

LẬP TRÌNH HỆ THỐNG



nc.uit.edu.vn

TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM
KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS

Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM
Điện thoại: (08)3 725 1993 (122)

Bit, Bytes và Integers



Nội dung

- **Biểu diễn thông tin dưới dạng bit**
- **Tính toán bit**
- **Integers – Số nguyên**
 - Biểu diễn: không dấu (unsigned) và có dấu (signed)
 - Cộng, nhân, dịch bit
- **Biểu diễn trong bộ nhớ, con trỏ, chuỗi**



Câu hỏi có điểm cộng

Trong máy tính: Mọi thứ đều dưới dạng bit

- Mỗi bit bằng 0 hoặc 1
- Sử dụng các chuỗi bit, máy tính có thể:
 - Biểu diễn các lệnh (instructions) → xác định cần làm gì
 - Biểu diễn các số, chuỗi, mảng, v.v... → xác định cần dùng dữ liệu gì

Ví dụ: biểu diễn số trong hệ nhị phân

■ Biểu diễn số dưới dạng nhị phân

- Biểu diễn 15213_{10} dưới dạng nhị phân?

$$15213_{10} = 11101101101101_2$$

- $1.20_{10} = 1.0011001100110011[0011]\dots_2$
- $1.5213 \times 10^4 = 1.1101101101101_2 \times 2^{13}$

