Biểu diễn các kiểu dữ liệu và các phép toán thao tác

Nội dung chương này dựa trên quyển Computer Systems: A programmer's perspective của R.E Bryant & D.R O'Hallaron

Link tự học thêm ngôn ngữ C: https://codelearn.io/learning/c-for-beginners

Table of Contents

Vì sao cần hiểu rõ cách biểu diễn dữ liệu

Trước hết chúng ta xem một vài ví dụ sau.

Ví dụ 1: sau phép gán, giá trị của biến a sẽ bằng bao nhiêu?

(1) `float a = (3.14+1e20)-1e20; //` 1.0×10^{20}

(2) `float a = 3.14+(1e20-1e20);`

Ví dụ 2: đoạn chương trình sau sẽ in ra nội dung gì?

```
float a = 0.1;
if (a+a == 0.2) printf(“Yes”) else printf(“No”);
```

Ví dụ 3: đoạn chương trình sau sẽ in ra nội dung gì?

```
unsigned int a = 1;
int b = -1;
if (a>b) printf("a is bigger\n”);
else printf("a is smaller\n);
```

Việc hiểu rõ cách biểu diễn dữ liệu giúp

- Tránh lỗi lập trình khi thực hiện các phép tính toán khoa học, gồm các lỗi tràn giá trị, lỗi chuyển kiểu dữ liệu, lỗi tính toán có vẻ đúng về mặt logic nhưng sai khi cài đặt
- Tránh các lỗi tràn bộ đệm, tạo lỗ hổng phần mềm. Hacker luôn tìm các lỗi phần mềm để tìm cách đột nhập vào hệ thống
- Thực hiện tính toán với các số lớn, tính toán hiệu quả khi thao tác trực tiếp ở mức bit, phù hợp với các ứng dụng mật mã

Các khái niệm cơ bản

Một từ trong máy tính (word)

Là kích thước bit tối đa mà bộ vi xử lý có thể sử dụng khi thực hiện các phép toán. Word cũng được hiểu là kích thước cơ sở của một số nguyên, một con trỏ trong máy tính. Hiện tại, các bộ vi xử lý có kích thước word là 32 hoặc 64 bit.

Các kiểu dữ liệu cơ bản

C declaration	32-bit	64-bit
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char *`	4	8
`float`	4	4
`double`	8	8

Một mẹo nhỏ để biết được word size của CPU là đo độ dài của 1 con trỏ:

```
printf("This CPU is %d-bit\n", (int)sizeof(void*)*8);
```

Biểu diễn xâu ký tự

Trong C, xâu ký tự là một mảng các ký tự kết thúc bởi ký tự NULL, có mã ascii 0. Các kiểu dữ liệu số nguyên và thập phân sẽ được trình bày chi tiết trong các phần tiếp theo.

Bài tập: In dưới dạng hexa giá trị các byte biểu diễn xâu ký tự, kể cả ký tự kết thúc (print_string_bytes.c)

char pMyString[]=”1234”;

Biểu diễn số nguyên

C data type	Minimum	Maximum
`char`	−128	127
`unsigned char`	0	255
`short [int]`	−32,768	32,767
`unsigned short [int]`	0	65,535
`int`	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	4,294,967,295
`long [int]`	−2,147,483,648	2,147,483,647
`unsigned long [int]`	0	4,294,967,295
`long long [int]`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned long long [int]`	0	18,446,744,073,709,551,615

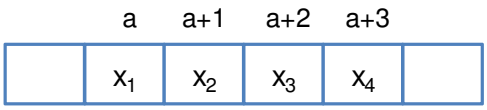
Figure 2.8 **Typical ranges for C integral data types on a 32-bit machine.** Text in square brackets is optional.

Thứ tự các bytes

Một biến *int x* gồm 4 bytes x4, x3, x2, x1 được đặt tại địa chỉ a sẽ chiếm các bytes từ a => a+3. Tùy theo bộ vi xử lý mà thứ tự của các bytes sẽ khác nhau.

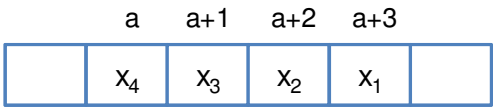
Quy ước Little Endian (Intel)

Byte có ý nghĩa giá trị cao được đặt tại vị trí địa chỉ cao. Vậy thứ tự các bytes đặt tại địa chỉ a sẽ là: (a:x1), (a+1:x2), (a+2:x3), (a+3:x4).



Quy ước Big Endian (IBM, Sun Micro System)

Byte có ý nghĩa giá trị cao sẽ được đặt trước, tức là tại vị trí địa chỉ thấp. Vậy thứ tự các bytes đặt tại địa chỉ a sẽ là: (a:x4), (a+1:x3), (a+2:x2), (a+3:x1).



