



南京大學

NANJING UNIVERSITY

# 运算方法和运算部件

殷亚凤

智能软件与工程学院

苏州校区南雍楼东区225

yafeng@nju.edu.cn , <https://yafengnju.github.io/>



# 运算方法和运算部件

- 高级语言和机器指令中的运算
- 基本运算部件
- **定点数运算**
- 整数乘除运算
- 浮点数运算





# 定点数运算——补码加减运算

## 补码加减运算公式

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

## 补码加减运算要点和运算部件

- 加、减法运算统一采用加法来处理
- 符号位(最高有效位MSB)和数值位一起参与运算
- 直接用ALU实现两个数的加运算(模运算系统)

### 问题：模是多少？

- 运算结果的高位丢弃，保留低 $n$ 位，相当于对和数取模 $2^n$
- 实现减法的主要工作在于：求 $[-Y]_{\text{补}}$

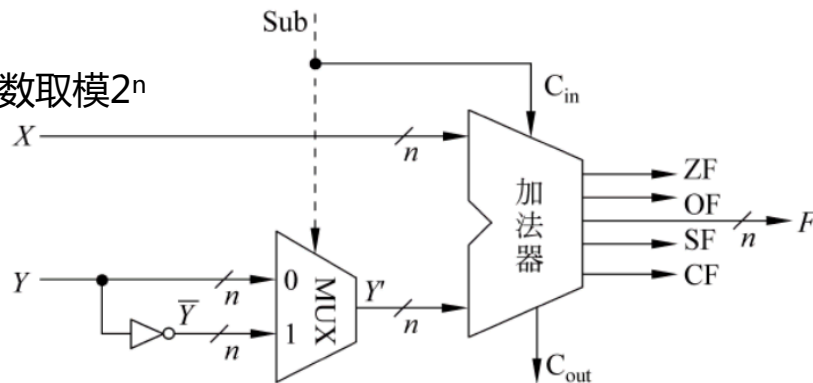
### 问题：如何求 $[-Y]_{\text{补}}$ ？

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

当控制端Sub为1时，做减法，实现 $A-B$ 当控制端Sub为0时，做加法，实现 $A+B$

问题：补码加减运算的用途是什么？

用于实现带符号整数的加减运算！



补码加减运算部件



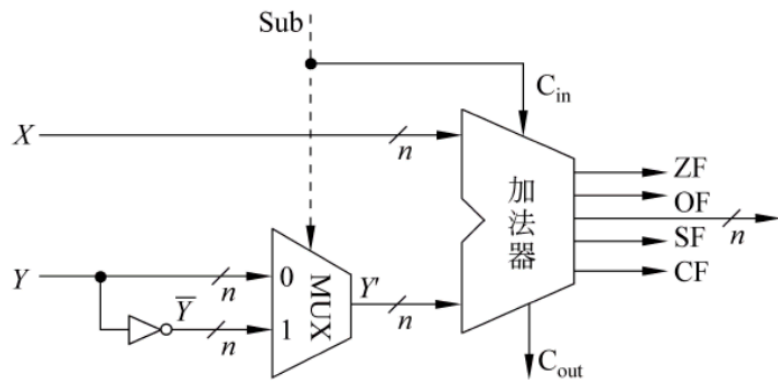


# 定点数运算——补码加减运算

- 利用带标志加法器，可构造整数加/减运算器，进行以下运算：

- 无符号整数加、无符号整数减
- 带符号整数加、带符号整数减

当Sub为1时，做减法  
当Sub为0时，做加法



- **ZF=1**，表示结果 $F=0$ ；
- **OF=1**，表示溢出（**仅指带符号运算**，  
 $OF = C_n \oplus C_{n-1}$ ， $OF = X_{n-1}Y_{n-1}\bar{F}_{n-1} + \bar{X}_{n-1}\bar{Y}_{n-1}F_{n-1}$ ）；
- **SF=1**，表示结果的符号/ $F$ 最高位（**仅指带符号运算**）；
- **CF=1**，表示进/借位（**仅指无符号运算**， $CF = Sub \oplus C_{out}$ ）



# 定点数运算——补码加减运算

例1:  $-7 - 6 = -7 + (-6) = +3$  **X**

1	0			
1	0	0	0	1
1	0	1	1	0
<div style="display: flex; justify-content: space-around;"> <span>0</span><span>0</span><span>1</span><span>1</span> </div>				

$-3 - 5 = -3 + (-5) = -8$  **✓**

1	1	1	1	
1	1	1	0	1
1	0	1	1	1
<div style="display: flex; justify-content: space-around;"> <span>1</span><span>0</span><span>0</span><span>0</span> </div>				

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

例2: 用8位补码求 107 和 46 的“和”，结果错误:  $107 + 46 = -103$ .

11	1	11		
107 <sub>10</sub>	=	0110	1011 <sub>2</sub>	
46 <sub>10</sub>	=	0010	1110 <sub>2</sub>	
<div style="display: flex; justify-content: space-around;"> <span>0</span><span>1</span><span>0</span><span>0</span><span>1</span><span>1</span><span>0</span><span>0</span><span>1</span> </div>				

溢出时，符号位的进位是真正的符号：+153

问题：若采用**变形补码**则结果怎样？有何好处？

结果的值为01 0011001，**左边第一位为真正的符号，数值部分进到了右边符号位上。**

**变形补码时的“溢出”判断条件：**结果的两个符号位不同。



## 定点数运算——原码加减运算（不要求）

- 用于浮点数尾数运算
- 符号位和数值部分分开处理
- 仅对数值部分进行加减运算，符号位起判断和控制作用
- 规则如下：
  - **比较两数符号，对加法实行“同号求和，异号求差”，对减法实行“异号求和，同号求差”。**
  - **求和**：数值位相加，若最高位产生进位，则结果溢出。和的符号取被加数（被减数）的符号。
  - **求差**：被加数（被减数）加上加数（减数）的补码。分二种情况讨论：
    - a) 最高数值位产生进位，表明加法结果为正，所得数值位正确。
    - b) 最高数值位没有产生进位，表明加法结果为负，得到的是数值位的补码形式，需对结果求补，还原为绝对值形式的数值位。
  - **差的符号位**：a) 情况下，符号位取被加数（被减数）的符号
    - b) 情况下，符号位为被加数（被减数）的符号取反





## 定点数运算——原码加减运算（不要求）

**例1：已知  $[X]_{\text{原}} = 1.0011$ ， $[Y]_{\text{原}} = 1.1010$ ，要求计算  $[X+Y]_{\text{原}}$**

解：根据原码加减运算规则，知：两数同号，用加法求和，和的符号同被加数符号。

- 和的数值位为： $0011 + 1010 = 1101$ （ALU中无符号数加）
- 和的符号位为：1
- $[X+Y]_{\text{原}} = 1.1101$

求和：直接加，有进位则溢出，符号同被

**例2：已知  $[X]_{\text{原}} = 1.0011$ ， $[Y]_{\text{原}} = 1.1010$ ，要求计算  $[X-Y]_{\text{原}}$**

解：根据原码加减运算规则，知：两数同号，用减法求差（补码减法）

- 差的数值位为： $0011 + (1010)_{\text{补}} = 0011 + 0110 = 1001$
- 最高数值位没有产生进位，表明加法结果为负，需对1001求补，还原为绝对值形式的数值位： $(1001)_{\text{补}} = 0111$
- 差的符号位为 $[X]_{\text{原}}$ 的符号位取反，即：0
- $[X-Y]_{\text{原}} = 0.0111$

求差：加补码，不会溢出，符号分情况



# 定点数运算——无符号数的乘法运算

假定： $[X]_{\text{原}} = x_0.x_1 \dots x_n$ ， $[Y]_{\text{原}} = y_0.y_1 \dots y_n$ ，求 $[X \times Y]_{\text{原}}$

数值部分  $z_1 \dots z_{2n} = (0.x_1 \dots x_n) \times (0.y_1 \dots y_n)$

(小数点位置约定，不区分小数还是整数)

## • 手算乘法例子:

被乘数	1000
乘数	1001
	<hr/>
	1000
	0000
	0000
	1000
	<hr/>
积	0.1001000

$$X \times Y = \sum_{i=1}^4 (X \times y_i \times 2^{-i})$$

$$X \times y_4 \times 2^{-4} \quad n=4$$

$$X \times y_3 \times 2^{-3}$$

$$X \times y_2 \times 2^{-2}$$

$$X \times y_1 \times 2^{-1}$$

整个运算过程中用到两种操作：加法 + 左移

因而，可用ALU和移位器来实现乘法运算





# 定点数运算——无符号数的乘法运算

- 手算乘法的特点：

- ① 每步计算： $X \times y_i$ ，若 $y_i = 0$ ，则得0；若 $y_i = 1$ ，则得 $X$
- ② 把①求得的各项结果 $X \times y_i$  逐次左移，可表示为 $X \times y_i \times 2^{-i}$
- ③ 对②中结果求和，即  $\sum (X \times y_i \times 2^{-i})$ ，这就是两个无符号数的乘积