Ví dụ: chuỗi text của chương trình Hello.c

```
# i n c l u d e <sp> < > s t d i o .  
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46  
  
h > \n \n i n t <sp> m a i n ( ) \n {  
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123  
  
\n <sp> <sp> <sp> p r i n t f ( " h e l l  
10 32 32 32 112 114 105 110 116 102 40 34 104 101 108  
  
l o , <sp> w o r l d \ n " ) ; \n }  
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

Các hệ biểu diễn số?

Biểu diễn 15213_{10} ở các hệ biểu diễn số khác nhau?

■ Hệ thập phân – Decimal (Base 10)

15213_{10}

■ Hệ nhị phân – Binary (Base 2)

- Chỉ dùng 1 và 0 trong biểu diễn số
- *Từ hệ 10: Chia số 15213 cho 2, lưu lại số dư của mỗi lần chia và viết theo thứ tự ngược lại.*

$15213_{10} = 11101101101101_2$

■ Hệ thập lục phân – Hexadecimal (Base 16)

- Sử dụng các ký tự từ ‘0’ – ‘9’ và ‘A’ – ‘F’
- *Từ hệ 10: Chia số 15213 cho 16, lưu lại số dư của mỗi lần chia và viết theo thứ tự ngược lại. $10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F$.*
- *Từ hệ 2: Gom từ phải sang trái từng nhóm 4 bit và chuyển sang giá trị tương ứng ở hệ 16.*

$15213_{10} = 11\ 1011\ 0110\ 1101_2 = 3B6D_{16} = 0x3B6D$

Các hệ biểu diễn số trong Code C?

Khai báo biến ở các hệ biểu diễn?

- **Hệ thập phân:**
10, 110, 25, 97,...
- **Hệ thập lục phân:** thêm tiền tố **0x** phía trước số, không phân biệt hoa thường
0x10, 0x25, 0xA, 0xcd...
- **Hệ nhị phân:** thêm tiền tố **0b** hoặc **0B** trước số
0b10, 0B100101

Nội dung

- Biểu diễn thông tin dưới dạng bit
- Các phép tính toán bit
- Integers – Số nguyên
 - Biểu diễn: không dấu (unsigned) và có dấu (signed)
 - Cộng, nhân, dịch bit
- Biểu diễn trong bộ nhớ, con trỏ, chuỗi

Phép toán trên bit (Bit-wise operations)

- Thực hiện trên các bit nhị phân 0 hoặc 1
- Áp dụng các phép toán Boolean trên từng bit:

And (&)

- $A \& B = 1$ khi cả $A=1$ và $B=1$

&	0	1
0	0	0
1	0	1

Not (~)

- $\sim A = 1$ khi $A=0$

\sim	
0	1
1	0

Or (|)

- $A | B = 1$ khi hoặc $A=1$ hoặc $B=1$

	0	1
0	0	1
1	1	1

Exclusive-Or (Xor) (^)

- $A ^ B = 1$ khi A và B khác nhau, và ngược lại

^	0	1
0	0	1
1	1	0

Phép toán trên bit với chuỗi nhiều bit?

- Các phép toán trên bit có thể thực hiện trên **chuỗi các bit**
 - Thực hiện trên từng cặp 1-bit tương ứng

$$\begin{array}{r} 01101001 \\ \& 01010101 \\ \hline 01000001 \end{array} \quad \begin{array}{r} 01101001 \\ | 01010101 \\ \hline 01111101 \end{array} \quad \begin{array}{r} 01101001 \\ ^ 01010101 \\ \hline 00111100 \end{array} \quad \begin{array}{r} 01101001 \\ \sim 01010101 \\ \hline 10101010 \end{array}$$

Phép toán trên bit trong C

■ Các phép toán &, |, ~, ^ đều hỗ trợ trong C

- Có thể dùng với bất kỳ kiểu dữ liệu nào: long, int, short, char, ...
- Khi đó, mỗi số hạng được xem là chuỗi nhiều bit
- Phép toán được áp dụng trên từng bit

■ Ví dụ:

- $\sim 0x41 \& 0xBE$
 - $\sim 01000001_2 \& 10111110_2$
- $\sim 0x00 | 0xFF$
 - $\sim 00000000_2 | 11111111_2$
- $0x69 \& 0x55 ^ 0x41$
 - $01101001_2 \& 01010101_2 ^ 01000001_2$
- $0x69 | 0x55 \& 0x7D$
 - $01101001_2 | 01010101_2 \& 01111101_2$

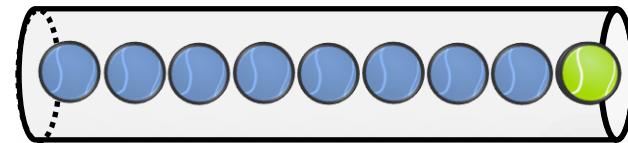
Các phép toán dịch bit (shift)

■ Dịch trái: $x \ll n$

- Dịch chuỗi bit biểu diễn x sang trái n lần
 - n bit bên trái bị bỏ đi
 - Điền vào bên phải n bit 0

■ Dịch phải: $x \gg n$

- Dịch chuỗi bit biểu diễn x sang phải n lần
 - n bit bên phải bị bỏ đi dần
- Dịch phải luận lý (Logical Shift Right – LSR)
 - Không quan tâm đến dấu của số x
 - Điền vào bên trái n bit 0
- Dịch phải toán học (Arithmetic Shift Right-ASR)
 - Quan tâm đến dấu của số x
 - Điền vào bên trái n bit dấu

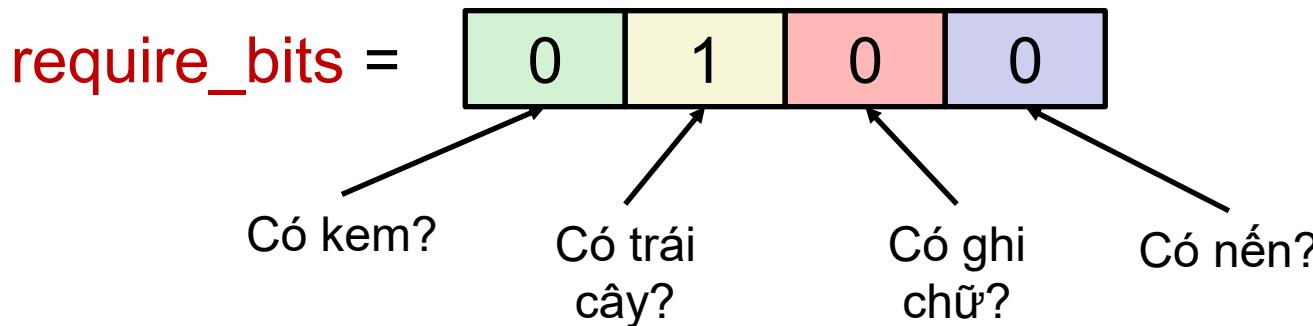


Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Phép toán trên bit: Ứng dụng (1)

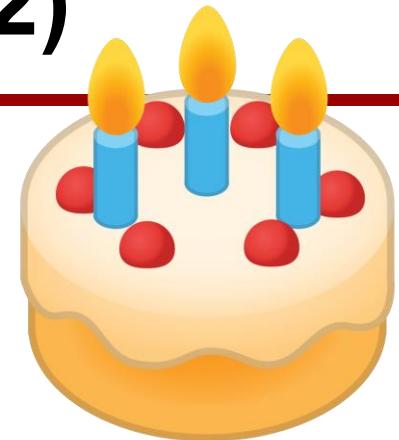
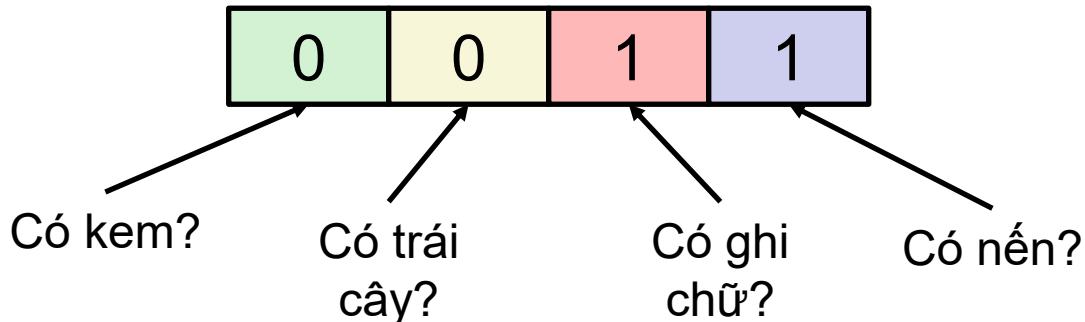
- **Case:** Dùng 1 số có **4 bit** đại diện cho các yêu cầu về đặc điểm của **1 cái bánh kem** được đặt trước.



- **Ví dụ:**

- 12 (1100): Bánh **có kem, có trái cây**, không ghi chữ và không nến
- 0 (0000): Bánh không ☺

Phép toán trên bit: Ứng dụng (2)



■ Case 1: KH muốn thêm trái cây cho bánh kem

- Giữ nguyên những yêu cầu còn lại → Chỉ cần **gán bit thứ 2 là 1**
- Giải pháp??

0	0	1	1
(or)	0	1	0
	0	1	1

(mask)

2 mask

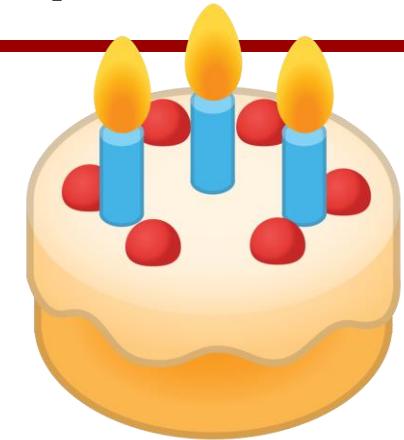
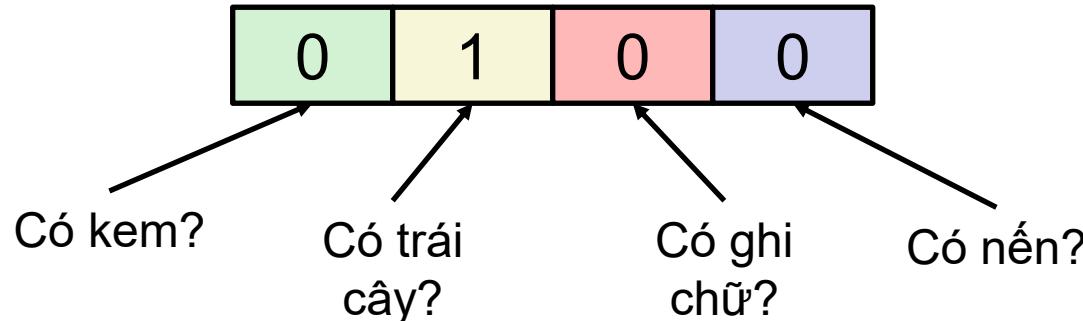
0	1	0	0
1	0	1	1

^ (xor) Hoặc | (or)

0	1	0	0
---	---	---	---

Muốn bật 1 bit bất kỳ của x thì dùng phép **or** x với **mask** có bit ở vị trí tương ứng là 1

Phép toán trên bit: Ứng dụng (3)



■ Case 2: Đổi yêu cầu thành không ghi chữ? ★

- Giữ nguyên những yêu cầu còn lại → Chỉ cần gán bit thứ 3 là 0
- Giải pháp??

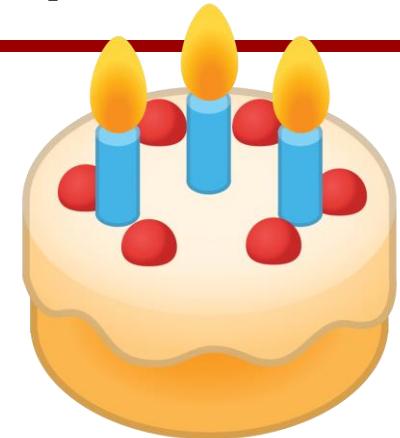
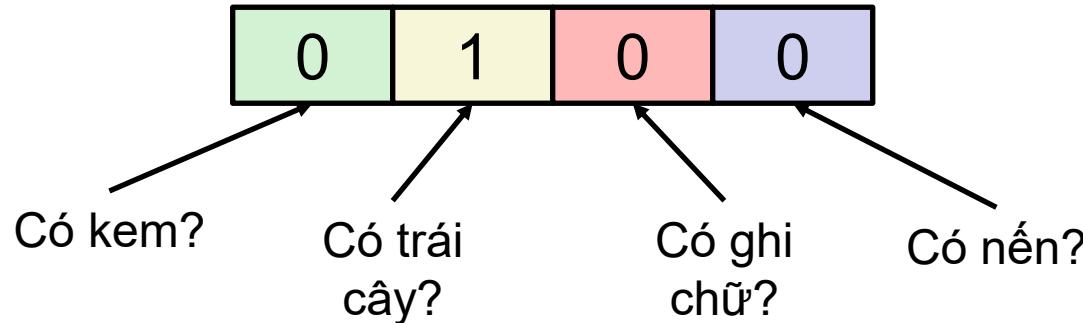
& (and)

0	1	1	1
1	1	0	1
0	1	0	1

(mask)

Muốn tắt 1 bit (gán bằng 0) bất kỳ của x thì dùng phép **and** x với **mask** có bit ở vị trí tương ứng là 0

Phép toán trên bit: Ứng dụng (4)



■ Case 3: Chỉ lấy yêu cầu về **có ghi chữ** của đơn hàng?

- Cần **lấy bit thứ 3** → giữ nguyên, các bit còn lại không lấy → đưa về 0
- Giải pháp??

& (and)

0	0	1	1
0	0	1	0
0	0	1	0

(mask)

★ Case 4: (*mở rộng*) Kiểm tra đơn **có ghi chữ** không??

Muốn lấy giá trị 1 bit bất kỳ của x thì **and** x với **mask** có bit ở vị trí tương ứng là 1 và dùng phép dịch phải để đưa về ngoài cùng bên phải (nếu cần)

Phép toán trên bit: Ứng dụng (5)

- **Các phép dịch bit (shift):** Các phép nhân và chia với luỹ thừa của 2 (2^n)

$$a \ll n \Leftrightarrow a * 2^n$$

$$a \gg n \Leftrightarrow a / 2^n$$

Lưu ý: dễ nhầm lẫn với Phép toán logic trong C

■ Khác biệt của các phép toán Logic

- `&&`, `||`, `!`
 - Vẫn áp dụng các phép boolean
 - **Xem 0 là False**
 - **Các giá trị khác 0 là True**
 - **Chỉ trả về 0 hoặc 1**
 - Điều kiện kết thúc sớm của if

■ Ví dụ:

- `!0x41 & 0x00`
- `!0x00 | 0x01`
- `0x69 && 0x55 | 0x01`
- `p && *p` (tránh truy xuất con trỏ có giá trị null)

Phép toán	Phép toán trên bit	Phép toán logic
AND	<code>&</code>	<code>&&</code>
OR	<code> </code>	<code> </code>
NOT	<code>~</code>	<code>!</code>
XOR	<code>^</code>	

Phép toán trên bit vs Phép toán logic trong C

■ Ví dụ so sánh

x	y	Phép toán trên bit	Phép toán logic
0x41	0x10	$0x41 \& 0x10$ = 0100 0001 & 0001 0000 = 0000 0000 = 0x0	$0x41 \&& 0x10$ = 0x1 && 0x1 = 0x1
0x41	0x10	$0x41 0x10$ = 0100 0001 0001 0000 = 0101 0001 = 0x51	$0x41 0x10$ = 0x1 0x1 = 0x1
0x41		$\sim 0x41$ = ~0100 0001 = 1011 1110 = 0xBE	$!0x41 =$!0x1 = 0x0

Ví dụ: If nào true?

1. if (1 & 6)
 printf("true");

3. if (1 ^ 6)
 printf("true");

2. if (1 && 6)
 printf("true");

4. if (1 == 6)
 printf("true");

Nội dung

- Biểu diễn thông tin dưới dạng bit
- Tính toán bit
- Integer – Số nguyên
 - Biểu diễn: không dấu (**unsigned**) và có dấu (**signed**)
 - Cộng, nhân, dịch bit
- Biểu diễn trong bộ nhớ, con trỏ, chuỗi

Biểu diễn số nguyên (integer)

- **Quy ước:** trong hệ biểu diễn w -bit, các bit được đánh thứ tự từ 0 đến $w-1$ từ phải sang trái.

Ví dụ: Hệ 8-bit ($w=8$) 0 1 0 0 1 1 1 0

7 6 5 4 3 2 1 0
—————
←

- **Số không dấu (unsigned)**

- Tất cả các bit đều biểu diễn giá trị
- Tính giá trị:

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

- **Số có dấu (signed)**

- Bit trọng số cao nhất ($w-1$) biểu diễn dấu
 - 0: không âm
 - 1: âm

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



Bit dấu