Bi-endian (ARM)

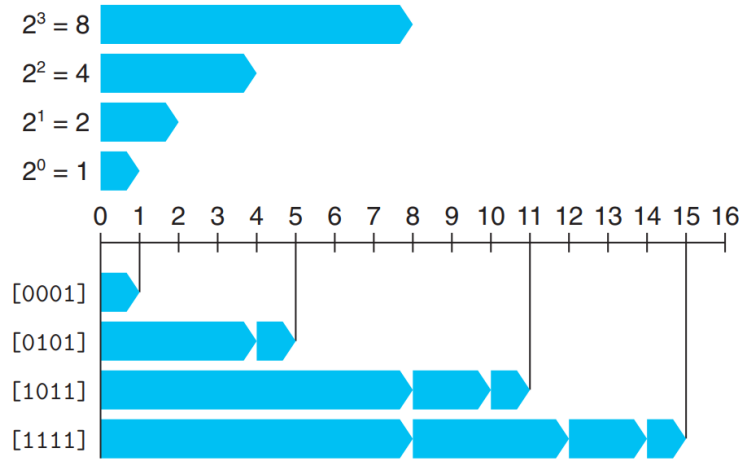
Cho phép cấu hình để chạy theo Little hoặc Big Endian

Bài tập: In các giá trị của từng byte biểu diễn dữ liệu biến nguyên a và địa chỉ của biến đó ra màn hình, dưới dạng hexa (print_int_bytes.c)

Số nguyên không dấu (unsigned)

Chuỗi bit $x = [x_{w-1}, x_{w-2}, \dots, x_0]$ biểu diễn cho giá trị $\sum_0^{w-1} x_i . 2^i$

Ví dụ: với w=4



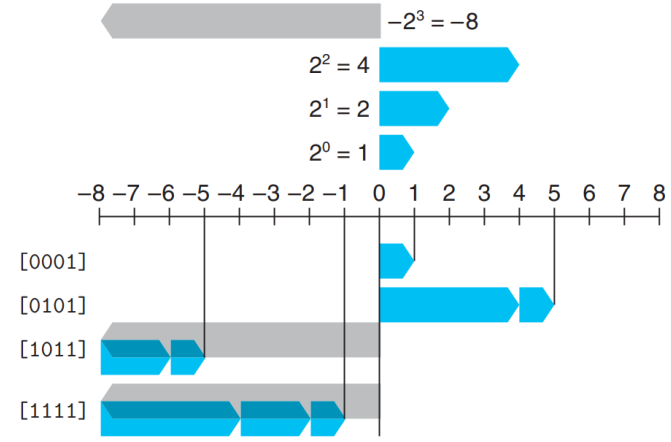
Số nguyên có dấu

Sử dụng mã hóa bù 2 (two's complement)

Chuỗi bit $x = [x_{w-1}, x_{w-2}, \dots, x_0]$ biểu diễn cho giá trị $-x_{w-1}2^{w-1} + \sum_0^{w-2} x_i . 2^i$

Với cách biểu diễn này các số âm có thứ tự sắp xếp như số nguyên không dấu, do vậy có thể dùng chung 1 thuật toán. Tuy nhiên cần phân biệt số âm và không âm khi sắp xếp.

Ví dụ: với w=4



Các giá trị cần lưu ý

	Word size w			
Value	8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	−128	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
−1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Figure 2.13 **Important numbers.** Both numeric values and hexadecimal representations are shown.

Các hằng được định nghĩa sẵn trong C, thư viện <limits.h>

INT_MAX: giá trị nguyên có dấu cực đại (Tmax, Two's complement max)

INT_MIN: giá trị nguyên có dấu cực tiểu (Tmin, Two's complement min)

UINT_MAX: giá trị nguyên không dấu cực đại (Umax, Unsigned max)

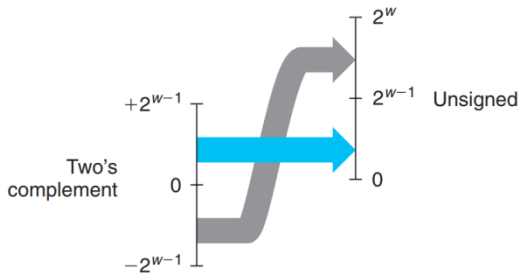
Chuyển đổi kiểu giữa không dấu và có dấu

Phép chuyển đổi kiểu (casting) trong C luôn giữ nguyên giá trị bit của biến và chỉ thay đổi cách các bit đó được diễn giải thành giá trị số như thế nào. Ví dụ

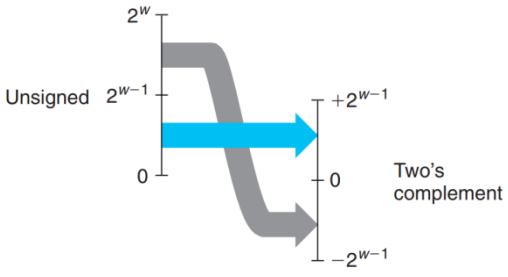
```
unsigned u = 4294967295u; // UMax_32
int tu = (int) u;
printf("u = %u, tu = %d\n", u, tu);
printf("u = %x, tu = %x\n", u, tu);
Kết quả u=4294967295, tu=-1 và cả hai đều có biểu diễn bit là FF
FF FF FF
```