- 计算机内部稍作以下改进：

- ① 每次得 $X \times y_i$ 后，与前面所得结果累加，得到 $P_i$ ，称之为部分积。因为不等到最后一次求和，减少了保存各次相乘结果 $X \times y_i$ 的开销。
- ② 每次得 $X \times y_i$ 后，不将它左移与前次部分积 $P_i$ 相加，而将部分积 $P_i$ 右移后与 $X \times y_i$ 相加。因为加法运算始终对部分积中高 $n$ 位进行。故用 $n$ 位加法器可实现二个 $n$ 位数相乘。
- ③ 对乘数中为“1”的位执行加法和右移，对为“0”的位只执行右移，而不执行加法运算。





# 定点数运算——无符号数的乘法运算

- 上述思想可写成如下**数学推导过程**：

$$\begin{aligned}
 X \times Y &= X \times (0.y_1 y_2 \dots y_n) \\
 &= X \times y_1 \times 2^{-1} + X \times y_2 \times 2^{-2} + \dots + X \times y_{n-1} \times 2^{-(n-1)} + X \times y_n \times 2^{-n} \\
 &= 2^{-n} \times X \times y_n + 2^{-(n-1)} \times X \times y_{n-1} + \dots + 2^{-2} \times X \times y_2 + 2^{-1} \times X \times y_1 \\
 &= 2^{-1} ( \underbrace{2^{-1} (2^{-1} \dots 2^{-1} (2^{-1} (0 + X \times y_n) + X \times y_{n-1}) + \dots + X \times y_2)}_{n \uparrow 2^{-1}} ) + X \times y_1
 \end{aligned}$$

- 上述推导过程**具有明显的递归性质**，因此，无符号数乘法过程可归结为循环计算下列算式的过程：

设 $P_0 = 0$ ，每步的乘积为：

$$P_1 = 2^{-1} (P_0 + X \times y_n)$$

$$P_2 = 2^{-1} (P_1 + X \times y_{n-1})$$

.....

$$P_n = 2^{-1} (P_{n-1} + X \times y_1)$$

- 其递推公式为： $P_{i+1} = 2^{-1} (P_i + X \times y_{n-i}) \quad (i = 0, 1, 2, 3, \dots, n-1)$
- 最终乘积 $P_n = X \times Y$

**迭代过程从乘数最低位 $y_n$ 和 $P_0=0$ 开始，经 $n$ 次“判断-加法-右移”循环，直到求出 $P_n$ 为止**





# 定点数运算——原码乘法运算

- 用于浮点数尾数乘运算
- 符号与数值分开处理：积符用两个符号异或得到，数值用无符号乘法运算

例：设 $[x]_{\text{原}}=0.1110$ ， $[y]_{\text{原}}=1.1101$ ，计算 $[X \times Y]_{\text{原}}$

解：数值部分用无符号数乘法算法计算： $1110 \times 1101 = 1011\ 0110$ ，  
符号位： $0 \oplus 1 = 1$ ，所以： $[X \times Y]_{\text{原}} = 1.10110110$

左侧是**原码一位乘法**：每次只取乘数中的一位进行判断，需 $n$ 次循环，速度相对较慢。

**原码两位乘法**的思想：对乘数的每两位取值进行判断，每步求出对应两位的部分积。

## ◆ 原码两位乘法的操作的递推公式：

00 —  $P_{i+1} = 2^{-2} P_i$   
 01 —  $P_{i+1} = 2^{-2} (P_i + X)$   
 10 —  $P_{i+1} = 2^{-2} (P_i + 2X)$   
 11 —  $P_{i+1} = 2^{-2} (P_i + 3X) = 2^{-2} (P_i + 4X - X)$   
 $= 2^{-2} (P_i - X) + X$

3X时，本次-X，下次+X！

采用两位一乘，运算速度提高多少？

**运算次数减少一半，速度提高一倍！**

$y_{i-1}$ $y_i$ T	操 作	迭代公式
0 0 0	$0 \rightarrow T$	$2^{-2} (P_i)$
0 0 1	$+X \quad 0 \rightarrow T$	$2^{-2} (P_i + X)$
0 1 0	$+X \quad 0 \rightarrow T$	$2^{-2} (P_i + X)$
0 1 1	$+2X \quad 0 \rightarrow T$	$2^{-2} (P_i + 2X)$
1 0 0	$+2X \quad 0 \rightarrow T$	$2^{-2} (P_i + 2X)$
1 0 1	$-X \quad 1 \rightarrow T$	$2^{-2} (P_i - X)$
1 1 0	$-X \quad 1 \rightarrow T$	$2^{-2} (P_i - X)$
1 1 1	$1 \rightarrow T$	$2^{-2} (P_i)$

T触发器用来记录下次是否要执行“+X”，其中“-X”运算用“+ $[-X]_{\text{补}}$ ”实现！



# 定点数运算——原码乘法运算

已知  $[X]_{\text{原}} = 0.111001$  ,  $[Y]_{\text{原}} = 0.100111$  , 用原码两位乘法计算  $[X \times Y]_{\text{原}}$

解：先采用无符号数乘法计算  $111001 \times 100111$  的乘积，原码两位乘法过程如下：

采用补码右移

数据为模8补码形式（三位符号位），为什么？

若采用模4补码，则进行P和Y同时右移2位操作时，按照补码右移规则，得到的P3是负数，显然，两个正数相乘，乘积不可能是负数



P		Y	T	说明
000	000000	100111	0	开始, $P_0=0$ , $T=0$
+111	000111			$y_5y_6T = 110$ , $-X$ , $T=1$
111	000111			P 和 Y 同时右移 2 位
111	110001	11	1001	1 得 $P_1$
+001	110010			$y_3y_4T = 011$ , $+2X$ , $T=0$
001	100011			P 和 Y 同时右移 2 位
000	011000	1111	10	0 得 $P_2$
+001	110010			$y_1y_2T = 100$ , $+2X$ , $T=0$
010	001010			P 和 Y 同时右移 2 位
000	100010	101111	0	得 $P_3$

加上符号位，得  $[X \times Y]_{\text{原}} = 0.100010101111$

速度快，但代价也大



# 定点数运算——补码乘法运算

- 用于对什么类型的数据计算？已知什么？求什么？
- 带符号整数！如C语句：`int x=-5, y=-4, z=x*y;`

**问题：已知 $[x]_{\text{补}}$ 和 $[y]_{\text{补}}$ ，求 $[x*y]_{\text{补}}$**

- 因为 $[x*y]_{\text{补}} \neq [x]_{\text{补}} * [y]_{\text{补}}$ ，故不能直接用无符号整数乘法计算。
- 例如，若 $x=-5$ ，求 $x*x=?$ ： $[-5]_{\text{补}}=1011$
- $[x*x]_{\text{补}} : [25]_{\text{补}}=0001\ 1001$
- $[x]_{\text{补}} * [x]_{\text{补}} ; [-5]_{\text{补}} * [-5]_{\text{补}}=1111\ 1001$

**思路：根据 $[y]_{\text{补}}$ 求 $y$ ，将 $y$ 的计算公式代入 $[x*y]_{\text{补}}$**

令： $[y]_{\text{补}} = y_{n-1}y_{n-2}\cdots y_1y_0$

则： $y = -y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0$

**因为 $[A+B]_{\text{补}} = [A]_{\text{补}} + [B]_{\text{补}}$ ，只要将 $[x*y]_{\text{补}}$ 转换为对若干数的和求补即可**





# 定点数运算——补码乘法运算

布斯 (Booth) 乘法推导如下：

假定： $[X]_{\text{补}} = x_{n-1}x_{n-2}\cdots x_1x_0$ ， $[Y]_{\text{补}} = y_{n-1}y_{n-2}\cdots y_1y_0$ ，求： $[X \times Y]_{\text{补}} = ?$

基于补码定义： $Y = -y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0$

令： $y_{-1} = 0$ ，则：

当 $n=32$ 时， $Y = -y_{31} \cdot 2^{31} + y_{30} \cdot 2^{30} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0 + y_{-1} \cdot 2^0$

$$\downarrow$$

$$\underline{-y_{31} \cdot 2^{31} + (y_{30} \cdot 2^{31} - y_{30} \cdot 2^{30})} + \cdots + (y_0 \cdot 2^1 - y_0 \cdot 2^0) + \underline{y_{-1} \cdot 2^0}$$

$$\downarrow$$

$$(y_{30} - y_{31}) \cdot 2^{31} + (y_{29} - y_{30}) \cdot 2^{30} + \cdots + (y_0 - y_1) \cdot 2^1 + (y_{-1} - y_0) \cdot 2^0$$

$$\begin{aligned} 2^{-32} \cdot [X \times Y]_{\text{补}} &= (y_{30} - y_{31})X \cdot 2^{-1} + (y_{29} - y_{30})X \cdot 2^{-2} + \cdots + (y_0 - y_1)X \cdot 2^{-31} + (y_{-1} - y_0)X \cdot 2^{-32} \\ &= 2^{-1}(2^{-1} \dots (2^{-1}(y_{-1} - y_0)X) + (y_0 - y_1)X) + \dots + (y_{30} - y_{31})X \end{aligned}$$

部分积公式： $P_i = 2^{-1}(P_{i-1} + (y_{i-1} - y_i)X)$

符号与数值统一处理





# 定点数运算——补码乘法运算

布斯 ( Booth ) 算法实质：



- | 当前位 | 右边位 | 操作       | 举例                  |
|-----|-----|----------|---------------------|
| 1   | 0   | 减被乘数     | 000111 <u>1</u> 000 |
| 1   | 1   | 加0 (不操作) | 000111 <u>1</u> 000 |
| 0   | 1   | 加被乘数     | 00 <u>0</u> 1111000 |
| 0   | 0   | 加0 (不操作) | 0 <u>0</u> 01111000 |
- 在 “**1串**” 中，第一个1时做减法，最后一个1做加法，其余情况只要移位。
  - 最初提出这种想法是因为在Booth的机器上移位操作比加法更快！

