

Memory, Data, & Addressing I

CSE 351, Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Anirudh Kumar

Catherine Guevara

Dara Stotland

Harrison Bay

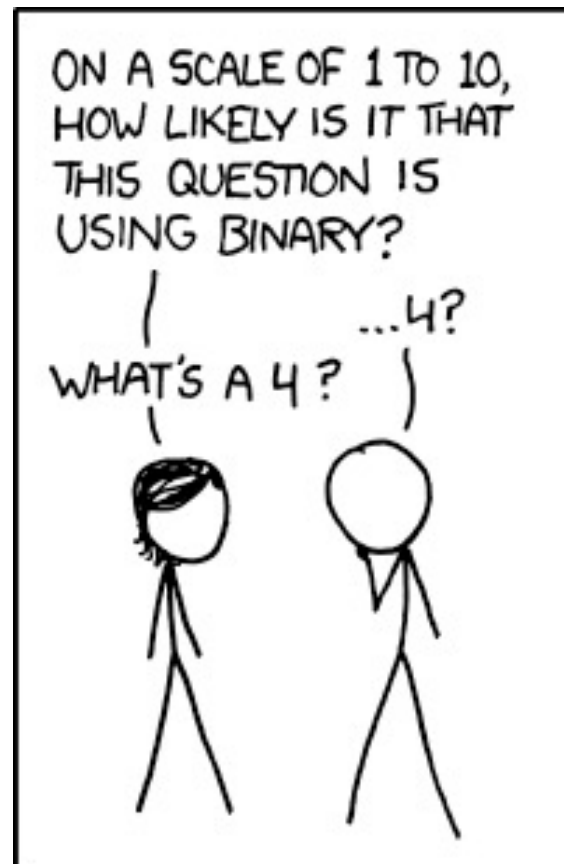
Ian Hsiao

Kevin Wang

Mara Kirdani-Ryan


Nick Durand

Sanjana Sridhar



<http://xkcd.com/953/>

Relevant Course Information

- ❖ Everything except readings due @ 11:59 pm
 - Course Policies homework due tonight
 - Pre-Course Survey and Binary homework due Friday (1/7)
 - Lab 0 due Monday (1/10)
 - This lab is *exploratory* and looks like a homework; 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 your question has been sufficiently answered, make sure that a response has a checkmark 
- ❖ What should you call me? Sam is fine!

Binary Encoding

0b0
0b1 } 2 things

0b00
0b01
0b10
0b11 } 4 things

- ❖ With n binary digits, how many “things” can you represent?
 - Need n binary digits to represent N things, where $2^n \geq N$
 - Example: 5 binary digits for alphabet because $2^5 = 32 > 26$
- ❖ A binary digit is known as a **bit**
- ❖ A group of 4 bits (1 hex digit) is called a **nibble (nybble?)**
- ❖ A group of 8 bits (2 hex digits) is called a **byte**
 - 1 bit \rightarrow 2 things, 1 nibble \rightarrow 16 things, 1 byte \rightarrow 256 things

So, What Does It Mean?

- ❖ *A sequence of bits can have many meanings!*
- ❖ Consider the hex sequence 0x4E6F21
 - Common interpretations include:
 - The decimal number 5,140,257
 - The real number 7.203034×10^{-39}
 - The characters “No!”
 - The background color of this slide
- ❖ It is up to the program/programmer (you!) to decide how to **interpret** the sequence of bits



Binary Encoding – Characters/Text

❖ ASCII Encoding (www.asciitable.com)

■ American Standard Code for Information Interchange

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	70	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	71	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	72	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	73	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	74	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	75	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	76	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	77	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	78	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	79	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	80	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	81	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	82	166	v	v
23	17	027	EB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	83	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	84	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	85	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	86	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	87	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	88	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	89	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	90	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	91	177		DEL

• emoji
• accents
• non-English characters
• etc.

Binary Encoding – Characters/Text

- ❖ ASCII Encoding (www.asciitable.com)
 - *American* Standard Code for Information Interchange
- ❖ Created in 1963 32000 bytes
 - Memory was expensive, 32KB in brand new machines
 - *Economic incentive* to use fewer bits for encoding
- ❖ **Design Goals:**
 - Represent everything on an *American typewriter* as *efficiently* as possible
 - Organize similar characters together
 - Numbers, uppercase, lowercase, then other stuff


Binary Encoding – Unicode & Emoji

❖ Unicode Standard is managed by the Unicode Consortium

→ 2^{32} chars

- “Universal language” that uses 1-4 bytes to represent a much larger range of characters/languages, including emoji
- Adds new emojis every year
 - Offer opportunities to be more inclusive of race and gender diversity
 - However, adoption often lags: 🤴 and 👑 added in 2015 and 2016, but non-gendered “person with crown” only added in 2021: 👤
 - <https://emojipedia.org/new/>

❖ Emojipedia demo: <http://www.emojipedia.org>

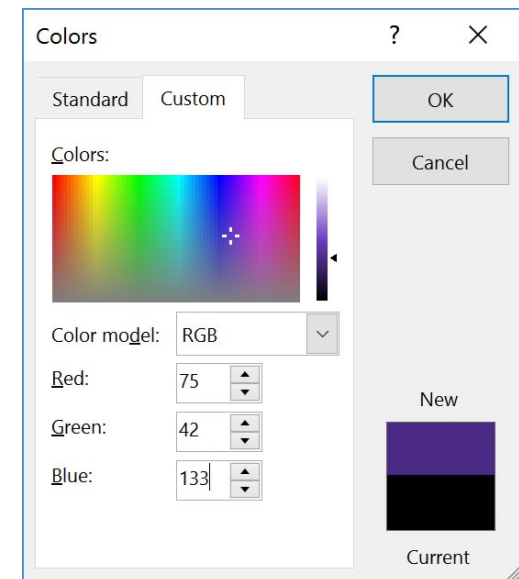
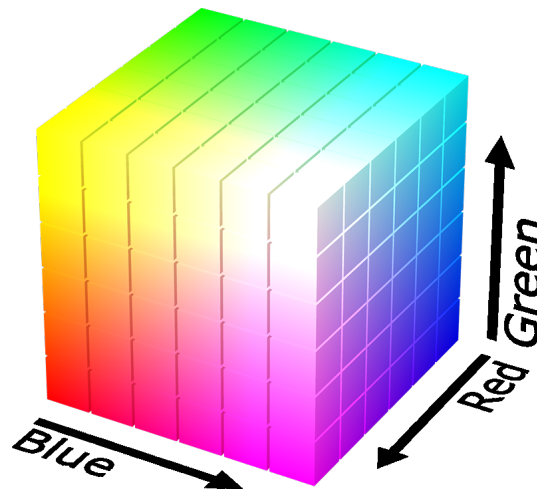
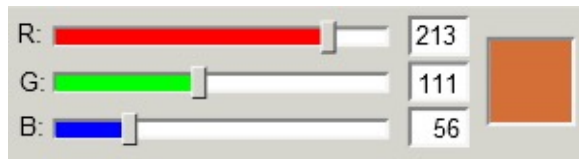
- Desktop Computer: 
- Code points: U+1F5A5, U+FE0F

