计算机组成原理

第6章 计算机的运算方法

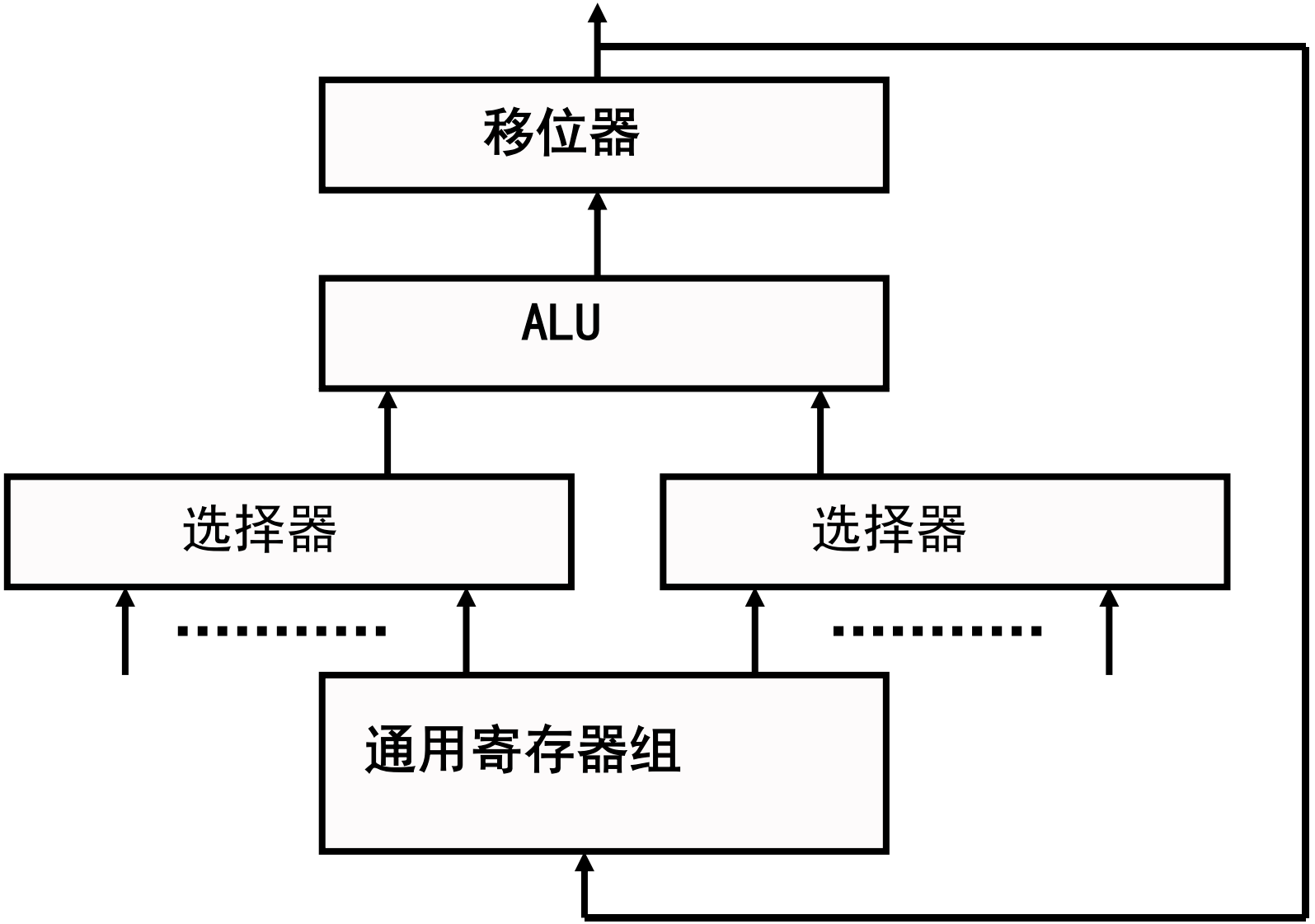
llxx@ustc.edu.cn

wjluo@ustc.edu.cn

6.3 定点运算

移位、加减、乘、除

运算器结构



移位运算

1. 移位的意义

2. 算术移位规则

3. 算术移位与逻辑移位

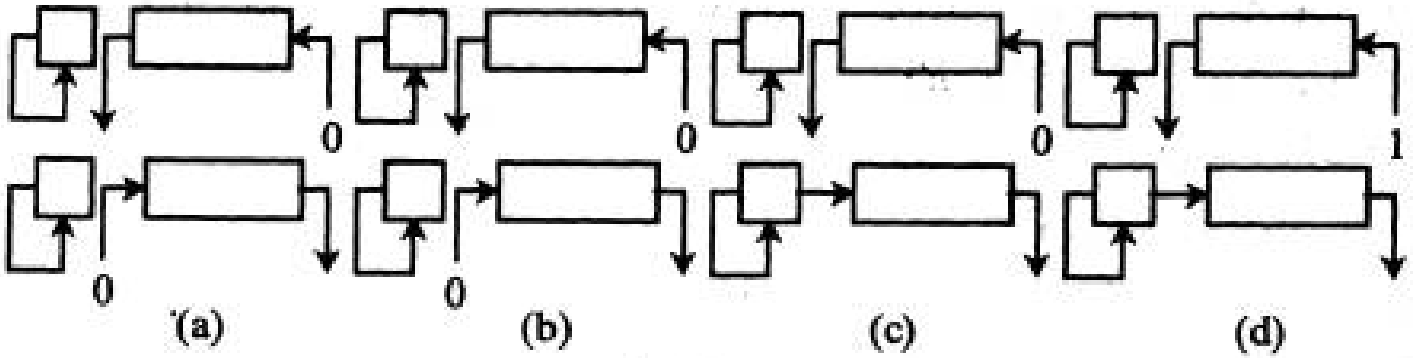
不同码制机器数算术移位后的空位添补规则

- **逻辑移位**：无符号数，空位均补**0**
- **算术移位**：有符号数，**符号位**保持不变
  - 左移或右移n位相当于乘以或除以2^n
- **算术移位**对空出的位应该补**0**还是**1**？
  - 算术运算是对真值操作。无论那种码制，应该保证真值的一致性和运算结果的正确性。

	码　制	添补代码
正数	原码、补码、反码	0
负数	原码	0
	补码	左移添0
		右移添1
	反　码	1

原：1,1011**1000**
反：1,0100**0111**
　　+　　　　1
补：1,0101**1000**

实现算术左移和右移操作的硬件框图



（a）真值为正的三种机器数的移位操作

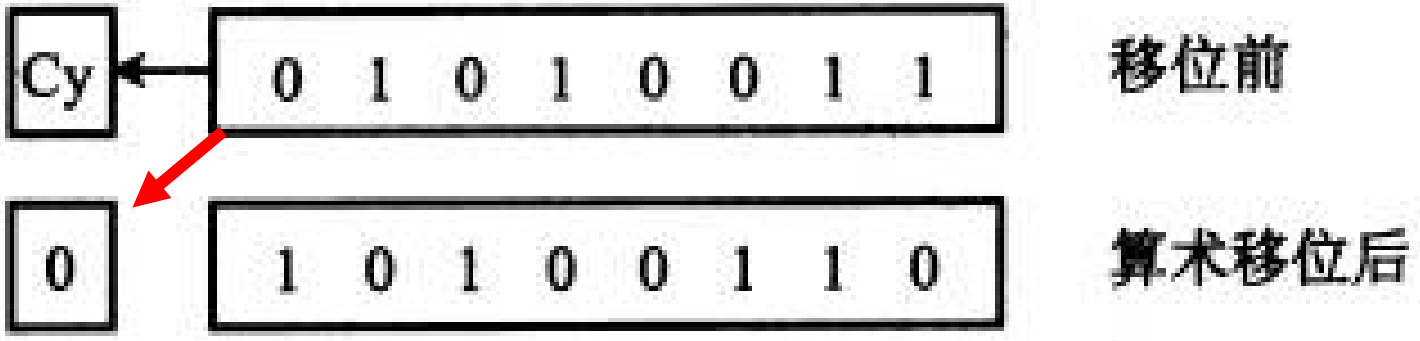
（b）负数原码的移位操作

（c）负数补码的移位操作

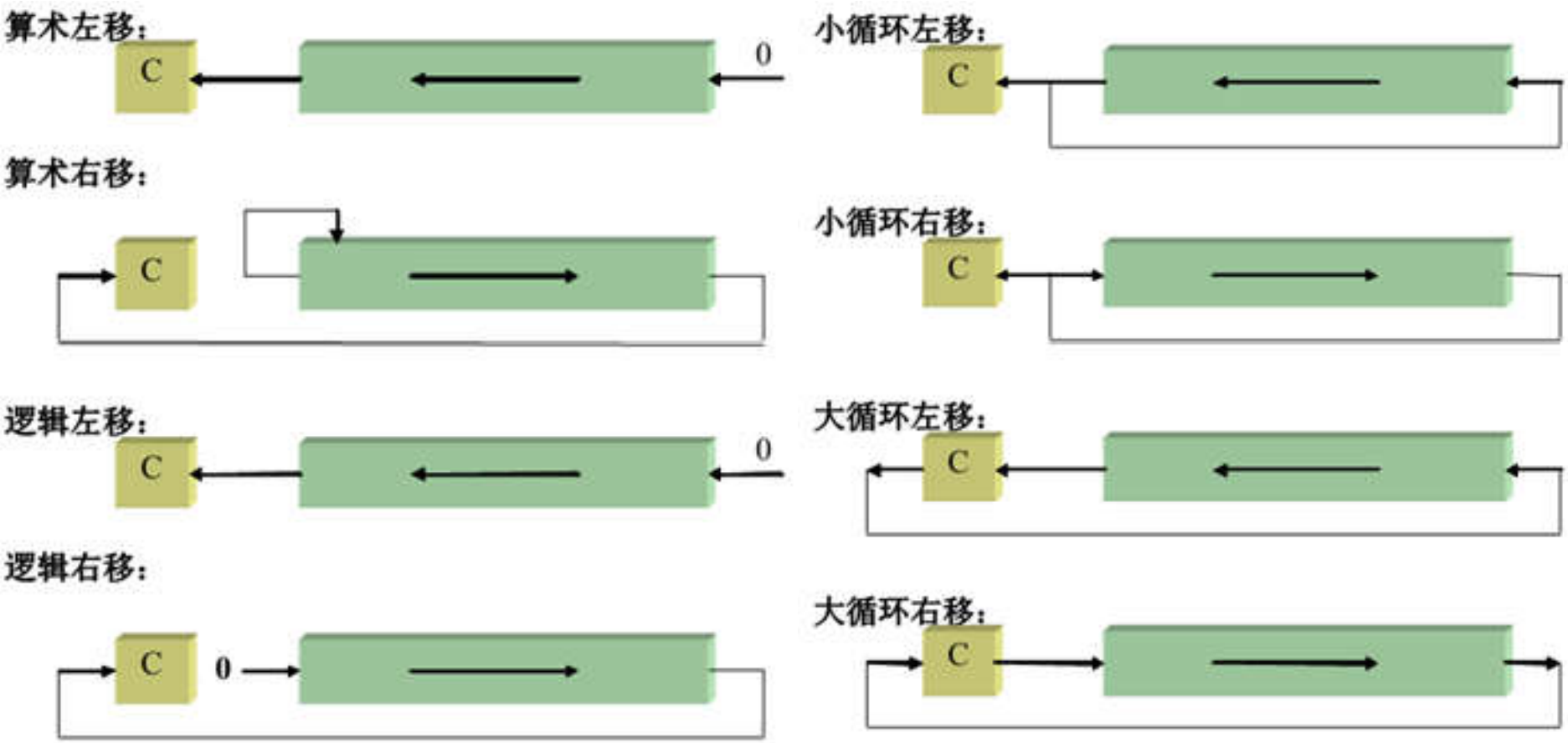
（d）负数反码的移位操作

采用带进位(C_y)的移位

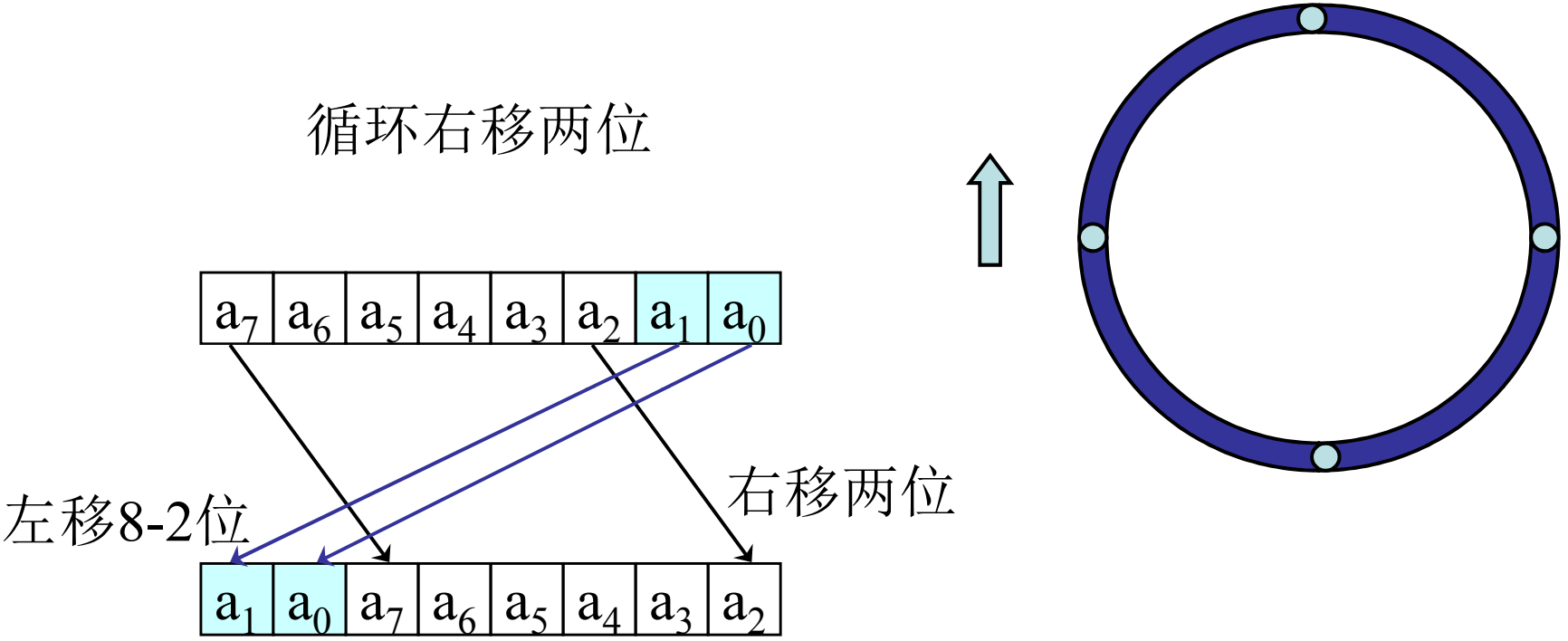
- 为了避免算术左移时最高数位丢1，可采用带进位(C_y)的移位
  - 符号位移至C_y，最高数位就可避免移出。



