CHAPTER

1

INTRODUCTION

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
TP. HỒ CHÍ MINH

cdio 4.0

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

fit@hcmus

OUTLINE

- Common systems of number: convert from a system of number to the other (2,10,16)
- Integer representation & comparison
- Operation with integer
- Dealing with overflow
- Character (ASCII, Unicode)
- Data format in C program
- Heterogeneous data structures
- Data alignment

Number systems

- An unsigned integer with n digits in q-base system is represented in general formular:

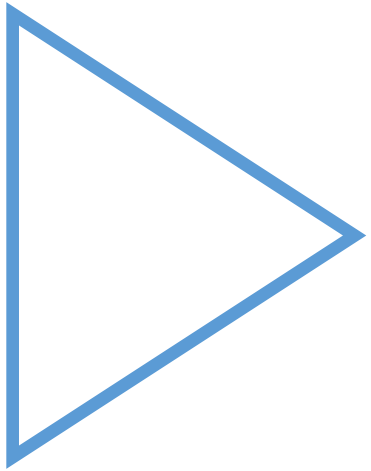
$$x_{n-1}\ldots x_1 x_0 = x_{n-1}.q^{n-1} + \ldots + x_1.q^1 + x_0.q^0$$

  - The base system commonly represented in the computer is base 2

cdio 4.0

fit@hcmus

Number systems



Watch these videos:

- Introduction to number system and library
- Hexadecimal number system
- Convert from decimal to binary
- Converting larger number from decimal to binary
- Converting from decimal to hexadecimal representation
- Converting directly from binary to hexadecimal

Unsigned Integers representation

- Represent positive values such as length, weight, ASCII code, color code, ...
- The values of an unsigned integer is the magnitude of its underlying binary pattern
- Ex: Suppose that n =8, the binary representation 10000001

Absolute value is 10000001-> 129 (decimal)

Hence, the integer is **129** (decimal)

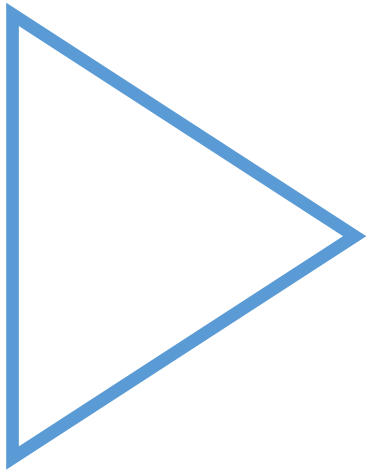
cdio 4.0
fit@hcmus

Unsigned Integers representation

- The n-bits binary pattern can represent the values from **0 to 2^n -1**

n	Minimum value	Maximum Value
8	0	2^8 -1 = 255
16	0	2^{16} -1 = 65535
32	0	2^{32}-1 = 4294967295
64	0	2^{64} -1 = 18446744073709551615

Unsigned Integers representation



Read the explanation in this link and watch these videos:

- The binary number system
- Converting the decimal numbers to binary (remind)
- Pattern in binary numbers

Take a practice

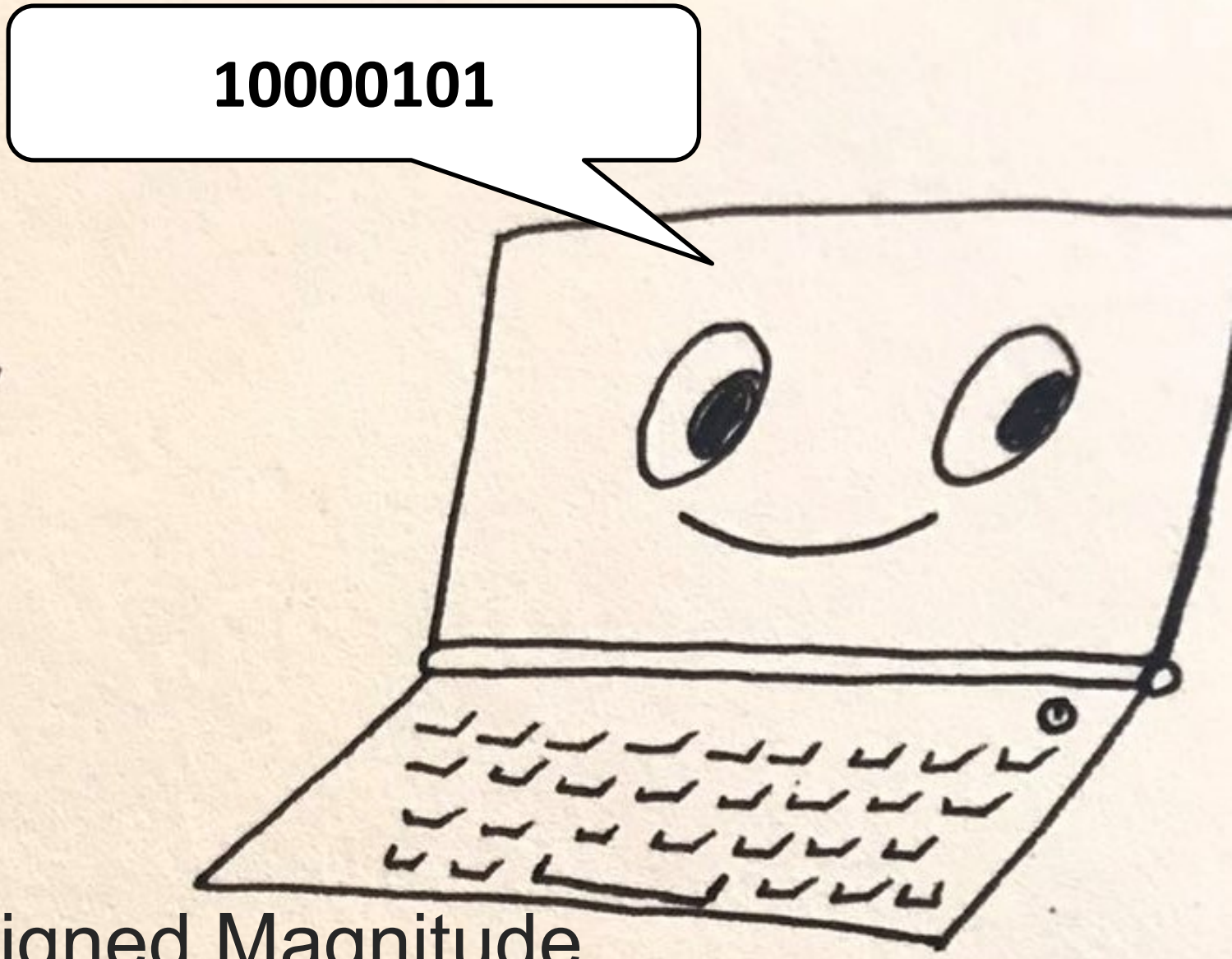
Signed Integers representation

Signed integers can represent zero, positive integers, as well as negative integers

Four representation schemes are available for signed integers:

- Sign-Magnitude representation
- 1's Complement representation
- 2's Complement representation
- Bias (k-excess)

Signed numbers in the computer system are represented in the 2’s Complement scheme



Signed Magnitude

Sign-Magnitude representation

- The most-significant bit (MSB) is the sign bit:

  0 -> positive integer

  1 -> negative integer

- The remaining n-1 bits represents the magnitude (absolute value) of the integer (n is the length of bit pattern)