- Display:       

Binary Encoding – Colors

❖ RGB – Red, Green, Blue

- Additive color model (light): byte (8 bits) for each color
- Commonly seen in hex (in HTML, photo editing, etc.)
- Examples: **Blue**→0x0000FF, **Gold**→0xFFD700,
White→0xFFFFFF, **Deep Pink**→0xFF1493



Binary Encoding – Files and Programs

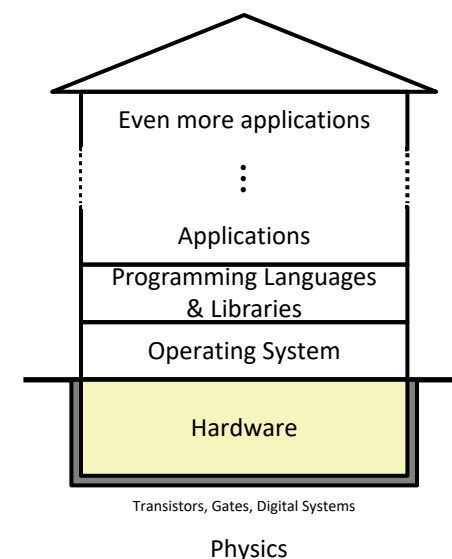
- ❖ At the lowest level, all digital data is stored as bits!
- ❖ Layers of abstraction keep everything comprehensible
 - Data/files are groups of bits interpreted by program
 - Program is groups of bits being interpreted by your CPU
- ❖ Computer Memory Demo (open a binary file with your favorite text editor and then do the following):
 - From vim: `% !xxd`
 - From emacs: `M-x hexl-mode`

Summary

- ❖ Humans think about numbers in decimal; computers think about numbers in binary
 - Base conversion to go between them
 - Hexadecimal is more human-readable than binary
- ❖ All information on a computer is binary
- ❖ Binary encoding can represent *anything*!
 - Computer/program needs to know how to interpret the bits
 - Encodings aren't "neutral"; priorities are baked in

The Hardware/Software Interface

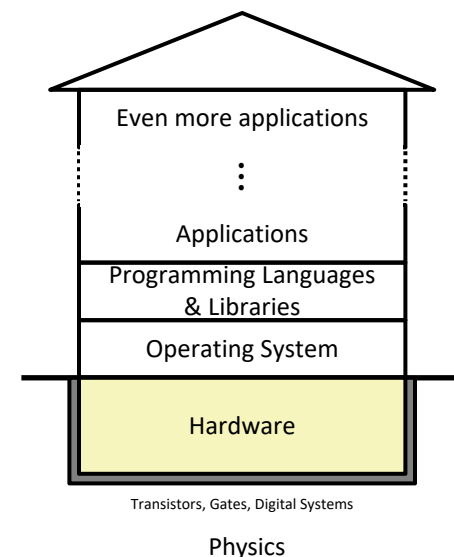
- ❖ Topic Group 1: **Data**
 - **Memory, Data**, Integers, Floating Point, Arrays, Structs
- ❖ Topic Group 2: **Programs**
 - x86-64 Assembly, Procedures, Stacks, Executables
- ❖ Topic Group 3: **Scale & Coherence**
 - Caches, Processes, Virtual Memory, Memory Allocation



