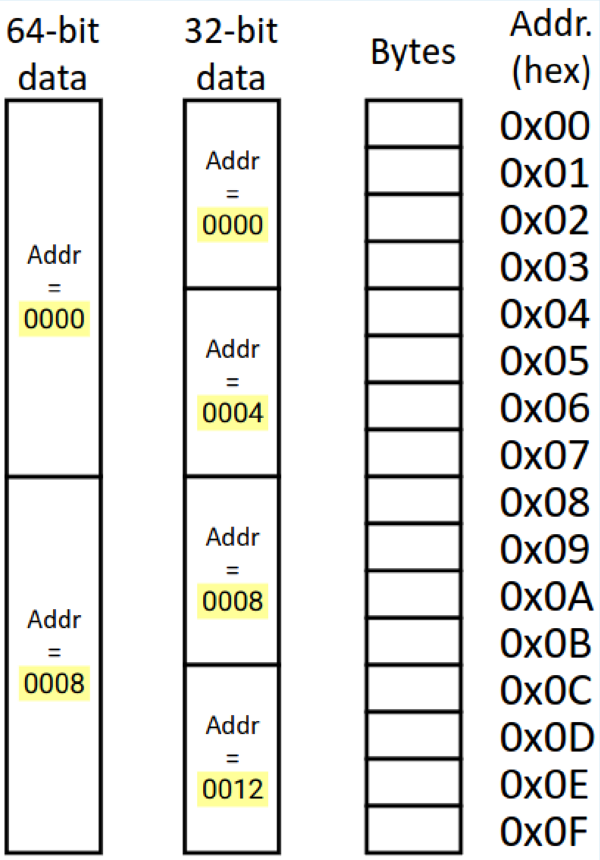
Machine “Words”

- <u>word size = address size</u>

- Most modern systems use **64-bit (8-byte) words**
    - Address space = 2 addresses, each represents a byte of data
    - 2^{64} bytes = 1.8*1019 bytes = 18 EB (exabytes)
        - (Side note: your computer does not actually have this much memory! We’ll talk about this more later)

Address of Multibyte Data

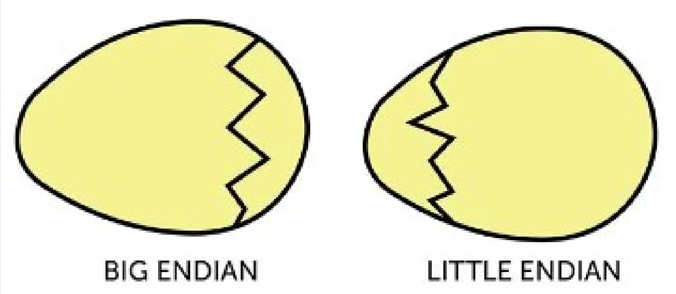
- Addresses still specify locations of bytes, but we can choose to view memory as a series of fixed-size chunks instead
  - Addresses of successive chunks differ by data size
- The address of any chunk of memory is the lowest address of the chunk
  - To specify a chunk, need *both* its address and size
  - Typically **aligned**, meaning their starting addresses are multiples of the data size



Byte Ordering

- How should bytes within a piece of data be ordered in memory?
  - Similar to writing in human languages
    - Ex: English reads left-right, but Arabic reads right-left

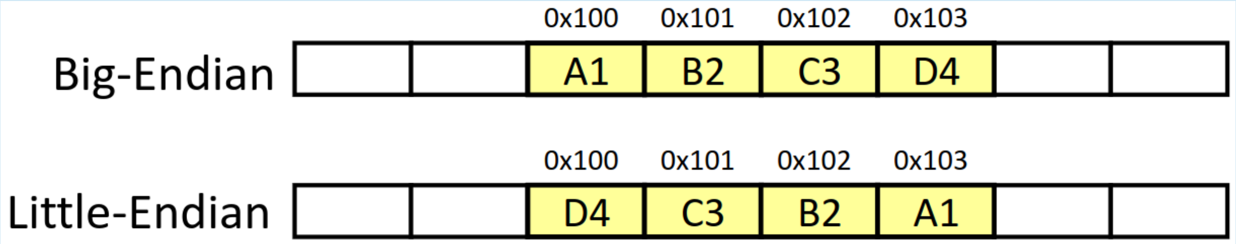
- By convention, ordering of bytes is called *endianness*
  - Two options: **big-endian** and **little-endian**
  - Reference to *Gulliver's Travels*: tribes cut their eggs on different sides



Endianness

- **Big-endian** (SPARC, z/Architecture)
  - Least-significant byte at the highest address
- **Little-endian** (x86, x86-64)
  - Least-significant byte at the lowest address
- **Bi-endian** (ARM, PowerPC)
  - Endianness can be specified as either big or little

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



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?

A) 0x04

B) 0x40

C) 0x01

D) 0x10

E) We're lost…

0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107

Endianness (pt 2)

- Endianness only applies to data storage
- Often, a programmer can ignored endianness because it is handled for them
  - Bytes wired into correct place when reading and storing from memory (hardware)
  - Compiler and assembler generate correct behavior (software)

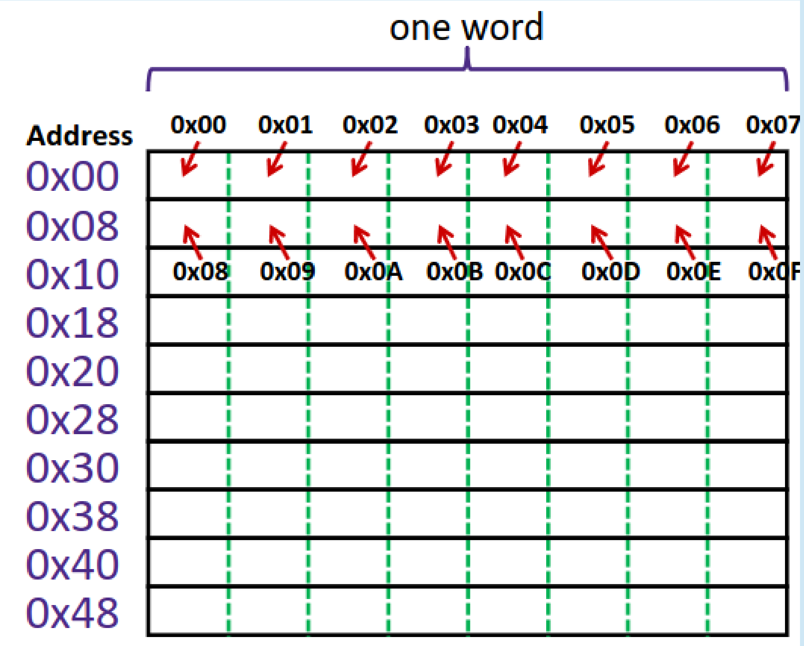
- Endianness still shows up:
  - Logical issues: accessing different amount of data than how you stored it (e.g., store `int`, access byte as a `char`)
  - Need to know exact values to debug memory errors
  - Manual translation of machine code

Data Representations in C

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

A Picture of Memory (64-bit view)

- A 64-bit (8-byte) aligned view of memory, big endian
  - Each cell is one byte
  - Each row is composed of 8 bytes
  - The labels on the left are the starting addresses of each row



Summary

- Memory is a long, *byte-addressed* array
  - **Word size** bounds the size of the address and memory (**address space**)
  - Different data types use different number of bytes
  - Address of multi-byte piece of memory given by the lowest address
- **Endianness** determines memory storage order for multi-byte data
  - Least significant byte in lowest (**little-endian**) or highest (**big-endian**) address of memory chunk