CHAPTER

1

DATA REPRESENTATION

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

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

fit@hcmus

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

$$X_{n-1} \times 2^{n-1} + \ldots + X_1 \times 2^1 + X_0 \times 2^0$$

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

Absolute value is 10000001-> 129 (decimal)

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

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

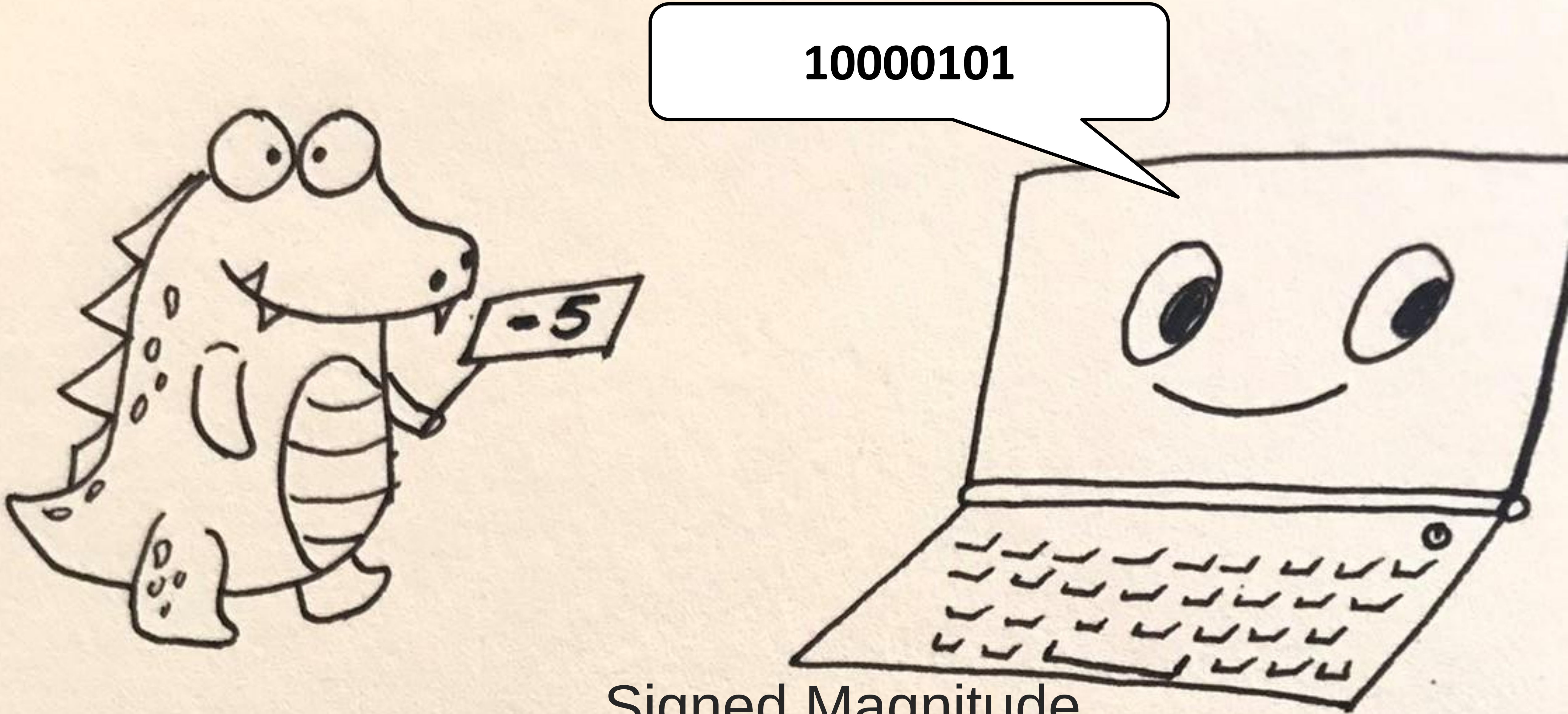
Signed Integers representation

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

fit@hcmus

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)

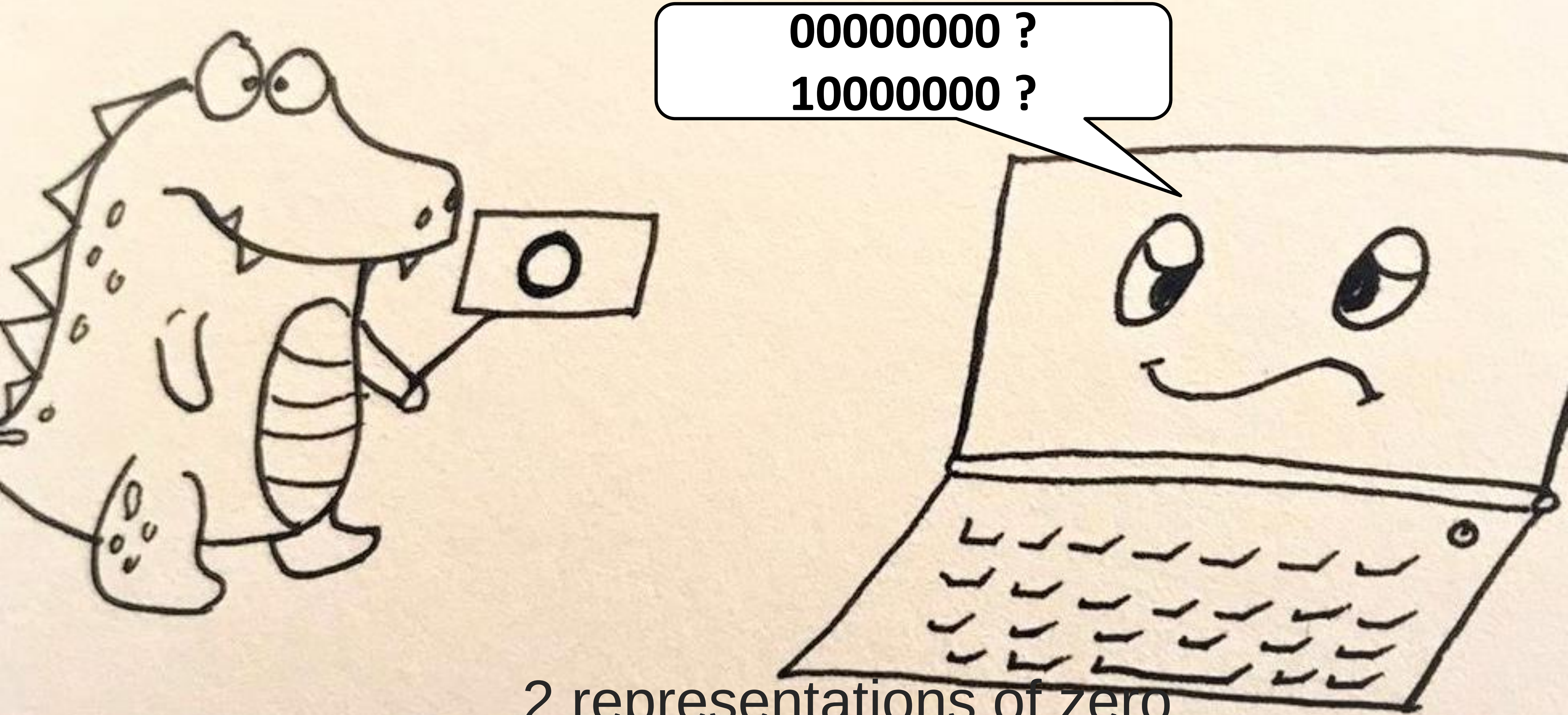
Sign-Magnitude representation

- 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)