The Hardware/Software Interface

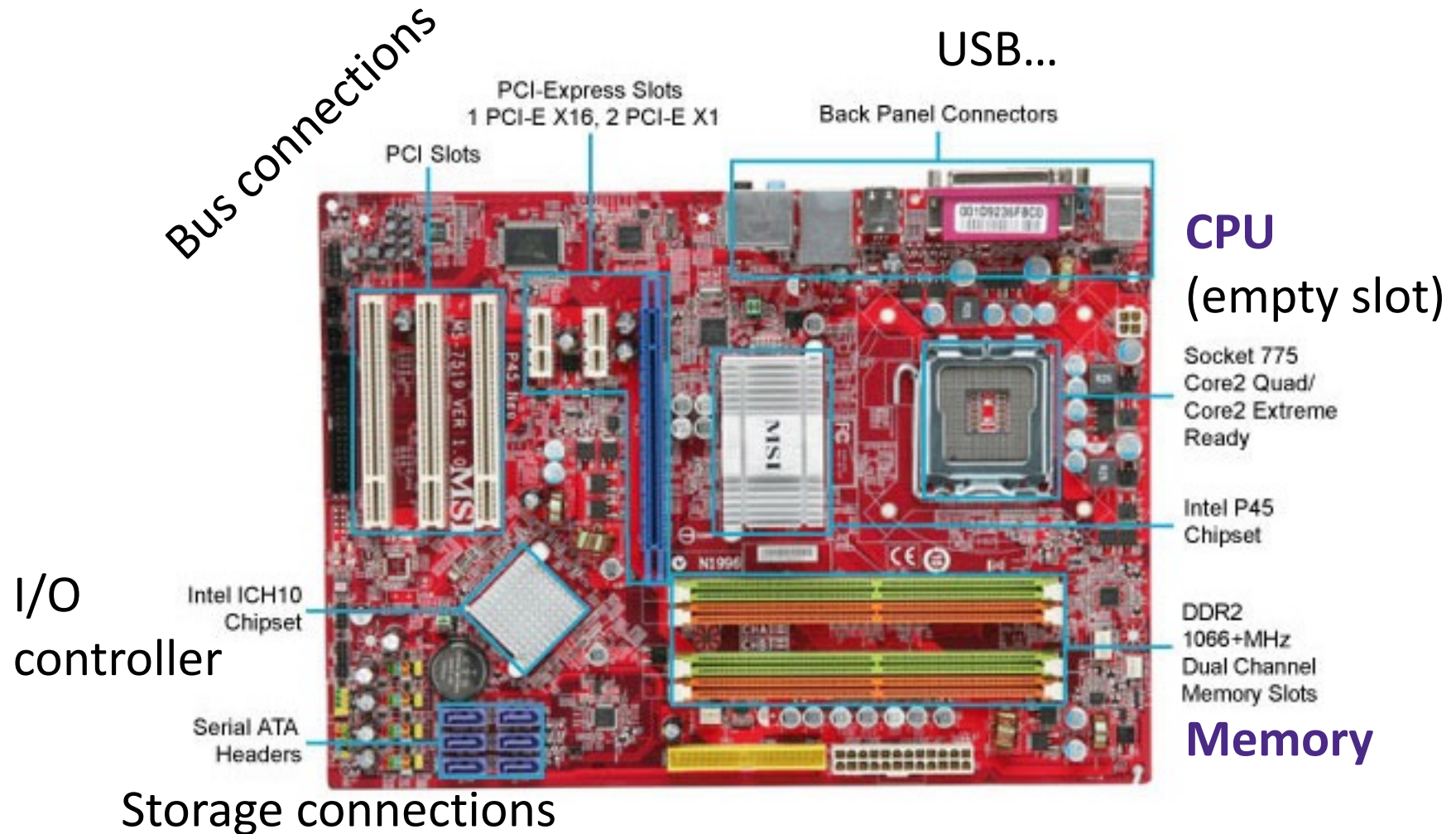
❖ Topic Group 1: **Data**

- **Memory, Data**, Integers, Floating Point, Arrays, Structs

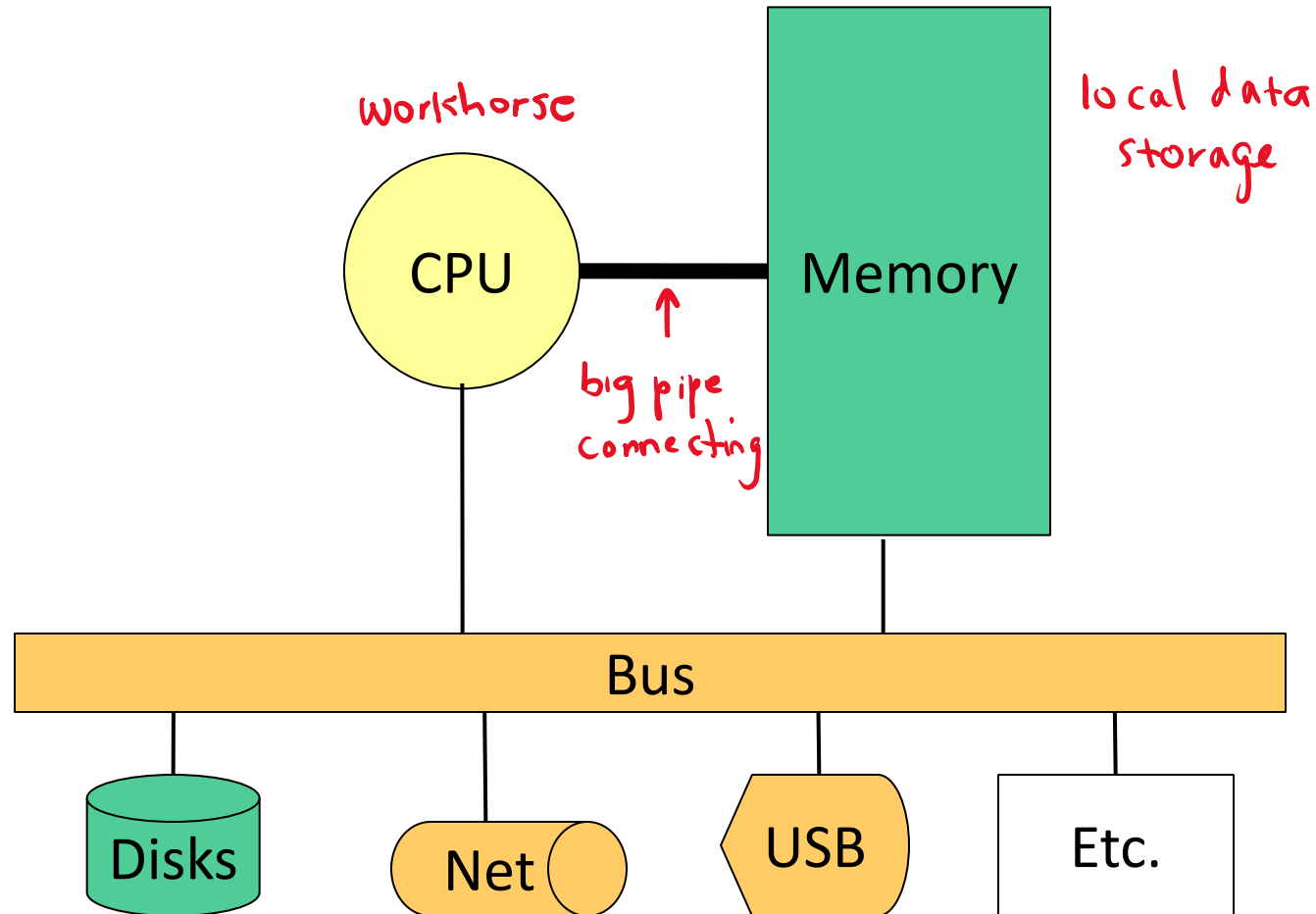


- ❖ How do we store information for other parts of the house of computing to access?
 - How do we represent data and what limitations exist?
 - What design decisions and priorities went into these encodings?