移位的类型



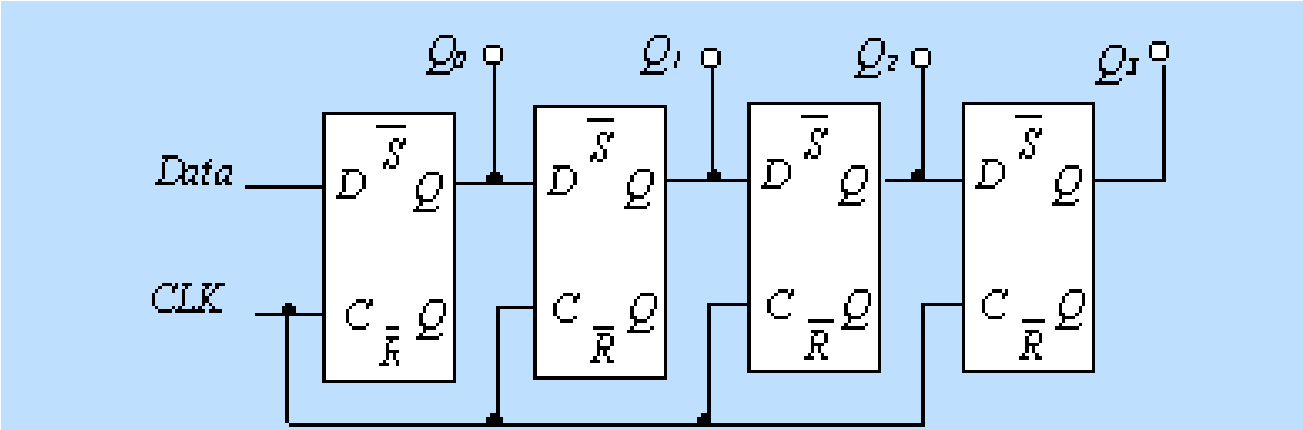
左移可以由右移位实现



左移 i 位等价于右移 $32 - i$ 位

移位寄存器

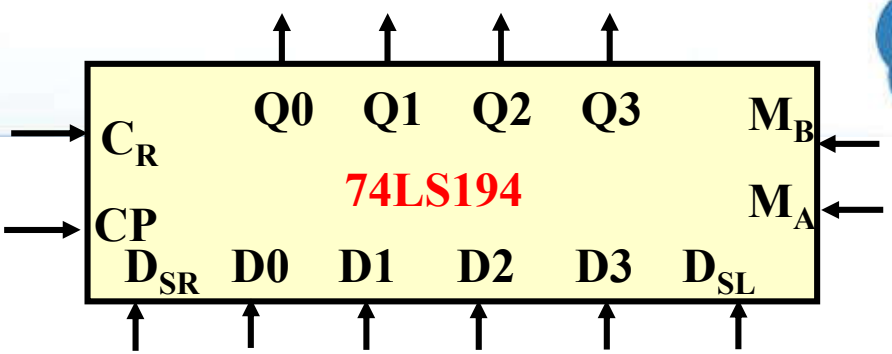
- 移位寄存器应用很广，是高速微处理器中的常用部件
  - 用于实现移位指令、浮点计算中的小数点对齐、串并数据转换等
  - 可构成
    - 移位寄存器型计数器
    - 顺序脉冲发生器
    - 串行累加器



D触发器设计的移位寄存器

双向移位寄存器

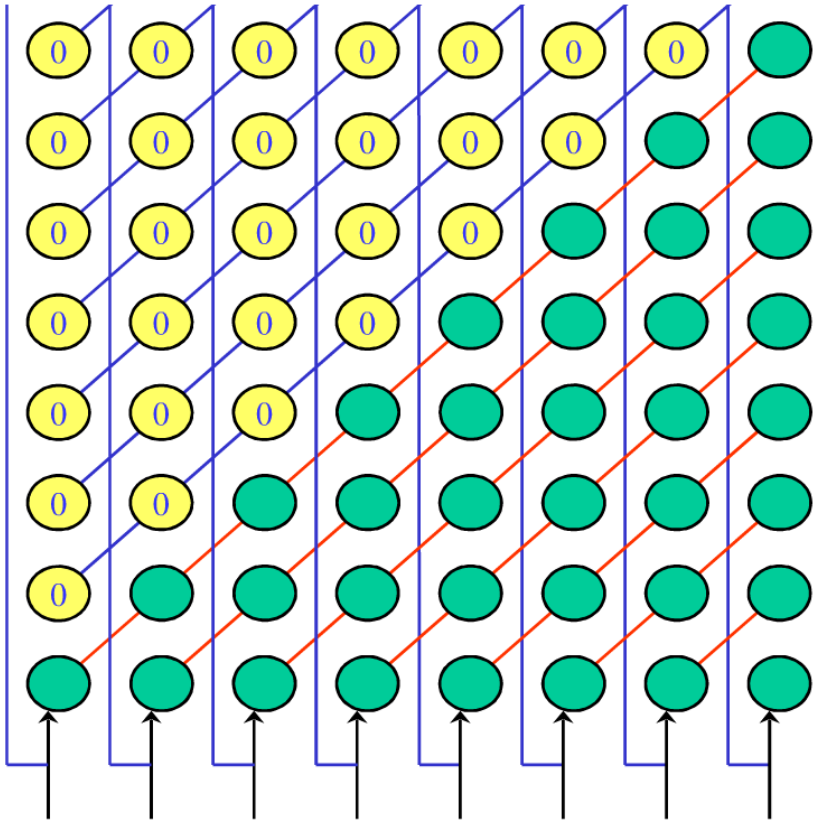
74LS194的功能

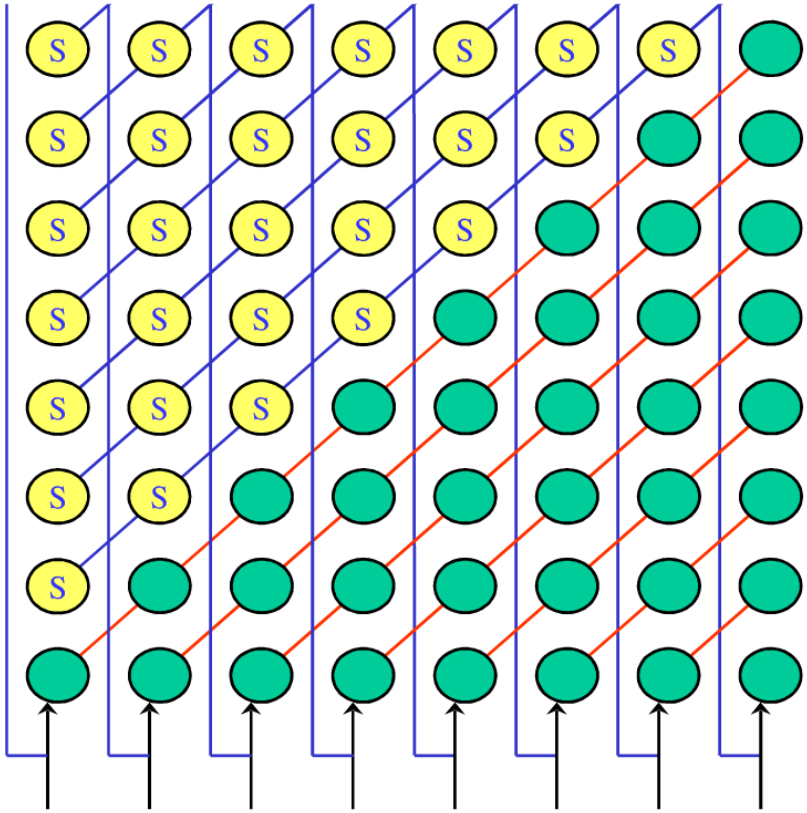


C_R	CP	M_B	M_A	Q0 Q1 Q2 Q3
0	Φ	Φ	Φ	0 0 0 0 清零
1	↑	0	0	保持
1	↑	0	1	D_{SR} → 右移一位
1	↑	1	0	左移一位 ← D_{SL}
1	↑	1	1	D0 D1 D2 D3 (并行输入)

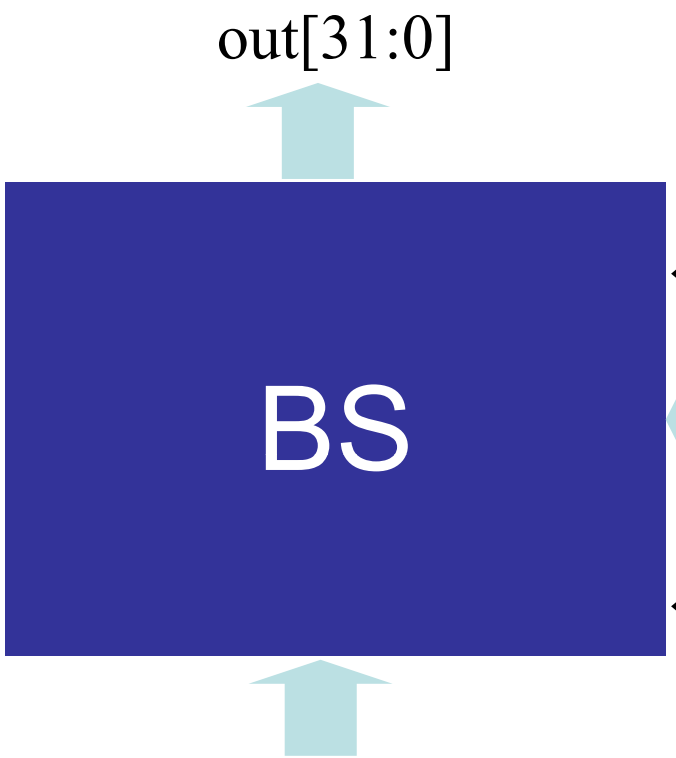
桶式移位器BS(Barrel Shifter)

- BS能在单周期内完成多种方式、各种位数的移位操作。
  - 算术移位、逻辑移位、循环移位





BS接口



In[31:0]：32位输入

Out[31:0]：32位输出

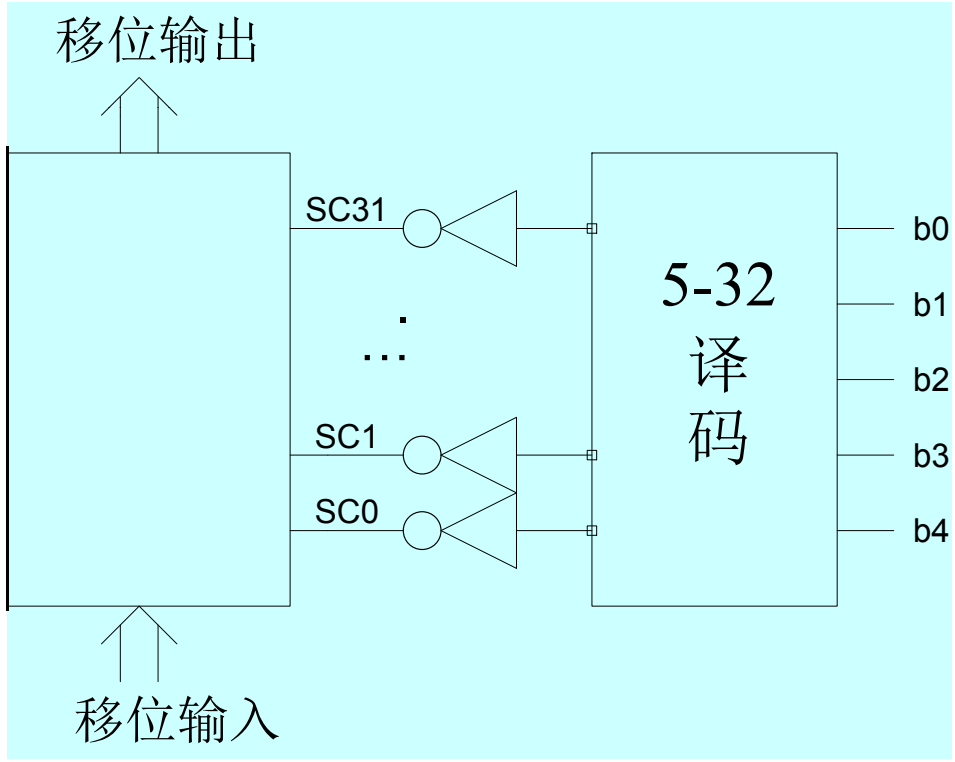
B[4:0]：移位位数

Sin1, Sin2：移位方式

左移、右移、循环、保持

BS的实现：全译码方式

- 对表示移位次数的二进制位进行完全译码，分别给出各种移位的单独控制线。
- 对于32位字长来说，移位部分有32根控制线SC_{31}~ SC_0分别控制移31~0位时的操作