Công thức chuyển đổi

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad (2.6)$$



$$U2T_w(u) = \begin{cases} u, & u < 2^{w-1} \\ u - 2^w, & u \geq 2^{w-1} \end{cases} \quad (2.8)$$



So sánh giá trị số nguyên có và không dấu

Đây là phép toán gây mù mờ về mặt ngữ nghĩa và kiểm soát lỗi. Theo mặc định ngôn ngữ, số nguyên có dấu sẽ được chuyển về kiểu không dấu trong phép so sánh. Tốt nhất là chỉ nên thực hiện so sánh các giá trị thuộc cùng một kiểu.

Ví dụ: compare_int_uint.c

```
void main()
{
	unsigned int a = 1;
	int b = -1;
	if (a>b) printf("a is BIG, a-b = %d\n", a-b);
	else printf("a is SMALL, a-b = %d\n", a-b);
}
```

Các thao tác mức bit

Toán tử logic AND, OR, XOR, NOT: &, |, ^, ~

Bài tập: (i) in giá trị của 1 byte theo biểu diễn nhị phân; (ii) đổi trực tiếp giá trị 2 biến nguyên

Thao tác dịch bit SHIFT LEFT << k

Dịch trái k bit. Các bit bên trái vượt giới hạn kích thước biểu diễn dữ liệu sẽ bị mất, k bit phải nhất được điền là 0.

$[x_{n-1}, x_{n-2}, \dots, x_0] \ll k$ trở thành $[x_{n-k-1}, x_{n-k-2}, \dots, x_0, 0, \dots, 0]$

Thao tác dịch bit SHIFT RIGHT >> k

Dịch phải k bit. Các bit phải vượt vị trí bit 0 sẽ bị mất. Đối với logical shift thì k bit trái nhất được điền là 0. Đối với arithmetic shift thì k bit trái nhất được điền giá trị của bit trái nhất ban đầu.

Trong C, khi dịch phải một số unsigned, phép dịch sẽ là logical shift. Nếu dịch một số có dấu, phép dịch sẽ là arithmetic shift.

$[x_{n-1}, x_{n-2}, \dots, x_0] \gg k$ logical shift nhận được $[0, \dots, 0, x_{n-1}, x_{n-2}, \dots, x_k]$

$[x_{n-1}, x_{n-2}, \dots, x_0] \gg k$ arithmetic shift nhận được $[x_{n-1}, \dots, x_{n-1}, x_{n-2}, \dots, x_k]$

Chuyển đổi số nguyên sang độ dài word nhỏ hơn

Một số nguyên độ dài w bit bị cắt ngắn còn k bit tương đương với việc coi chuỗi w bit như một số nguyên không dấu và chia dư cho 2^k

$[x_{w-1}, \dots x_{k-1}, \dots, x_0]$ cắt ngắn còn k bit thành $[x_{k-1}, \dots, x_0]$

Điều này đúng cho cả số nguyên có dấu và không dấu.

Chuyển đổi số nguyên sang độ dài word lớn hơn

Chuyển đổi số nguyên không dấu sang định dạng lớn hơn chỉ đơn giản là đặt các bit mở rộng bằng 0, cụ thể $[x_{w-1}, \dots, x_0]$ => $[0, \dots, 0, x_{w-1}, \dots, x_0]$.

Chuyển đổi số nguyên có dấu sang định dạng lớn hơn đòi hỏi việc sao lặp bit dấu vào tất cả các bit mở rộng do chúng ta dùng mã hóa two's complement, cụ thể $[x_{w-1}, \dots, x_0]$ => $[x_{w-1}, \dots, x_{w-1}, x_{w-1}, \dots, x_0]$.

Giải thích: chúng ta chứng tỏ rằng

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$$

$$
\begin{aligned}
B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\
&= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}\left(2^w - 2^{w-1}\right) + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])
\end{aligned}
$$

Bài tập: Chạy thử 2 chương trình sau với input là 0x00000076 và 0x000000C9

```
// Utcasting.c
int func1(unsigned int word) {
        return (int) ((word << 24) >> 24);
}
int func2(unsigned int word) {
        return ((int) word << 24) >> 24;
}
```

Bài tập: Lý giải vì sao khi gọi sum_element với length=0 lại nhận được lỗi core dump thay vì result=0

```
// sum_elements.c
float sum_elements(float a[], unsigned length) {
        int i;
        float result = 0;

        for (i = 0; i <= length-1; i++)
                result += a[i];
        return result;
}
```