Biểu diễn số nguyên (integer): Ví dụ

- Trong hệ biểu diễn 8-bit có dấu, đây là những số nguyên nào?

- 0000 0110 = 6
- 0001 0101 = 21
- 1100 0001 = -63
- 1000 1010 = -118

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

Biểu diễn số nguyên – Giới hạn biểu diễn?

- **Quy ước:** trong hệ biểu diễn w -bit, các bit được đánh thứ tự từ 0 đến $w-1$ từ phải sang trái.

■ Số không dấu (**unsigned**)

- Giá trị lớn nhất? Tất cả các bit là 1 = $2^w - 1$
- Giá trị nhỏ nhất? Tất cả các bit là 0 = 0

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

■ Số có dấu (**signed**)

- Bit trọng số cao nhất ($w-1$) biểu diễn dấu
- Giá trị lớn nhất? Bit dấu là 0, tất cả các bit còn lại là 1 = $2^{w-1} - 1$
- Giá trị nhỏ nhất? Bit dấu là 1, tất cả các bit còn lại là 0 = -2^{w-1}

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

Biểu diễn số đối (negation): Ví dụ (1)

■ Biểu diễn các số (hệ biểu diễn 16-bit):

- $x = 15213 = 0011\ 1011\ 0110\ 1101$
- $y = -15213 = \text{Biểu diễn bù 2 của } 15213$

B1: Thực hiện phép \sim trên biểu diễn nhị phân của 15213

$$\sim x = \sim 0011\ 1011\ 0110\ 1101 = 1100\ 0100\ 1001\ 0010$$

B2: Cộng thêm 1 vào bit thấp nhất bên phải

$$\begin{aligned}\sim x + 1 &= 1100\ 0100\ 1001\ 0010 + 1 \\ &= 1100\ 0100\ 1001\ 0011\end{aligned}$$

1100 0100 1001 0011 chính là biểu diễn của -15213

Với số nguyên x : $-x = \sim x + 1$

Biểu diễn số đối (negation): Ví dụ (2)

x = 15213:

00111011 01101101

y = -15213:

11000100 10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		15213	-15213	

Biểu diễn số đối (negative)

■ Biểu diễn số (hệ 32 bit):

$$\begin{aligned}x = -1 &= \sim(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001) + 1 \\&= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110 + 1 \\&= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \\&= 0xFFFFFFFF\end{aligned}$$

$$\begin{aligned}y = -128 &= \sim(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000\ 0000) + 1 \\&= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0111\ 1111 + 1 \\&= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000\ 0000 \\&= 0xFFFFF80\end{aligned}$$

Biểu diễn số không và có dấu

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

■ Tương đương

- Các số không âm có biểu diễn giống nhau trong cả trường hợp có và không dấu

■ Duy nhất

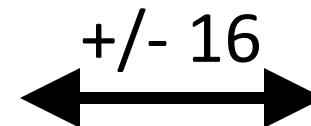
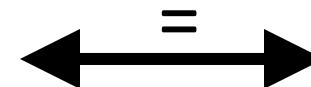
- Mỗi chuỗi bit biểu diễn một giá trị số duy nhất
- Mỗi giá trị biểu diễn được có duy nhất một chuỗi biểu diễn

Ánh xạ giữa số có và không dấu (1)

- *Cùng 1 chuỗi bit → tương ứng giá trị bao nhiêu trong biểu diễn có dấu và không dấu?*
- **Nguyên tắc:**
 - Trường hợp **chuỗi biểu diễn có bit trọng số cao nhất là 0**, giá trị khi biểu diễn không và có dấu là **như nhau**.
 - Ngược lại, **bit trọng số cao nhất là 1**:
 - Giữ nguyên chuỗi bit biểu diễn
 - Thay đổi giá trị của số theo bit cao nhất
- **Trong hệ sử dụng n bit để biểu diễn số**, với mỗi chuỗi biểu diễn có **bit trọng số cao nhất = 1**:
 - Giá trị không dấu (unsigned) = giá trị có dấu (signed) **+ 2^n**
 - Giá trị có dấu (signed) = giá trị không dấu (unsigned) **- 2^n**

Ánh xạ giữa số có và không dấu (2)

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15



Thêm: số không và có dấu trong C

- Mặc định trong C, các số nguyên là số nguyên **có dấu (signed)**
- **Số nguyên không dấu (unsigned)**: thêm hậu tố **U** phía sau:

0U, 4294967259U

- Ép kiểu giữa unsigned và signed trong C tương tự như phép ánh xạ giá trị.
- Lưu ý: trong biểu thức chứa cả số có dấu và không dấu, các số có dấu sẽ được chuyển sang không dấu
 - <, >, ==, <=, >=

Số không và có dấu trong C

■ Cho đoạn mã C