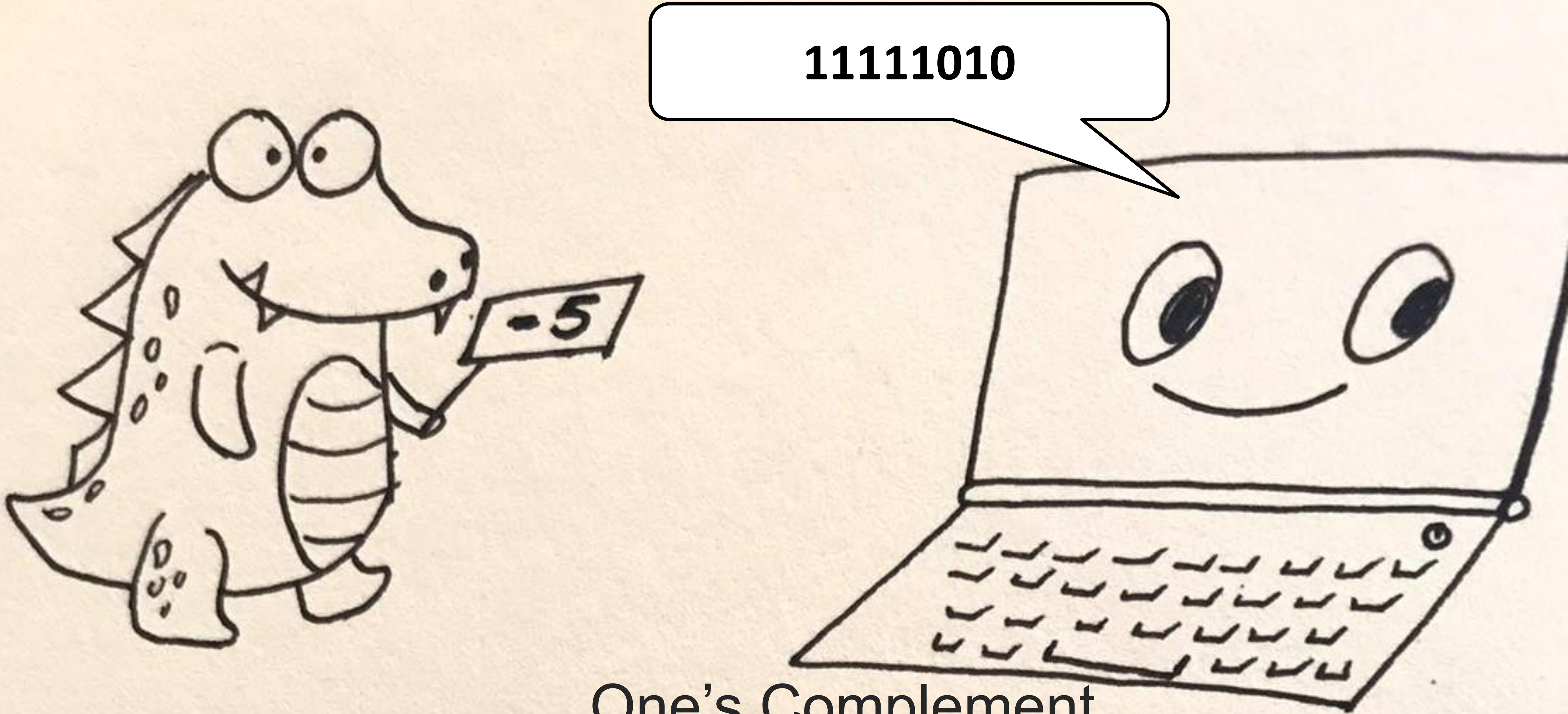
2 representations of zero

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

fit@hcmus

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

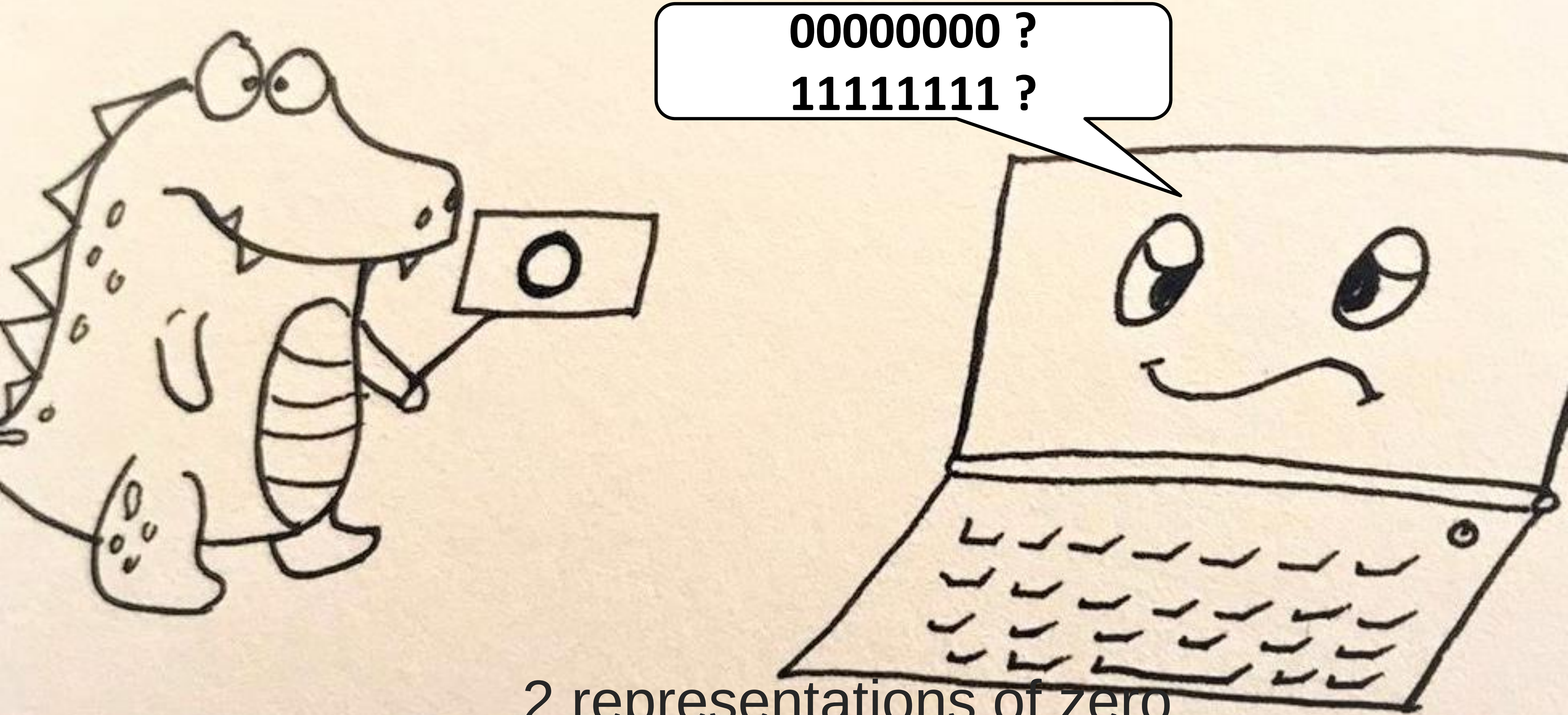
One’s Complement representation

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

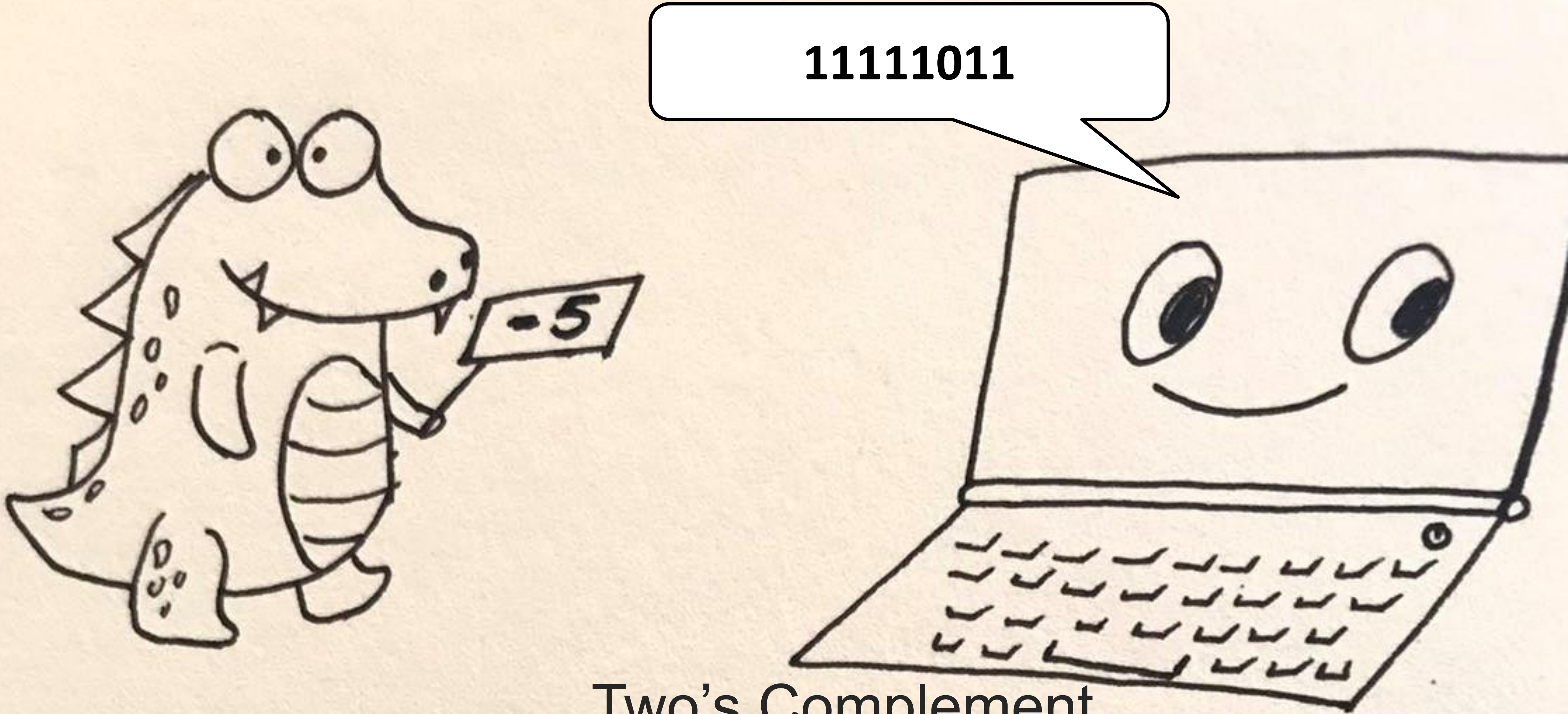
- Values:

$$X_{n-1} \times (-2^{n-1}+1) + X_{n-2} \times (2^{n-2}) + \ldots + X_1 \times 2^1 + X_0 \times 2^0$$

- 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

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

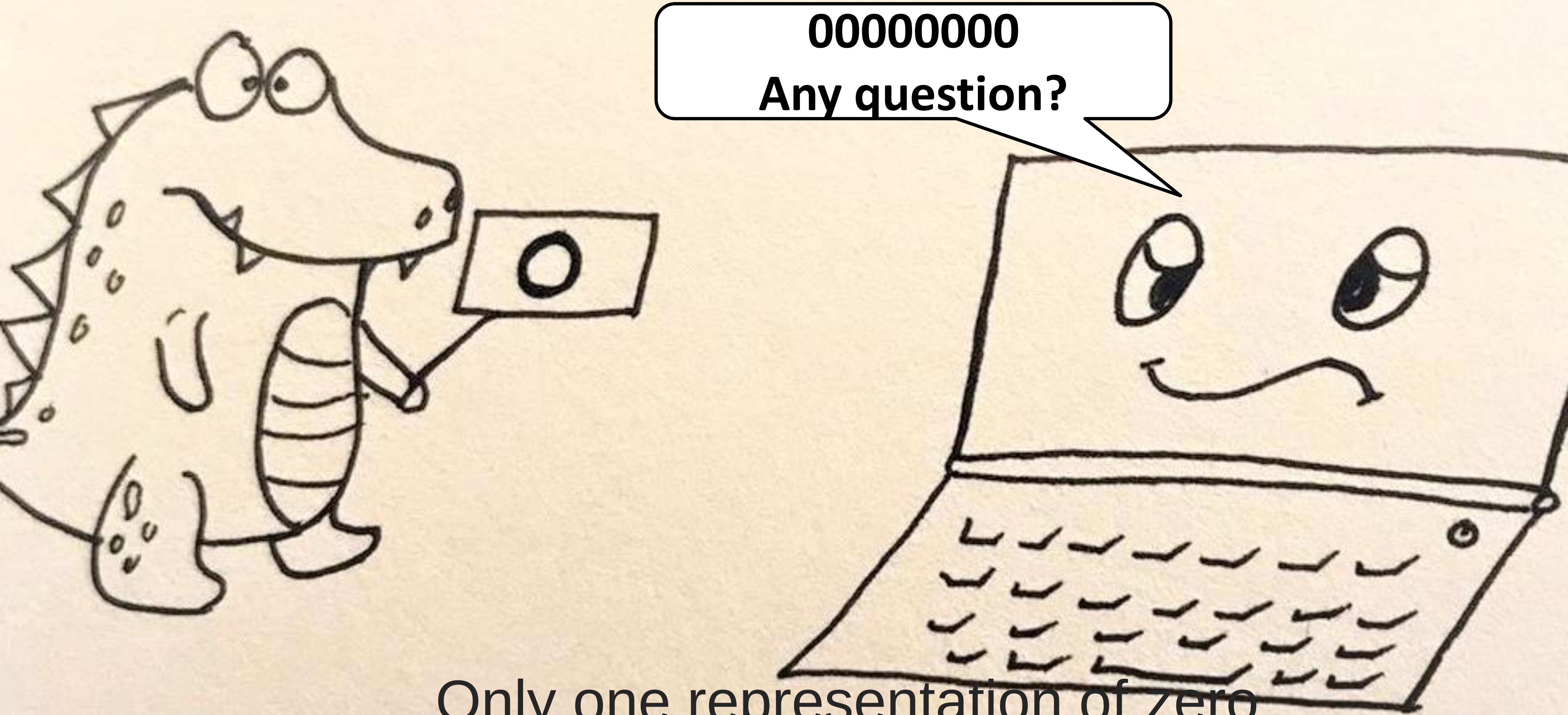
Sign bit is 1 -> negative number

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

- Values:

$$X_{n-1} \times (-2^{n-1}) + X_{n-2} \times (2^{n-2}) + \ldots + X_1 \times 2^1 + X_0 \times 2^0$$

