Ch3: Arithmetic and Logic Operate and ALU

第三章 运算方法和运算部件

一、定点乘法的运算方法和实现

- 串行算法：
  - 原码一位乘法、原码两位乘法
  - Booth算法、补码两位乘法
- 并行算法：
  - 原码阵列乘法器、补码阵列乘法器
  - 直接补码乘法器

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：符号异或得到，数值为数值部分的积。

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

符号：$z_1 = x_0 \oplus y_0$

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)

定点数中小数点的位置只是一个**约定**，无论是定点小数还是定点整数计算中没有小数点，因此相乘时数值部分的积可以看成两个**无符号数的积**。

一、原码乘法

1. 运算方法

如：设 x =0. 1101，y =0. 1011.

人工算法：

```
        0. 1 1 0 1  (x)
 ×      0. 1 0 1 1  (y)
 ---------------------
            1 1 0 1
          1 1 0 1
        0 0 0 0
 +    1 1 0 1
 ---------------------
 0. 1 0 0 0 1 1 1 1  (z)
```

$$
\begin{array}{rcccccccccc}
 & & & & a_{m-1} & a_{m-2} & \cdots & a_1 & a_0 & = A \\
\times) & & & & & b_{n-1} & \cdots & b_1 & b_0 & = B \\
\hline
 & & & & a_{m-1}b_0 & a_{m-2}b_0 & \cdots & a_1b_0 & a_0b_0 & \\
 & & & a_{m-1}b_1 & a_{m-2}b_1 & \cdots & a_1b_1 & a_0b_1 & & \\
 & & \cdot & & & & \cdot & & & \\
+) & a_{m-1}b_{n-1} & a_{m-2}b_{n-1} & \cdots & a_1b_{n-1} & a_0b_{n-1} & & & & \\
\hline
p_{m+n-1} & p_{m+n-2} & p_{m+n-3} & \cdots & p_{n-1} & \cdots & & p_1 & p_0 & = P
\end{array}
$$

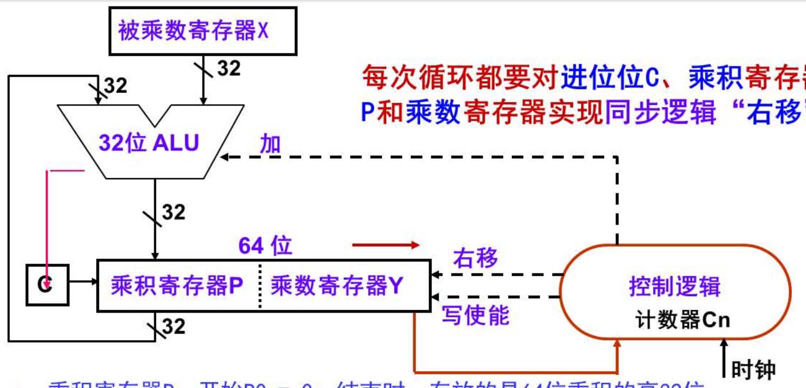
整个运算过程中用到两种操作：加法 + 左移

因而，可用ALU和移位器来实现乘法运算

一、原码乘法运算

- **手工乘法的特点：**
  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，表示为$X\times y_i\times 2^{-i}$
  ③ 对②中结果求和，即 $\sum (X\times y_i\times 2^{-i})$即为两数的乘积
- **计算机稍作改进：**
  ① 每次得$X\times y_i$后，与前面的结果累加得到P_i，称之为部分积，减少了保存各次相乘结果的开销。
  ② 每次得$X\times y_i$后，不是左移与P_i相加，而将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”的位加并右移，对为“0”的位只右移，不加。

32位乘法运算的硬件实现



每次循环都要对进位位C、乘积寄存器P和乘数寄存器实现同步逻辑“右移”

- 乘积寄存器P：开始P0 = 0；结束时，存放的是64位乘积的高32位
- 乘数寄存器Y：开始时置乘数；结束时，存放的是64位乘积的低32位
- ALU：对P和X相加，运算结果送回P，进位位在C中
- 进位触发器C：保存加法器的进位信号
- 循环次数计数器Cn：存放循环次数。初值32，每次减1，Cn=0时结束
- ALU：乘法核心部件。控制逻辑控制下，对P和X 加”，在“写使能”控制下运算结果被送回P，进位位在C中