```
#include <stdio.h>

int main()
{
    int a;
    unsigned int b;

    printf("Enter an integer: ");
    scanf("%d", &a);
    printf("Enter an unsigned integer: ");
    scanf("%u", &b);

    printf("Your a: %d", a);
    printf("Your b: %u", b);
}
```

Nhập **a = -1** và **b = -1**, dự đoán và giải thích kết quả chương trình?

A. Chương trình lỗi

B. Chương trình in ra
2 giá trị -1

C. Chương trình in ra giá
trị -1 và 1 giá trị khác

Nội dung

- Biểu diễn thông tin dưới dạng bit
- Tính toán bit
- **Integers – Số nguyên**
 - Biểu diễn: không dấu (unsigned) và có dấu (signed)
 - **Cộng, nhân, dịch bit**
- Biểu diễn trong bộ nhớ, con trỏ, chuỗi

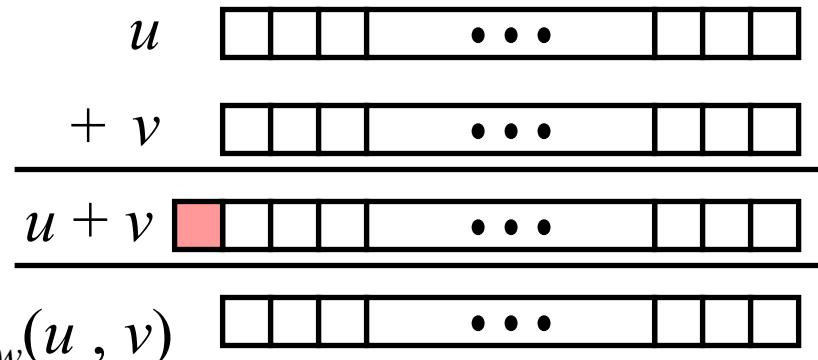
Phép cộng

■ Cộng (hệ biểu diễn w -bit)

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits



- Tổng thực tế có thể yêu cầu $w+1$ bit, tuy nhiên hệ biểu diễn w bit bỏ bit cao nhất (MSB).
→ Tràn số (overflow)

Tràn số trong phép cộng: Ví dụ

- Giả sử dùng **4 bit** để biểu diễn số.
 - Không dấu (unsigned): biểu diễn từ 0 đến 15
 - Có dấu (signed): biểu diễn từ -8 đến +7.
- **Cộng số không dấu** (unsigned):
 - $8 + 8 = 1000 + 1000 = \textcolor{red}{1}0000 = 0$
 - $9 + 10 = 1001 + 1010 = \textcolor{red}{1}0011 = 3$
- **Cộng số có dấu** (signed):
 - $7 + 7 = 0111 + 0111 = 1110 = \textcolor{red}{-2}$
 $\rightarrow \text{sum} > \text{giá trị dương lớn nhất} \rightarrow \text{tràn số âm}$
 - $-5 + -5 = 1011 + 1011 = \textcolor{red}{1}0110 = 6$
 $\rightarrow \text{sum} < \text{giá trị âm nhỏ nhất} \rightarrow \text{tràn số dương}$

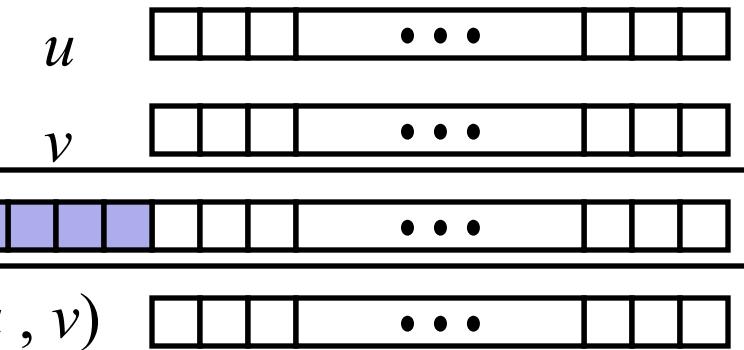
Phép nhân

■ Nhân (hệ biểu diễn w -bit)

Operands: w bits

True Product: 2^w bits

Discard w bits: w bits



- Tích thực tế có thể yêu cầu 2^w bit, tuy nhiên hệ biểu diễn w bit bỏ các bit cao hơn w .
 - Tràn số (overflow)
- Phép nhân có thể khác nhau trong một vài trường hợp của số có dấu và không dấu
 - Các bit thấp vẫn giống nhau

Phép nhân với 2^n bằng shift trái (1)

- $u \ll k$ tương đương với $u * 2^k$
- Áp dụng được cho cả số nguyên có dấu (signed) và không dấu (unsigned)
- Với u được biểu diễn bằng w bit, kết quả có thể cần $w + k$ bit để biểu diễn \rightarrow tràn số
- Ví dụ:
 - $u \ll 3 == u * 8$
 - $(u \ll 5) - (u \ll 3) == u * 24$

Phép nhân với 2^n bằng shift trái (2)

- Hầu hết các máy tính thực hiện **shift** và **cộng** nhanh hơn phép **nhân**
- Compiler tự động tạo ra mã **shift/cộng** (*nếu được*) khi nhân **hằng số**

Hàm C

```
long mul9(long x)
{
    return x*9;
}
```

Các lệnh toán học được biên dịch

```
movq %rax, %rdx
salq $3, %rax
addq %rdx, %rax
```

Giải thích

```
t = x;
x = x << 3; # 8x
x += t;      # 8x + x
return t;
```

Phép nhân với 2^n bằng shift trái (3)

- Hầu hết các máy tính thực hiện **shift** và **cộng** nhanh hơn phép **nhân**
- Compiler tự động tạo ra mã **shift/cộng** (*nếu được*) khi nhân **hằng số**

Hàm C

```
long mul12(long x)
{
    return x*12;
}
```

Các lệnh toán học được biên dịch

```
leaq (%rax, %rax, 2), %rax
salq $2, %rax
```

Giải thích

```
t = x + x*2;
return t << 2;
```

Phép chia không dấu cho 2^n bằng shift phải (1)

■ $u >> k$ tương đương với $u / 2^k$

- Giá trị **nguyên** của phép chia ($[u/2^k]$)
- Sử dụng shift luận lý (logic shift)
 - Không quan tâm đến dấu
 - Diền bit 0 dần vào các bit trọng số cao bên trái

	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Phép chia không dấu cho 2^n bằng shift phải (2)

Hàm C

```
unsigned long udiv8
(unsigned long x)
{
    return x/8;
}
```

Mã assembly đã biên dịch

