计算机组成原理与系统结构

第四章

运算方法与运算器

http://jpkc.hdu.edu.cn/computer/zcyl/dzkjdx/

杭州電子科技大學
HANGZHOU DIANZI UNIVERSITY

本章概要

- 运算器
  - 算数运算
  - 逻辑运算
- 定点数运算
  - 加减乘除运算方法
  - 定点运算器的组成与结构
- 浮点数运算
  - 加减乘除运算方法
  - 浮点运算器的组成与结构

第4章 运算方法与运算器

BACK

4.1 定点数的加减运算及实现

一 补码加减运算与运算器

二 机器数的移位运算

三 移码加减运算与判溢

四 十进制加法运算

一、补码加减运算与运算器

一 补码加减运算方法

二 补码加减运算的溢出判断

三 补码加减运算器的实现

补码定义

模：计量器具的容量，或称为模数。4位字长的机器表示的二进制整数为：

0000~1111 共16种状态，模为16= 2^4 。

n位定点整数的补码机器数（n+1位）的模值为2^{n+1}，一位符号位的纯小数的模值为2。

补码的定义：正数的补码就是正数的本身，负数的补码是原负数加上模。

n位定点小数X（其补码机器数包含符号位在内为n+1位）：

$$[X]_{补}=\begin{cases} X & 1-2^{-n}\geqslant X\geqslant 0 \\ 2+X & 0>X\geqslant -1 \end{cases}$$

$$[X]_{补}=2+X \quad (\mathrm{mod}\ 2)$$

n位定点整数X（其补码机器数包含符号位在内为n+1位）：

$$[x]_{补}=\begin{cases} X & 2^{n}-1\geqslant X\geqslant 0 \\ 2^{n+1}+X & 0>X\geqslant -2^{n} \end{cases}$$

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

1、补码加减运算方法

- 补码的加减运算的公式是：
  - $[X+Y]_{补} = [X]_{补} + [Y]_{补}$
  - $[X-Y]_{补} = [X]_{补} + [-Y]_{补}$
- 特点：
  - 使用补码进行加减运算，**符号位和数值位一样参加运算**。
  - **补码的减法可以用加法来实现**，任意两数之差的补码等于被减数的补码与减数相反数的补码之和 。

求补运算：$[Y]_{补} \rightarrow [-Y]_{补}$

- 求补规则：将$[Y]_{补}$包括符号位在内每一位取反，末位加1。
- 若$[Y]_{补} = Y0, Y1 \ldots\ldots Yn$，则：

$$[-Y]_{补} = \overline{Y_0}\,\overline{Y_1}\cdots\cdots\overline{Y_n} + 1$$

- 若$[Y]_{补} = Y0.Y1 \ldots\ldots Yn$，则：

$$[-Y]_{补} = \overline{Y_0}\,\overline{Y_1}\cdots\cdots\overline{Y_n} + 0.0\cdots\cdots01$$

- 例：$[X]_{补} = 0.1101$，则：$[-X]_{补} =$ 1.0011
- $[Y]_{补} = 1.1101$，则：$[-Y]_{补} =$ 0.0011

补码加法运算举例

例：X=+0.1011，Y＝-0.0101，求X+Y

解：　$[X]_{补}$= 0.1011 ，　$[Y]_{补}$= 1.1011

$$
\begin{array}{rll}
 & [X]_{补} & 0.1011 \\
+ & [Y]_{补} & 1.1011 \\
\hline
 & [X+Y]_{补} & 10.0110
\end{array}
$$

所以， X+Y= +0.0110

补码加法的特点：

① 符号位作为数的一部分参加运算，符号位的进位丢掉。

② 运算结果为补码形式。

补码减法运算举例

【例】 已知$[x]_{补}=1.1010$，$[y]_{补}=1.0110$，求$[x-y]_{补}$。

解： $[x-y]_{补}=[x]_{补}+[-y]_{补}$

$[-y]_{补}=0.1010$

$\therefore \quad [x-y]_{补}=1.1010+0.1010$

$=10.0100=0.0100 \ (\mathrm{mod}\ 2)$

用真值运算并加以验算

$\because \quad x=-0.0110 \quad y=-0.1010$

$\therefore \ x-y=-0.0110-(-0.1010)$

$=0.0100$

补码加减运算举例

- 例：已知X=+1011，Y=−0100，用补码计算X+Y和X−Y。
  - 写出补码：

$[X]_{补}$ =0, 1011　　$[Y]_{补}$ =1, 1100

$[-Y]_{补}$ =0, 0100

  - 计算：

$$\begin{array}{r} 0,1011 \\ +\ 1,1100 \\ \hline 0,0111 \end{array}$$

$[X+Y]_{补}$ = 0, 0111

$$\begin{array}{r} 0,1011 \\ +\ 0,0100 \\ \hline 0,1111 \end{array}$$

$[X-Y]_{补}$ = 0, 1111

补码加减运算的规则可归纳如下：

① 参加运算的操作数均为补码表示的形式；

② 加减运算可统一为加法运算进行，符号位作为数的一部分参加运算，符号位的进位去掉；

③ 运算结果为补码形式。

2、补码加减运算的溢出判断

- 当运算结果超出机器数的表示范围时，称为溢出。计算机必须具备检测运算结果是否发生溢出的能力，否则会得到错误的结果。

【例】 设字长为8位(包括1位符号)，若十进制数 x= –73，y= –83，用二进制补码求$[x+y]_{补}$。

解：$x=(-73)_{10}=(-1001001)_2$，$[x]_{补}=10110111$

$y=(-83)_{10}=(-1010011)_2$，$[y]_{补}=10101101$

则 $[x+y]_{补}=10110111+10101101$

$=01100100$ （溢出）

2、补码加减运算的溢出判断

- 对于加减运算，可能发生溢出的情况：同号（两数）相加，或者异号（两数）相减。
- 确定发生溢出的情况：
  - 正数相加，且结果符号位为1；（教材P105 例4.2（1））
  - 负数相加，且结果符号位为0；（教材P105 例4.2（2））
  - 负数－正数，且结果符号位为0；（教材P105 例4.2（3））
  - 正数－负数，且结果符号位为1；（教材P105 例4.2（4））

常用的判溢方法（补码加减运算）

（1）单符号位的判溢

两操作数同号且和数的符号与操作数的符号不同。

先以补码加法为例，

设两操作数 $[A]_{补}=a_s.\,a_1a_2\ldots a_n$，$[B]_{补}=b_s.\,b_1b_2\ldots b_n$

和数 $[S]_{补}=S_s.\,S_1S_2\ldots S_n$，则：

$$OF=\overline{a_s}\cdot\overline{b_s}\cdot S_s+a_s\cdot b_s\cdot\overline{S_s}=(a_s\oplus S_s)(b_s\oplus S_s)$$

综合加减法时，见教材P106式（4.3）。

注意，减法时，因为Y必须取反加1，所以在式（4.3）中的减法部分跟加法部分相比，Y_f取反。

式（4.3）容易理解，但硬件实现稍显复杂。

常用的判溢方法（补码加减运算）

（2）进位判溢方法

根据教材P106 表4.2可以得到结论：

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

C_n—最高位（符号位）进位，C_{n-1}—次高位（最高有效位）进位。当OF=1时溢出。

也就是，当最高有效位产生的进位(C_{n-1})和符号位产生的进位(C_n)不同时，加减运算发生了溢出。

例如：x=+0.1011, y=+0.1001, 求x+y

解： $[x]_{补}=0.1011$ ，　$[y]_{补}=0.1001$

$$
\begin{array}{rll}
 & [x]_{补} & 0.1011 \\
+ & [y]_{补} & 0.1001 \\
\hline
 & [x+y]_{补} & 1.0100
\end{array}
$$

可见：两个正数相加的结果为负数，显然是由于溢出造成的错误。

常用的判溢方法（补码加减运算）

- （3）双符号位判溢方法
  - X和Y采用双符号位补码参加运算，正数的双符号位为00，负数的双符号位为11；当运算结果的两位符号S_{f1} S_{f2}不同时（01或10），发生溢出。
  - $V = S_{f1} \oplus S_{f2} = X_f \oplus Y_f \oplus C_f \oplus S_f$
  - S_{f1} S_{f2}=01，则正溢出；S_{f1} S_{f2}=10，则负溢出。

但无论溢出与否，第一符号位s_{f1}始终表示正确的数符。

双符号位判溢方法举例

- 例：用补码计算X+Y和X−Y
  - （1）X=+1000，Y=+1001
  - （2）X=−1000，Y=1001

	$[X]_{补}$	00，1000
+	$[Y]_{补}$	00，1001
	$[X+Y]_{补}$	01，0001

$S_{f1}\ S_{f2}$=01，正溢出，或
C_f=0,C_1=1,不同，溢出

	$[X]_{补}$	00，1000
+	$[-Y]_{补}$	11，0111
	$[X-Y]_{补}$	11，1111

$S_{f1}\ S_{f2}$=11，无溢出，或
C_f=0,C_1=0,相同，无溢出

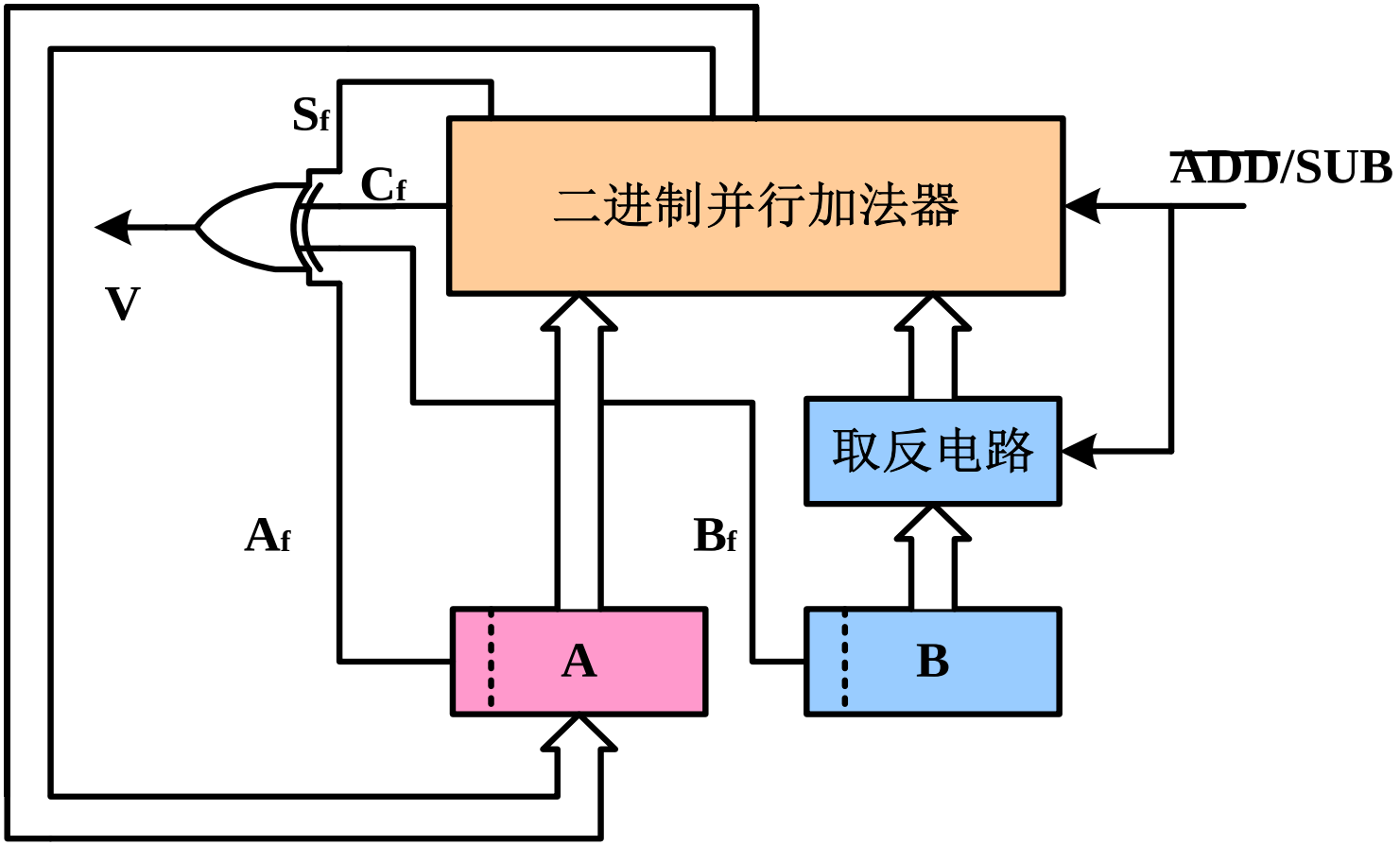
	$[X]_{补}$	11，1000
+	$[Y]_{补}$	00，1001
	$[X+Y]_{补}$	00，0001