乘法运算在硬件中的执行

例: **x, y, z**都是无符号数，**x=1110, y=1101**。计算**z=x*y**。

递推： $P_i=2^{-1}(x*y_i+ P_{i-1})$

	C	乘积P	乘数Y	
P初始为0	0	0000	110$\boxed{1}$	
ALU加	+	1110		X*1
	0	1110	1101	→
→	0	0111	0110	
→	0	0011	1011	
	+	1110		
	1	0001	1011	
→	0	1000	1101	
	+	1110		
	1	0110	1101	
→	0	1011	0110	

- 双倍字长的乘积寄存器；或两个单倍字长的寄存器。
- 部分积初始为0。
- 右移时进位、部分积和剩余乘数一起进行逻辑右移。

验证：x=14, y=13, z=x*y=182

当z取4位时，结果发生溢出，因为高4位不为全0！

原码乘法算法

正负数相乘例：$[x]_原=0.1110$，$[y]_原=1.1101$，计算 $[x\times y]_原$

解：数值：1110×1101= 1011 0110
　　符号位：0⊕1=1，所以：$[x\times y]_原$=1. 10110110

原码一位乘：每次只取乘数的一位，需n次循环，速度慢。
原码两位乘：每次取乘数两位判断，需n/2次循环快一倍。

◆原码两位乘：

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

y_{i-1} y_i T	操　作	迭 代 公 式
0　0　0	$0\to T$	$2^{-2} (P_i)$
0　0　1	+X　$0\to T$	$2^{-2} (P_i + X)$
0　1　0	+X　$0\to T$	$2^{-2} (P_i + X)$
0　1　1	+2X　$0\to T$	$2^{-2} (P_i + 2X)$
1　0　0	+2X　$0\to T$	$2^{-2} (P_i + 2X)$
1　0　1	–X　$1\to T$	$2^{-2} (P_i - X)$
1　1　0	–X　$1\to T$	$2^{-2} (P_i - X)$
1　1　1	$1\to T$	$2^{-2} (P_i)$

11时先 -X，右移两位后再+X，相当于4X。-X用“+[-X]补”实现。

实际操作中，常用Yi-1、Yi、T三位来控制T触发器用来记录下次是否要执行“+X”“–X”运算用“$+[-X]_补$”实现！

原码两位乘法举例（略）

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

解： 先用无符号数乘法计算111001×100111，原码两位乘法过程如下：

$[|X|]_补=000\ 111001$，$[-|X|]_补=111\ 000111$

为模8补码形式（三位符号位）

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 $[X\times Y]_原=0.100010101111$

补码乘法运算

机器带符号整数都用补码表示，需要实现补码乘法运算。

带符号数可以使用补码直接相乘吗？

假定：若x=6，y=-7,求x*y=?

$[X]_补=x_{n-1}x_{n-2}\cdots x_1x_0$

$[Y]_补=y_{n-1}y_{n-2}\cdots y_1y_0$

求：$[X\times Y]_补=?$

验证发现 $[X\times Y]_补\neq[X]_补\times[Y]_补$，故不能直接用无符号数乘法计算。

补码乘法之一：校正法（略）

补码乘法之二：Booth算法

补码乘法运算——Booth算法

- Booth's Algorithm：

end of run　　middle of run　　beginning of run

0	1	1	1	1	0

当前位	右边位	操作	Example
1	0	减被乘数	0001111000
1	1	加0（不操作）	0001111000
0	1	加被乘数	0001111000
0	0	加0（不操作）	0001111000

- 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）

布斯算法举例

已知X = -3，Y = 6，计算 $[X\times Y]_补$

$[X]_补=1\ 101$

$[Y]_补=0\ 110$

$[-X]_补=0011$

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

验证：当X×Y取8位时，结果 - 0010010B = -18；为4位时，结果溢出！

补码两位乘法（略）

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

补码两位乘法举例

- 已知 $[X]_{补}$ = 1 101， $[Y]_{补}$ = 0 110，计算 $[X\times Y]_{补}$
- 解：$[-X]_{补}$= 0 011

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

因此 $[X \times Y]$补=1110 1110

验证：-3 ×6=-18 （-10010B）

快速乘法器

- 串行乘法部件的特点
  - 通过一个ALU多次做“加+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
  - 乘法运算耗时多
  - 比例大，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）

