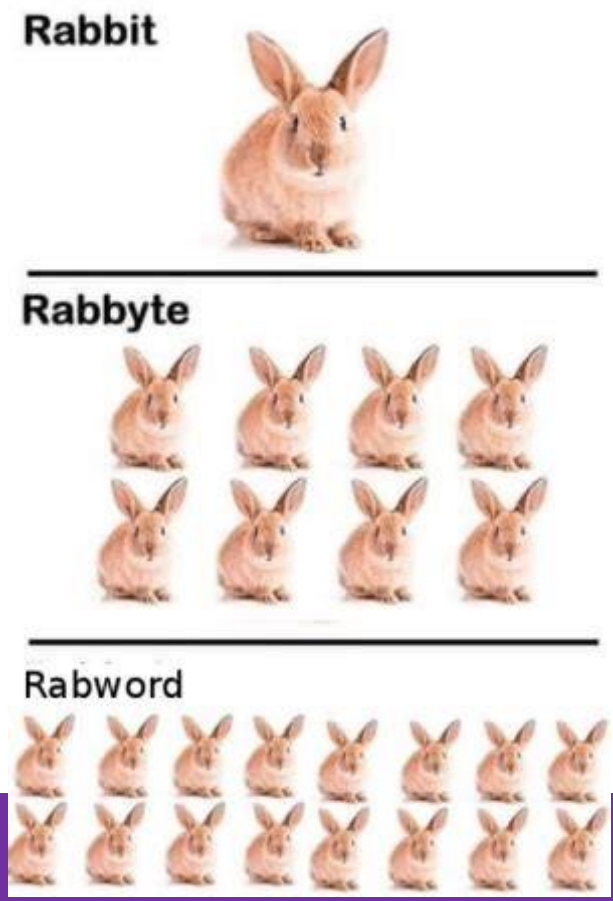
Memory, Data, & Addressing I

CSE 351 Summer 2024

Instructor:
Ellis Haker

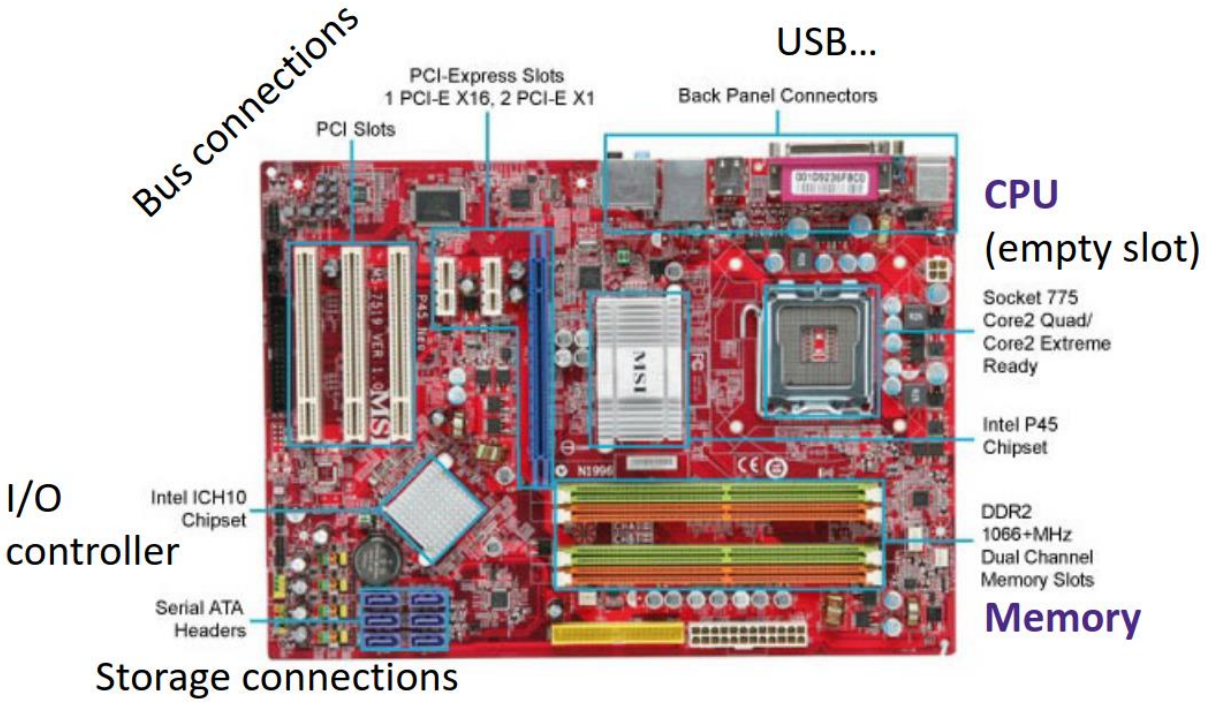
Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



