Introduction to Computer Science:

Data Representation

Mar. 2020

Honguk Woo

- *How many ones are there in 642 ?*

Positional Notation

- *How many ones are there in 642 ?*

600 + 40 + 2

or is it

384 + 32 + 2

or maybe…

1536 + 64 + 2

Positional Notation

- The **base** of a number determines the number of different digit symbols and the values of digit positions
- Positional notation: the rightmost digit represent its value multiplied by the base to the 0th power …, the next digits by the 1st power, …
- $$\begin{array}{lllll} & 6 \text{ x } 10^2 & = & 6 \text{ x } 100 & = 600 \\ + & 4 \text{ x } 10^1 & = & 4 \text{ x } 10 & = 40 \\ + & 2 \text{ x } 10^0 & = & 2 \text{ x } 1 & = 2 \\ & & & & = 642 \text{ in base } 10 \end{array}$$

Positional Notation

- *What if 642 has the base of 13?*

$$
\begin{aligned}
& 6 \times 13^2 = 6 \times 169 = 1014 \\
+\ & 4 \times 13^1 = 4 \times 13 = 52 \\
+\ & 2 \times 13^0 = 2 \times 1 = 2 \\
& = 1068 \text{ in base } 10
\end{aligned}
$$

- 642 in base 13 is equivalent to 1068 in base 10

Binary

- Binary is base 2 and has 2 digit symbols:

  0,1

- Decimal is base 10 and has 10 digit symbols:

  0,1,2,3,4,5,6,7,8,9

- For a number to exist in a given base, it can only contain the digits in that base, which range from 0 up to (but not including) the base

Binary : Arithmetic

- Recall that there are only 2 digit symbols in binary, 0 and 1
- 0 + 0 = 0, 1+ 0 = 1
- 1 + 1 is 0 with a carry
- This rule is applied to every column

1 0 1 1 1 1 1

1 0 1 0 1 1 1

+1 0 0 1 0 1 1

1 0 1 0 0 0 1 0

Binary to Octal (base 8)

- Mark groups of *three* (from right)
- Convert each group

10101011 $\underline{10}\ \underline{101}\ \underline{011}$

2 5 3

- 10101011 is 253 in base 8

Binary to Hexadecimal

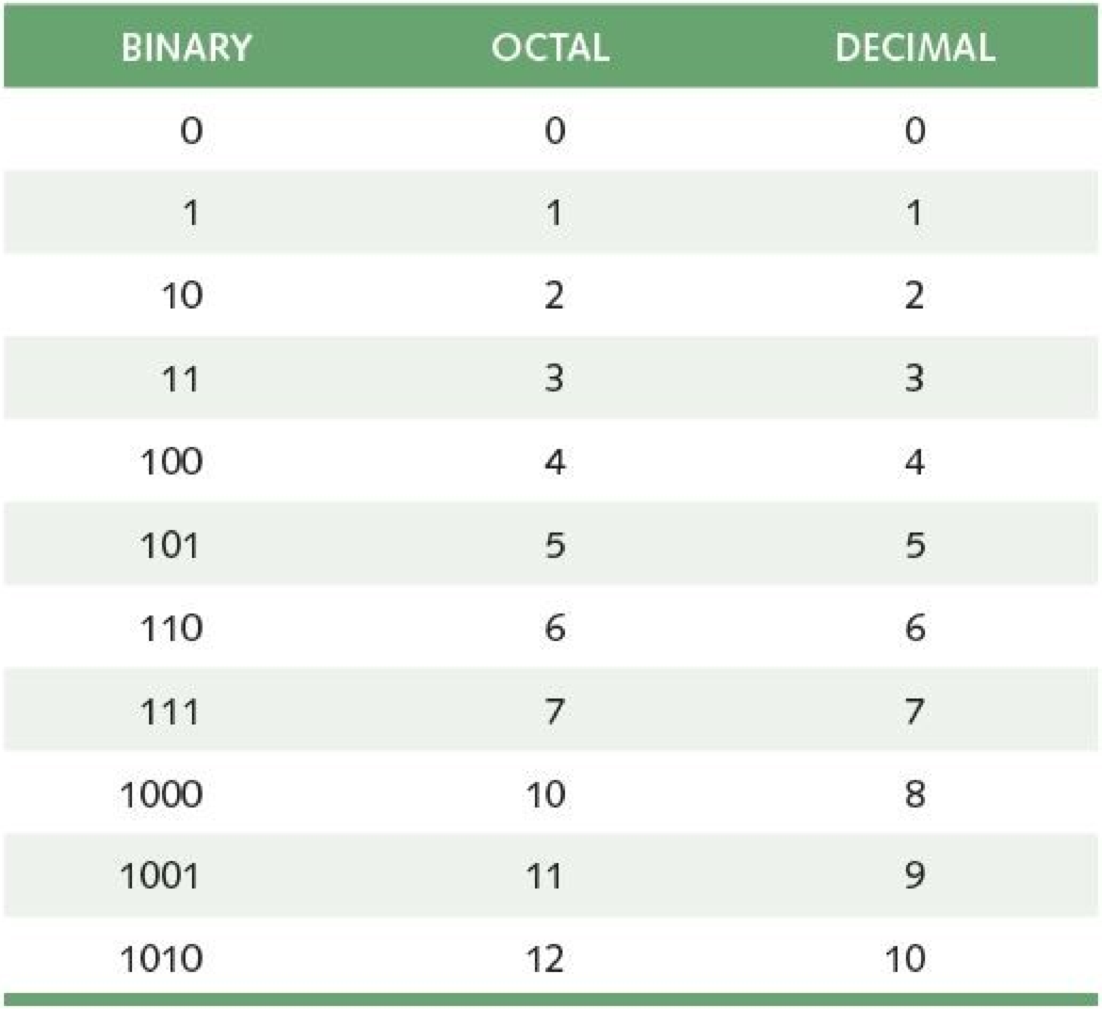
- Base 16 has 16 digits
  - 10 (0~9) plus 6 distinct symbols : A, B, C, D, E, F
- Mark groups of *four* (from right)
- Convert each group