In[31:0]：32位输入

Out[31:0]：32位输出

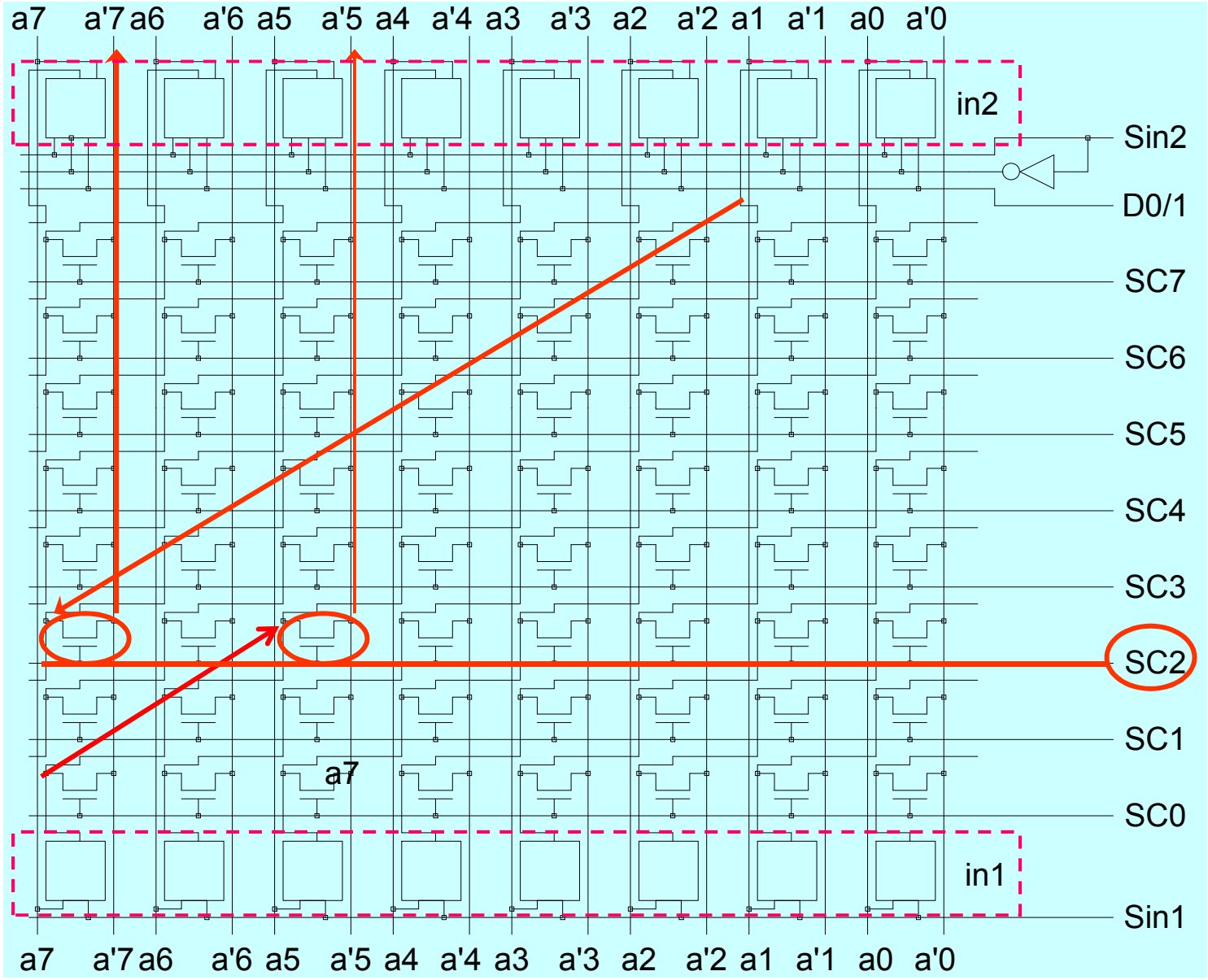
B[4:0]：移位位数

Sin1, Sin2：移位方式

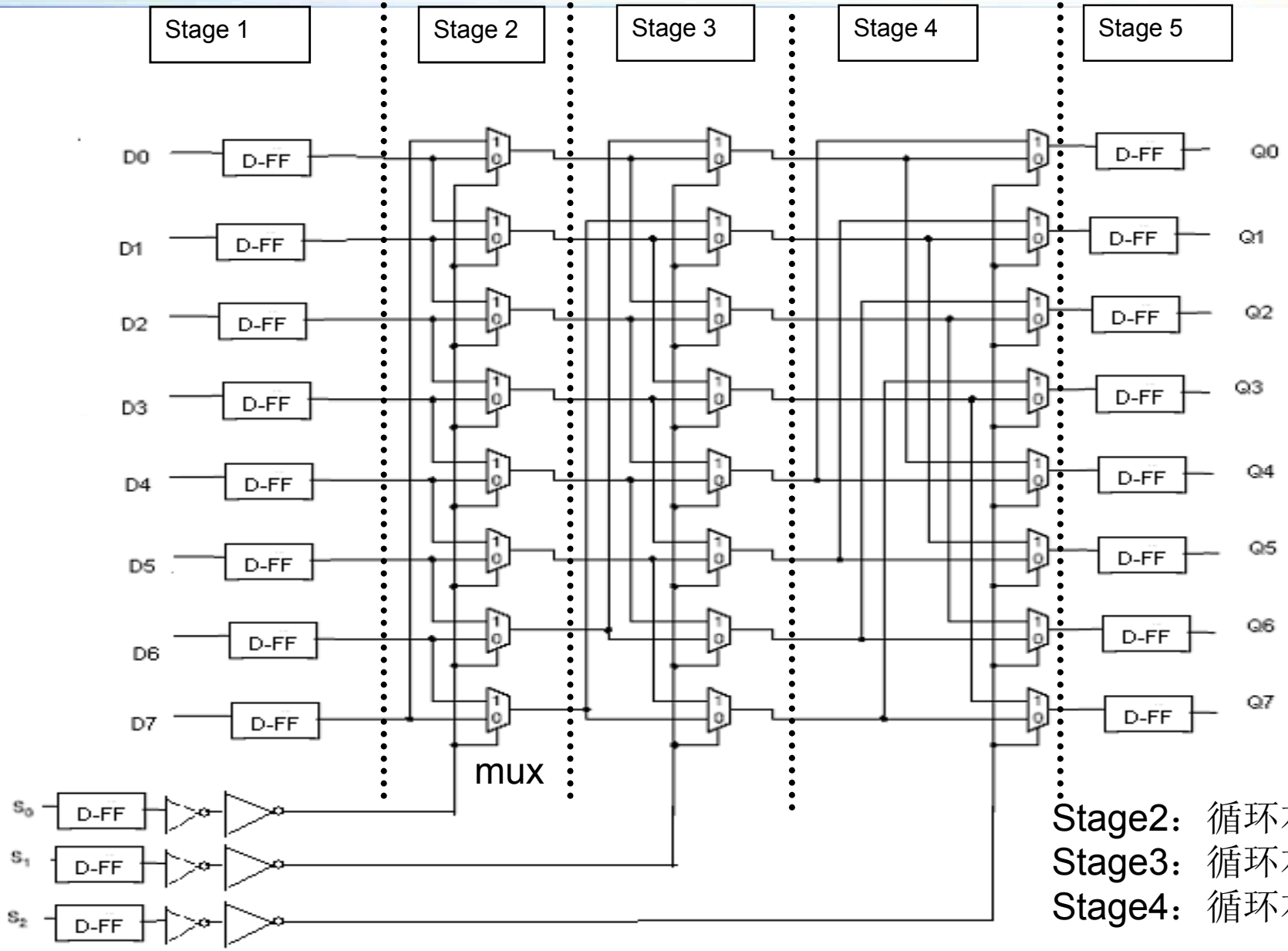
左移、右移、循环、保持

8位全译码方式BS

右移两位: 010



8 Bit Barrel Shifter：传输门实现模式



Stage2：循环右移1位
Stage3：循环右移2位
Stage4：循环右移3位

加法和减法运算

1. 补码加减的基本公式

2. 溢出判断

3. 补码加减法所需的硬件配置

4. 补码加减运算控制流程

加减法运算

- 原码加减法复杂，需要事先判断数的符号，然后决定做加法还是做减法运算。

- 补码加减法运算简单，可将“正数加负数”的操作，转化为“正数加正数”的操作。一般计算机采取补码进行加减法运算。

- 因减法运算可看作被减数加上一个减数的负值，即**A-B=A+(-B)**，故在此将机器中的减法运算和加法运算合在一起讨论。

- 符号位参与运算

1. 补码加减的基本公式

- 补码加法的基本公式为：

  整数　$[A]_{补}+[B]_{补}=[A+B]_{补} \pmod{2^{n+1}}$

  小数　$[A]_{补}+[B]_{补}=[A+B]_{补} \pmod{2}$

- 对于减法
  - 因$A-B=A+(-B)$，则$[A-B]_{补}=[A+(-B)]_{补}$
  - 由补码加法基本公式可得：

  整数　$[A-B]_{补}=[A]_{补}+[-B]_{补} \pmod{2^{n+1}}$

  小数　$[A-B]_{补}=[A]_{补}+[-B]_{补} \pmod{2}$

补码的加、减法的例子

$[X+Y]_{补}=[X]_{补}+[Y]_{补}$　　$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$

运算过程举例（假设机器字长4位，其中1位表示符号位）：

（a）（-7）+（+5）

1,001
0,101
1,110　=-2

（b）（-4）+（+4）

1
1,010
1 0,011　=溢出

丢掉

（c）（+5）+（+4）

0,101
0,100
1,001　=溢出

计算机中这种超出机器字长的现象，称为溢出。

在补码定点运算中，必须对结果是否溢出进行判断。

2.溢出判断

- 如果运算的结果，超出了计算机能表示的数的范围，会得出错误的结果，这种情况称为溢出。
  - 对于字长为n的计算机，那么它能表示的定点补码范围为$-2^{n-1} \leqslant X \leqslant 2^{n-1}-1$
  - 若运算结果小于-2^{n-1}或大于$2^{n-1}-1$，则发生溢出
  - 发生溢出时，数值的有效位占据了符号位。

- 两种方法

① 用一位符号位判断溢出

② 用两位符号位判断溢出

用一位符号位判断溢出

- 两个相同符号数相加，其运算结果符号应与被加数相同，否则产生溢出；
- 相异符号数相加，相同符号数相减，不会产生溢出。
- 两个相异符号数相减，其运算结果符号应与被减数相同，否则产生溢出。

- 由于减法运算在机器中是用加法器实现的，如此有如下结论：
  - 无论是加法还是减法，只要实际参加操作的两个数（减法时即为被减数和“求补”以后的减数）符号相同，结果又与原操作数的符号不同，即为溢出。

用一位符号位判断溢出

- 准则：“两个相同符号数相加，其运算结果符号应与被加数相同，否则产生溢出”
  - **x0**、**y0**为加数与被加数的符号位，**z0**为和的符号位。
  - **V=1**，则溢出。

$$V = x_0\ y_0\ \overline{z_0} + \overline{x_0}\ \overline{y_0}\ z_0$$

- 通常“用符号位产生的进位和最高有效位向符号位产生的进位进行异或操作后，按其结果进行判断”？
  - 若异或结果为**1**（即不同），则溢出；
  - 若异或结果为**0**（即相同），则没有溢出。
  - 快！

$$V = C_0\ \overline{C_1} + \overline{C_0}\ C_1$$

补码的加、减法的溢出判断

$[X+Y]_{补}=[X]_{补}+[Y]_{补}$　　$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$

运算过程举例（假设机器字长4位，其中1位表示符号位）：

（a）（-7）+（+5）

1,001
0,101
1,110　=-2

（b）（-4）+（+4）

1,100
0,100
1 0,000　=0

（c）（+5）+（+4）

0,101
0,100
1,001　=溢出

（d）（-7）+（-6）

1,001
1,010
1 0,011　=溢出

丢掉

用两位符号位判断溢出

- 变形补码

$$[x]_{补'}=\begin{cases} x & 1>x\geq 0 \\ 4+x \quad (\mathrm{mod}\ 4) & 0>x\geq -1 \end{cases}$$

- 用变形补码做加法操作时，两位符号位连同数值部分一起参加运算。

- 运算结果溢出判断规则：
  - 正常时两个符号位的值相同
  - 两个符号位不同，则表明发生了溢出。

双符号位溢出判断法

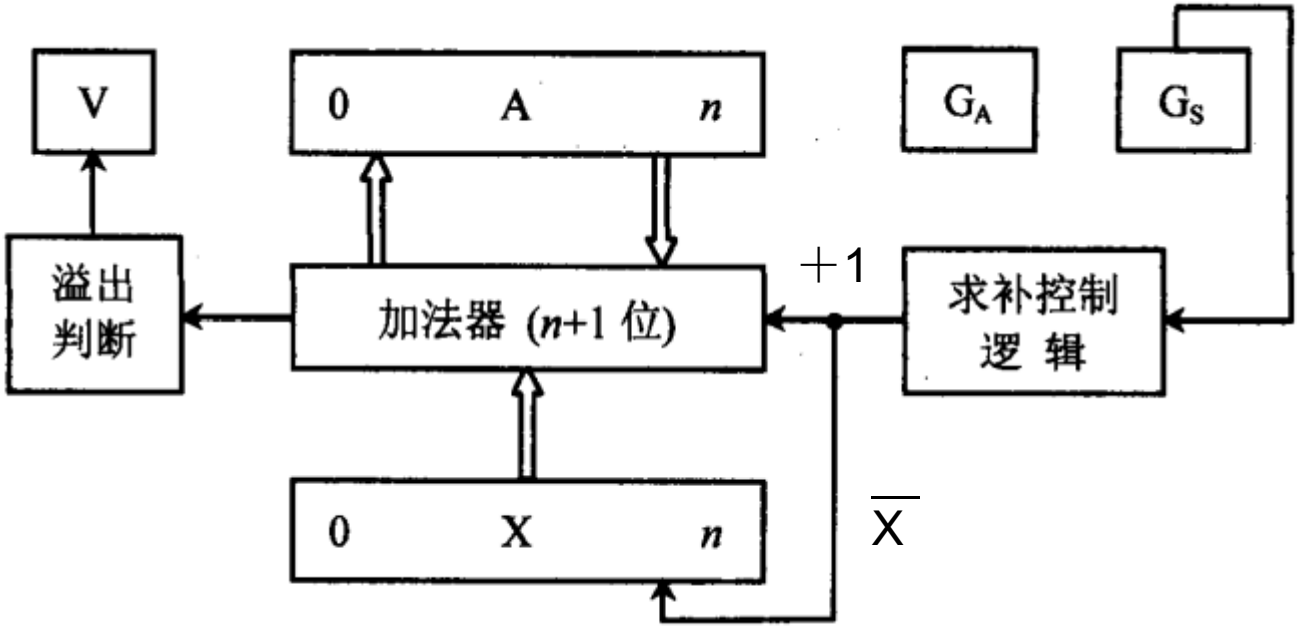
双符号含义：
00表示运算结果为正数；
01表示运算结果正溢出；
10表示运算结果负溢出；
11表示运算结果为负数。

第一位符号位为运算结果的真正符号位。

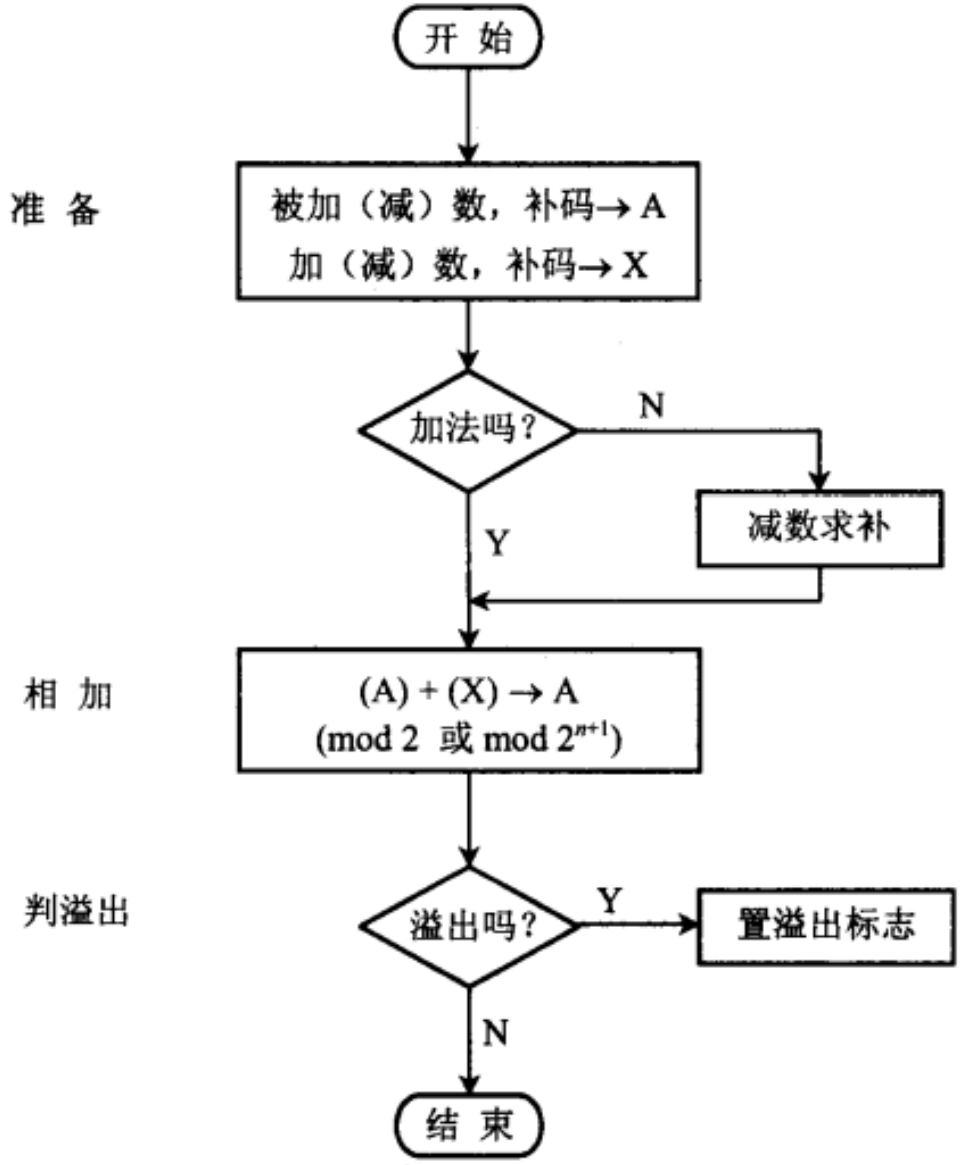
- 设：z0’、z0为和的双符号位
  - 则：V=1，则溢出。

$$V = z_0' \overline{z_0} + \overline{z'_0}\, z_0 = z_0' \oplus z_0$$

3.补码加减法所需的硬件配置



4.补码加减运算控制流程



乘法运算

1. 笔乘算法的分析与改进

2. 原码乘法
   - 原码一位乘运算规则
   - 原码一位乘所需的硬件配置
   - 原码一位乘控制流程
   - 原码两位乘

3. 补码乘法
   - 校正法
   - 比较法乘运算规则
   - 补码比较法（**Booth**乘法）所需的硬件配置
   - 补码比较法（**Booth**乘法）控制流程
   - 补码两位乘

1. 分析笔算乘法

- 设A=0.1101，B=0.1011，求A×B。
- 乘积的符号由两数符号心算而得：正正得正
- 数值部分的运算如下：

```
        0.1101
  ×     0.1011
-------------------
        1101      ·····A
       1101       ·····A
      0000        ·····0
     1101         ·····A
-------------------
  0.  10001111
```

若计算机完全模仿笔算乘法步骤，将会有两大困难：
其一，将四个位积一次相加，机器难以实现；
其二，乘积位数增长了一倍，这将造成器材的浪费和运算时间的增加。

- 所以 A×B=+0.10001111
- 被乘数A的多次左移，以及四个位积的相加运算。

笔算乘法的改进

- **A•B= A•0.1011**
  =0.1A+0.00•A+0.001•A+0.0001•A
  =0.1A+0.00•A+0.001(A+0.1A）
  =0.1A+0.01[0•A+0.1(A+0.1A)]
  =0.1{A+0.1[0•A+0.1(A+0.1A)]}
  =2^{-1}{A+2^{-1} [0•A+2^{-1} (A+2^{-1}A)]}
  =2^{-1}{A+2^{-1} [0•A+2^{-1} (A+2^{-1}(A+0))]}

- 如果**B＝0.1010**？

  – 两数相乘的过程，可视作加法及移位运算(乘2^{-1}相当于做一位右移)，这对计算机来说是非常容易实现的。

笔算乘法的改进

- $2^{-1}\{A+2^{-1}\ [0\bullet A+2^{-1}\ (A+2^{-1}(A+0))]\}$

- 从初始值为0开始，对上式作分步运算，则
  第一步：$A+0=0.1101+0.0000=0.1101$
  第二步：$2^{-1}\ (A+0)=0.01101$
  第三步：$A+2^{-1}(A+0)=0.1101+0.01101=1.00111$
  第四步：$2^{-1}\ [A+2^{-1}\ (A+0)]=0.100111$
  第五步：$0\bullet A+2^{-1}\ [A+2^{-1}\ (A+0)]=0.100111$
  第六步：$2^{-1}\{0\bullet A+2^{-1}\ [A+2^{-1}\ (A+0)]\}=0.0100111$
  第七步：$A+2^{-1}\{0\bullet A+2^{-1}\ [A+2^{-1}\ (A+0)]\}=1.0001111$
  第八步：$2^{-1}\ \{A+2^{-1}[0\bullet A+2^{-1}\ (A+2^{-1}\ (A+0))]\}$
  $=0.10001111$

判断位：乘数的末位，确定被乘数是否与原部分积相加

笔算乘法规则

- ①乘法运算可用移位和加法来实现，当两个四位数相乘，总共需做四次加法和四次移位。
  ②由乘数的末位值（判断位）确定被乘数是否与原部分积相加，然后右移一位，形成新的部分积；同时，乘数也右移一位，由次低位作新的末位，空出最高位放部分积的最低位。
  ③每次做加法时，被乘数仅仅与原部分积的高位相加，其低位被移至乘数所空出的高位位置。