Ví dụ thực tế: đoạn mã từ FreeBSD

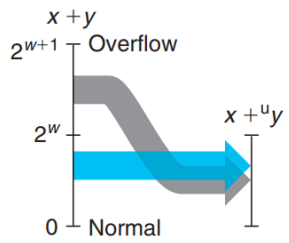
```
1   /*
2    * Illustration of code vulnerability similar to that found in
3    * FreeBSD's implementation of getpeername()
4    */
5
6   /* Declaration of library function memcpy */
7   void *memcpy(void *dest, void *src, size_t n);
8
9   /* Kernel memory region holding user-accessible data */
10  #define KSIZE 1024
11  char kbuf[KSIZE];
12
13  /* Copy at most maxlen bytes from kernel region to user buffer */
14  int copy_from_kernel(void *user_dest, int maxlen) {
15      /* Byte count len is minimum of buffer size and maxlen */
16      int len = KSIZE < maxlen ? KSIZE : maxlen;
17      memcpy(user_dest, kbuf, len);
18      return len;
19  }
```

Tham số maxlen (dòng 14) và len (dòng 16) là kiểu int, có thể bị kẻ xấu lợi dụng để gọi với tham số có giá trị âm. Khi đó len sẽ nhận giá trị âm. Tuy nhiên hàm memcpy có tham số n là kiểu size_t (kiểu unsigned) nên giá trị âm được chuyển thành một giá trị lớn không dấu, cho phép kẻ tấn công copy, lấy được nội dung không được phép trong kernel.

Các phép toán với số nguyên

Phép cộng số nguyên không dấu

Cộng hai số nguyên không dấu x, y độ dài w bit tương đương với phép cộng $s = x + y \ mod \ 2^w$

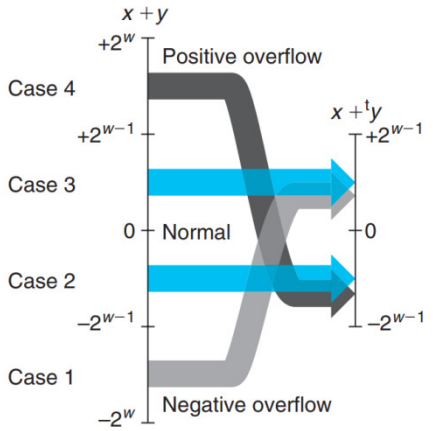


Trường hợp bị tràn bit có thể được nhận biết khi $s < x, y$ do $s = x + y - 2^w < x, y$

Khi $x = 2^w - y$ thì tổng $x + y = 0$

Cộng số nguyên có dấu bù hai

Cộng hai số nguyên có dấu tương đương với việc chuyển hai số đó sang dạng không dấu và cộng chúng với nhau, kết quả cuối cùng được chuyển lại thành số nguyên có dấu.



$$
\begin{aligned}
x +_w^{\mathrm{t}} y &= U2T_w(T2U_w(x) +_w^{\mathrm{u}} T2U_w(y)) \\
&= U2T_w[(x_{w-1}2^w + x + y_{w-1}2^w + y) \bmod 2^w] \\
&= U2T_w[(x + y) \bmod 2^w]
\end{aligned}
$$

Hình trên có thể được cụ thể hóa trong công thức sau

$$
x +_w^{\mathrm{t}} y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y \quad \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} \quad \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} \quad \text{Negative overflow} \end{cases} \tag{2.14}
$$

Ví dụ

x	y	$x + y$	$x +_4^{\mathrm{t}} y$	Case
−8	−5	−13	3	1
[1000]	[1011]	[10011]	[0011]	
−8	−8	−16	0	1
[1000]	[1000]	[10000]	[0000]	
−8	5	−3	−3	2
[1000]	[0101]	[11101]	[1101]	
2	5	7	7	3
[0010]	[0101]	[00111]	[0111]	
5	5	10	−6	4
[0101]	[0101]	[01010]	[1010]	

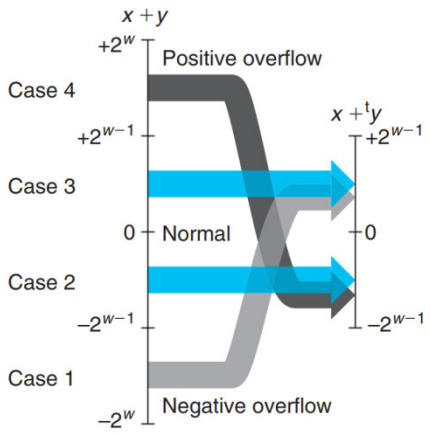


Figure 2.24 **Two's-complement addition examples.** The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

Giá trị đối của số nguyên mã bù hai

Một số nguyên có dấu x dài w bit có giá trị nằm trong khoảng $-2^{w-1} \leq x < 2^{w-1}$.