10101011 1010 1011

A B

- 10101011 is AB in base 16

Power-of-2 number system



BINARY	OCTAL	DECIMAL
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	10	8
1001	11	9
1010	12	10

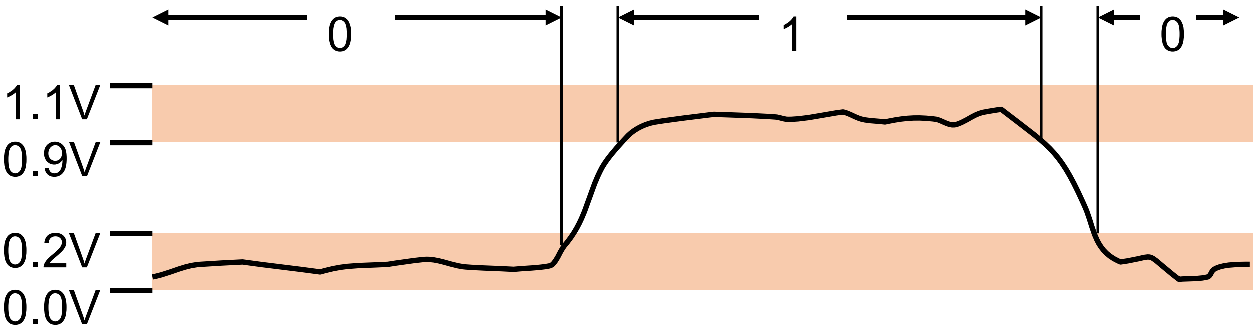
Why Don’t Computers Use Base 10?

- Base 10 Number Representation
  - That’s why **fingers** are known as “**digits**”
  - Natural representation for financial transactions
    - Floating point number cannot exactly represent $1.20
  - Even carries through in **scientific notation**
    - 1.5213 X 10^4

- Implementing Electronically
  - Hard to store
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.

Binary Representations

- Computers have storage units called binary digits or bits
- Base 2 Number Representation
- Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires
  - Low Voltage = 0, High Voltage = 1



  - Straightforward implementation of arithmetic functions

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - **`0xFA1D37B` or `0xfa1d37b`**

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

Binary Representations

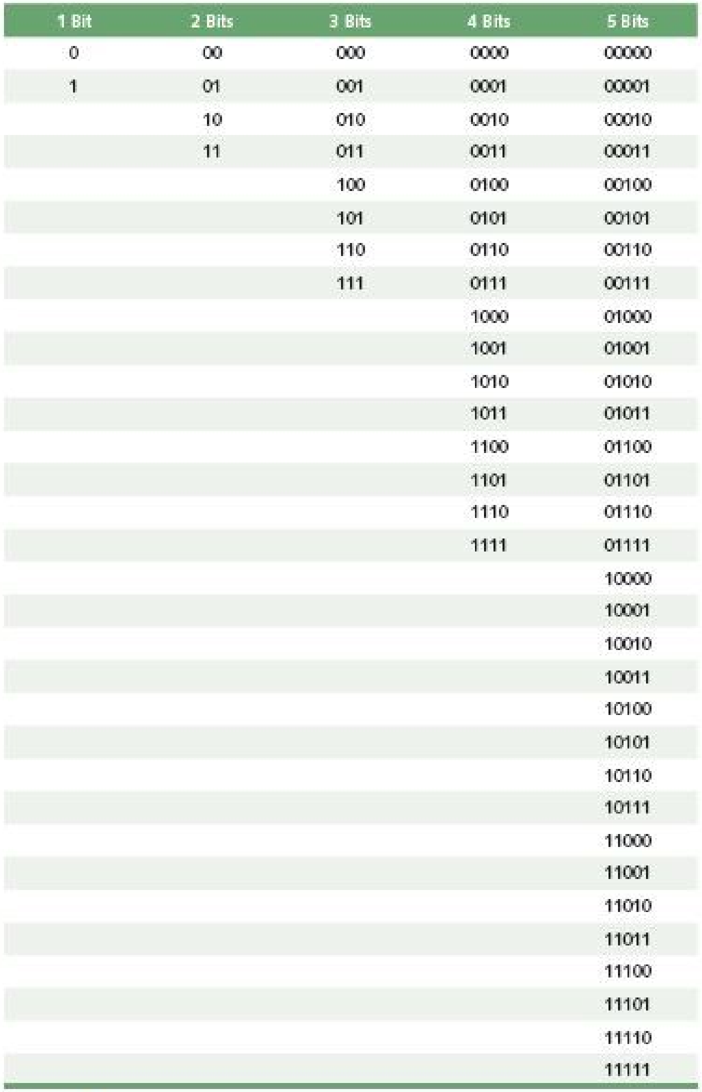
- Each bit can be either 0 or 1, so it can represent a choice between two possibilities (or "two things")
- Two bits can represent four things

How many things can three bits represent?