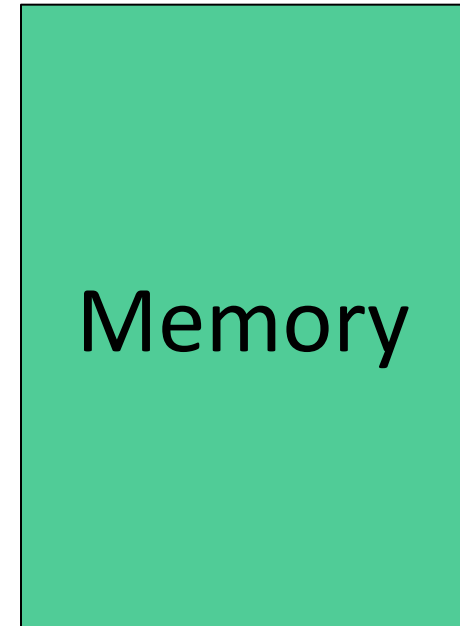
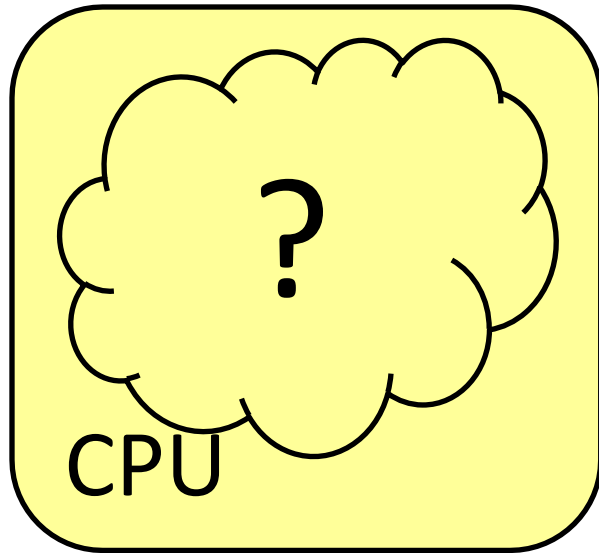
Hardware: Physical View



Hardware: Logical View



Hardware: 351 View (version 0)



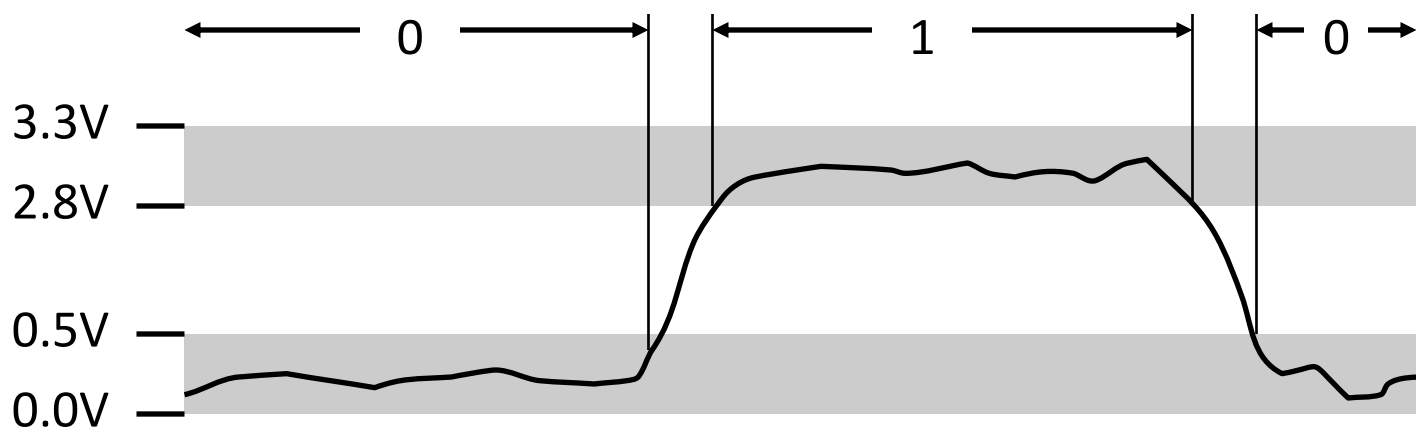
- ❖ The CPU **executes** instructions
- ❖ Memory **stores** data
- ❖ Binary encoding!
 - Instructions *are* just data

Q1: How are data and instructions represented?

Aside: Why Base 2?

❖ Electronic implementation

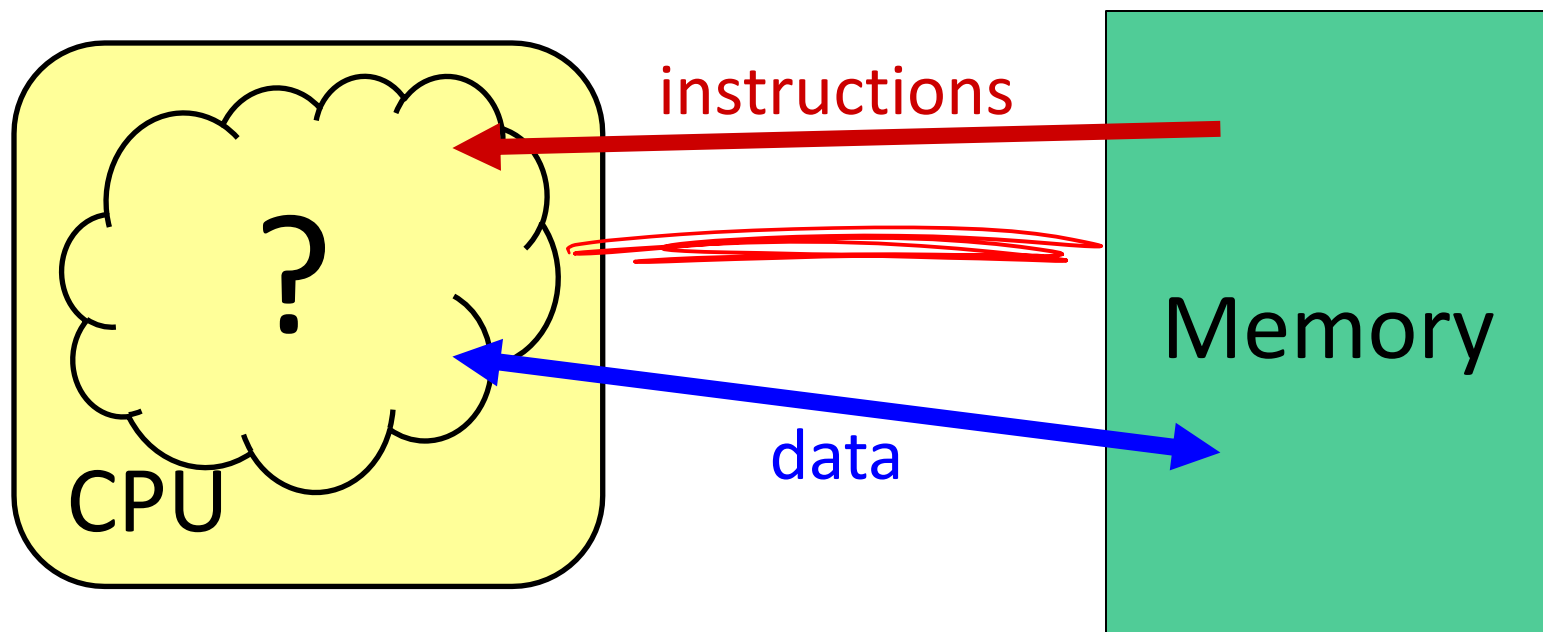
- Easy to store with bi-stable elements
- Reliably transmitted on noisy and inaccurate wires