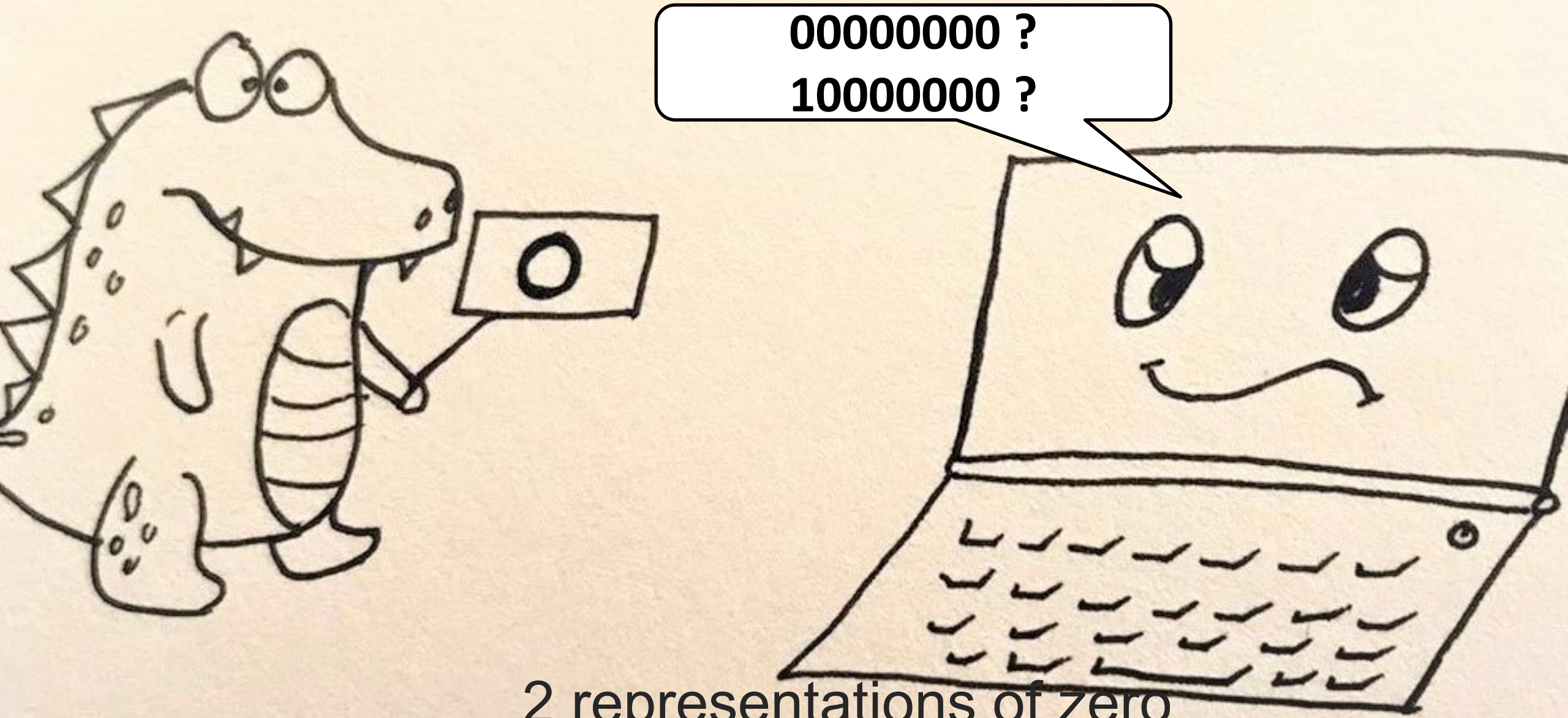
Sign-Magnitude representation

- Ex: Suppose that n =8, the binary representation 10000001

Sign bit is 1 -> negative number

Absolute value is 0000001-> 1 (decimal)

Hence, the integer is **-1** (decimal)



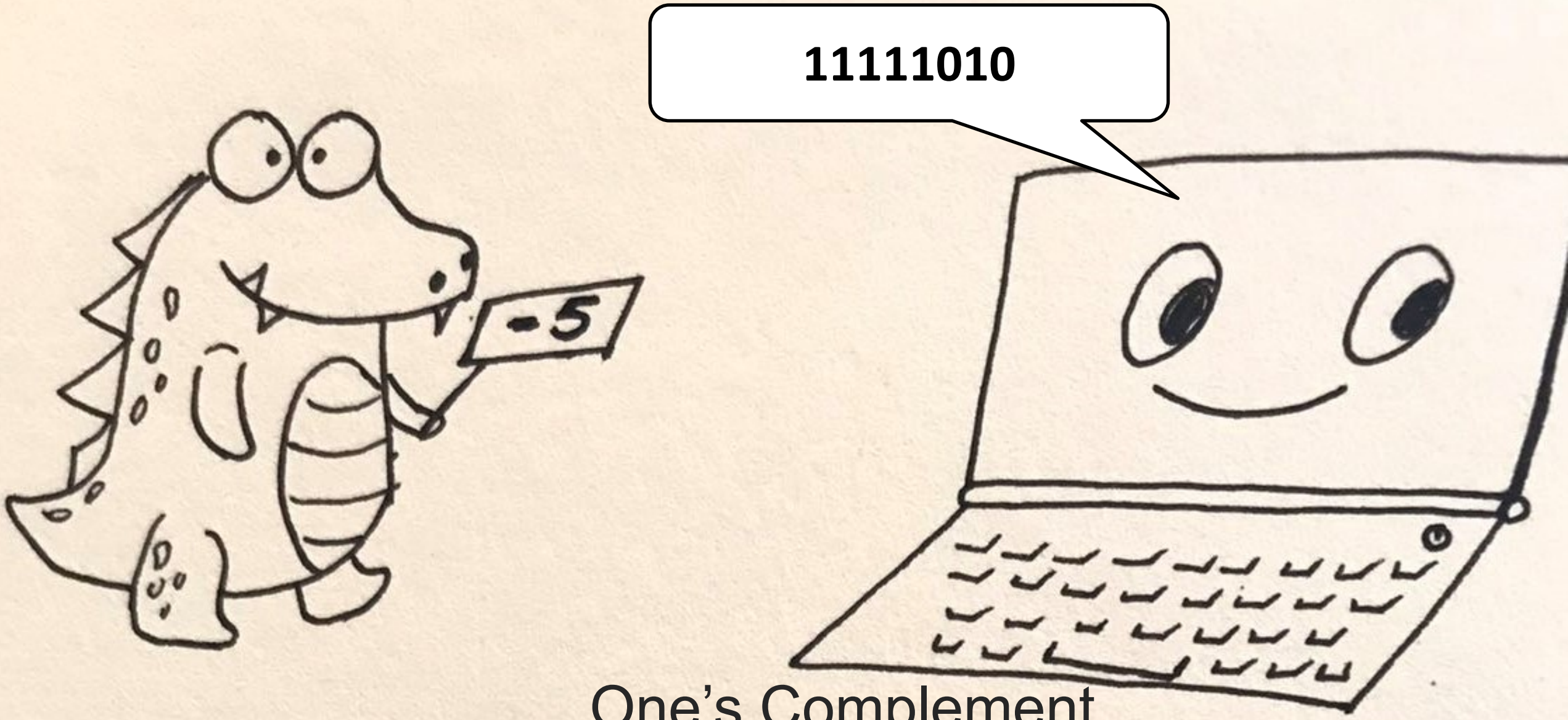
2 representations of zero

Sign-Magnitude representation

- The n-bits binary pattern can represent the values from **$(-2^{n-1})+1$ to $2^{n-1} - 1$**

Ex: Suppose that ***n = 8***, the range of values is ***-127 to 127***

- Positive number and negative number differ MSB value(sign bit), the absolute value are the same
- There are two representations for the number zero, which could lead to inefficiency and confusion.



One’s Complement

One's Complement representation

- The most-significant bit (MSB) is the sign bit:

  0 -> positive integer

  1 -> negative integer

- The remaining n-1 bits represents the magnitude of the integer (n is the length of bit pattern) as follow:

  positive integers: the absolute value of the integer is equal to the magnitude of the (n-1)-bit binary pattern

  negative integers: the absolute value of the integer is equal to the magnitude of the *complement (inverse)* of the (n-1)-bit binary pattern

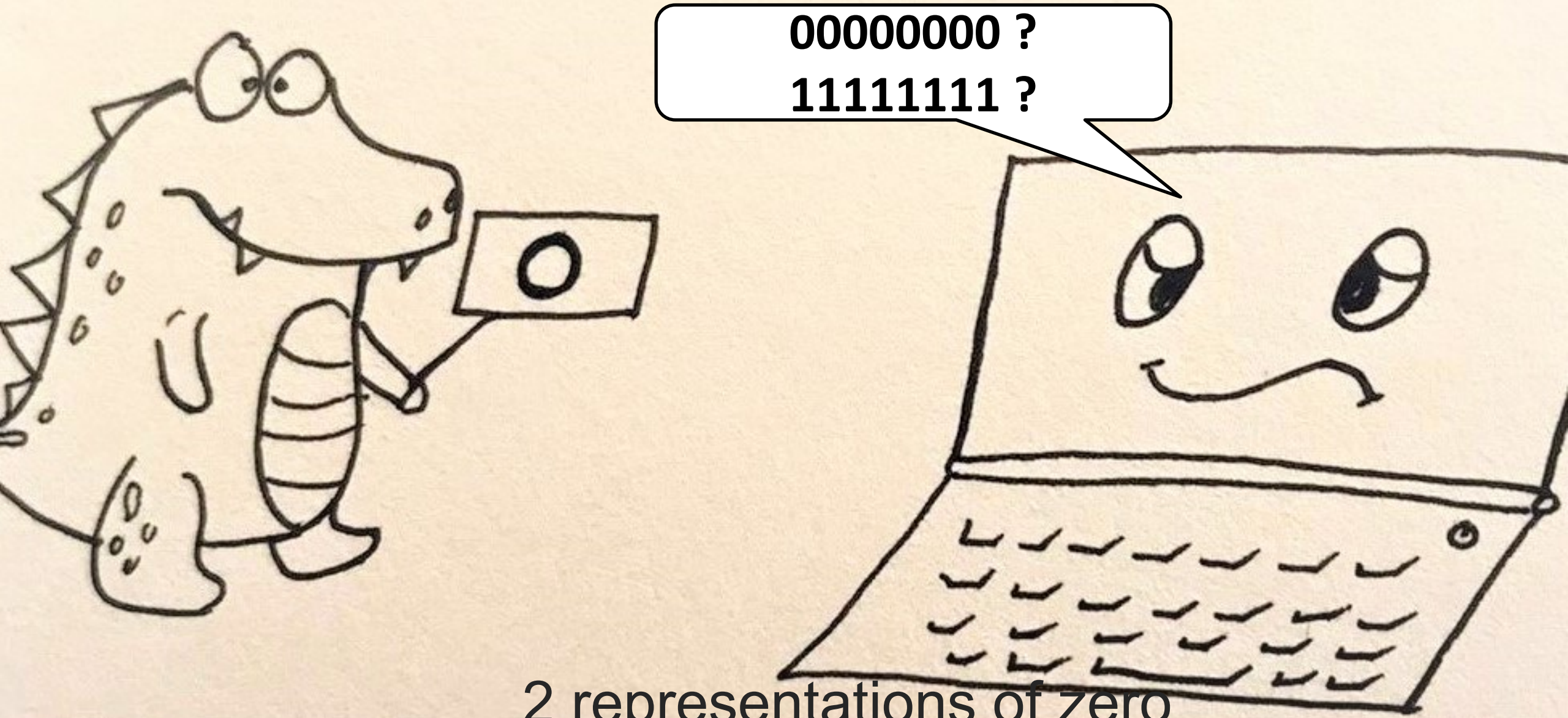
One's Complement representation

Ex: Suppose that n =8, the binary representation 10000001

Sign bit is 1 -> negative number

Absolute value is the complement of 0000001-> 1111110 -> 126 (decimal)