How many things can four bits represent?

How many things can eight bits represent?

Binary Representations



1 Bit	2 Bits	3 Bits	4 Bits	5 Bits
0	00	000	0000	00000
1	01	001	0001	00001
	10	010	0010	00010
	11	011	0011	00011
		100	0100	00100
		101	0101	00101
		110	0110	00110
		111	0111	00111
			1000	01000
			1001	01001
			1010	01010
			1011	01011
			1100	01100
			1101	01101
			1110	01110
			1111	01111
				10000
				10001
				10010
				10011
				10100
				10101
				10110
				10111
				11000
				11001
				11010
				11011
				11100
				11101
				11110
				11111

FIGURE 3.4 Bit combinations

Binary Representations

- *How many bits are needed to represent 32 things? One hundred things?*

- *How many things can* n bits *represent? Why?*

- *What happens every time you increase the number of bits by one?*

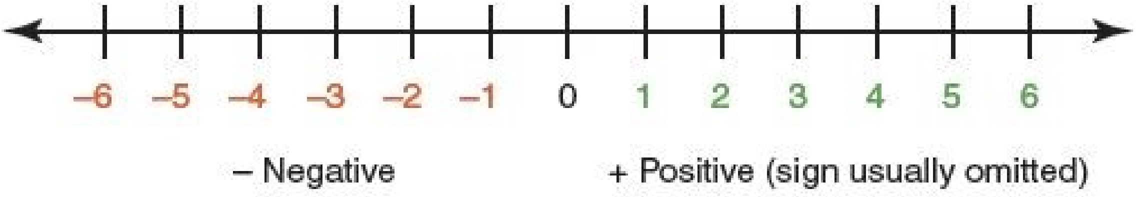
Representing Numbers

- Mapping binary code to numbers
  - Positive numbers seem ok
  - What about negative numbers, and real numbers ?

8-bit Binary Representation	Numbers
01111111	127
01111110	126
…	…
00000011	3
00000010	2
00000001	1
00000000	0

Representing Negative Values

- **Signed-magnitude number representation**
  - Used by humans
  - The sign represents the ordering (the negatives come before the positives in ascending order)
  - The digits represent the magnitude (the distance from zero)



- How do we represent singed binary numbers if all we have is a bund of 1s or 0s ?

Representing Negative Values

8-bit binary word:

00110101 = 53

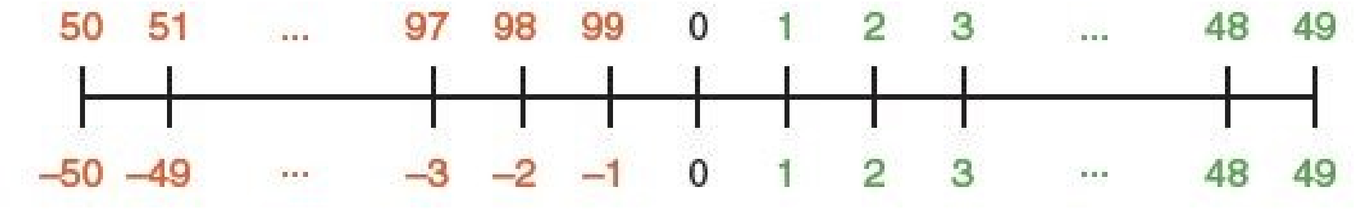
10110101 = -53

- Signed magnitude number Problem:
  - Two zeroes (positive and negative)
  - No problem for humans, but would cause unnecessary complexity in computers

- Solution: Represent integers by associating them with natural numbers
  - Half the natural numbers will represent themselves
  - The other half will represent negative integers

Representing Negative Values

- Using two decimal digits (0~99), we can
  - let 0 through 49 represent 0 through 49
  - let 50 through 99 represent -50 through -1



Representing Negative Values

- Called ten’s complement representation, because we can use this formula to compute the representation of a negative number

$$\text{Negative}(I) = 10^k - I, \text{ where } k \text{ is the number of digits}$$