$S_{f1}\ S_{f2}$=00，无溢出，或
C_f=1,C_1=1,相同，无溢出

	$[X]_{补}$	11，1000
+	$[-Y]_{补}$	11，0111
	$[X-Y]_{补}$	10，1111

$S_{f1}\ S_{f2}$=10，负溢出，或
C_f=1,C_1=0,不同，溢出

3、补码加减运算器的实现



3、补码加减运算器的实现

- **核心部件**：一个普通的二进制并行加法器。
- **A**：累加器，存放$[X]_{补}$；**B**：寄存器，存放$[Y]_{补}$；
- **取反电路**:实际上为一组两输入异或门电路。
- $\overline{ADD/SUB}$ =0时，补码加法器，将B寄存器直接送入并行加法器；
- $\overline{ADD/SUB}$ =1时，补码减法器，将$\overline{B}$送入并行加法器，同时，并行加法器的最低位产生进位，即B取反加1，此时并行加法器的运算相当于$[A]_{补}$加$[-B]_{补}$ ，完成减法运算。

二、机器数的移位运算

- 二进制数据（真值）每相对于小数点左移一位，相当于乘以2；每相对于小数点右移一位，相当于除以2。
- 计算机中的移位运算分为：
  - 1、逻辑移位：将移位的数据视为无符号数据，各数据位在位置上发生了变化，导致无符号数据的数值（无正负）放大或缩小。逻辑左移时，高位移出，低位补“0”；逻辑右移时，低位移出，高位补“0”。移出的数据位一般置入标志位CF（进位/借位标志）。
  - 2、算术移位：将移位的数据视为带符号数据（机器数）。算术移位的结果，在数值的绝对值上进行放大或缩小，同时，符号位必须要保持不变。
  - 3、循环移位：所有的数据位在自身范围内进行左移或者右移，左移时最高位移入最低位，右移时最低位移入最高位。若与CF标志位一起循环，称为大循环，否则，称为小循环。

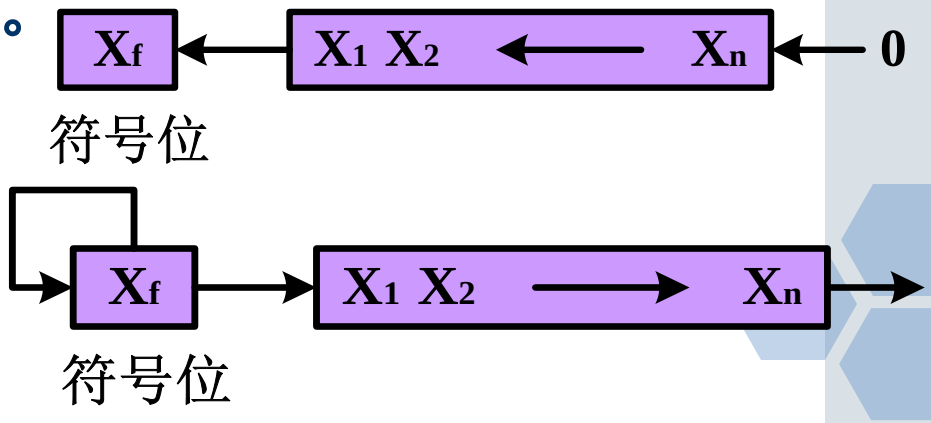
原码、反码的算术移位

- 原码的算术左移：符号位不变，高位移出，低位补0。
  - 当左移移出的数据位为“1”时，发生溢出。
- 原码的算术右移：符号位不变，低位移出，高位补0。

- 反码的算术左移：最高有效位移入符号位，低位正数补0，负数补“1”。
- 反码的算术右移：符号位不变，高位补符号位，低位移出。

补码的算术移位

- **补码的算术左移：符号位不变，高位移出，低位补0。**
  - 当左移移出的数据位正数为“1”、负数为“0”时，发生溢出。
  - 为保证补码算术左移时不发生溢出，**移位的数据最高有效位必须与符号位相同**。
  - **在不发生溢出的前提下，**用硬件实现补码的算术左移时，直接将数据最高有效位移入符号位，不会改变机器数的符号。
- **补码的算术右移：符号位不变，低位移出，高位正数补0，负数补1，即高位补符号位。**



补码的算术移位举例

- 例：设X＝0.1001, Y＝－0.0101，求
  - $[X]_{补}$＝ 0.1001
  - $[2X]_{补}$＝ 1.0010（溢出）
  - $[X/2]_{补}$＝ 0.0100
  - $[Y]_{补}$＝ 1.1011
  - $[2Y]_{补}$＝ 1.0110
  - $[Y/2]_{补}$＝ 1.1101
- 原码、反码的算术移位举例见教材P110 例4.4。

三、移码加减运算与判溢

- 为什么要进行移码运算？
  - 移码经常作为浮点数的阶码
- 移码主要运算
  - 加减

三、移码加减运算与判溢

- 移码和移码计算

$$[X]_{移} + [Y]_{移} = 2^n + X + 2^n + Y = 2^n + (2^n + X + Y) = 2^n + [X+Y]_{移}$$

$$[X]_{移} + [-Y]_{移} = 2^n + [X-Y]_{移}$$

- 上面两式表明：

1）使用移码求和，只需直接将移码相加，并将结果的符号位取反；

2）使用移码求差，直接将被减数移码与减数相反数的移码相加，并将结果的符号位取反；

三、移码加减运算与判溢

- **移码和补码混合计算**

$$[X]_{移} + [Y]_{补} = 2^n + X + 2^{n+1} + Y = 2^{n+1} + (2^n + X + Y) = 2^{n+1} + [X+Y]_{移} = [X+Y]_{移} \quad (\mathrm{mod}\ 2^{n+1})$$

$$[X-Y]_{移} = [X]_{移} + [-Y]_{补} \quad (\mathrm{mod}\ 2^{n+1})$$

- **移码运算结果判溢：**

第一操作数采用移码表示，使用双符号位，规定最高符号位为0）；

第二操作数采用补码表示，使用变形补码。

$$\begin{array}{r} 0\ X_f\ X_1\ X_2 \ldots\ldots X_n \\ +\ Y_f\ Y_f\ Y_1\ Y_2 \ldots\ldots Y_n \\ \hline S_{f1}\ S_{f2}\ S_1\ S_2 \ldots\ldots S_n \end{array}$$

三、移码加减运算与判溢

- 移码运算结果溢出的判断条件是：
  - 当结果的最高符号位S_{f1}=1时溢出，S_{f1}=0时结果正确。
    - S_{f1} S_{f2}=10时，结果正溢出；
    - S_{f1} S_{f2}=11时，结果负溢出。
  - 由于移码运算用于浮点数的阶码，当运算结果正溢出时，浮点数上溢；当运算结果负溢出时，浮点数下溢，当作机器零处理。

三、移码加减运算与判溢

- [例] 已知x=1011，y=-1110，用移码运算方法计算x+y，同时指出运算结果是否发生溢出。

解：$[x]_{补}=011011$，$[y]_{补}=110010$

$$
\begin{array}{rll}
 & [x]_{补} & 011011 \\
+ & [y]_{补} & 110010 \\
\hline
 & [x+y]_{补} & 001101
\end{array}
$$

运算结果未发生溢出

所以 $x+y=-0011$

- [例] 已知x=1001，y=-1100，用移码运算方法计算x-y，同时指出运算结果是否发生溢出。

解：$[x]_{补}=011001$，$[-y]_{补}=001100$

$$
\begin{array}{rll}
 & [x]_{补} & 011001 \\
+ & [-y]_{补} & 001100 \\
\hline
 & [x-y]_{补} & 100101
\end{array}
$$

运算结果发生正溢

- 例，教材P111例4.5

四、十进制加法运算

计算机中实现十进制加法的方法：

① 直接用十进制加法器实现

② 用二进制加法指令和十进制修正指令实现

十进制加法器——实现两个十进制数求和的电路。

1. 十进制加法器的主要特点

① 采用BCD码；

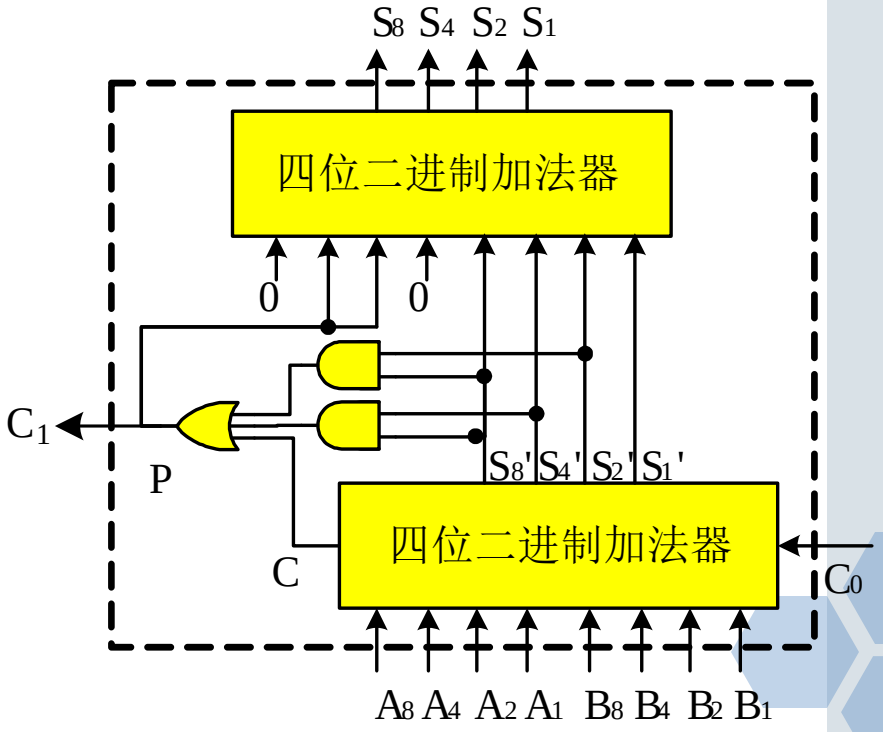
② 十进制位内按二进制加法规则运算，十进制位间按“逢十进一”规则运算。

四、十进制加法运算

- 计算机中的十进制加法器通常采用BCD码设计，在二进制加法器的基础上，加上适当的校正电路，可以实现BCD码的加法器。
- 对于8421BCD码来说，当相加的两数之和S>9时，加6校正；当S≤9时，且无进位时，结果正确，不需校正。

