

第二章 数据的机器级表示与处理

数值数据的表示

非数值数据的表示

数据的存储

数据的运算

数据的表示和运算

- 主要教学目标

- 掌握计算机内部各种数据的编码表示及其运算方法
- 了解高级语言程序中的各种类型变量对应的表示形式
- 在高级语言程序中的**变量、机器数和底层硬件**（寄存器、加法器、ALU等）**之间建立关联**
- 综合运用所学知识，分析高级语言和机器级语言程序设计中遇到的各种与数据表示和运算相关的问题，解释相应的执行结果

C语言参考网站: <http://docs.huihoo.com/c/linux-c-programming/>

数据的表示和运算

- 分以下三个部分介绍

- **第一讲：数值数据的表示**

- 定点数的编码表示、整数的表示、无符号整数、带符号整数、浮点数的表示
 - C语言程序的整数类型和浮点数类型

- **第二讲：非数值数据的表示、数据的存储**

- 逻辑值、西文字符、汉字字符
 - 数据宽度单位、大端/小端、对齐存放

- **第三讲：数据的运算**

- 按位运算\逻辑运算\移位运算
 - 位扩展和位截断运算
 - 无符号和带符号整数的加减运算
 - 无符号和带符号整数的乘除运算
 - 变量与常数之间的乘除运算
 - 浮点数的加减乘除运算

围绕C语言中的运算，解释其在底层机器级的实现

从C程序的表达式出发，用机器数在电路中的执行来解释表达式的执行结果

课程内容概要

```
/*---sum.c---*/
```

```
int sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

```
/*---main.c---*/
```

```
int main()
{
    int a[1]={100};
    int sum;
    sum=sum(a,0);
    printf("%d",sum);
}
```

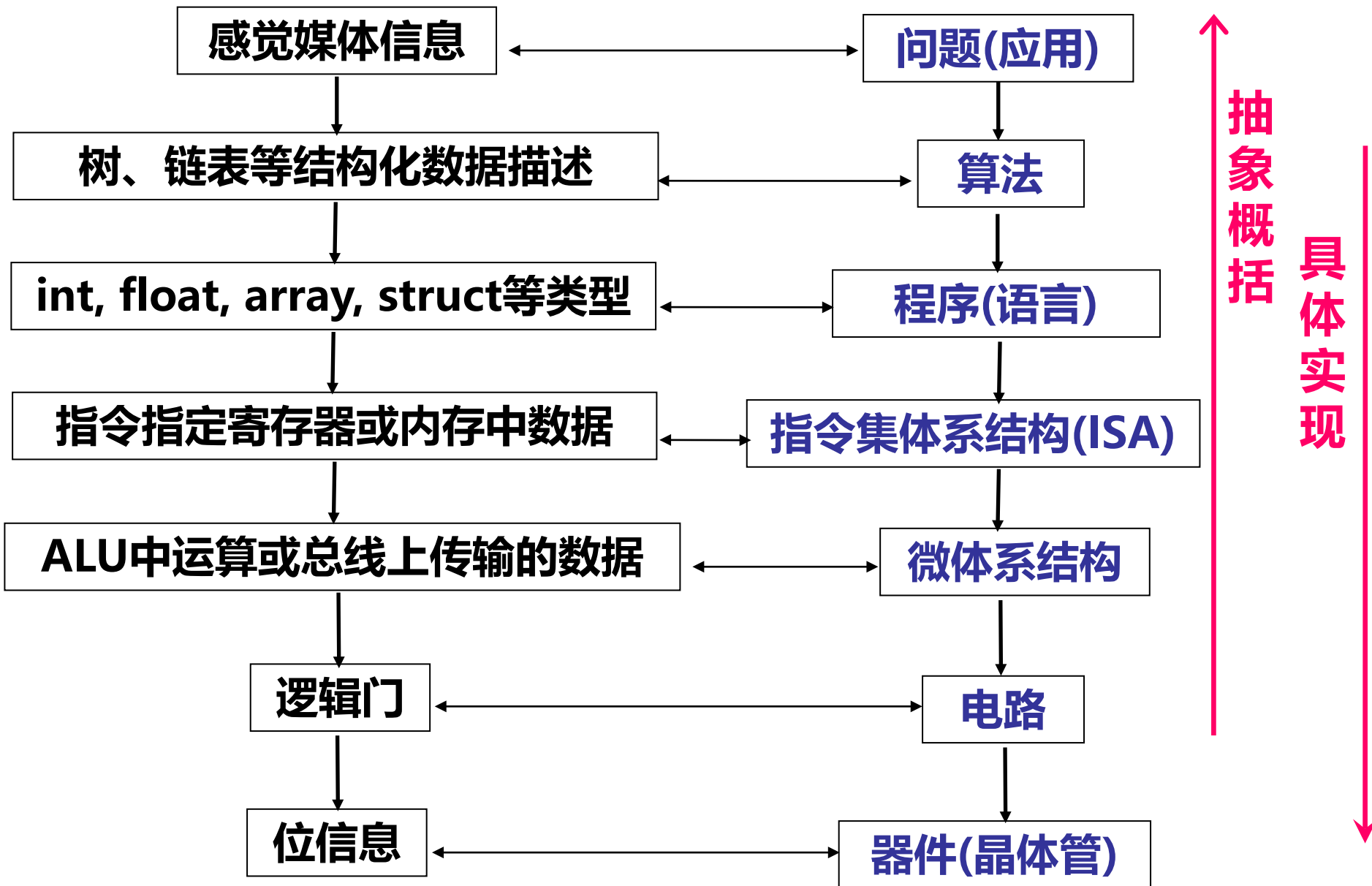
数据的表示

数据的运算

如果程序处理的是图像、视频、声音、文字等数据，那么，

- (1) 如何获得这些数据？
- (2) 如何表示这些数据？
- (3) 如何处理这些数据？

“转换”的概念在数据表示中的反映



对连续信息采样,
以使信息离散化

对离散样本用0和1
进行编码

文字、图、表、声音、
视频等各种媒体信息

最终用户角度

输入设备

输出设备

各类数据之间的
转换关系

二进制编码表示的各种数据

数组、结构、字符串等结构化数据

高级语言程序员角度

指令系统能识别
的基本类型数据

低级语言程序员和
硬件系统设计者角度

数值型数据

非数值型数据

定点运算指令

二进制数

二进制编码的
十进制数

逻辑数据

编码字符
如:西文字符和汉字

整数(定点数)

实数(浮点数)

逻辑、位操作或字符处理指令

浮点运算指令

无符号整数

带符号整数

数值数据的表示

- 数值数据表示的三要素

- 进位记数制

- 定、浮点表示

- 如何用二进制编码

即：要确定一个数值数据的值必须先确定这三个要素。

例如，机器数 01011001 的值是多少？ 答案是：不知道！

- 进位记数制

- 十进制、二进制、十六进制、八进制数及其相互转换

- 定/浮点表示 (解决小数点问题)

- 定点整数、定点小数

- 浮点数 (可用一个定点小数和一个定点整数来表示)

- 定点数的编码 (解决正负号问题)

- 原码、补码、反码、移码 (反码很少用)

Unsigned integer(无符号整数)

- 机器中字的位排列顺序有两种方式：（例：32位字： $0\dots01011_2$ ）
 - 高到低位从左到右： $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011$ ← **LSB**
 - 高到低位从右到左： $1101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$ ← **MSB**
 - Leftmost和rightmost这两个词有歧义，故用**LSB(Least Significant Bit)**来表示最低有效位，用**MSB**来表示最高有效位
 - 高位到低位多采用从左往右排列
- 一般在全部是正数运算且不出现负值结果的情况下，可使用无符号数表示。例如，地址运算，编号表示，等等
- 无符号整数的编码中**没有符号位**

\vec{x} : 位向量表示 (w 位机器数) $[x_{w-1}, x_{w-2}, \dots, x_0]$

$$\text{真值: } B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

Signed integer（带符号整数，定点整数）

- 计算机必须能处理正数(positive) 和负数(negative), MSB表示数符
- 有三种定点编码方式
 - Signed magnitude（原码）
现用来表示浮点（实）数的尾数
 - One's complement（反码）
现已不用于表示数值数据
 - Two's complement（补码）
50年代以来，所有计算机都用补码来表示定点整数
- 为什么用补码表示带符号整数？
 - 补码运算系统是模运算系统，加、减运算统一
 - 数0的表示唯一，方便使用
 - 比原码和反码多表示一个最小负数

Sign and Magnitude（原码的表示）

Decimal	Binary	Decimal	Binary
0	0000	-0	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

◆ 容易理解， 但是：

- ✓ 0 的表示不唯一，故不利于程序员编程
- ✓ 加、减运算方式不统一
- ✓ 需额外对符号位进行处理，故不利于硬件设计
- ✓ 特别当 $a < b$ 时，实现 $a - b$ 比较困难

从 50年代开始，整数都采用补码来表示
但浮点数的尾数用原码定点小数表示

补码 - 模运算 (modular 运算)

重要概念：在一个模运算系统中，一个数与它除以“模”后的余数等价。

时钟是一种模12系统 现实世界中的模运算系统

假定钟表时针指向10点，要将它拨向6点， 则有两种拨法：

① 倒拨4格： $10 - 4 = 6$

② 顺拨8格： $10 + 8 = 18 \equiv 6 \pmod{12}$

模12系统中： $10 - 4 \equiv 10 + 8 \pmod{12}$

$$-4 \equiv 8 \pmod{12}$$

则，称8是-4对模12的补码（即：-4的模12补码等于8）。

$$\text{同样有 } -3 \equiv 9 \pmod{12}$$

$$-5 \equiv 7 \pmod{12} \text{ 等}$$

结论1：一个负数的补码等于模减该负数的绝对值。

结论2：对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一数负数的补码来代替。

补码 (modular 运算)：+ 和 - 的统一

计算机中的运算器是模运算系统

运算器只有有限位，假设为 n 位，则运算结果只能保留低 n 位，故可看成是个只有 n 档的二进制算盘，因此，其模为 2^n 。

假定一个正数 X 的补码表示为 n 位二进制数字 $a_{n-1}\dots a_1a_0$

则负数 $-X$ 的补码表示为： $2^n - X = 1_n 0_{n-1}\dots 00 - a_{n-1}\dots a_1a_0$
——即各位取反后再加1

结论： 一个负数的补码等于对应正数补码的 “各位取反、末位加1”

$$123 = 127 - 4 = 01111111B - 100B = 01111011B$$

$$[-123]_{\text{补}} = [-01111011]_{\text{补}} = 2^8 - 01111011$$

$$= 10000\ 0000 - 01111011$$

$$= 1111\ 1111 - 0111\ 1011 + 1 = 1000\ 0101$$

$[x]_{\text{补}}$ 代表真值 x 的机器数

真值和机器数的含义是什么？

补码表示的带符号整数

\vec{x} : 位向量表示 (w 位机器数) $[x_{w-1}, x_{w-2}, \dots, x_0]$

真值: $B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$



$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

$$B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$$

特殊数的补码

假定机器数有n位

$$\textcircled{1} [-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = 10\dots0 \text{ (n-1个0)} \quad (\text{mod } 2^n)$$

$$\textcircled{2} [-1]_{\text{补}} = 2^n - 0\dots01 = 11\dots1 \text{ (n个1)} \quad (\text{mod } 2^n)$$

$$\textcircled{3} [+0]_{\text{补}} = [-0]_{\text{补}} = 00\dots0 \text{ (n个0)}$$

32位机器中，int、short、char型数据的机器数各占几位？

C语言程序中的整数类型

无符号数: unsigned int (short / long)

带符号整数: int (short / long)

C语言标准规定了各类型最小取值范围，如: int型至少应为16位，取值范围为-32 768到32 767，而int型数据具体的取值范围则由ABI规范规定。

C语言中整数常量的类型

A plain decimal constant will be fitted into the first in the list that can hold the value:

int unsigned int (C90) long (long) unsigned long (long) (C90)

Plain octal or hexadecimal constants will use the list:

int unsigned int long (long) unsigned long (long)

There are *no* negative constants; writing *-23* is an expression involving a *positive constant* and an *operator*.

在一个整数常量的后面加 “u” 或 “U” 表示一个无符号数

2147483647

int

2147483648

unsigned int (C90) / long (C99)

-2147483647

int

-2147483648

unsigned int (C90) / long (C99)

2147483647U

unsigned int

编译器处理常量时默认的类型

- C90



范围	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{32}-1$	unsigned int
$2^{32} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long

$2^{31} = 2147483648$, 机器数为: 100 ... 0 (31个0)

- C99



范围	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long

C语言程序中的整数

例如，考虑以下C代码：