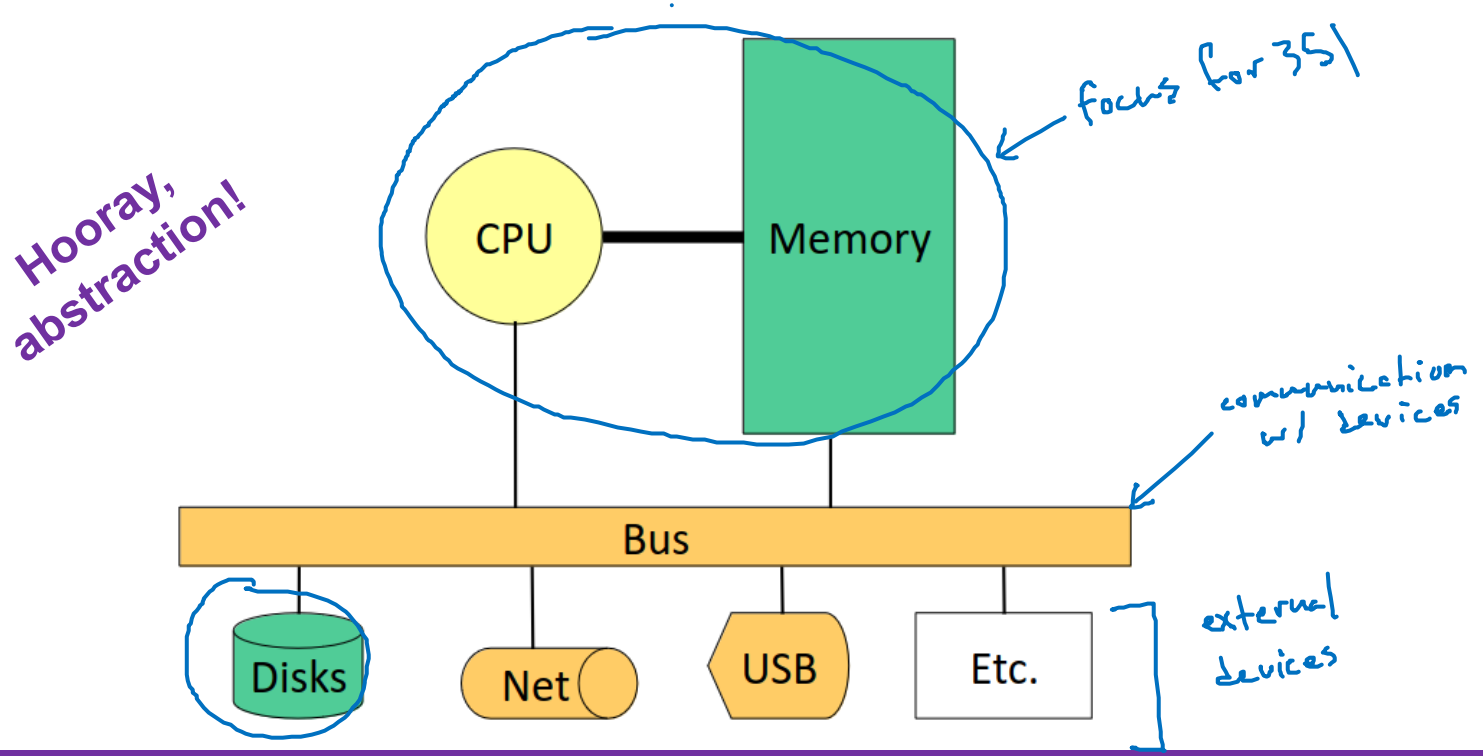
Administrivia

- **<u>HW Deadlines changed</u>**
  - Now due 11:59pm instead of before lecture
  - Pre-lecture readings are still due at 1!
- Office hours are now available!
  - See calendar on the course website
- RD3 due 6/24 (Monday), before 1pm
- HW1 and Lab0 due 6/24 (Monday), before 11:59pm
  - Not like a normal lab - on Ed, individual only, no late days
  - You likely won't understand much of what happens in this lab, and that's ok!
    - We'll revisit it throughout the quarter