2．十进制加法器的组成

每位十进制加法器可由4位二进制加法器，和数修正及进位形成电路组成 。



四、十进制加法运算

【例】 设被加数x=$(25)_{10}$，加数y=$(68)_{10}$，用十进制加法求x+y，要求写出BCD码执行相加的过程。

解： x的BCD码为0010 0101，y的BCD码为0110 1000，

0010 0101　被加数x

+ 0110 1000　加数　y

1000 1101

+ 1 110　和数个位加6，并向十位进1

1001 0011　和数为93

∴ $x+y=93_{10}$

4.2 定点数的乘法运算及实现

一 原码乘法及实现

二 补码乘法及实现

三 阵列乘法器

4.2 定点数的乘法运算及实现

- 由于计算机的软硬件在逻辑上具有一定的等价性，因此实现乘除法运算，可以有三种方式：
- Ⅰ.用软件实现。
  - **硬件上**：设计简单，没有乘法器和除法器。
  - **指令系统**：没有乘除指令，但有加/减法和移位指令
  - **实现**：乘除运算通过编制一段子程序来实现
  - **算法**：程序中运用串行乘除运算算法，循环累加、右移指令→乘法，循环减、左移指令→除法。
  - **运算速度**：较慢。
  - **适用场合**：单片机。

4.2 定点数的乘法运算及实现

- II. 用硬件乘法器和除法器实现。
  - **硬件上**：设置有并行加法器、移位器和若干循环、计数控制逻辑电路搭成的**串行乘除法器**。
  - **指令系统**：具有乘除法指令。
  - **实现**：乘除运算通过**微程序一级（硬件＋微程序）**来实现。
  - **算法**：在微程序中**依据串行乘除运算算法**，循环累加、右移指令→乘法，循环减、左移指令→除法。
  - **运算速度**：有所提高，但硬件设计也相对复杂。
  - **适用场合**：低性能CPU。

4.2 定点数的乘法运算及实现

- Ⅲ. 用高速的阵列乘法器和阵列除法器来实现。
  - 硬件上：设置有专用的、并行运算的阵列乘法器和阵列除法器。
  - 指令系统：具有乘除法指令。
  - 实现：完全通过硬件来实现。
  - 算法：并行乘/除法。
  - 运算速度：很快，但硬件设计相当复杂。
  - 适用场合：高性能CPU。

一、原码乘法及实现

1、手工乘法算法

- 手工计算1011×1101，步骤：
- **手工算法**：对应每1位乘数求得1项位积，并将位积逐位左移，然后将所有的位积一次相加，得到最后的乘积。
- 手工算法在机器中实现存在的问题：
  - ① n位乘n位则有2n位字长的乘积，需2n位宽的、具有n个输入的加法器；
  - ② 机器一次运算只能完成两个数相加。
- **乘法的机器算法**：从乘数的最低位开始，每次根据乘数位得到其**位积**，乘数位为0，位积为0，乘数位为1，则位积为被乘数；用**原部分积右移1位加上本次位积**，得新部分积；初始部分积为0；循环累加右移n次（乘数的位数）。

```
      1011
×     1101
------------
      1011
     0000
    1011
   1011
------------
  10001111
```

二、原码乘法算法

❖2、原码一位乘法算法：

假设$[X]_{原}=X_S\ X_1\ X_2\ \ldots\ldots X_n$，$[Y]_{原}=Y_S\ Y_1\ Y_2\ \ldots\ldots Y_n$，$P=X\cdot Y$，$P_S$是积的符号：

- 符号位单独处理　$Ps=Xs \oplus Ys$
- 绝对值进行数值运算　$|P|=|X|*|Y|$

❖例如：X=+1011，Y=−1101，用原码一位乘法计算P=X·Y。

举例

例如：X=+1011，Y=-1101，用原码一位乘法计算P=X·Y。

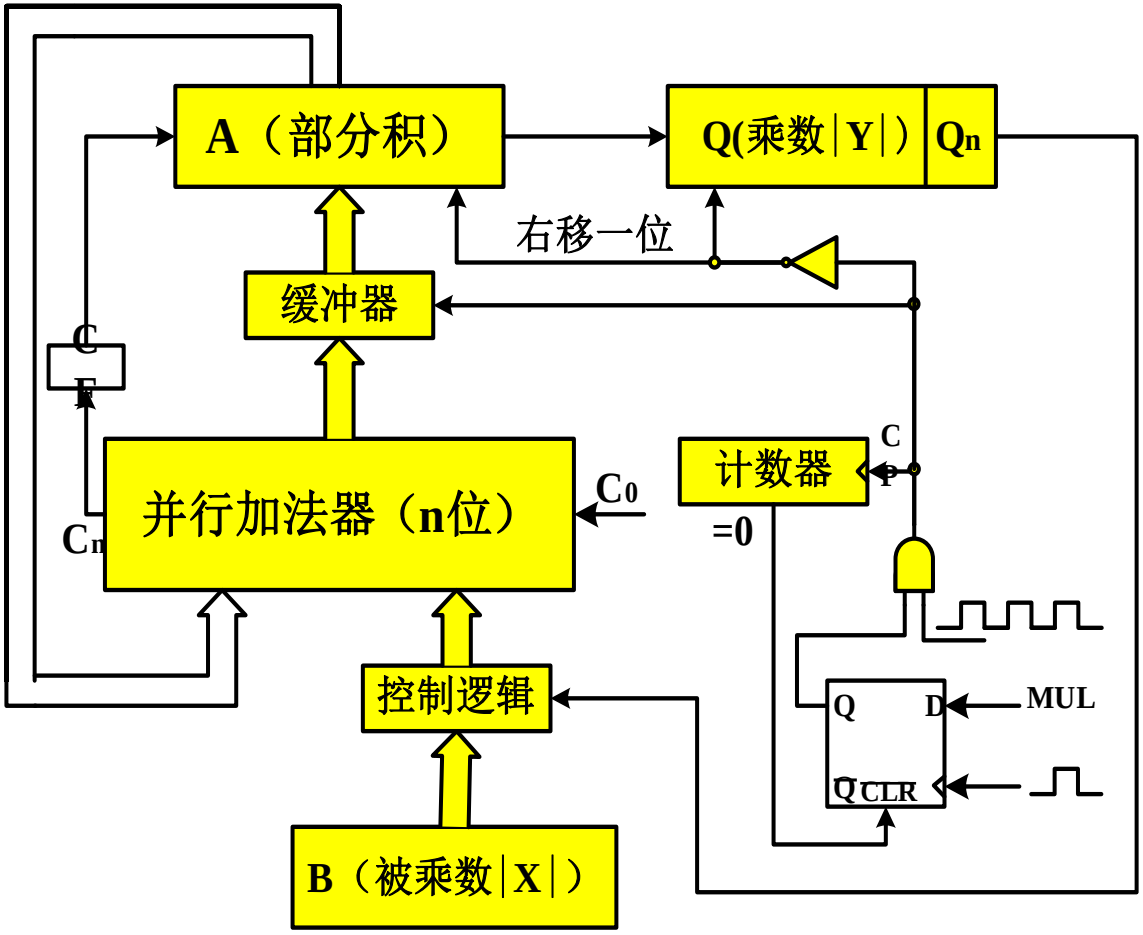
- $[X]_{原}$=0，1011
- $[Y]_{原}$=1，1101
- Ps=Xs⊕Ys
  =0⊕1=1
- |P| = |X|·|Y|

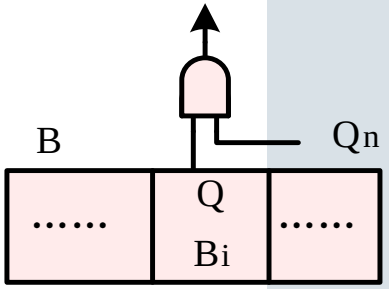
$[P]_{原}$=1，10001111

部分积	乘数Y	操作说明
0，0000	1 1 0 1	
+ 0，1011		Y_4=1，+|X|
0，1011		
0，0101	1 1 1 0	右移一位
+ 0，0000		Y_3=0，+0
0，0101		
0，0010	1 1 1 1	右移一位
+ 0，1011		Y_2=1，+|X|
0，1101		
0，0110	1 1 1 1	右移一位
+ 0，1011		Y_1=1，+|X|
1，0001		
0，1000	1 1 1 1	右移一位