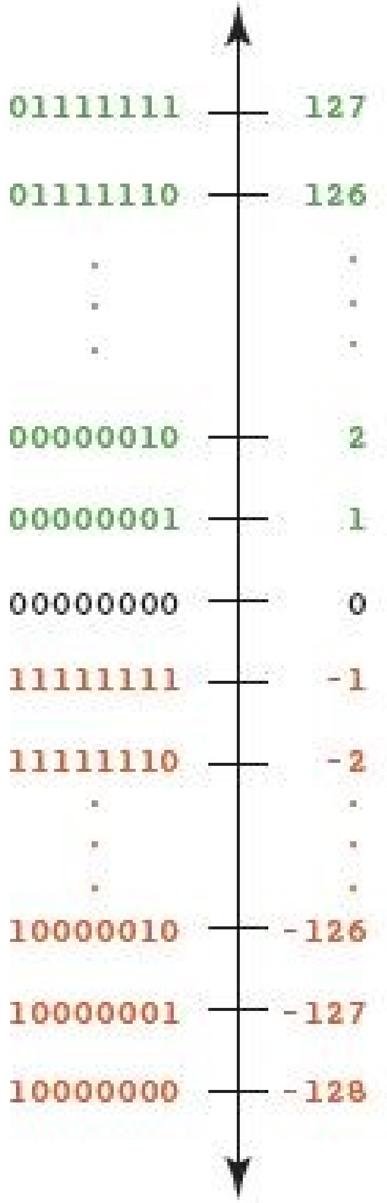
- For example, -3 is Negative(3), so using two digits, its representation is

  Negative(3) = 100 – 3 = 97

Representing Negative Values

- **Two' s Complement**
  - (The binary number line is easier to read when written vertically)

- *Remember our table showing how to represent natural numbers*
- *Do you notice something interesting about the left-most bit ?*



Encoding Integers : Two’s Complement

- *w*-bit vector : [x_{w-1}, x_{w-2}, …., x_0]

Unsigned (0, positive)

$$B2U(X) \quad = \quad \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two’s Complement (negative, 0, positive)

$$B2T(X) \quad = \quad -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit
(most significant bit)

- e.g., 4bit binary -> integer
  - 1111
  - Unsigned :
  - Two’s Complement :

Range of unsigned integers

- **Unsigned**, 4 bits

1111 (15, max)
:
:
0111 (7)
:
:
0000 (0, min)

$$B2U(X) \quad = \quad \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Range of integers

- **Two's complement**, 4 bits

0111 (7, max)
:
:
0000 (0)
1111 (-1)
:
:
1000 (-8, min)

$$B2T(X) \quad = \quad -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Two’s Complement

- Addition and subtraction are the same

$$\begin{array}{rr} -127 & 10000001 \\ \underline{+\quad 1} & \underline{00000001} \\ -126 & 10000010 \end{array}$$

- *What if the computed value won't fit?*

Number Overflow

- If each value is stored using 8 bits, then 127 + 3 overflows:

$$\begin{array}{r} 01111111 \\ +\ 00000011 \\ \hline 10000010 \end{array}$$

- *Apparently, 127 + 3 is -126. Remember when we said we would always fail in our attempt to map an infinite world onto a finite machine?*
- Most computers use 32 or 64 bits for integers, but there are always infinitely many that aren’t represented

Real Numbers

- Real numbers are numbers with a whole (integer) part and a **fractional** part (either of which may be zero)