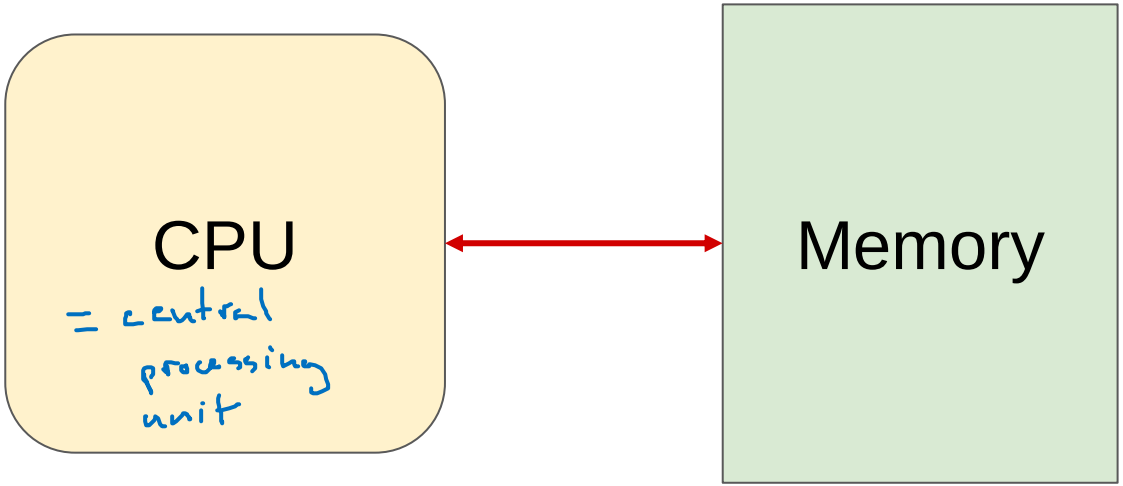
Hardware: Physical View



Hardware: Logical View

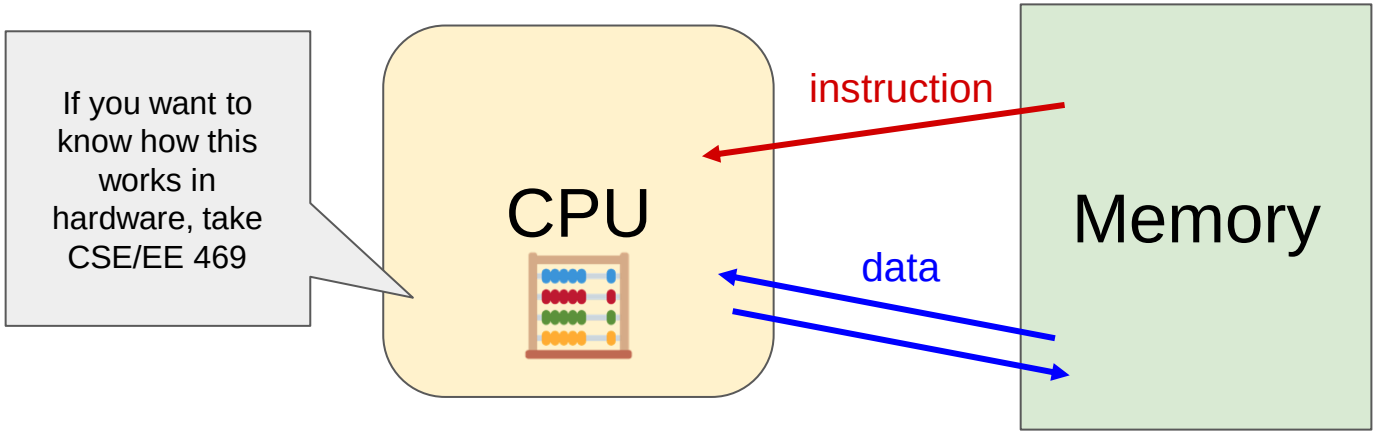


Hardware: 351 View



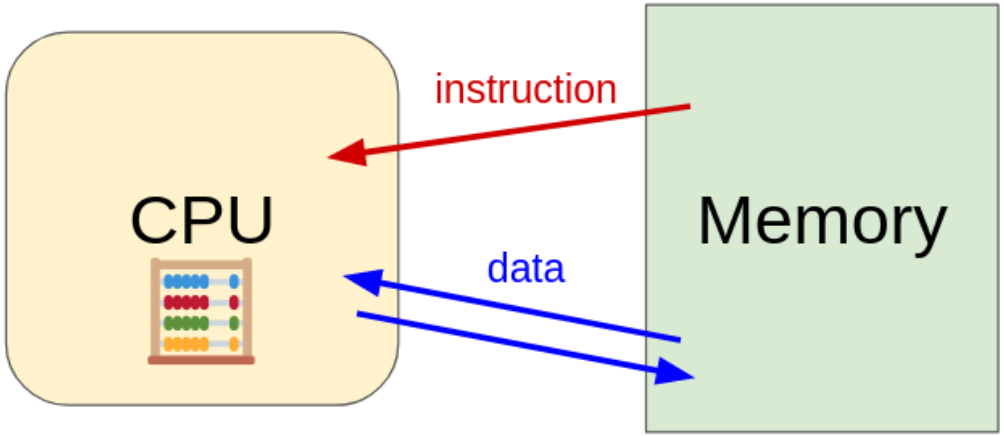
- The **CPU** executes instructions
- **Memory** is where data (including instructions) is stored
- How is data encoded? ***Binary encoding!***

Hardware: 351 View (pt 2)



- 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: 351 View (pt 2)



- We'll start by focusing on memory
  a. How does the CPU know where to find its data? ← today's focus
  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.

depends on the encoding scheme!

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

also need size

Review Questions (pt 2)