一、原码乘法及实现

3、原码乘法的硬件实现



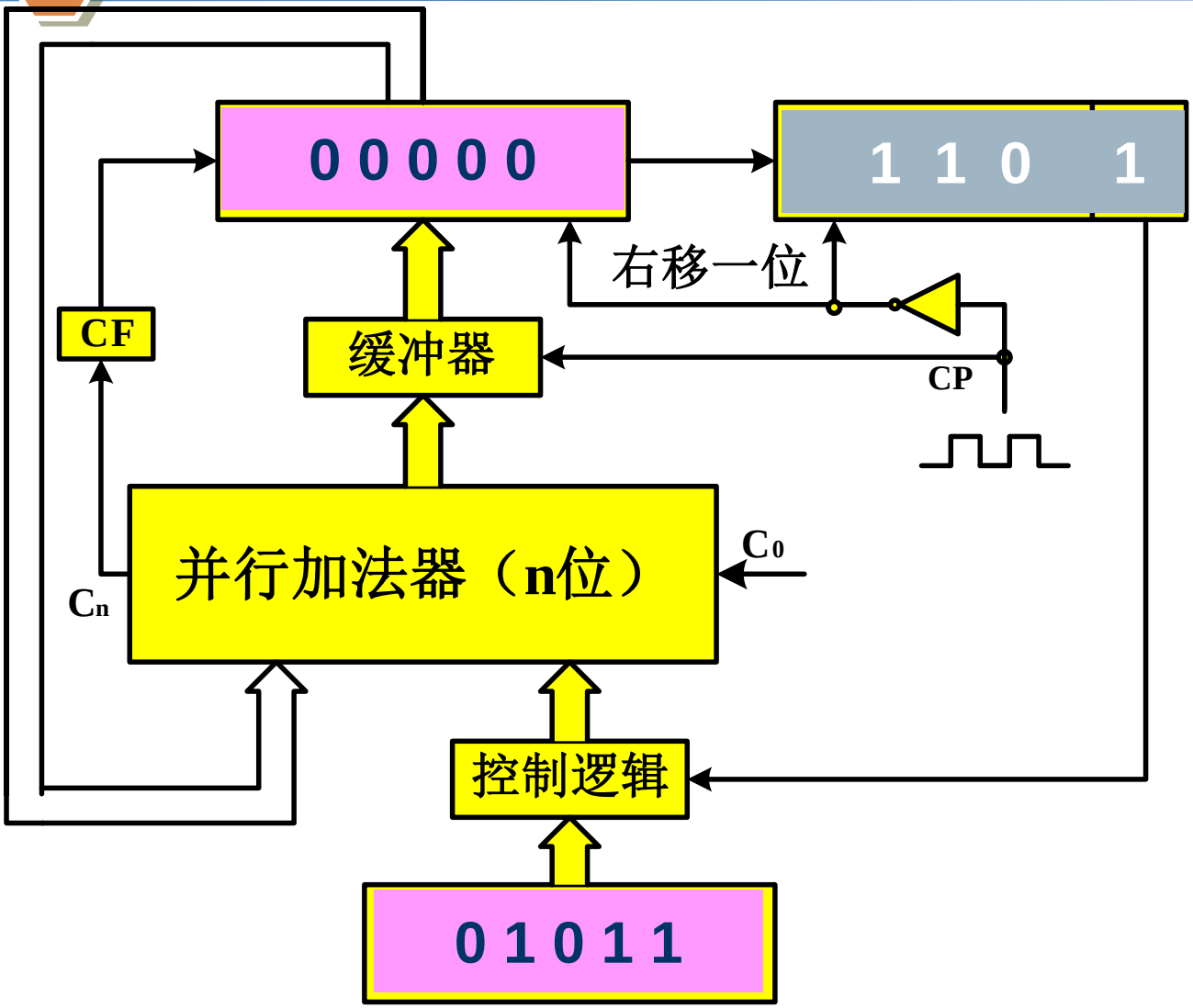


控制逻辑电路

原码一位乘法

为各寄存器给初值

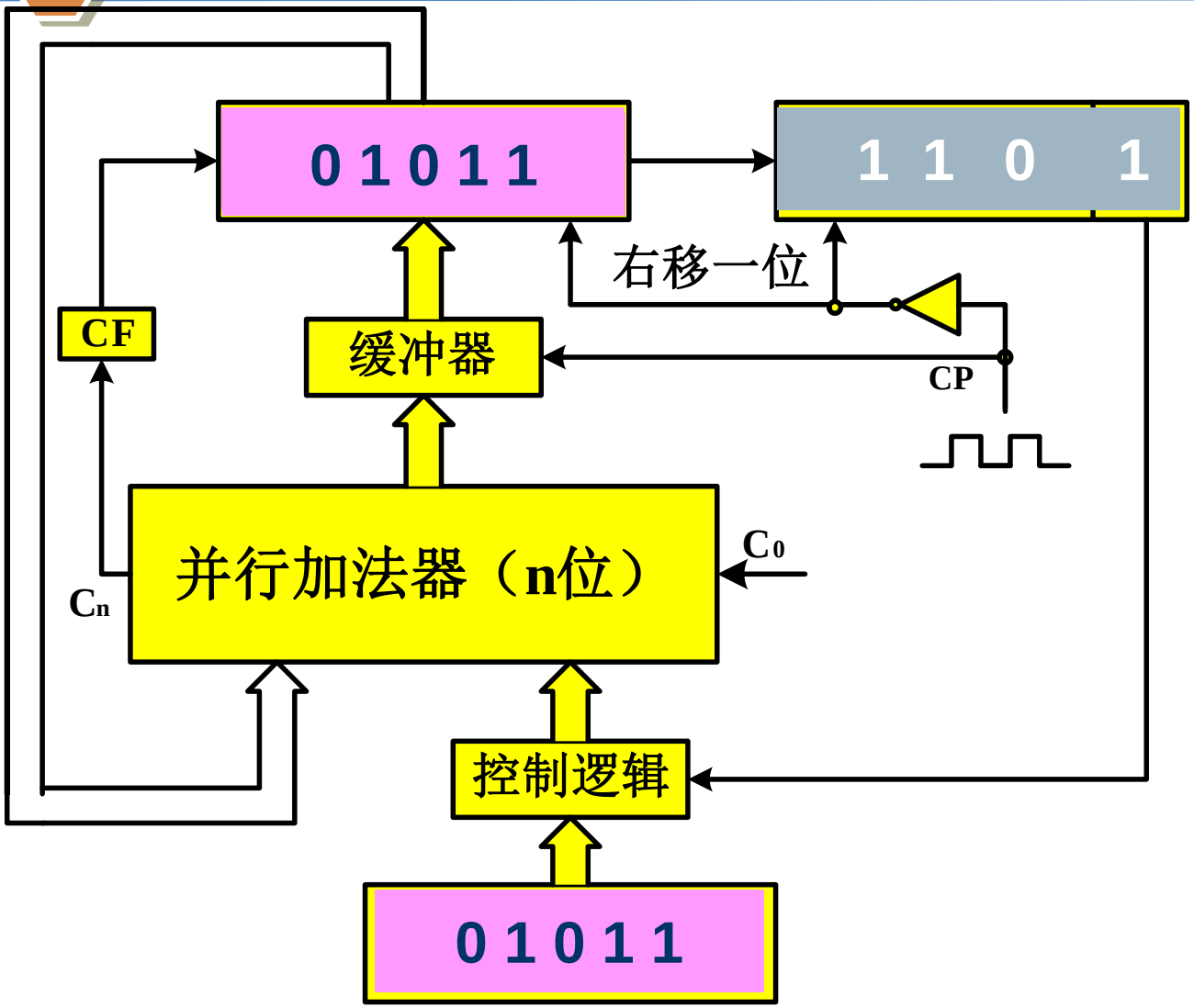
00000 1101



第一次求部分积　加运算：＋|X|

00000　1101

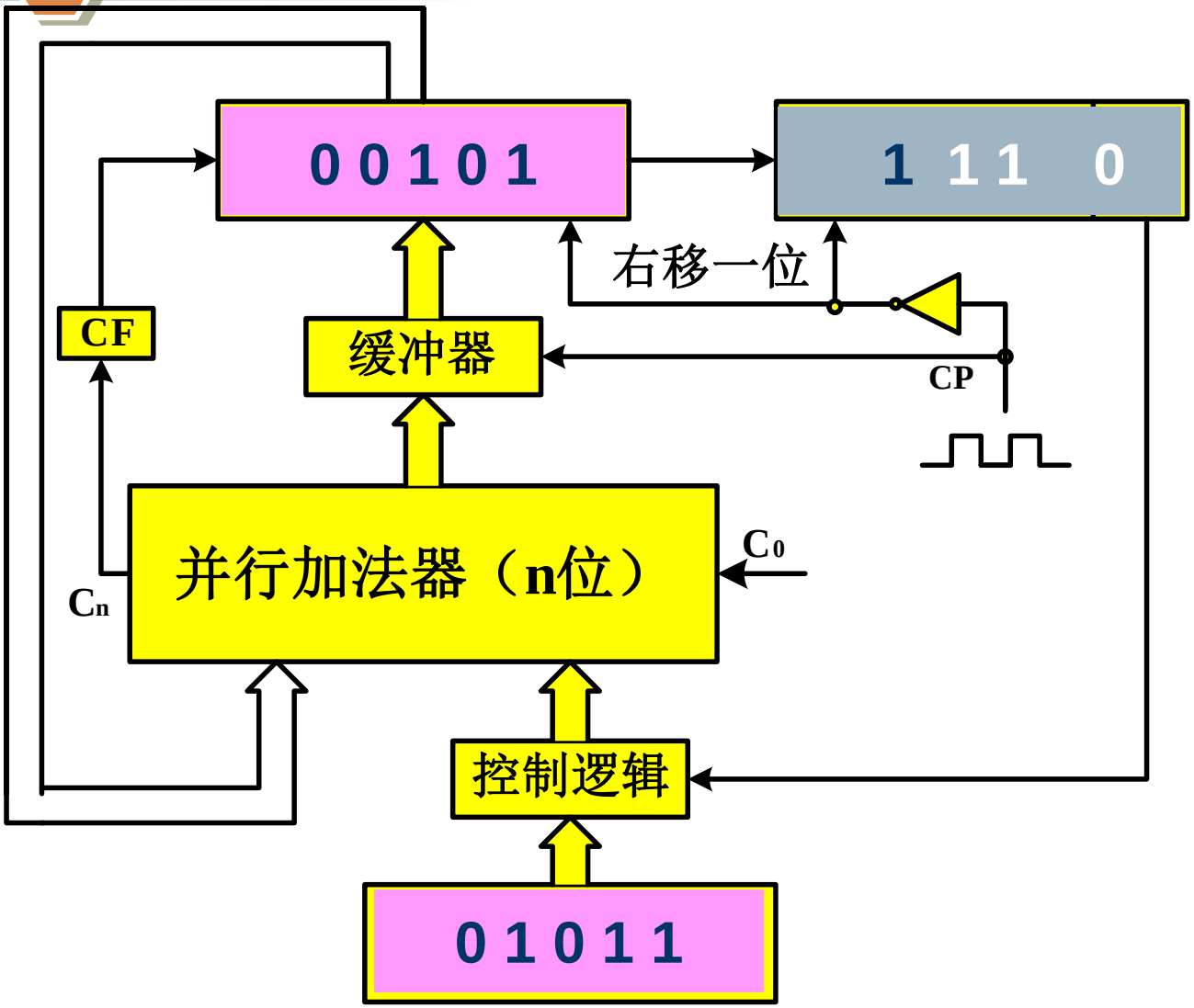