- 计算机很容易实现这种运算规则: 用一个寄存器存放被乘数，一个寄存器存放乘积的高位，又用一个寄存器存放乘数及乘积的低位，再配上加法器及其他相应电路，就可组成乘法器。

2. 原码一位乘

- 原码一位乘运算规则
- 原码一位乘所需的硬件配置
- 原码一位乘控制流程

原码一位乘运算规则

- 以小数为例，设$[x]_{原}=x_0.x_1x_2\ldots x_n$, $[y]_{原}=y_0.y_1y_2\ldots y_n$, 则

$$[x]_{原}\bullet[y]_{原}=x_0\oplus y_0.(0.x_1x_2\cdots x_n)(0.y_1y_2\cdots y_n)$$

- 其中

$0.x_1x_2\cdots x_n$为x的绝对值，记作 x^*

$0.y_1y_2\cdots y_n$为y的绝对值，记作 y^*

- 乘积的符号通过两数符号的逻辑异或求得。
- 乘积的数值部分由两数绝对值相乘，其通式为：

$$\begin{aligned}x^*\cdot y^* &= x^*(0.y_1y_2\cdots y_n)\\ &= x^*(y_12^{-1}+y_22^{-2}+\cdots+y_n2^{-n})\\ &= 2^{-1}(y_1x^*+2^{-1}(y_2x^*+2^{-1}(\cdots+2^{-1}(y_{n-1}x^*+2^{-1}(y_nx^*+0)\cdots)))\end{aligned}$$

z_0

z_1

z_2

……

z_{n-1}

z_n

原码一位乘运算规则

再令 z_i 表示第 i 次部分积，式(6.9)可写成递推公式：

$$
\begin{aligned}
z_0 &= 0 \\
z_1 &= 2^{-1}(y_n \cdot x^* + z_0) \\
z_2 &= 2^{-1}(y_{n-1} \cdot x^* + z_1) \\
&\vdots \\
z_i &= 2^{-1}(y_{n-i+1} \cdot x^* + z_{i-1}) \\
&\vdots \\
z_n &= 2^{-1}(y_1 \cdot x^* + z_{n-1})
\end{aligned}
$$

- 对操作数的绝对值进行运算
- 加法和右移
  - 乘数末位为判断位，决定部分积是否加被乘数
- 符号位单独生成

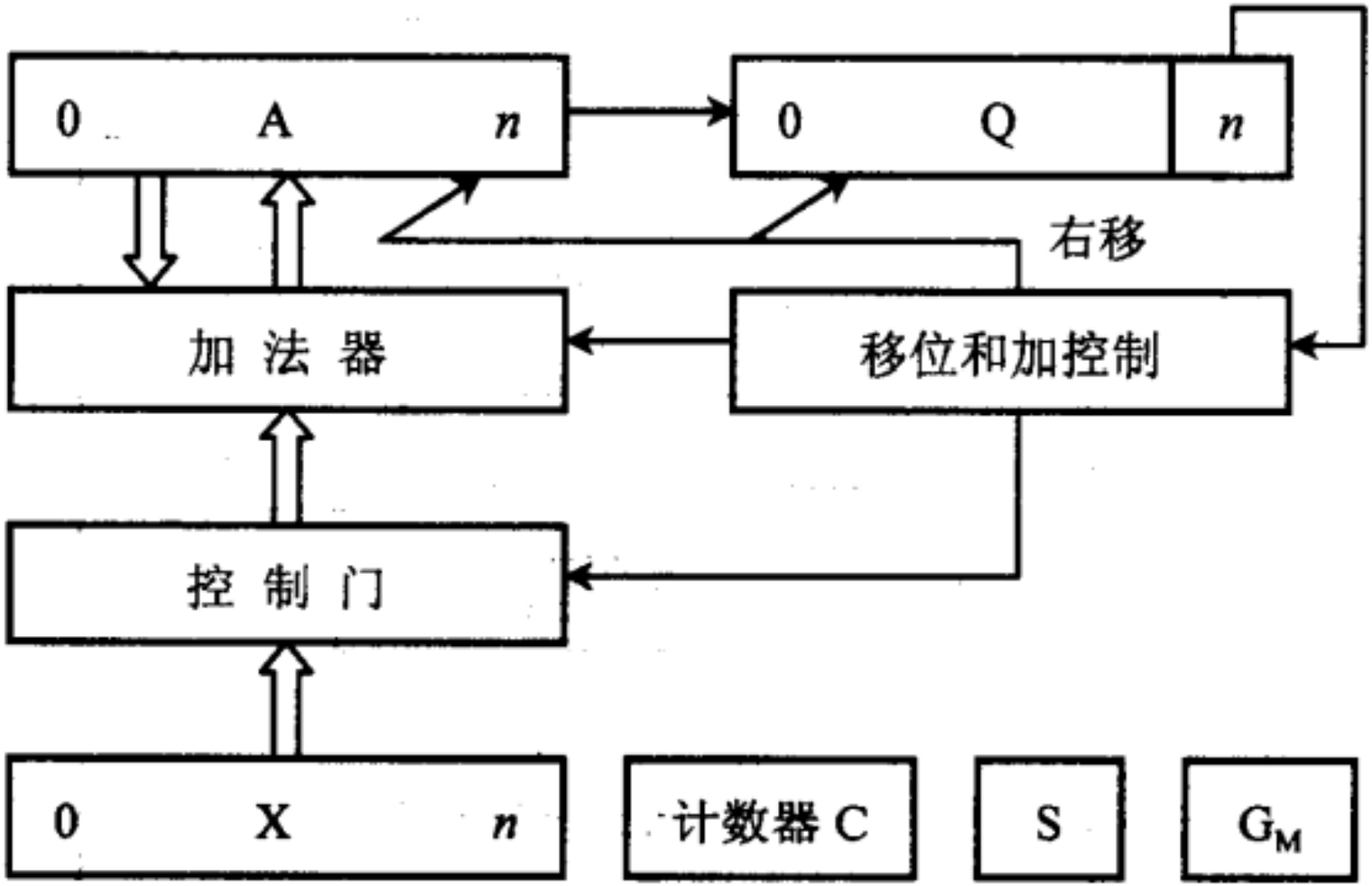
例

- 已知：**x=-0.1110，y=-0.1101；求：$[x\cdot y]_{原}$。**
- 解：**$[x]_{原}$=1.1110，x*=0.1110，x_0=1**
  $[y]_{原}$=1.1101，y*=0.1101，y_0=1

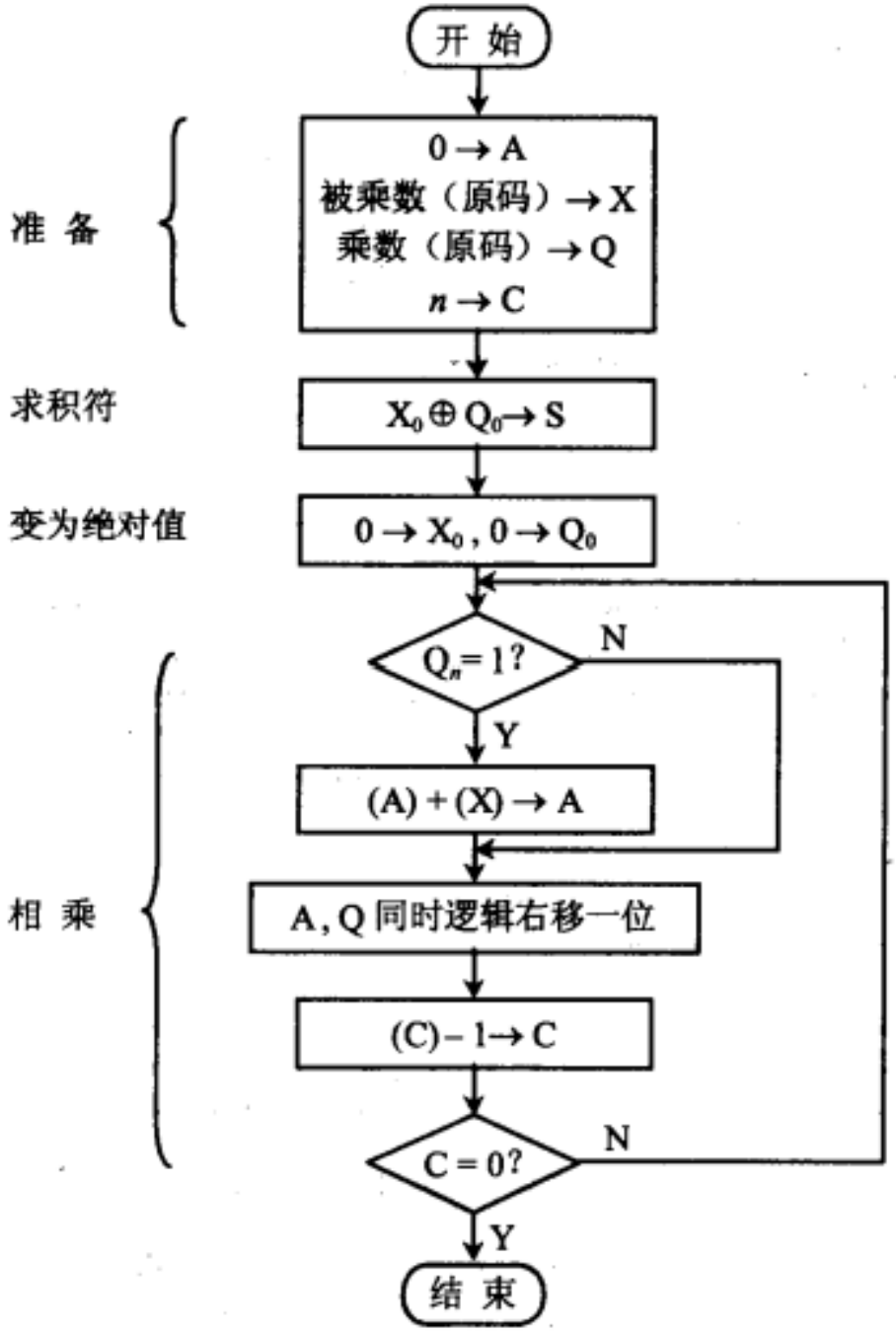
部分积	乘数	说明
0.0000 + 0.1110	1101	开始部分积z_0=0 乘数为1，加上x*
0.1110 0.0111 + 0.0000	0110	→1位得z_1， 乘数同时→1位 乘数为0，加上0
0.0111 0.0011 + 0.1110	1011	→1位得z_2， 乘数同时→1位 乘数为1，加上x*
1. 0001 0.1000 + 0.1110	1101	→1位得z_3， 乘数同时→1位 乘数为1，加上x*
1.0110 0.1011	0110	→1位得z_4 乘数已全部移出

即x*·y*=0.10110110
乘积的符号位为x0和y0的异或，即0。
故$[x\cdot y]_{原}$=0.10110110。

原码一位乘所需的硬件配置



原码一位乘控制流程


开 始
准 备
0 → A
被乘数（原码）→ X
乘数（原码）→ Q
n → C
求积符
X₀ ⊕ Q₀ → S
变为绝对值
0 → X₀ , 0 → Q₀
相 乘
Qₙ = 1?
N
Y
(A) + (X) → A
A , Q 同时逻辑右移一位
(C) − 1 → C
C = 0?
N
Y
结 束

4. 补码乘法

- 补码一位乘运算规则
  - 校正法
  - 比较法（**Booth**算法）
    - Booth's multiplication algorithm
    - was invented by Andrew Donald Booth in 1951
- 补码**Booth**算法所需的硬件配置
- 补码**Booth**算法控制流程
- 补码两位乘

补码一位乘运算规则

- 设被乘数$[x]_{补}=x_0.x_1x_2\cdots x_n$，乘数$[y]_{补}=y_0.y_1y_2\cdots y_n$

- 两个引理

① 被乘数x符号任意，乘数y的符号为正时，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot(0.y_1y_2\cdots y_n)$$

② 被乘数x符号任意，乘数y的符号为负时，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot(0.y_1y_2\cdots y_n)+[-x]_{补}$$

补码一位乘运算规则证明

- 设被乘数$[x]_{补}=x_0.x_1x_2\cdots x_n$，乘数$[y]_{补}=y_0.y_1y_2\cdots y_n$

① 被乘数x符号任意，乘数y的符号为正时，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot y$$

- 证明：

$$[x]_{补}=x_0.x_1x_2\cdots x_n=2+x=2^{n+1}+x(\mathrm{mod}\quad 2)$$

$$[y]_{补}=0.y_1y_2\cdots y_n=y$$

$$则\ [x]_{补}\cdot[y]_{补}=[x]_{补}\cdot y=(2^{n+1}+x)y=2^{n+1}y+xy$$

$$由于\ y=0.y_1y_2\cdots y_n=\sum_{i=1}^{n}y_i2^{-i},$$

$$则\ 2^{n+1}y=2\sum_{i=1}^{n}y_i2^{n-i},$$

$$又\sum_{i=1}^{n}y_i2^{n-i}是一个大于或等于\quad 1的正整数，则$$

$$根据模运算的性质，有\quad 2^{n+1}y=2(\mathrm{mod}\quad 2)$$

$$[x]_{补}\cdot[y]_{补}=2^{n+1}y+xy=2+xy=[xy]_{补}(\mathrm{mod}\quad 2)$$

$$则\ [x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot y$$

补码一位乘运算规则证明（续）

- 设被乘数$[x]_{补}=x_0.x_1x_2\ldots x_n$，乘数$[y]_{补}=y_0.y_1y_2\ldots y_n$

② 被乘数x符号任意，乘数y的符号为负时，有：

$$[x \bullet y]_{补}=[x]_{补} \bullet [y]_{补}=[x]_{补} \bullet (0.y_1y_2\ldots y_n)+[-x]_{补}$$

证明：

$$[x]_{补} = x_0.x_1x_2\cdots x_n$$

$$[y]_{补} = 1.y_1y_2\cdots y_n = 2 + y(\mathrm{mod}\quad 2)$$

则$$y = [y]_{补} - 2 = 1.y_1y_2\cdots y_n - 2 = 0.y_1y_2\cdots y_n - 1$$

$$xy = x(0.y_1y_2\cdots y_n - 1) = x(0.y_1y_2\cdots y_n) - x$$

则$$[x \bullet y]_{补} = [x(0.y_1y_2\cdots y_n)]_{补} + [-x]_{补}$$

将上式中的$0.y_1y_2\cdots y_n$视为一个正数，正好与 第一种情况相同。

$$[x(0.y_1y_2\cdots y_n)]_{补} = [x]_{补} \bullet (0.y_1y_2\cdots y_n)$$

所以，$$[x \bullet y]_{补} = [x]_{补} \bullet (0.y_1y_2\cdots y_n) + [-x]_{补}$$

补码一位乘运算规则（校正法）

① 被乘数x符号任意，乘数y的符号为正时，有：

$$[x \cdot y]_{补}=[x]_{补} \cdot [y]_{补}=[x]_{补} \cdot y$$

$$[z_0]_{补} = 0$$

$$[z_1]_{补} = 2^{-1}(y_n[x]_{补} + [z_0]_{补})$$

$$[z_2]_{补} = 2^{-1}(y_{n-1}[x]_{补} + [z_1]_{补})$$

$$\vdots$$

$$[z_i]_{补} = 2^{-1}(y_{n-i+1}[x]_{补} + [z_{i-1}]_{补})$$

$$\vdots$$

$$[x \cdot y]_{补} = [z_n]_{补} = 2^{-1}(y_1[x]_{补} + [z_{n-1}]_{补})$$

②被乘数x符号任意，乘数y的符号为负时，有：

$$[x \cdot y]_{补}=[x]_{补} \cdot [y]_{补}=[x]_{补} \cdot (0.y_1y_2\ldots y_n)+[-x]_{补}$$

补码一位乘运算规则——校正法

- 当乘数y为正时，可按类似原码乘法的规则进行运算。
- 当乘数为负时，把乘数的补码$[y]_{补}$去掉符号位，看成一个正数与$[x]_{补}$相乘，然后加上$[-x]_{补}$进行校正。
  - 按补码进行运算
    - 按补码的规则进行移位：右移补1，符号位一起移
    - “乘数的补码$[y]_{补}$去掉符号位，当成一个正数与$[x]_{补}$相乘”——y仍然是补码
  - 符号位参与运算，自动生成——与原码的不同之处
- 考虑到运算时可能出现绝对值大于1的情形（但此刻并不是溢出），故部分积和被乘数取双符号位。