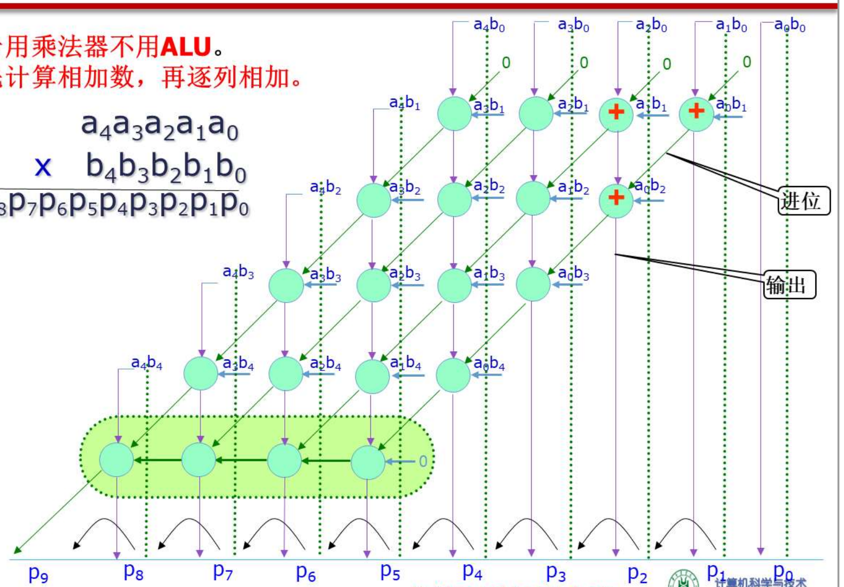
  如：阵列乘法器

◆阵列乘法器是原码乘去掉符号位，即为无符号数乘法

阵列乘法器

- 专用乘法器不用**ALU**。
- 先计算相加数，再逐列相加。

$$\begin{array}{r} a_4a_3a_2a_1a_0 \\ \times \quad b_4b_3b_2b_1b_0 \\ \hline p_9p_8p_7p_6p_5p_4p_3p_2p_1p_0 \end{array}$$