```
shrq $3, %rax
```

Giải thích

```
# Logical shift
return x >> 3;
```

- Sử dụng shift luận lý với số unsigned
- Trong Java
 - Logical shift ký hiệu là >>>

Nội dung

- Biểu diễn thông tin dưới dạng bit
- Tính toán bit
- Integers – Số nguyên
 - Biểu diễn: không dấu (unsigned) và có dấu (signed)
 - Cộng, nhân, dịch bit
- Biểu diễn trong bộ nhớ, con trỏ, chuỗi

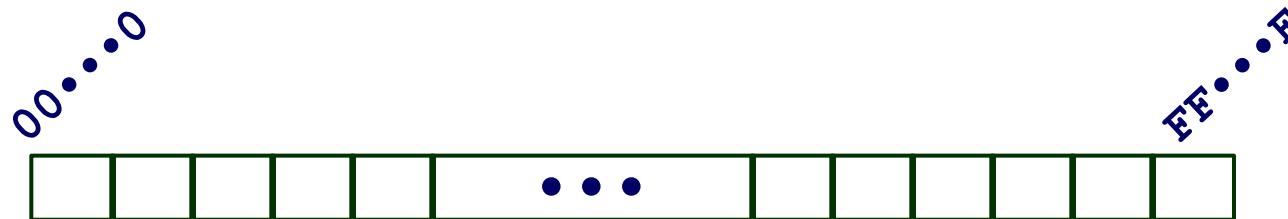
Bytes

■ Byte = 8 bits

- Biểu diễn giá trị nhị phân từ 00000000_2 đến 11111111_2
- Trong hệ 10 (decimal): giá trị từ 0_{10} đến 255_{10}
- Trong hệ 16 (hexadecimal): 00_{16} đến FF_{16}

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

Tổ chức bộ nhớ theo byte



- **Bộ nhớ “như” một mảng byte rất lớn**
- **Mỗi byte trong bộ nhớ được xác định bằng địa chỉ**
 - Một địa chỉ như một index trong mảng byte đó
 - Kiểu dữ liệu *pointer* (*con trỏ*) dùng để chứa một địa chỉ: char*, int*...
- **Lưu ý: hệ thống cung cấp các không gian địa chỉ riêng cho mỗi “tiến trình”**
 - 1 tiến trình = 1 chương trình được thực thi

Các kiểu dữ liệu

Đơn vị: bytes

Kiểu dữ liệu C	Hệ thống 32-bit	Hệ thống 64-bit	Hệ thống 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
long double	-	-	10/16
pointer	4	8	8

Kích thước phụ thuộc vào kích thước của 1 địa chỉ

Word trong máy tính

■ Một máy tính có 1 “word size”

- Kích thước của 1 địa chỉ
 - Hệ thống dùng bao nhiêu bit (bytes) để đánh địa chỉ trong bộ nhớ?
- Hầu hết các máy tính có word size 32 bits (4 bytes)
- Ngày càng nhiều các máy tính có word size 64 bits (8 bytes)

Biểu diễn con trỏ (pointer)