Hence, the integer is **-126** (decimal)



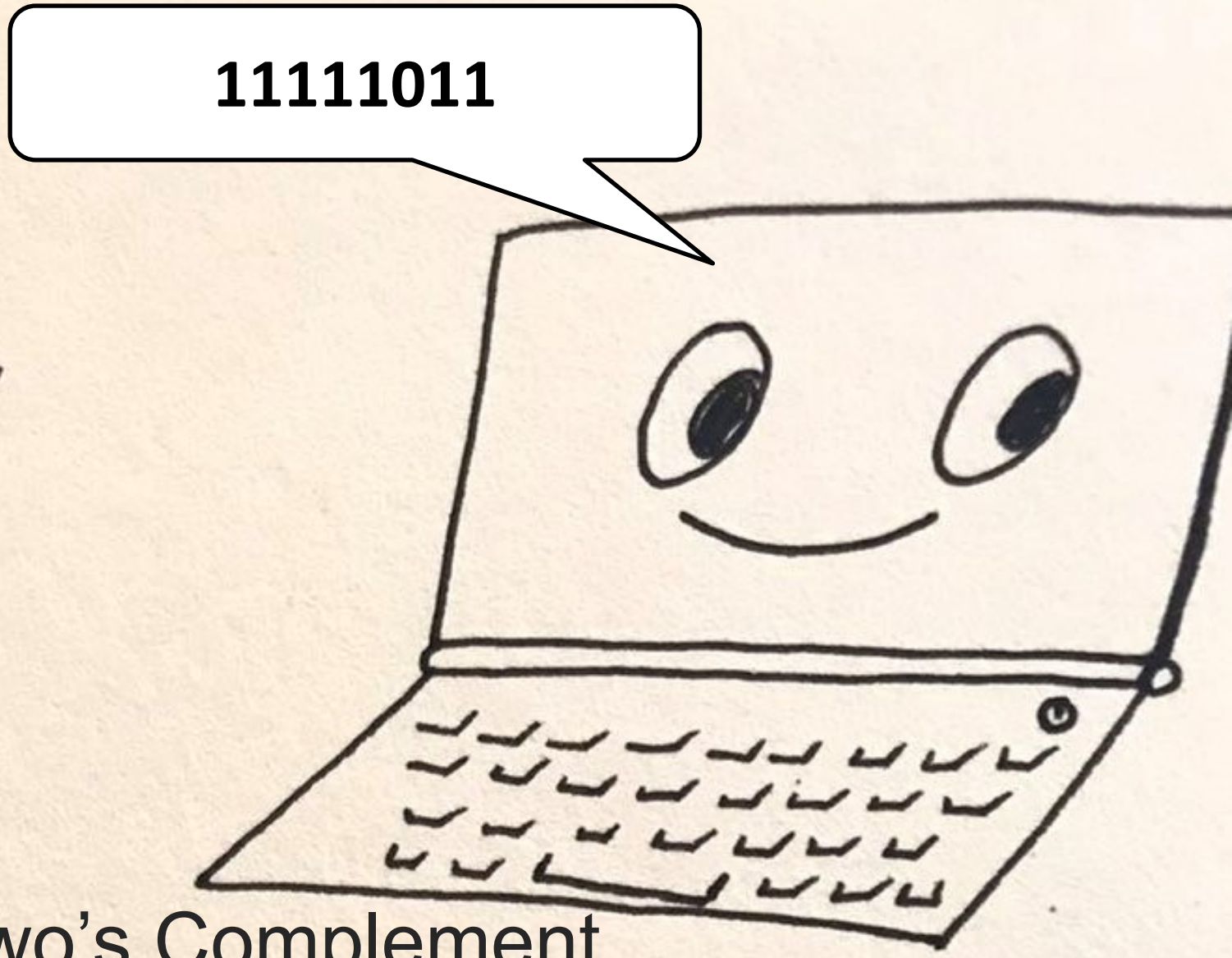
2 representations of zero

One's Complement representation

- The n-bits binary pattern can represent the values from $(-2^{n-1})+1$ to $2^{n-1}-1$

Ex: Suppose that n = 8, the range of values is -127 to 127

- There are two representations for the number zero, which could lead to inefficiency and confusion.
- The positive integers and negative integers need to be processed separately



Two’s Complement

cdio 4.0
fit@hcmus

Two's Complement representation

- The most-significant bit (MSB) is the sign bit:

  0 -> positive integer

  1 -> negative integer

- The remaining n-1 bits represents the magnitude of the integer (n is the length of bit pattern) as follow:

  positive integers: the absolute value of the integer is equal to the magnitude of the (n-1)-bit binary pattern

  negative integers: the absolute value of the integer is equal to the magnitude of the *complement (inverse)* of the (n-1)-bit binary pattern *plus one*

Two's Complement representation

Ex: Suppose that n =8, the binary representation 10000001

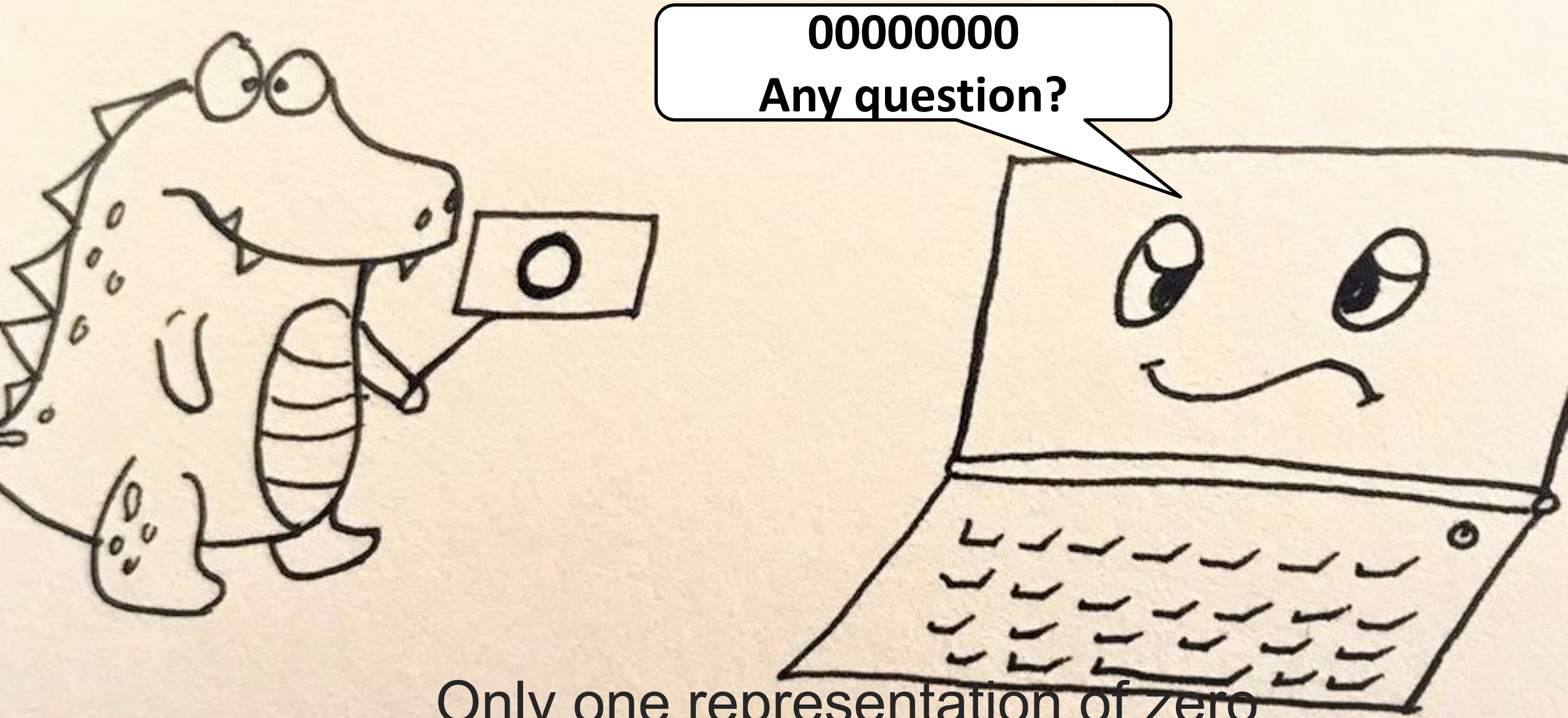
Sign bit is 1 -> negative number

Absolute value is the complement of 0000001 + 1

-> 1111110 + 1

-> 127 (decimal)