```
1 int x = -1;
2 unsigned u = 2147483648;
3
4 printf ( "x = %u = %d\n" , x, x);
5 printf ( "u = %u = %d\n" , u, u);
```

在32位机器上运行上述代码时，它的输出结果是什么？为什么？

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

- ◆ 因为-1的补码整数表示为“11...1”，作为32位无符号数解释时，其值为 $2^{32}-1 = 4\ 294\ 967\ 296-1 = 4\ 294\ 967\ 295$ 。
- ◆ 2^{31} 的无符号数表示为“100...0”，被解释为32位带符号整数时，其值为最小负数： $-2^{32-1} = -2^{31} = -2\ 147\ 483\ 648$ 。

C语言整数运算中的类型转换

- 三种常见转换情况：
 - 表达式中操作数类型不一致而引起的转换（**Usual Arithmetic Conversions**）
 - 例： $20 * 1.5 - 199 > -100U$
 - 赋值操作引发的类型转换（**Conversion During Assignment**）
 - 例： `unsigned int x; x = -1;`
 - 强制类型转换（**Type Casting**）
 - 例： `(unsigned int)199`

Usual Arithmetic Conversions

If the operands in an expression have different types, there have to be conversions

- to convert the operand of the narrower type to the type of the other operand, which is the "narrowest" type that will safely accommodate both values – *promotion*

1.If either operand is a **long double**, then the other one is converted to **long double** and also the type of result.

2.Otherwise, **double** ...

3.Otherwise, **float** ...

4.Otherwise, *integral promotions* are first applied, then:

① If either operand is an **unsigned long int** ...

② Otherwise, **long int** ...

③ Otherwise, **unsigned int** ...

④ Finally, both operands must be of type **int**, so that is the result.

C语言程序中的整数

- 1) 在有些32位系统上, C表达式 $-2147483648 < 2147483647$ 的执行结果为false。Why?
- 2) 若定义变量 `int i=-2147483648;` , 则 `i < 2147483647` 的执行结果为true。Why?
- 3) 如果将表达式写成 `-2147483647-1 < 2147483647` , 则结果会怎样呢? Why?

1) 在ISO C90标准下 , 2147483648为unsigned int型, 因此 $-2147483648 < 2147483647$ 按无符号数比较, 10.....0B比01.....1大, 结果为false。

在ISO C99标准下 , 2147483648为long long型, 因此 $-2147483648 < 2147483647$ 按带符号整数比较, 10.....0B比01.....1小, 结果为true。

2) `i < 2147483647` 按int型数比较, 结果为true。

3) `-2147483647-1 < 2147483647` 按int型比较, 结果为true。

C语言程序中的整数

关系 表达式	类型	结果	说明
<code>0 == 0U</code>	无	1	<code>00...0B = 00...0B</code>
<code>-1 < 0</code>	带	1	<code>11...1B (-1) < 00...0B (0)</code>
<code>-1 < 0U</code>	无	0*	<code>11...1B ($2^{32}-1$) > 00...0B(0)</code>
<code>2147483647 > -2147483647 - 1</code>	带	1	<code>011...1B ($2^{31}-1$) > 100...0B (-2^{31})</code>
<code>2147483647U > -2147483647 - 1</code>	无	0*	<code>011...1B ($2^{31}-1$) < 100...0B(2^{31})</code>
<code>2147483647 > (int) 2147483648U</code>	带	1*	<code>011...1B ($2^{31}-1$) > 100...0B (-2^{31})</code>
<code>-1 > -2</code>	带	1	<code>11...1B (-1) > 11...10B (-2)</code>
<code>(unsigned) -1 > -2</code>	无	1	<code>11...1B ($2^{32}-1$) > 11...10B ($2^{32}-2$)</code>

带*的结果与常规预想的相反！

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int x=-1;
```

```
    unsigned u=2147483648;
```

```
    printf("x = %u = %d\n", x, x);
```

```
    printf("u = %u = %d\n", u, u);
```

```
    if(-2147483648 < 2147483647)
```

```
        printf("-2147483648 < 2147483647 is true\n");
```

```
    else
```

```
        printf("-2147483648 < 2147483647 is false\n");
```

```
    if(-2147483648-1 < 2147483647)
```

```
        printf("-2147483648-1 < 2147483647\n");
```

```
    else if(-2147483648-1 == 2147483647)
```

```
        printf("-2147483648-1 == 2147483647\n");
```

```
    else
```

```
        printf("-2147483648-1 > 2147483647\n");
```

```
}
```

C99的结果大家回去试试。

C90上的运行结果是什么？

```
x = 4294967295 = -1
```

```
u = 2147483648 = -2147483648
```

```
-2147483648 < 2147483647 is false
```

```
-2147483648-1 == 2147483647
```

Conversion During Assignment

- For assignment, C follows the simple rule:
 - **The expression on the right side of the assignment is converted to the type of the variable on the left side.**

下述代码有无问题？

```
int j = 80000;  
long long int i;  
i = j * j;
```

Overflow !

可改为 `i = (long long int) j * j ;`

改成 “`i = (int) (j * j) ;`” 又如何？

The **cast operator** takes precedence over *

- The first j is converted to int, forcing the second j to be converted as well.

Type Casting in C

`float` quotient;

`int` dividend=30, divisor=50;

quotient = dividend / divisor;

- quotient = ?

The result of the division - an integer - will be converted to float before being stored in quotient.

quotient = (float) dividend / divisor;

- quotient = ?

The dividend and divisor are converted to float before the division.

C regards type cast (`type-name`) as a unary operator, which has higher precedence than binary operator.

科学计数法(Scientific Notation)与浮点数

Example:

mantissa (尾数) \rightarrow **6.02** \times **10**²¹ \leftarrow *exponent* (阶码、指数)

\nwarrow *decimal point* \nearrow *radix* (base, 基)

- **Normalized form** (规格化形式): 小数点前只有一位非0数
- 同一个数有多种表示形式。例: 对于数 1/1,000,000,000
 - Normalized (唯一的规格化形式): 1.0×10^{-9}
 - Unnormalized (非规格化形式不唯一): 0.1×10^{-8} , 10.0×10^{-10}

for Binary Numbers:

mantissa (尾数) \rightarrow **0.101**_{two} \times **2**⁻¹⁰ \leftarrow *exponent* (指数)

\nwarrow *binary point* \nearrow 基为2

只要对尾数和指数分别编码, 就可表示一个浮点数 (即: 实数)

浮点数(Floating Point)的表示范围

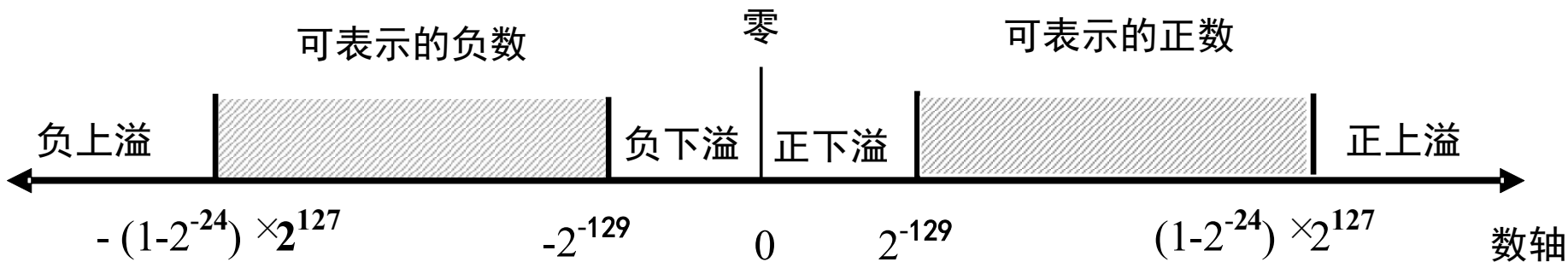
例：画出下述32位浮点数格式的规格化数的表示范围。



第0位数符S；第1~8位为8位移码表示阶码E（偏置常数为128）；第9~31位为24位二进制原码小数表示的尾数数值部分M。规格化尾数的小数点后第一位总是1，故规定第一位默认的“1”不明显表示出来。这样可用23个数位表示24位尾数。

最大正数: $0.11\dots1 \times 2^{11\dots1} = (1-2^{-24}) \times 2^{127}$ 最小正数: $0.10\dots0 \times 2^{00\dots0} = (1/2) \times 2^{-128}$

因为原码是对称的，所以其表示范围关于原点对称。



机器0：尾数为0 或 落在下溢区中的数

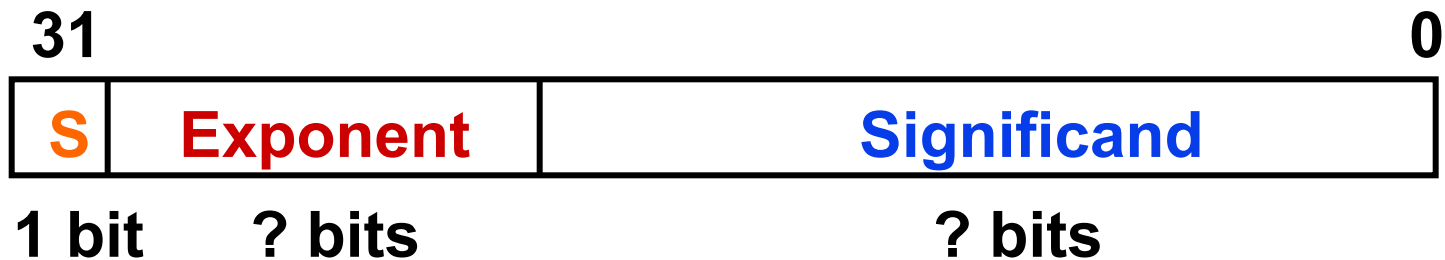
浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

浮点数的表示

- Normal format（规格化数形式）：

+/-1.xxxxxxxxxxx × **R^{Exponent}**

- 32-bit 规格化数：



S 是符号位（Sign）

Exponent 用移码（增码）来表示

Significand 表示 xxxxxxxxxxxxxx，尾数部分

（基可以是 2 / 4 / 8 / 16，约定信息，无需显式表示）

- 早期的计算机，各自定义自己的浮点数格式

问题：浮点数表示不统一会带来什么问题？

“Father” of the IEEE 754 standard

直到80年代初，各个机器内部的浮点数表示格式还没有统一
因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

This standard was primarily the work of one person, UC Berkeley math professor William Kahan.

**1989
ACM Turing
Award Winner!**

[www.cs.berkeley.edu/~wkahan/
ieee754status/754story.html](http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html)



Prof. William Kahan

IEEE 754标准

规格化数: $+/-1.\text{xxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$

规定: 小数点前总是“1”, 故可隐含表示。
注意: 和前面例子规定不一样!

Single Precision :

S	Exponent	Significand
1 bit	8 bits	23 bits

- Sign bit: 1 表示negative ; 0表示 positive
- Exponent (阶码 / 指数编码) : 全0和全1用来表示特殊值!
 - SP规格化数阶码范围为0000 0001 (-126) ~ 1111 1110 (127)
 - bias为127 (single), 1023 (double)
- Significand (尾数) :
 - 规格化尾数最高位总是1, 所以隐含表示, 省1位
 - 1 + 23 bits (single) , 1 + 52 bits (double)

为什么用127? 若用128, 则阶码范围为多少?

SP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

DP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-1023)}$

0000 0001 (-127) ~
1111 1110 (126)

Ex: Converting Binary FP to Decimal

BEE00000H is the hex. Rep. Of an IEEE 754 SP FP number

1011 11101 110 0000 0000 0000 0000 0000

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- **Sign:** 1 => negative
- **Exponent:**
 - $0111\ 1101_{\text{two}} = 125_{\text{ten}}$
 - Bias adjustment: $125 - 127 = -2$
- **Significand:**
$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$
- **Represents:** $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$

Ex: Converting Decimal to FP

-12.75

1. Denormalize: -12.75

2. Convert integer part:

$$12 = 8 + 4 = 1100_2$$

3. Convert fractional part:

$$.75 = .5 + .25 = .11_2$$

4. Put parts together and normalize:

$$1100.11 = 1.10011 \times 2^3$$

5. Convert exponent: $127 + 3 = 128 + 2 = 1000\ 0010_2$

11000	0010	100	1100	0000	0000	0000	0000
-------	------	-----	------	------	------	------	------

The Hex rep. is **C14C0000H**

Normalized numbers (规格化数)

前面的定义都是针对规格化数 (normalized form)

How about other patterns?

Exponent	Significand	Object
1-254	anything implicit leading 1	Norms
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

Representation for 0

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

Representation for $+\infty/-\infty$

In FP, 除数为0的结果是 $\pm\infty$, 不是溢出异常. (整数除0为异常)

为什么要这样处理?

∞ : infinity

- 可以利用 $+\infty/-\infty$ 作比较。 例如: $X/0 > Y$ 可作为有效比较

How to represent $+\infty/-\infty$?

- Exponent** : all ones (11111111B = 255)
- Significand**: all zeros

$+\infty$: 0 11111111 00000000000000000000000000000000

$-\infty$: 1 11111111 00000000000000000000000000000000

Operations

$$5.0 / 0 = +\infty, \quad -5.0 / 0 = -\infty$$

$$5 + (+\infty) = +\infty, \quad (+\infty) + (+\infty) = +\infty$$

$$5 - (+\infty) = -\infty, \quad (-\infty) - (+\infty) = -\infty \quad \text{etc}$$

Representation for “Not a Number”

$\text{Sqrt}(-4.0) = ?$ $0/0 = ?$

- Called **Not a Number (NaN)** - “非数”

How to represent NaN

Exponent = 255

Significand: nonzero

NaNs can help with debugging

Operations

$\text{sqrt}(-4.0) = \text{NaN}$

$\text{op}(\text{NaN}, x) = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$

etc.