Nếu $x \neq -2^{w-1}$ thì giá trị đối của x là $-x$. Đối với $x = -2^{w-1}$ thì giá trị $-x = 2^{w-1}$ không thể biểu diễn được bởi w bit, trong trường hợp này thì x chính là giá trị đối của nó, tức là $-x = x$. Lý do là

$$x + (-x) = -2^{w-1} + (-2^{w-1}) = -2^w = 0$$

Một kỹ thuật để tính nhanh giá trị đối viết theo dạng nhị phân của x là đảo các bit của x rồi cộng thêm 1. Hoặc cũng có thể xác định bằng cách coi x như số nguyên không dấu và tính $2^w - x$, rồi chuyển lại thành số nguyên có dấu.

Bài tập: cho x dưới dạng hexa, tính $-x$

x		$-_4^t x$	
Hex	Decimal	Decimal	Hex
0			
5			
8			
D			
F			

Nhân số nguyên không dấu

Phép nhân 2 số nguyên không dấu w bit là phép nhân 2 số đó và lấy $mod\ 2^w$, do chúng ta chỉ có w bit để lưu giá trị.

$$x *_w^u y = (x \cdot y) \bmod 2^w \qquad (2.16)$$

Nhân số nguyên có dấu mã bù hai

Tích của hai số nguyên có dấu là kết quả của việc nhân không dấu, lấy $mod\ 2^w$ và chuyển lại về biểu diễn có dấu. Do vậy có thể dùng cùng một phép tính chung cho cả số nguyên có và không dấu.

$$x *_w^t y = U2T_w((x \cdot y) \bmod 2^w) \qquad (2.17)$$

Để thấy rằng phần chia dư theo có dấu và không dấu cho cùng một kết quả bit, chúng ta ký hiệu x, y là hai giá trị có dấu và x', y' là hai giá trị không dấu có chung biểu diễn bit. Ta có $x' = x + x_{w-1}2^w$ và $y' = y + y_{w-1}2^w$. Khi đó

$$\begin{aligned}(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \qquad (2.18)\\ &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w\\ &= (x \cdot y) \bmod 2^w\end{aligned}$$

Bài tập: điền vào bảng dưới đây

Mode	x		y		$x \cdot y$	Truncated $x \cdot y$
Unsigned	_____	[100]	_____	[101]	_____	_____
Two's comp.	_____	[100]	_____	[101]	_____	_____
Unsigned	_____	[010]	_____	[111]	_____	_____
Two's comp.	_____	[010]	_____	[111]	_____	_____
Unsigned	_____	[110]	_____	[110]	_____	_____
Two's comp.	_____	[110]	_____	[110]	_____	_____

Bài tập: làm sao kiểm tra phép nhân cho kết quả bị tràn bit (overflow)

```
x = a * b;
if (a != 0 && x / a != b) {
    // overflow handling
}
```

Ví dụ thực tiễn: thư viện XDR (eXternal Data Representation) của Sun Microsystems có lỗi tràn bit như dưới đây. Thư viện XDR được dùng rộng rãi trong nhiều phần mềm, chẳng hạn IE, hệ thống xác thực Kerberos vv..

```
1  /*
2   * Illustration of code vulnerability similar to that found in
3   * Sun's XDR library.
4   */
5  void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
6      /*
7       * Allocate buffer for ele_cnt objects, each of ele_size bytes
8       * and copy from locations designated by ele_src
9       */
10     void *result = malloc(ele_cnt * ele_size);
11     if (result == NULL)
12         /* malloc failed */
13         return NULL;
14     void *next = result;
15     int i;
16     for (i = 0; i < ele_cnt; i++) {
17         /* Copy object i to destination */
18         memcpy(next, ele_src[i], ele_size);
19         /* Move pointer to next memory region */
20         next += ele_size;
21     }
22     return result;
23 }
```

Hàm copy_elements chép 1 mảng cấu trúc dữ liệu ele_src gồm ele_cnt thành phần tới 1 bộ đệm được cấp phát bởi malloc(ele_cnt * ele_size). Kẻ tấn công có thể gọi hàm với tham số ele_cnt = 1,048,577 (tức

$2^{20} + 1$) và kích thước ele_size=4096 tức (2^{12}). Khi đó phép nhân gây tràn bit, đối với CPU 32 bit, và kết quả là chỉ có 4096 bytes được cấp phát. Vòng lặp for tiếp theo sẽ gây tràn bộ đệm.

Nhân với hằng số

Phép nhân rất đắt giá, thường tốn tới hơn 10 chu kỳ đồng hồ để thực hiện, so với 1 chu kỳ dành cho phép cộng, trừ. Trong trường hợp nhân với 1 hằng số, chương trình dịch sẽ chuyển phép nhân thành một loạt các phép dịch bit và cộng, do vậy có thể thực hiện nhanh hơn rất nhiều.

Trước hết chúng ta xét phép nhân với bội mũ 2, thực hiện qua phép SHIFT LEFT

$$x.2^k == x \ll k$$

Do phép nhân $x.2^k$ trên w bit là như nhau đối với cả số có và không dấu nên phép dịch trái áp dụng được cho cả hai loại số nguyên.