Hence, the integer is **-127** (decimal)



Only one representation of zero

Two's Complement representation

- The n-bits binary pattern can represent the values from -2^{n-1} to $(2^{n-1})-1$

Ex: Suppose that n = 8, the range of values is -128 to 127

- There is one representations for the number zero
- The two's complement number of N is the negative form of N

Ex: How to represent the -5 (decimal) in binary:

The bit binary pattern of 5 is 00000101

The two's complement number of 5 is 11111010 plus 1

-> 11111011

Two's Complement representation

Ex: Suppose that n =8, the binary representation 10000001

Sign bit is 1 -> negative number

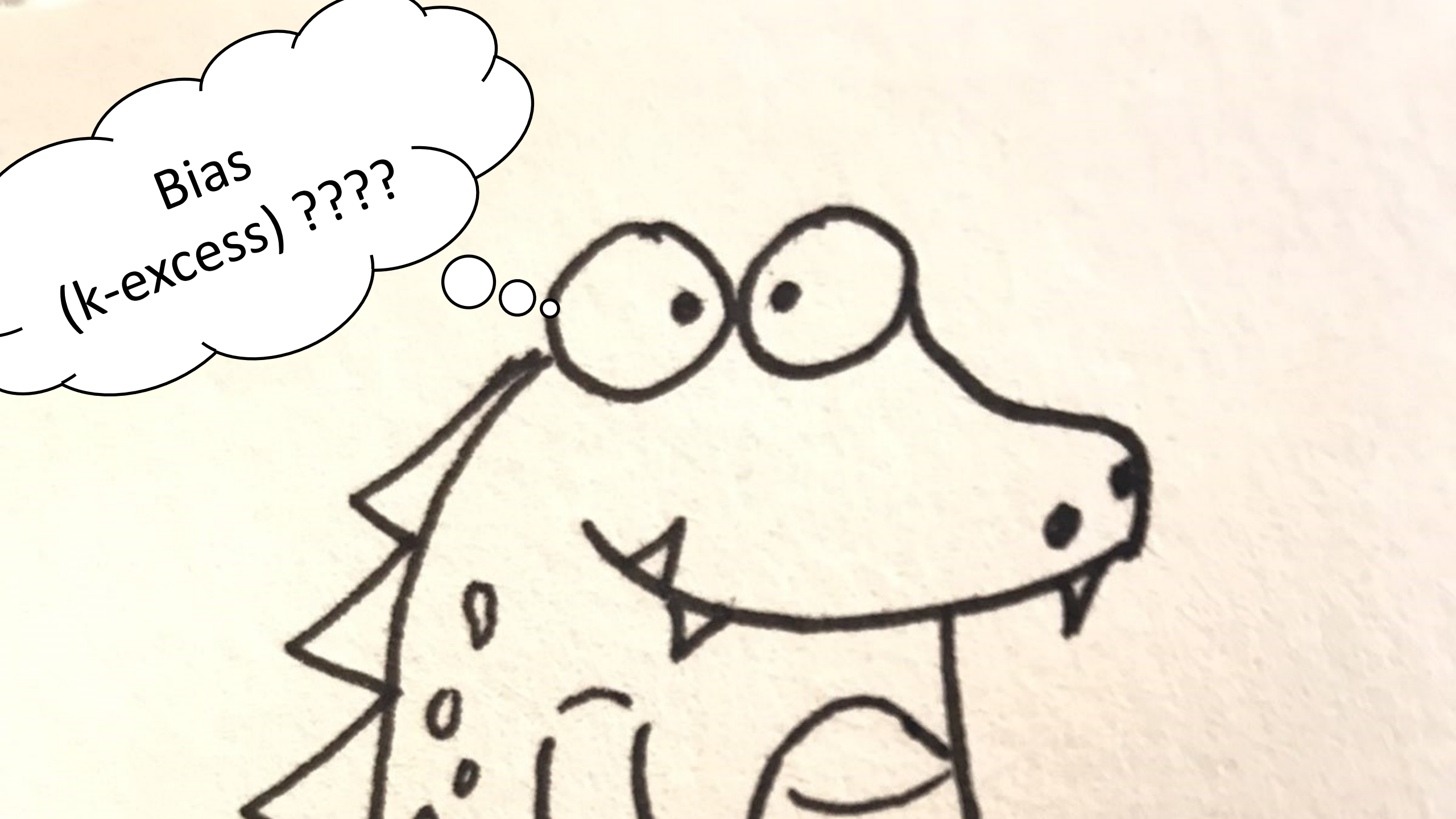
Absolute value is ***the complement*** of 0000001 + 1

-> 1111110 + 1

-> 1111111

-> 127 (decimal)

Hence, the integer is **-127** (decimal)


Bias
(k-excess) ????

Biased (K-excess)

- Choose K (a positive integer) to allows operations on the biased numbers to be the same as for unsigned integers, but represents both positive and negative values
- The remaining n bits represents the biased value (n is the length of bit pattern)
- The biased value is equal to the magnitude of the n-bits binary pattern
- The absolute value is equal to the bias value subtract/add to K

  positive integers: N - K

  negative integers: N + K

Biased (K-excess)

Ex: Suppose that n =8, K =128, the binary representation of N is 10000001 Biased value is 129 decimal (greater than K)

-> N is positive integer

-> Absolute value is N – K = 129 – 128 = 1 (decimal)

Hence, the integer is **1**(decimal)

Biased (K-excess)

- The n-bits binary pattern can represent the values from -2^{n-1} to $2^{n-1} - 1$

Ex: Suppose that ***n = 8***, ***K = 128***, the range of values is ***-128 to 127***

- There is one representations for the number zero: 10000000
- Biased representations are now primarily used for the exponent of *floating-point numbers*

Biased (K-excess)

Ex: Suppose that n =8, K=128, how to represent a number in binary

Positive number: N = 25 (decimal)

N + K = 25 + 128 = 153

The bit binary pattern is -> 10011001

Negative number: N = -25 (decimal)

N + K = -25 + 128 = 103

The bit binary pattern is -> 01100111

Integer Operations

- Logical operations

  AND, OR, XOR, NOT

  SHL, SHR, SAR
- Arithmetic operation

  Add/Subtract

  Multiply

  Division

Logical Operations

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	0	1
	1	0

```
AND 11010011
    00001111
    --------
    00000011
```

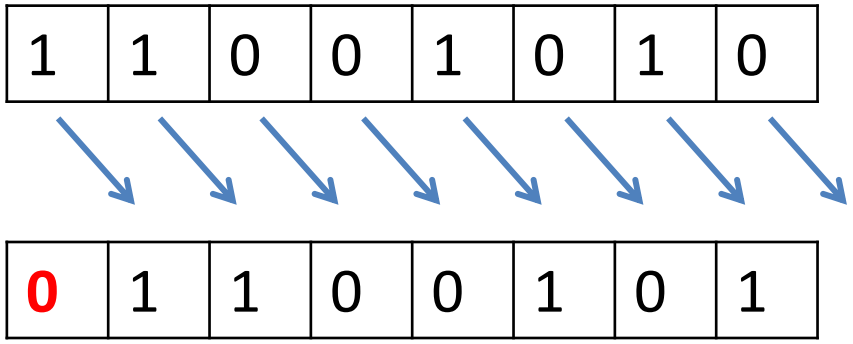
```
OR  00000011
    01100000
    --------
    01100011
```

```
XOR 01100011
    01100011
    --------
    00000000
```

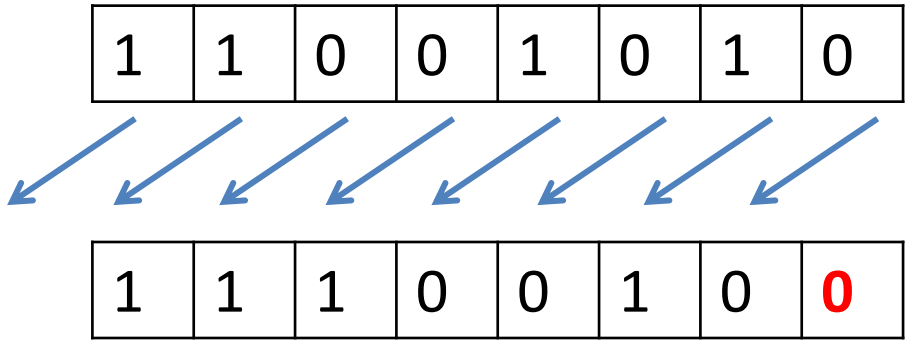
```
NOT 11010011
  = 00101100
```

Shift Operations

A logical shift moves bits to the left/ right and places a 0's form in the vacated bit on either end. The bits "fall off" will be discarded.