01011　1101



第一次求部分积

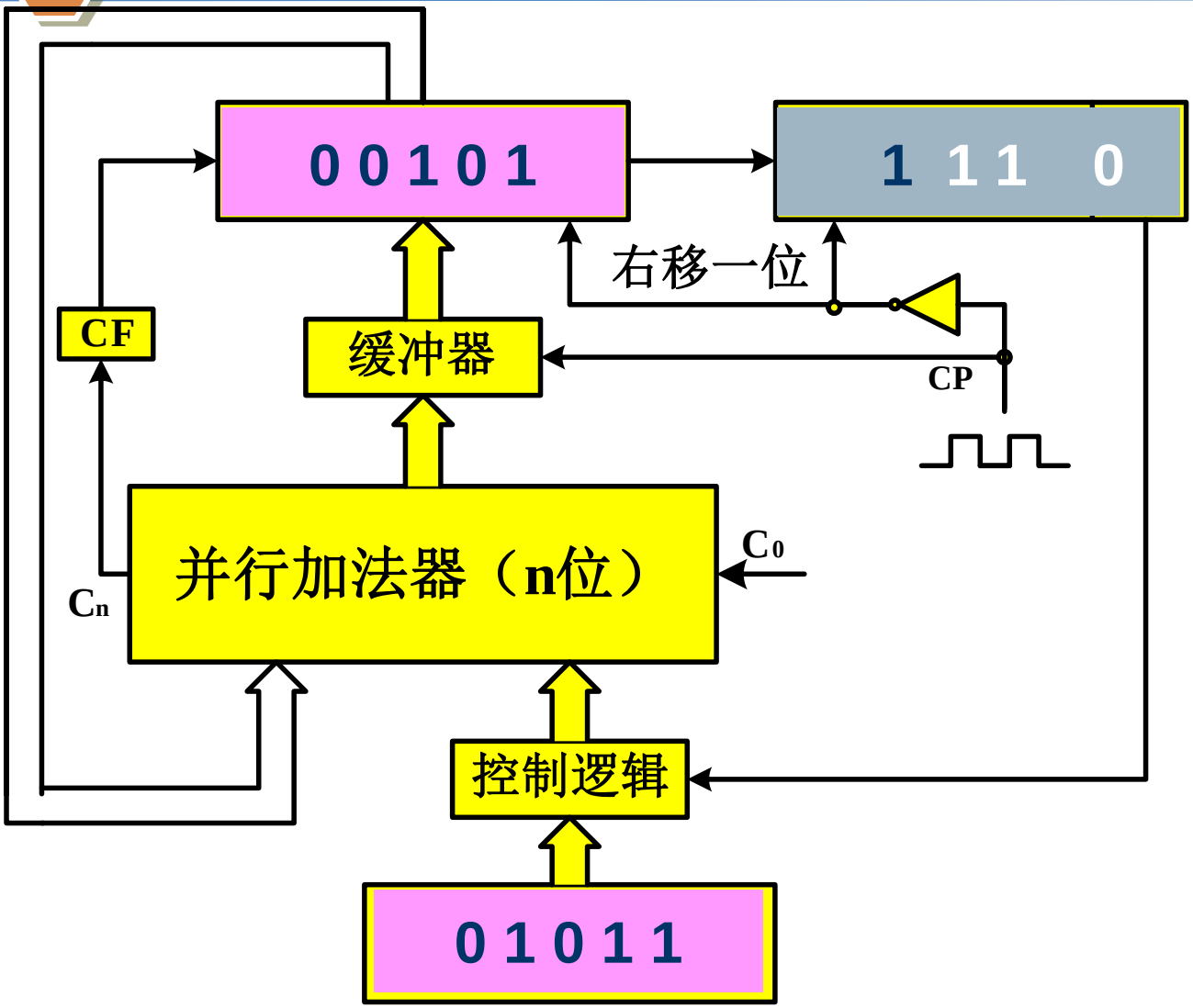
右移1位

00000	1101
01011	1101
00101	1110



第二次求部分积　加运算：＋0

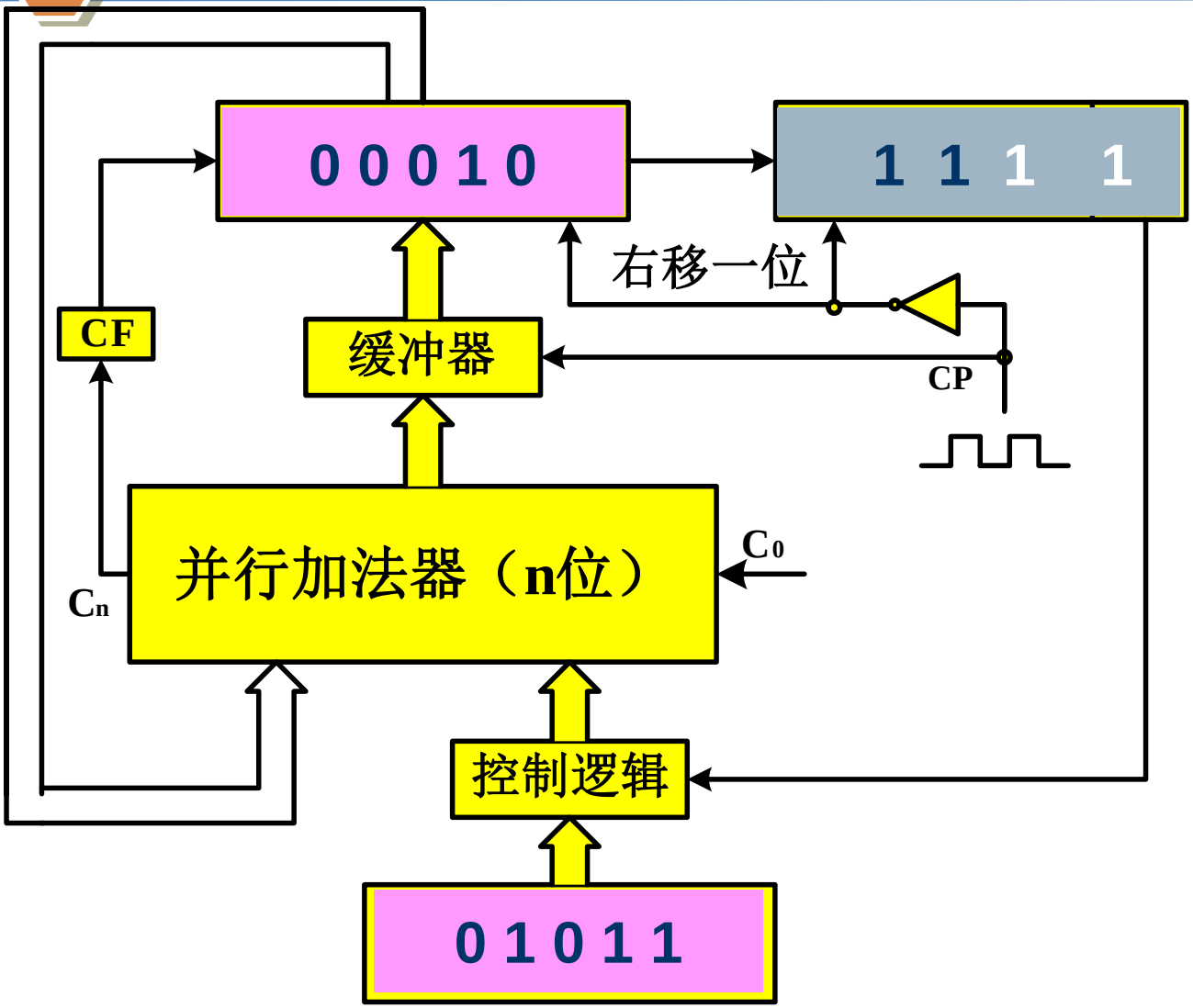
00000	1101
01011	1101
00101	1110
00101	1110