- 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

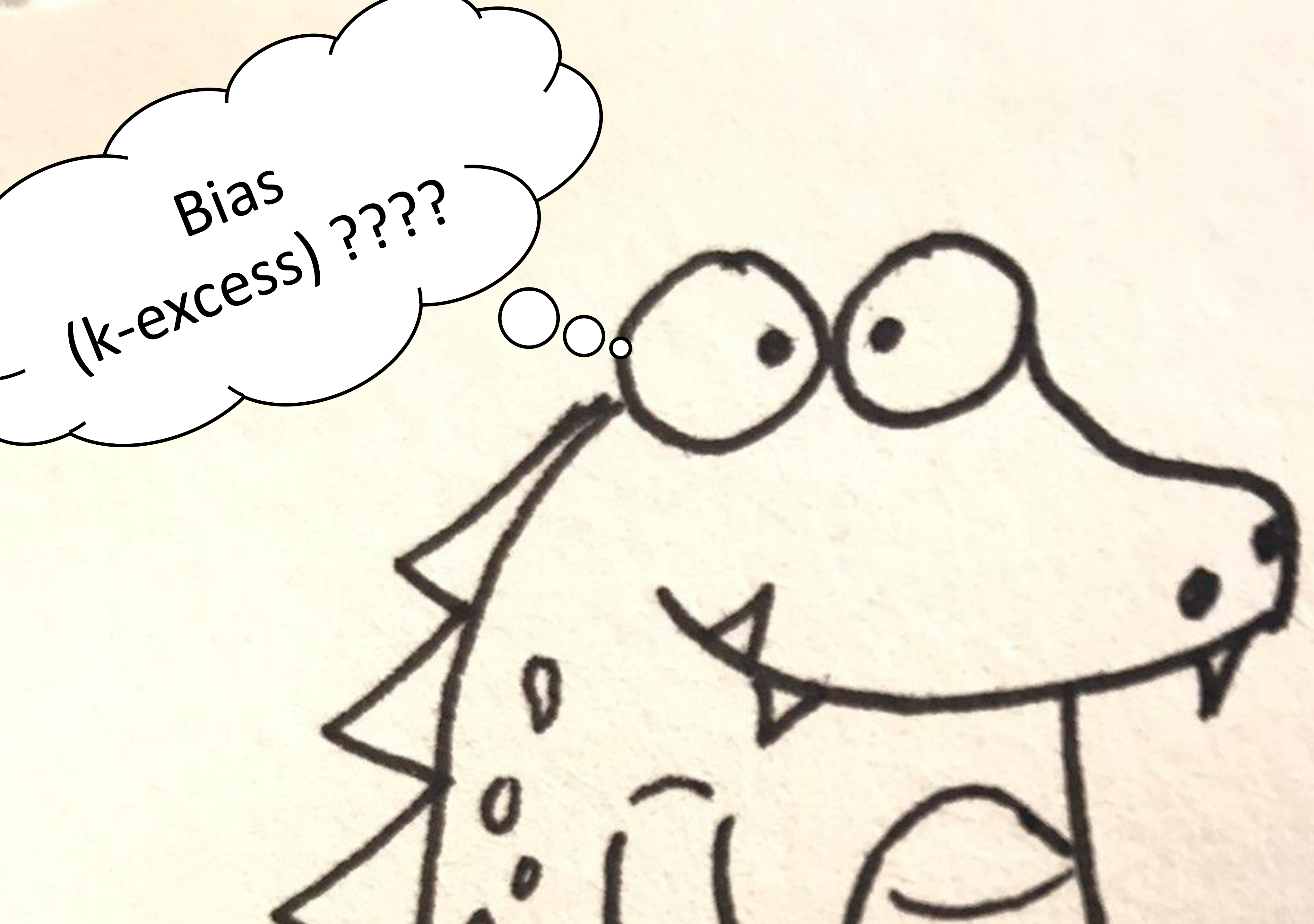
Absolute value is ***the complement*** of 0000001 + 1

-> 1111110 + 1

-> <u>1111111</u>

-> 127 (decimal)

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


Bias
(k-excess) ????

fit@hcmus

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. K is usually $2^{n-1}-1$ or 2^{n-1}
- The most-significant bit (MSB) is recognized as the sign bit:

  1 -> positive integer

  0 -> negative integer
- All n bits represents the magnitude of the integer (n is the length of bit pattern) by subtraction between the bias value K and unsigned value N of that n-bits binary pattern

  positive integers: N > K, the absolute value of the integer is equal to N - K

  negative integers: N < K, the absolute value of the integer is equal to K - N

Biased (K-excess)

Ex: Suppose that n =8, K =127, the binary representation of N is 10000001 with unsigned value is 129 decimal (greater than K)

-> N is positive integer

-> Absolute value is N – K = 129 – 127 = 2 (decimal)

Hence, the integer is **2** (decimal)

Biased (K-excess)

- The n-bits binary pattern can represent the values

  $-K \rightarrow 2^{n-1} - K$

  $-(2^{n-1} - 1) \rightarrow 2^{n-1}$, with $K = 2^{n-1} - 1$

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

- There is one representations for the number zero: 01111111
- Biased representations are now primarily used for the exponent of *floating-point numbers*

Biased (K-excess)

Ex: Suppose that n =8, K=127, how to represent a number in binary

Positive number: 25 (decimal)

N = 25 + K	=	25 + 127 = 152
The bit binary pattern is	->	10011000

Negative number: -25 (decimal)

N = -25 + K	=	-25 + 127 = 102
The bit binary pattern is	->	01100110

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.

1	1	0	0	1	0	1	0

0	1	1	0	0	1	0	1

Shift Right Logical (SHR)

1	1	0	0	1	0	1	0

1	1	1	0	0	1	0	**0**

Shift Left Logical (SHL)

Shift Operations

An arithmetic shift right preserves the sign bit

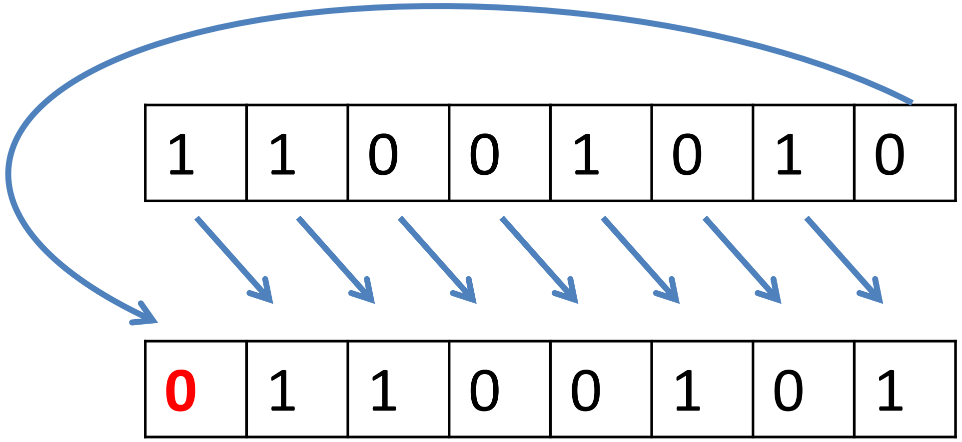
1	1	0	0	1	0	1	0

1	1	1	0	0	1	0	1

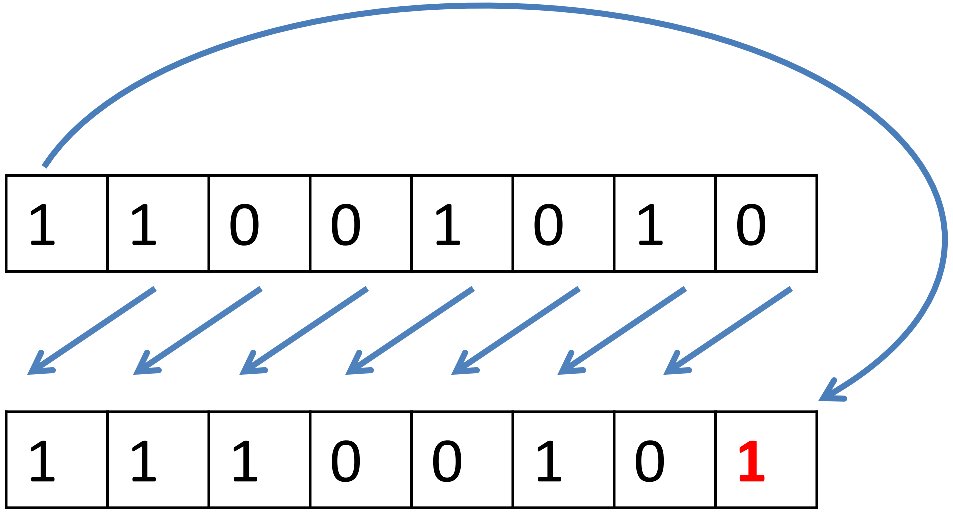
Shift Right Arithmetic
(SAR)

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)

  Convert lower case to upper case

'a' (61h)	0110 0001
Mask (DFh)	1101 1111
'A' (41h)	0100 0001

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

  Convert upper case to lower case

'B' (42h)	0100 0010
Mask (20h)	0010 0000
'b' (62h)	0011 0010

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

					1	
		1	1	1	0	1
+		1	0	0	0	1
	1	0	1	1	1	0

$$\begin{array}{rcr} 1001 & = & -7 \\ +0101 & = & 5 \\ \hline 1110 & = & -2 \end{array}$$

(a) (−7) + (+5)

$$\begin{array}{rcr} 1100 & = & -4 \\ +0100 & = & 4 \\ \hline 10000 & = & 0 \end{array}$$

(b) (−4) + (+4)

$$\begin{array}{rcr} 0011 & = & 3 \\ +0100 & = & 4 \\ \hline 0111 & = & 7 \end{array}$$

(c) (+3) + (+4)

$$\begin{array}{rcr} 1100 & = & -4 \\ +1111 & = & -1 \\ \hline 11011 & = & -5 \end{array}$$

(d) (−4) + (−1)

$$\begin{array}{rcl} 0101 & = & 5 \\ +0100 & = & 4 \\ \hline 1001 & = & \text{Overflow} \end{array}$$

(c) (+5) + (+4)

$$\begin{array}{rcl} 1001 & = & -7 \\ +1010 & = & -6 \\ \hline 10011 & = & \text{Overflow} \end{array}$$

(f) (−7) + (−6)

Subtract Operation

Rule:

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

Ex: 11101 – 10011 = 11101 + 01101

$$
\begin{array}{r}
1 \\
11101 \\
+\ 01101 \\
\hline
101010
\end{array}
$$

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 $\boxed{1}$0011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 $\boxed{1}$1001 = −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 $\boxed{1}$0110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