```
int B = -15213;  
int *P = &B;
```

Sun (32-bit)

P = 0xFFFFFB2C

IA32 (32-bit)

P = 0xFFF528AC

x86-64 (64-bit)

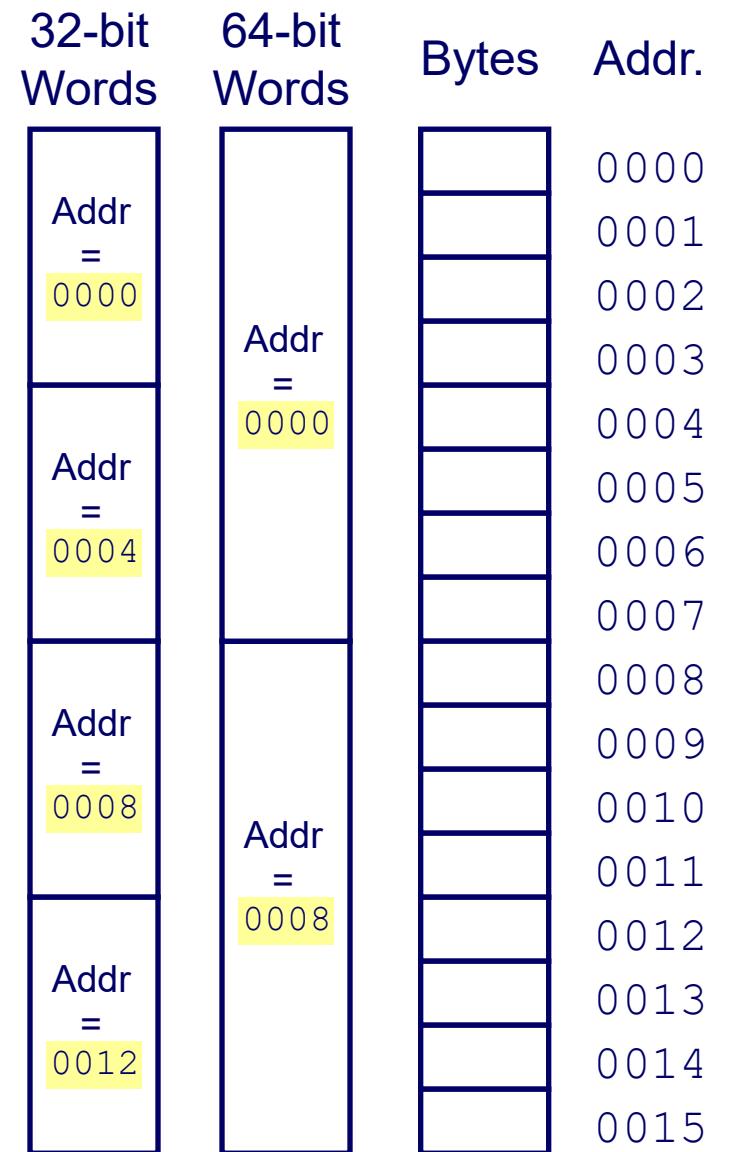
P = 0x00007FFD82FE1B3C

- Các compilers và máy tính khác nhau sẽ gán những vị trí khác nhau cho các object.
- Thậm chí khác nhau trong mỗi lần chạy chương trình.

Tổ chức bộ nhớ theo word

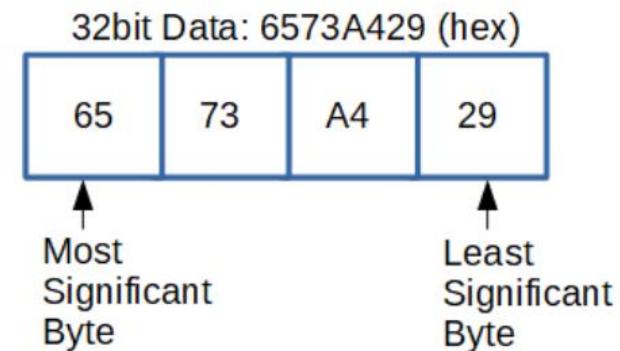
■ Địa chỉ xác định vị trí của byte

- Địa chỉ của byte đầu tiên trong word
- Địa chỉ của các word tiếp theo cách nhau 4 (32 bit) hoặc 8 (64 bit)



Thứ tự byte – Byte ordering

- Bộ nhớ như một mảng lưu các byte liên tục
→ Vậy với **một dữ liệu gồm nhiều byte**, các byte sẽ được lưu trữ theo thứ tự nào trong bộ nhớ?

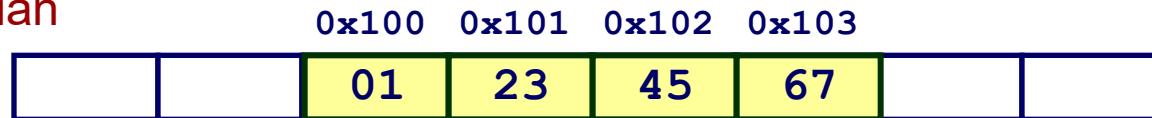


- 2 dạng:
 - **BigEndian**: byte có trọng số thấp nhất nằm ở địa chỉ cao nhất
 - Sun, PPC Mac, Internet
 - **LittleEndian**: byte có trọng số thấp nhất nằm ở địa chỉ thấp nhất
 - x86, bộ xử lý ARM chạy Android, iOS và Windows

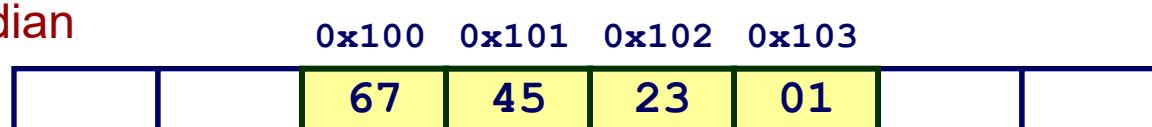
Thứ tự byte – Byte ordering: Ví dụ

- Cho biến **x** có giá trị **0x1234567**
- Địa chỉ để lưu **x** là **0x100**
- Byte thấp nhất 0x67 sẽ lưu ở đâu?**

BigEndian



LittleEndian



Ví dụ: Biểu diễn và lưu trữ số nguyên

■ Cho

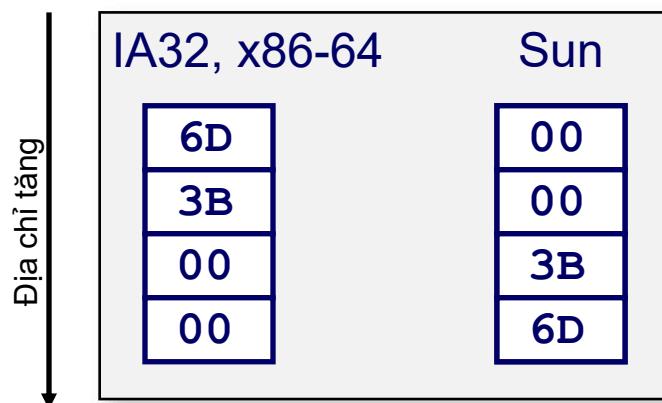
int A = 15213 = 0x00003B6D;

int B = -15213 = 0xFFFFC493;

Lưu trữ A, B như thế nào trong các hệ thống:

- IA32, x86-64 (Little Endian)
- Sun (Big Endian)?

int A = 15213;



Decimal: 15213

Binary: 0... 0011 1011 0110 1101

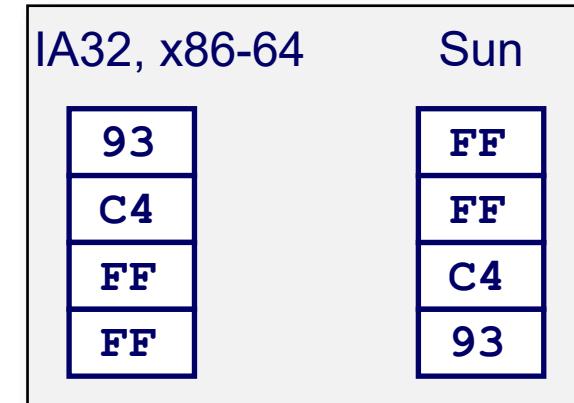
Hex: 00... 3 B 6 D

Decimal: -15213

Binary: 1... 1100 0100 1001 0011

Hex: FF... C 4 9 3

int B = -15213;



Ví dụ: Code hiển thị byte của 1 dữ liệu (1)

- Code in biểu diễn dưới dạng các byte với đúng thứ tự trong bộ nhớ của dữ liệu
 - Tham số **start** là vị trí lưu của dữ liệu
 - Vì sao phải dùng kiểu **unsigned char***?
 - Giả sử kiểu dữ liệu là **int**, start sẽ là **int***, start[i] sẽ cách nhau mỗi 4 bytes
 - Với ép kiểu pointer sang **unsigned char***, start[i] sẽ cách nhau 1 byte → truy xuất được từng byte của dữ liệu với i

Trong hàm **printf**:

%p: Print pointer

%x: Print Hexadecimal

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Ví dụ: Code hiển thị byte của 1 dữ liệu (2)

```
int a = 15213;  
printf("int a = 15213;\\n");  
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
    int a = 15213;  
    0x7ffb7f71dbc      6d  
    0x7ffb7f71dbd      3b  
    0x7ffb7f71dbe      00  
    0x7ffb7f71dbf      00
```

Biểu diễn chuỗi (strings)

■ String trong C

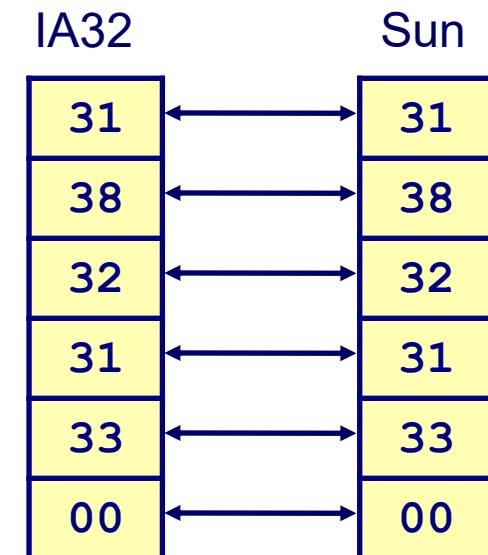
- Là một mảng các ký tự
- Mỗi ký tự ở dạng mã ASCII
 - Chuẩn 7-bit
 - Ký tự '0' tương ứng mã 0x30
 - Số i tương ứng với mã $0x30 + i$
- String cần được kết thúc bằng null
 - Ký tự cuối cùng là giá trị 0 (\neq ký tự '0')

■ Lưu ý

- Thứ tự byte của hệ thống không ảnh hưởng đến cách lưu chuỗi
 - Ký tự đầu tiên **luôn luôn** lưu ở địa chỉ thấp nhất