❖ Other bases possible, but not yet viable:

- DNA data storage (base 4: A, C, G, T) is hot @UW
- Quantum computing

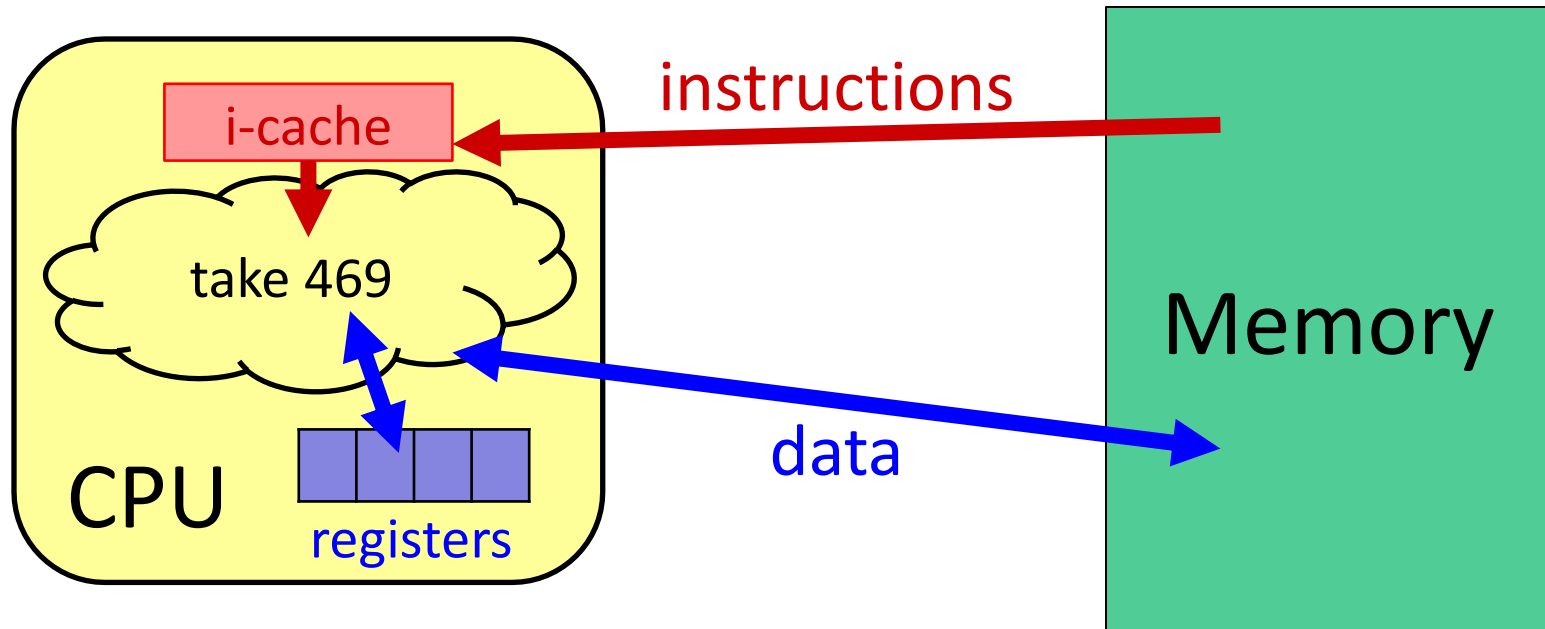
Hardware: 351 View (version 0)



- ❖ 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

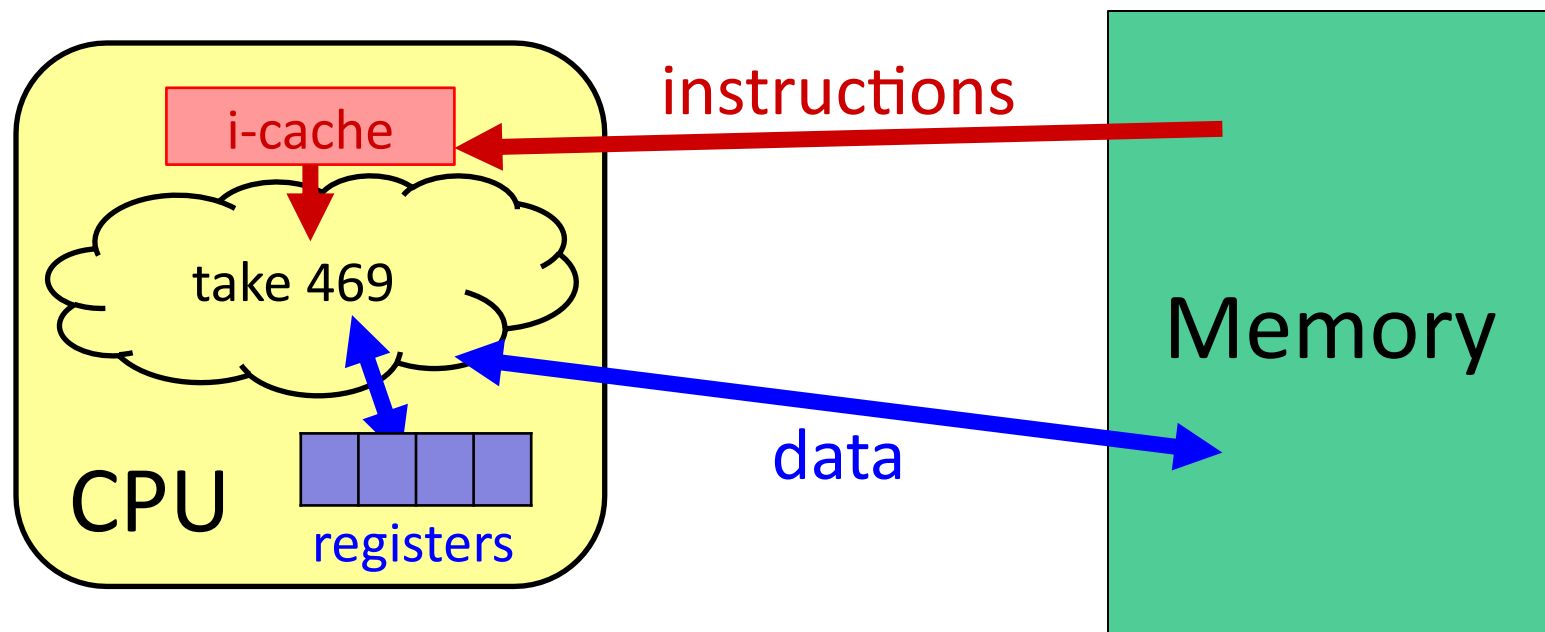
Hardware: 351 View (version 1)