$0/0 = \text{NaN}$


$+\infty + (-\infty) = \text{NaN}$

$\infty/\infty = \text{NaN}$

Representation for Denorms(非规格化数)

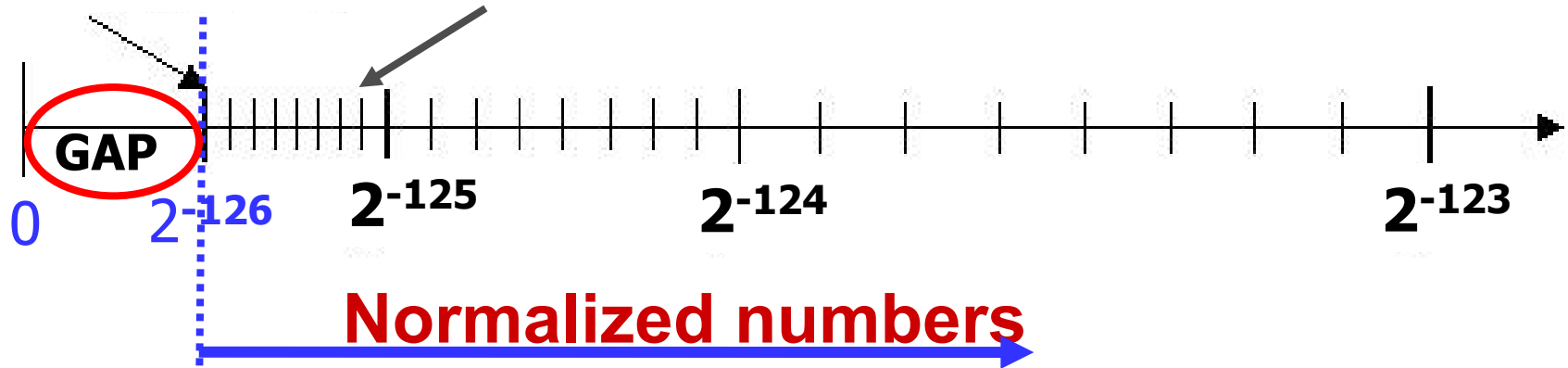
What have we defined so far? (for SP)

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

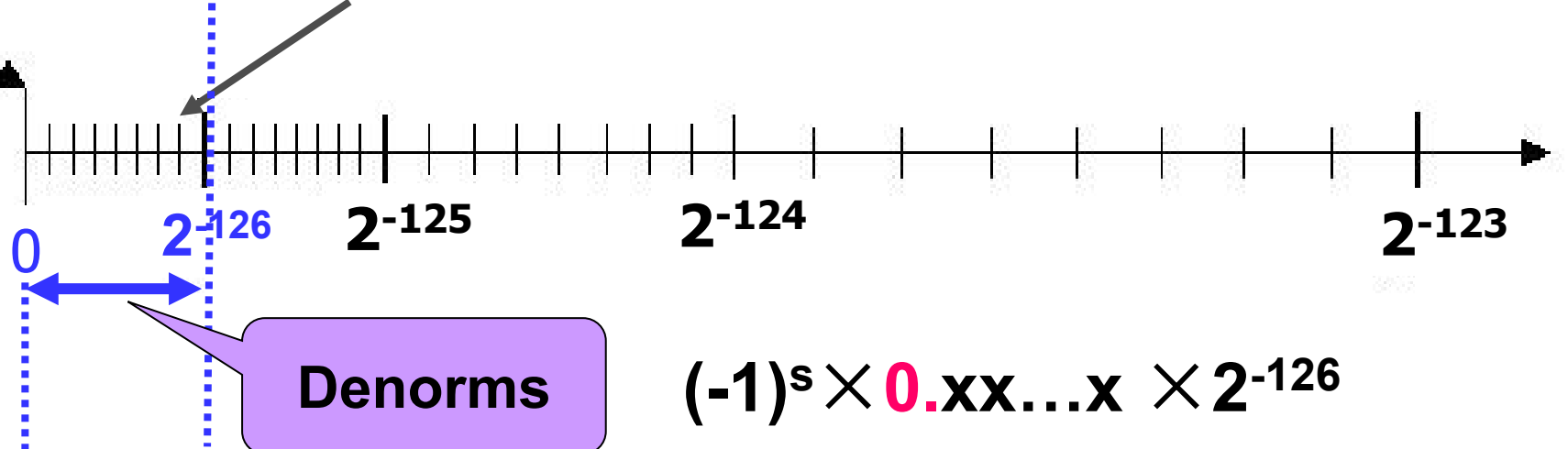


Representation for Denorms

$1.0...0 \times 2^{-126} \sim 1.1...1 \times 2^{-126}$



$0.0...0 \times 2^{-126} \sim 0.1...1 \times 2^{-126}$



$$(-1)^s \times 0.\text{xx}\dots\text{x} \times 2^{-126}$$

关于浮点数精度的一个例子

```
#include <iostream>
using namespace std;
int main()
{
    float heads;
    cout.setf(ios::fixed,ios::floatfield);
    while(1)
    {
        cout << "Please enter a number: ";
        cin>> heads;
    }
```

运行结果:

```
Please enter a number: 61.419997
61.419998
Please enter a number: 61.419998
61.419998
Please enter a number: 61.419999
61.419998
Please enter a number: 61.42
61.419998
Please enter a number: 61.420001
61.420002
Please enter a number:
```

61.419998和61.420002是两个可表示数，两者之间相差0.000004。当输入数据是一个不可表示数时，机器将其转换为最邻近的可表示数。

第一讲小结

10在计算机中有几种可能的表示？
-10呢？

- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进制、定点/浮点、编码
- 整数的表示
 - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- C语言中的整数
 - 无符号数：unsigned int (short / long)；带符号数：int (short / long)
- 浮点数的表示
 - 符号；尾数：定点小数；指数（阶）：定点整数（基不用表示）
- 浮点数的范围
 - 正上溢、正下溢、负上溢、负下溢；与阶码的位数和基的大小有关
- 浮点数的精度：与尾数的位数和是否规格化有关
- 浮点数的表示（IEEE 754标准）：单精度SP（float）和双精度DP（double）
 - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
 - “零”（阶为全0，尾为全0）
 - ∞ （阶为全1，尾为全0）
 - NaN（阶为全1，尾为非0）
 - 非规格化数（阶为全0，尾为非0，隐藏位为0）
- 十进制数的表示：用ASCII码或BCD码表示

数据的表示和运算

- 分以下三个部分介绍
 - 第一讲：数值数据的表示
 - 定点数的编码表示
 - 整数的表示
 - 无符号整数、带符号整数
 - 浮点数的表示
 - C语言程序的整数类型和浮点数类型
 - 第二讲：非数值数据的表示、数据的存储
 - 逻辑值、西文字符、汉字字符
 - 数据宽度单位
 - 大端/小端、对齐存放

逻辑数据的编码表示

- 表示

- 用一位表示。例如，真：1 / 假：0
- N位二进制数可表示N个逻辑数据，或一个位串

- 运算

- 按位进行
- 如：按位与 / 按位或 / 逻辑左移 / 逻辑右移 等

- 识别

- 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器靠指令来识别。

- 位串

- 用来表示若干个状态位或控制位（OS中使用较多）

例如，x86的标志寄存器含义如下：

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	--	--	--	----	----	----	----	----	----	--	----	--	----	--	----

西文字符的编码表示

- 特点

- 是一种拼音文字，用有限几个字母可拼写出所有单词
- 只对有限个字母和数学符号、标点符号等辅助字符编码
- 所有字符总数不超过256个，使用7或8个二进位可表示

- 表示（常用编码为7位ASCII码）

- 十进制数字：0/1/2.../9
- 英文字母：A/B/.../Z/a/b/.../z
- 专用符号：+/-/%/*/&/.....
- 控制字符（不可打印或显示）

} 必须熟悉对应的ASCII码！

- 操作

- 字符串操作，如：传送/比较 等

汉字及国际字符的编码表示

- 特点

- 汉字是表意文字，一个字就是一个方块图形。
- 汉字数量巨大，总数超过6万字，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。

- 编码形式

- 有以下几种汉字代码：
 - 输入码：对汉字用相应按键进行编码表示，用于输入
 - 内码：用于在系统中进行存储、查找、传送等处理
 - 字模点阵或轮廓描述：描述汉字字模点阵或轮廓，用于显示/打印

问题：西文字符有没有输入码？有没有内码？
有没有字模点阵或轮廓描述？

汉字内码

- 至少需2个字节才能表示一个汉字内码。为什么？
 - 由汉字的总数决定！
 - 可在GB2312国标码的基础上产生汉字内码
 - 为与ASCII码区别，将国标码的两个字节的第一位置“1”后得到一种汉字内码
- 例如，汉字“大”在码表中位于第20行、第83列。因此区位码为0010100 1010011，国标码为00110100 01110011，即3473H。前面的34H和字符“4”的ASCII码相同，后面的73H和字符“s”的ASCII码相同，将每个字节的最高位各设为“1”后，就得到其内码：B4F3H (1011 0100 1111 0011B)，因而不会和ASCII码混淆。

汉字的字模点阵码和轮廓描述

- 为便于打印、显示汉字，汉字字形必须预先存在机内
 - 字库 (font): 所有汉字形状的描述信息集合
 - 不同字体 (如宋体、仿宋、楷体、黑体等) 对应不同字库
 - 从字库中找到字形描述信息，然后送设备输出

问题：如何知道到哪里找相应的字形信息？

汉字内码与其在字库中的位置有关！！

- 字形主要有两种描述方法：
 - 字模点阵描述（图像方式）
 - 轮廓描述（图形方式）
 - 直线向量轮廓
 - 曲线轮廓（True Type字形）

数据的基本宽度

- 比特 (bit) 是计算机中处理、存储、传输信息的最小单位
- 二进制信息的计量单位是“字节” (Byte), 也称“位组”
 - 现代计算机中, 存储器按字节编址
 - 字节是最小可寻址单位 (*addressable unit*)
 - 如果以字节为一个排列单位, 则LSB表示最低有效字节, MSB表示最高有效字节
- 除比特和字节外, 还经常使用“字” (word)作为单位
- “字” 和 “字长” 的概念不同
 - IA-32中的“字” 有多少位? 字长多少位呢?
16位 32位
 - DWORD : 32位
 - QWORD: 64位

数据的基本宽度

- “字” 和 “字长” 的概念不同

- “字长” 指数据通路的宽度。

(数据通路指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。因此，“字长”等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。)

- “字” 表示被处理信息的单位，用来度量数据类型的宽度。

- 字和字长的宽度可以一样，也可不同。

例如，x86体系结构定义“字”的宽度为16位，但从386开始字长就是32位了。

数据量的度量单位

- 存储二进制信息时的度量单位要比字节或字大得多
- 容量经常使用的单位有：
 - “千字节” (KB), $1\text{KB}=2^{10}\text{字节}=1024\text{B}$
 - “兆字节” (MB), $1\text{MB}=2^{20}\text{字节}=1024\text{KB}$
 - “千兆字节” (GB), $1\text{GB}=2^{30}\text{字节}=1024\text{MB}$
 - “兆兆字节” (TB), $1\text{TB}=2^{40}\text{字节}=1024\text{GB}$
- 通信中的带宽使用的单位有：
 - “千比特/秒” (kb/s), $1\text{kbps}=10^3\text{ b/s}=1000\text{ bps}$
 - “兆比特/秒” (Mb/s), $1\text{Mbps}=10^6\text{ b/s}=1000\text{ kbps}$
 - “千兆比特/秒” (Gb/s), $1\text{Gbps}=10^9\text{ b/s}=1000\text{ Mbps}$
 - “兆兆比特/秒” (Tb/s), $1\text{Tbps}=10^{12}\text{ b/s}=1000\text{ Gbps}$

如果把b换成B，则表示字节而不是比特（位）

例如，10MBps表示 10兆字节/秒

程序中数据类型的宽度

- 高级语言支持多种类型、多种长度的数据

- 例如，C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）

- 不同机器上表示的同一种类型的数据可能宽度不同

- 必须确定相应的机器级数据表示方式和相应的处理指令

从表中看出：同类型数据并不是所有机器都采用相同的宽度，分配的字节数随机器字长和编译器的不同而不同。

C语言中数值数据类型的宽度 (单位：字节)

C声明	典型32位机器	Compaq Alpha机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

Compaq Alpha是一个针对高端应用的64位机器，即字长为64位

数据的存储和排列顺序

- 80年代开始，几乎所有机器都用**字节编址** $65535 = 2^{16} - 1$
- ISA设计时要考虑的两个问题： $[-65535]_{\text{补}} = \text{FFFF0001H}$
 - 如何根据一个字节地址取到一个32位的字？ - **字的存放问题**
 - 一个字能否存放在任何字节边界？ - **字的边界对齐问题**

若 $\text{int } i = -65535$ ，存放在100号单元（占100~103），则用“取数”指令访问100号单元取出 i 时，必须清楚 i 的4个字节是如何存放的。

Word:	little endian word 100#			
	FF	FF	00	01
	103	102	101	100
	msb			lsb
	100	101	102	103
big endian word 100#				

大端方式 (Big Endian) : MSB所在的地址是数的地址

e.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

小端方式 (Little Endian) : LSB所在的地址是数的地址

e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持，可通过特定控制位来设定采用哪种方式。

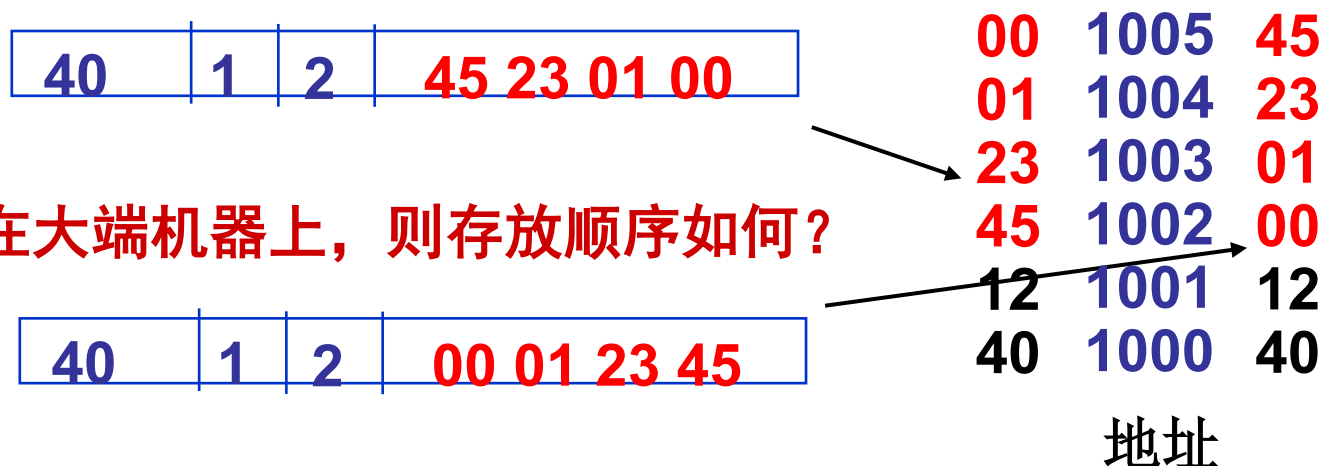
BIG Endian versus Little Endian

Ex3: Memory layout of a instruction located in 1000

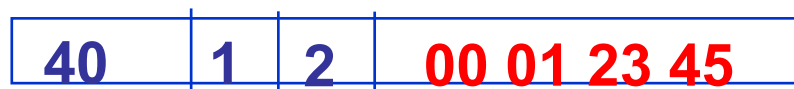
即指令地址为1000

假定小端机器中指令: `mov AX, 0x12345(BX)`

其中操作码mov为40H, 寄存器AX和BX的编号分别为0001B和0010B, 立即数占32位, 则存放顺序为:



若在大端机器上, 则存放顺序如何?