```
char S[6] = "18213";
```

```
0x31 0x38 0x32 0x31 0x33
```



Nội dung thêm

- Phép chia có dấu cho 2^n bằng shift phải
- Đọc các giá trị gồm nhiều bytes trong assembly

Phép chia có dấu cho 2^n bằng shift phải

■ $u >> k$ tương đương với $u / 2^k$

- Giá trị **nguyên** của phép chia
- Sử dụng shift toán học
 - Có quan tâm đến dấu
 - Diền bit dấu dần vào các bit trọng số cao bên trái
- Làm tròn sai trong trường hợp $u < 0$!

	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

Phép chia có dấu cho 2^n ĐÚNG

■ Phép chia $u / 2^k$ của số âm

- Giá trị nguyên của phép chia làm tròn về 0
- Cách tính: $[(x + 2^k - 1) / 2^k]$
 - Trong C: $(x + (1 << k) - 1) >> k$
 - Đưa số bị chia dần về 0

	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

Ví dụ: Code Phép chia có dấu cho 2^n

C Function

```
long idiv8(long x)
{
    return x/8;
}
```

Mã assembly được biên dịch

```
testq %rax, %rax
js L4
L3:
sarq $3, %rax
ret
L4:
addq $7, %rax
jmp L3
```

Giải thích ý nghĩa

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Sử dụng shift toán học cho int
- Trong Java
 - Shift toán học ký hiệu là $>>$

Đọc các giá trị gồm nhiều byte trong assembly

■ Disassembly

- Biểu diễn dưới dạng text các mã máy nhị phân
- Tạo bởi các chương trình đọc mã máy

■ Ví dụ

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0,0x28(%ebx)

■ Giải mã số

- Giá trị:
- Mở rộng thành 32 bits:
- Chia thành nhiều bytes:
- Đảo thứ tự:

0x12ab

0x000012ab

00 00 12 ab

ab 12 00 00

Nội dung tự tìm hiểu: Floating point

- Cách biểu diễn số thực: Floating point
- Phép cộng với số thực
- (Optional) Bài tập thu hoạch



Giải thích vì sao có sự khác biệt bên dưới?

$$(1e20 + -1e20) + 3.14 = \textcolor{red}{3.14}$$

$$1e20 + (-1e20 + 3.14) = \textcolor{red}{0}$$

Nội dung

■ Các chủ đề chính:

- 1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
- 2) Ngôn ngữ assembly
- 3) Điều khiển luồng trong C với assembly
- 4) Các thủ tục/hàm (procedure) trong C ở mức assembly
- 5) Biểu diễn mảng, cấu trúc dữ liệu trong C
- 6) Một số topic ATTT: reverse engineering, bufferoverflow
- 7) Phân cấp bộ nhớ, cache
- 8) Linking trong biên dịch file thực thi

■ Lab liên quan

- Lab 1: Nội dung 1
- Lab 2: Nội dung 1, 2, 3
- Lab 3: Nội dung 1, 2, 3, 4, 5, 6
- Lab 4: Nội dung 1, 2, 3, 4, 5, 6
- Lab 5: Nội dung 1, 2, 3, 4, 5, 6
- Lab 6: Nội dung 1, 2, 3, 4, 5, 6

Môi trường - Công cụ hỗ trợ

■ Hệ điều hành Linux

- Máy ảo/thật
- Hệ thống 32/64 bit
- (Khuyến khích) Tương tác qua giao diện command



Linux

■ GCC - Trình biên dịch C trên Linux

■ Các IDE lập trình

■ Phần mềm dịch ngược:

- IDA Pro (GUI)
- GDB (command line)

IDA



Đánh giá

30% quá trình/giữa kỳ + 20% thực hành + 50% cuối kỳ

❑ **Quá trình/giữa kỳ:**

- Bài tập assignment trên lớp
- Kiểm tra giữa kỳ

❑ **Thực hành:**

- 6 labs
- Vắng từ 3 buổi thực hành trở lên → trừ tối thiểu **1/3** số điểm

❑ **Cuối kỳ:**

- Trắc nghiệm + Tự luận
- Có thể cho phép sử dụng 01 tờ A4 viết tay

Yêu cầu

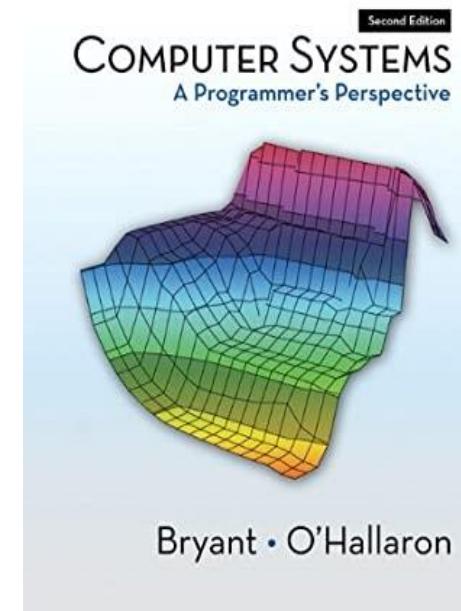
- Đến lớp đúng giờ
- Tìm hiểu trước bài giảng
- Thực hiện đủ Bài tập trên lớp
- Khi làm nhóm:
 - Không ghi nhóm → sao chép
 - Sao chép bài → **0**

Giáo trình

■ Giáo trình chính

Computer Systems: A Programmer's Perspective

- Second Edition (CS:APP2e), Pearson, 2010
- Randal E. Bryant, David R. O'Hallaron
- <http://csapp.cs.cmu.edu>
- Slide: **Tiếng Việt** (+ Tiếng Anh)
 - Giáo trình của ĐH Carnegie Mellon (Mỹ)



■ Tài liệu khác

- *The C Programming Language*, Second Edition, Prentice Hall, 1988
 - Brian Kernighan and Dennis Ritchie
- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 1st Edition, 2008
 - Chris Eagle
- *Reversing: Secrets of Reverse Engineering*, 1st Edition, 2011
 - Eldad Eilam



KEEP
CALM
AND
ENJOY YOUR
SEMESTER :)