This is extra
(non-testable)
material



- ❖ More CPU details:
 - Instructions are held temporarily in the **instruction cache**
 - 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

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 4 bytes is being used to represent.

A. True

B. False

many possible encodings

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

A. True

B. False

need size too

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

A. 0x63

0b01100011

B. 0x90

0b10010000

C. 0xCA

0b11001010

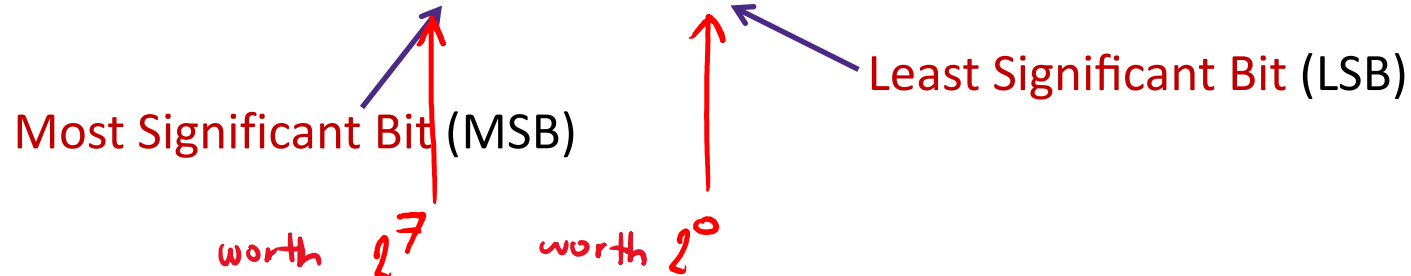
D. 0xF

0b00001111

implicit leading 0

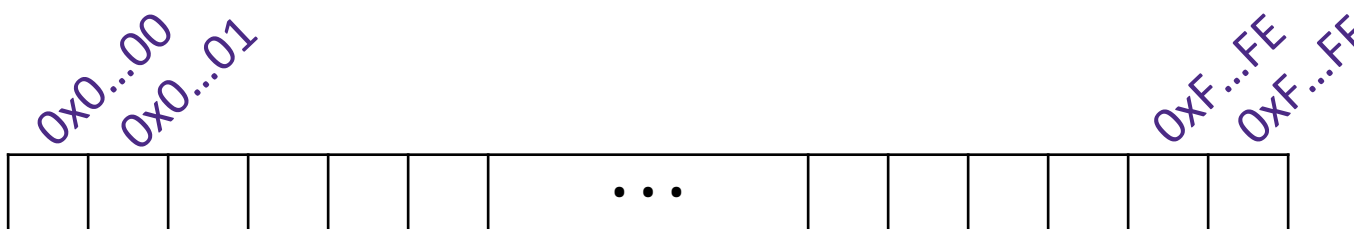
Fixed-Length Binary (Review)

- ❖ 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” representation of the number 4 is 0b00000100



Bits and Bytes and Things (Review)

- ❖ 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
- ❖ What does each address refer to?



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**
- ❖ We have *chosen* to tie word size to address size/width
 - word size = address size = register size
 - word size = w bits $\rightarrow 2^w$ addresses
- ❖ Current x86 systems use **64-bit (8-byte) words**
 - Potential address space: 2^{64} addresses
 2^{64} bytes \approx **1.8×10^{19} bytes**
= 18 billion billion bytes = 18 EB (exabytes)
 - Actual physical address space: **48 bits**

Data Representations

❖ Sizes of data types (in bytes)

32 bit

64 bit

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

We'll only use this

address size = word size

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?