  104.32

  0.999999

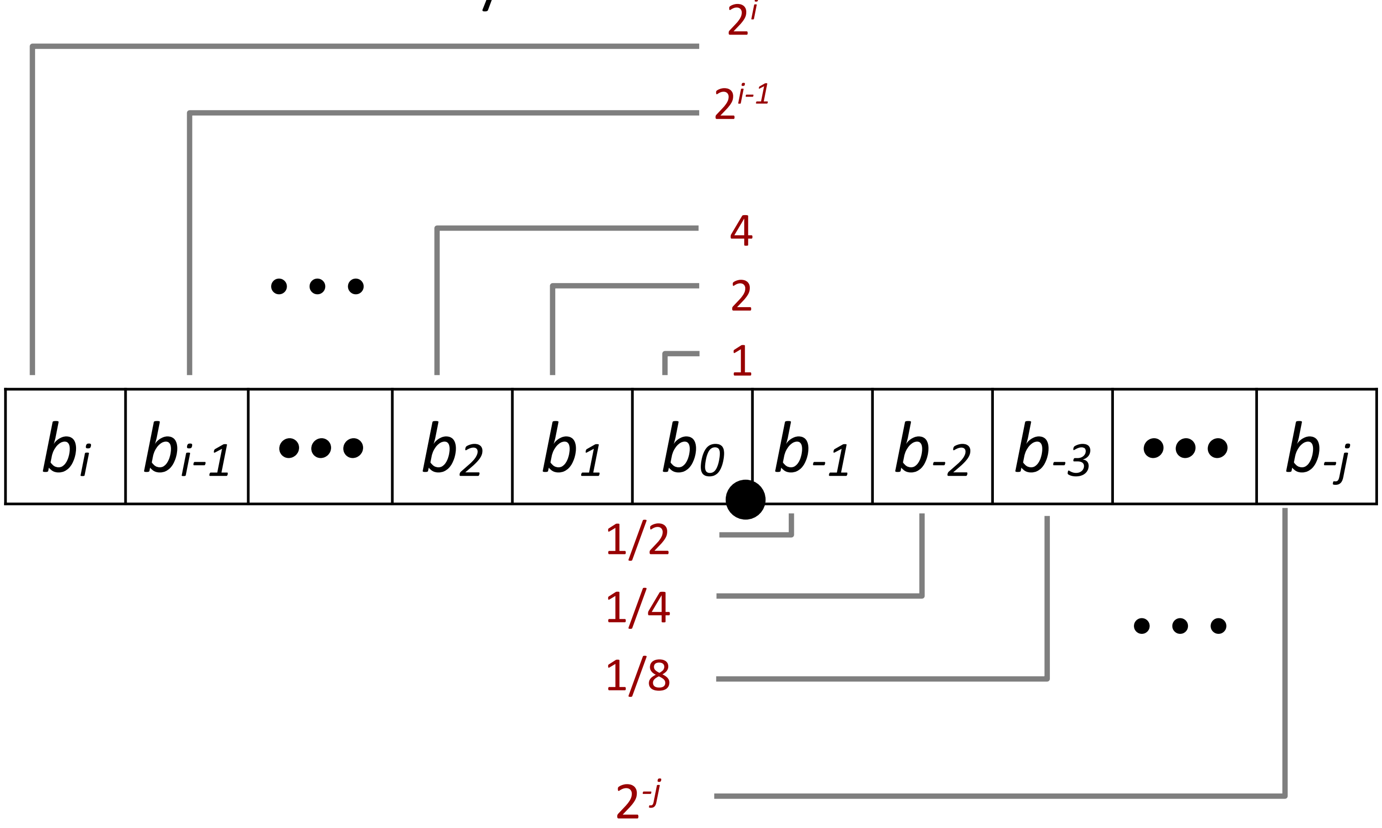
  357.0

  3.14159

- In decimal, positions to the right of the decimal point are the tenths, hundredths, thousandths, etc.:

  10^{-1}, 10^{-2} , 10^{-3} …

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- Value / Representation

Value	Representation
5 3/4	$\mathbf{101.11}_2$
2 7/8	$\mathbf{10.111}_2$
1 7/16	$\mathbf{1.0111}_2$
63/64	$\mathbf{0.111111}_2$

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Fractional Binary Numbers: Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations

Value	Representation
1/3	0.0101010101[01]…$_2$
1/5	0.001100110011[0011]…$_2$
1/10	0.0001100110011[0011]…$_2$

- Limitation #2
  - Just one setting of binary point within the w bits
    - Limited range of numbers (very small values? very large ones?)

Representing Real Numbers

- **Scientific notation**
  - A form of **floating-point representation** in which the decimal point is kept to the right of the leftmost digit

- 12001.32708 is 1.200132708E+4 in scientific notation (E+4 is how computers display x10^4)

- *What is 123.332 in scientific notation?*
- *What is 0.0034 in scientific notation?*

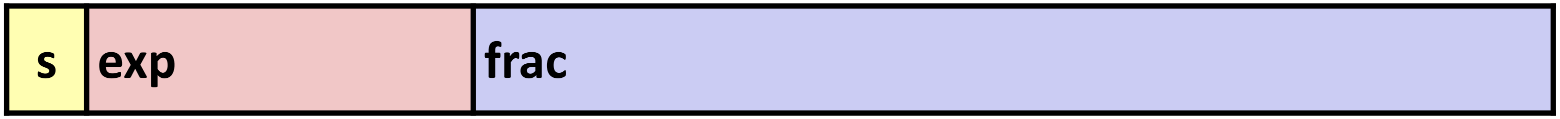
Floating Point Representation

- Numerical Form:

  $(-1)^s * M * 2^E$

  - **Sign** bit s determines whether number is negative or positive
  - **Significand (fraction, mantissa)** M normally a fractional value in range [1,2)
  - **Exponent** E weights value by power of two

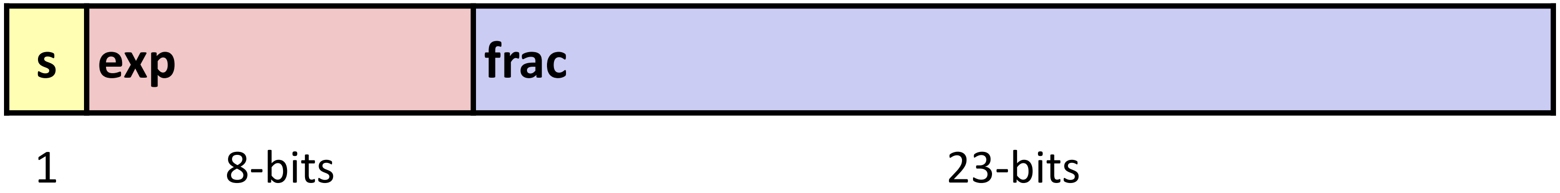
- Encoding



  - MSB is sign bit s
  - **`exp`** field encodes E (Exponent, but is not equal to E)
  - **`frac`** field encodes M (Mantissa, but is not equal to M)