1 byte = 8 bits 1 hex digit = 4 bits

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

look up binary in chart

A) 0x63 — 0b 0110 0011

B) 0x90 — 0b 1001 0000

C) 0xCA — 0b 1100 1010

D) 0xD — 0b 0000 1101

need leading 0s to get up to a full byte

4. Consider the following declared variables:

```
int x = 351;
char* p = &x;
```

= 8 bytes, size of an address

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

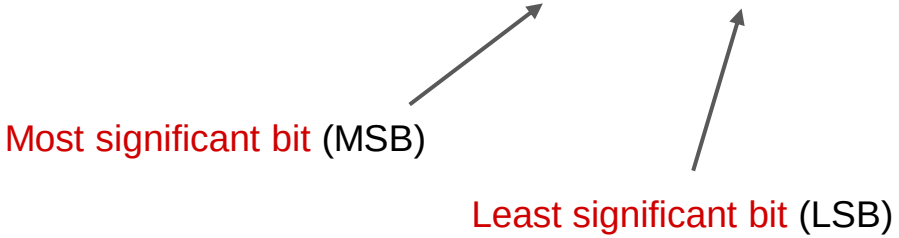
A) 1 byte **B)** 2 bytes **C)** 4 bytes **D)** 8 bytes

ptr size = word size

Fixed-length Binary (Review)

- 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

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



Bits and Bytes and Things (Review)