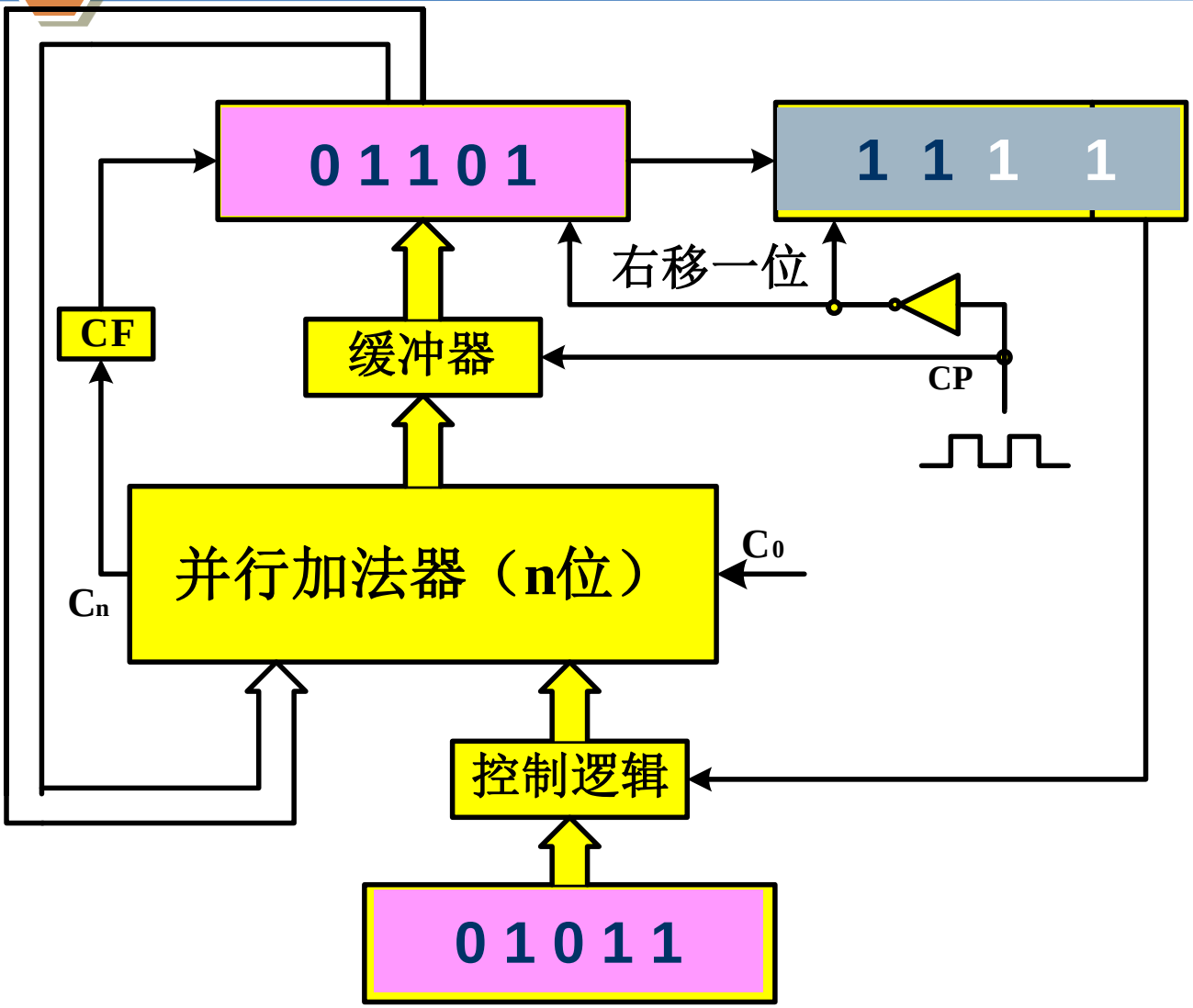
第二次求部分积

右移1位

00000	1101
01011	1101
00101	1110
00101	1110
00010	1111



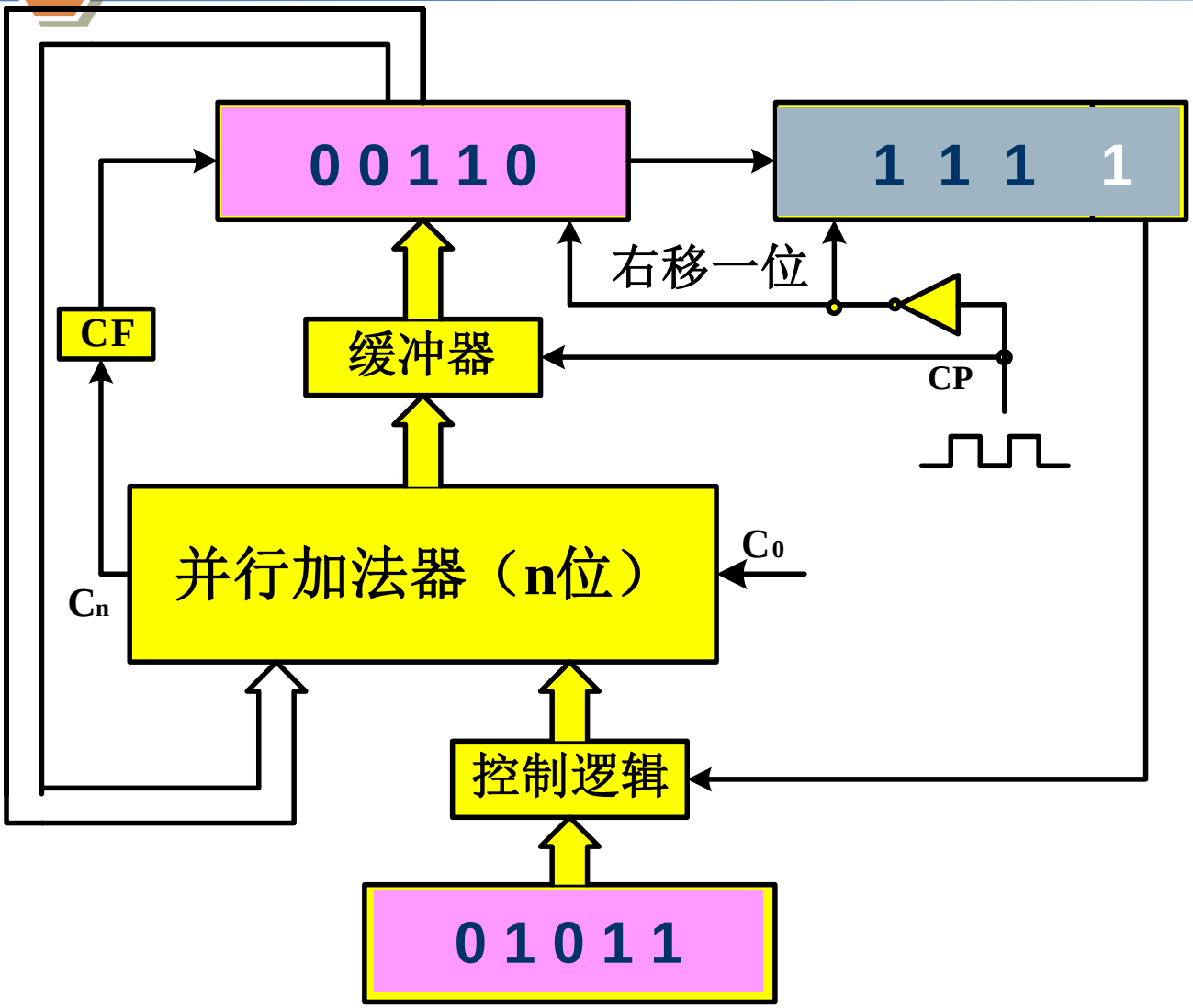
第三次求部分积　加运算：＋|X|



00000	1101
01011	1101
00101	1110
00101	1110
00010	1111
01101	1111

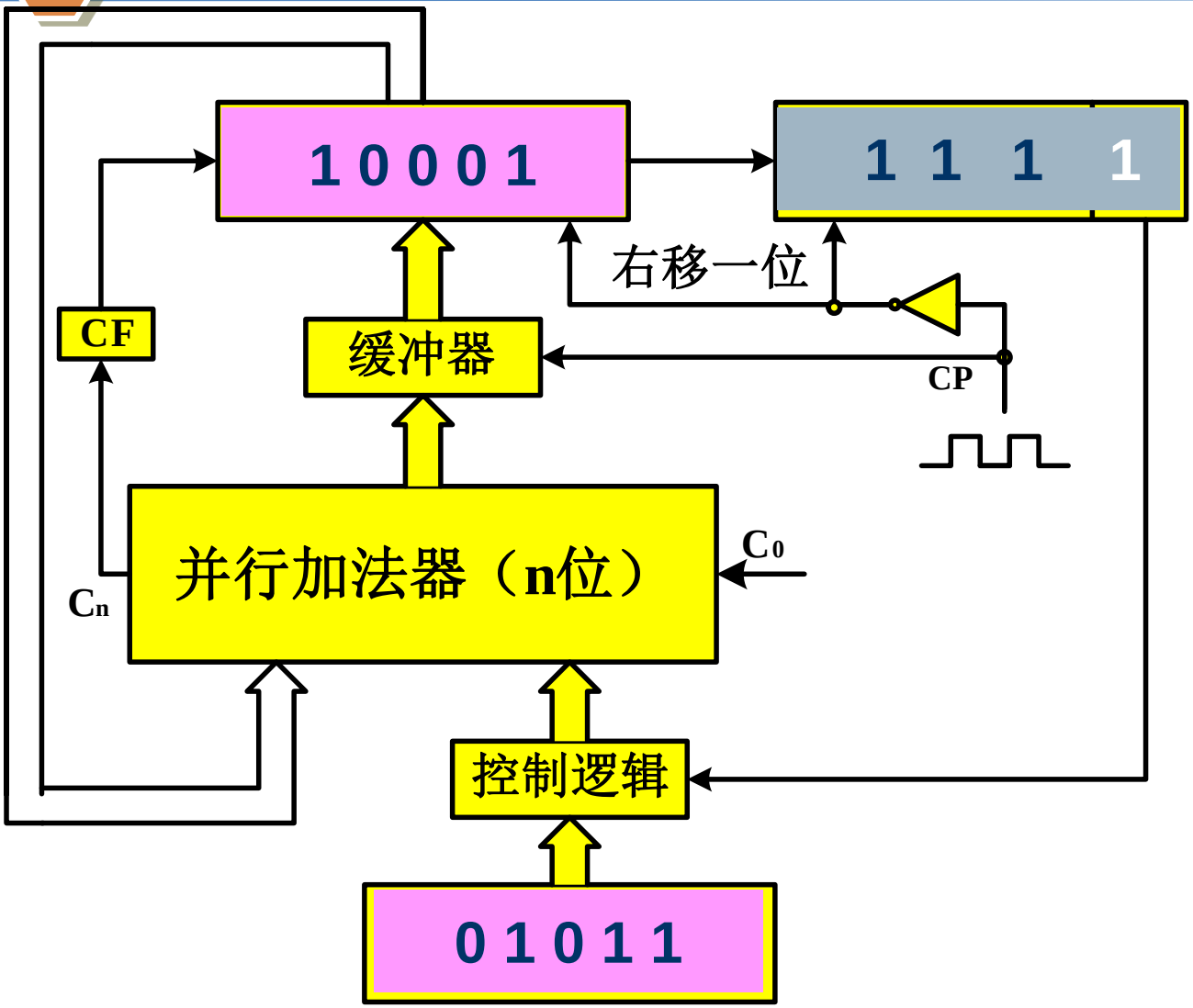
第三次求部分积

右移1位



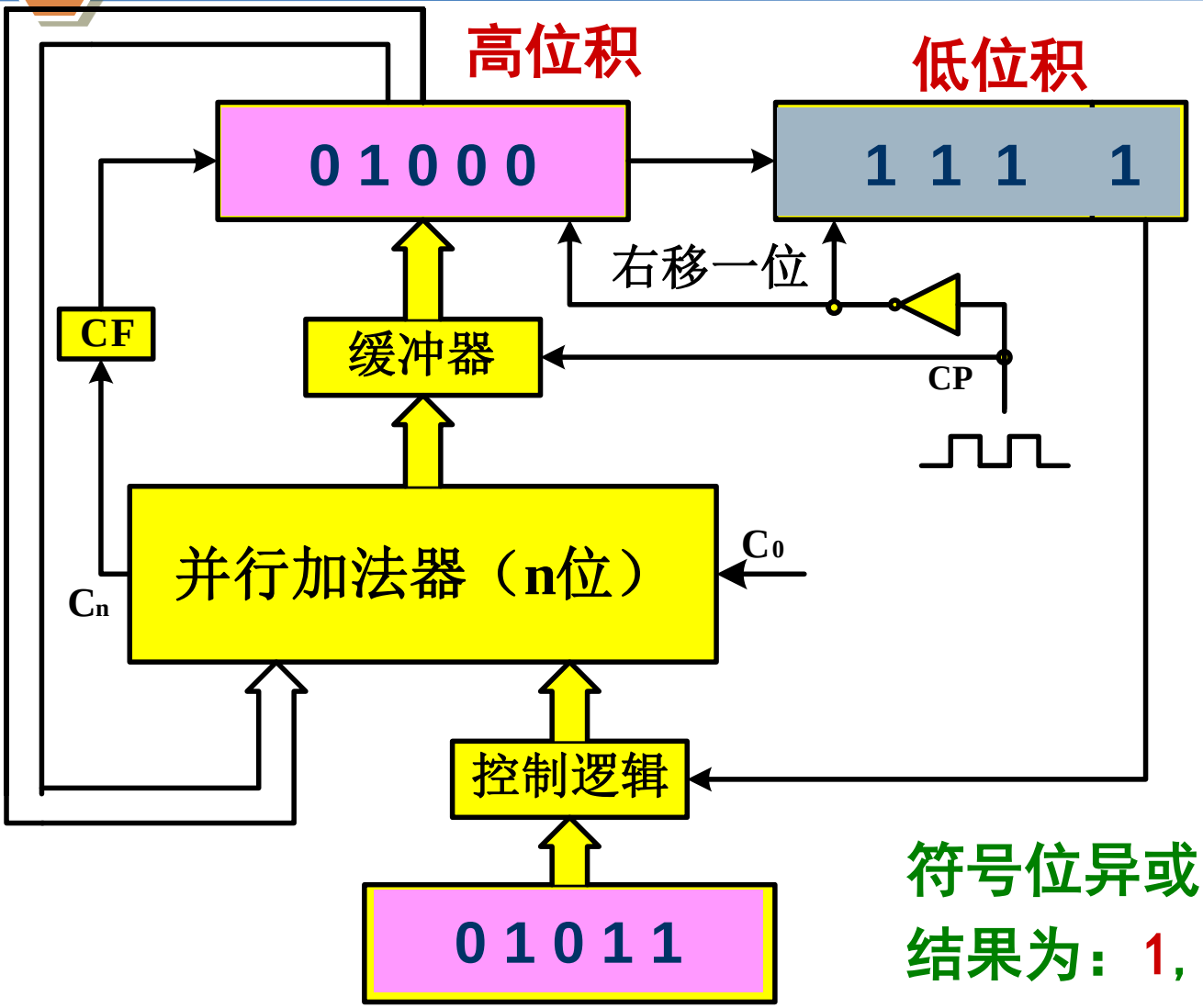
00000	1101
01011	1101
00101	1110
00101	1110
00010	1111
01101	1111
00110	1111

第四次求部分积　加运算：＋|X|



00000	1101
01011	1101
00101	1110
00101	1110
00010	1111
01101	1111
00110	1111
10001	1111

第四次求部分积　右移1位

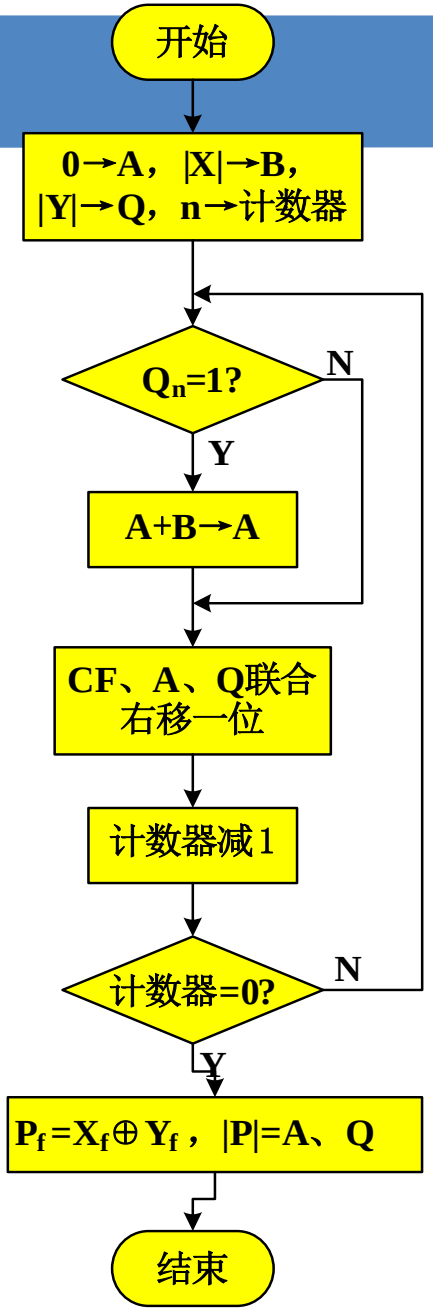


00000	1101
01011	1101
00101	1110
00101	1110
00010	1111
01101	1111
00110	1111
01000	1111

符号位异或

结果为：1, 10001111

原码一位乘法流程：


开始
0→A，|X|→B，
|Y|→Q，n→计数器
Qn=1?
N
Y
A+B→A
CF、A、Q联合
右移一位
计数器减1
计数器=0?
N
Y
Pf=Xf⊕Yf，|P|=A、Q
结束

二、补码乘法及实现

- 原码乘法存在的缺点是符号位需要单独运算，并要在最后给乘积冠以正确的符号。

- 补码乘法是指采用操作数的补码进行乘法运算，最后乘积仍为补码，能自然得到乘积的正确符号。

二、补码乘法及实现

- 1、补码乘法算法
- （1）补码一位乘法——校正法

假设$[X]_{补} = X_0 . X_1 \cdots\cdots X_n$，

$[Y]_{补} = Y_0 . Y_1 \cdots\cdots Y_n$，

则有：

$$[X \cdot Y]_{补} = [X]_{补} \cdot (0. Y_1 \cdots\cdots Y_n) + Y_0 \cdot [-X]_{补}$$