例

- 已知：$[x]_{补}$=1.0101，$[y]_{补}$=0.1101，求：$[x \cdot y]_{补}$。
- 解：因为乘数y>0，不用校正。

部分积	乘数	说明
00.0000 + 11.0101	1101	初值$[z_0]_{补}$=0 Y_4=1，+$[x]_{补}$
11.0101 11.1010	1110	→1位，得$[z_1]_{补}$，乘数同时→1位 Y3=0，不加$[x]_{补}$
11.1101 + 11.0101	0111	→1位，得$[z_2]_{补}$, 乘数同时→1位 Y_2=1，+$[x]_{补}$
11.0010 11.1001 + 11.0101	0011	→1位，得$[z_3]_{补}$，乘数同时→1位 Y_1=1，+$[x]_{补}$
10.1110 11.0111	0001	→1位，得$[z_4]_{补}$

故乘积$[x \cdot y]_{补}$=1.01110001

注意：符号位参加运算！——与原码不同

补码一位乘比较法（Booth算法）

- 现在广泛使用的是Booth算法，也称为比较法，其运算规则由校正法导出。
- 设被乘数$[x]_{补}=x_0.x_1x_2\cdots x_n$，乘数$[y]_{补}=y_0.y_1y_2\cdots y_n$

① 被乘数x符号任意，乘数y的符号为正时，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot y$$

② 被乘数x符号任意，乘数y的符号为负时，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot(0.y_1y_2\cdots y_n)+[-x]_{补}$$

综合①和②，被乘数x符号任意，乘数y符号任意，有：

$$[x\cdot y]_{补}=[x]_{补}\cdot[y]_{补}=[x]_{补}\cdot(0.y_1y_2\cdots y_n)-[x]_{补}\cdot y_0$$

前提：$[-x]_{补}=-[x]_{补}$

补码一位乘比较法（Booth算法）

$$[x\cdot y]_{补}=[x]_{补}(y_1 2^{-1}+y_2 2^{-2}+\cdots y_n 2^{-n})-[x]_{补}\,y_0$$

$$=[x]_{补}(-y_0+y_1 2^{-1}+y_2 2^{-2}+\cdots y_n 2^{-n})$$

$$=[x]_{补}[-y_0+(y_1-y_1 2^{-1})+(y_2 2^{-1}-y_2 2^{-2})+\cdots+(y_n 2^{-(n-1)}-y_n 2^{-n})]$$

$$=[x]_{补}[(y_1-y_0)+(y_2-y_1)2^{-1}+\cdots+(y_n-y_{n-1})2^{-(n-1)}+(0-y_n)2^{-n})]$$

$$=[x]_{补}[(y_1-y_0)+(y_2-y_1)2^{-1}+\cdots+(y_n-y_{n-1})2^{-(n-1)}+(y_{n+1}-y_n)2^{-n})]$$

其中$y_{n+1}=0$

如此，可得递推公式：

$$[z_0]_{补}=0$$

$$[z_1]_{补}=2^{-1}\{[z_0]_{补}+(y_{n+1}-y_n)[x]_{补}\}$$

$$[z_2]_{补}=2^{-1}\{[z_1]_{补}+(y_n-y_{n-1})[x]_{补}\}$$

$$\vdots$$

$$[z_i]_{补}=2^{-1}\{[z_{i-1}]_{补}+(y_{n-i+2}-y_{n-i+1})[x]_{补}\}$$

$$\vdots$$

$$[z_n]_{补}=2^{-1}\{[z_{n-1}]_{补}+(y_2-y_1)[x]_{补}\}$$

$$[x\cdot y]_{补}=[z_{n+1}]_{补}=[z_n]_{补}+(y_1-y_0)[x]_{补}$$

Booth算法规则

- 符号位参与计算
- 乘数最低位后增加一位附加位，初值置0
- 乘数的最低两位决定每一步的操作
- 右移，补码高位补1
- 共需n+1次累加，n次移位。第n+1次不移位

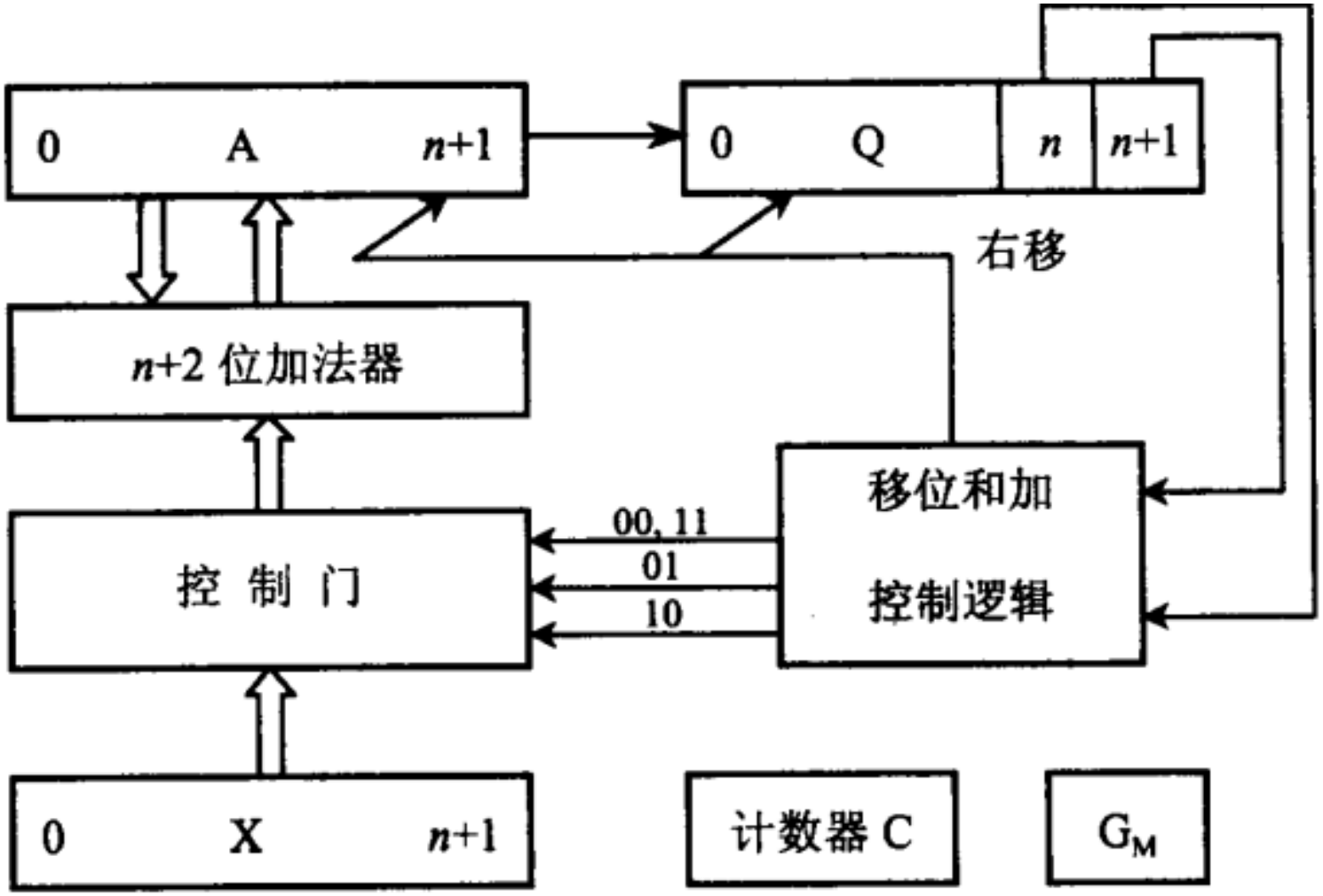
y_i	y_{i+1}	$y_{i+1} - y_i$	操作
0	0	0	部分积右移一位
0	1	1	部分积加$[x]_{补}$，再右移一位
1	0	-1	部分积减$[x]_{补}$，再右移一位
1	1	0	部分积右移一位

例

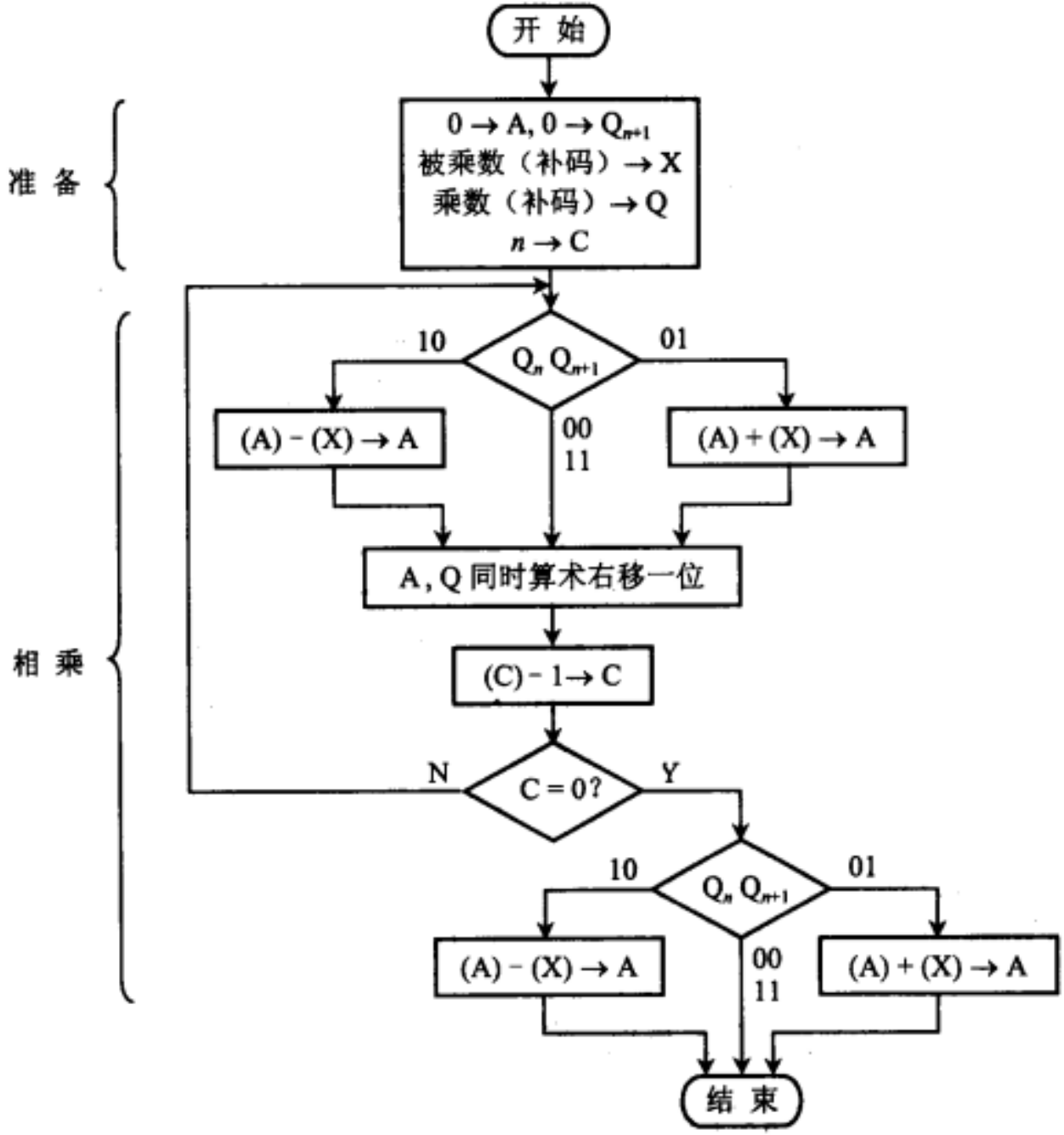
- 已知：$[x]_{补}=0.1101$，$[y]_{补}=0.1011$，求：$[x\cdot y]_{补}$。
- 解：如下表，可得$[x\cdot y]_{补}=0.10001111$

部分积	乘数y_n	附加位y_{n+1}	说明
00.0000 + 11.0011	01011	0	初值$[z_0]_{补}=0$ $Y_ny_{n+1}=10$，部分积加$[-x]_{补}$
11.0011 11.1001 11.1100 + 00.1101	10101 11010	1 1	→1位，得$[z_1]_{补}$ $Y_ny_{n+1}=11$，部分积→1位得$[z_2]_{补}$ $Y_ny_{n+1}=01$，部分积加$[x]_{补}$
00.1001 00.0100 + 11.0011	11 11101	0	→1位，得$[z_3]_{补}$ $Y_ny_{n+1}=10$，部分积加$[-x]_{补}$
11.0111 11.1011 + 00.1101	11110	1	→1位，得$[z_4]_{补}$ $Y_ny_{n+1}=01$，部分积加$[x]_{补}$
00.1000	1111		最后一步不移位，得$[x\cdot y]_{补}$

补码比较法（Booth算法）所需的硬件配置



补码一位乘比较法控制流程



补码一位乘法算法

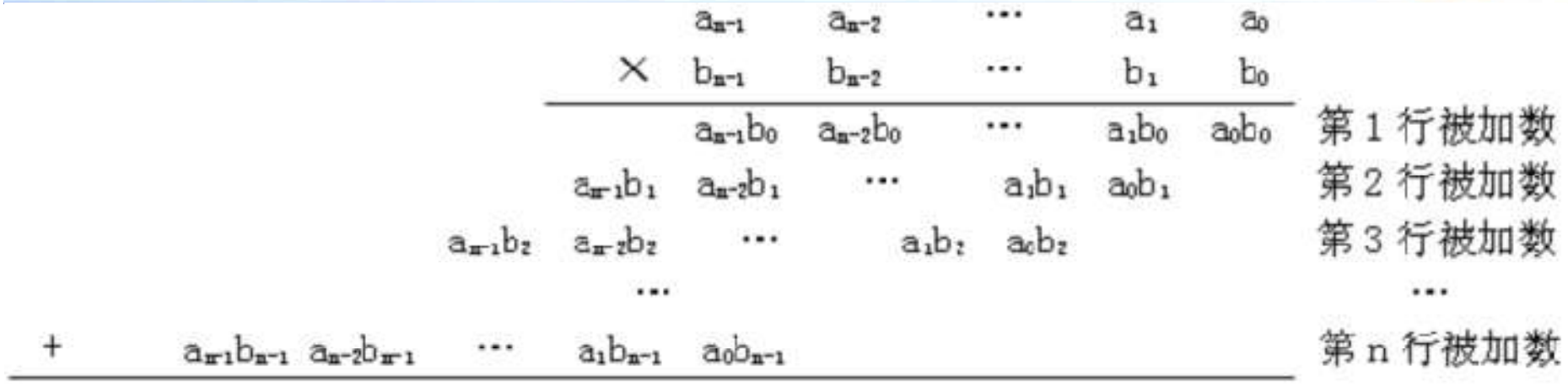
- *校正法：区分y的正负*
  - *乘数y的符号为负时，有：*

    $[x \cdot y]_{补}=[x]_{补} \cdot [y]_{补}=[x]_{补} \cdot (0.y_1y_2\ldots y_n)+[-x]_{补}$

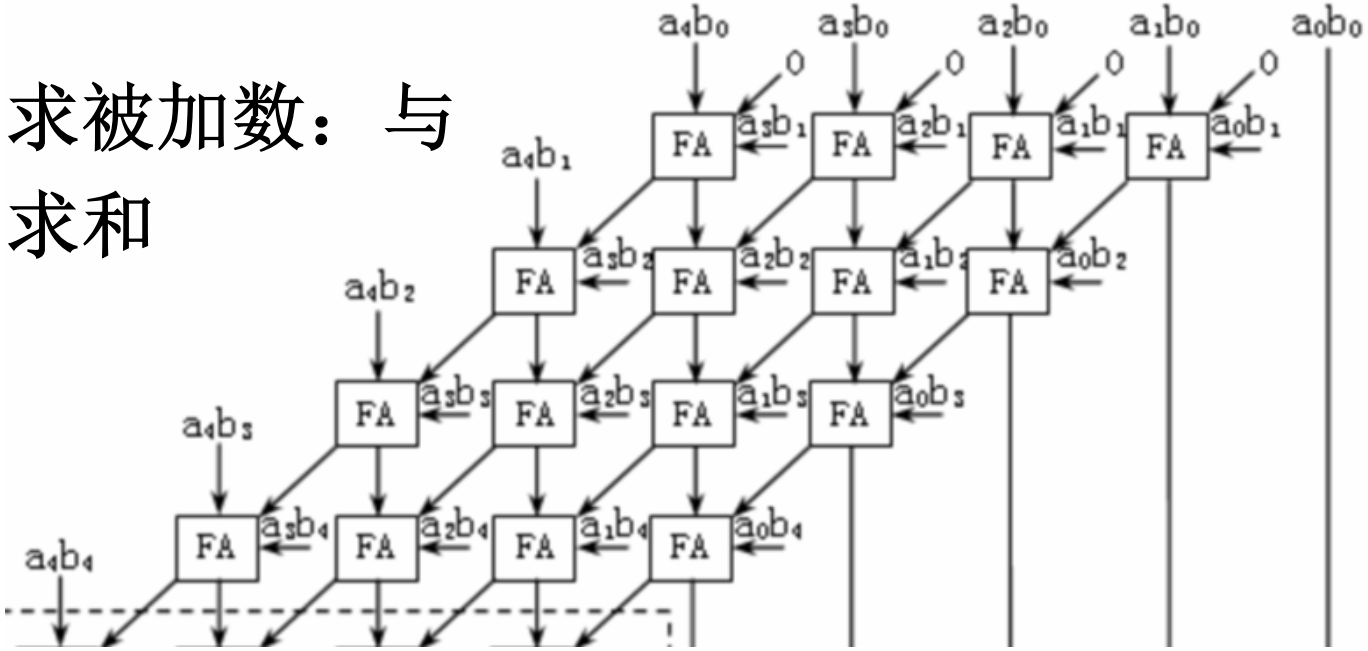
- 比较法（Booth算法）：不再区分y的正负
  - 操作的方式取决于表达式$(y_{i+1}-y_i)$的值。

y_i	y_{i+1}	$y_{i+1} - y_i$	操作
0	0	0	部分积右移一位
0	1	1	部分积加$[x]_{补}$，再右移一位
1	0	-1	部分积减$[x]_{补}$，再右移一位
1	1	0	部分积右移一位