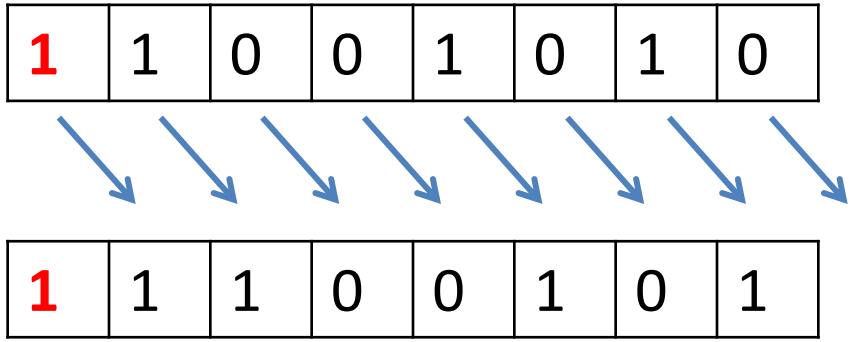
Shift Right Logical (SHR)



Shift Left Logical (SHL)

Shift Operations

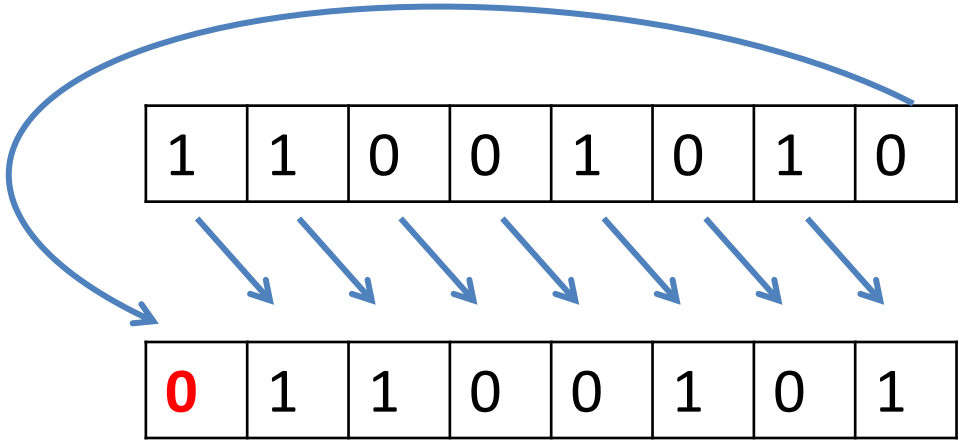
An arithmetic shift right preserves the sign bit



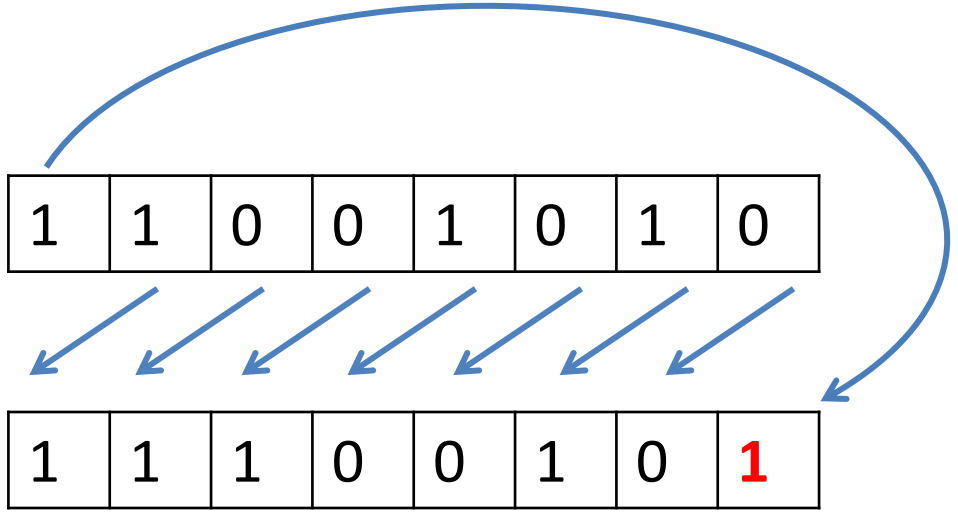
Shift Right Arithmetic (SAR)

Shift Operations

A circular shift (rotate) places the bit shifted out of one end into the vacated position on the other end.



Rotate Right (ROR)



Rotate left (ROL)

Advanced

x SHL y = x . 2^y

x SAR y = x / 2^y

AND uses to switch off a bit(AND with 0 = 0)

OR uses to switch on a bit (OR with 1 = 1)

XOR, NOT uses to reverse a bit (bit i XOR with 1 = NOT(i))

x AND 0 = 0

x XOR x = 0

Advanced

Suppose that x is an integer:

- Get the value of bit i: (x SHR i) AND 1
- Set the value 1's of bit i: (1 SHL i) OR x
- Set the value 0's of bit i : NOT(1 SHL i) AND x
- Reverse bit i: (1 SHL i) XOR x

Integer Operations

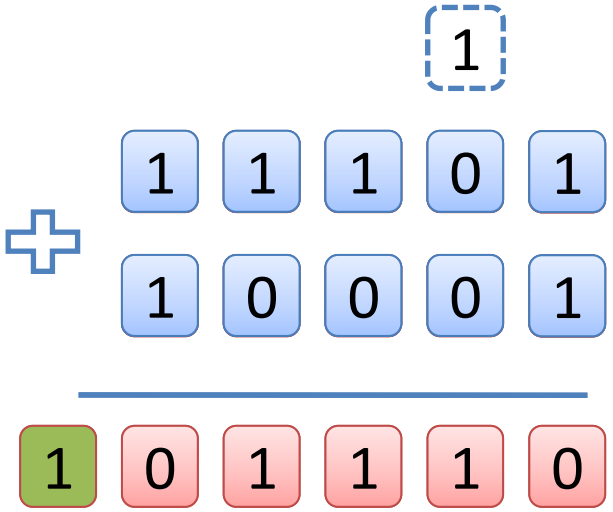
- Logical operations
  - AND, OR, XOR, NOT
  - SHL, SHR, SAR
- Arithmetic operation
  - Add/Subtract
  - Multiply
  - Division

Add Operation

Rule:

+	0	1
0	0	1
1	1	10

Ex:



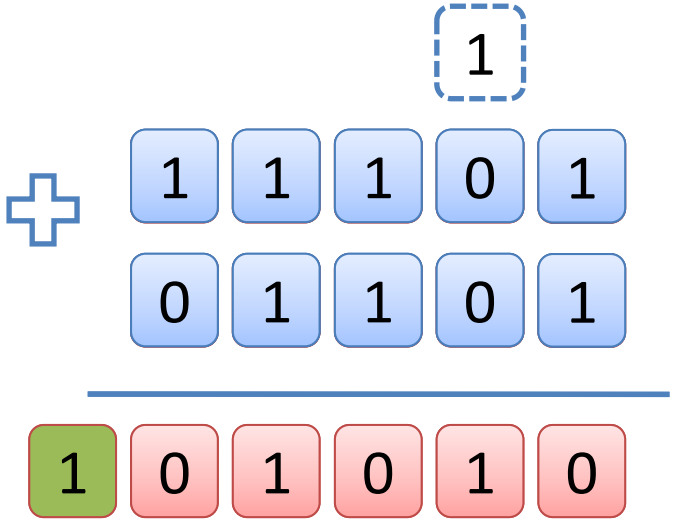
1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Subtract Operation

Rule:

A – B = A + (-B) = A + (the 2's complement of B)

Ex: 11101 – 10011 = 11101 + 01101



```
      0010 =  2
     +1001 = −7
      1011 = −5

(a) M =  2 = 0010
    S =  7 = 0111
   −S =     1001
```

```
      0101 =  5
     +1110 = −2
     10011 =  3

(b) M =  5 = 0101
    S =  2 = 0010
   −S =     1110
```

```
      1011 = −5
     +1110 = −2
     11001 = −7

(c) M = −5 = 1011
    S =  2 = 0010
   −S =     1110
```

```
      0101 = 5
     +0010 = 2
      0111 = 7

(d) M =  5 = 0101
    S = −2 = 1110
   −S =     0010
```

```
      0111 = 7
     +0111 = 7
      1110 = Overflow

(e) M =  7 = 0111
    S = −7 = 1001
   −S =     0111
```

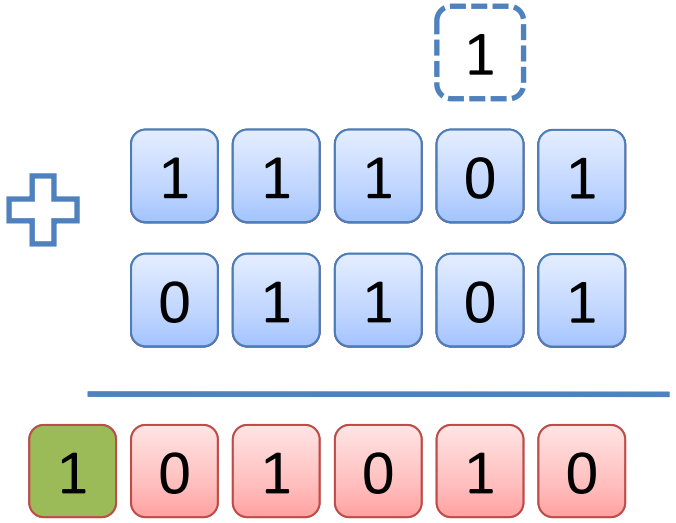
```
      1010 = −6
     +1100 = −4
     10110 = Overflow

(f) M = −6 = 1010
    S =  4 = 0100
   −S =     1100
```

Subtract Operation

Rule:

A – B = A + (-B) = A + (the 2's complement of B)

Ex: 11101 – 10011 = 11101 + 01101



Relational of integer and two's complement addition.