只需要考虑指令中立即数的顺序!

Byte Swap Problem (字节交换问题)

78	3
56	2
34	1
12	0

Big Endian

↑
increasing
byte
address

12	3
34	2
56	1
78	0

Little Endian

上述存放在0号单元的数据（字）是什么？ **12345678H?** **78563412H?**

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- ◆ 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- ◆ 因为顺序不同，需要进行顺序转换

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

ex. Little endian: GIF, PC Paintbrush, Microsoft RTF, etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

检测系统的字节顺序

- union的存放顺序是所有成员从低地址开始，利用该特性可测试CPU的大/小端方式。

```
#include <stdio.h>
void main()
{
    union NUM
    {
        int a;
        char b;
    } num;
    num.a = 0x12345678;
    if(num.b == 0x12)
        printf("Big Endian\n");
    else
        printf("Little Endian\n");
    printf("num.b = 0x%X\n", num.b);
}
```

Little Endian num.b = 0x78

请猜测在IA-32上的打印结果。

关于大端小端

有学生告诉我，他的同学写了一下程序，判断出他的PC是大端！

```
union test {  
    int  a;  
    char b;  
}
```

```
main() {  
    test.a=0xff;  
    if (test.b==0xff)  
        printf("Little endian");  
    else  
        printf("Big endian");  
}
```

未确定行为 (unspecified behavior) 语句



	大地址			小地址
小端	00	00	00	FF

C语言标准没有明确规定char为无符号还是带符号整型，当程序移植到另一个系统时，其行为可能发生变化，从而造成难以理解的结果。为避免这种情况，程序员应尽量编写行为确定的程序，对于一字节整数，应显式定义成signed char或unsigned char，作字符处理时，则可使用char型。

按照C语言标准，test.b从char型提升为int型，0xff为int型，故按int型比在IA-32中，char为signed char，扩展为32位后为全1，真值为-1；而0xff的真值是255；等式左右不等！

在RISC-V中，char为unsigned char，扩展为32位后为0x0000 00ff，因而，等式左右都是255，相等！

第二讲小结

- 非数值数据的表示

- 逻辑数据用来表示真/假或N位位串，按位运算
- 西文字符：用ASCII码表示
- 汉字：汉字输入码、汉字内码、汉字字模码

- 数据的宽度

- 位、字节、字（不一定等于字长）
- k / K / M / G / T / P / E / Z / Y 有不同的含义

- 数据的存储排列

- 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
 - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么？
- 大端方式：用MSB存放的地址表示数据的地址
- 小端方式：用LSB存放的地址表示数据的地址
- 按边界对齐可减少访存次数

数据的表示和运算

- 分以下三个部分介绍（续）

- **第三讲：数据的运算**

- 按位运算和逻辑运算
 - 移位运算
 - 位扩展和位截断运算
 - 无符号和带符号整数的加减运算
 - 无符号和带符号整数的乘除运算
 - 变量与常数之间的乘除运算
 - 浮点数的加减乘除运算

围绕C语言中的
运算，解释其
在底层机器级
的实现方法

从高级语言程序中的表达式出发，用机器数在具体电路中的
执行过程，来解释表达式的执行结果

数据的运算

- 高级语言程序中涉及的运算（以C语言为例）
 - 整数算术运算、浮点数算术运算
 - 按位、逻辑、移位、位扩展和位截断
- 指令集中涉及到的运算
 - 涉及到的定点数运算
 - 算术运算
 - 带符号整数运算：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
 - 无符号整数运算：0扩展 / 加 / 减 / 乘 / 除
 - 逻辑运算
 - 逻辑操作：与 / 或 / 非 / ...
 - 移位操作：逻辑左移 / 逻辑右移
 - 涉及到的浮点数运算：加、减、乘、除
- 基本运算部件ALU的设计



浮点数有没有移位操作和扩展操作？为什么？

C语言程序中涉及的运算

- 算术运算（最基本的运算）

- 无符号数、带符号整数、浮点数的+、-、*、/ 运算等

- 按位运算

- 用途

- 对位串实现“掩码”（mask）操作或相应的其他处理
（主要用于对多媒体数据或状态/控制信息进行处理）

- 操作

- 按位或： “|”
 - 按位与： “&”
 - 按位取反： “~”
 - 按位异或： “^”

问题：如何从16位采样数据y中提取高位字节，并使低字节为0？

可用 “&” 实现 “掩码” 操作： $y \& 0xFF00$

例如，当 $y=0x2C0B$ 时，得到结果为： $0x2C00$

C语言程序中涉及的运算

- 逻辑运算

- 用途

- 用于关系表达式的运算

例如, if (x>y and i<100) then中的 “and” 运算

- 操作

- “||” 表示 “OR” 运算
 - “&&” 表示 “AND” 运算

例如, if ((x>y) && (i<100)) then

- “!” 表示 “NOT” 运算

- 与按位运算的差别

- 符号表示不同: & 对 && ; | 对 ||;
 - 运算过程不同: 按位 对 整体
 - 结果类型不同: 位串 对 逻辑值

C语言程序中涉及的运算

- 移位运算

- 用途

- 提取部分信息

- 扩大或缩小数值的2、4、8...倍

- 操作

- 左移:: $x \ll k$; 右移: $x \gg k$
 - 不区分是逻辑移位还是算术移位, 由x的类型确定
 - 无符号数: 逻辑左移、逻辑右移

高(低)位移出, 低(高)位补0, 可能溢出!

问题: 何时可能发生溢出? 如何判断溢出?

若高位移出的是1, 则左移时发生溢出

- 带符号整数: 算术左移、算术右移

左移: 高位移出, 低位补0。可能溢出!

溢出判断: 若移出的位不等于新的符号位, 则溢出。

右移: 低位移出, 高位补符, 可能发生有效数据丢失。

如何从16位数据y中提取高位字节?

某字长为8的机器中, x、y和z都是8位带符号整数, 已知 $x=-81$, 则 $y=x/2=?$ $z=2x=?$

($y \gg 8$) 送8位寄存器

移位! $y=-40?$ $z=-162?$

C语言程序中涉及的运算

• 位扩展和位截断运算

– 用途

- 类型转换时可能需要数据扩展或截断

– 操作

- 没有专门操作运算符，根据类型转换前后数据长短确定是扩展还是截断
- 扩展：短转长
 - 无符号数：0扩展，前面补0
 - 带符号整数：符号扩展，前面补符
- 截断：长转短
 - 强行将高位丢弃，故可能发生“溢出”

例1（扩展操作）：在大端机上输出si, usi, i, ui的十进制和十六进制值是什么？

```
short si = -32768;      si = -32768    80 00
unsigned short usi = si; usi = 32768    80 00
int i = si;             i = -32768    FF FF 80 00
unsigned ui = usi;      ui = 32768    00 00 80 00
```

例2（截断操作）：i和j是否相等？

```
int i = 32768;
short si = (short) i;
int j = si;
```

不相等！

i = 32768 00 00 80 00

si = -32768 80 00

j = -32768 FF FF 80 00

原因：对i截断时发生了溢出，即：32768截断为16位数时，因其超出16位能表示的最大值，故无法截断为正确的16位数！

C语言标准规定，长数转换为短数的结果是未定义的，没有规定编译器必须报错

如何实现高级语言源程序中的运算？

- 总结：C语言程序中的基本数据类型及其基本运算类型

- 基本数据类型

- 无符号数、带符号整数、浮点数、位串、字符（串）

- 基本运算类型

- 算术、按位、逻辑、移位、扩展和截断、匹配

- 计算机如何实现高级语言程序中的运算？

- 将各类表达式编译（转换）为指令序列

- 计算机直接执行指令来完成运算

逻辑运算、移位、扩展和截断等指令实现较容易，算术运算指令难！

例：C语言赋值语句“ $f = (g+h) - (i+j);$ ”中变量i、j、f、g、h由编译器分别分配给MIPS寄存器\$*t*0~\$*t*4。寄存器\$*t*0~\$*t*7的编号对应8~15，上述程序段对应的MIPS机器代码和汇编表示（#后为注释）如下：

000000 01011 01100 01101 00000 100000 add \$*t*5, \$*t*3, \$*t*4 # $g+h$

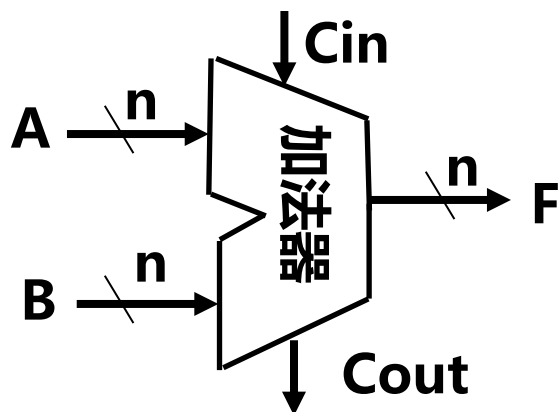
000000 01000 01001 01110 00000 100000 add \$*t*6, \$*t*0, \$*t*1 # $i+j$

000000 01101 01110 01010 00000 100010 sub \$*t*2, \$*t*5, \$*t*6 # $f=(g+h)-(i+j)$

需要提供哪些运算类指令才能支持高级语言需求呢？

整数加、减运算

- C语言程序中的整数有
 - 带符号整数，如char、short、int、long型等
 - 无符号整数，如unsigned char、unsigned short、unsigned等
- 指针、地址等通常被说明为无符号整数，因而在进行指针或地址运算时，需要进行无符号整数的加、减运算
- 无符号整数和带符号整数的加、减运算电路完全一样，这个运算电路称为**整数加减运算部件**，基于**带标志加法器**实现
- **最基本的加法器**，因为只有n位，所以是一种**模 2^n 运算系统**！



例：n=4, A=1001, B=1100

则：F=0101, $C_{out}=1$

还记得这个加法器是如何实现的？

n位整数加/减运算器

- 补码加减运算公式

$$[A+B]_{\text{补}} = [A]_{\text{补}} + [B]_{\text{补}} \pmod{2^n}$$

$$[A-B]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}} \pmod{2^n}$$

— 实现减法的主要工作在于：求 $[-B]_{\text{补}}$

问题：如何求 $[-B]_{\text{补}}$ ？

$$[-B]_{\text{补}} = \overline{[B]_{\text{补}}} + 1$$

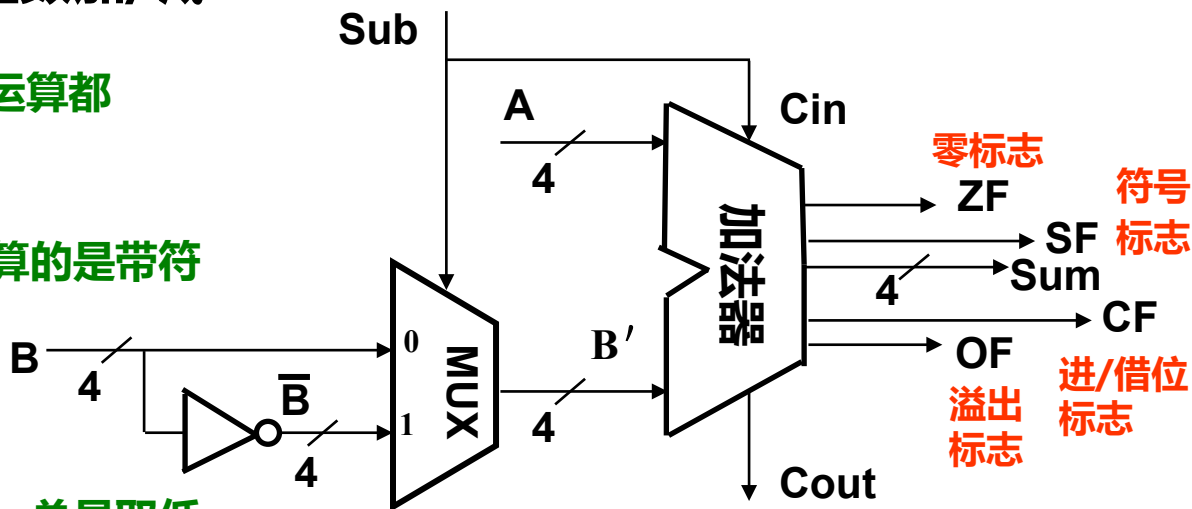
- 利用带标志加法器，可构造整数加/减运算器，进行无/带符号整数加/减

当Sub为1时，做减法
当Sub为0时，做加法

重要认识1：计算机中所有算术运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

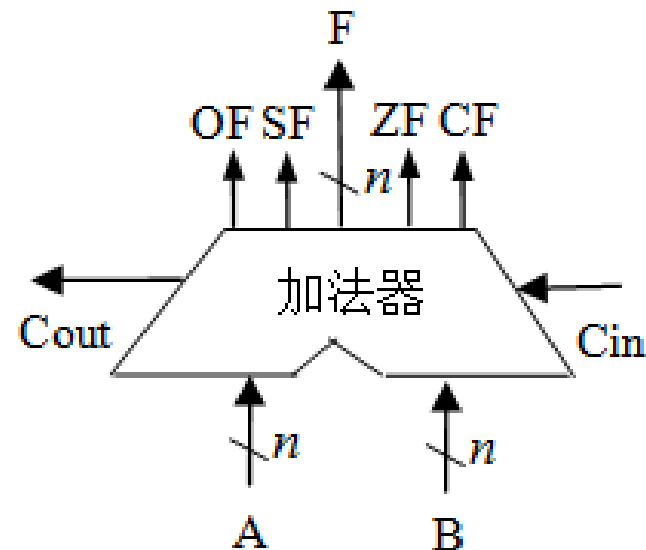
重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。