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

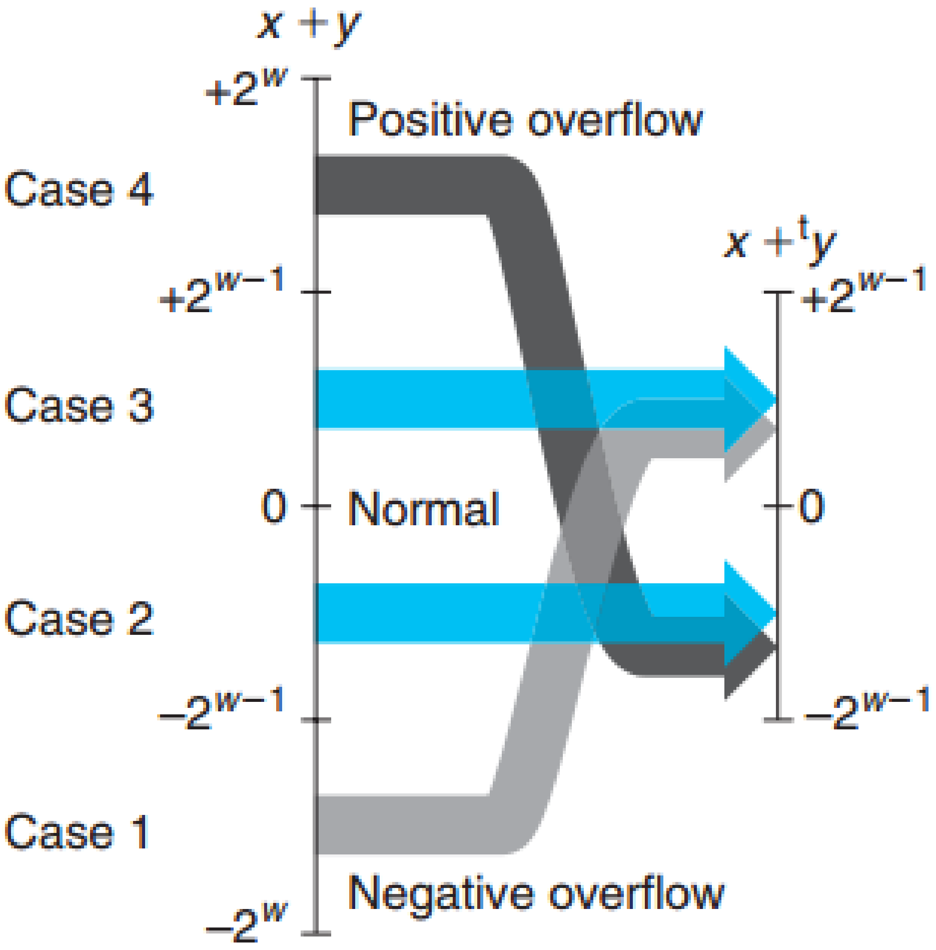
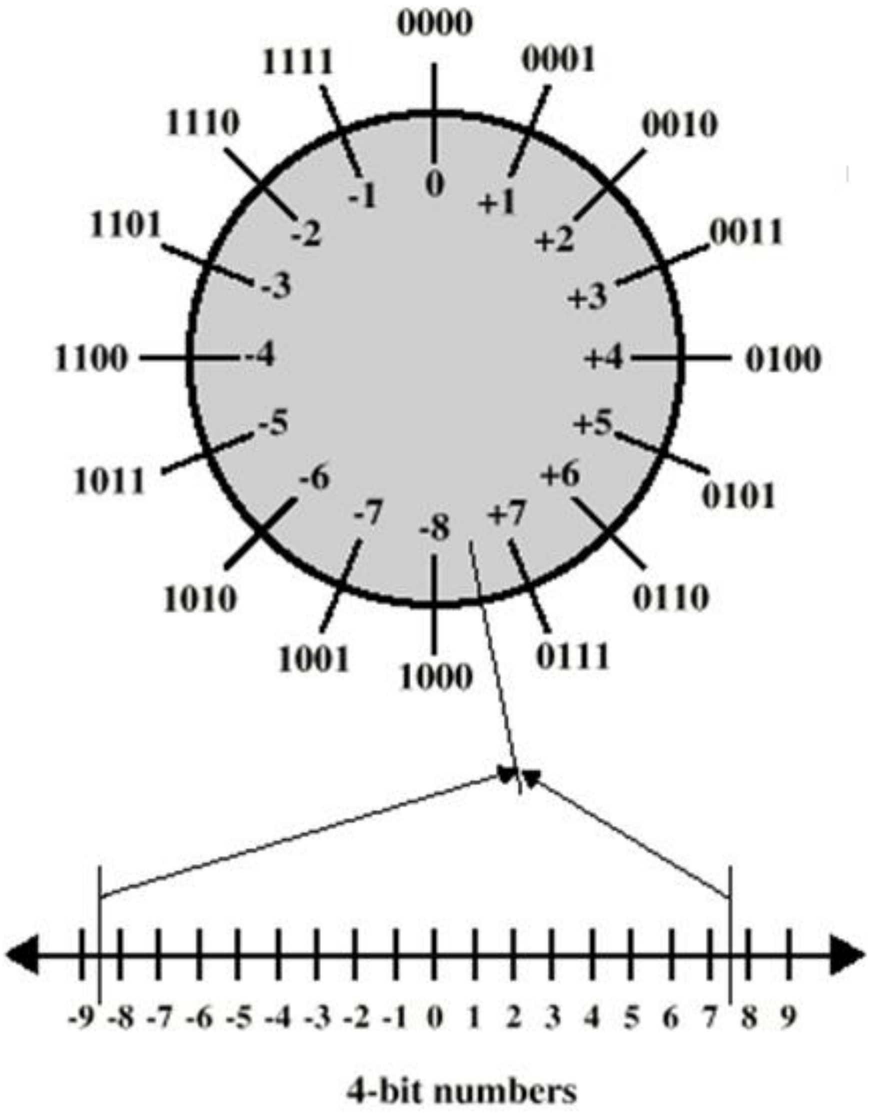


Photo by Chap2, Prentice.Hall.Computer.Systems.A.Programmers.Perspective.2nd.2011



Multiply Operation

Rule:

×	0	1
0	0	0
1	0	1

Ex:

			1	0	0	0	1
×					1	1	0
			0	0	0	0	0
+		1	0	0	0	1	
	1	0	0	0	1		
	1	1	0	0	1	1	0

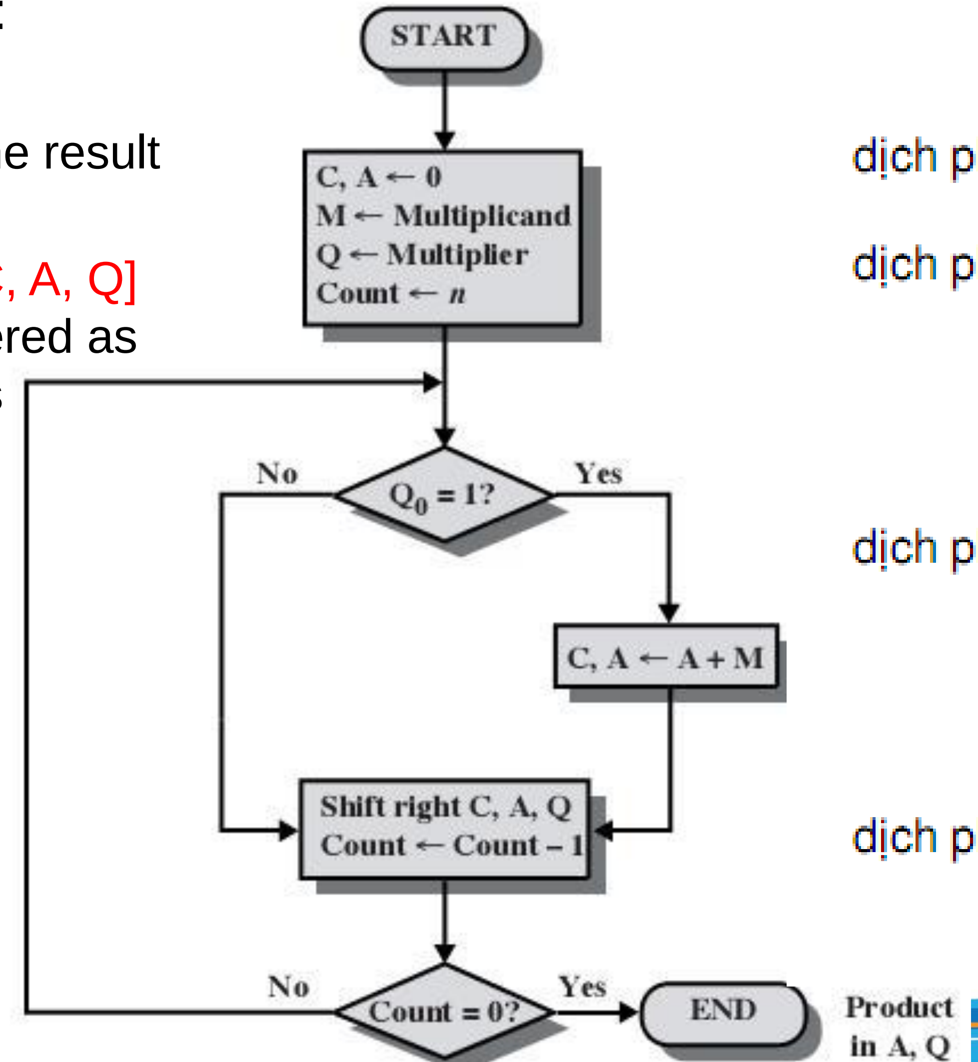
Multiply Operation

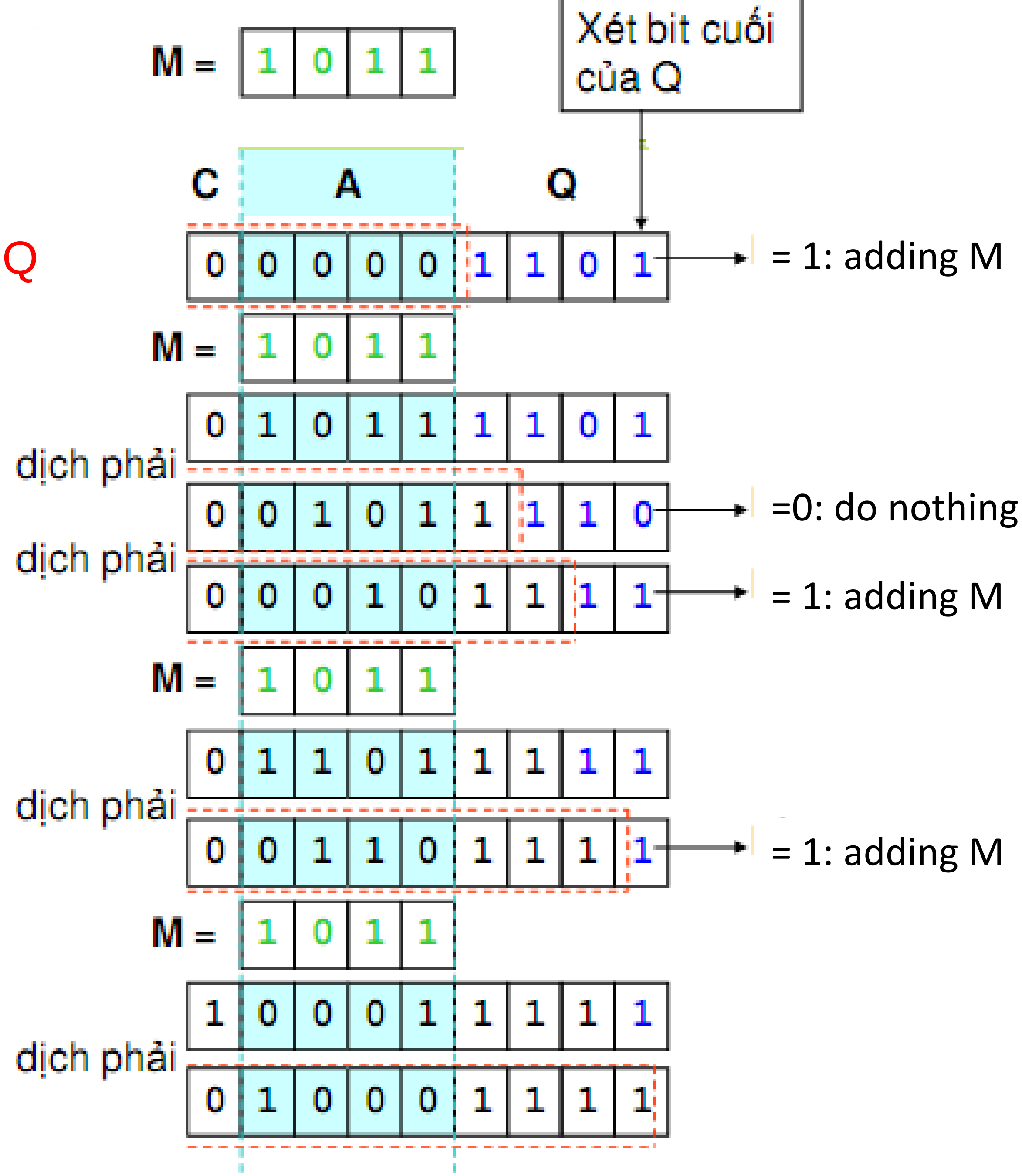
```
    M
       1011     = 11
 x  Q  1101     = 13
 ---------------
       1011
      0000
     1011
    1011
 ---------------
    10001111    = 143
```

```
      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
- 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 Operation – Two's Complement Numbers

1001	(-7)	1001	(-7)
× 0011	(3)	× 1100	(-4)
11111001	$(-7) \times 2^0 = (-7)$	11100100	(-28)
11110010	$(-7) \times 2^1 = (-14)$	11001000	(-56)
11101011	(-21)	10101100	(-84) ???

- Why's wrong ?
  - Second factor: 1100 $\neq -(2^3 + 2^2)$ $(1100 = -2^2)$
- Solution 1
  - Convert 2 factors to positive
  - Multiply as unsigned numbers previously
  - Adjust sign of result
- Solution 2
  - Booth Algorithm

Booth Algorithm – Ideas

Positive

$2^n + 2^{n-1} + \ldots + 2^{n-K} = 2^{n+1} – 2^{n-K}$

$M \times (01111010) = M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1)$

$= M \times (2^7 - 2^3 + 2^2 - 2^1)$

Negative

$X = \{111..10x_{k-1}x_{k-2}\ldots x_1x_0\}$

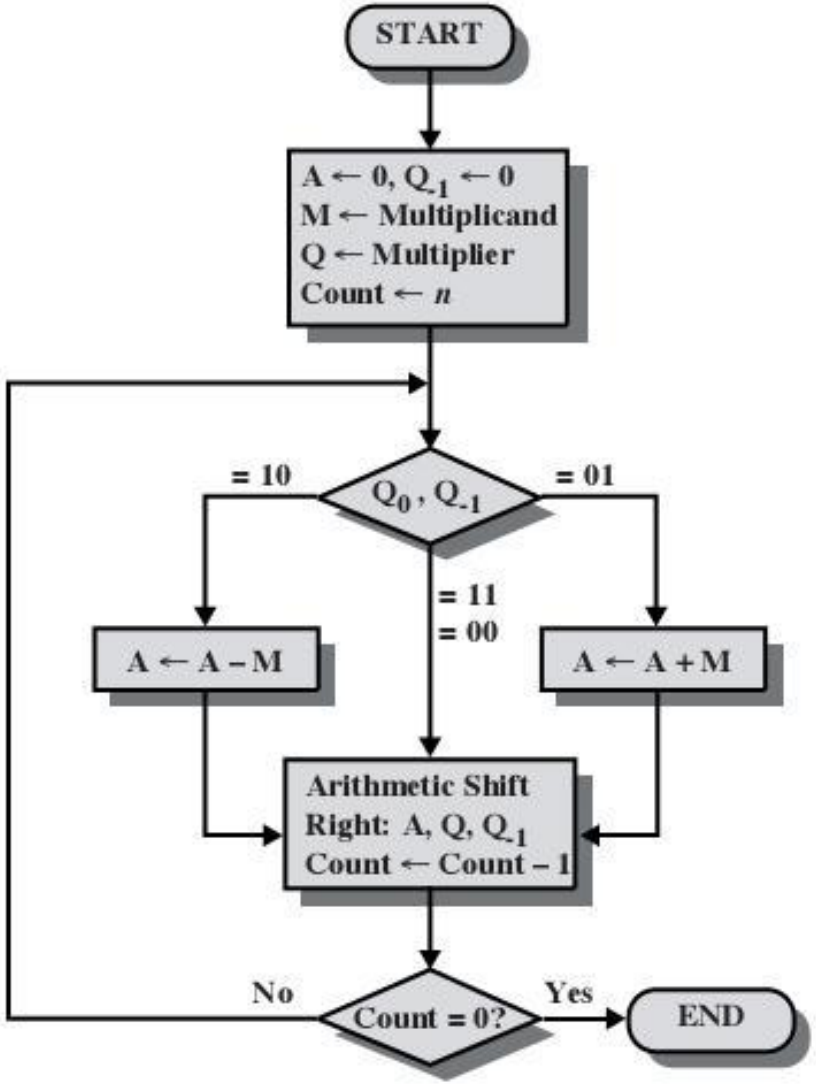
$-2^{n-1} + 2^{n-2} + \ldots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \ldots + (x_0 \times 2^0)=$

$-2^{k+1} + (x_{k-1} \times 2^{k-1}) + \ldots + (x_0 \times 2^0)$

$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$

$= M \times (-2^3 + 2^1)$

$= M \times (-2^3 + 2^2 -2^1)$



fit@hcmus

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)

  Q0 (1 bit): the LSB bit of Q

  [Q,Q-1] (n + 1 bit)

Booth’s Multiplication Algorithm

```
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 then 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