IEEE Floating Point Number

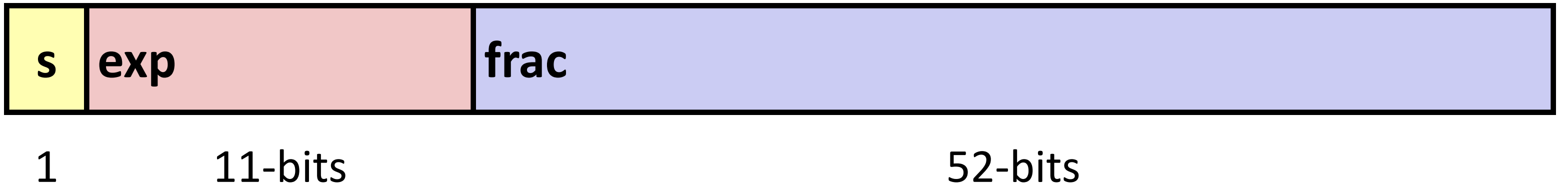
- Single precision: 32 bits



s exp frac

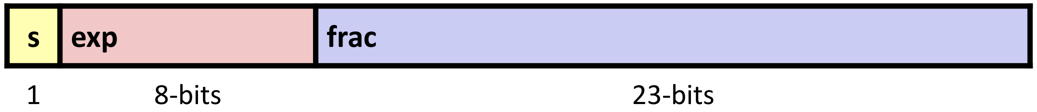
1 8-bits 23-bits

- Double precision: 64 bits



s exp frac

1 11-bits 52-bits



$$v = (-1)^s \boldsymbol{M}\, 2^{\boldsymbol{E}}$$

- Value: float F = 15213.0;
  - Numerical form 15213_{10} = 11101101101101_2

    = $\mathbf{1.1101101101101_2 \ x \ 2^{13}}$

- **Is** $(x + y) + z = x + (y + z)$ **?**

(1e20 + -1e20) + 3.14 --> 3.14

1e20 + (-1e20 + 3.14) --> ??

Representing Text

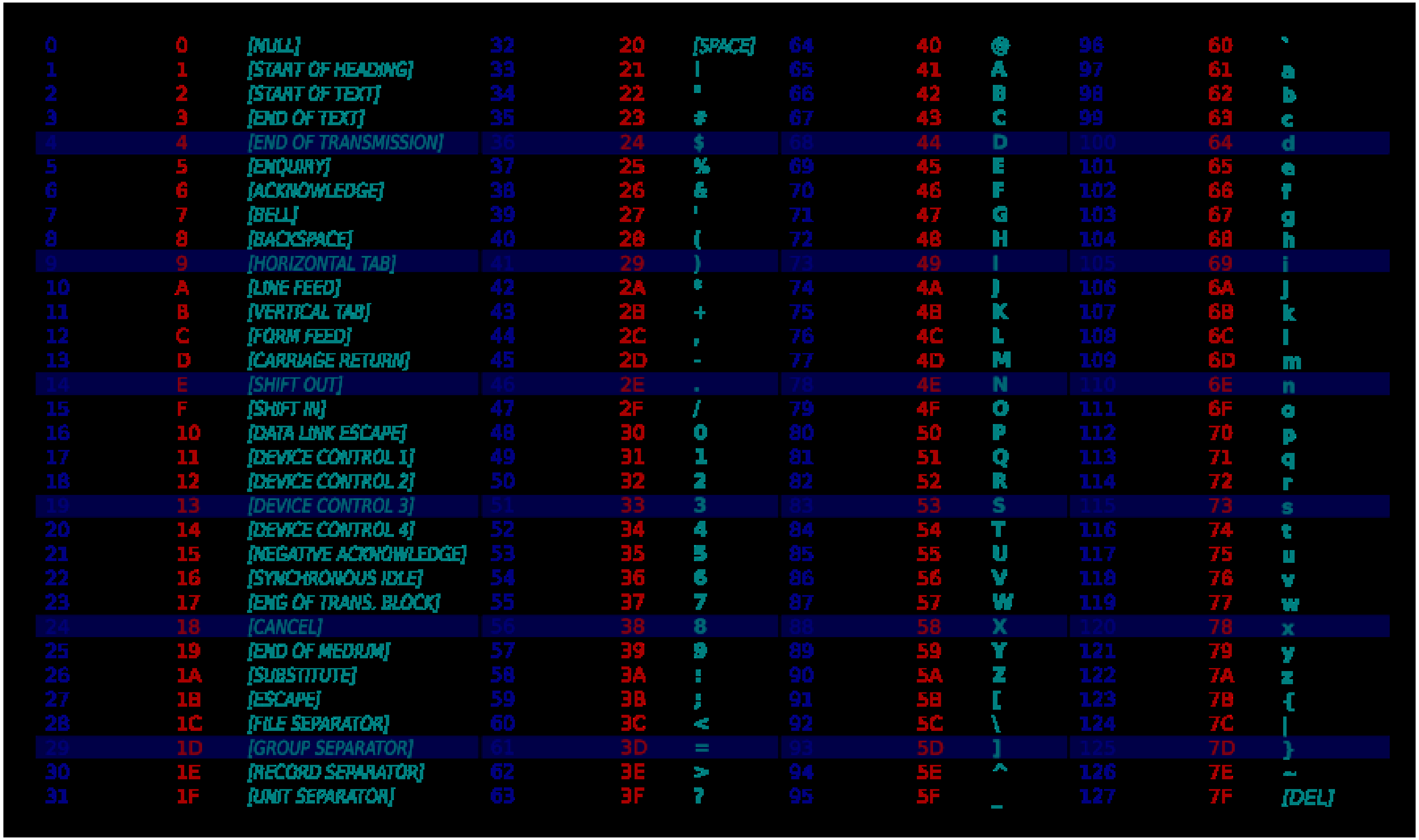
- The number of characters to represent is finite, so list them all and assign each a binary string

- **Character set**
  - A list of characters and the codes used to represent each one
  - Computer manufacturers agreed to standardize

The ASCII Character Set

- ASCII stands for *American Standard Code for Information Interchange*
- ASCII originally used **seven** bits to represent each character, allowing for 128 unique characters
- Later extended ASCII evolved so that all eight bits were used
- How many characters could be represented?