Đối với một hằng số bất kỳ, có thể biểu diễn thành đa thức gồm tổng các bội mũ 2. Chẳng hạn $14 = 2^3 + 2^2 + 2^1$ và do vậy $x.14 = (x \ll 3) + (x \ll 2) + (x \ll 1)$, hoặc gọn hơn $(x \ll 4) - (x \ll 1)$.

Bài tập: tính x.k

K	Shifts	Add/Subs	Expression
6	2	1	_____
31	1	1	_____
−6	2	1	_____
55	2	2	_____

Chia cho bội mũ 2

Trường hợp số nguyên không dấu

Phép chia số nguyên x cho y nhận được kết quả $\lfloor x/y \rfloor$ tức là phần nguyên của phép chia. Chia cho $y = 2^k$ tương ứng với dịch phải k bit

$$x/2^k = x \gg k$$

Cụ thể

$$\frac{x}{2^k} = (\sum_0^{w-1} x.2^i)/2^k = \sum_k^{w-1} x.2^i = x \gg k$$

Trường hợp số nguyên có dấu

Nếu $x \geq 0$, phép chia cho bội mũ 2 được thực hiện như với số nguyên không dấu.

Còn với $x < 0$ (và $y > 0$) thì kết quả phép chia cần được làm tròn lên về phía 0, tức là $\lceil x/y \rceil$. Tuy nhiên nếu dùng phép dịch phải có dấu (arithmetic right shift) thì lại nhận được giá trị làm tròn xuống. Ví dụ $-5\ div\ 4\ =\ -1$ nhưng nếu dịch phải -5 thì $1011 \gg 2 = 1110 = -2$. Có thể khắc phục vấn đề này bằng cách áp dụng tính chất $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ với $y > 0$. Do vậy phép chia x cho $y = 2^k$ được tính là

$$(x < 0\ ?\ x + (1 \ll k) - 1 :\ x) \gg\ k$$

Tính chất trên được lý giải như sau:

Giả sử $x = ny + r$, với $0 \leq r < y$, ta có $\lfloor (x + y - 1)/y \rfloor = n + \lfloor (r + y - 1)/y \rfloor$. Số hạng thứ 2 của tổng bằng 0 khi $r = 0$ và bằng 1 khi $r > 0$. Do vậy ta có kết quả là n khi y là ước của x và là $n + 1$ nếu khác đi.

Số thập phân dấu phẩy động

Lưu ý: trong phần này mặc dù nói tới "dấu phẩy" phân tách phần thập phân nhưng khi biểu diễn trong ngôn ngữ lập trình sẽ là dấu chấm.

Do giới hạn kích thước bit của một từ máy tính, chúng ta không thể biểu diễn chính xác giá trị của một số thực tùy ý. Cách biểu diễn thường được áp dụng là định dạng số thực với dấu phẩy động, có dạng

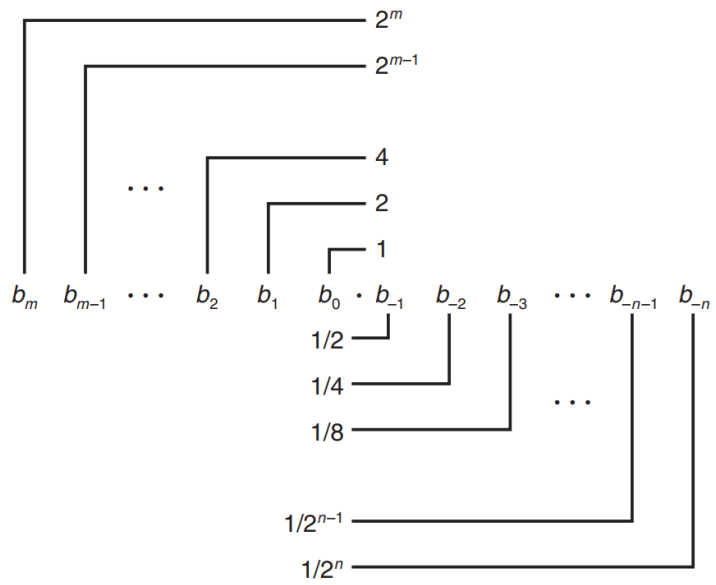
$$V = M \times R^E$$

Trong đó M (Mantissa) là phần định trị, R (Radix) là cơ số, E (Exponent) là số mũ. Phần định trị được chuẩn hóa là phần định trị có một chữ số khác không trước dấu phẩy.

Ví dụ: $256{,}5$ được biểu diễn dưới dạng chuẩn hóa thành $2{,}565 \times 10^2$

Cách biểu diễn dưới dạng nhị phân:

Cho chuỗi bit với phần nguyên $b_m \dots b_0$ và phần lẻ sau dấu phẩy $b_{-1} \dots b_{-n}$ như trong hình



Chúng ta có giá trị của chuỗi bit được tính theo công thức $\sum_{-n}^{m} b_i . 2^i$. Cũng ví dụ trên khi biểu diễn theo cơ số 2 chúng ta có $256{,}5_D = 1\ 0000\ 0000{,}1_B$ và biểu diễn theo dạng dấu phẩy động chuẩn hóa là $1{,}0000\ 0000\ 1\ \times 2^8$, trong đó $R = 2,\ M = 1{,}0000\ 0000\ 1$ và $E = 8$.