When $x + y < -2^{w-1}$, there is a *negative* overflow

When $x + y > 2^{w-1} - 1$, there is a *positive* overflow

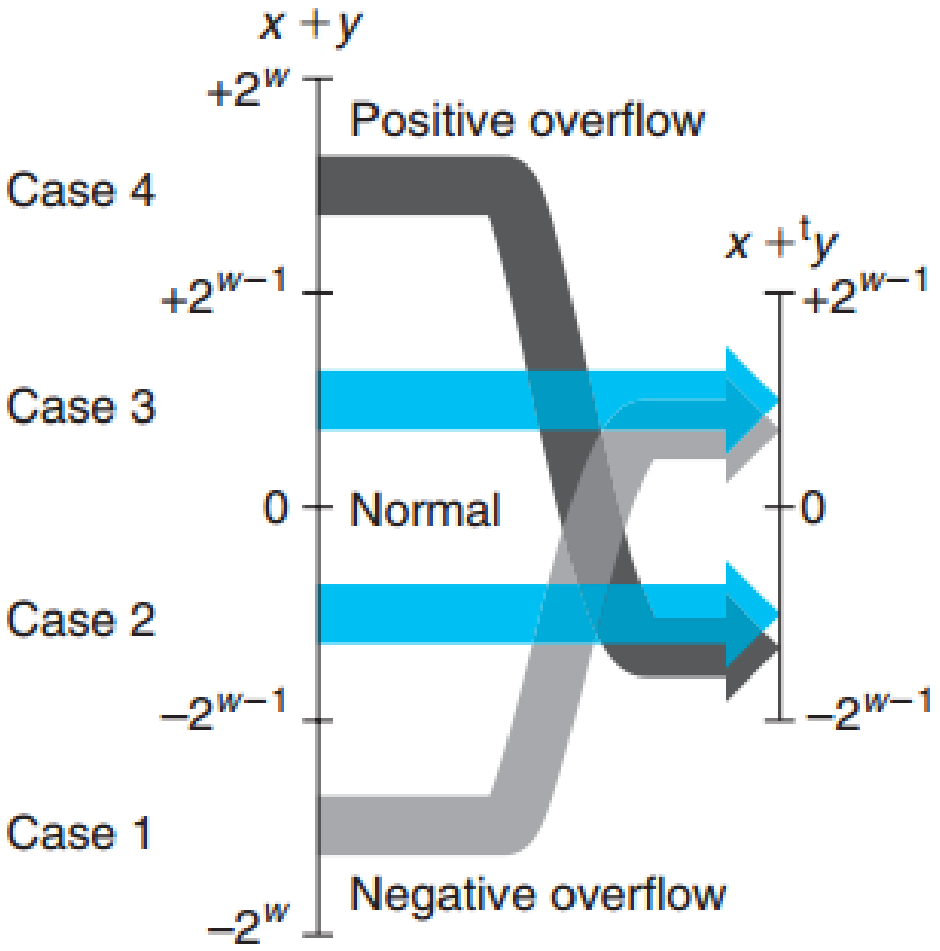


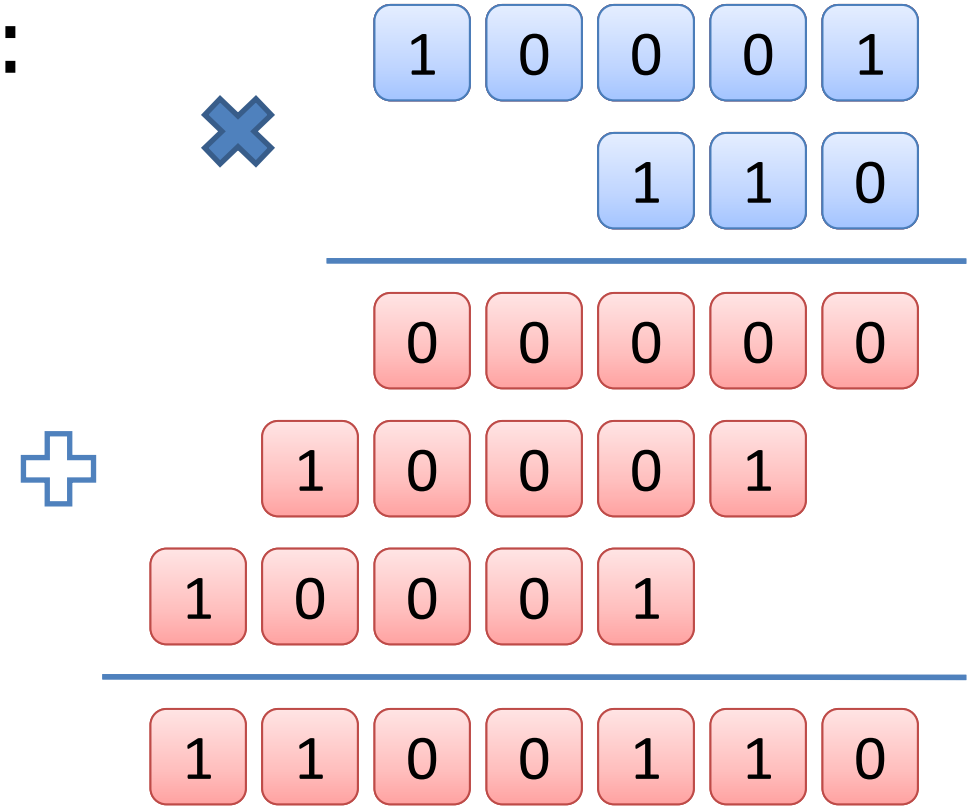
Photo by Chap2, Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011

Multiply Operation

Rule:

×	0	1
0	0	0
1	0	1

Ex:



Multiply Operation

```
      1011      = 11
    x 1101      = 13
----------
      1011
     0000
    1011
   1011
----------
  10001111      = 143
```

```
      1011
    x 1101
----------
  00000000
+     1011
----------
  00001011
+    0000
----------
  00001011
+   1011
----------
  00110111
+ 1011
----------
  10001111
```

Multiply Algorithm

- Suppose that Q's binary pattern has n-bits length, M x Q

M: multiplicand

Q: multiplier

- Variable definition :