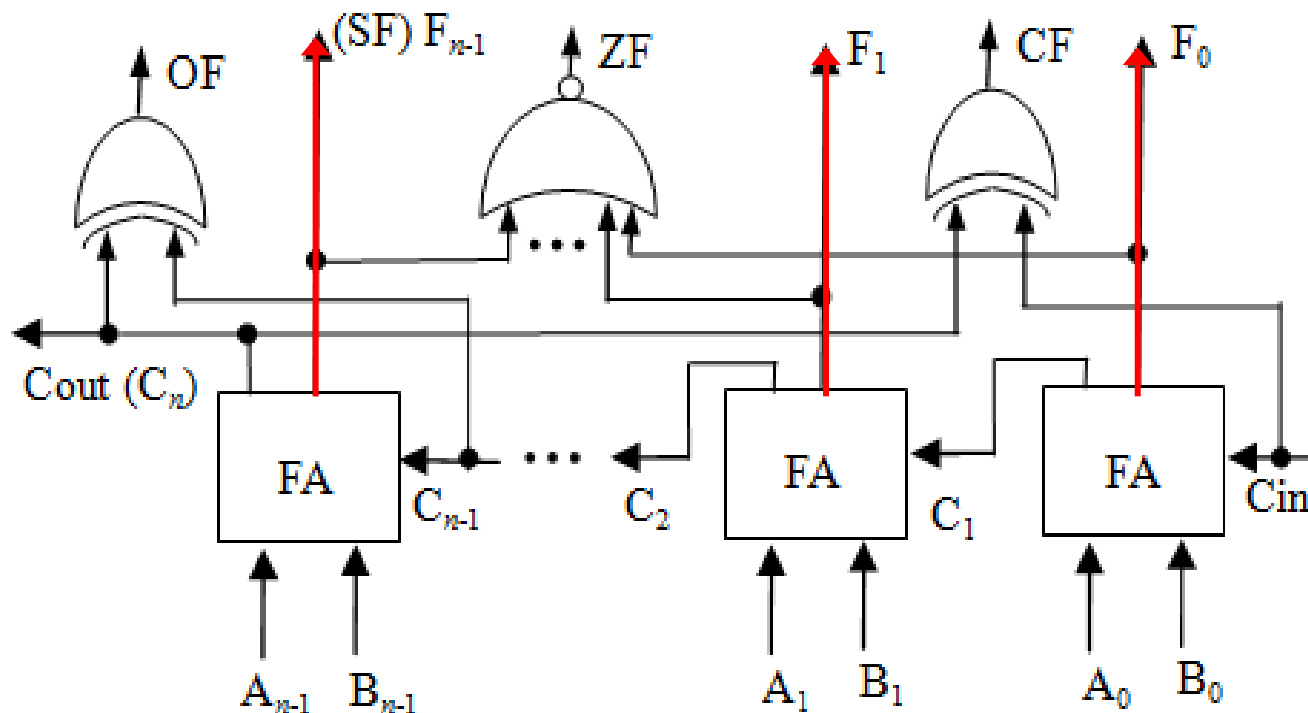
整数加/减运算部件

n位带标志加法器

- n位加法器无法用于两个n位带符号整数（补码）相加，无法判断是否溢出
- 程序中经常需要比较大小，通过（在加法器中）做减法得到的标志信息来判断



带标志加法器符号



带标志加法器的逻辑电路

溢出标志OF:

$$OF = C_n \oplus C_{n-1}$$

符号标志SF:

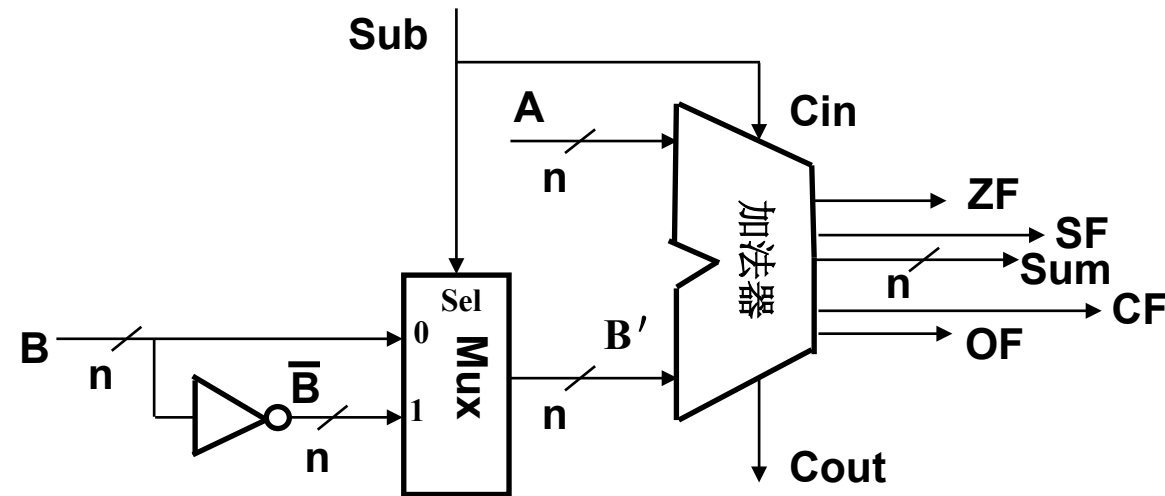
$$SF = F_{n-1}$$

零标志ZF=1当且仅当F=0;

进位/借位标志CF:

$$CF = Cout \oplus Cin$$

条件标志位（条件码CC）



整数加/减运算部件

问题：为什么要生成并保存条件标志？

为了在分支指令（条件转移指令）中被用作是否转移执行的条件！

问题：OF=? ZF=?
SF=? CF=?

```
if (i>j) {  
    ...  
}
```

OF：若A与B' 同号但与Sum不同号，则1；否则0。 **SF**：sum符号

ZF：如Sum为0，则1，否则0。 **CF**： $Cout \oplus sub$

- 零标志**ZF**、溢出标志**OF**、进/借位标志**CF**、符号标志**SF**称为条件标志。
- 条件标志（Flag）在运算电路中产生，被记录到专门的寄存器中
- 存放标志的寄存器通常称为程序/状态字寄存器或标志寄存器。每个标志对应标志寄存器中的一个标志位。 如，IA-32中的EFLAGS寄存器

整数加法举例

```
unsigned int x=134;  
unsigned int y=246;  
int m=x;  
int n=y;  
unsigned int z1=x-y;  
unsigned int z2=x+y;  
int k1=m-n;  
int k2=m+n;
```

无符号加公式:

$$\text{result} = \begin{cases} x+y & (x+y < 2^n) \\ x+y-2^n & (2^n \leq x+y < 2^{n+1}) \end{cases}$$

带符号加公式:

$$\text{result} = \begin{cases} x+y-2^n & (2^{n-1} \leq x+y) & \text{正溢出} \\ x+y & (-2^{n-1} \leq x+y < 2^{n-1}) & \text{正常} \\ x+y+2^n & (x+y < -2^{n-1}) & \text{负溢出} \end{cases}$$

x和m的机器数一样: 1000 0110, y和n的机器数一样: 1111 0110

z2和k2的机器数一样: 0111 1100, CF=1, OF=1, SF=0

z2的值为124 (=134+246-256, x+y>256)

k2的值为124 (=-122+(-10)+256, m+n=-132<-128, 即负溢出)

整数加法举例

做加法时，主要判断是否溢出

无符号加溢出条件：CF=1

带符号加溢出条件：OF=1

若 $n=8$ ，计算 $107+46=?$

$$107_{10} = 0110\ 1011_2$$

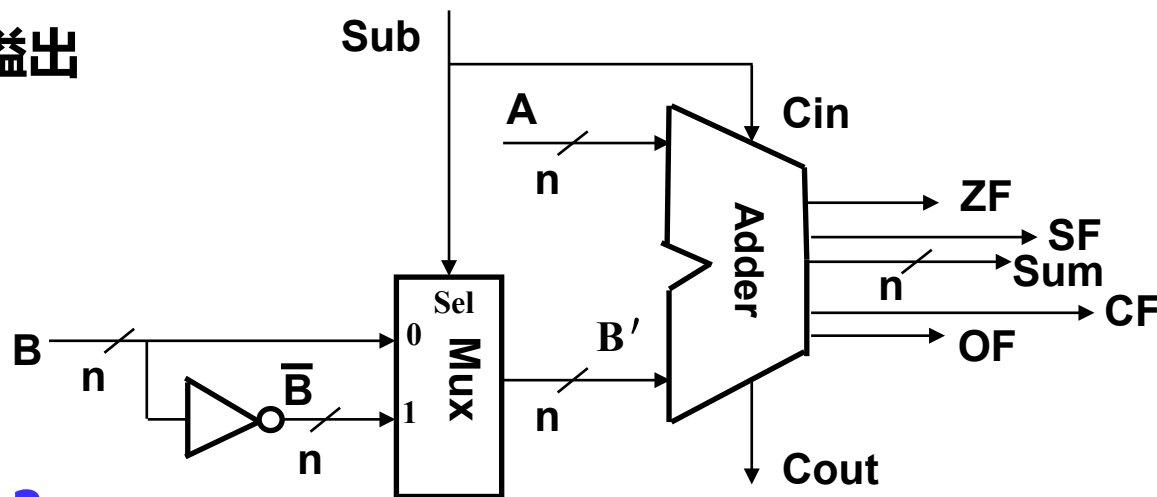
$$46_{10} = 0010\ 1110_2$$

$$\boxed{0}1001\ 1001$$

进位是真正的符号：+153

无符号：sum=153，因为CF=0，故未发生溢出，结果正确！

带符号：sum= -103，因为OF=1，故发生溢出，结果错误！



整数加/减运算部件

两个正数相加，结果为负数，故溢出！即OF=1

溢出标志OF=1、零标志ZF=0、符号标志SF=1、进位标志CF=0

整数减法举例

```
unsigned int x=134;  
unsigned int y=246;  
int m=x;  
int n=y;  
unsigned int z1=x-y;  
unsigned int z2=x+y;  
int k1=m-n;  
int k2=m+n;
```

无符号减公式:

$$\text{result} = \begin{cases} x-y & (x-y > 0) \\ x-y+2^n & (x-y < 0) \end{cases}$$

带符号减公式:

$$\text{result} = \begin{cases} x-y-2^n & (2^{n-1} \leq x-y) & \text{正溢出} \\ x-y & (-2^{n-1} \leq x-y < 2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y < -2^{n-1}) & \text{负溢出} \end{cases}$$

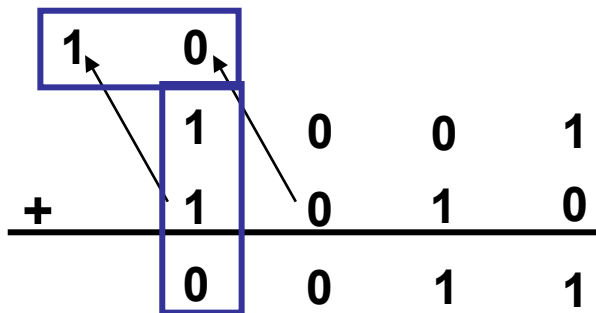
x和m的机器数一样: 1000 0110, y和n的机器数一样: 1111 0110

z1和k1的机器数一样: 1001 0000, CF=1, OF=0, SF=1

z1的值为144 (=134-246+256, x-y<0), k1的值为-112。

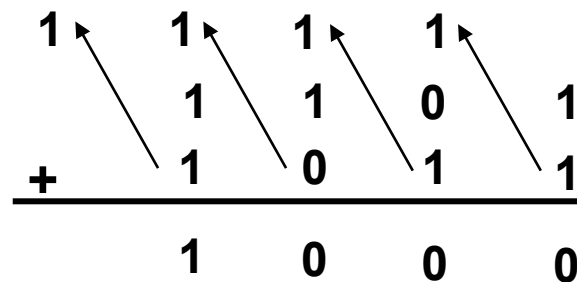
整数减法举例

$$\begin{aligned} -7 - 6 &= -7 + (-6) = +3 \times \\ 9 - 6 &= 3 \checkmark \end{aligned}$$



OF=1、ZF=0
SF=0、借位CF=0

$$\begin{aligned} -3 - 5 &= -3 + (-5) = -8 \checkmark \\ 13 - 5 &= 8 \checkmark \end{aligned}$$



OF=0、ZF=0、
SF=1、借位CF=0

带符号 (1) 最高位和次高位的进位不同
溢出: (2) 和的符号位和加数的符号位不同

无符号减溢出: 差为负数, 即借位CF=1

做减法以比较大小, 规则:
Unsigned: CF=0时, 大于
Signed: OF=SF时, 大于

验证: $9 > 6$, 故CF=0; $13 > 5$, 故CF=0

验证: $-7 < 6$, 故OF≠SF
 $-3 < 5$, 故OF≠SF

无符号整数加法溢出判断程序

如何用程序判断一个**无符号数相加**没有发生溢出

$$\text{result} = \begin{cases} x+y & (x+y < 2^n) \\ x+y-2^n & (2^n \leq x+y < 2^{n+1}) \end{cases}$$

发生溢出时，一定满足 $\text{result} < x$ and $\text{result} < y$
否则，若 $x+y-2^n \geq x$ ，则 $y \geq 2^n$ ，这是不可能的！

做模拟器实验
时，模拟Add
指令需要用！

/ Determine whether arguments can be added
without overflow */*

```
int uadd_ok(unsigned x, unsigned y)
{
    unsigned sum = x+y;
    return sum >= x;
}
```


带符号整数加法溢出判断程序

如何用程序判断一个带符号整数相加没有发生溢出

- `/* Determine whether arguments can be added without overflow */`

```
int tadd_ok(int x, int y) {  
    int sum = x+y;  
    int neg_over = x < 0 && y < 0 && sum >= 0;  
    int pos_over = x >= 0 && y >= 0 && sum < 0;  
    return !neg_over && !pos_over;  
}
```