同前面算法一样，将乘积寄存器右移一位。（这里是算术右移）





# 定点数运算——补码乘法运算

布斯 ( Booth ) 算法举例：

已知 $[X]_{\text{补}} = 1\ 101$ ， $[Y]_{\text{补}} = 0\ 110$ ，计算 $[X \times Y]_{\text{补}}$   $[-X]_{\text{补}} = 0011$   
 $X = -3$ ， $Y = 6$ ， $X \times Y = -18$ ， $[X \times Y]_{\text{补}}$ 应等于11101110或结果溢出

P	Y	$y_{-1}$	说明
0000	0110 0		设 $y_{-1} = 0$ ， $[P_0]_{\text{补}} = 0$
		→ 1	$y_0 y_{-1} = 00$ ，P、Y 直接右移一位
0000	0011 0		得 $[P_1]_{\text{补}}$
+0011			$y_1 y_0 = 10$ ， $+[-X]_{\text{补}}$
0011		→ 1	P、Y 同时右移一位
0001	1001 1		得 $[P_2]_{\text{补}}$
		→ 1	$y_2 y_1 = 11$ ，P、Y 直接右移一位
0000	1100 1		得 $[P_3]_{\text{补}}$
+1101			$y_3 y_2 = 01$ ， $+ [X]_{\text{补}}$
1101		→ 1	P、Y 同时右移一位
1110	1110 0		得 $[P_4]_{\text{补}}$

如何判断结果是否溢出？

高4位是否全为符号位！

验证：当 $X \times Y$ 取8位时，结果 -0010010B = -18；  
 取4位时，结果溢出







# 定点数运算——补码两位乘法

## 补码两位乘可用布斯算法推导如下：

$$\begin{aligned}
 - [P_{i+1}]_{\text{补}} &= 2^{-1} ( [P_i]_{\text{补}} + (y_{i-1} - y_i) [X]_{\text{补}} ) \\
 - [P_{i+2}]_{\text{补}} &= 2^{-1} ( [P_{i+1}]_{\text{补}} + (y_i - y_{i+1}) [X]_{\text{补}} ) \\
 &= 2^{-1} ( 2^{-1} ( [P_i]_{\text{补}} + (y_{i-1} - y_i) [X]_{\text{补}} ) + (y_i - y_{i+1}) [X]_{\text{补}} ) \\
 &= 2^{-2} ( [P_i]_{\text{补}} + (y_{i-1} + y_i - 2y_{i+1}) [X]_{\text{补}} )
 \end{aligned}$$

- 开始置附加位 $y_{-1}$ 为0，乘积寄存器最高位前面添加一位附加符号位0。
- 最终的乘积高位部分在乘积寄存器P中，低位部分在乘数寄存器Y中。
- 因为字长总是8的倍数，所以补码的位数 $n$ 应该是偶数，因此总循环次数为 $n/2$ 。

$y_{i+1}$	$y_i$	$y_{i-1}$	操 作	迭 代 公 式
0	0	0	0	$2^{-2}[P_i]_{\text{补}}$
0	0	1	$+ [X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [X]_{\text{补}}\}$
0	1	0	$+ [X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [X]_{\text{补}}\}$
0	1	1	$+ 2[X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + 2[X]_{\text{补}}\}$
1	0	0	$+ 2[-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + 2[-X]_{\text{补}}\}$
1	0	1	$+ [-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [-X]_{\text{补}}\}$
1	1	0	$+ [-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [-X]_{\text{补}}\}$
1	1	1	0	$2^{-2}[P_i]_{\text{补}}$



# 定点数运算——补码两位乘法

- 已知  $[X]_{\text{补}} = 1\ 101$  ,  $[Y]_{\text{补}} = 0\ 110$  , 用补码两位乘法计算  $[X \times Y]_{\text{补}}$  。
- 解：  $[-X]_{\text{补}} = 0\ 011$  , 用补码二位乘法计算  $[X \times Y]_{\text{补}}$  的过程如下。

$P_n$	$P$	$Y$	$y_{-1}$	说明
0	0000	0110	0	开始, 设 $y_{-1} = 0$ , $[P_0]_{\text{补}} = 0$
+ 0	0110			$y_1y_0y_{-1} = 100$ , $+2[-X]_{\text{补}}$
0	0110		→ 2	P和Y同时右移二位
0	0001	1001	1	得 $[P_2]_{\text{补}}$
+ 1	1010			$y_3y_2y_1 = 011$ , $+2[X]_{\text{补}}$
1	1011		→ 2	P和Y同时右移二位
1	1110	1110		得 $[P_4]_{\text{补}}$

因此  $[X \times Y]_{\text{补}} = 1110\ 1110$  , 与一位补码乘法 (布斯乘法) 所得结果相同, 但循环次数减少了一半。

验证：  $-3 \times 6 = -18$  ( -10010B )



# 定点数运算——快速乘法器

- 前面介绍的乘法部件的特点

- 通过一个ALU多次做“加/减+右移”来实现
  - 一位乘法：约 $n$ 次“加+右移”
  - 两位乘法：约 $n/2$ 次“加+右移”

所需时间随位数增多而加长，由时钟和控制电路控制

- 设计快速乘法部件的必要性

- 大约1/3是乘法运算

- 快速乘法器的实现（由特定功能的组合逻辑单元构成）

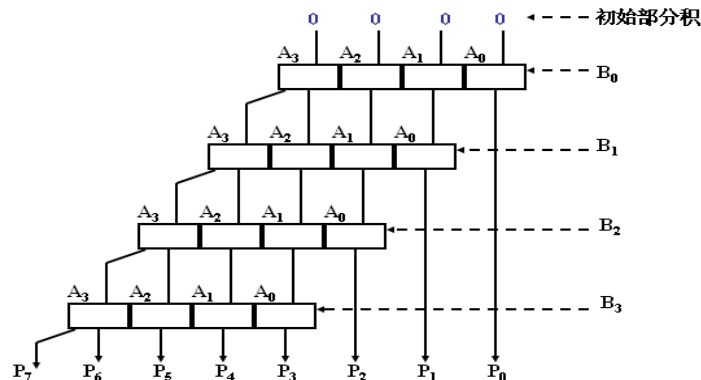
- 流水线方式
- 硬件叠加方式（如：阵列乘法器）





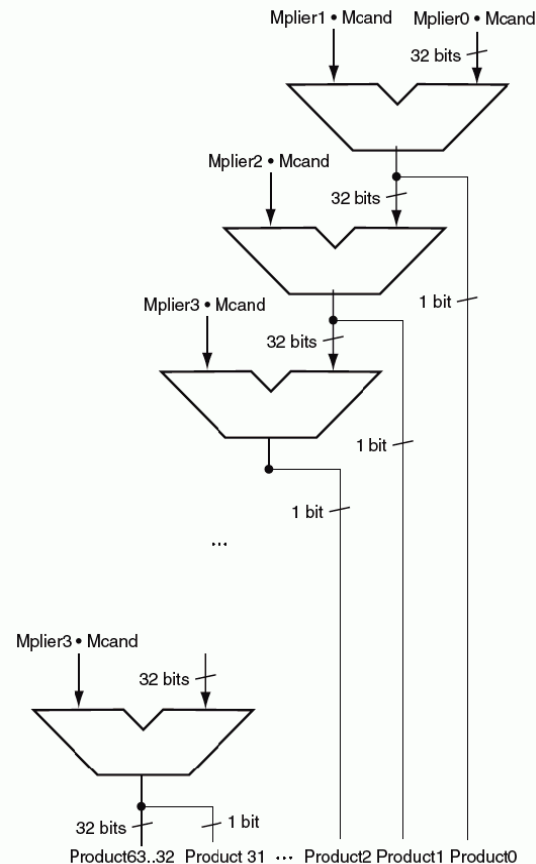
# 定点数运算——快速乘法器

## 流水线方式的快速乘法器



**组合逻辑电路！  
无需控制器控制**

- 为乘数的每位提供一个n位加法器
- 每个加法器的两个输入端分别是：
  - 本次乘数对应的位与被乘数相与的结果（即：0或被乘数）
  - 上次部分积
- 每个加法器的输出分为两部分：
  - 和的最低有效位(LSB)作为本位乘积
  - 进位和高31位的和数组成一个32位数作为本次部分积