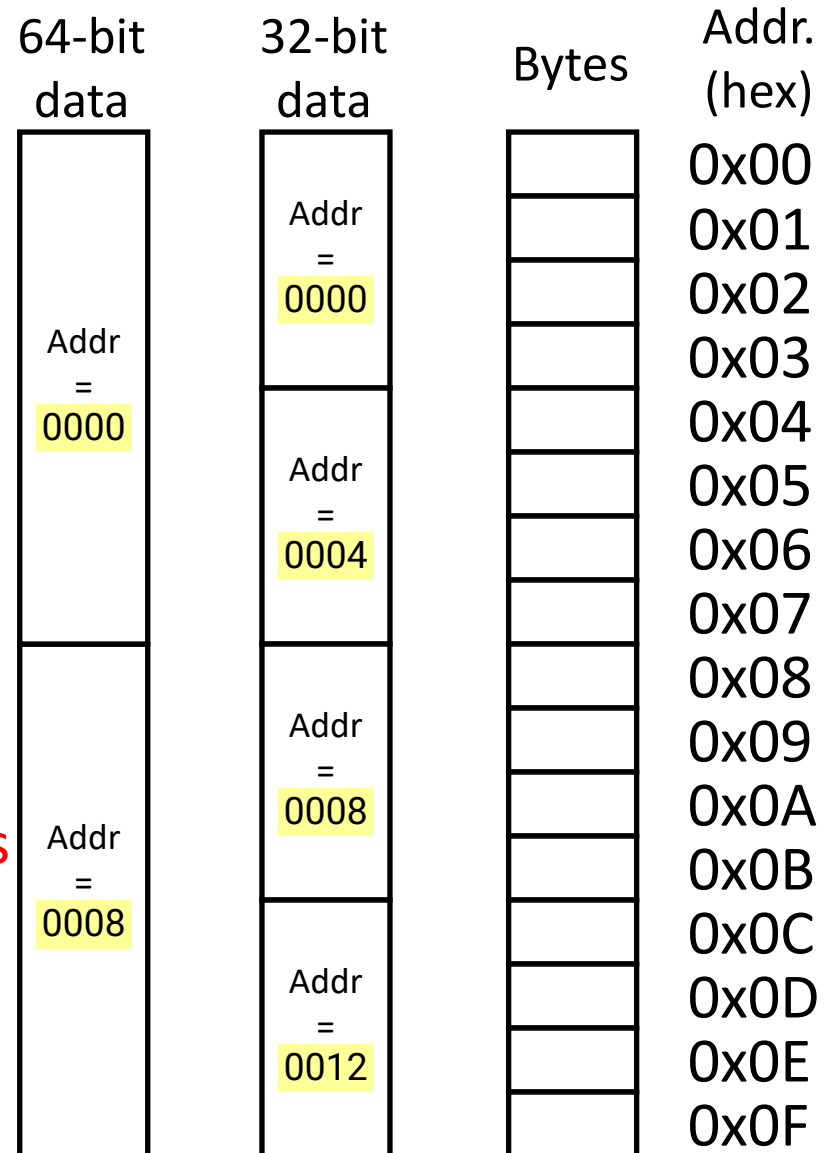
causes: tech development, cheaper memory

adv: access more total mem

dis: more complex/expensive thus larger pointers

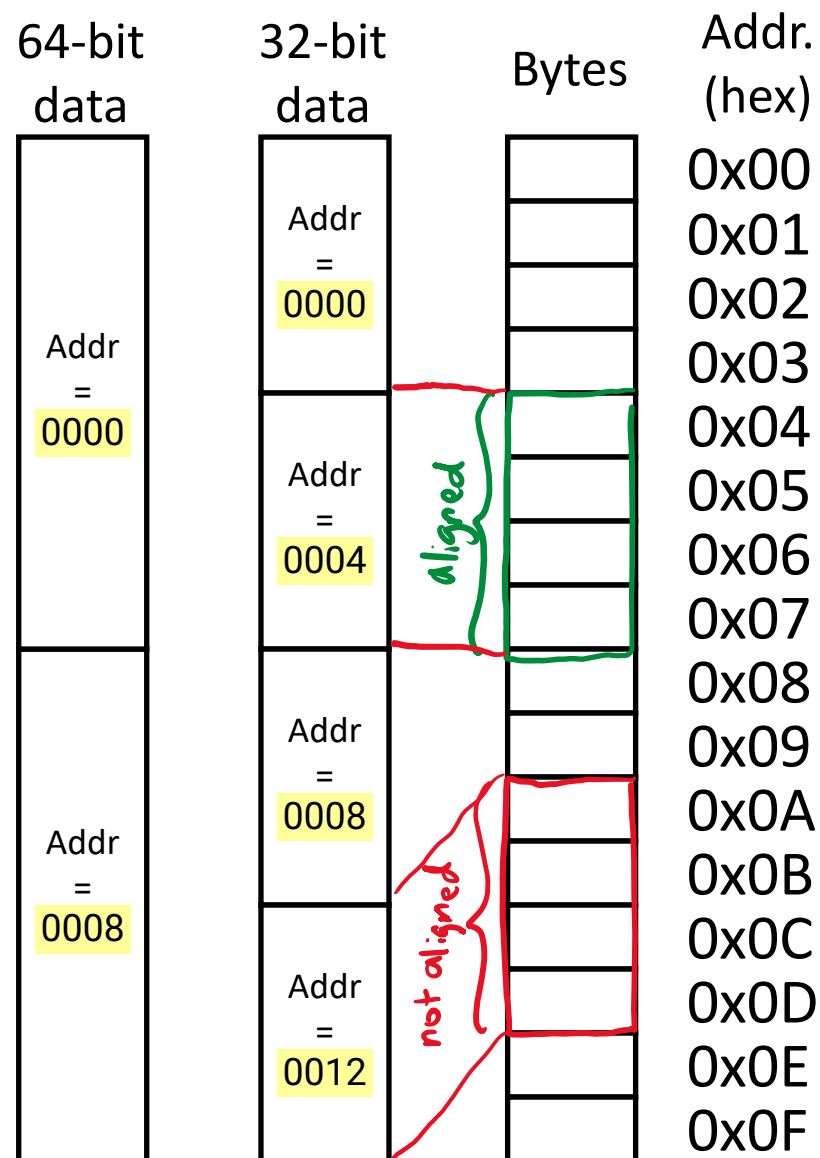
Address of Multibyte Data (Review)