ASCII Character Set Mapping



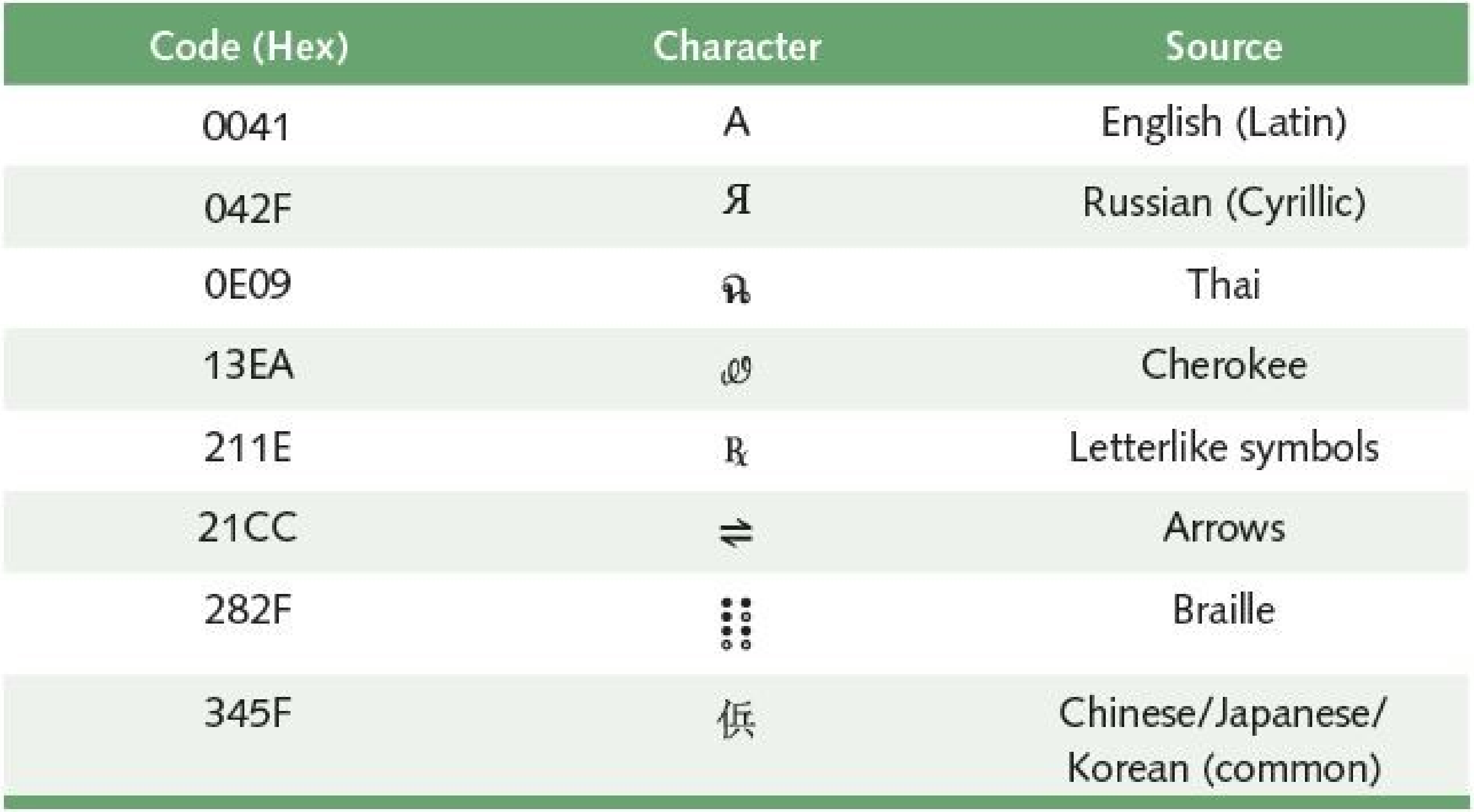
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

https://simple.wikipedia.org/wiki/ASCII

The Unicode Character Set

- ASCII is not enough for international use
- One Unicode mapping uses 16 bits per character
- The first 256 characters correspond exactly to the extended ASCII character set

The Unicode Character Set



Code (Hex)	Character	Source
0041	A	English (Latin)
042F	Я	Russian (Cyrillic)
0E09	ฉ	Thai
13EA	Ꮺ	Cherokee
211E	℞	Letterlike symbols
21CC	⇌	Arrows
282F	⠯	Braille
345F	㑟	Chinese/Japanese/ Korean (common)