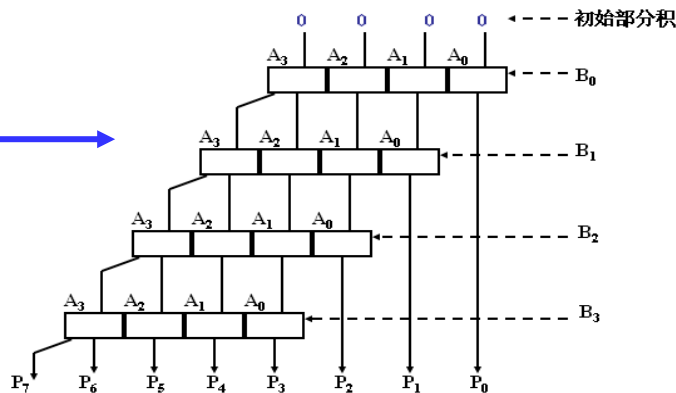
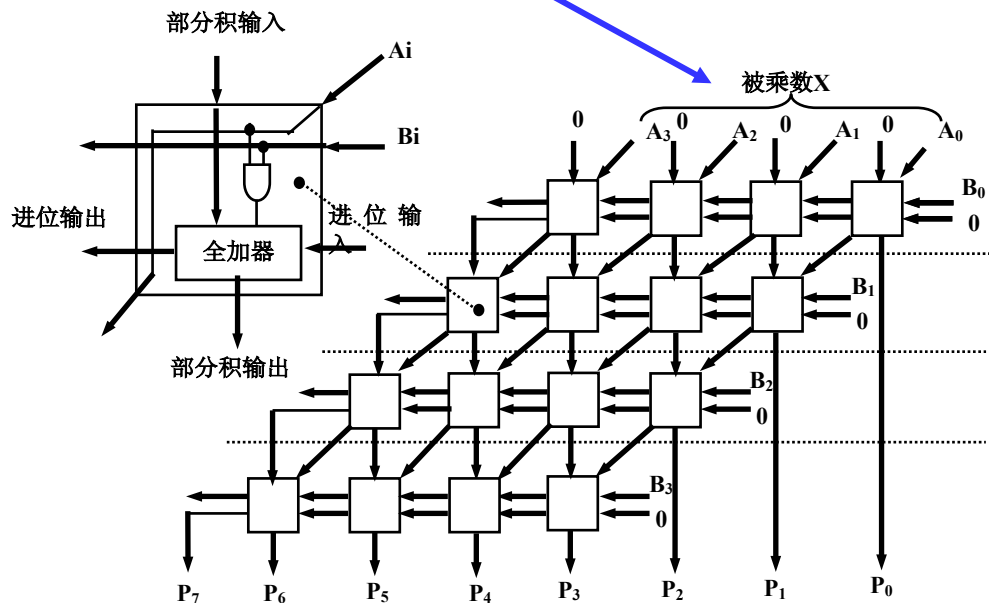




# 定点数运算——快速乘法器

## 阵列乘法器的实现

- 手算乘法过程
- 阵列乘法器



速度仅取决于逻辑门和  
加法器的传输延迟

无符号阵列乘法器

增加符号处理电路、乘前及乘后求  
补电路，即可实现带符号数乘法器。



# 定点数运算——除法运算

## 手算除法：

$$\begin{array}{r} \text{除数 } 1000 \overline{) 1001010} \\ \underline{-1000} \\ 10 \\ \underline{101} \\ 1010 \\ \underline{-1000} \\ 10 \end{array}$$

商 1001  
被除数 1001010  
中间余数 10, 101, 1010  
余数 10

### ◆ 手算除法的基本要点

- ① **被除数与除数相减**，够减则上商为1；不够减则上商为0。
- ② 每次得到的**差为中间余数**，将除数右移后与上次的中间余数比较。用中间余数减除数，够减则上商为1；不够减则上商为0。
- ③ 重复执行第②步，直到求得的**商的位数足够为止**。





# 定点数运算——定点除法运算

## • 除前预处理

- ①若被除数=0且除数 $\neq 0$ ，或定点整数除法时 $|被除数| < |除数|$ ，则商为0，不再继续
- ②若被除数 $\neq 0$ 、除数=0，则发生“除数为0”异常（浮点数时为 $\infty$ ）  
（若浮点除法被除数和除数都为0，则有些机器产生一个不发信号的NaN，即“quiet NaN”）
- ③只有当被除数和除数都 $\neq 0$ ，且商 $\neq 0$ 时，才进一步进行除法运算。

## • 计算机内部无符号数除法运算

- 与手算一样，通过被除数（中间余数）减除数来得到每一位商
  - 够减上商1；不够减上商0（从msb $\rightarrow$ lsb得到各位商）
- 基本操作为减（加）法和移位，故可与乘法合用同一套硬件

两个n位数相除的情况：

- (1)**定点正整数（即无符号数）相除**：在被除数的高位添n个0
- (2)**定点正小数（即原码小数）相除**：在被除数的低位添加n个0

这样，就将所有情况都统一为：一个 $2n$ 位数除以一个 $n$ 位数





# 定点数运算——定点除法运算

问题：第一次试商为1，说明什么？

商有 $n+1$ 位数，因而溢出！

若是 $2n$ 位除以 $n$ 位的无符号整数运算，则说明将会得到多于 $n$ 位的商，因而结果“溢出”（即：无法用 $n$ 位表示商）。

例： $1111\ 1111/1111 = 1\ 0001$

若是两个 $n$ 位数相除，则第一位商为0，且肯定不会溢出，为什么？

最大商为： $0000\ 1111/0001=1111$

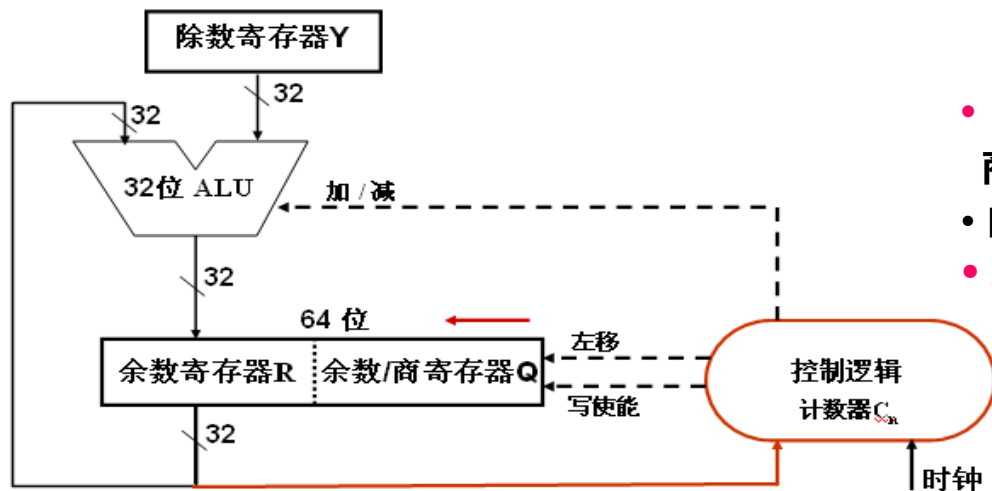
若是浮点数中尾数原码小数运算，第一次试商为1，则说明尾数部分有“溢出”，可通过浮点数的“右规”消除“溢出”。所以，在浮点数运算器中，第一次得到的商“1”要保留。

例： $0.11110000/0.1000=+1.1110$





# 定点数运算——定点除法运算



- R和Q同步“左移”，Q空出位上“商”，商的各位逐次左移到Q中。
- 由控制逻辑根据加减结果决定商为0还是1
- 减-----试商，加-----恢复余数。

- **除数寄存器Y**：存放除数。
- **余数寄存器R**：初始时高位部分为高32位被除数；结束时是余数。
- **余数/商寄存器Q**：初始时为低32位被除数；结束时是32位商。
- **循环次数计数器C<sub>n</sub>**：存放循环次数。初值是32，每循环一次，C<sub>n</sub>减1，当C<sub>n</sub>=0时，除法运算结束。
- **ALU**：除法核心部件。在控制逻辑控制下，对于寄存器R和Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器R。



# 定点数运算——定点除法运算

验证：7 / 2 = 3 余 1

-D = 1110

R：被除数（中间余数）；

D：除数

这里是两个n位无符号数相除，肯定不会溢出，故余数先左移而省略判断溢出过程。

	D: 0010	R: 0000 0111
Shl R	D: 0010	R: 0000 1110
R = R-D	D: 0010	R: 1110 1110
+D, sl R, 0	D: 0010	R: 0001 1100
R = R-D	D: 0010	R: 1111 1100
+D, sl R, 0	D: 0010	R: 0011 1000
R = R-D	D: 0010	R: 0001 1000
sl R, 1	D: 0010	R: 0011 0001
R = R-D	D: 0010	R: 0001 0001
sl R, 1	D: 0010	R: 0010 0011
Shr R(rh)	D: 0010	R: 0001 0011

从例子可看出：

每次上商为0时，需做加法以“恢复余数”。所以，称为“恢复余数法”。

开始余数先左移了一位，故最后余数需向右移一位



## 定点数运算——定点除法运算

**不恢复余数除法（加减交替法）**：在下一步运算时把当前多减的除数补回来

根据恢复余数法(设 $B$ 为除数， $R_i$ 为第 $i$ 次中间余数)，有：

- 若 $R_i < 0$ ，则商上“0”，并做加法恢复余数，即：  
$$R_{i+1} = 2(R_i + 2^n|B|) - 2^n|B| = 2R_i + 2^n|B| \quad (\text{“负，0，加”})$$
- 若 $R_i \geq 0$ ，则商上“1”，不需恢复余数，即：  
$$R_{i+1} = 2R_i - 2^n|B| \quad (\text{“正，1，减”})$$

省去了恢复余数的过程

注意：最后一次上商为“0”的话，需要“纠余”处理，即把试商时被减掉的除数加回去，恢复真正的余数。

不恢复余数法也称为加减交替法





# 定点数运算——带符号数除法

- **原码除法**

- **商符和商值分开处理**

- 商的数值部分由无符号数除法求得
    - 商符由被除数和除数的符号确定：同号为0，异号为1

- **余数的符号同被除数的符号**

- **补码除法**

- 方法1：**同原码除法一样**，先转换为正数，先用无符号数除法，然后修正商和余数。
  - 方法2：**直接用补码除法**，符号和数值一起进行运算，商符直接在运算中产生。

**若是两个 $n$ 位补码整数除法运算，则被除数进行符号扩展。**

若被除数为 $2n$ 位，除数为 $n$ 位，则被除数无需扩展。





# 定点数运算——原码除法举例

已知  $[X]_{\text{原}} = 0.1011$ ,  $[Y]_{\text{原}} = 1.1101$

用恢复余数法计算  $[X/Y]_{\text{原}}$  ?