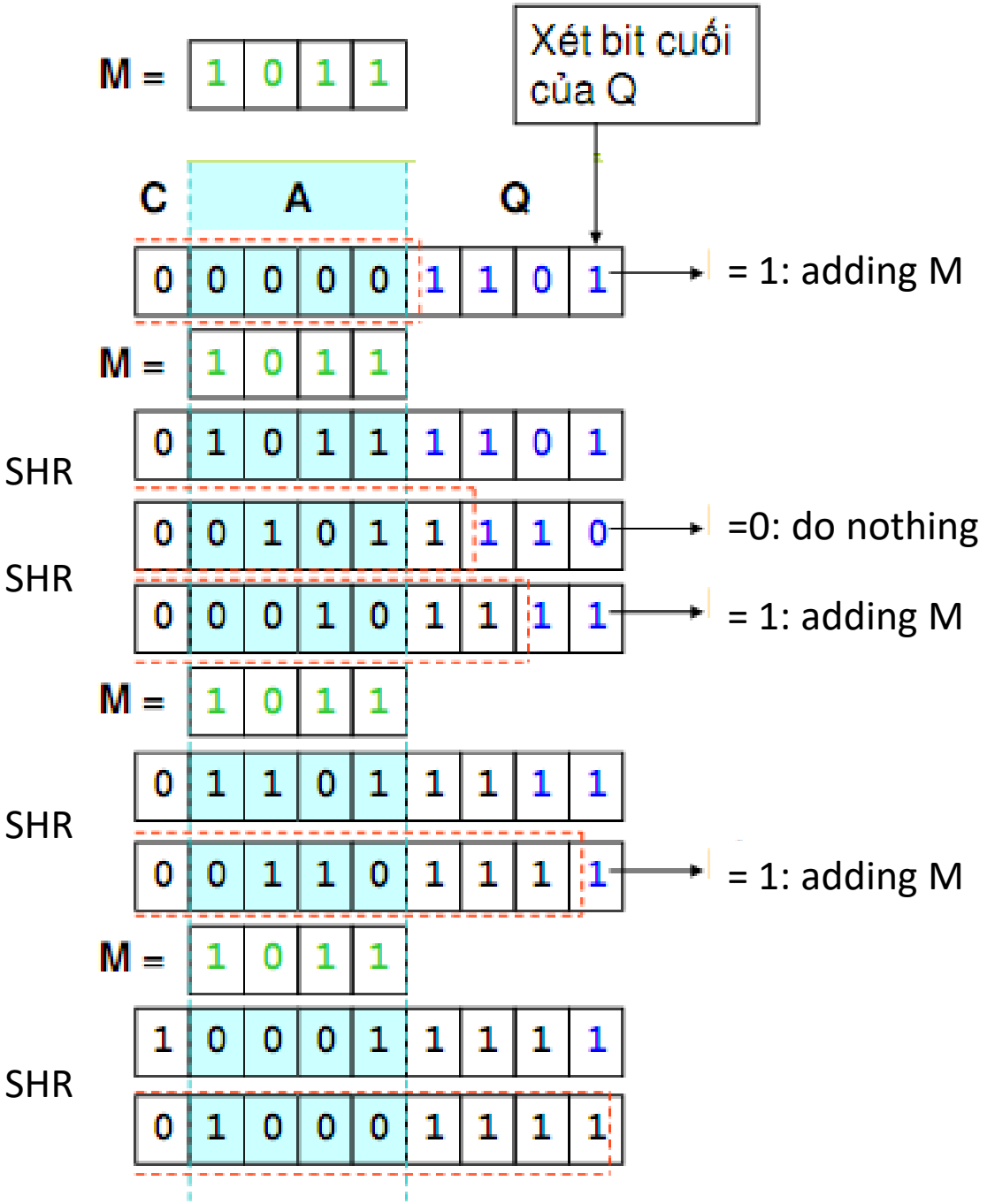
  C (1 bit): carry bit

  A (n bit): a part of the result ([C, A, Q]: product)

  [C, A] (n + 1 bit) ; [C, A, Q] (2n + 1 bit): considered as compound registers

Multiply Algorithm

```
Initialize: [C, A] = 0; k = n
While (k > 0)
{
If LSB bit of Q equal to 1 then
    (A + M) -> [C, A]
SHR [C, A, Q] 1 bit
k = k – 1
}
Return [C,A,Q]
```



Booth's Multiplication Algorithm

- A multiplication algorithm that multiplies two signed binary numbers in 2's complement notation
- Suppose that Q's binary pattern has n-bits length, M x Q

M: multiplicand

Q: multiplier

- Variable definition :

A (n bit): a part of the result ([A, Q]: product)

Q_0 (1 bit): the LSB bit of Q

$[Q,Q_{-1}]$ (n + 1 bit)

```
Initialize: A = 0; k = n; Q-1 = 0
#Add 1-bit Q-1 in the end of Q
While (k > 0)
{
  If Q0Q-1
  {
       = 10 then A - M -> A
       = 01 thì A + M -> A
       = 00, 11 then A -> A
  # Ignore any overflow
  }
  SAR[A, Q, Q-1] 1 bit
  k = k - 1
  }
  Return [A, Q]
```

Ex: Suppose that n = 4, M = 7, Q = -3

		A	Q	Q_{-1}	M
	Initialize	0000	1101	0	0111
k = 4	A = A-M	1001	1101	0	0111
	SAR	1100	1110	1	0111
k = 3	A = A+M	0011	1110	1	0111
	SAR	0001	1111	0	0111
k = 2	A = A-M	1010	1111	0	0111
	SAR	1101	0111	1	0111
k = 1	SAR	1110	1011	1	0111

A	Q	M = 0011
0000	0111	Initial value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Subtract
0001	0010	Restore

(a) (7)/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	Shift
1101		Add
0000	1110	Restore
0001	1100	Shift
1110		Add
0001	1100	Restore
0011	1000	Shift
0000		Add
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Add
0001	0010	Restore

(b) (7)/(−3)


Representation of -5.5 ?

Fixed point numbers

Represent 123.375_{10} in 2-base system ?

Idea: Represent the integer part and fraction part separately

Integer part: use 8-bits for representation. Range of values is [0, 255] (decimal)

$123_{10} = 64 + 32 + 16 + 8 + 2 + 1 = 0111\ 1011_2$

Fraction part: use 8-bits for representation.

$0.375 = 0.25 + 0.125 = 2^{-2} + 2^{-3} = 0110\ 0000_2$

Signed fixed point	Signed bit	Integer (8-bits)	Fraction (8-bits)
	0	0111 1011	0110 0000

Formular: $$x_{n-1}x_{n-2}...x_0.x_{-1}x_{-2}...x_{-m} = x_{n-1}.2^{n-1} + x_{n-2}.2^{n-2}...+ x_0.2^0 + x_{-1}.2^{-1} + x_{-2}.2^{-2} + ...+ x_{-m}2^{-m}$$

Fixed point numbers

- Suppose that *n =8-bits*

  Largest integer value can be represented: 255

  Smallest fraction value can be represented: $2^{-8} \sim 10^{-3} = 0.001$

- Problem: limited range of values can be represented, it does not allow enough numbers and accuracy
- Solution: ***Floating point Number***

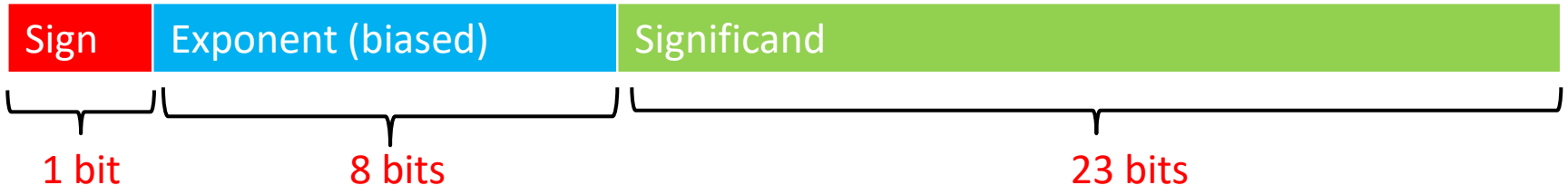
Floating point number

- Express in the follow notation: Fx 2^E (with: F: fraction, E: Exponent, radix of 2)
- IEEE-754 standard: modern computer use to represent floating point number in form: $V = (-1)^S \times F \times 2^E$

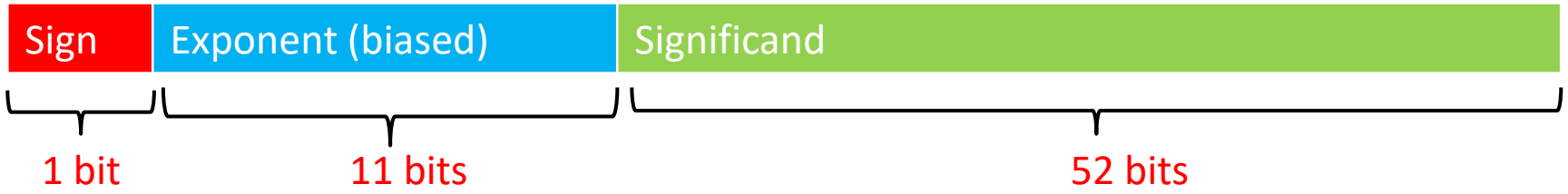
- **Sign:** 1: Negative, 0: Positive
- **Exponent:** saved in n-bits pattern. Represented in K-excess form with
  Single precision: K = 127 ($2^{n-1} - 1 = 2^{8-1} - 1$)
  Double precision: K = 1023 ($2^{n-1} - 1 = 2^{11-1} - 1$)
- **Significand (Fraction):** the remaining bits after dot sign

IEEE-754 standard

Single precision (32-bits)



Double precision (64-bits)



cdio 4.0
fit@hcmus

Ex: Represent X = -5.25 in single precision scheme

Step 1: Convert X to binary system

$X = -5.25_{10} = -101.01_2$

Step 2: Normalize X in this form $\pm 1.F * 2^E$

$X = -5.25 = -101.01 = -1.0101 * 2^2$

Step 3: Represent X in floating point

Signed bit = 1 (Negative number)

Exponent= represent E in K-excess form (with K = 127)

->Exponent = E + 127 = 2 + 127 = 129_{10} = $1000\ 0001_2$

Significant (Fraction)= 0101 0000 0000 0000 0000 000 (plus 19 times of 0's)

-> Result: 1 1000 0001 0101 0000 0000 0000 0000 000

Discussion

1. Why is the exponent stored in K-excess form?
2. Why do we choose K=127(in single precision scheme) instead K=128 (original biased value in 8-bits pattern)
3. How can we represent the zero value in floating point number?