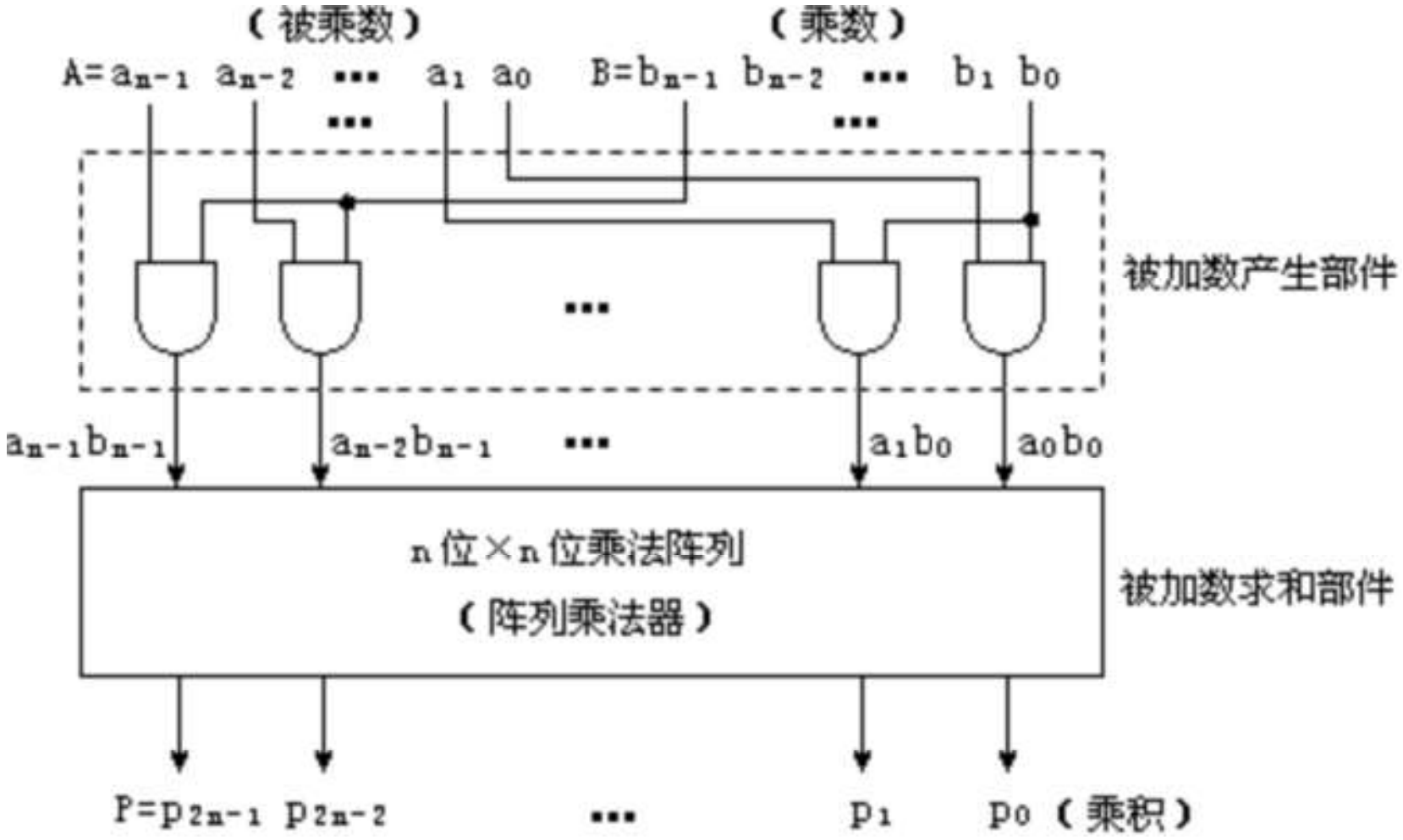
不带符号阵列乘法器



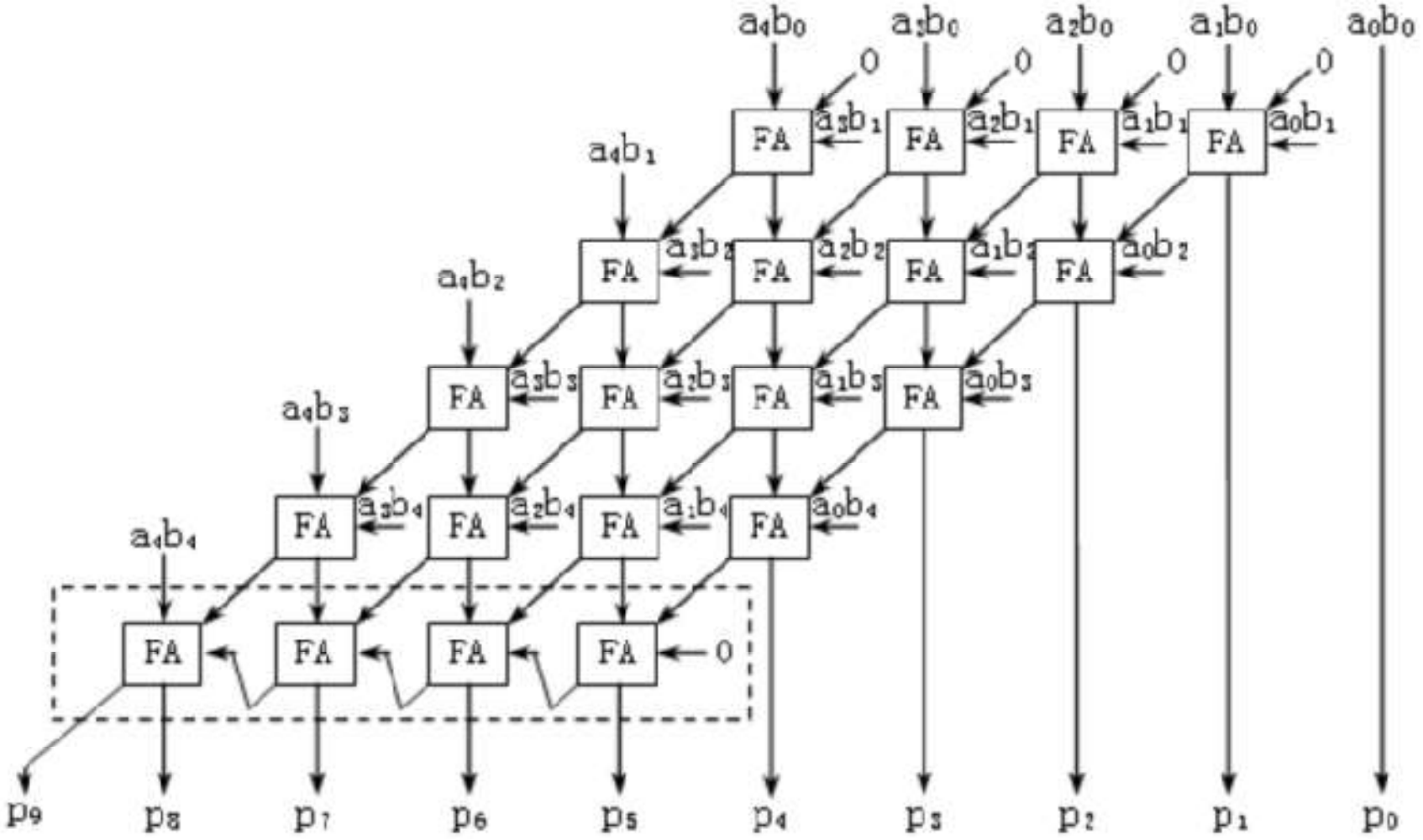
- 求被加数：与
- 求和



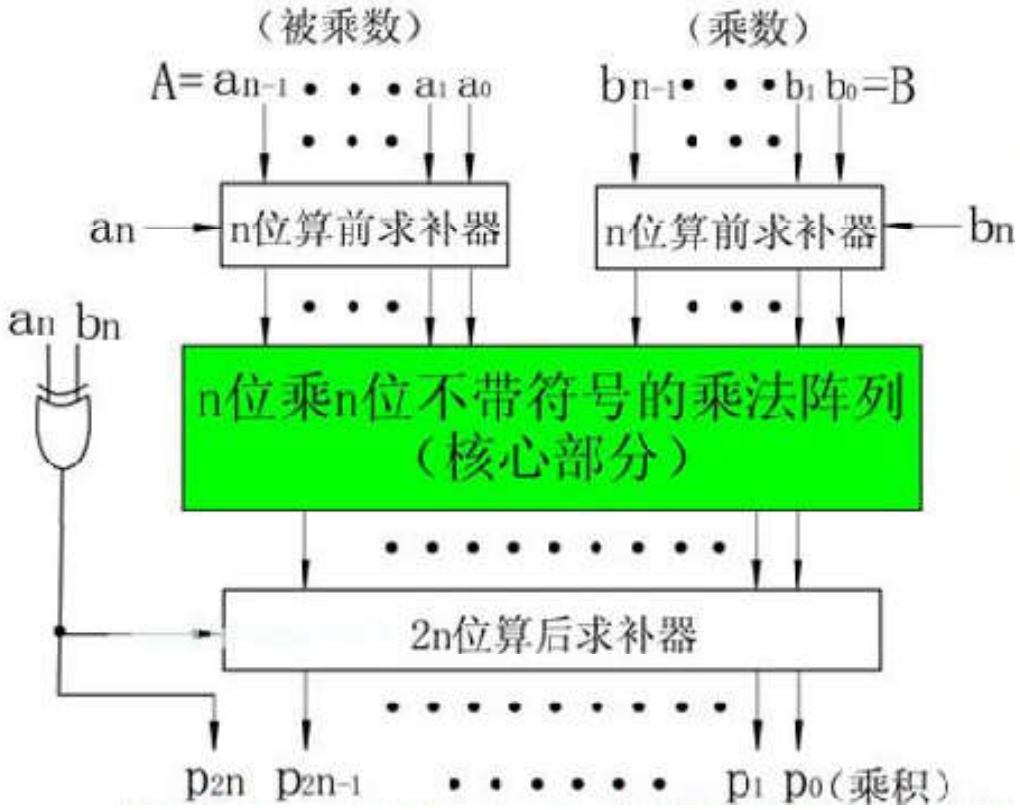
n位×n位不带符号的阵列乘法器



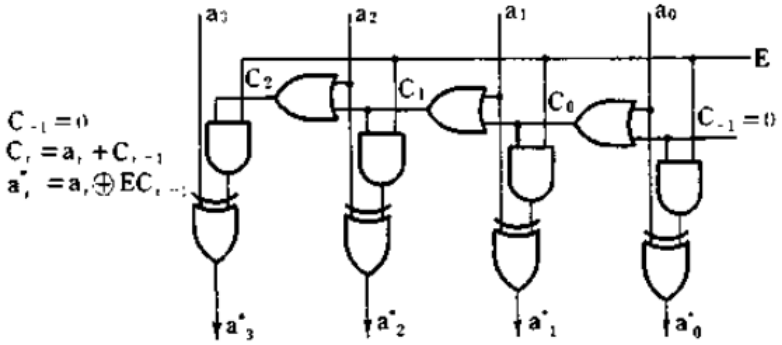
被加数求和阵列(串行进位)

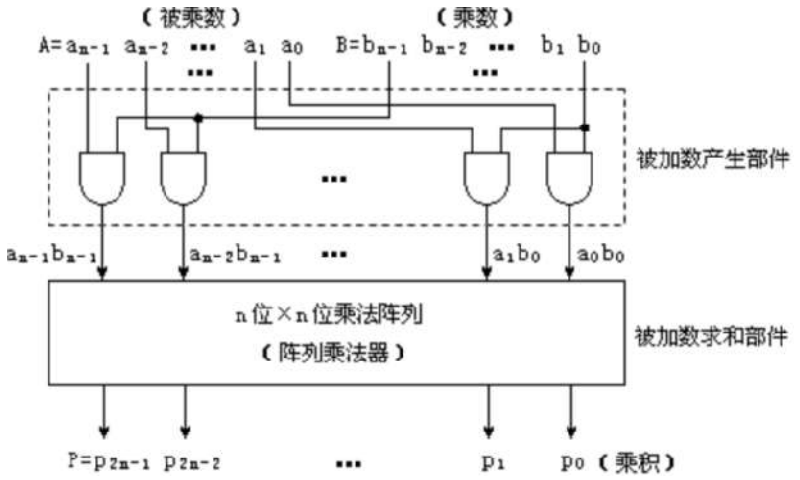


带符号阵列乘法器



求补器电路图





阵列乘法器（Array）性能改进

- 求和是乘法运算中最耗时步骤。
- 阵列乘法中部分积(Parial Product，PP)数目与乘数位数相等，决定了求和运算的次数。
  - 修正布斯算法(Modified Booth Algorithm，MBA)对乘数重新编码，以压缩PP。
  - 华莱士树(wallace tree，WT) 改变求和方式，将求和级数从O(N)降为O(logN) 。
    - 但WT结构不规整，布线困难。用4:2 compressor代替全加器(FA)，可以解决这问题。
  - 将MBA算法和WT结构的优点相结合，形成了MBA－WT乘法器。

除法运算

1. 分析笔算除法
2. 原码除法
   - 恢复余数法**Restoring division**
   - 加减交替法（不恢复余数法 **Non-restoring division**）
   - 原码加减交替法所需的硬件配置
   - 原码加减交替除法控制流程
3. 补码除法
   - *恢复余数法（少用，不讲）*
   - 补码加减交替法运算规则（浮点运算要用）
   - 补码加减交替法所需的硬件配置
   - 补码加减交替除法控制流程

1. 分析笔算除法

- 以小数为例，设 **x=-0.1011，y=0.1101，求x/y。**
- 商的符号：心算而得（负正得负）
- 数值运算：试商

$$
\begin{array}{r}
0.1101 \\
0.1101\overline{)\,0.10110} \\
\underline{0.01101} \qquad 2^{-1}\cdot y \\
0.010010 \\
\underline{0.001101} \qquad 2^{-2}\cdot y \\
0.00010100 \\
\underline{0.00001101} \qquad 2^{-4}\cdot y \\
0.00000111
\end{array}
$$

- 所以商**x/y=0.1101**，余数**=-0.00000111**

分析笔算除法

- 二进制除法实质是“作被除数（余数）和除数的减法，求新的余数”的过程

  ①每次上商都是由心算来比较余数(被除数)和除数的大小，确定商为1还是0。

  ②每做一次减法，总是保持余数不动，低位补0，再减去右移后的除数。

  ③余数为0时，或商的位数与操作数的位数相同时，停止计算。商符单独处理。

- 主要问题：

  ① 机器不能“心算”上商。

  ② 按照每次减法总是保持余数不动低位补0，再减去右移后的除数这一规则，则要求加法器的位数必须为除数的两倍。

  ③ 笔算求商时是从高位向低位。要求机器把每位商直接写到寄存器的不同位也是不可取的。

分析笔算除法

- 解决办法：

① 机器不能“心算”上商

  - 必须通过比较被除数(或余数)和除数绝对值的大小来确定商值，即|x|-|y|，若差为正(够减)上商1,差为负(不够减)上商0。

② **按照每次减法总是保持余数不动低位补0，再减去右移后的除数这一规则，则要求加法器的位数必须为除数的两倍。**

  - 右移除数可以用左移余数的办法代替，其运算结果是一样的，但对线路结构更有利。不过此刻所得到的余数不是真正的余数，只有将它乘上2^{-n}才是真正的余数。

③ **笔算求商时是从高位向低位逐位求的，而要求机器把每位商直接写到寄存器的不同位也是不可取的。**

  - 计算机可将每一位商直接写到寄存器的最低位，并把原来的部分商左移一位。

2. 原码除法

① 恢复余数法

② 加减交替法（不恢复余数法）

③ 原码加减交替法所需的硬件配置

④ 原码加减交替除法控制流程

原码除法

- 原码除法和原码乘法一样，符号位是单独处理的。
- 以小数为例，设

$$[x]_{原} = x_0.x_1x_2\cdots x_n \qquad [y]_{原} = y_0.y_1y_2\cdots y_n$$

$$[\frac{x}{y}]_{原} = (x_0 \oplus y_0) \bullet \frac{0.x_1x_2\cdots x_n}{0.y_1y_2\cdots y_n}$$