- ❖ 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**



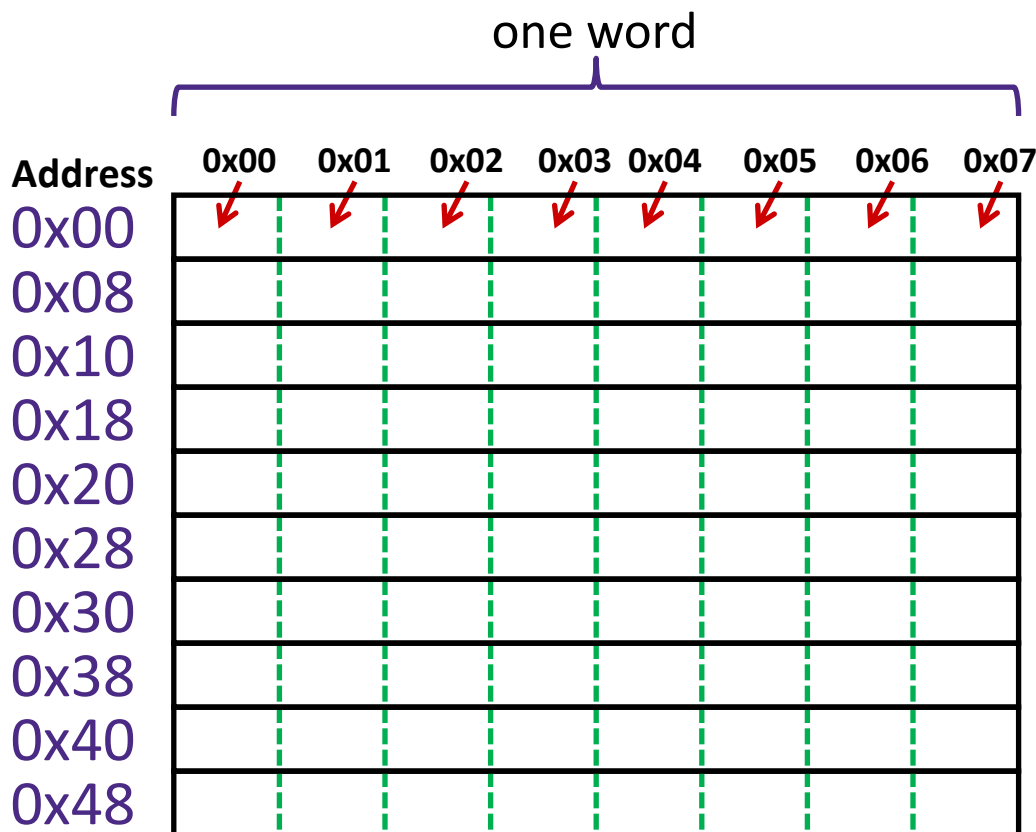
Alignment

- ❖ 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



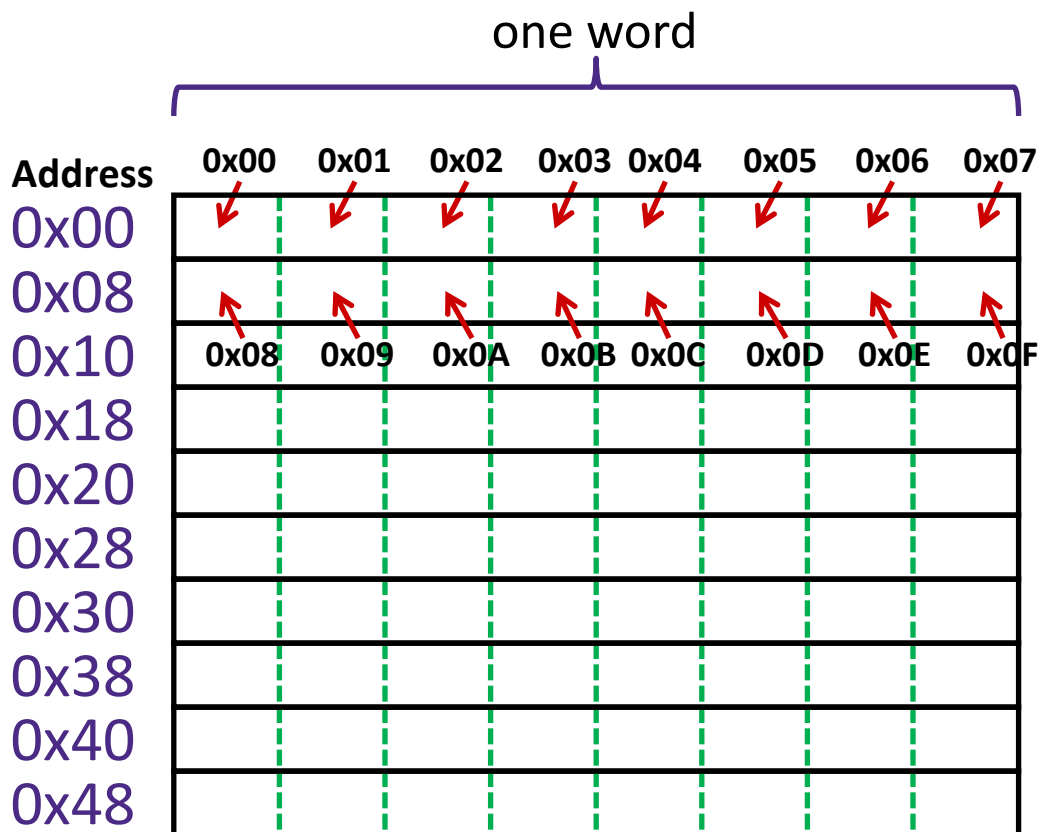
A Picture of Memory (64-bit view)

- ❖ 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
- ❖ 8-byte 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

0x10

0x18

0x20

0x28

0x30

0x38

0x40

0x48

00	00	00	00	00	00	01	F8	
00	00	00	00	00	00	00	08	

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

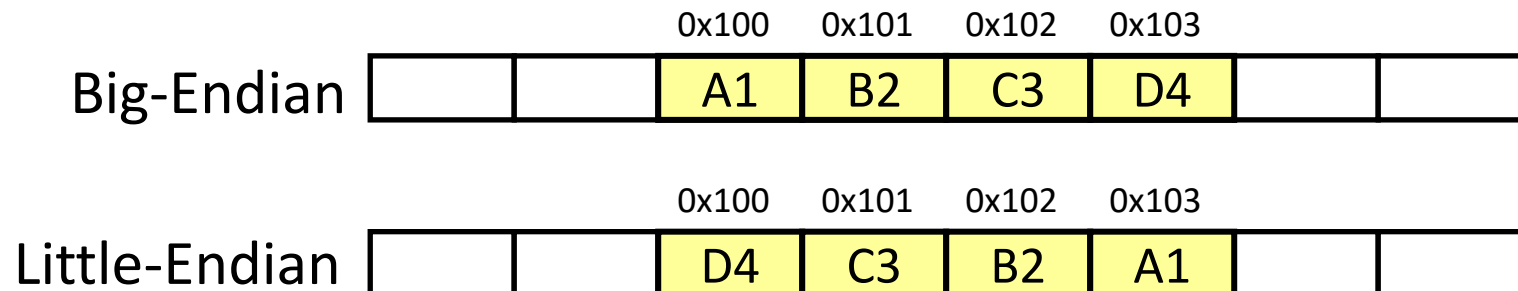


Byte Ordering (Review)

- ❖ 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?
 - 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
- ❖ **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?
 - Vote at [PollEv.com/wolfson](https://pollev.com/wolfson)

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)

Exploration 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	...

- ❖ 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
 - 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