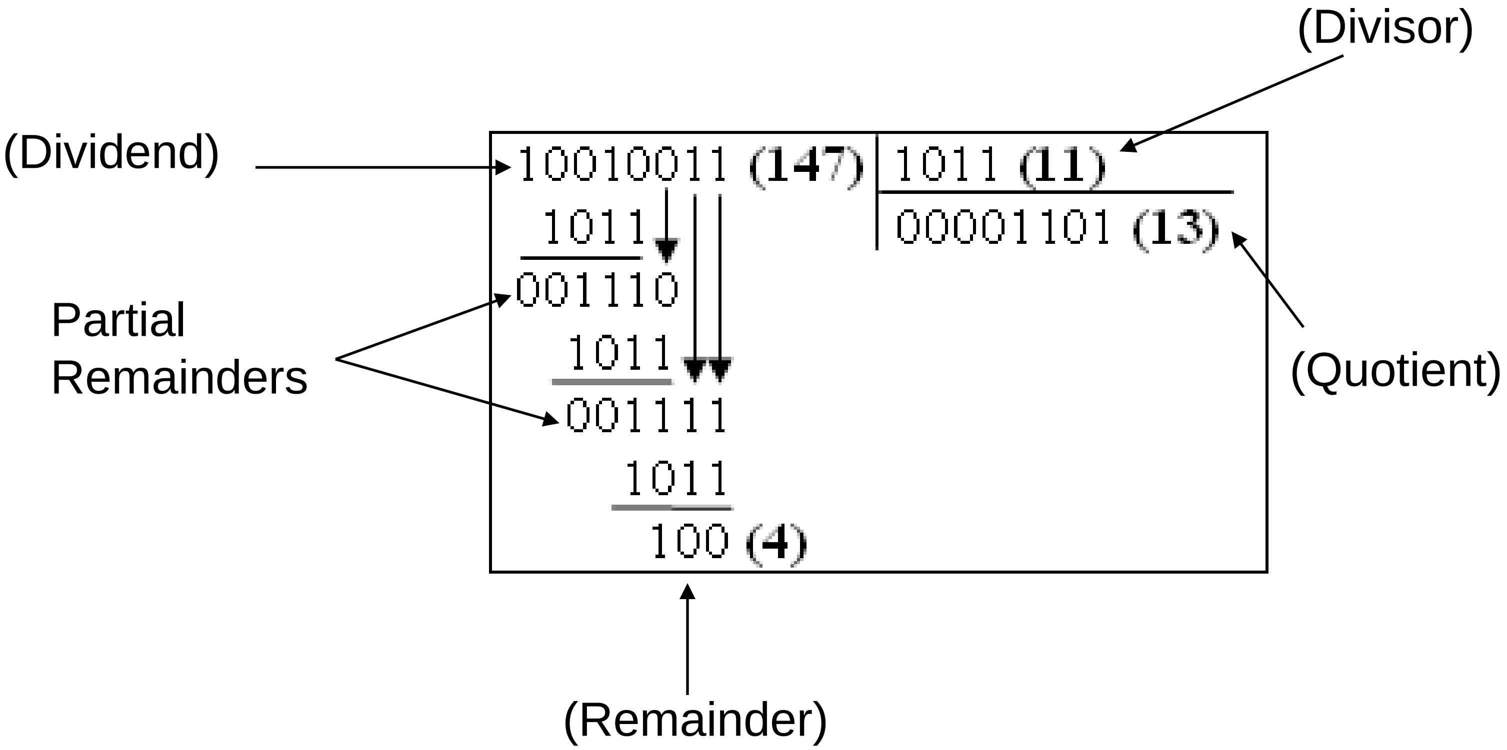
cdio 4.0
fit@hcmus

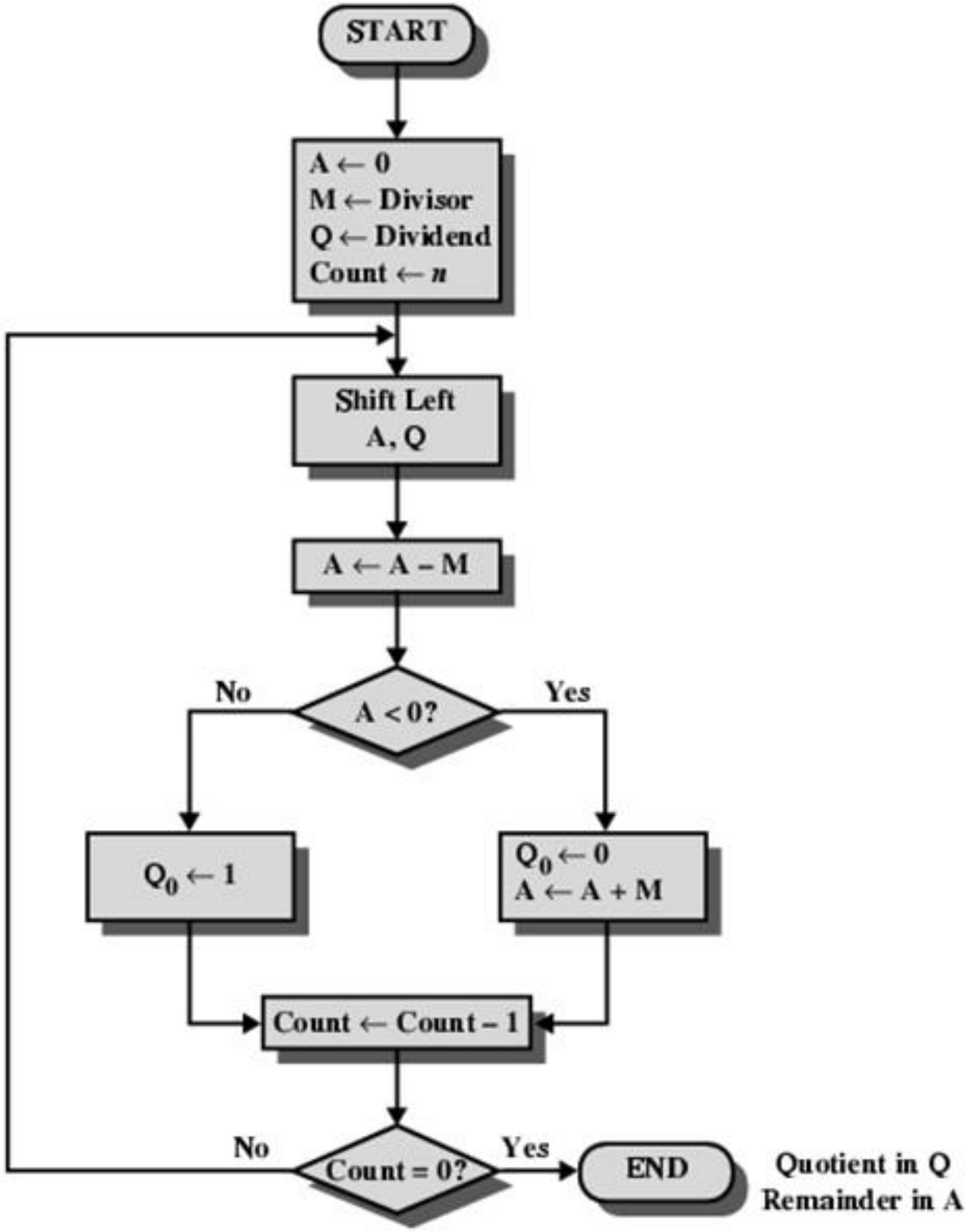
Booth – Algorithmic Basis

- Bước 0: $A = (0 + (Q_{-1}-Q_0).M)$
- Bước 1: $A = (0 + (Q_{-1}-Q_0).M + (Q_0-Q_1).M.2^1)$
  $= M.(Q_{-1}-Q_0 + Q_0.2-Q_1.2)$
- Bước 2: $A = (M.(Q_{-1}-Q_0 + Q_0.2-Q_1.2) + (Q_1-Q_2).M.2^2)$
  $= M.(Q_{-1}-Q_0 + Q_0.2-Q_1.2 + Q_1.2^2-Q_2.2^2)$
- Bước 3:
  $A = M.(Q_{-1}-Q_0 + Q_0.2-Q_1.2 + Q_1.2^2-Q_2.2^2 + Q_2.2^3-Q_3.2^3)$
  $= M.(Q_{-1}+Q_0+Q_1.2 + Q_2.2^2-Q_3.2^3)$
- Bước n-1:
  $A = M.(Q_{-1}+Q_0+Q_1.2 + Q_2.2^2+Q_3.2^3+\ldots+Q_{n-2}.2^{n-2}-Q_{n-1}.2^{n-1})$

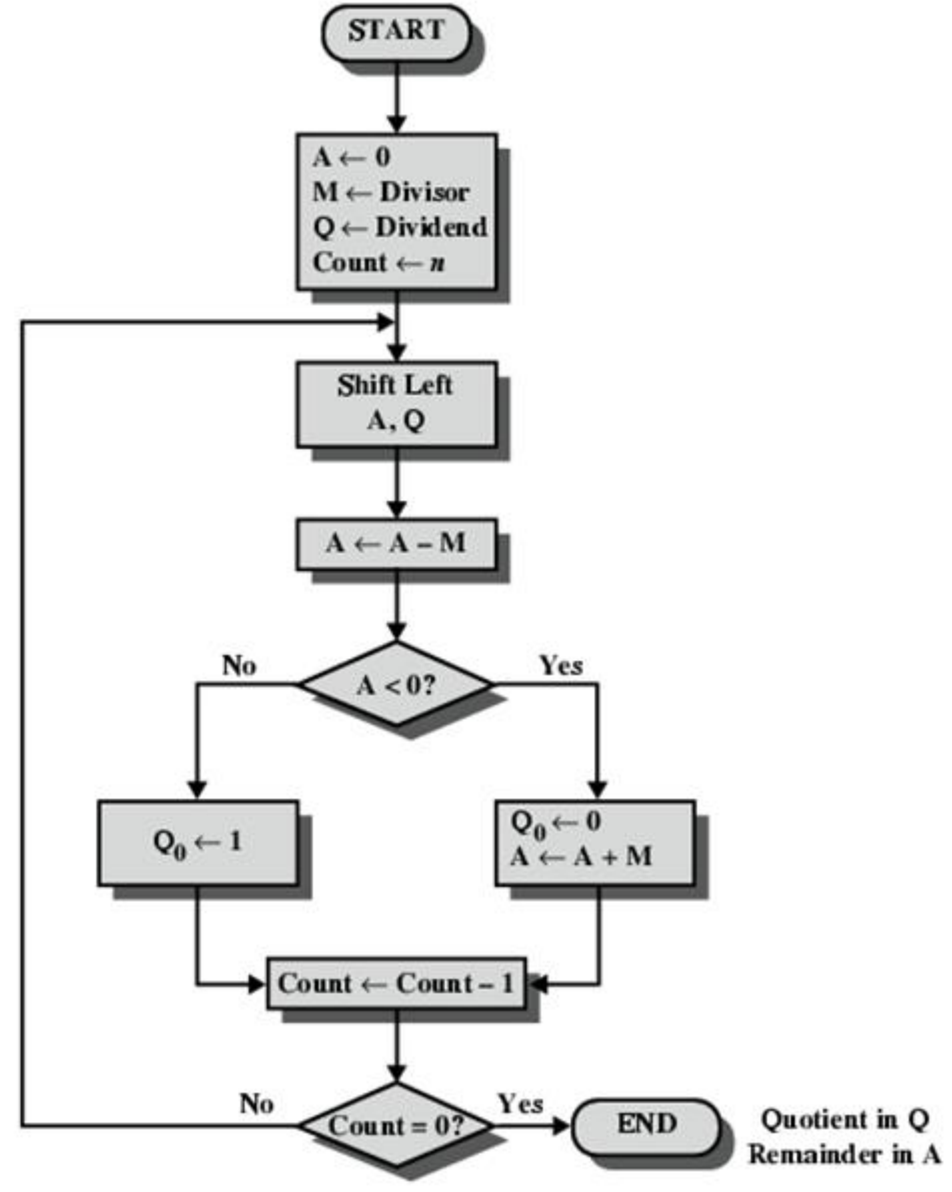
$\rightarrow A = M.Q$

Divide Operation





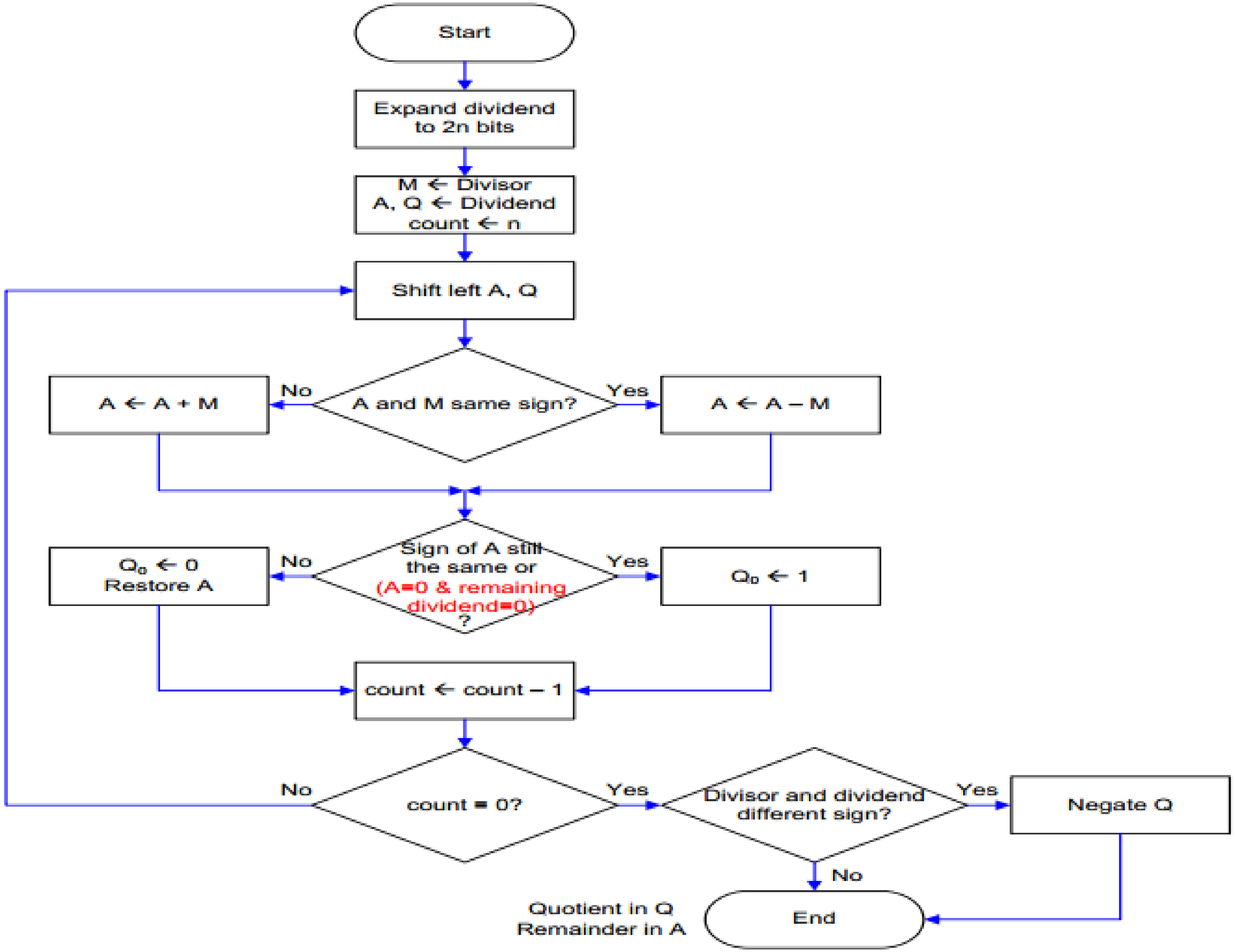
Divide Algorithm



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

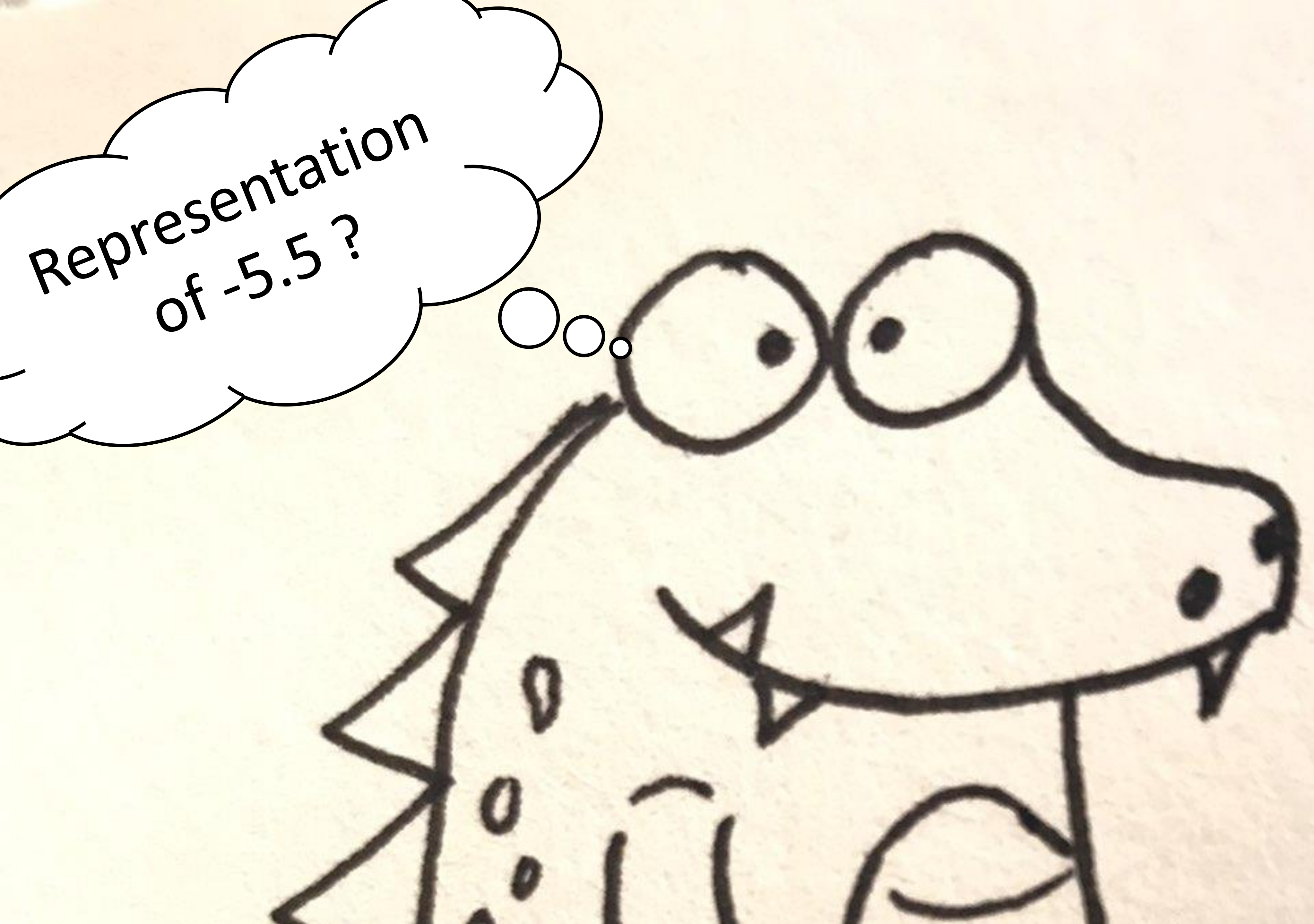
(7)/(3)