- 该式表明，当使用补码做乘法时，可以将乘数Y的补码的数值部分，像原码一样直接做乘法，最后再根据乘数Y的符号位对结果做修正：若符号位为正（即为“0”），无须修正；若符号位为负（即为“1”），减X修正。

二、补码乘法及实现

证明如下：

- 当被乘数X的符号任意，Y为正数时：

根据补码定义有：

$[X]_{补} = 2 + X = 2^{n+1} + X \quad (\mathrm{mod}\ 2)$

$[Y]_{补} = Y$

则：

$[X]_{补} \cdot [Y]_{补} = (2^{n+1} + X) \cdot Y = 2^{n+1} \cdot Y + X \cdot Y$

$= 2^{n+1} \cdot (0.Y_1 \ldots\ldots Y_n) + X \cdot Y$

$= 2 \cdot (Y_1 \ldots\ldots Y_n) + X \cdot Y = 2 + X \cdot Y \quad (\mathrm{mod}\ 2)$

$= [X \cdot Y]_{补}$

即：$Y > 0$ 时，$[X \cdot Y]_{补} = [X]_{补} \cdot [Y]_{补}$

$= [X]_{补} \cdot (0.Y_1 \ldots\ldots Y_n)$

二、补码乘法及实现

- 当被乘数X的符号任意，Y为负数时：

$[Y]_{补} = 2 + Y = 1.Y_1\cdots\cdots Y_n$ 则：

$Y = [Y]_{补} - 2 = 0.Y_1\cdots\cdots Y_n - 1$

$[X\cdot Y]_{补} = [X\cdot 0.Y_1\cdots\cdots Y_n - X]_{补}$

$= [X\cdot 0.Y_1\cdots\cdots Y_n]_{补} + [-X]_{补}$

因为$0.Y_1\cdots\cdots Y_n > 0$，所以：

$[X\cdot 0.Y_1\cdots\cdots Y_n]_{补} = [X]_{补}\cdot(0.Y_1\cdots\cdots Y_n)$

所以：$Y<0$时，

$[X\cdot Y]_{补} = [X]_{补}\cdot(0.Y_1\cdots\cdots Y_n) + [-X]_{补}$

所以，$[X\cdot Y]_{补} = [X]_{补}\cdot(0.Y_1\cdots\cdots Y_n) + Y_0\cdot[-X]_{补}$成立

二、补码乘法及实现

例如： X=+0.1011，Y=−0.1101，用补码一位乘法的校正法计算P=X·Y。

$[X]_{补}$=00.1011　　$[Y]_{补}$=11.0011　　$[-X]_{补}$=11.0101

	部分积	乘数Y	操作说明
	00.0000	0 0 1 1	
+	00.1011		Y_4=1，$+[X]_{补}$
	00.1011		
	00.0101	1 0 0 1	右移一位
+	00.1011		Y_3=1，$+[X]_{补}$
	01.0000		
	00.1000	0 1 0 0	右移一位
+	00.0000		Y_2=0，+0
	00.1000		
	00.0100	0 0 1 0	右移一位
+	00.0000		Y_1=0，+0
	00.0100		
	00.0010	0 0 0 1	右移一位
+	11.0101		Y_0=1，$+[-X]_{补}$校正
	11.0111	0 0 0 1	

$[X\cdot Y]_{补}$ = 1.0111 0001

X·Y = -0.1000 1111

例：设X=-0.1101，Y=-0.1011，即：$[X]_{补}$=11.0011，$[Y]_{补}$=11.0101，求$[X*Y]_{补}$

部分积	乘数	说明
0 0. 0 0 0 0	0 1 0 1	初始值
+[X]补 1 1. 0 0 1 1		+[X]补
1 1. 0 0 1 1		
右移一位 1 1. 1 0 0 1	1 0 1 0	右移一位
+0 0 0. 0 0 0 0		+0
1 1. 1 0 0 1		
右移一位 1 1. 1 1 0 0	1 1 0 1	右移一位
+[X]补 1 1. 0 0 1 1		+[X]补
1 0. 1 1 1 1		
右移一位 1 1. 0 1 1 1	1 1 1 0	右移一位
+0 0 0. 0 0 0 0		+0
1 1. 0 1 1 1		
右移一位 1 1. 1 0 1 1	1 1 1 1	右移一位
+[-X]补 0 0. 1 1 0 1		+[-X]补
0 0. 1 0 0 0	1 1 1 1	

计算结果：　　$[X*Y]_{补}$=0.10001111

二、补码乘法及实现

（2）补码一位乘法——Booth算法

做出如下推导 ：

$[X\cdot Y]_{补} = [X]_{补}\cdot (0.Y_1.....Y_n) + Y_0\cdot[-X]_{补}$

$= [X]_{补}\cdot (Y_1\cdot 2^{-1} + Y_2\cdot 2^{-2} + ...+ Y_n\cdot 2^{-n} - Y_0)$

$= [X]_{补}\cdot[Y_1\cdot (2^0 - 2^{-1}) + Y_2\cdot (2^{-1} - 2^{-2}) + ...+ Y_n\cdot (2^{-n+1} - 2^{-n}) - Y_0\cdot 2^0]$

$= [X]_{补}\cdot[Y_1\cdot 2^0 - Y_1\cdot 2^{-1} + Y_2\cdot 2^{-1} - Y_2\cdot 2^{-2} + ...+ Y_n\cdot 2^{-n+1} - Y_n\cdot 2^{-n} - Y_0\cdot 2^0]$

$= [X]_{补}\cdot[(Y_1-Y_0)\cdot 2^0 + (Y_2- Y_1)\cdot 2^{-1} + (Y_3- Y_2)\cdot 2^{-2} + ...+ (Y_n- Y_{n-1})\cdot 2^{-n+1} - Y_n\cdot 2^{-n}]$

$= [X]_{补}\cdot[(Y_1-Y_0)\cdot 2^0 + (Y_2- Y_1)\cdot 2^{-1} + (Y_3- Y_2)\cdot 2^{-2} + ...+ (Y_n- Y_{n-1})\cdot 2^{-n+1} + (Y_{n+1} - Y_n)\cdot 2^{-n}]$

$= [X]_{补}\cdot (a_0\cdot 2^0 + a_1\cdot 2^{-1} + a_2\cdot 2^{-2} + ...+ a_{n-1}\cdot 2^{-n+1} + a_n\cdot 2^{-n})$

0

其中，将乘数Y的补码在最末位添加一位附加位Y_{n+1}（初始为0），$a_i = Y_{i+1} - Y_i$ ，i=0，1，……，n-1，n。

Booth算法的运算规则

假设$[Y]_{补}$ = Y0 .Y1……Yn

① 被乘数X和乘数Y均以补码的形式参加乘法运算，运算的结果是积的补码。

② 部分积和被乘数X采用双符号位，乘数Y采用单符号位。

③ 初始部分积为0；运算前，在乘数Y的补码末位后添加一位附加位Yn+1，初始为0。

④ 根据YnYn+1的值，按照表4.3进行累加右移操作，右移时遵循补码的移位规则。

⑤ 累加n+1次，右移n次，即最后一次不右移。

表4-3 补码乘法的 Booth 算法操作表

Y_n	Y_{n+1}	操作
0	0	+0，右移一位
0	1	$+[X]_{补}$，右移一位
1	0	$+[-X]_{补}$，右移一位
1	1	+0，右移一位

Booth算法操作时间

$$T_m = (n+1)\,t_a + n\,t_r$$

t_a为执行一次加法的时间，t_r为执行一次右移的时间

若加法和右移操作合在一起进行，则

$$T_m = (n+1)\,t_a$$

例如： X=+0.1011，Y=−0.1101，用补码一位乘法的Booth算法计算P=X·Y。

解： $[X]_{补}$=00.1011　　$[Y]_{补}$=11.0011　　$[-X]_{补}$=11.0101

部分积	乘数Y（$Y_n\ Y_{n+1}$）	操作说明
00.0000	1.0 0 1 1 0	
+ 11.0101		Y_4Y_5=10，+$[-X]_{补}$
11.0101		
11.1010	1 1.0 0 1 1	右移一位
+ 00.0000		Y_3Y_4=11，+0
11.1010		
11.1101	0 1 1.0 0 1	右移一位
+ 00.1011		Y_2Y_3=01，+$[X]_{补}$
00.1000		
00.0100	0 0 1 1.0 0	右移一位
+ 00.0000		Y_1Y_2=00，+0
00.0100		
00.0010	0 0 0 1 1.0	右移一位
+ 11.0101		Y_0Y_1=10，+$[-X]_{补}$
11.0111	0 0 0 1	

$[X\cdot Y]_{补}$ = 1.0111 0001

X·Y = −0.1000 1111

使用补码Booth乘法计算X*Y： X=0.01111，Y=-0.11101 。

解： ［X］补＝00.01111
［Y］补＝11.00011
［-X］补＝11.10001

部分积	乘数Y(Y_nY_{n+1})	操作说明
00.00000	1.000110	
+ 11.10001		Y_5Y_6=10，+［−X］补
11.10001		
11.11000	1 1.00011	右移一位
+ 00.00000		Y_4Y_5=11，+0
11.11000		
11.11100	0 1 1.0001	右移一位
+ 00.01111		Y_3Y_4=01，+［X］补
00.01011		
00.00101	1 0 1 1.000	右移一位
+ 00.00000		Y_2Y_3=00，+0
00.00101		
00.00010	1101 1.00	右移一位
+ 00.00000		Y_1Y_2=00，+0
00.00010		
00.00001	01101 1.0	右移一位
+ 11.10001		Y_0Y_1=10，+［−X］补
11.10010	01101	