解：商的符号位： $0 \oplus 1 = 1$

减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。

$$[|X|]_{\text{补}} = 0.1011$$

$$[|Y|]_{\text{补}} = 0.1101$$

$$[-|Y|]_{\text{补}} = 1.0011$$

小数在低位扩展0

思考：若实现无符号数相除，即1011除以1101，则有何不同？  
结果是什么？

被除数高位补0，1011除以1101，  
结果等于0

余数寄存器 R	余数/商寄存器 Q	说 明
0 1 0 1 1	0 0 0 0 □	开始 $R_0 = X$
+ 1 0 0 1 1		$R_1 = X - Y$
1 1 1 1 0	0 0 0 0 0	$R_1 < 0$ , 则 $q_4 = 0$
+ 0 1 1 0 1		恢复余数: $R_1 = R_1 + Y$
0 1 0 1 1		得 $R_1$
1 0 1 1 0	0 0 0 0 □	$2R_1$ (R 和 Q 同时左移, 空出一位商)
+ 1 0 0 1 1		$R_2 = 2R_1 - Y$
0 1 0 0 1	0 0 0 0 1	$R_2 > 0$ , 则 $q_3 = 1$
1 0 0 1 0	0 0 0 1 □	$2R_2$ (R 和 Q 同时左移, 空出一位商)
+ 1 0 0 1 1		$R_3 = 2R_2 - Y$
0 0 1 0 1	0 0 0 1 1	$R_3 > 0$ , 则 $q_2 = 1$
0 1 0 1 0	0 0 1 1 □	$2R_3$ (R 和 Q 同时左移, 空出一位商)
+ 1 0 0 1 1		$R_4 = 2R_3 - Y$
1 1 1 0 1	0 0 1 1 0	$R_4 < 0$ , 则 $q_1 = 0$
+ 0 1 1 0 1		恢复余数: $R_4 = R_4 + Y$
0 1 0 1 0	0 0 1 1 0	得 $R_4$
1 0 1 0 0	0 1 1 0 □	$2R_4$ (R 和 Q 同时左移, 空出一位商)
+ 1 0 0 1 1		$R_5 = 2R_4 - Y$
0 0 1 1 1	0 1 1 0 1	$R_5 > 0$ , 则 $q_0 = 1$

商的最高位为0，说明没有溢出，商的数值部分为：1101。

所以， $[X/Y]_{\text{原}} = 1.1101$  (最高位为符号位)，余数为  $0.0111 \times 2^4$ 。

用于  
判断  
是否  
溢出

若求  $[Y/X]_{\text{原}}$   
结果如何？



# 定点数运算——原码除法举例

已知  $[X]_{\text{原}} = 0.1011$ ,  $[Y]_{\text{原}} = 1.1101$

用**加减交替法**计算 $[X/Y]_{\text{原}}$

解:  $[|X|]_{\text{补}} = 0.1011$

$[|Y|]_{\text{补}} = 0.1101$

$[-|Y|]_{\text{补}} = 1.0011$

加减交替法的要点:

负、0、加

正、1、减

得到的结果与恢复余数法一样!

用被除数 (中间余数) 减除数试商时, 怎样确定是否“够减”?

中间余数的符号! (正数-正数)

余数寄存器 R	余数/商寄存器 Q	说 明
01011	0000□	开始 $R_0 = X$
+10011		$R_1 = X - Y$
<b>1</b> 1110	0000 <b>0</b>	$R_1 < 0$ , 则 $q_4 = 0$ , 没有溢出
11100	0000□	$2R_1$ (R 和 Q 同时左移, 空出一位商)
<b>+01101</b>		$R_2 = 2R_1 + Y$
01001	00001	$R_2 > 0$ , 则 $q_3 = 1$
10010	0001□	$2R_2$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_3 = 2R_2 - Y$
<b>0</b> 0101	0001 <b>1</b>	$R_3 > 0$ , 则 $q_2 = 1$
01010	0011□	$2R_3$ (R 和 Q 同时左移, 空出一位商)
<b>+10011</b>		$R_4 = 2R_3 - Y$
11101	00110	$R_4 < 0$ , 则 $q_1 = 0$
11010	0110□	$2R_4$ (R 和 Q 同时左移, 空出一位商)
+01101		$R_5 = 2R_4 + Y$
00111	01101	$R_5 > 0$ , 则 $q_0 = 1$

补码除法能否这样来判断呢?

不能, 因为符号可能不同!





# 定点数运算——补码除法运算

## • 补码除法判断是否“够减”的规则

- (1) 当被除数（或中间余数）与除数**同号**时，做**减法**，若新余数的符号与被除数符号**一致**（**正(负)-正(负)=正(负)**）表示**够减**，否则为不够减
- (2) 当被除数（或中间余数）与除数**异号**时，做**加法**，若新余数的符号与被除数符号**一致**表示**够减**（**正(负)+负(正)=正(负)**），否则为不够减

**总结：余数变号不够减，不变号够减**

上述判断规则归纳如下：

中间余数R 的符号	除数Y的符 号	同号：新中间余数= R-Y（同号为正商）		异号：新中间余数= R+Y（异号为负商）	
		0	1	0	1
0	0	够减	不够减		
0	1			够减	不够减
1	0			不够减	够减
1	1	不够减	够减		





# 定点数运算——补码恢复余数法

- 两个n位带符号整数相除算法要点：

(1) 操作数的预置：

除数装入除数寄存器Y，**被除数经符号扩展**后装入余数寄存器R和余数/商寄存器Q

(2) R和Q同步串行左移一位。

(3) 若R与Y同号，则 $R = R - Y$ ；否则 $R = R + Y$ ，并按以下规则确定商值 $q_0$ ：

① 若中间余数 $R=0$ 或R操作前后符号未变，表示够减，则 $q_0$ 置1，转下一步；

② 若操作前后R的符号已变，表示不够减，则 $q_0$ 置0，恢复R值后转下一步；

(4) 重复第(2)和第(3)步，直到取得n位商为止。

(5) 若被除数与除数同号，则Q中就是真正的商；否则，将Q求补后是真正的商。

(即：若商应为负数时，则需要“各位取反，末位加1”来得到真正的商)

(6) 余数在R中。

**问题：如何恢复余数？通过“做加法”来恢复吗？**

**无符号数（或原码）除法通过“做加法”恢复余数，但补码不是！**

**补码：若原来是  $R = R - Y$ ，则执行  $R = R + Y$  来恢复余数；**

**若原来是  $R = R + Y$ ，则执行  $R = R - Y$  来恢复余数。**







# 定点数运算——补码恢复余数法

举例：7/3=?

被除数：0000 0111 除数 0011  
M=0011

A	Q	
0000	0111	
← 0000	1110	
+ 1101		减 (同号)
1101	1110	变号
+ 0011		恢复(加)商0
0000	1110	
← 0001	1100	
+ 1101		减
1110	1100	变号
+ 0011		恢复(加)商0
0001	1100	
← 0011	1000	
+ 1101		减
0000	1000	符同, 商1
← 0001	0001	
+ 1101		减
1110	0001	变号
+ 0011		恢复(加)商0
0001	0010	

余:0001/商:0010 验证：7/3 = 2，余数为1

(-7)/3=?