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

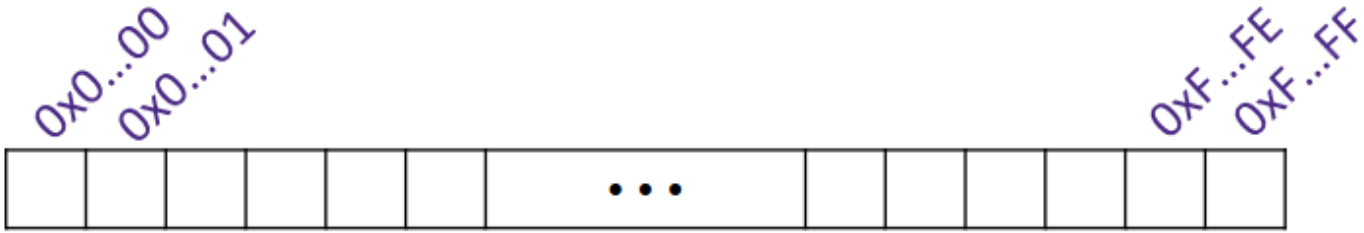
    Example: how many bits would we need to represent the letters a-z?

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

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

Addresses in Memory (Review)

- 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? 2^a
  - So how big is the address space? 2^a bytes



Machine “Words” (Review)

- Instructions are encoded in machine code (in binary)
  - Historically (and still true in some assembly languages), all instructions were exactly the same size, we call this a **word**
- We have chosen to tie word size with address width
  - Still true for languages that don’t have fixed-width instructions: word size = address size

- Most modern systems use **64-bit (8-byte) words**
  - Address space = 2^{64} addresses, each represents a byte of data
  - 2^{64} bytes = $1.8*10^{19}$ bytes = 18 EB (exabytes)
    - (Side note: your computer does not actually have this much memory! We’ll talk about this more later)

Discussion Question

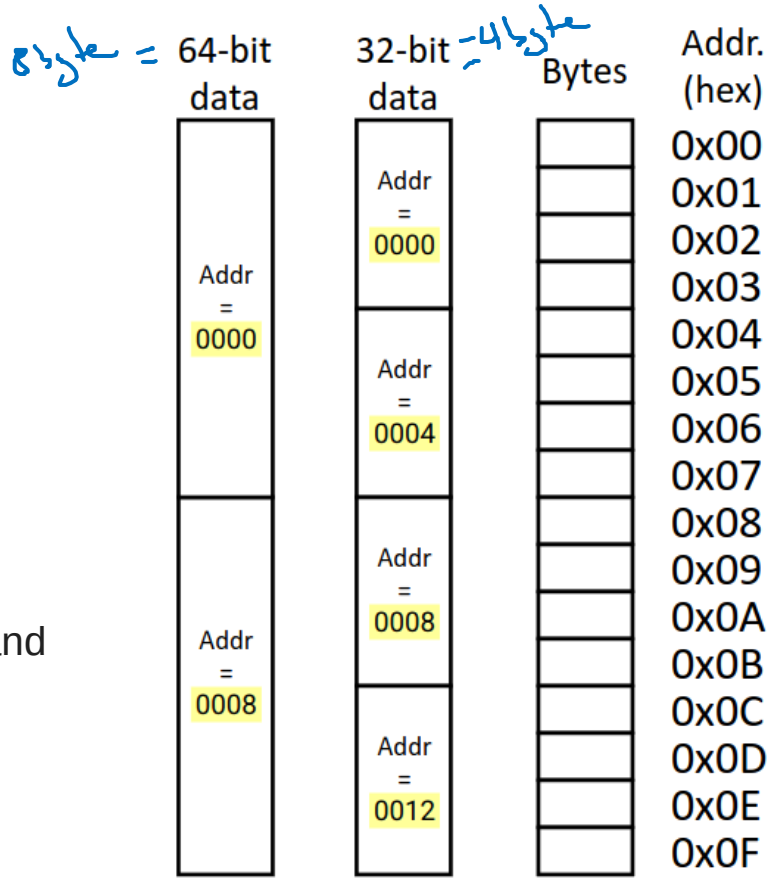
Over time, computers have grown in word size:

Word Size	Instruction Set Architecture	First Intel CPU	Year Introduced
8-bit	?? (Poor & Pyle)	Intel 8008	Early 70s
16-bit	x86	Intel 8086	1978
32-bit	IA-32	Intel 386	1985
64-bit	IA-64 (2001), x86-64(2004)	Itanium	2001