做PA2模拟Add指令需要用以下公式:

CF=?, ZF=?, OF=?, SF=?

$$\text{result} = \begin{cases} x+y-2^n & (2^{n-1} \leq x+y) & \text{正溢出} & \text{CF=0, ZF=0, OF=1, SF=1} \\ x+y & (-2^{n-1} \leq x+y < 2^{n-1}) & \text{正常} & \\ x+y+2^n & (x+y < -2^{n-1}) & \text{负溢出} & \text{CF=1, ZF=0, OF=1, SF=0} \end{cases}$$

带符号整数减法溢出判断程序

$$\text{result} = \begin{cases} x+y-2^n & (2^{n-1} \leq x+y) & \text{正溢出} \\ x+y & (-2^{n-1} \leq x+y < 2^{n-1}) & \text{正常} \\ x+y+2^n & (x+y < -2^{n-1}) & \text{负溢出} \end{cases} \quad \text{带符号整数加}$$

$$\text{result} = \begin{cases} x-y-2^n & (2^{n-1} \leq x-y) & \text{正溢出} \\ x-y & (-2^{n-1} \leq x-y < 2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y < -2^{n-1}) & \text{负溢出} \end{cases} \quad \begin{array}{l} \text{带符号整数减} \\ \text{模拟Sub指令需要!} \end{array}$$

以下程序检查带符号整数相减是否溢出有没有问题？

```
/* Determine whether arguments can be subtracted
without overflow */
```

```
/* WARNING: This code is buggy. */
```

```
int tsub_ok(int x, int y) {
    return tadd_ok(x, -y);
```

```
}
```

当 $x \geq 0$, $y = 0x80000000$ 时, 该函数判断错误

带符号减的溢出判断
函数如何实现呢？

无符号减的溢出判断
函数又如何实现呢？

带符号整数减法溢出判断程序

• /*

* subOK - Determine if can compute x-y without overflow

* Example: subOK(0x80000000,0x80000000) = 1,

* subOK(0x80000000,0x70000000) = 0,

* Legal ops: ! ~ & ^ | + << >>

* Max ops: 20

* Rating: 3

*/

```
int subOK(int x, int y) {
```

```
    int diff = x + ~y + 1;
```

```
    int x_neg = x >> 31;
```

```
    int y_neg = y >> 31;
```

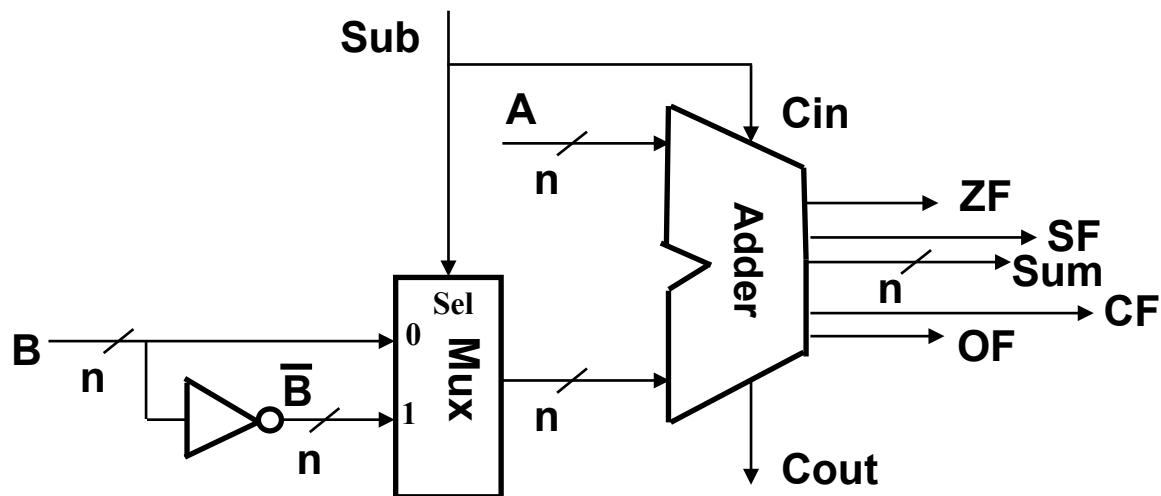
```
    int d_neg = diff >> 31;
```

```
    /* Overflow when x and y have opposite sign, and d different from x
```

```
    */
```

```
    return !((~(x_neg ^ ~y_neg) & (x_neg ^ d_neg)));
```

```
}
```



当x和~y同号，且和diff不同号，则发生溢出！

整数的乘运算

- 通常，高级语言中两个n位整数相乘得到的结果通常也是一个n位整数，也即，结果只取2n位乘积中的低n位。
 - 例如，在C语言中，参加运算的两个操作数的类型和结果的类型必须一致，如果不一致则会先转换为一致的数据类型再进行计算。

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

x*y 被转换为乘法指令，在乘法运算电路中得到的乘积是64位，但是，只取其低32位赋给z。

整数的乘运算

结论：假定两个n位无符号整数 x_u 和 y_u 对应的机器数为 X_u 和 Y_u ,

$p_u = x_u \times y_u$, p_u 为n位无符号整数且对应的机器数为 P_u ;

两个n位带符号整数 x_s 和 y_s 对应的机器数为 X_s 和 Y_s , $p_s = x_s \times y_s$, p_s

为n位带符号整数且对应的机器数为 P_s 。

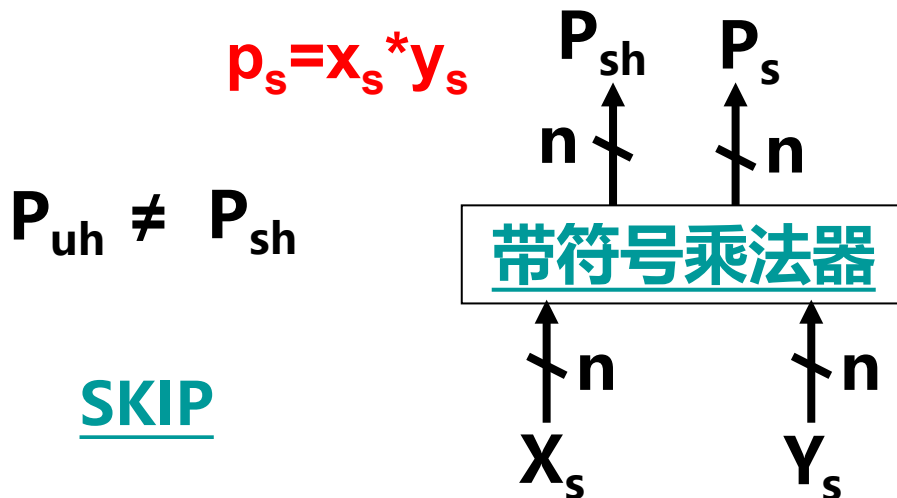
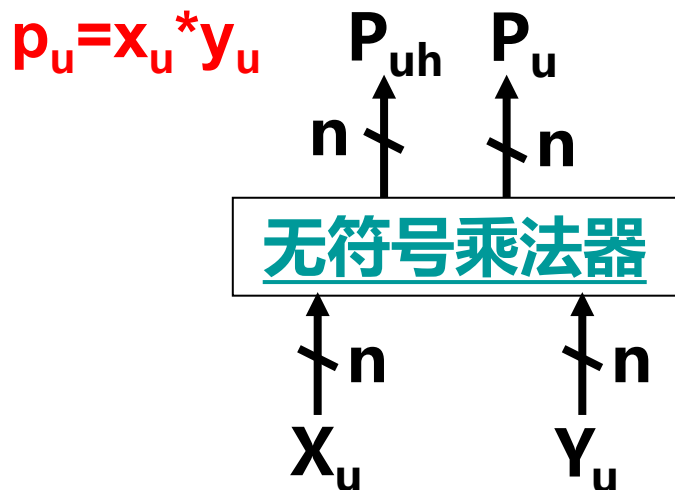
例: $X_u = X_s = ?$, $Y_u = Y_s = ?$

若 $X_u = X_s$ 且 $Y_u = Y_s$, 则 $P_u = P_s$ 。

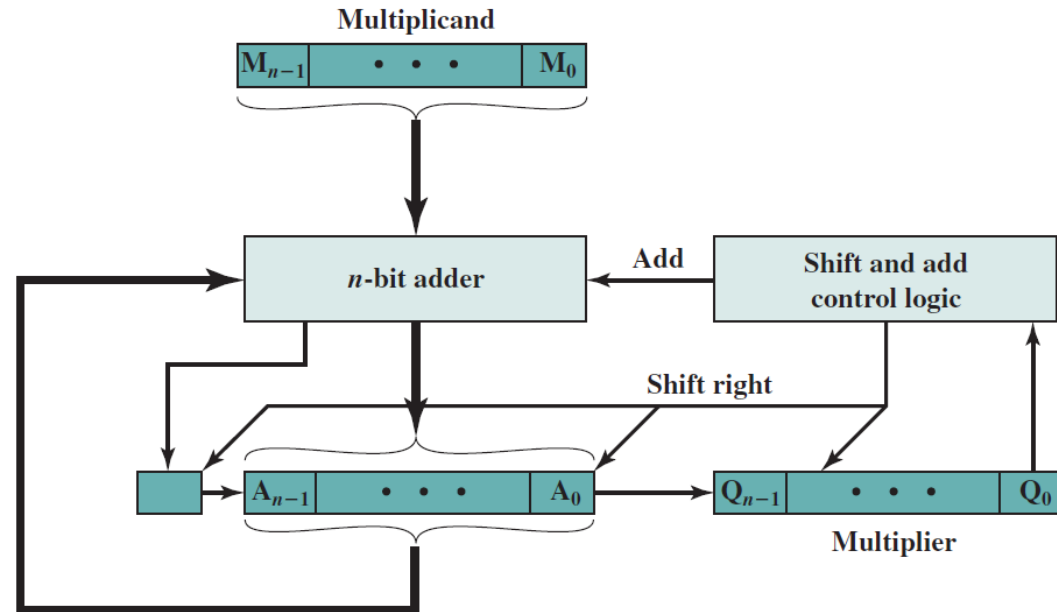
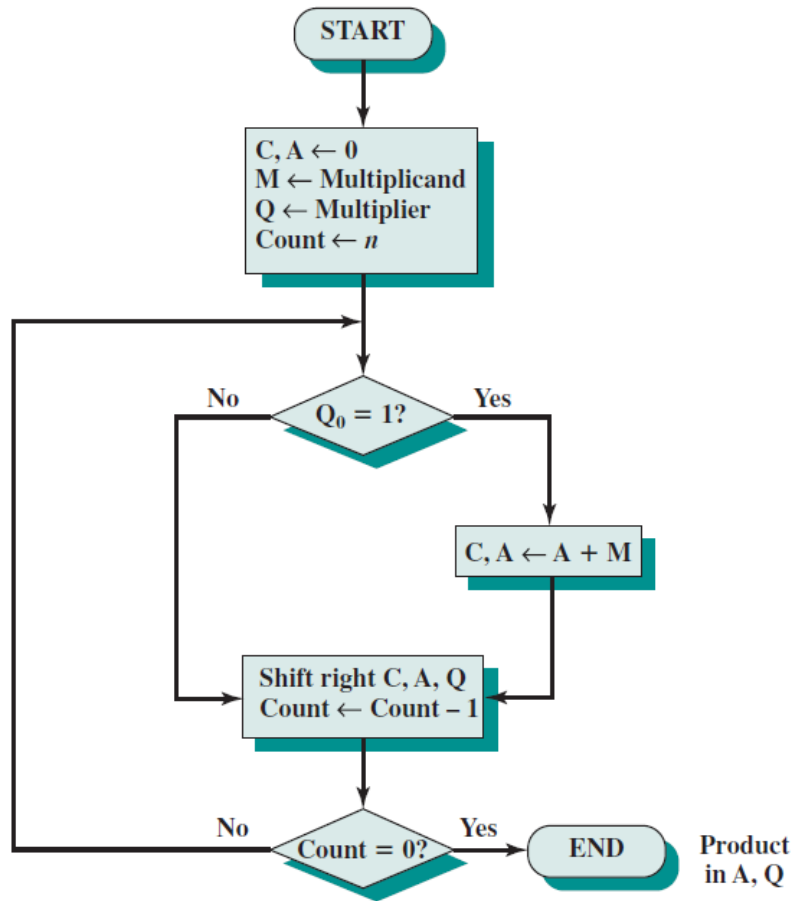
无符号: 若 $P_{uh} = 0$, 则不溢出

可用无符号乘来实现带符号乘, 但高n位无法得到, 故不能判断溢出。

带符号: 若 P_{sh} 每位都等于 P_s 的最高位, 则不溢出



无符号整数乘法



$ \begin{array}{r} 1001 \quad (9) \\ \times 0011 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ 00010010 \quad 1001 \times 2^1 \\ \hline 00011011 \quad (27) \end{array} $		$ \begin{array}{r} 1001 \quad (-7) \\ \times 0011 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 = (-7) \\ 11110010 \quad (-7) \times 2^1 = (-14) \\ \hline 11101011 \quad (-21) \end{array} $	
---	--	--	--