被除数：1111 1001 除数 0011  
M=0011

A	Q	
1111	1001	
← 1111	0010	
+ 0011		加 (异号)
0010	0010	变号
+ 1101		恢复(减)商0
1111	0010	
← 1110	0100	
+ 0011		加
0001	0100	变号
+ 1101		恢复(减)商0
1110	0100	
← 1100	1000	
+ 0011		加
1111	1000	符同, 商1
← 1111	0001	
+ 0011		加
0010	0001	变号
+ 1101		恢复(减)商0
1111	0010	

余:1111/商:1110

验证：-7/3 = -2，  
余数为-1

商应为负数，需求补：0010→1110



南京大学  
NANJING UNIVERSITY



# 定点数运算——补码不恢复余数法

## • 算法要点：

### (1) 操作数的预置：

除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q。

### (2) 根据以下规则求第一位商 $q_n$ ：

若被除数X与Y同号，则 $R1 = X - Y$ ；否则 $R1 = X + Y$ ，并按以下规则确定商值 $q_n$ ：

① 若新的中间余数 $R1$ 与Y同号，则 $q_n$ 置1，转下一步；

② 若新的中间余数 $R1$ 与Y异号，则 $q_n$ 置0，转下一步；

$q_n$ 用来判断是否溢出，而不是真正的商。以下情况下会发生溢出：

若X与Y同号且上商 $q_n = 1$ ，或者，若X与Y异号且上商 $q_n = 0$ 。

### (3) 对于 $i = 1$ 到 $n$ ，按以下规则求出 $n$ 位商：

① 若 $R_i$ 与Y同号，则 $q_{n-i}$ 置1， $R_{i+1} = 2R_i - [Y]_{\text{补}}$ ， $i = i + 1$ ；

② 若 $R_i$ 与Y异号，则 $q_{n-i}$ 置0， $R_{i+1} = 2R_i + [Y]_{\text{补}}$ ， $i = i + 1$ ；

(4) 商的修正：最后一次Q寄存器左移一位，将最高位 $q_n$ 移出，最低位置上商 $q_0$ 。若被除数与除数同号，Q中就是真正的商；否则，将Q中商的末位加1。

(5) 余数的修正：若余数符号同被除数符号，则不需修正，余数在R中；否则，按下列规则进行修正：当被除数和除数符号相同时，最后余数加除数；否则，最后余数减除数。

判断是否同号与恢复余数法不同，不是新老余数之间！而是余数和除数之间！

补码不恢复余数法也有一个六字口诀“同、1、减；异、0、加”。其运算过程也呈加/减交替方式，因此也称为“加减交替法”。

商已经是“反码”





# 定点数运算——补码不恢复余数法

## 举例：-9/2

将 $X=-9$ 和 $Y=2$ 分别表示成5位补码形式为：

$[X]_{\text{补}} = 1\ 0111$

$[Y]_{\text{补}} = 0\ 0010$

被除数符号扩展为：

$[X]_{\text{补}} = 11111\ 10111$

$[-Y]_{\text{补}} = 1\ 1110$

同、1、减  
异、0、加

$X/Y = -0100B = -4$

余数为  $-0001B = -1$

将各数代入公式：

“除数 $\times$ 商+余数=被除数”进行验证，得

$2 \times (-4) + (-1) = -9$

余数寄存器 R	余数/商寄存器 Q	说 明
11111	10111	开始 $R_0 = [X]_{\text{补}}$
+00010		$R_1 = [X]_{\text{补}} + [Y]_{\text{补}}$
00001	10111	$R_1$ 与 $[Y]_{\text{补}}$ 同号，则 $q_5 = 1$
00011	01111	$2R_1$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_2 = 2R_1 + [-Y]_{\text{补}}$
00001	01111	$R_2$ 与 $[Y]_{\text{补}}$ 同号，则 $q_4 = 1$
00010	11111	$2R_2$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_3 = 2R_2 + [-Y]_{\text{补}}$
00000	11111	$R_3$ 与 $[Y]_{\text{补}}$ 同号，则 $q_3 = 1$
00001	11111	$2R_3$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_4 = 2R_3 + [-Y]_{\text{补}}$
11111	11111	$R_4$ 与 $[Y]_{\text{补}}$ 异号，则 $q_2 = 0$
11111	11110	$2R_4$ (R 和 Q 同时左移，空出一位上商 0)
+00010		$R_5 = 2R_4 + [Y]_{\text{补}}$
00001	11110	$R_5$ 与 $[Y]_{\text{补}}$ 同号，则 $q_1 = 1$
00011	11101	$2R_5$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_6 = 2R_5 + [-Y]_{\text{补}}$
00001	11011	$R_6$ 与 $[Y]_{\text{补}}$ 同号，则 $q_0 = 1$ ，Q 左移，空出一位上商 1
+11110	+	商为负数，末位加 1；减除数以修正余数
11111	11100	

所以， $[X/Y]_{\text{补}} = 11100$ 。余数为 11111。



# 运算方法和运算部件

- 高级语言和机器指令中的运算
- 基本运算部件
- 定点数运算
- **整数乘除运算**
- 浮点数运算

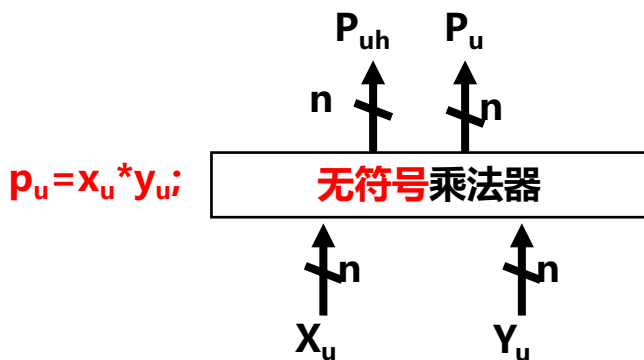




# 整数乘除运算——整数的乘运算

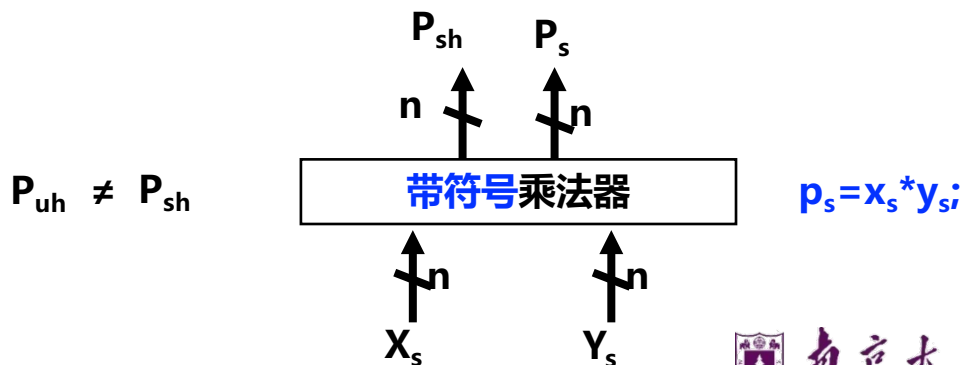
- 假定**两个n位无符号**整数 $x_u$ 和 $y_u$ 对应的机器数为 $X_u$ 和 $Y_u$ ， $p_u = x_u \times y_u$ ， $p_u$ 为**n位无符号整数**且对应的机器数为 $P_u$ ；
- 两个n位带符号**整数 $x_s$ 和 $y_s$ 对应的机器数为 $X_s$ 和 $Y_s$ ， $p_s = x_s \times y_s$ ， $p_s$ 为**n位带符号整数**且对应的机器数为 $P_s$ 。
- 若 $X_u = X_s$ 且 $Y_u = Y_s$ ，则 $P_u = P_s$ 。

✓ 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。



✓ **无符号**：若 $P_{uh} = 0$ ，则不溢出

✓ **带符号**：若 $P_{sh}$ 每位都等于 $P_s$ 的最高位，则不溢出





# 整数乘除运算——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	0000 0110	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	1111 1000	-8	1000	不溢出





# 整数乘除运算——整数的乘运算

在计算机内部，一定有 $x^2 \geq 0$ 吗？

若 $x$ 是**带符号整数**，则不一定！

如 $x$ 是浮点数，则一定！

例如，当  $n=4$  时,  $5^2 = -7 < 0$ ！

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

若 $x$ 、 $y$ 和 $z$ 都改成unsigned类型，  
则判断方式为

乘积的高 $n$ 位为全0，则不溢出

	0101	
×	0101	
<hr/>		
	0101	
+	0101	
<hr/>		
	00011001	结果溢出

只取低4位，值为-111B=-7

- 高级语言程序如何判断 $z$ 是正确值？

当  $!x \parallel z/x == y$  为真时

- 编译器如何判断？

当  $-2^{n-1} \leq x*y < 2^{n-1}$ （不溢出）时

即：乘积的高 $n$ 位为全0或全1，并等于低 $n$ 位的最高位！

即：乘积的高 $n+1$ 位为全0或全1



# 整数乘除运算——常量的乘除运算

## 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**：移出的低位直接丢弃
  - 带符号负整数**：**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:  $1111\ 0010 + 0000\ 0011 = 1111\ 0101$

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

$-9/2: 10111 + 00001 = 11000$

$11000 \gg 1 = 11100 = -4$







## 整数乘除运算——常量的乘除运算

- 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:  **$(x >= 0 ? x : (x + 31)) >> 5$**