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

Address of Multibyte Data

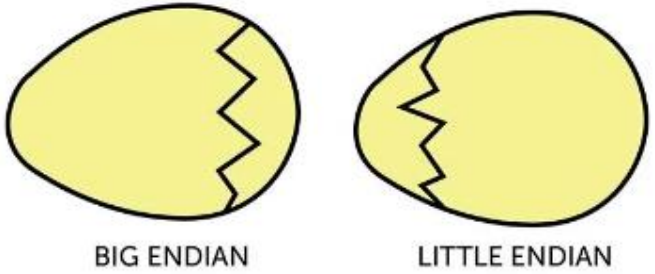
- Addresses still specify locations of bytes, but we can choose to view memory as a series of fixed-size chunks instead
  - Addresses of successive chunks differ by data size
- The address of any chunk of memory is the lowest address of the chunk
  - To specify a chunk, need *both* its address and size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



Byte Ordering (Review)

- 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

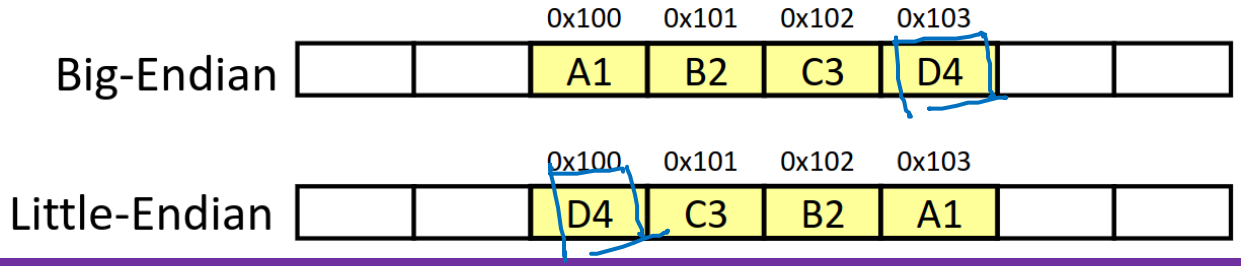
- By convention, ordering of bytes is called *endianness*
  - Two options: **big-endian** and **little-endian**
  - Reference to *Gulliver's Travels*: tribes cut their eggs on different sides



Endianness

- **Big-endian** (SPARC, z/Architecture)
  - Least-significant byte at the highest address
- **Little-endian** (x86, x86-64) what we use in 351
  - 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
00	00	00	00	01	02	03	04

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**

to use bool in C, must `#include <stdbool.h>`

common to use char in place of bool

address size = word size

Pointers in C (Review)

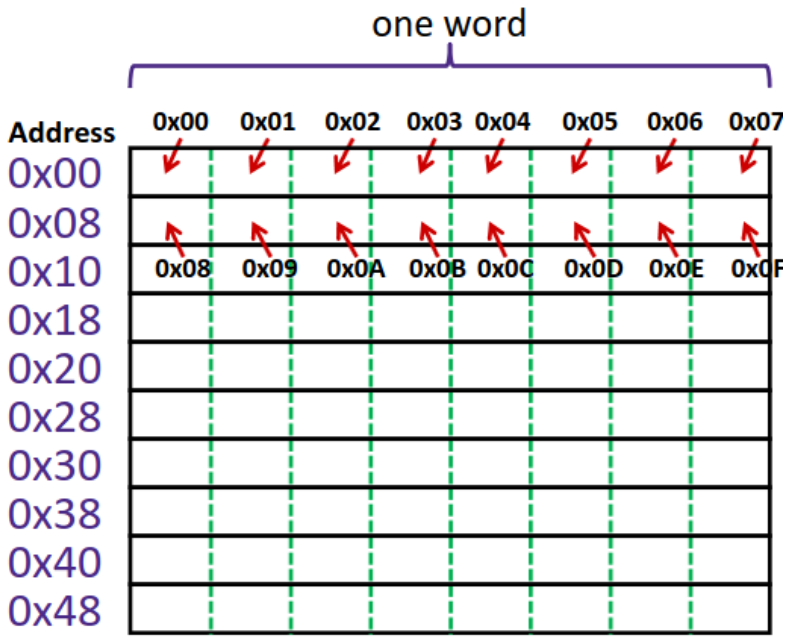
- Data type that stores an address
  - We say it *points to* the data at that address
- Declaration: `<type>* <name>;`
  - The star is *part of the data type* - indicates that it is a pointer