Nhận xét: phần thập phân được biểu diễn theo tổng các phân số dạng $1/2^i$ nên sẽ có những giá trị mà chúng ta không thể biểu diễn chính xác được. Tuy nhiên đây là điều bình thường bởi vì chúng ta chỉ có kích thước bit của một từ máy tính để biểu diễn.

Bài tập: số thập phân 0.1 được biểu diễn dưới dạng chuỗi bit nhị phân như thế nào?

Trả lời: $0.1 = 1/10$. Làm phép chia ở hệ nhị phân $1_B : 1010_B = 0.0(0011)^*$ mẫu 0011 được lặp lại vô tận.

Với kiểu half precision (16 bit) thì giá trị biểu diễn nhỏ hơn 1 một chút do phần bit còn lại bị cắt bỏ

$$0.1_D \approx 1.100110011 \times 2^{-4} = 0.0001100110011_B = 0.0999755859375_D$$

Còn với kiểu single precision (32 bit) khi đó giá trị biểu diễn lớn hơn 1 một chút do phần bit còn lại được làm tròn lên (...110**011** => ...110**1**)

$$0.1_D \approx 1.10011001100110\ 1 \times 2^{-4} = 0.0001100110011001101_B$$
$$= 0.100000001490116119384765625_D$$

Định dạng dấu phẩy động của IEEE:

Một số thực dấu phẩy động được biểu diễn dưới dạng

$$V = (-1)^s \times M \times R^E$$

Trong đó

- s là bit dấu
- Phần định trị M có giá trị thuộc khoảng $1\,..2-\epsilon$ hoặc $0..1-\epsilon$ tùy theo trường hợp biểu diễn
- E (exponent) là số mũ của cơ số 2
- R (radix) bằng 2

Dưới đây là biểu diễn bit cho số thực dấu phẩy động với các trường đã mô tả

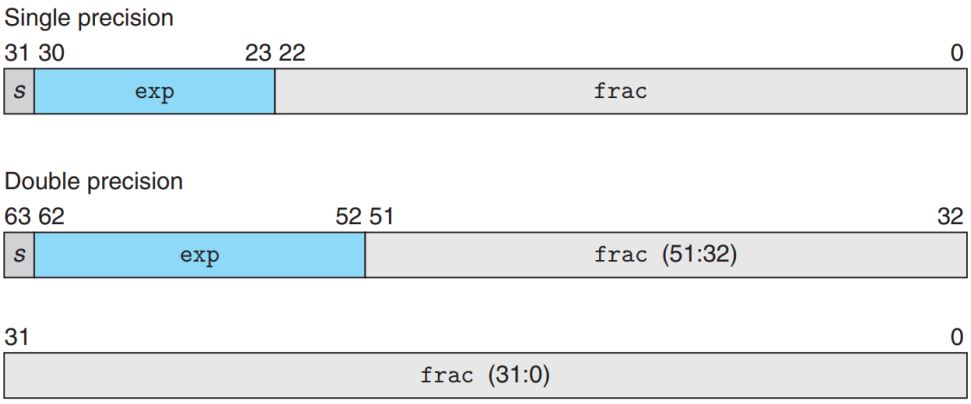


Figure 2.31 **Standard floating-point formats.** Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single precision) or 64-bit (double precision) words.

IEEE định nghĩa 3 trường hợp biểu diễn số thực dấu phẩy động

Trường hợp 1: Định dạng chuẩn hóa

Là trường hợp khi các bit của E khác 0 hoặc khác toàn 1. Đây là trường hợp phổ biến nhất, biểu diễn cho các số thực trên trục số. Định dạng chuẩn hóa đặt giá trị $M \geq 1$ (do quy ước chữ số trước dấu phẩy phải khác không), và do vậy không biểu diễn được giá trị 0.

Không có bit dấu nào cho số mũ, do vậy để biểu diễn cho số mũ âm chúng ta trừ đi 1 giá trị Bias cố định. Cụ thể với k bit biểu diễn số mũ e thì giá trị $E = e - Bias$, trong đó $Bias = 2^{k-1} - 1$. Chẳng hạn đối với số thực single precision $k = 8$ bits cho số mũ e thì $Bias = 127$.