5位阵列乘法器电路

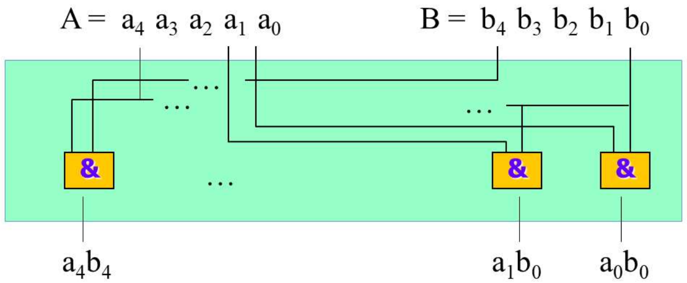
一位乘法逻辑实现

- 1 × 1 = 1
- 1 × 0 = 0
- 0 × 1 = 0
- 0 × 0 = 0

一个与门即可实现一位乘法

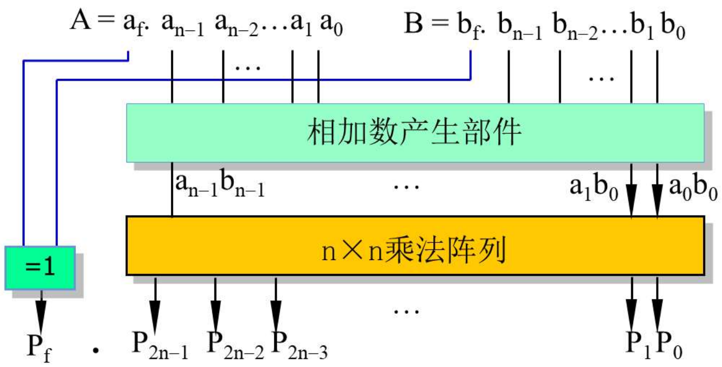
R=X*Y

相加数产生部件

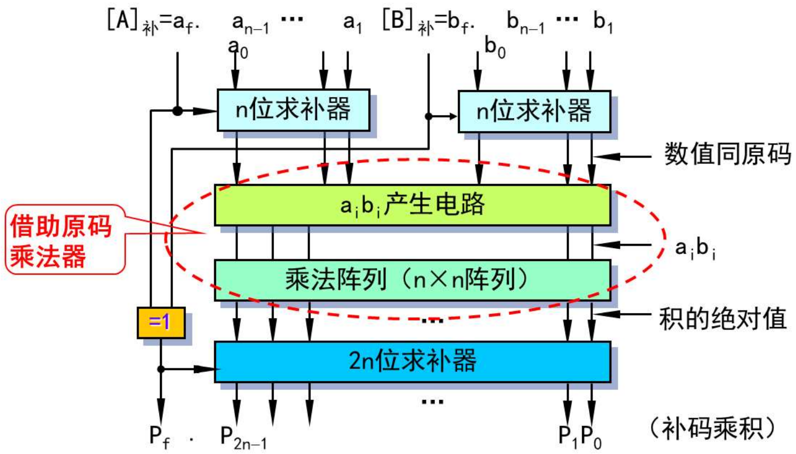


经过一级门电路延迟，即可得到所有的相加数

n×n位原码阵列乘法器



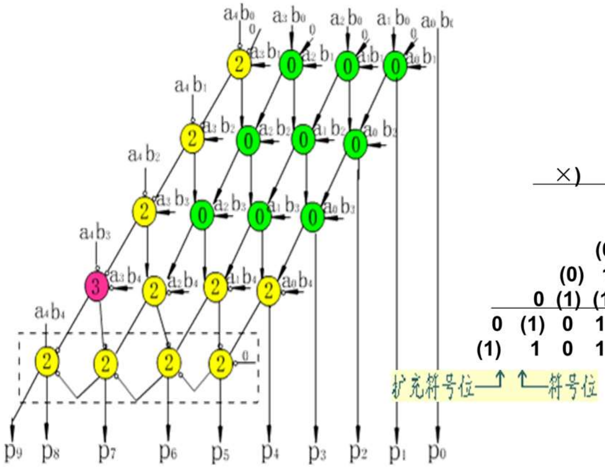
补码阵列乘法器



3. 直接补码阵列乘法器（略）

						(a4)	a3	a2	a1	a0	=A
×)						(b4)	b3	b2	b1	b0	=B
						(a4b0)	a3b0	a1b0	a1b0	a0b0	
					(a4b1)	a3b1	a2b1	a1b1	a0b1		
				(a4b2)	a3b2	a2b2	a1b2	a0b2			
			(a4b3)	a3b3	a2b3	a1b3	a0b3				
+)		a4b4	(a3b4)	(a2b4)	(a1b4)	(a0b4)					
	p9	p8	p7	p6	p5	p4	p3	p2	p1	p0	=P

5位乘5位的直接补码阵列乘法器




```
            (0) 1  1  0  1 = +13
×)          (1) 1  0  1  1 = −5
            (0) 1  1  0  1
         (0) 1  1  0  1
      (0) 0  0  0  0
   (0) 1  1  0  1
  0 (1) (1) (0) (1)
0 (1) 0  1  1  1  1  1  1
(1) 1  0  1  1  1  1  1  1 = −65
```

二、定点除法的运算方法和实现

定点除法运算

◆手算方式

例. 0.10110÷0.11111

```
              0.10110
0.11111 ) 0.10110 0
          −11111
           1101 00
          −11111
           10101 0
          − 11111
0.00000 1011 0
```

商： 0.10110

余数：0.10110×2^{-5}

实现除法的关键：

比较余数、除数绝对值大小，以决定上商。

ALU中定点除法流程

- 除前预处理
  - ①若被除数=0且除数≠0，|被除数|<|除数|，则商为0，不再继续
  - ②若被除数≠0、除数=0，则发生“除数为0”异常
  - ③若被除数和除数都为0，产生一个的NaN。
  - ④ 当被除数和除数都≠ 0，且商≠ 0时，才进行除法运算。
- 无符号数除法运算
  - 比较被除数/中间余数和除数：够减商1；不够减商0
  - 基本操作为减法和移位，故可与乘法合用同一套硬件