Categories of floating-point values

- Normalized number

Signed bit	1-255 (single precision) 1-2046 (double precision)	Sigfinicant

- Denormalized number

Signed bit	all bits set as 0's	Sigfinicant

- Infinity

Signed bit	all bits set as 1's	all bits set as 0's

- Not a number (NaN)

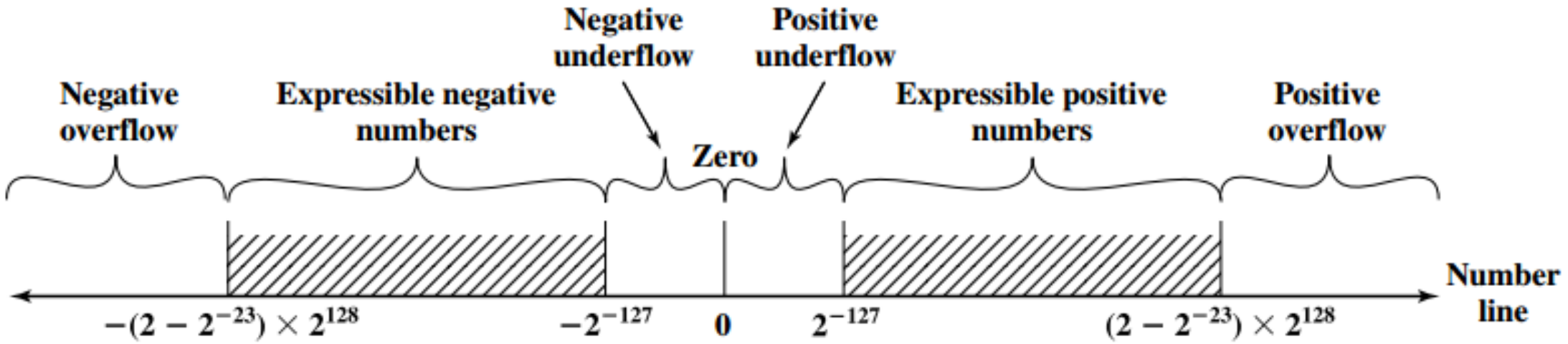
Signed bit	all bits set as 1's	Nonzero

Nonnegative floating-point numbers

			Single precision		Double precision	
Description	exp	frac	Value	Decimal	Value	Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times 2^{-126}$	1.4×10^{-45}	$2^{-52}\times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00\cdots01$	$0\cdots00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01\cdots11$	$0\cdots00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times 2^{127}$	3.4×10^{38}	$(2-\epsilon)\times 2^{1023}$	1.8×10^{308}

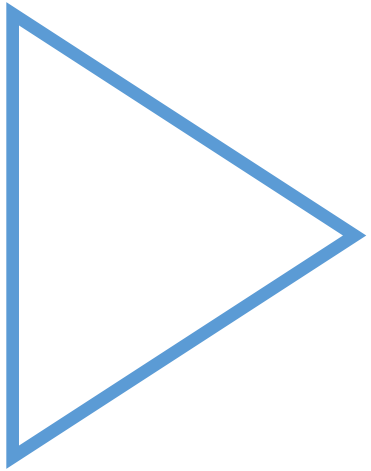
Chap2, Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011, Figure 2.35

Distribution of Single-precision floating-point numbers



Chap9, Computer Organization and Architecture: Design for performance, 8th edition Figure 9.19

Number limits, Overflow and Roundoff



Watch this video:

Number limits, Overflow and roundoff

Take a practice bellow the video

Store text in binary

Read this document and take a practice

cdio 4.0

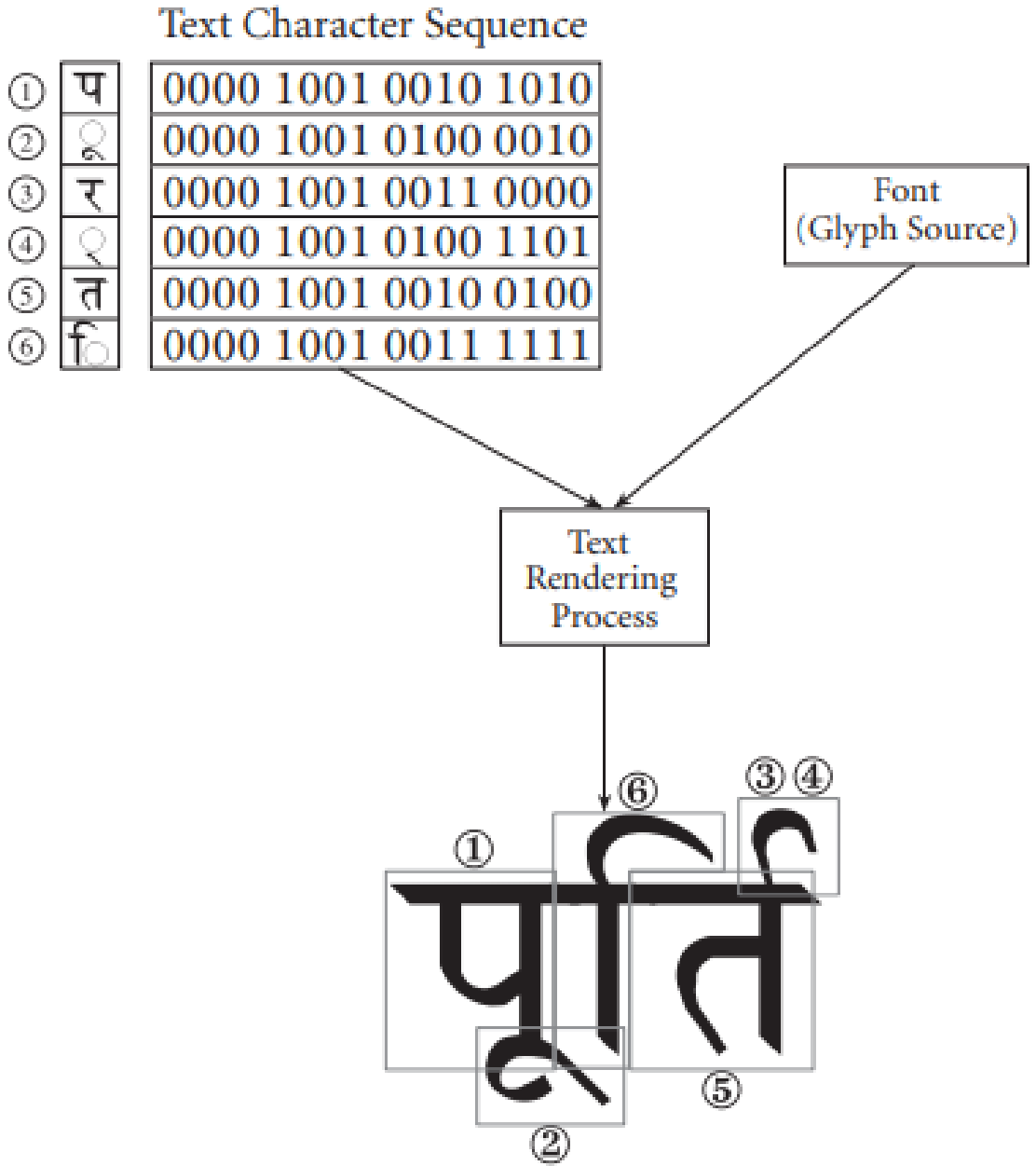
fit@hcmus

ASCII representation of character

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Unicode Standard

- Unicode version 4.0 has more than 160 “blocks,” which is their name for a collectCion of symbols. Each block is a multiple of 16
- A 16-bit encoding, called UTF-16, is the default.
- A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 32 bits for the other characters.
- UTF-32 uses 32 bits per character



Unicode Character Code to Rendered Glyphs

Data Format in C Programs

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

Figure3.1 - Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011

Size of C data type in IA32

Heterogeneous Data

- C provides two mechanisms for creating data types by
  - Combining objects of different types: *structures,* declared using the keyword ***struct***
  - Aggregate multiple objects into a single unit: *unions*, declared using the keyword ***union***
- Read more information in Prentice Hall, 2011, *Computer Systems A Programmers Perspective 2nd,* Section 3.9, page 241

Data Alignment

- Alignment restrictions simplify the design of the hardware forming the interface between the processor and the memory system
- The compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement

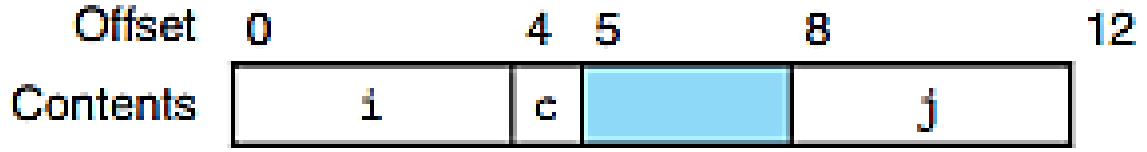
Data Alignment

Ex: consider the following structure declaration

```
struct S1 {
    int  i;
    char c;
    int  j;
};
```

Size of struct S1 without alignment

Size of struct S1 with 4-bytes alignment



- 04_Floating-point.pdf
- Willian Stalling, **Computer Organization and Architecture: Design for performance**, 8th edition, *Chapter 9*
- Patterson and Hennessy, **Computer Organization and Design: The Hardware / Software Interface**, 5th edition, *Chapter 3*
- Prentice Hall, **Computer Systems A Programmers Perspective** 2nd, 2011, *Chapter 2*