  Example: `int* ptr;` ← without the *, it would be an int variable

- Type of pointer tells you how to interpret the data at that address
  - Ex: a `char*` points to a `char`, an `int*` points to an `int`, etc.
- Probably unfamiliar to most of you, but *not entirely new!*
  - Java uses pointers to reference objects, it just hides this from the programmer

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

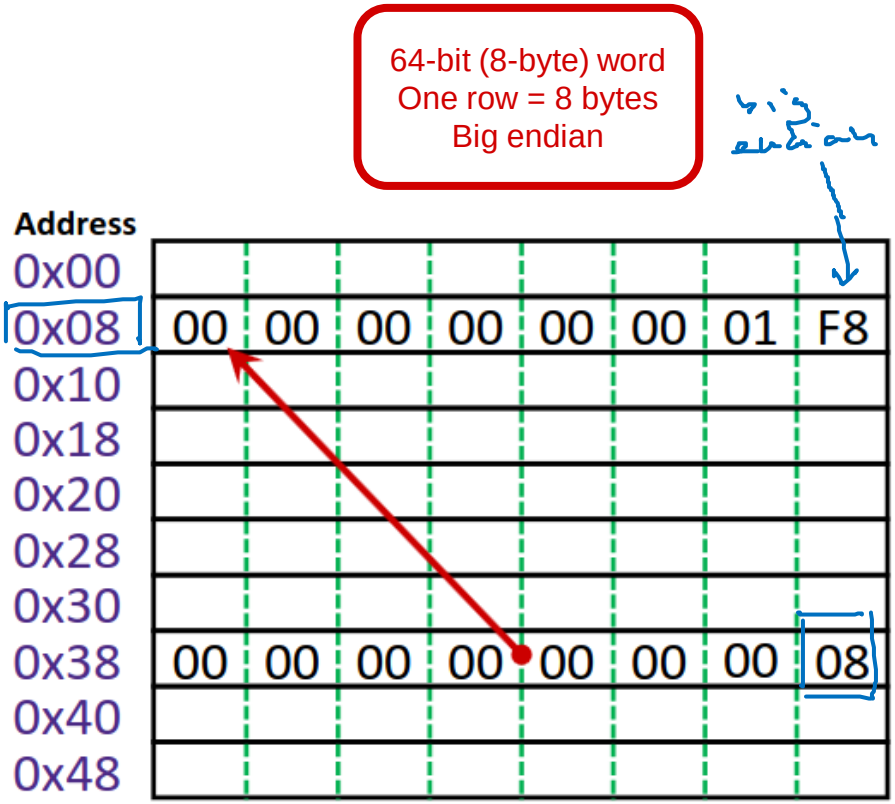


Addresses and Pointers

- A **pointer** is a data object that holds an address
  - Address can point to *any* data

Example:

- 8-byte value 504 (0x1F8) stored at address 0x08
- Pointer stored at 0x38 *points to* the data at 0x08