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

**$x >> 31$  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;
}
```





# 运算方法和运算部件

- 高级语言和机器指令中的运算
- 基本运算部件
- 定点数运算
- 整数乘除运算
- 浮点数运算





# 浮点数运算——浮点数加减运算

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 :

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

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

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

上述运算结果可能出现以下几种情况：

**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  $\Rightarrow +0/-0$

**尾数溢出**：最高有效位有进位  $\Rightarrow$  右规

**非规格化尾数**：数值部分高位为0  $\Rightarrow$  左规

**右规或对阶时**，右段有效位丢失  $\Rightarrow$  尾数舍入

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。





# 浮点数运算——浮点数加减运算

## ① 无效运算（无意义）

- 运算时有一个数是非有限数，如：加 / 减 $\infty$ 、 $0 \times \infty$ 、 $\infty/\infty$ 等
- 结果无效，如：源操作数是NaN、 $0/0$ 、 $x \text{ REM } 0$ 、 $\infty \text{ REM } y$  等

## ② 除以0（即：无穷大）

③ 数太大（阶码上溢）：对于单精度浮点数，阶码  $E > 1111\ 1110$ （阶大于127）

④ 数太小（阶码下溢）：对于单精度浮点数，**阶码  $E < 0000\ 0001$ （阶小于-126-23）**

⑤ 结果不精确（舍入时引起），例如 $1/3$ ， $1/10$ 等不能精确表示成浮点数

上述情况硬件可以捕捉到，因此这些异常可设定让硬件处理，也可设定让软件处理。让硬件处理时，称为硬件陷阱。

**注：硬件陷阱：事先设定好是否要进行硬件处理（即挖一个陷阱），当出现相应异常时，就由硬件自动进行相应的异常处理（掉入陷阱）。**



# 浮点数运算——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





# 浮点数运算——浮点数加减运算

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

例：4位移码， $Ex=7$ ， $Ey=-7$ ，则 $[\Delta E]_{\text{补}}=1111+1111=1110$ ， $\Delta E < 0$ , 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（即小数被大数吃掉）？ 24！**

$1.xx...x \rightarrow 0.00...01xx...x$  (右移24位后，尾数变为0)

**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

$$\begin{aligned} [\Delta E]_{\text{补}} &= 256 + Ex - Ey = 256 + 127 + Ex - (127 + Ey) \\ &= 256 + [Ex]_{\text{移}} - [Ey]_{\text{移}} = [Ex]_{\text{移}} + [-[Ey]_{\text{移}}]_{\text{补}} \pmod{256} \end{aligned}$$

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。



# 浮点数运算——浮点数加减运算

( 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 $\Delta e$ 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**：

$0.00...0001x2^{-126}$

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 )。





## 浮点数运算——浮点数加减运算

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。







# 浮点数运算——浮点数加减运算

在计算机内部执行上述运算时，必须解决哪些问题？

(1) 如何表示？ **用IEEE 754标准！**

(2) 如何判断阶码的大小？求 $[\Delta E]_{\text{补}}$  = ？

(3) 对阶后**尾数的隐含位**如何处理？ **右移到数值部分，高位补0，保留移出低位部分**

(4) 如何进行**尾数加减**？ **隐藏位还原后，按原码进行加减运算，附加位一起运算**

(5) 何时需要规格化，如何**规格化**？  **$\pm 1x.xx.....x$  形式时，则右规：尾数右移1位，阶码加1  
 $\pm 0.0...01x...x$  形式时，则左规：尾数左移k位，阶码减k**

(6) 如何**舍入**？

**最终须把附加位去掉，此时需考虑舍入（IEEE 754有四种舍入方式）**

(7) 如何判断**溢出**？

**若最终阶码为全1，则上溢；若尾数为全0，则下溢**





# 浮点数运算——浮点数加减运算

已知 $x=0.5$ ,  $y=-0.4375$ , 求 $x+y=?$  (用IEEE 754标准单精度格式计算)

解:  $x=0.5=1/2=(0.100...0)_2=(1.00...0)_2 \times 2^{-1}$

$y=-0.4325=(-0.01110...0)_2=(-1.110..0)_2 \times 2^{-2}$

$[x]_{\text{浮}}=0\ 01111110,00...0$      $[y]_{\text{浮}}=1\ 01111101,110...0$

**对阶:**  $[\Delta E]_{\text{补}}=0111\ 1110 + 1000\ 0011=0000\ 0001$ ,  $\Delta E=1$

故对 $y$ 进行对阶:  $[y]_{\text{浮}}=1\ 0111\ 1110\ 1110...0$ (高位补隐藏位)

**尾数相加:**  $01.0000...0 + (10.1110...0) = 00.00100...0$

(原码加法, 最左边一位为符号位, 符号位分开处理)

**左规:**  $+(0.00100...0)_2 \times 2^{-1} = +(1.00...0)_2 \times 2^{-4}$

(阶码减3, 实际上是加了三次11111111)

$[x+y]_{\text{浮}}=0\ 0111\ 1011\ 00...0$

$x+y=(1.0)_2 \times 2^{-4}=1/16=0.0625$

**问题: 尾数加法器最多需要多少位?**

**1+1+23+3=28位**





# 浮点数运算——浮点运算的精度和舍入

在浮点数运算中，保留多少附加位才能保证运算精度？ **无法给出准确答案！**

**保留附加位可以得到比不保留附加位更高的精度。**

- IEEE754规定: 中间结果须在右边至少加2个附加位 ( guard & round )
  - **Guard bit(保护位/警戒位)**：在浮点数尾数右边的位
  - **Rounding bit(舍入位)**：在保护位右边的位
  - **Sticky(粘位)**: 舍入位右边任何非0数字，粘位置1，否则置0

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左规时被移到尾数中；② 作为舍入的依据。





# 浮点数运算——浮点运算的精度和舍入

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

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





# 浮点数运算——浮点运算的精度和舍入

以下情况下，可能会导致阶码溢出

## – 左规（阶码 - 1）时

- 左规（- 1）时：先判断阶码是否为全0，若是，则直接置阶码下溢；否则，阶码减1后判断阶码是否为全0，若是，则阶码下溢。

## – 右规（阶码 + 1）时

- 右规（+ 1）时，先判断阶码是否为全1，若是，则直接置阶码上溢；否则，阶码加1后判断阶码是否为全1，若是，则阶码上溢。

问题：机器内部如何减1？

$$+[-1]_{\text{补}} = + 11 \dots 1$$





# 浮点数运算——浮点运算的精度和舍入

以下情况下，可能会导致阶码溢出（续）

- 乘法运算求阶码的和时

- 若 $E_x$ 和 $E_y$ 最高位皆1，而 $E_b$ 最高位是0或 $E_b$ 为全1，则阶码上溢
- 若 $E_x$ 和 $E_y$ 最高位皆0，而 $E_b$ 最高位是1或 $E_b$ 为全0，则阶码下溢

- 除法运算求阶码的差时

- 若 $E_x$ 的最高位是1， $E_y$ 的最高位是0， $E_b$ 的最高位是0或 $E_b$ 为全1，则阶码上溢。
- 若 $E_x$ 的最高位是0， $E_y$ 的最高位是1， $E_b$ 的最高位是1或 $E_b$ 为全0，则阶码下溢。

例：若 $E_b = 0000\ 0001$ ，则左规一次后，结果的阶码  $E_b = ?$

解： $E_b = E_b + [-1]_{\text{补}} = 0000\ 0001 + 1111\ 1111 = 0000\ 0000$  阶码下溢！

例：若 $E_x = 1111\ 1110$ ， $E_y = 1000\ 0000$ ，则乘法运算时结果的阶码  $E_b = ?$

解： $E_b = E_x + E_y + 129 = 1111\ 1110 + 1000\ 0000 + 1000\ 0001 = 1111\ 1111$   
阶码上溢！





# 浮点数运算——浮点数乘/除运算

- 浮点数乘法： $A*B = (M_a * M_b) \cdot 2^{E_a+E_b}$
- 浮点数除法： $A/B = (M_a / M_b) \cdot 2^{E_a-E_b}$

浮点数**尾数采用原码乘/除**运算

## 浮点数乘 / 除法步骤

(  $X_m$ 、 $Y_m$ 分别是X和Y尾数原码，  $X_e$ 和 $Y_e$  分别是X和Y阶移码 )

(1) **求阶**： $X_e \pm Y_e \bar{+} 127$

(2) **尾数相乘除**： $X_m * / Y_m$  ( 两个形为1.xxx的数相乘/除 )

(3) 两数符号相同，结果为正；两数符号相异，结果为负；

(4) **当尾数高位为0，需左规；当尾数最高位有进位，需右规。**

(5) 如果尾数比规定的长，则需考虑舍入。

(6) 若尾数是0，则需要将阶码也置0。

(7) 阶码溢出判断

问题1：**乘法**运算结果最多左规几次？最多右规几次？

**不需左规！最多右规1次！**

问题2：**除法**呢？

**左规次数不定！不需右规！**



# 浮点数运算——浮点数乘/除运算

## • 求阶码的和与差

设 $E_x$ 和 $E_y$ 分别是两个操作数的阶码， $E_b$ 是结果的阶码，则：

- 阶码加法公式为： $E_b \leftarrow E_x + E_y + 129 \pmod{2^8}$

$$\begin{aligned}[E_1 + E_2]_{\text{移}} &= 127 + E_1 + E_2 = 127 + E_1 + 127 + E_2 - 127 \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} - 127 \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} + [-127]_{\text{补}} \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} + 10000001\text{B} \pmod{2^8}\end{aligned}$$

- 阶码减法公式为： $E_b \leftarrow E_x + [-E_y]_{\text{补}} + 127 \pmod{2^8}$

$$\begin{aligned}[E_1 - E_2]_{\text{移}} &= 127 + E_1 - E_2 = 127 + E_1 - (127 + E_2) + 127 \\ &= [E_1]_{\text{移}} - [E_2]_{\text{移}} + 127 \\ &= [E_1]_{\text{移}} + [-[E_2]_{\text{移}}]_{\text{补}} + 01111111\text{B} \pmod{2^8}\end{aligned}$$







## 浮点数运算——浮点数乘/除运算

设 $E_x$ 和 $E_y$ 分别是两个操作数阶码的移码， $E_b$ 是结果阶码的移码表示

例：若两个阶码分别为10和-5，求 $10+(-5)$ 和 $10-(-5)$ 的移码。

解： $E_x = 127 + 10 = 137 = 1000\ 1001B$

$$E_y = 127 + (-5) = 122 = 0111\ 1010B$$

$$[-E_y]_{\text{补}} = 1000\ 0110B$$

将 $E_x$ 和 $E_y$ 代入上述公式，得：

$$\begin{aligned} E_b &= E_x + E_y + 129 = 1000\ 1001 + 0111\ 1010 + 1000\ 0001 \\ &= 1000\ 0100B = 132 \pmod{2^8} \end{aligned}$$

其阶码的和为 $132 - 127 = 5$ ，正好等于 $10 + (-5) = 5$ 。

$$\begin{aligned} E_b &= E_x + [-E_y]_{\text{补}} + 127 = 1000\ 1001 + 1000\ 0110 + 0111\ 1111 \\ &= 1000\ 1110B = 142 \pmod{2^8} \end{aligned}$$

其阶码的差为 $142 - 127 = 15$ ，正好等于 $10 - (-5) = 15$ 。





# 课程习题（作业）——截止日期：10月20日晚23:59

- **课本88-90页**：第3、4、5、7、11、12题
- 提交方式：<https://selearning.nju.edu.cn/>（教学支持系统）



第3章-运算方法和运算部件-课后习题

课本88-90页：第3、4、5、7、11、12题

- 命名：学号+姓名+第\*章。
- 若提交遇到问题请及时发邮件或在下一次上课时反馈。





# 课程习题（作业）——截止日期：10月20日晚23:59

3. 考虑以下 C 语言程序代码：