(a) Unsigned integers

(b) Twos complement integers

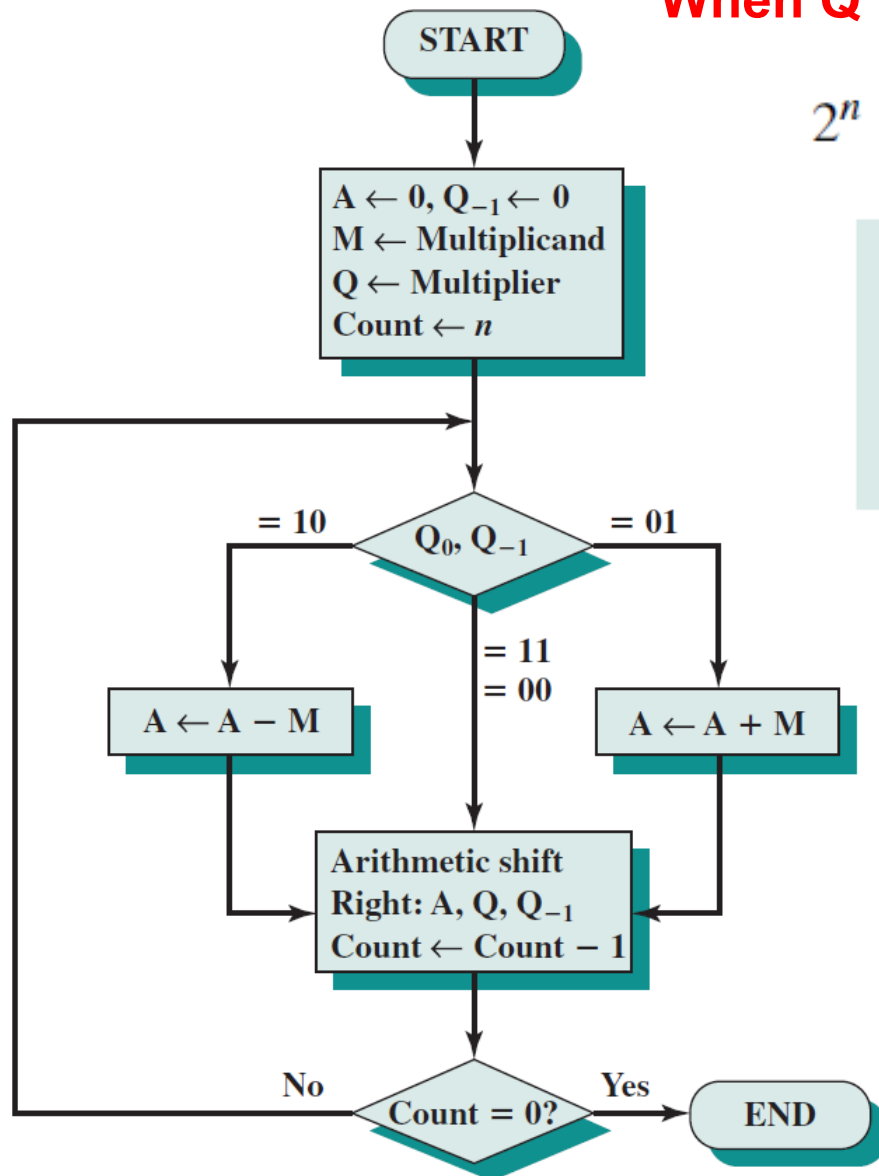
Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

补码乘法：Booth's algorithm

When Q is positive:

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

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$



Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1-0) and an addition when the end of the block is encountered (0-1).

补码乘法：Booth's algorithm

When Q(X) is negative:

$$\begin{aligned} \text{Representation of } X &= \{1x_n-2x_{n-3} \dots x_1x_0\} \\ &= \{111 \dots 1\underbrace{0}_{x_k}x_{k-1}x_{k-2} \dots x_1x_0\} \end{aligned}$$

$$X = -2^{n-1} + \underbrace{2^{n-2} + \dots + 2^{k+1}}_{\text{highlighted in yellow}} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0)$$

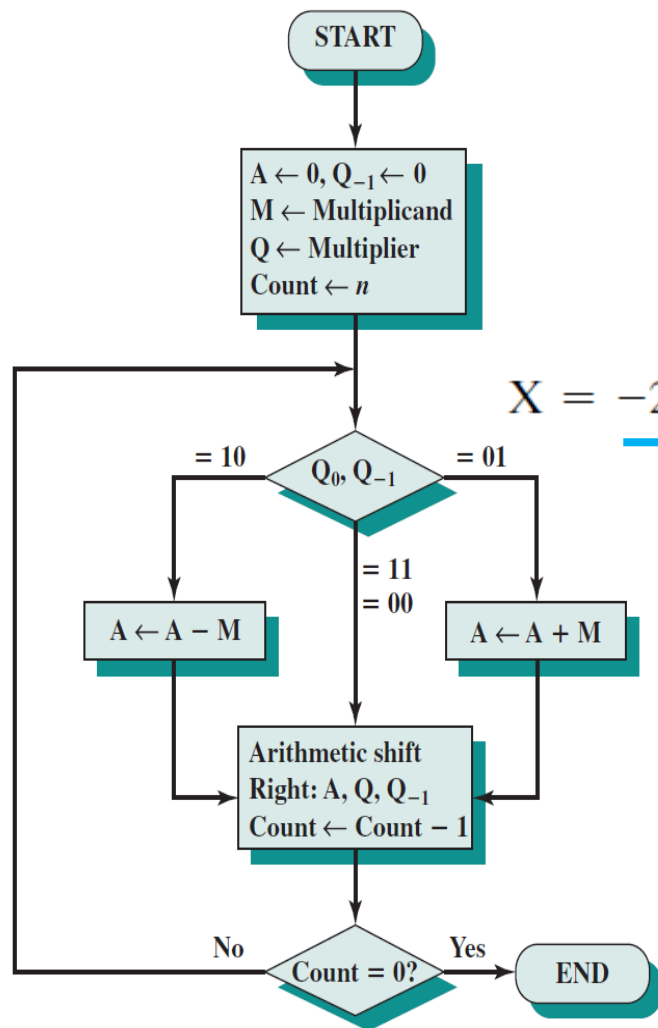
$$2^{n-2} + 2^{n-3} + \dots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^{k+1} = -2^{k+1}$$

$$X = \underbrace{-2^{k+1}}_{\text{highlighted in blue}} + (x_{k-1} \times 2^{k-1}) + \dots + (x_0 \times 2^0)$$

10转换
对应'-'

为正乘数 ($x_k=0$)，按前述可正确处理



整数的乘运算

- $X \times Y$ 的高n位可以用来判断溢出，规则如下：
 - 无符号：若高n位全0，则不溢出，否则溢出
 - 带符号：若高n位全0或全1且等于低n位的最高位，则不溢出。

运算	x	X	y	Y	$x \times y$	$X \times Y$	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	<u>0000 0110</u>	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	<u>1111 1000</u>	-8	1000	不溢出

整数乘法溢出漏洞

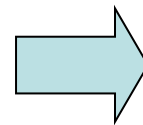
以下程序存在什么漏洞，引起该漏洞的原因是什么。

```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
    int i;
    /* 在堆区申请一块内存 */
    int *myarray = (int *) malloc(count*sizeof(int));
    if (myarray == NULL)
        return -1;
    for (i = 0; i < count; i++)
        myarray[i] = array[i];
    return count;
}
```

2002年，Sun Microsystems公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

攻击者可构造特殊参数来触发整数溢出，以一段**预设信息覆盖一个已分配的堆缓冲区**，造成远程服务器崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。
如count= $2^{30}+1$ 时，
count*sizeof(int)=4。



堆 (heap) 中大量数据被破坏!

变量与常数之间的乘运算

- 整数乘法运算比移位和加法等运算所用时间长，通常一次乘法运算需要多个时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期，因此，编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。

例如，对于表达式 $x*20$ ，编译器可以利用

$20=16+4=2^4+2^2$ ，将 $x*20$ 转换为 $(x<<4)+(x<<2)$ ，这样，一次乘法转换成了两次移位和一次加法。

- 不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

关于乘运算的几个问题

- 假定CPU中没有乘法器，只有ALU、移位器以及与、或、非等逻辑电路，则其指令系统能提供乘法指令吗？
- 假定ISA中不包含乘法指令，但包含加、减、移位以及与、或、非三种逻辑运算指令，则基于该ISA的系统能执行以下程序吗？

```
int imul(int x, int y) {  
    return x*y;  
}
```

- 以下两个函数的机器级代码相同吗？返回的结果一定相同吗？什么情况下相同？返回的结果一定正确吗？

```
int imul(int x, int y) {  
    return x*y;  
}
```

```
unsigned mul(unsigned x, unsigned y) {  
    return x*y;  
}
```

- ①用循环程序段实现
- ②直接用较慢乘法指令实现
- ③直接用较快乘法指令实现

- 上述程序执行时间比较

①无乘法指令 > ②用ALU实现乘法指令 > ③用乘法器实现乘法指令

整数的除运算

- 对于带符号整数来说， n 位整数除以 n 位整数，除 $-2^{n-1}/-1 = 2^{n-1}$ 会发生溢出外，其余情况（除数为0外）都不会发生溢出。Why?

因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞。

- 因为整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即正数商取比自身小的最接近整数（Floor，地板），负数商取比自身大的最接近整数（Ceiling，天板）。

例如， $7/2=?$ ， $-7/2=?$

$7/2=3$ ， $-7/2=-3$

- 整数除0的结果可以用什么机器数表示？

整数除0的结果无法用一个机器数表示！

- 整数除法时，除数不能为0，否则会发生“异常”，此时，需要调出操作系统中的异常处理程序来处理。

变量与常数之间的除运算

- 对于整数除法运算，由于计算机中除法运算比较复杂，而且不能用流水线方式实现，所以一次除法运算大致需要几十个或更多个时钟周期，比乘法指令的时间还要长！
- 为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。
 - 无符号：逻辑右移
 - 带符号：算术右移
- 结果一定取整数
 - 能整除时，直接右移得到结果，移出的为全0
例如， $12/4=3$ ：0000 1100 >> 2 = 0000 0011
 $-12/4=-3$ ：1111 0100 >> 2 = 1111 1101
 - 不能整除时，右移移出的位中有非0，需要进行相应处理

变量与常数之间的除运算

- 不能整除时，采用朝零舍入，即截断方式
 - 无符号数、带符号正整数（地板）：移出的低位直接丢弃
 - 带符号负整数（天板）：加偏移量(2^k-1)，然后再右移 k 位，低位截断（这里 k 是右移位数）

举例：

无符号数 $14/4=3$: $0000\ 1110 \gg 2 = 0000\ 0011$

带符号负整数 $-14/4=-3$

若直接截断，则 $1111\ 0010 \gg 2 = 1111\ 1100 = -4 \neq -3$

应先纠偏，再右移: $k=2$, 故 $(-14+2^2-1)/4=-3$

即: $1111\ 0010 + 0000\ 0011 = 1111\ 0101$

$1111\ 0101 \gg 2 = 1111\ 1101 = -3$

变量与常数之间的除运算—举例

- 假设 x 为一个int型变量，请给出一个用来计算 $x/32$ 的值的函数div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

解：若 x 为正数，则将 x 右移 k 位得到商；若 x 为负数，则 x 需要加一个偏移量 (2^k-1) 后再右移 k 位得到商。因为 $32=2^5$ ，所以 $k=5$ 。

即结果为: $(x \geq 0 ? x : (x+31)) \gg 5$

但题目要求不能用比较和条件语句，因此要找一个计算偏移量 b 的方式

这里， x 为正时 $b=0$ ， x 为负时 $b=31$ 。因此，可以从 x 的符号得到 b

$x \gg 31$ 得到的是32位符号，取出最低5位，就是偏移量 b 。

```
int div32(int x)
{ /* 根据x的符号得到偏移量b */
    int b=(x>>31) & 0x1F;
    return (x+b)>>5;
}
```


浮点数运算及结果

设两个规格化浮点数分别为 $A = M_a \cdot 2^{E_a}$ $B = M_b \cdot 2^{E_b}$,则:

$$A \pm B = (M_a \pm M_b \cdot 2^{-(E_a - E_b)}) \cdot 2^{E_a} \quad (\text{假设 } E_a \geq E_b) \quad 1.5 + 1.5 = ?$$

$$A * B = (M_a * M_b) \cdot 2^{E_a + E_b} \quad 1.5 - 1.0 = ?$$

$$A / B = (M_a / M_b) \cdot 2^{E_a - E_b}$$

上述运算结果可能出现以下几种情况:

SP最大指数为多少? 127

阶码上溢: 一个正指数超过了最大允许值 $\Rightarrow +\infty / -\infty / \text{溢出}$

阶码下溢: 一个负指数比最小允许值还小 $\Rightarrow +0 / -0$ SP最小指数呢?

尾数溢出: 最高有效位有进位 \Rightarrow 右规

-126-23

非规格化尾数: 数值部分高位为0 \Rightarrow 左规

尾数溢出, 结果不一定溢出

右规或对阶时, 右段有效位丢失 \Rightarrow 尾数舍入 运算过程中添加保护位

IEEE建议实现时为每种异常情况提供一个自陷允许位。若某异常对应的位为1, 则发生相应异常时, 就调用一个特定的异常处理程序执行。

浮点数加/减运算

- 十进制科学计数法的加法例子

$$1.123 \times 10^5 + 2.560 \times 10^2$$

其计算过程为：

$$\begin{aligned} 1.123 \times 10^5 + 2.560 \times 10^2 &= 1.123 \times 10^5 + 0.002560 \times 10^5 \\ &= (1.123 + 0.00256) \times 10^5 = 1.12556 \times 10^5 \\ &= 1.126 \times 10^5 \end{aligned}$$

进行尾数加减运算前，必须“对阶”！最后还要考虑舍入
计算机内部的二进制运算也一样！

- “对阶”操作：目的是使两数阶码相等
 - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
 - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

浮点数加减法基本要点

(假定: X_m 、 Y_m 分别是X和Y的尾数, X_e 和 Y_e 分别是X和Y的阶码)

- (1) 求阶差: $\Delta e = Y_e - X_e$ (若 $Y_e > X_e$, 则结果的阶码为 Y_e)
- (2) 对阶: 将 X_m 右移 Δe 位, 尾数变为 $X_m * 2^{X_e - Y_e}$ (保留右移部分**附加位**)
- (3) 尾数加减: $X_m * 2^{X_e - Y_e} \pm Y_m$

(4) 规格化:

当尾数高位为0, 则需左规: 尾数左移一次, 阶码减1, 直到MSB为1或阶码为00000000 (-126, 非规格化数)

每次阶码减1后要判断阶码是否下溢 (比最小可表示的阶码还要小)

当尾数最高位有进位, 需右规: 尾数右移一次, 阶码加1, 直到MSB为1

每次阶码加1后要判断阶码是否上溢 (比最大可表示的阶码还要大)

阶码溢出异常处理: 阶码上溢, 则结果溢出; 阶码下溢到无法用非规格化数表示, 则结果为0

- (5) 如果尾数比规定位数长 (有附加位), 则需考虑舍入 (有多种舍入方式)
- (6) 若**运算结果尾数**是0, 则需要将阶码也置0。为什么?