FIGURE 3.6 A few characters in the Unicode character set

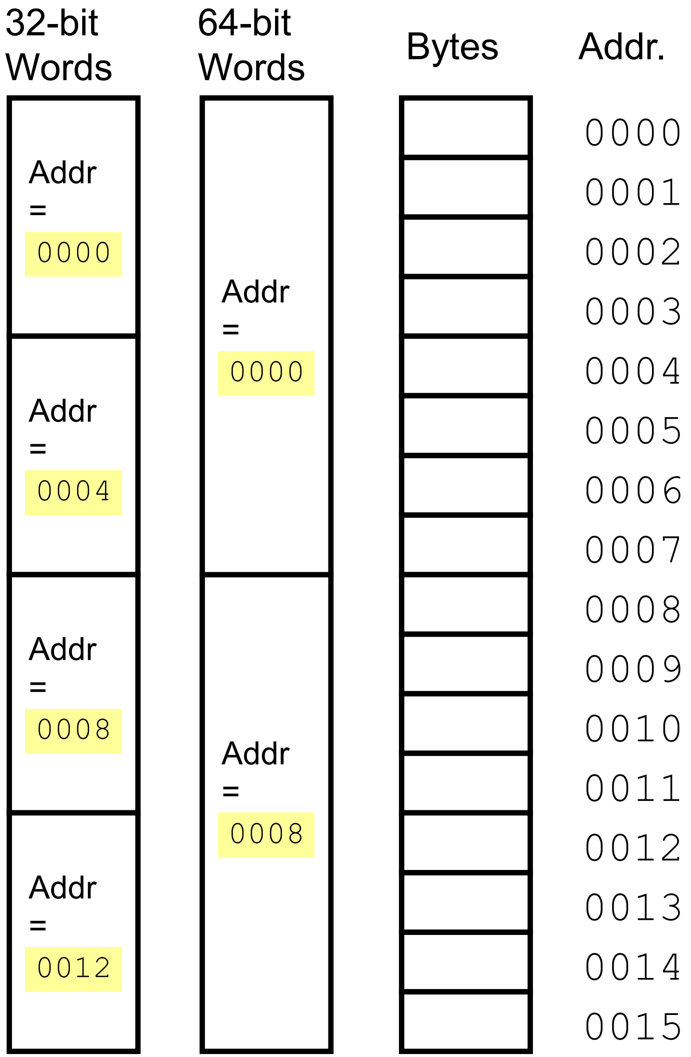
Example Data Representations

- Sizes of C Objects (in Bytes)

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	**4**	**8**	8

Accessing **Words** in Memory

- Addresses specify **Byte** locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



32-bit Words

Addr = 0000

Addr = 0004

Addr = 0008

Addr = 0012

64-bit Words

Addr = 0000

Addr = 0008

Bytes

Addr.

0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Boolean Algebra

- Operate on **Bit Vectors**
    - Operations applied bit-wise

```
   01101001     01101001     01101001
&  01010101   | 01010101   ^ 01010101   ~ 01010101

   01000001     01111101     00111100     10101010
```

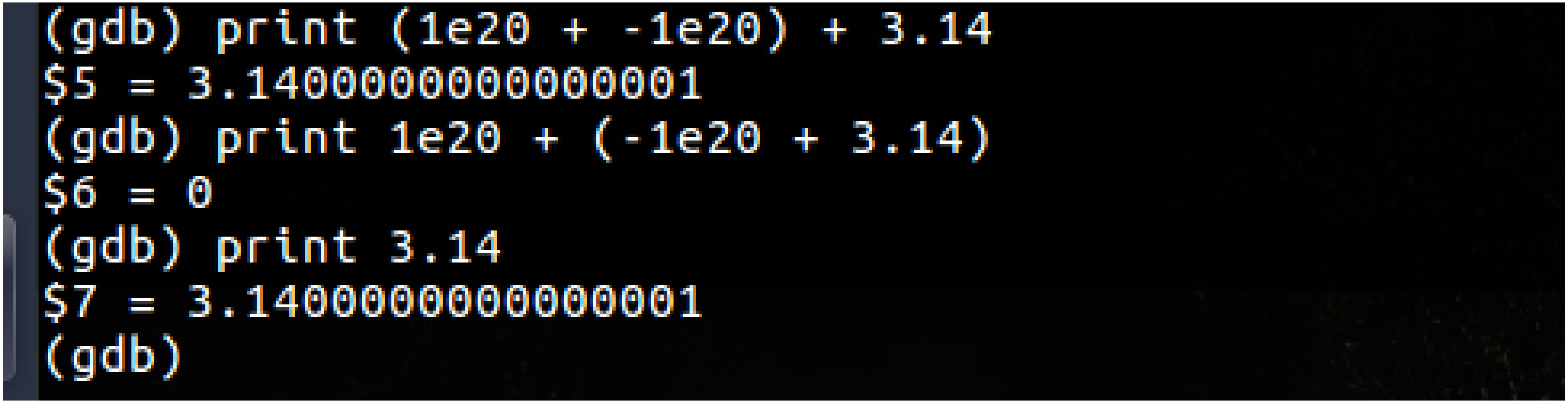
- All of the Properties of Boolean Algebra Apply

Discussion

(1e20 + -1e20) + 3.14 --> 3.14

1e20 + (-1e20 + 3.14) --> 0

- Why do we have these results ?



```
(gdb) print (1e20 + -1e20) + 3.14
$5 = 3.1400000000000001
(gdb) print 1e20 + (-1e20 + 3.14)
$6 = 0
(gdb) print 3.14
$7 = 3.1400000000000001
(gdb)
```