Phần định trị được xác định bởi $M = 1 + f$ trong đó f (fraction) là phần thập phân biểu diễn bởi n bit $0 \leq f < 1$, cụ thể biểu diễn bit của f là $f = 0, f_{n-1} \ldots f_0$. Như vậy $1 \leq M < 2$.

$$V = (-1)^s \times (1 + f) \times 2^{e-Bias}$$

Trường hợp 2: Định dạng phi chuẩn hóa

Định dạng phi chuẩn hóa được dùng để biểu diễn số 0 và các số rất sát 0. Trong trường hợp này *các bit của e bằng 0* và khi đó chúng ta tính $E = 1 - Bias$, và $M = f$ và do vậy $0 \leq M < 1$.

$$V = (-1)^s \times f \times 2^{1-Bias}$$

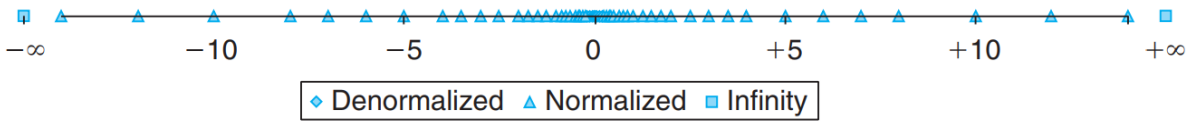
Để biểu diễn số 0 thì chúng ta đặt $f = 0$. Thú vị là bit dấu s có thể là 0 hoặc 1, cho phép chúng ta biểu diễn hai số -0.0 và $+0.0$ riêng biệt.

Để biểu diễn các số thập phân sát 0, chúng ta dùng phần thập phân f và $E = 1 - Bias$. Lưu ý rằng với biểu diễn $E = e - Bias$ của định dạng chuẩn và $E = 1 - Bias$ của định dạng phi chuẩn chúng ta có được sự chuyển tiếp khá mịn về giá trị giữa hai cách biểu diễn.

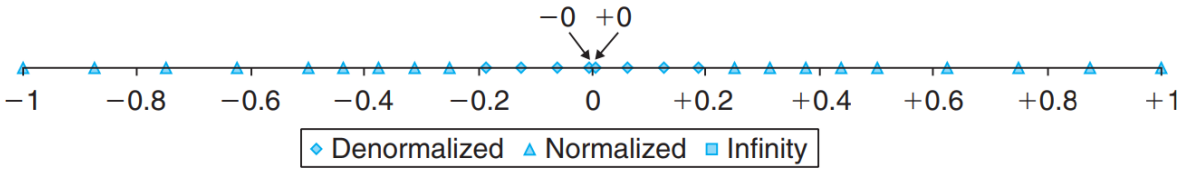
Trường hợp 3: Overflow

Khi nhân các số quá lớn, tình trạng tràn sẽ xảy ra và chúng ta coi kết quả là giá trị vô tận, dương hoặc âm tùy vào bit dấu s. Trong trường hợp này, tất cả *các bit của e được đặt bằng 1*, phần thập phân f được đặt bằng 0. Khi phần thập phân f khác 0 thì kết quả được coi là không phải 1 số (NaN – Not a Number), chẳng hạn khi tính kết quả của $\sqrt{-1}$ hoặc $\infty - \infty$.

Ví dụ: Khoảng biểu diễn các giá trị thực của các trường hợp khác nhau



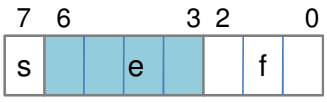
(a) Complete range



(b) Values between −1.0 and +1.0

Figure 2.33 **Representable values for 6-bit floating-point format.** There are $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is 3.

Ví dụ: Số thực dấu phẩy động biểu diễn bởi 8 bit, theo định dạng 1 bit dấu s, k=4 bit cho số mũ e, n=3 bit cho phần thập phân f. Giá trị bias được tính bằng $2^3 - 1 = 7$



Description	Bit representation	Exponent			Fraction		Value		
		e	E	2^E	f	M	$2^E \times M$	V	Decimal
Zero	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest pos.	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denorm.	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest norm.	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest norm.	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

Figure 2.34 **Example nonnegative values for 8-bit floating-point format.** There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

Dải đầu tiên là 0 và các số sát 0 được biểu diễn theo trường hợp phi chuẩn.

Dải tiếp theo là các số biểu diễn theo trường hợp chuẩn.

Cuối cùng là số thập phân có giá trị quá lớn so với kích thước bit có thể biểu diễn, được coi là vô tận.

Đối với mỗi số biểu diễn dưới dạng nhị phân, bit đầu là dấu s, 4 bit tiếp theo là số mũ e, 3 bit cuối là phần thập phân f.

Bài tập: Với độ dài 5 bit, 1 bit dấu, 2 bit mũ, 2 bit thập phân. Hãy điền vào bảng sau

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00								
0 00 01								
0 00 10								
0 00 11								
0 01 00								
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10								
0 01 11								
0 10 00								
0 10 01								
0 10 10								
0 10 11								
0 11 00	—	—	—	—	—	—		—
0 11 01	—	—	—	—	—	—		—
0 11 10	—	—	—	—	—	—		—
0 11 11	—	—	—	—	—	—		—