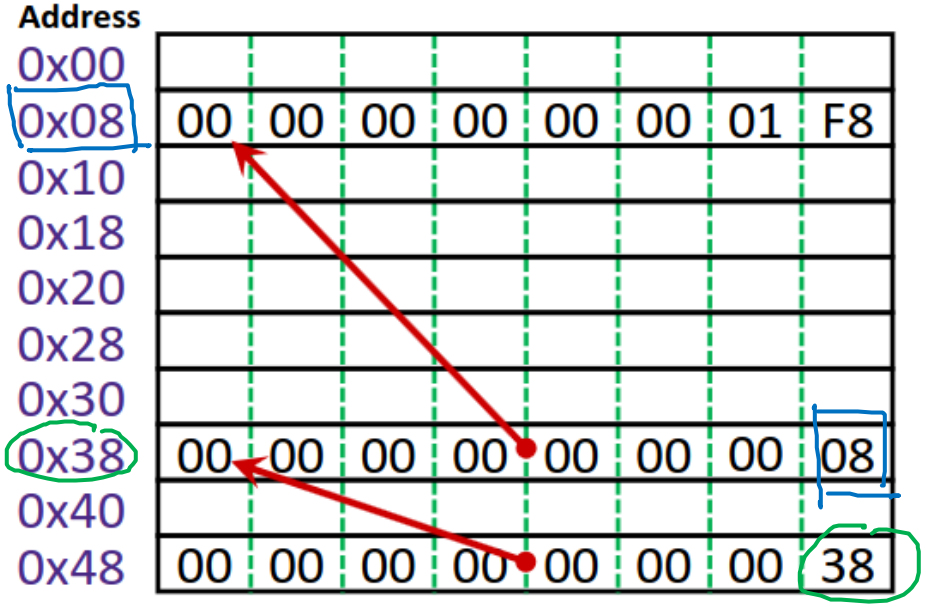


Addresses and Pointers (pt 2)

64-bit (8-byte) word
One row = 8 bytes
Big endian

Example cont.:

- Pointer stored at 0x48 points to the data at 0x38
  - Pointer to a pointer!
- What data type is address 0x08?
  - We don't know! It depends on how you use it
  - Could be a pointer, long, etc.



Pointer Operators

attach to any variable – "give me a pointer to this thing"

pointers only – "give me the thing this points to"

- & = "address of" operator
- * = "value at address" or "dereference" operator
  - Note: * is also used when declaring a pointer variable, this is NOT an operator in this context
- Operator confusion
  - These are *unary* operators (i.e., they take 1 operand)
  - Both & and * are used as *binary* operators as well
    - x & y is bitwise AND (we'll talk about this in the next couple lectures)
    - x * y is multiplication

Addresses and Pointers (pt 3)

64-bit (8-byte) word
One row = 8 bytes
Big endian

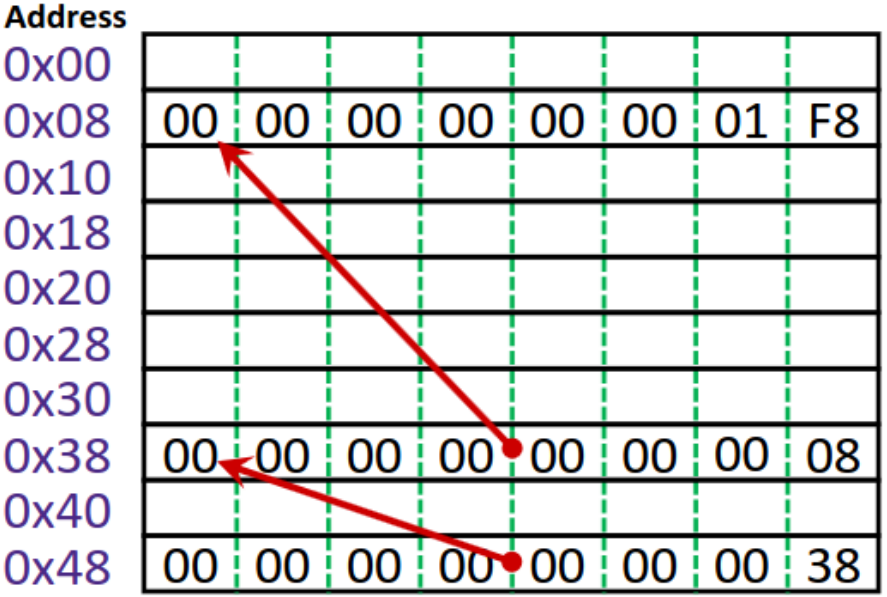
Here's some example C code that may create the memory layout from before:

```
long x = 504;
long* p1 = &x;
long** p2 = &p1;
```

What is the value of `*p2`? What is its data type?

p2 = 0x38, long**

p2 = 0x8, long



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
- **Pointers** are data objects that hold addresses
  - All pointers are the same length as the system's word size
  - Type of pointer determines size of thing being pointed at
  - We use * (**dereference**) and & (**address-of**) operators to interact with pointers