```
int func1(unsigned word)
{
    return (int) ((word<<24)>>24);
}
int func2(unsigned word)
{
    return ((int) word<<24)>>24;
}
```

假设在一个 32 位机器上执行这些函数,该机器使用二进制补码表示带符号整数。无符号数采用逻辑移位,带符号整数采用算术移位。请填写下表,并说明函数 func1 和 func2 的功能。

$w$		$\text{func1}(w)$		$\text{func2}(w)$	
机器数	值	机器数	值	机器数	值
	127				
	128				
	255				
	256				



# 课程习题（作业）——截止日期：10月20日晚23:59

4. 填写下表,注意对比无符号整数和带符号整数(用补码表示)的乘法结果,包括截断操作前后的结果。

模 式	$x$		$y$		$x \times y$ (截断前)		$x \times y$ (截断后)	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号整数	110		010					
带符号整数	110		010					
无符号整数	001		111					
带符号整数	001		111					
无符号整数	111		111					
带符号整数	111		111					





# 课程习题（作业）——截止日期：10月20日晚23:59

5. 以下是两段 C 语言代码,函数 arith()是直接用 C 语言写的,而 optarith()是对 arith()函数以某个确定的 M 和 N 编译生成的机器代码反编译生成的。根据 optarith()推断函数 arith() 中 M 和 N 的值各是多少?

```
#define M
#define N
int arith(int x, int y)
{
    int result=0;
    result=x * M+y/N;
    return result;
}

int optarith(int x, int y)
{
    int t=x;
    x<<=4;
    x-=t;
    if(y<0) y+=3;
    y>>=2;
    return x+y;
}
```





## 课程习题（作业）——截止日期：10月20日晚23:59

7. 已知  $x=10, y=-6$ , 采用 6 位机器数表示。请按如下要求计算, 并把结果还原成真值。

- (1) 求  $[x+y]_{\text{补}}, [x-y]_{\text{补}}$ 。
- (2) 用原码一位乘法计算  $[x \times y]_{\text{原}}$ 。
- (3) 用 MBA(基 4 布斯算法) 计算  $[x \times y]_{\text{补}}$ 。
- (4) 用不恢复余数法计算  $[x/y]_{\text{原}}$  的商和余数。
- (5) 用不恢复余数法计算  $[x/y]_{\text{补}}$  的商和余数。





# 课程习题（作业）——截止日期：10月20日晚23:59

11. 假设浮点数格式为：阶码是 4 位移码，偏置常数为 8，尾数是 6 位补码（采用双符号位），用浮点运算规则分别计算以下表达式在不采用任何附加位和采用 2 位附加位（保护位、舍入位）两种情况下的值（假定采用就近舍入到偶数方式）。

$$(1) (15/16) \times 2^7 + (2/16) \times 2^5$$

$$(2) (15/16) \times 2^7 - (2/16) \times 2^5$$

$$(3) (15/16) \times 2^5 + (2/16) \times 2^7$$

$$(4) (15/16) \times 2^5 - (2/16) \times 2^7$$

12. 采用 IEEE 754 单精度浮点数格式计算下列表达式的值。

$$(1) 0.75 + (-65.25)$$

$$(2) 0.75 - (-65.25)$$





# 课程实验——截止日期：11月3日晚23:59

- 提交方式：<https://selearning.nju.edu.cn/>（教学支持系统）



- 命名：学号+姓名+实验\*。
- 提交：文件打包，提交ZIP压缩文件。**







# 课程实验——截止日期：11月3日晚23:59

## 实验三 类型转换和移位操作运算

**实验目的:** 了解高级语言中数据类型的转换和移位操作结果，从而能更好地理解指令系统设计和计算机硬件设计所需满足的要求和需要考虑的问题。

**实验要求:** 编程实现以下各种操作:

- (1) 给定一个 short 型数据 -12345，分别转换为 int、unsigned short、unsigned int、float 类型的数据;
- (2) 给定一个 int 型数据 2147483647，分别转换为 short、unsigned short、unsigned int、float 类型的数据;
- (3) 给定一个 float 型数据 123456.789e5，转换成 double 型数据;
- (4) 给定一个 double 型数据 123456.789e5，转换成 float 型数据;
- (5) 按 short 和 unsigned short 类型分别对 -12345 进行左移 2 位和右移 2 位操作。要求分别用十进制和十六进制形式打印输出以上各种操作的结果。

**实验报告:**

1. 给出源程序(文本文件)和执行结果。
2. 根据实验结果，回答下列问题。

(1) 无符号数和带符号整数的扩展操作方式是否相同?各是如何进行的? (2) 补码整数(如 int 型数)是否总能转换为等值的 float 类型数据?为什么? (3) float 型数据是否总能转换成等值的 double 型数据?为什么? (4) 长数被截断成短数后可能发生什么现象?为什么? (5) C 语言中移位操作规则与操作对象的数据类型有关吗? (6) 左移 2 位和右移 2 位操作分别相当于扩大和缩小几倍?



# 课程实验——截止日期：11月3日晚23:59

## 实验四 整数和浮点数的算术运算

**实验目的:** 通过检查高级语言中数据运算的不同结果，进一步理解机器代码在 CPU 中的执行过程，从而为更好地学习指令系统设计和 CPU 设计打下良好的基础。

**实验要求:** 编程计算下列表达式的值:

- (1) unsigned int 型数据:  $1+4294967295=?; 1-4294967295=?$
- (2) int 型数据:  $2147483647+1=?; -2147483648-1=?$
- (3) float 型数据:  $(1.0 + 123456.789e30) + (-123456.789e30) = ?;$   
 $1.0 + (123456.789e30 + (-123456.789e30)) = ?$

要求分别用十进制和十六进制形式显示各种操作的结果。

### 实验报告:

- 1. 给出源程序(文本文件)和执行结果。
- 2. 分别给出每个运算结果的解释说明





## 其他情况说明

### • 课程习题：第2章第10题（课本50页）

10. 设某浮点数格式为

数符	阶码	尾数
1位	5位移码	<u>6位补码数值部分</u>

可按**补码**计算，  
也可按**原码**计算  
(需说明采用哪种方式)

其中，移码的偏置常数为 16，补码采用一位符号位，基数为 4。

(1) 用这种格式表示下列十进制数： $+1.75$ ， $+19$ ， $-1/8$ 。

(2) 写出该格式浮点数的表示范围，并与 12 位定点补码整数和定点补码小数表示范围比较。





# 其他情况说明

## 教学支持系统

### 课程

#### ▼ 2023 Fall

- ▶ 本科生一年级
- ▶ 本科生二年级
- ▶ 本科生三年级
- ▶ 本科生四年级
- ▶ 研究生一年级
- ▶ 智能软件与工程学院

搜索课程



计算机组织结构

教师: 殷亚凤

## 教学内容讨论-本课程与《计算机系统基础》异同



第1章-计算机系统概述 (请投票)



第2章-数据的机器级表示 (请投票)



第3章-运算方法和运算部件 (请投票)

## 第1章-计算机系统概述 (请投票)



查看0个投票

请各位同学对本课程“第1章：计算机系统概述”教学内容与《计算机系统基础》中涉及的教学内容相似程度进行投票，谢谢大家！

此活动的结果将不会在您投票后公布。



☐ 0%(完全不同) ☐ 10% ☐ 20% ☐ 30% ☐ 40% ☐ 50% ☐ 60% ☐ 70% ☐ 80% ☐ 90% ☐ 100%(完全相同)



南京大学  
NANJING UNIVERSITY



# 提问

## Q & A

殷亚凤

智能软件与工程学院

苏州校区南雍楼东区225

yafeng@nju.edu.cn , <https://yafengnju.github.io/>



南京大學  
NANJING UNIVERSITY