式中 为x的绝对值，记作x*

为y的绝对值，记作y*

- 即商符由两数符号位“异或”运算求得，商值由两数绝对值相除(x*/y*)求得。
- 小数定点除法对被除数和除数有一定的约束，即必须满足条件：0＜|被除数|≤|除数|

(一)恢复余数法

试商：商值的确定是通过比较被除数和除数的绝对值大小，即通过**x*-y***实现的：如果余数为正，说明“够减”，商上“1”；如果余数为负，说明“不够减”，商上“0”。

由于不够减而减了，必须将除数加回去，恢复成原来的余数。

机器内只设加法器，故需将**x*-y***操作变为$[x^*]_{补}+[-y^*]_{补}$的操作。因此，在判断“够减”，“不够减”作减法时，采用的是补码运算。

例

- 已知：**x=-0.1011,y=-0.1101,求：$[x/y]_{原}$**
- 解：由**x^*=0.1011，$[x]_{原}$=1.1011**

  y^*=0.1101，$[-y^*]_{补}$=1.0011,$[y]_{原}$=1.1101

  商值的求解过程如下：

被除数(余数)	商	说　明
0.1011 + 1.0011	0.0000	$+[-y^*]_{补}$（减去除数）
1.1110 + 0.1101	0	余数为负，上商0 恢复余数$+[y^*]_{补}$
0.1011 1.0110 + 1.0011	0	被恢复的被除数 ← 1位 $+[-y^*]_{补}$（减去除数）
0.1001 1.0010 + 1.0011	0 1 0 1	余数为正，上商1 ← 1位 $+[-y^*]_{补}$（减去除数）
0.0101 0.1010 + 1.0011	0 1 1 0 1 1	余数为正，上商1 ← 1位 $+[-y^*]_{补}$（减去除数）
1.1101 + 0.1101	0 1 1 0	余数为负，上商0 恢复余数$+[y^*]_{补}$
0.1010 1.0100 + 1.0011	0 1 1 0	被恢复的被除数 ← 1位 $+[-y^*]_{补}$（减去除数）
0.0111	0 1 1 0 1	余数为正，上商1

例（续）

- 故商值为**0.1101**
- 商的符号位为 $x_0 \oplus y_0 = 1 \oplus 1 = 0$, 故 $[x \div y]_{原} = 0.1101$
- 该例中，共上商**5**次

- 第一次上的商在商的整数位上，对小数除法而言，可用它作溢出判断。
  - 即当该位为“**1**”时，表示此除法为溢出，不能计算进行，应由程序进行处理，重新选择比例因子；
    - 采用定点数表示时，对于既有整数又有小数的原始数据，需要设定一个比例因子，数据按比例因子缩小成定点小数或扩大成定点整数再参加运算，结果输出时再按比例折算成实际值。
  - 当该位为“**0**”时，说明除法合法，可以进行计算。

(二)加减交替法

- 根据原码恢复余数法，有：
  - 当余数$R_i>0$时，可上商“1”，再对R_i左移一位后减除数，即$2R_i-y^*$。
  - 当余数$R_i<0$时，可上商“0”，然后再做R_i+y^*，即完成恢复余数的运算，再做左移和减除数，即$2(R_i+y^*)-y^*$，也即$2R_i+y^*$。
  - 因此，原码恢复余数法可归纳为：
    当余数$R_i>0$时，商上“1”，做$2R_i-y^*$的运算；
    当余数$R_i<0$时，商上“0”，做$2R_i+y^*$的运算。
- 这里已看不出余数的恢复问题了，而只是做加y^*或减y^*，因此，一般把它叫做加减交替法或不恢复余数法。

例

- 已知x=-0.1011,y=0.1101,求：$[x/y]_{原}$
- 解：由x*=0.1011，$[x]_{原}$=1.1011

  y*=0.1101，$[-y]_{补}$=1.0011, $[y]_{原}$=0.1101

商值的求解过程如下：

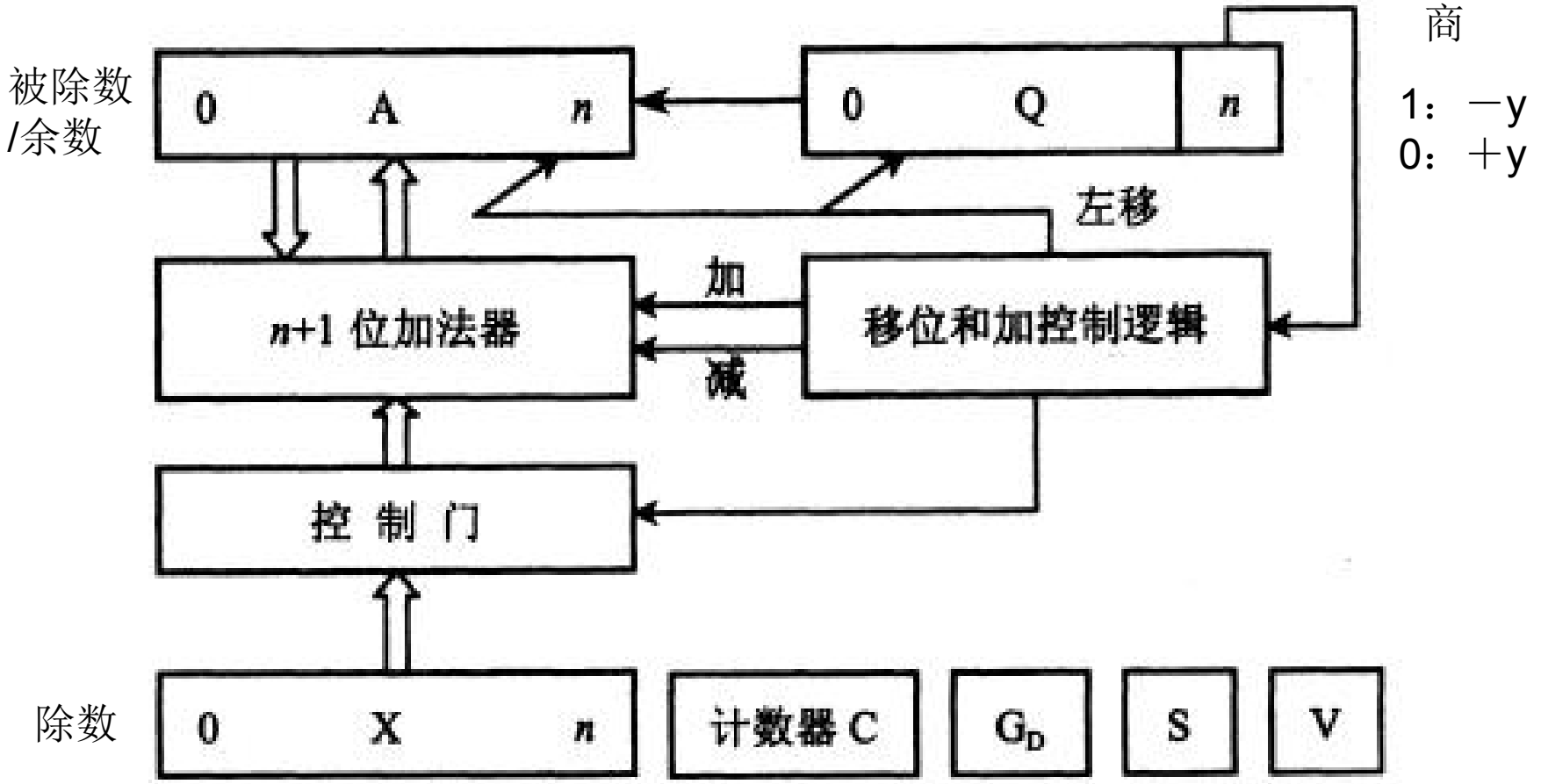
被除数(余数)	商	说　明
0.1011 + 1. 0011	0.0000	+$[-y^*]_{补}$（减除数）
1.1110 1.1100 + 0.1101	0 0	余数为负，上商0 ← 1位 +$[y^*]_{补}$ (加除数)
0.1001 1.0010 + 1.0011	0 1 0 1	余数为正，上商1 ← 1位 +$[-y^*]_{补}$（减除数）
0.0101 0.1010 + 1.0011	0 1 1 0 1 1	余数为正，上商1 ← 1位 +$[-y^*]_{补}$（减除数）
1.1101 1.1010 + 0.1101	0 1 1 0 0 1 1 0	余数为负，上商0 ← 1位 +$[y^*]_{补}$ (加除数)
0.0111	0 1 1 0 1	余数为正，上商1

例（续）

- 商的符号位为 $x_0 \oplus y_0 = 1 \oplus 0 = 1$
- 所以$[x/y]_{原}$=1.1101

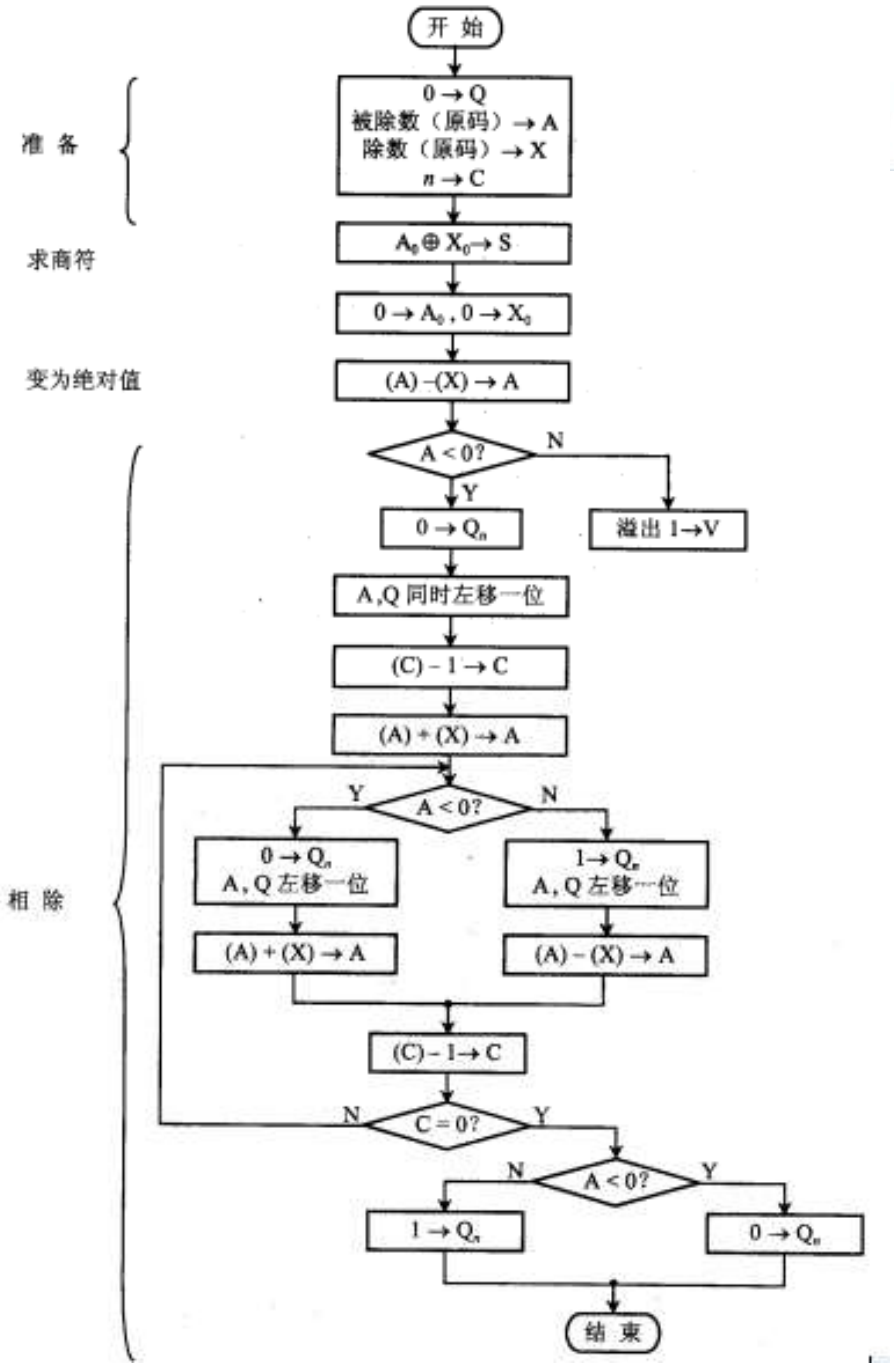
- 分析此例可见，n位小数的除法共上商n+1次，第一次商用来判断是否溢出。
- 倘若比例因子选择恰当，除数结果不溢出，则第一次商肯定是0。如果省去这位商，只需上商n次即可，此时除法运算一开始应将被除数左移一位减去除数，然后再根据余数上商。