Divide Operation – Two's Complement Numbers



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

$(7)/(-3)$


Representation
of -5.5 ?

fit@hcmus

Fixed point numbers

i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.25	1/4
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	...
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

Represent 5.375_{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)
- $5_{10} = 4 + 1 = 0000\ 0101_2$
- 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	0000 0101	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***

fit@hcmus

Floating point number - Ideas

- Normalized form:

  $123000000000 \sim 1.23\times10^{11}$ and $0.0000000000123 \sim \mathbf{1.23\times10^{-11}}$

- So apply to the fixed point number:

  $x = 00000101.01100000 = 2^2 + 2^0 + 2^{-2} + 2^{-3}$

  $x = 1.01011 \times 2^2$

- Then instead of storing 16 bit, it can only be stored by 7 bit (5 bit fraction + 2 bit exponent)

  fraction: 01011

  exponent: 10

- It's the idea basis of floating point number

  Need to store: the fraction, the exponent and … the sign

Floating point number

- Express in the follow notation: $\pm F \times 2^E$ (with: F: fraction, E: Exponent, radix of 2)

Sign	Exponent (biased form)	Significand

- IEEE-754 standard: represent floating point number in form: $V = (-1)^S \times 1.F \times 2^E$

Single precision (32-bits)

Sign	Exponent (biased)	Significand
1 bit	8 bits	23 bits

Double precision (64-bits)

Sign	Exponent (biased)	Significand
1 bit	11 bits	52 bits

- **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

Single-precision Floating Point

Sign	Exponent (biased)	Significand
1 bit	8 bits	23 bits

- Value: 5.375

  Normalized form: 1.01011×2^{2}

0	1000 0001	0101 1000 0000 0000 0000 000

- Vice versa, values:

$$V = (-1)^{S} \times 1.F \times 2^{E}$$

$$+1.0101100\ldots00\times 2^{10000001} \sim +(1+2^{-2} + 2^{-4} + 2^{-5}) \times 2^{2} = 5.375$$

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

fit@hcmus

Ex: Represent -23.40625 in single precision scheme

1. Integer part:

   23 = 16 + 4 + 2 + 1 = 10111

2. Fraction part:

   .40625 = .25 + .125 + .03125 = .01101

3. Combine and normalize:

   $10111.01101 = 1.011101101 \times 2^4$

4. Exponent: 127 + 4 = 10000011

1	**1000 0011**	**011 1011 0100 0000 0000 0000**

Ex: Represent X = 0.1 in single precision scheme

- 0.1

  = 0.0625 + 0.03125 + 0.00390625 + 0.001953125 + …

  = 1/16 + 1/32 + 1/256 + 1/512 + …

  = $2^{-4} + 2^{-5} + 2^{-8} + 2^{-9}$ + …

  = 0.0001100110… * 2^0

  = 1.1001100110… * 2^{-4}

  - Sign: 0
  - Exponent = -4 + 127 = 123 = 01111011
  - Significand = 100110011001…

0	0111 1011	1001 1001 1001 1001 1001 100

Ex: Single Precision Floating Point to Decimal Value

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 → Positive
- Exponent:
  - 0110 1000 có giá trị (dạng biased) là 104 - 127 = -23
- Significand:

  $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$

  $= 1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

  $= 1.0 + 0.666115$

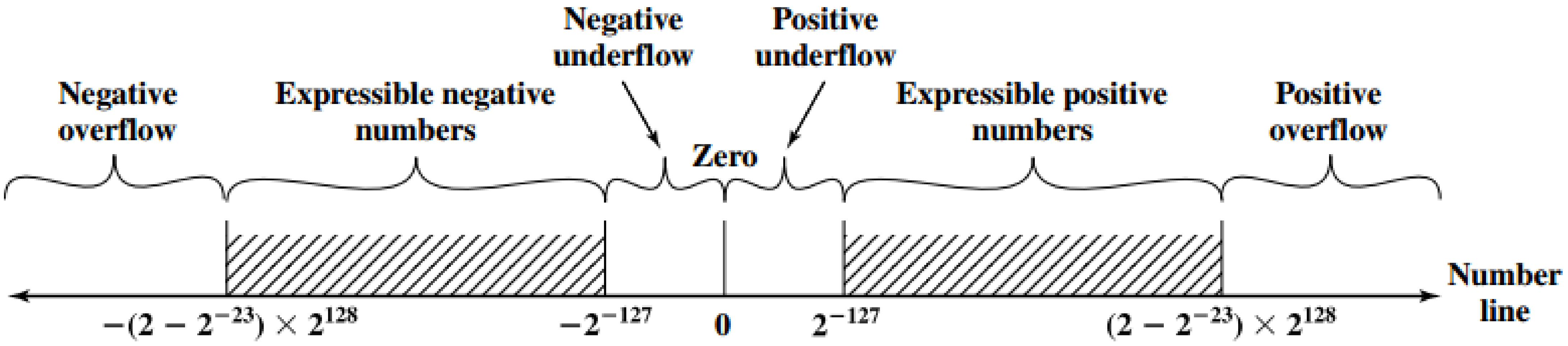
- Result: $1.666115 \times 2^{-23} \sim 1.986 \times 10^{-7}$

  (~ 2/10,000,000)

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?

Distribution of Single-precision floating-point numbers



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

fit@hcmus

Categories of floating-point values

- Normalized number

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

- Denormalized number

Signed bit	all bits set as 0's	Significand

- 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

cdio 4.0

fit@hcmus

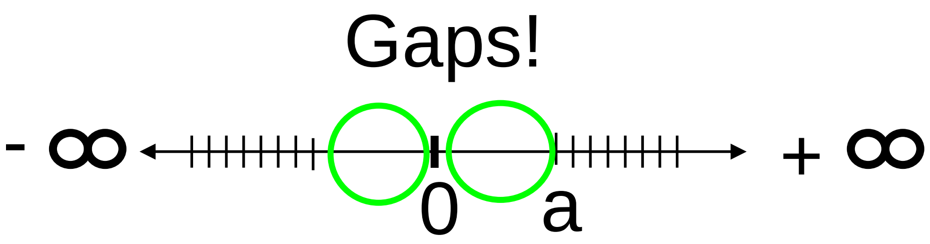
Not a number (NaN)

1. $XX - (+\infty)$
2. $+ (+\infty)$
3. $X + (-\infty)$
4. $X - (-\infty)$
5. $X \times (+\infty)$
6. $X / (-\infty)$
7. $(+\infty) + (+\infty)$
8. $(-\infty) + (-\infty)$
9. $(-\infty) - (+\infty)$
10. $(+\infty) - (-\infty)$
11. $(+\infty) + (-\infty)$
12. $(-\infty) + (+\infty)$
13. $(+\infty) - (+\infty)$
14. $(-\infty) - (-\infty)$
15. $\infty \times 0$
16. $\infty / 0$
17. $X / 0$
18. $0 / 0$
19. ∞ / ∞
20. $sqrt(X), X<0$
21. ……………

Denormalized number

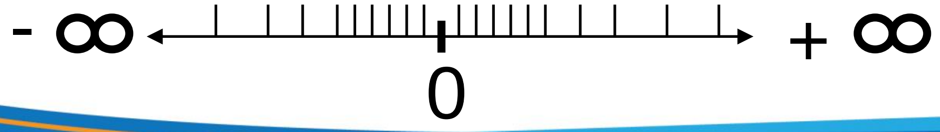
- Positive Min of normalized number:

  $a = 1.0\ldots_2 \times 2^{-126} = 2^{-126}$



  Reason: implicitly **1** + fraction part

- Solution:
  - Consider all bits of exponent part are 0 (Significand ≠ 0) as denormalized form. So the significand will not implicitly plus 1 anymore
  - Positive Min of denormalized number:
    - $a = 0.00\ldots1_2 \times 2^{-126} = 2^{-23} \times 2^{-126} = 2^{-149}$



fit@hcmus

Nonnegative floating-point numbers

Description	exp	frac	Single precision Value	Single precision Decimal	Double precision Value	Double precision Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times2^{-126}$	1.4×10^{-45}	$2^{-52}\times2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times2^{-1022}$	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.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times2^{127}$	3.4×10^{38}	$(2-\epsilon)\times2^{1023}$	1.8×10^{308}