尾数为0说明结果应该为0 (阶码和尾数为全0)。

0.00...0001 $\times 2^{-126}$



浮点数加/减运算-对阶

问题：如何对阶？

通过计算 $[\Delta E]_{\text{补}}$ 来判断两数的阶差：

$$[\Delta E]_{\text{补}} = [Ex - Ey]_{\text{补}} = [Ex]_{\text{移}} + [-[Ey]_{\text{移}}]_{\text{补}} \pmod{2^n}$$

问题：在 ΔE 为何值时无法根据 $[\Delta E]_{\text{补}}$ 来判断阶差？ 溢出时！

例如，4位移码， $Ex=7$ ， $Ey=-7$ ，则 $[\Delta E]_{\text{补}}=1110+0000=1110$ ， $\Delta E<0$ ，错！

问题：对IEEE754 SP格式来说， $|\Delta E|$ 大于多少时，结果就等于阶大的那个数（小数被大数吃掉）？

25！

1.xx...x

0.00...01xx...x

24：吃不掉☹

1.xx...x

0.00...001xx...x

25：吃掉了☺

问题：IEEE754 SP格式的偏置常数是127，这会不会影响阶码运算电路的复杂度？ 对计算 $[Ex - Ey]_{\text{补}} \pmod{2^n}$ 没有影响

$$[\Delta E]_{\text{补}} = 256 + Ex - Ey = 256 + 127 + Ex - (127 + Ey)$$

$$= 256 + [Ex]_{\text{移}} - [Ey]_{\text{移}} = [Ex]_{\text{移}} + [-[Ey]_{\text{移}}]_{\text{补}} \pmod{256}$$

但 $[Ex + Ey]_{\text{移}}$ 和 $[Ex - Ey]_{\text{移}}$ 的计算会变复杂！ 浮点乘除运算涉及之。

浮点数加法运算举例

例：用二进制浮点数形式计算 $0.5 + (-0.4375) = ?$

$$0.4375 = 0.25 + 0.125 + 0.0625 = 0.0111\text{B}$$

解： $0.5 = 1.000 \times 2^{-1}$, $-0.4375 = -1.110 \times 2^{-2}$

对阶： $-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

加减： $1.000 \times 2^{-1} + (-0.111 \times 2^{-1}) = 0.001 \times 2^{-1}$

左规： $0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

判溢出：无

结果为： $1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

问题：为何IEEE 754 加减运算右规时最多只需一次？

因为即使是两个最大的尾数相加，得到的和的尾数也不会达到4，故尾数的整数部分最多有两位，保留一个隐含的“1”后，最多只有一位被右移到小数部分。

Extra Bits(附加位)

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt. "

“浮点数就像一堆沙，每动一次就会失去一点‘沙’，并捡回一点‘脏’”

如何才能使失去的“沙”和捡回的“脏”都尽量少呢？在后面加附加位！

加多少附加位才合适？

无法给出准确的答案！

Add/Sub:

1.xxxxx	1.xxxxx	1.xxxxx	1.xxxxxxxxx
+ 1.xxxxx	0.001xxxx	0.01xxxx	-1.xxxxxxxxx
<hr/>	<hr/>	<hr/>	<hr/>
1x.xxxx y	1.xxxxx yyy	1x.xxxx yyy	0.0...0xxxx

IEEE754规定: 中间结果须在右边加2个附加位 (guard & round)

Guard (保护位): 在significand右边的位

Round (舍入位): 在保护位右边的位

附加位的作用: 用以保护对阶时右移的位或运算的中间结果。

附加位的处理: ①左规时被移到significand中; ② 作为舍入的依据。

Rounding Digits(舍入位)

举例：若十进制数最终有效位数为 3，采用两位附加位（**G**、**R**）。

问题：若没有舍入位，采用就近舍入到偶数，则结果是什么？

结果为2.36！精度没有2.37高！

$$2.34\textcolor{blue}{0}\textcolor{red}{0} * 10^2$$

$$0.02\textcolor{blue}{5}\textcolor{red}{3} * 10^2$$

$$\hline 2.36\textcolor{blue}{5}\textcolor{red}{3} * 10^2$$

IEEE Standard: four rounding modes (用图说明)

round to nearest (**default**)

round towards plus infinity (always round up)

round towards minus infinity (always round down)

round towards 0

round to nearest: **称为就近舍入到偶数**

round digit < 1/2 then truncate (**截断、丢弃**)

> 1/2 then round up (**末位加1**)

= 1/2 then round to nearest even digit (**最近偶数**)

可以证明默认方式得到的平均误差最小。

IEEE 754的舍入方式的说明

IEEE 754的舍入方式



(Z1和Z2分别是结果Z的最近的可表示的左、右两个数)

(1) 就近舍入：舍入为最近可表示的数

非中间值：0舍1入；

中间值：强迫结果为偶数-慢

例如：附加位为

01：舍

11：入

10：(强迫结果为偶数)

例：1.1101**11** → 1.1110; 1.1101**01** → 1.1101;
1.1101**10** → 1.1110; 1.1111**10** → 10.0000;

(2) 朝 $+\infty$ 方向舍入：舍入为Z2(正向舍入)

(3) 朝 $-\infty$ 方向舍入：舍入为Z1(负向舍入)

(4) 朝0方向舍入：截去。正数：取Z1；负数：取Z2

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    float a;
```

```
    double b;
```

```
    a = 123456.789e4;
```

```
    b = 123456.789e4;
```

```
    printf( "%f/n%f/n" ,a,b);
```

```
}
```

运行结果如下：

```
1234567936.000000
```

```
1234567890.000000
```

为什么float情况下输出的结果
会比原来的大？这到底有没有
根本性原因还是随机发生的？
为什么会出现这样的情况？

float可精确表示7个十
进制有效数位，后面的
数位是舍入后的结果，
舍入后的值可能会更大，
也可能更小

问题：为什么同一个实数赋值给float型变量
和double型变量，输出结果会有所不同呢？

C语言中的浮点数类型

- C语言中有**float**和**double**类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- **long double**类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是**80位扩展精度**格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

浮点数比较运算举例

- 对于以下给定的关系表达式，判断是否永真。

```
int x ;  
float f ;  
double d ;
```

Assume neither
d nor f is NaN

自己写程序
测试一下!

$x == (\text{int})(\text{float}) x$	否
$x == (\text{int})(\text{double}) x$	是
$f == (\text{float})(\text{double}) f$	是
$d == (\text{float}) d$	否
$f == -(-f);$	是
$2/3 == 2/3.0$	否
$d < 0.0 \Rightarrow ((d*2) < 0.0)$	是
$d > f \Rightarrow -f > -d$	是
$d * d \geq 0.0$	是
$x*x \geq 0$	否
$(d+f)-d == f$	否

IEEE 754 的范围和精度

- 单精度浮点数 (float型) 的表示范围多大?

最大的数据: $+1.11...1 \times 2^{127}$ 约 $+3.4 \times 10^{38}$

双精度浮点数 (double型) 呢? 约 $+1.8 \times 10^{308}$

- 以下关系表达式是否永真?

```
if ( i == (int) ((float) i) ) {    Not always true!
    printf ( "true" );      How about double?    True!
}
```

```
if ( f == (float) ((int) f) ) {    Not always true!
    printf ( "true" );      How about double?    False!
}
```

- 浮点数加法结合律是否正确? FALSE!

$x = -1.5 \times 10^{38}, \quad y = 1.5 \times 10^{38}, \quad z = 1.0$

$(x+y)+z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1.0$

$x+(y+z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = 0.0$

举例：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

举例：爱国者导弹定位错误

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数 x 来乘以计数值作为以秒为单位的时间
- 这个 x 的机器数是多少呢？
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]..., $x=0.0001100\ 1100\ 1100\ 1100\ 1100\ 1100\text{B}$ 。显然， x 是0.1的近似表示， $0.1-x$
 $= 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ [1100]... -$
 $0.000\ 1100\ 1100\ 1100\ 1100\ 1100\text{B}$ ，即为：
 $= 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ [1100]... \text{B}$
 $= 2^{-20} \times 0.1 \approx 9.54 \times 10^{-8}$ 这就是机器值与真值之间的误差！

举例：爱国者导弹定位错误

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少？

100小时相当于计数了 $100 \times 60 \times 60 \times 10 = 36 \times 10^5$ 次，因而导弹的时钟已经偏差了 $9.54 \times 10^{-8} \times 36 \times 10^5 \approx 0.343$ 秒

因此，距离误差是 $2000 \times 0.343 \text{秒} \approx 687 \text{米}$

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

举例：爱国者导弹定位错误

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
 - $0.1 = 0.0\ 0011[0011]B = +1.1\ 0011\ 0011\ 0011\ 0011\ 0011\ 00B \times 2^{-4}$ ，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
 - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为： $|x-0.1| = 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ [1100]\dots B$ 。这个值等于 $2^{-24} \times 0.1 \approx 5.96 \times 10^{-9}$ 。
100小时后时钟偏差 $5.96 \times 10^{-9} \times 36 \times 10^5 \approx 0.0215$ 秒。距离偏差 $0.0215 \times 2000 \approx 43$ 米。比爱国者导弹系统精确约16倍。

举例：爱国者导弹定位错误

- 若用32位二进制定点小数 $x=0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ B$ 表示0.1，则误差比用float表示误差更大还是更小？
 - 当 $x=0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ B$ 时，与0.1之间的误差约为： $|x-0.1|=0.000\ 0000\ 0000\ 0000\ 0000\ 00\ 1100\ [1100]...B$ 。这个值等于 $2^{-30} \times 0.1 \approx 9.31 \times 10^{-11}$ 。100小时后时钟偏差 $9.31 \times 10^{-11} \times 36 \times 10^5 \approx 0.000335$ 秒。预测的距离偏差仅为 $0.000335 \times 2000 \approx 0.67$ 米。

举例：浮点数运算的精度问题

- 从上述结果可以看出：
 - 用32位定点小数表示0.1，比采用float精度高64倍
 - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢
- Ariana 5火箭和爱国者导弹的例子带来的启示
 - ✓ 程序员应对底层机器级数据的表示和运算有深刻理解
 - ✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确
 - ✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

浮点数累加

对阶引入精度损失, 舍入误差

大数吃小数

```
jie@debian: ~/class/ch2/3
1 #include <stdio.h>
2 void main()
3
4 {
5     int i;
6     float tem, sum;
7     tem=0.1;
8     sum=0;
9     for( i=0; i<4000000; i++)
10         sum += tem;
11     printf("%f\n", sum);
12 }
~
~
<3/tt.c[+1] [c] unix utf-8 Ln 1, Col 1/12
```

```
jie@debian: ~/class/ch2/3
jie@debian:~/class/ch2/3$ ./tt
384524.781250
jie@debian:~/class/ch2/3$
```

Kahan累加算法

http://en.wikipedia.org/wiki/Kahan_summation_algorithm

```
jie@debian: ~/class/ch2/3
1  #include <stdio.h>
2
3  void main()
4  {
5      float tem = 0.1f;
6      float sum = tem;
7      float c = 0;
8      float y, t;
9      int i;
10     for( i=1;i<4000000;i++)
11     {
12         y = tem - c;
13         t = sum + y;
14         c = (t - sum) - y;
15         sum = t;
16     }
17     printf("%f\n", sum);
18 }
```



将每次累加因舍入造成的截断误差保存起来，再加入到下一次的累加中。

```
jie@debian: ~/class/ch2/3
jie@debian:~/class/ch2/3$ ./precision
400000.000000
jie@debian:~/class/ch2/3$
```

Kahan累加算法

- 算法

将每次累加因舍入造成的截断误差保存起来，再加入到下一次的累加中。



```
function KahanSum(input)
```

```
    var sum = 0.0
```

```
    var c = 0.0
```

```
    // A running compensation for lost low-order bits.
```

```
    for i = 1 to input.length do
```

```
        var y = input[i] - c
```

```
        // So far, so good: c is zero.
```

```
        var t = sum + y
```

```
        // Alas, sum is big, y small, so low-order digits of y
        // are lost.
```

```
        c = (t - sum) - y
```

```
        // (t - sum) cancels the high-order part of y;
        // subtracting y recovers negative (low part of y)
```

```
        sum = t
```

```
        // Algebraically, c should always be zero.
```

```
        Beware overly-aggressive optimizing compilers!
```

```
    next i
```

```
    // Next time around, the lost low part will be added
    // to y in a fresh attempt.
```

```
    return sum
```

数据的运算小结

- C语言中涉及的运算
 - 整数算术运算、浮点数算术运算
 - 按位、逻辑、移位、位扩展和位截断
- 整数的加、减运算
 - 计算机中的“算盘”：模运算系统（高位丢弃、用标志信息表示）
 - 带符号整数和无符号数的加、减都在同一个“算盘”中
 - 现实与计算机中的运算结果有差异（计算机是模运算系统）
- 整数的乘、除运算
 - 无符号整数：逻辑左移 k 位等于乘 2^k 、逻辑右移 k 位等于除 2^k
 - 带符号整数乘：算术左移 k 位等于乘 2^k
 - 带符号整数除： $(x \geq 0 ? x : x + 2^k - 1)$ 算术右移 k 位，等于 x 除以 2^k
- 浮点数运算
 - 加减：对阶/尾数加减/规格化/舍入(就近舍入到偶数)（大数吃小数）
 - 乘除：尾数相乘除，阶码相加减