两个n位数相除：

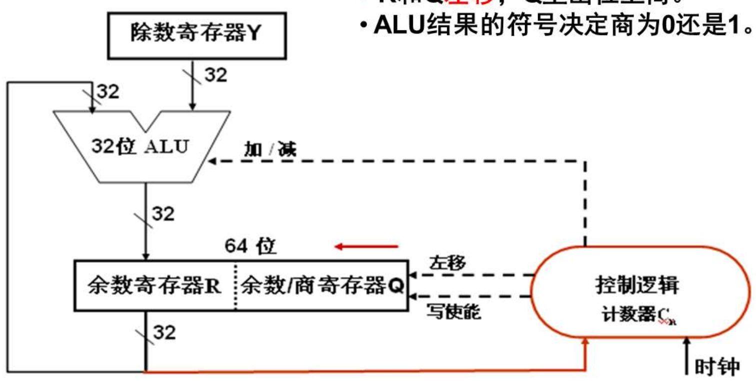
(1)定点正整数（即无符号数 ）相除：在被除数的高位添n个0

(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

无符号数除法算法的硬件实现

- R和Q左移，Q空出位上商。
- ALU结果的符号决定商为0还是1。



- 余数寄存器R：初始时高32位被除数；结束时是余数。
- 余数/商寄存器Q：初始时低32位被除数；结束时是商。
- ALU：对R和Y进行“加/减”，运算结果送回寄存器R。

小组讨论：第一次试商为1时的情况

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+1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

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(简)
  - 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。

原码除法

以小数为例

$$[x]_原 = x_0.x_1x_2 \cdots x_n$$

$$[y]_原 = y_0.y_1y_2 \cdots y_n$$

$$\left[\frac{x}{y}\right]_原 = (x_0 \oplus y_0).\frac{x^*}{y^*}$$

式中 $x^* = 0.x_1x_2\cdots$ x_n 为 x 的绝对值

$y^* = 0.y_1y_2\cdots$ y_n 为 y 的绝对值

商的符号位单独处理 $x_0 \oplus y_0$

数值部分为绝对值相除 $\frac{x^*}{y^*}$

1 原码恢复余数法

算法思想：减法-移位

2×余数-除数=新余数 $\begin{cases} \text{为正：够减，商1。} \\ \text{为负：不够减，商0，恢复原余数。} \end{cases}$

原码恢复余数法应用举例

例 $[x]_{原}=1.1011$ $[y]_{原}=1.1101$， 求 $[\frac{x}{y}]_{原}$

解：① $x_0 \oplus y_0 = 1 \oplus 1 = 0$

② 数值部分用**恢复余数**实现，减法用**补码加法**实现

数值部分是非负数，原码与补码相同。用补码可以实现**加减统一**，而且**是否够减**可以用中间余数的符号判断。

$x^*-y^* => [x^*]_{补}+[-y^*]_{补}$

$[y^*]_{补}=0.1101$

$[-y^*]_{补}=1.0011$

例 $[x]_{原}=1.1011$ $[y]_{原}=1.1101$，求 $[\frac{x}{y}]_{原}$

解：② $[y^*]_{补}=0.1101$ $[-y^*]_{补}=1.0011$

恢复余数的实现：$x^*-y^*=>[x^*]_{补}+[-y^*]_{补}$

	被除数（余数）	商	说明
	$\boxed{0.1011}$	0 000□	
	+1.0011		$+[-y^*]_{补}$
	1.1110	0	余数为负，上商0
	+0.1101		恢复余数 $+[y^*]_{补}$
	$\boxed{0.1011}$	0	恢复后的余数
逻辑左移	1.0110	0□	←1
	+1.0011		$+[-y^*]_{补}$
	0.1001	01	余数为正，上商1
逻辑左移	1.0010	01□	←1
	+1.0011		$+[-y^*]_{补}$

例 $[x]_{原}=1.1011$ $[y]_{原}=1.1101$，求 $[\frac{x}{y}]_{原}$

	被除数（余数）	商	说明
	0.0101	011	余数为正，上商1
逻辑左移	$\boxed{0.1010}$	011	←1
	+1.0011		$+[-y^*]_{补}$
	1.1101	0110	余数为负，上商0
	+0.1101		恢复余数 $+[y^*]_{补}$
	$\boxed{0.1010}$	0110	恢复后的余数
逻辑左移	1.0100	0110□	←1
	+1.0011		$+[-y^*]_{补}$
	0.0111	01101	余数为正，上商1