原码加减交替法所需的硬件配置



被除数字长可以是除数的两倍，开始时其低位放在Q中，逐步左移到A中

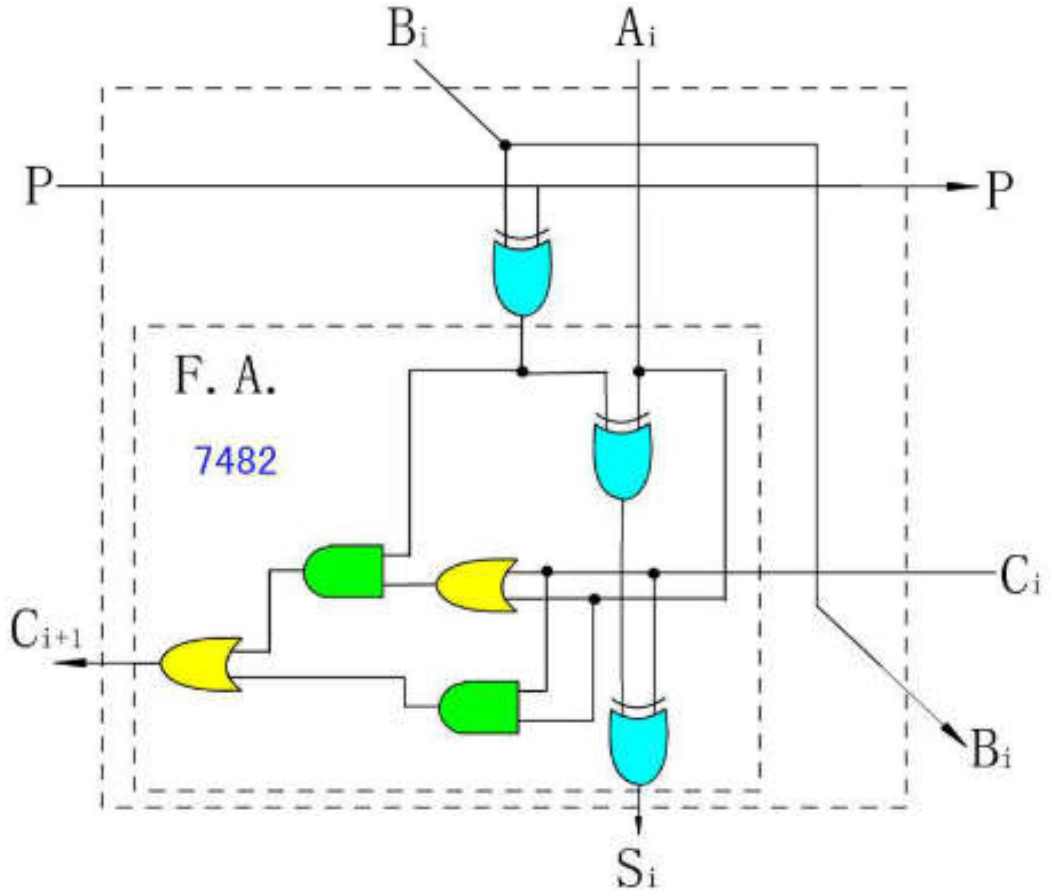
原码加减交替除法控制流程



阵列除法器—不恢复余数阵列除法器

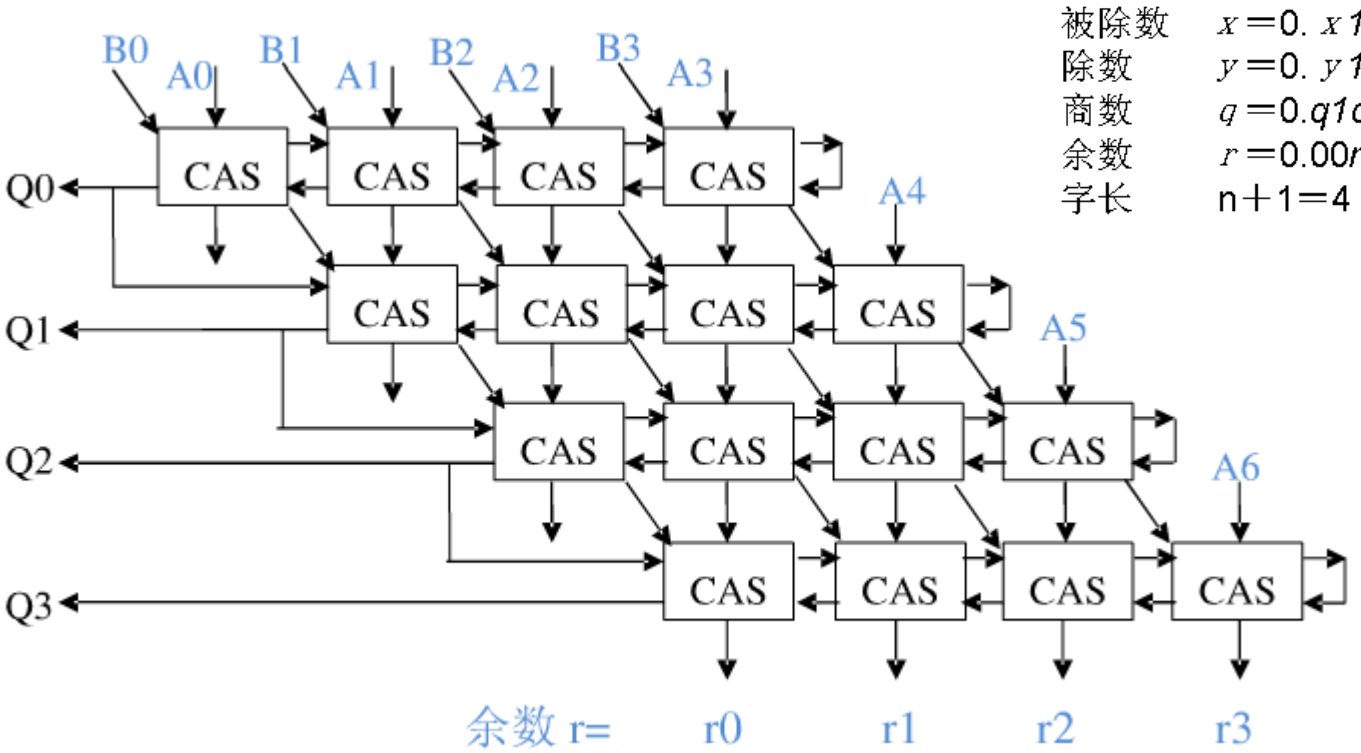
- 可控加法/减法单元(CAS)
  - 当输入线P＝0时，加法运算；当P＝1时，减法运算。

输入：A、B
结果：S

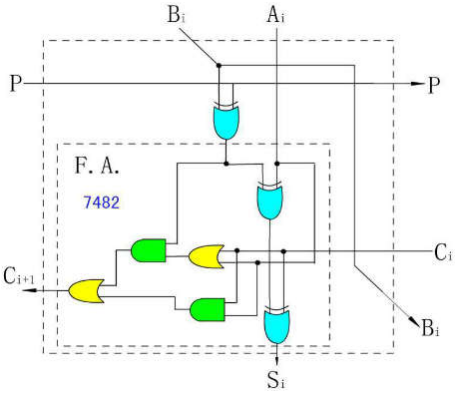


阵列除法器（加减交替）

- 每一行所执行的操作究竟是加法还是减法，取决于前一行输出的符号与被除数的符号是否一致。
  - 当不够减时，部分余数相对于被除数来说要改变符号。
    - 这时应该产生一个商位“0”，除数首先沿对角线右移，然后加到下一行的部分余数上。
  - 当部分余数不改变它的符号时，即产生商位“1”，下一行的操作应该是减法。



被除数　$x=0.x1\,x2\,x3\,x4\,x5\,x6$ (双倍长)
除数　$y=0.y1\,y2\,y3$
商数　$q=0.q1q2q3$
余数　$r=0.00r3r4r5r6$
字长　n+1=4



3.补码除法

① 恢复余数法（不讲）

② 补码加减交替法运算规则

③ 补码加减交替法所需的硬件配置

④ 补码加减交替除法控制流程

补码加减交替法运算规则

- 补码除法其符号位和数值部分是一起参加运算的。
- 主要需解决三个问题：
  a) 如何确定商值；
  b) 如何形成商符；
  c) 如何获得新的余数。

如何确定商值

- 比较被除数(余数)和除数的大小。
  - 操作数都为补码，不能简单地用$[x]_{补}$(或余数$[R_i]_{补}$) 减去$[y]_{补}$。
  - 实质上是比较它们所对应的绝对值的大小。

比较$[x]_{补}$与$[y]_{补}$的符号	求余数	比较 $[R_i]_{补}$与$[y]_{补}$的符号
同号	$[x]_{补}-[y]_{补}$	同号，表示“够减”
异号	$[x]_{补}+[y]_{补}$	异号，表示“够减”

- 商值的确定。 “末位恒置一”

$[x]_{补}$与$[y]_{补}$	商	$[R]_{补}$与$[y]_{补}$	商值
同号	正	同号，表示“够减”	1
		异号，表示“不够减”	0
异号	负	异号，表示“够减”	0
		同号，表示“不够减”	1

$[R]_{补}$与$[y]_{补}$	商值
同号	1
异号	0

如何形成商符

- 补码除法中，商符是在求商的过程中自动形成的。
- 在小数定点除法中，被除数的绝对值必须小于除数的绝对值，否则商大于1而溢出。因此，
  - 当$[x]_{补}$与$[y]_{补}$同号时，$[x]_{补}-[y]_{补}$所得的余数$[R_0]_{补}$必然与$[y]_{补}$异号，商上“0”，恰好与商的符号(正)一致；
  - 当$[x]_{补}$与$[y]_{补}$异号时，$[x]_{补}+[y]_{补}$所得的余数$[R_0]_{补}$必然与$[y]_{补}$同号，商上“1”，这也与商的符号(负)一致。
- 此外，商的符号还可用来判断商是否溢出。
  - 当$[x]_{补}$与$[y]_{补}$同号时，若$[R_0]_{补}$与$[y]_{补}$同号，上商“1”，即溢出。
  - 当$[x]_{补}$与$[y]_{补}$异号时，若$[R_0]_{补}$与$[y]_{补}$异号，上商“0”，即溢出。

如何获得新的余数

- 新余数$[R_{i+1}]_{补}$的获得方法与原码加减交替法极相似，其算法规则为：

  当$[R_i]_{补}$与$[y]_{补}$同号时，商上“1”，新余数

  $[R_{i+1}]_{补}=2[R_i]_{补}-[y]_{补}=2[R_i]_{补}+[-y]_{补}$

  当$[R_i]_{补}$与$[y]_{补}$异号时，商上“0”，新余数

  $[R_{i+1}]_{补}=2[R_i]_{补}+[y]_{补}$

- 将此法列于下表：

$[R_i]_{补}$与$[y]_{补}$	商	新余数$[R_{i+1}]_{补}$
同号	1	$[R_{i+1}]_{补}=2[R_i]_{补}+[-y]_{补}$
异号	0	$[R_{i+1}]_{补}=2[R_i]_{补}+[y]_{补}$

•如果对商的精度没有特殊要求，一般可采用“末位恒置1”法，这种方法操作简单，易于实现，而且最大误差仅为2^{-n}。

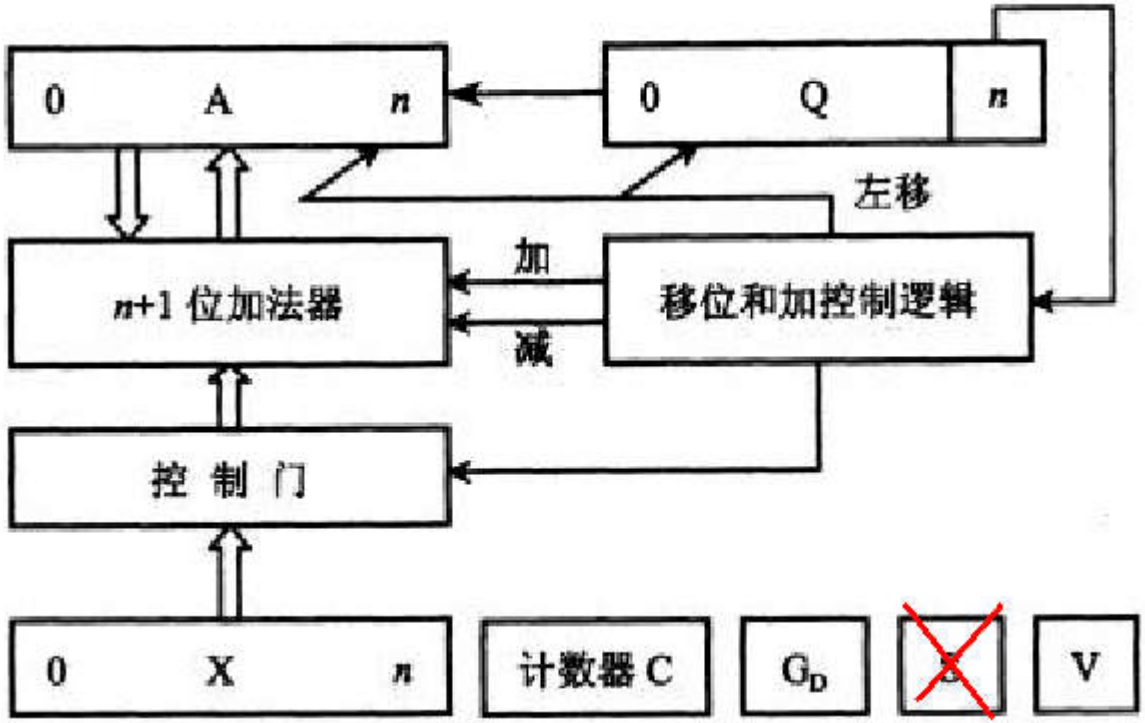
例

- 已知：x=-0.1001, y=+0.1101 求： $[x/y]_{补}$
- 解：$[x]_{补}$=1.0111,$[y]_{补}$=0.1101,$[-y]_{补}$=1.0011
  运算过程如下，所以所以$[x / y]_{补}$=1.0101

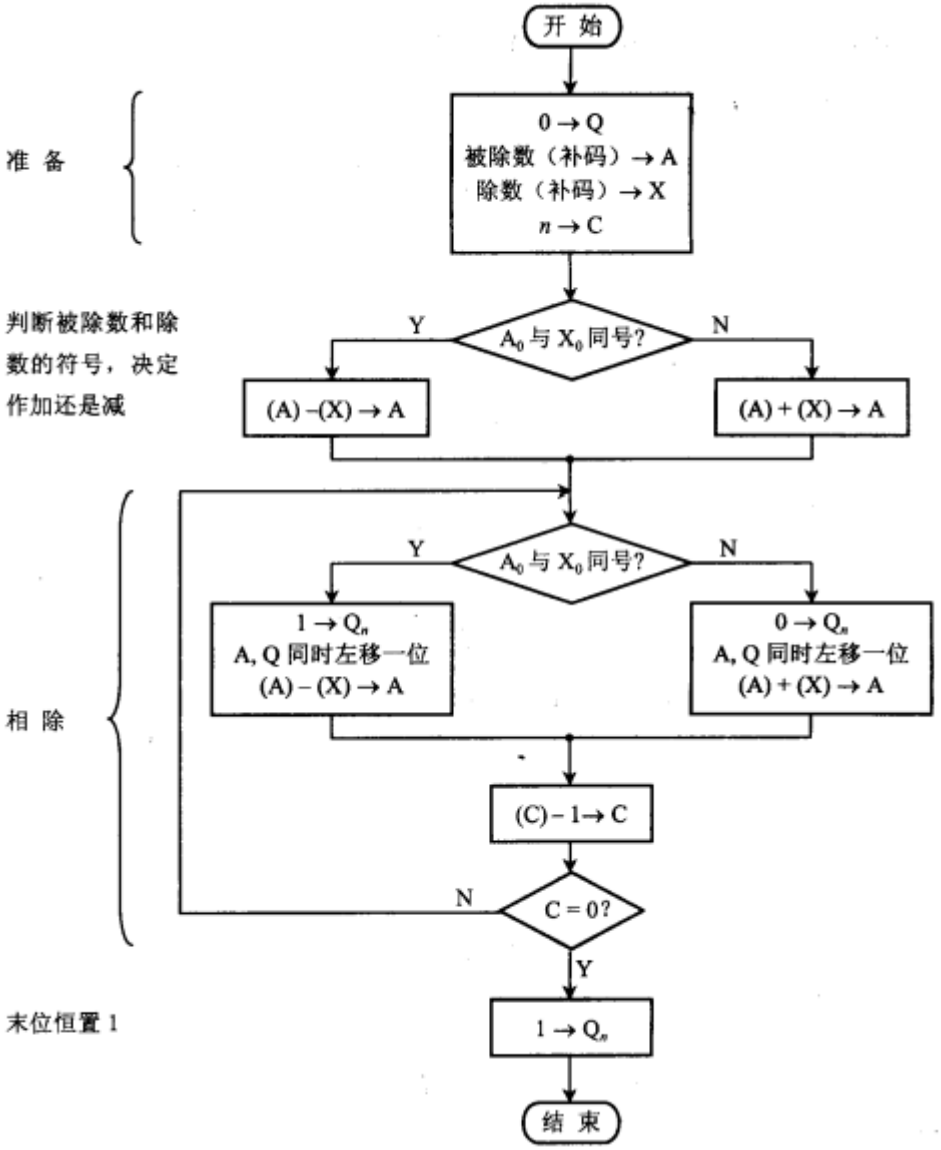
被除数（余数）	商 上商	说 明
1.0111 + 0.1101	0.0000	 $[x]_{补}$与$[y]_{补}$异号，+$[y]_{补}$
0.0100 0.1000 + 1.0011	1 1	$[R]_{补}$与$[y]_{补}$同号，上商1 ← 1位 +$[-y]_{补}$
1.1011 1.0110 + 0.1101	1 0 1 0	$[R]_{补}$与$[y]_{补}$异号，上商0 ← 1位 +$[y]_{补}$
0.0011 0.0110 + 1.0011	1 0 1 1 0 1	$[R]_{补}$与$[y]_{补}$同号，上商1 ← 1位 +$[-y]_{补}$
1.1001 1.0010	1 0 1 0 1 0 1 0 1	$[R]_{补}$与$[y]_{补}$异号，上商0 ← 1位,末位商恒置“1”

补码加减交替法所需的硬件配置

- 补码加减交替法所需的硬件配置基本上与原码加减交替法所需的硬件配置相似。
- 因为补码除法的商符在运算中自动形成，其中的**S**触发器可以省掉。



补码加减交替除法控制流程



①图中未画出补码除法溢出判断的内容；②按流程图所示，多作一次加(或减)法，其实末位恒置“1”前，只需移位不必作加(或减)法；③与原码除一样，图中均未指出对0进行检测，实际上在除法运算前，先检测被除数和除数是否为0，若被除数为0，结果即为0；若除数为0，结果为无穷大，这两种情况都无需继续作除法运算；④为了节省时间，上商和移位操作可以同时进行。

整数除法

- 算法与小数除法相同
- 初始条件
  - **0<|除数|<=|被除数|**

小结

- 移位
  - 算术移位、逻辑移位
- 加、减
  - 补码
  - 溢出判断（1位符号位、两位符号位）
- 乘
  - 原码一位乘
  - 补码（校正法、比较法（Booth法））
- 除
  - 原码（恢复余数法、不恢复余数法（加减交替法））
  - 补码（加减交替法）
- 阵列乘法器、阵列除法器

- 符号位是否参与运算？
- 如何判断溢出？

- **6.14、 6.20（1）、 6.21（2）、 6.23、 6.29（1）、 6.30（2）**