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

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

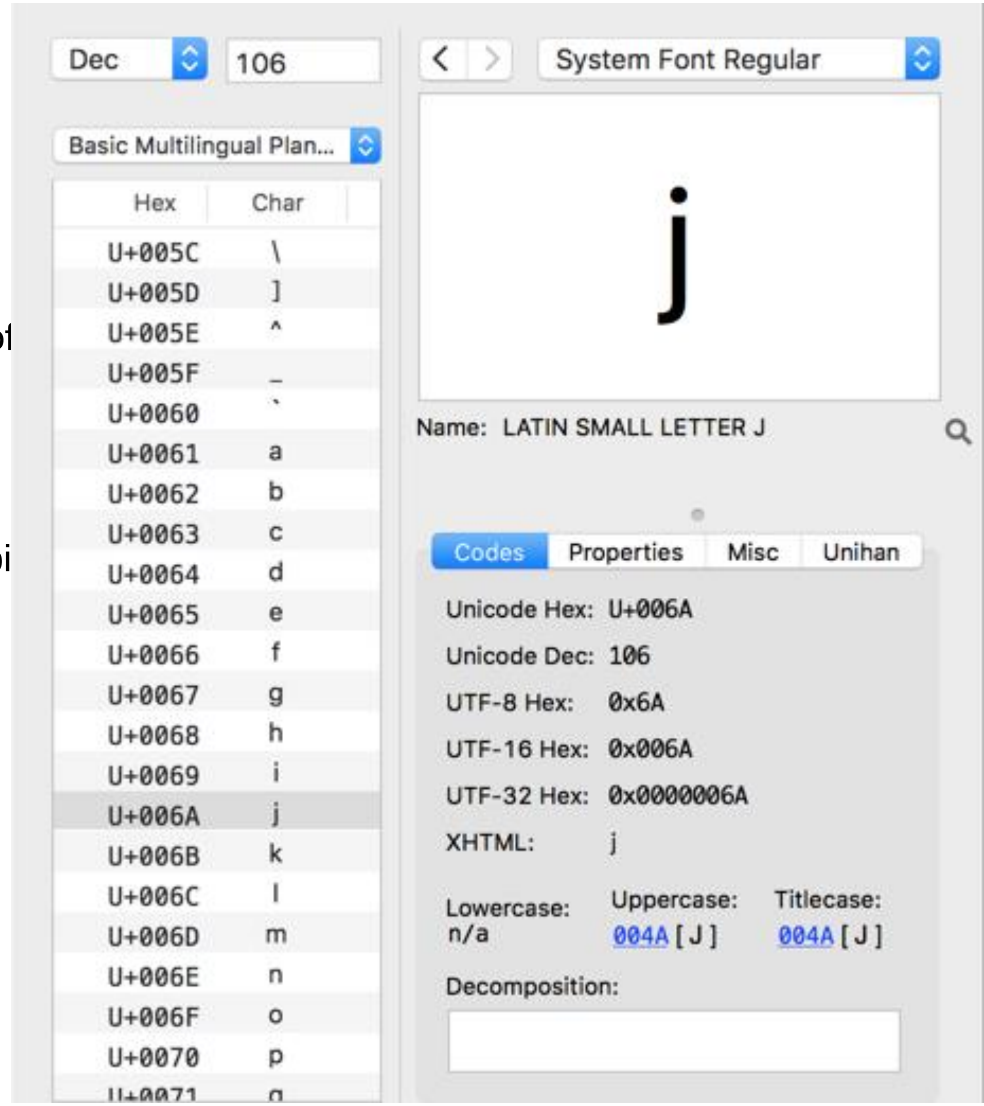
ASCII representation of character

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

fit@hcmus

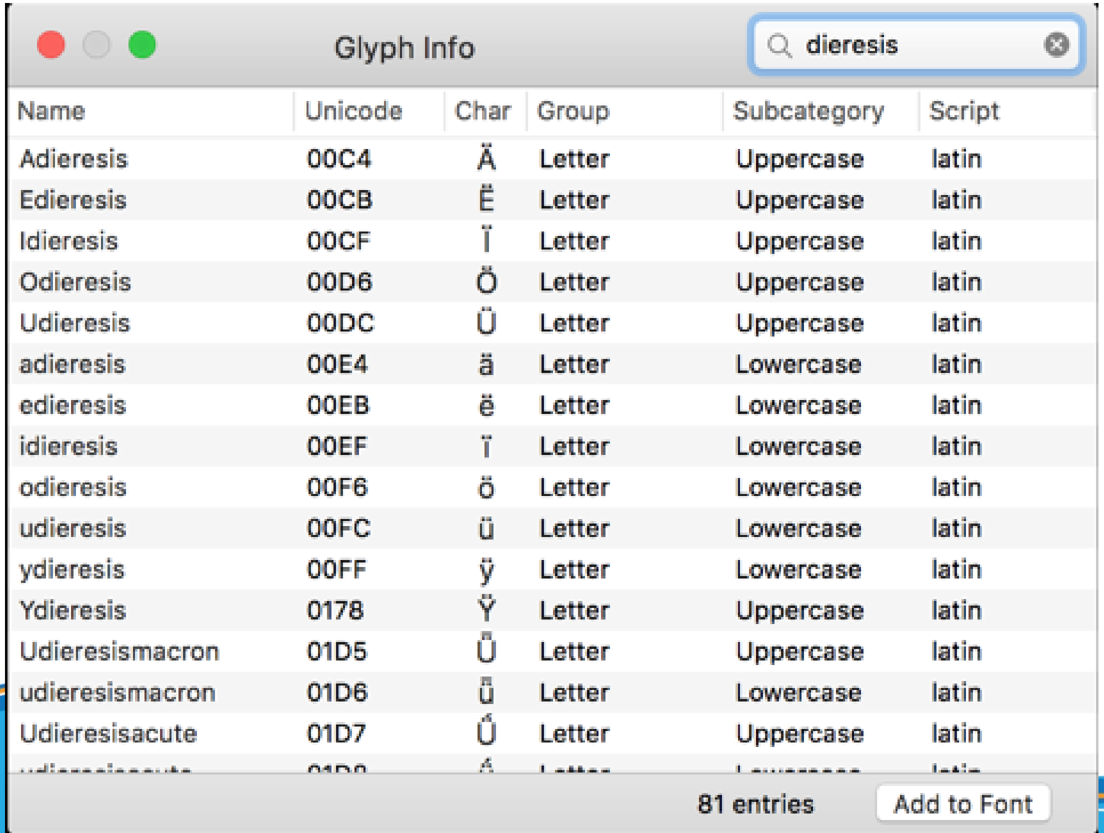
- Unicode has 17 planes of 65,536 characters each
  - BMP or Plane 0: ranging from U+0000 to U+FFFF
  - SMP or Plane 1: from U+10000 to U+1FFFF
- Planes are subdivided in many character blocks, usually comprising a script
  - Unicode version 14.0 has 320 “blocks”, which is their name for a collection of symbols. Each block is a multiple of 16 code points (~ characters)
- UTF (Unicode transformation formats)
  - A 16-bit encoding, called UTF-16, is the default.
  - A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bi and uses 16 or 32 bits for the other characters.
  - UTF-32 uses 32 bits per character
- The Vietnamese alphabets are listed in several noncontiguous Unicode ranges:
  - Block “Basic Latin” {U+0000..U+007F}
  - Block “Latin-1 Supplement” {U+0080..U+00FF}
  - Block “Extended-A”, “Extended-B” {U+0100..U+024F}
  - Block “Extended Additional” {U+1E00..U+1EFF}
  - Block “Combining Diacritical Marks” {U+0300..U+036F}



fit@hcmus

Unicode Standard

- Glyphs is pictures representing characters
  - Characters are what you type, glyphs are what you see
- One glyph usually corresponds to one Unicode character. A glyph can also represent more than one character at once

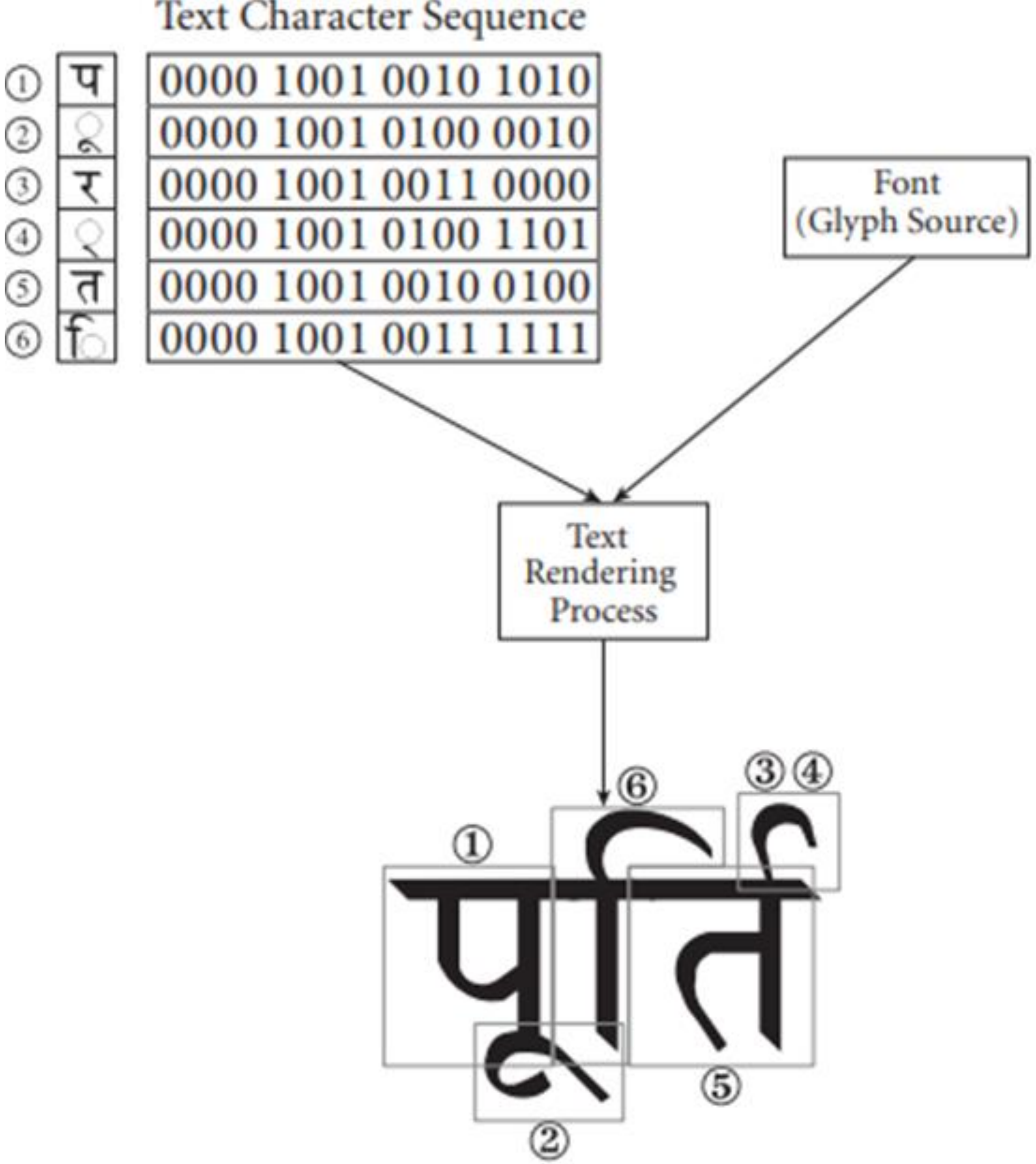


Glyph Info

dieresis

Name	Unicode	Char	Group	Subcategory	Script
Adieresis	00C4	Ä	Letter	Uppercase	latin
Edieresis	00CB	Ë	Letter	Uppercase	latin
Idieresis	00CF	Ï	Letter	Uppercase	latin
Odieresis	00D6	Ö	Letter	Uppercase	latin
Udieresis	00DC	Ü	Letter	Uppercase	latin
adieresis	00E4	ä	Letter	Lowercase	latin
edieresis	00EB	ë	Letter	Lowercase	latin
idieresis	00EF	ï	Letter	Lowercase	latin
odieresis	00F6	ö	Letter	Lowercase	latin
udieresis	00FC	ü	Letter	Lowercase	latin
ydieresis	00FF	ÿ	Letter	Lowercase	latin
Ydieresis	0178	Ÿ	Letter	Uppercase	latin
Udieresismacron	01D5	Ǖ	Letter	Uppercase	latin
udieresismacron	01D6	ǖ	Letter	Lowercase	latin
Udieresisacute	01D7	Ǘ	Letter	Uppercase	latin

81 entries | Add to Font



Unicode Character Code to Rendered Glyphs

fit@hcmus

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

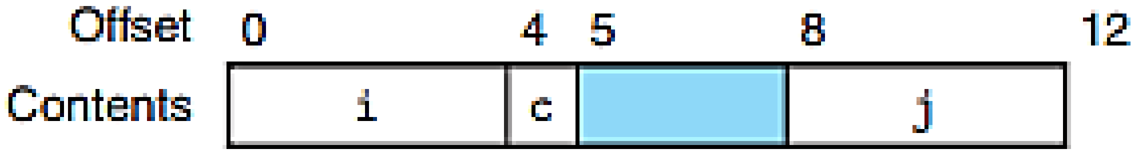
Data Alignment

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*