$\frac{x^*}{y^*}=0.1101$

$\therefore [\frac{x}{y}]_{原}=0.1101$

要点：
余数为**正** 商 1
余数为**负** 商 0，恢复余数

原码不恢复余数法（加减交替法）

推导

恢复余数法中：除数每一步所得余数Ri：

- 若$R_i>0$，则商1，并$2R_i-Y$
- 若$R_i<0$，则商0，并恢复余数，再左移一位，减Y，才得到正确的余数，即：

$$R_{i+1}=\underbrace{2}_{左移}\underbrace{(R_i+Y)}_{恢复}\underbrace{-Y}_{减}\quad=2R_i+Y$$

→不恢复余数法：

ri为正，则Qi为1，第i+1步作2ri−Y；

ri为负，则Qi为0，第i+1步作2ri+Y。

上例：$[x]_{原}=1.1011$ $[y]_{原}=1.1101$ 求 $[\frac{x}{y}]_{原}$

$[x^*]_{补}=0.1011$

$[y^*]_{补}=0.1101$

$[-y^*]_{补}=1.0011$

解：

被除数/余数	商	说明
0.1011	0000□	
+1.0011		$+[-y^*]_{补}$
1.1110	0	余数为负，上商0
1.1100	0□	←1
+0.1101		$+[y^*]_{补}$
0.1001	01	余数为正，上商1
1.0010	01□	←1
+1.0011		$+[-y^*]_{补}$
0.0101	011	余数为正，上商1
0.1010	011□	←1
+1.0011		$+[-y^*]_{补}$
1.1101	0110	余数为负，上商0
1.1010	0110□	←1
+0.1101		$+[y^*]_{补}$
0.0111	01101	余数为正，上商1

余数为负

余数为正

例题 结果

① $x_0 \oplus y_0 = 1 \oplus 1 = 0$

② $\frac{x^*}{y^*} = 0.1101$

$\therefore\ [\frac{x}{y}]_{原} = 0.1101$

特点：上商 n+1 次

移 n 次，加 n+1 次

用移位的次数判断除法是否结束。

补码除法（略）

- 补码除法符号位自动形成
- 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，若新余数的符号与除数符号一致表示够减，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，若得到的新余数的符号与除数符号一致表示不够减，否则为够减。

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		

总结：余数变号不够减，不变号够减

实现补码除法的基本思想

从上表可得到补码除法的基本算法思想：

（1）运算规则：

当被除数（或中间余数）与除数同号时，做减法；

异号时，做加法。

（2）上商规则：

若余数符号不变，则够减，商1；否则不够减，商0。

（3）修正规则：

若被除数与除数符号一致，则商为正。此时，“够减，商1；不够减，商0，故上商规则正确，无需修正”

若被除数与除数符号不一致，则商为负。此时，“够减，商0；不够减，商1，故上商规则相反，需修正”

即：若商为负值，则需要“各位取反，末位加1”来得到真正的商

补码除法也有：恢复余数法和不恢复余数法

定点运算小结

逻辑运算、移位运算、扩展运算等电路简单

主要考虑算术运算

- 定点运算涉及的对象

  无符号数；带符号整数(补码)；原码小数；移码整数

- 定点运算：(ALU实现基本算术和逻辑运算，ALU+移位器实现其他运算)

  补码加/减：符号位和数值位一起运算，减法用加法实现。同号相加时可能溢出

  原码加/减：符号位和数值位分开运算，用于浮点数尾数加/减运算

  移码加减：移码的和、差等于和、差的补码，用于浮点数阶码加/减运算

小 结

乘法运算：

无符号数乘法：“加”+“右移”

原码（一位/两位）乘法：符号和数值分开运算，数值部分用无符号数乘法实现，用于浮点数尾数乘法运算。

补码（一位/两位）乘法：符号和数值一起运算，采用Booth算法。

快速乘法器：流水化乘法器、阵列乘法器

除法运算：

无符号数除法：用“加/减”+“左移”，有恢复余数和不恢复余数两种。

原码除法：符号和数值分开，数值部分用无符号数除法实现，用于浮点数尾数除法运算。

补码除法：符号位和数值位一起。有恢复余数和不恢复余数两种。

快速除法器：很难实现流水化除法器，可实现阵列除法器，或用乘法实现

- 定点部件：ALU、GRS、MUX、Shifter、Q寄存器等，CU控制执行