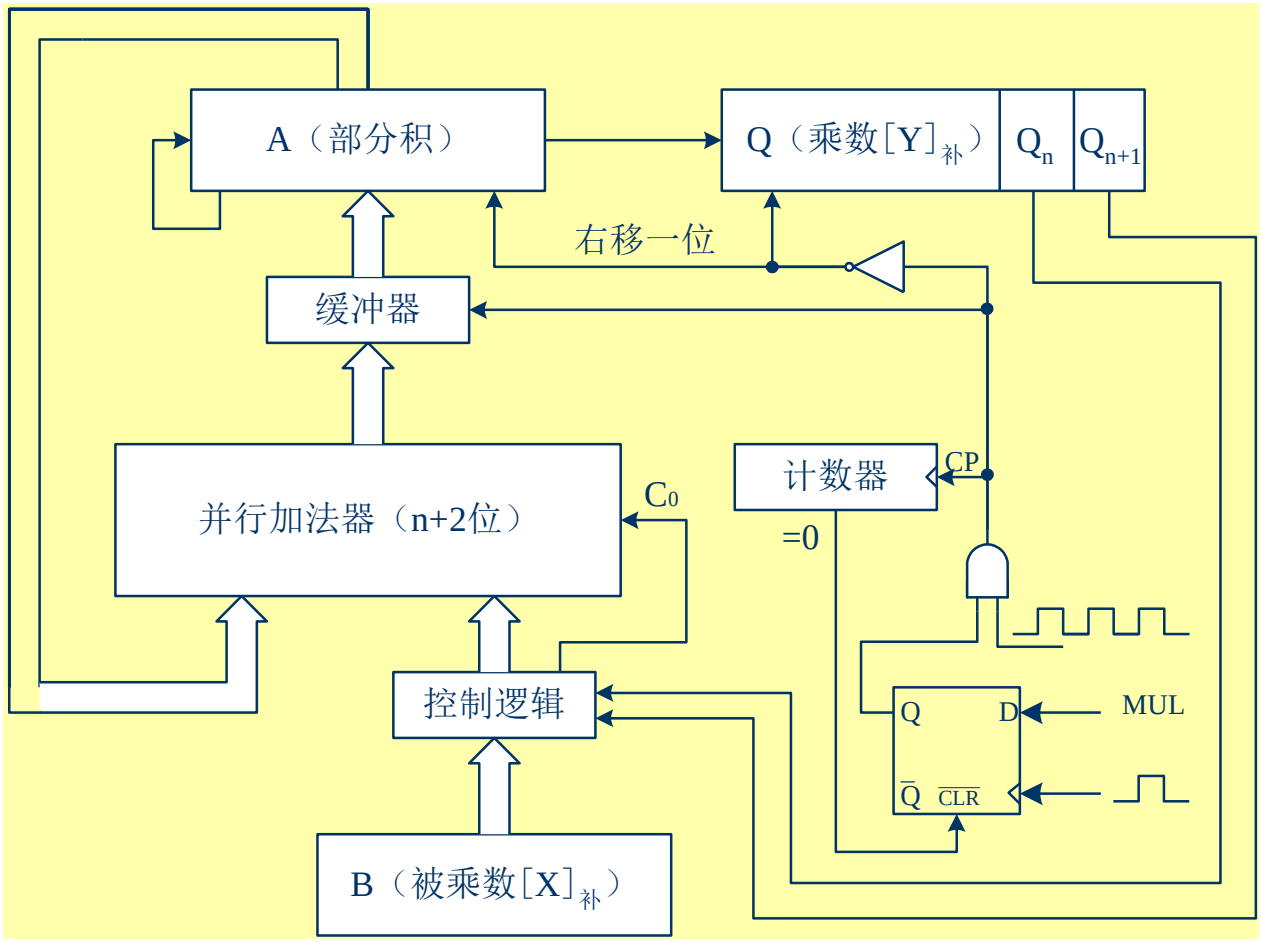
所以：

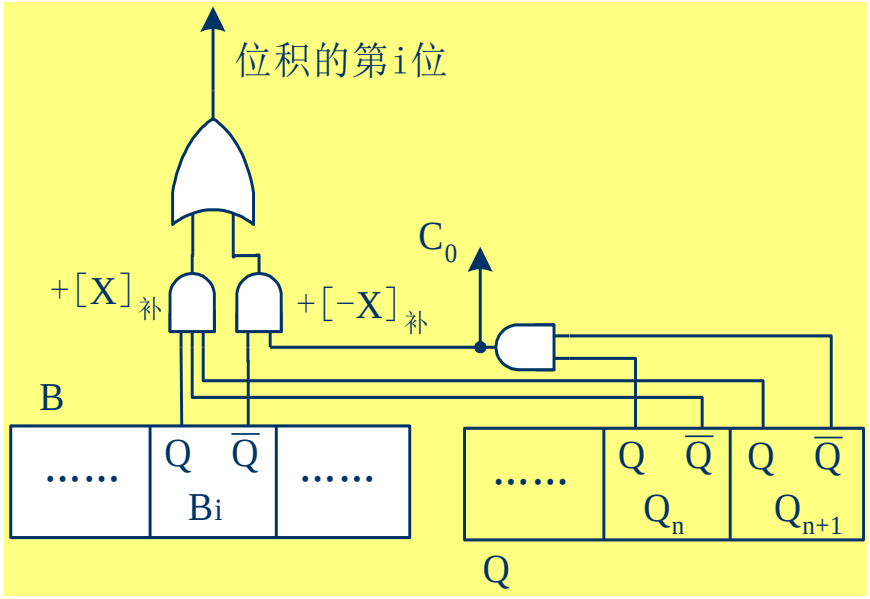
［X*Y］补＝1.1001001101

X*Y＝-0.0110110011

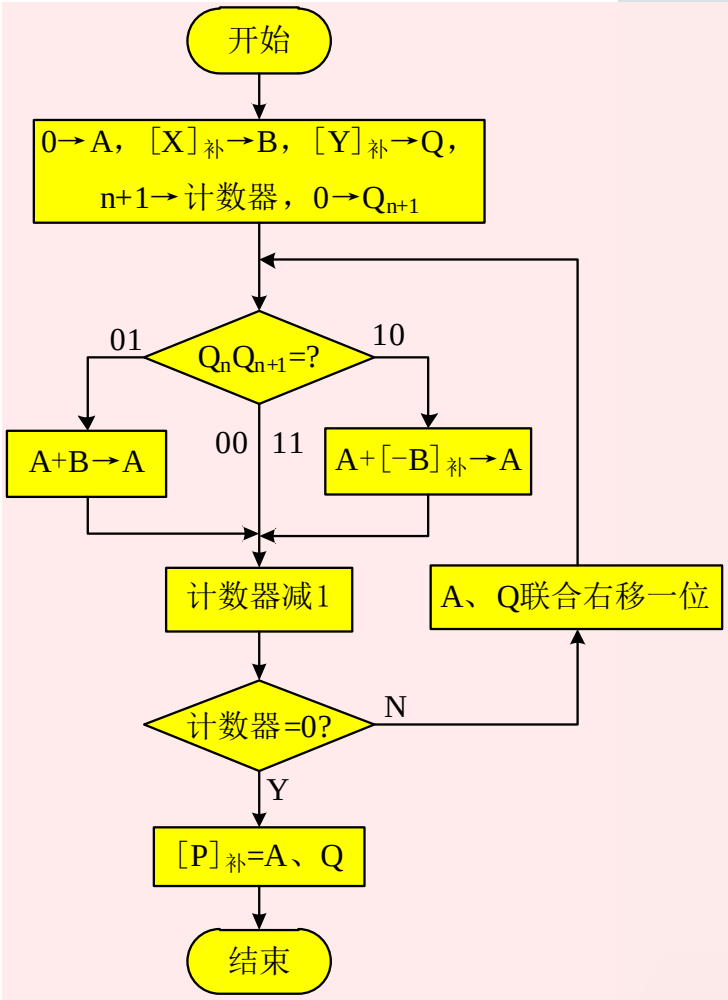
二、补码乘法及实现

2、补码乘法的硬件实现





控制逻辑电路



补码乘法的Booth算法流程

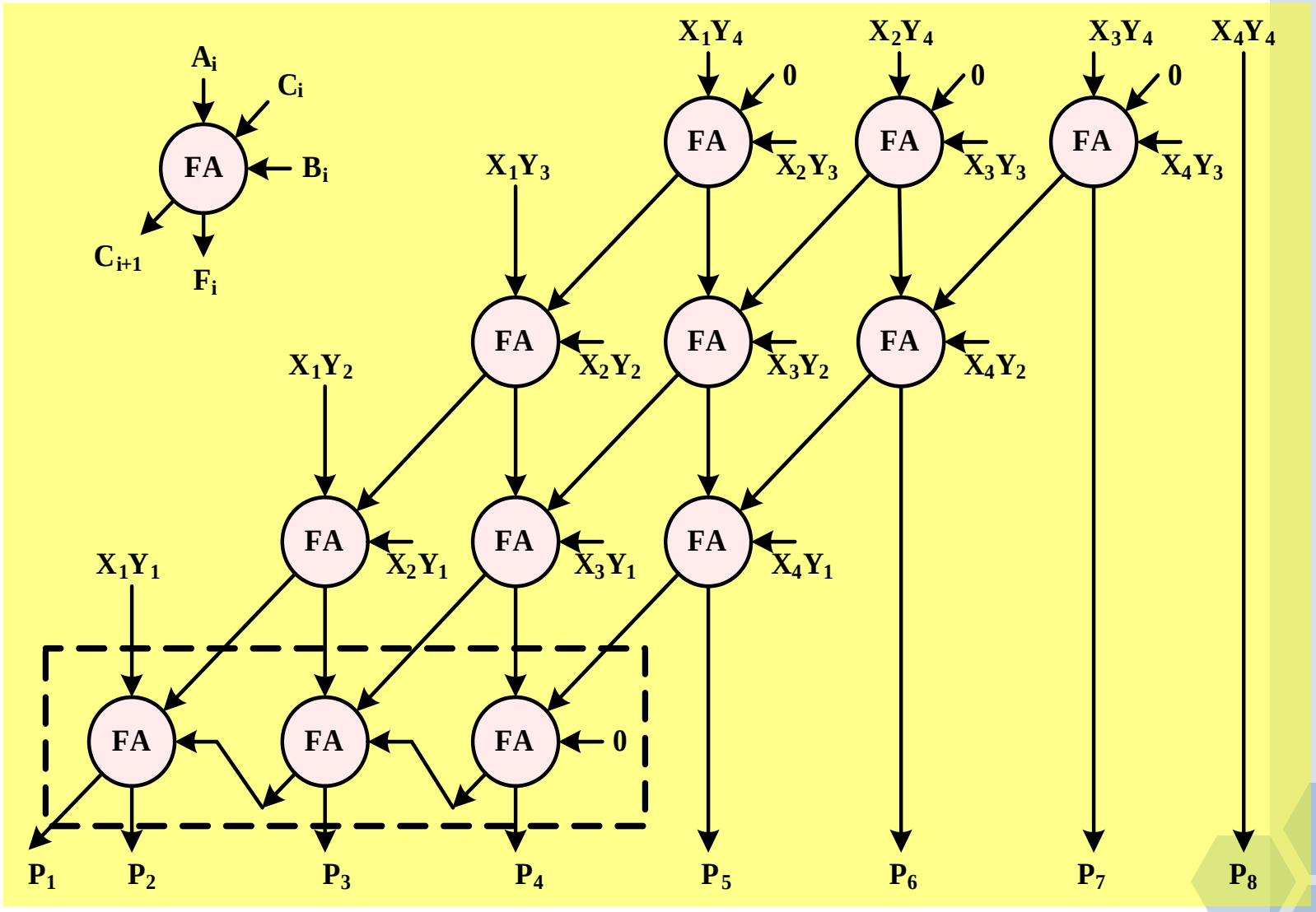
三、阵列乘法器

- 原理类似于二进制手工算法
  - 下图为4位二进制手工算式。算式中，位积的每一位XiYj都可以用一个与门实现，而每一位的相加均可以使用一个全加器来实现。

$$
\begin{array}{cccccccc}
 & & & & X_1 & X_2 & X_3 & X_4 \\
 & & & \times & Y_1 & Y_2 & Y_3 & Y_4 \\
\hline
 & & & & X_1Y_4 & X_2Y_4 & X_3Y_4 & X_4Y_4 \\
 & & & X_1Y_3 & X_2Y_3 & X_3Y_3 & X_4Y_3 & \\
 & & X_1Y_2 & X_2Y_2 & X_3Y_2 & X_4Y_2 & & \\
+ & X_1Y_1 & X_2Y_1 & X_3Y_1 & X_4Y_1 & & & \\
\hline
P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8
\end{array}
$$

图4.8　4位二进制手工算式

绝对值阵列乘法器


Ai
Ci
FA
Bi
Ci+1
Fi
X1Y4
X2Y4
X3Y4
X4Y4
0
X1Y3
X2Y3
X3Y3
X4Y3
X1Y2
X2Y2
X3Y2
X4Y2
X1Y1
X2Y1
X3Y1
X4Y1
P1
P2
P3
P4
P5
P6
P7
P8

n×n位阵列乘法器的构成：

① 被加数位积产生部件： n×n个“与门”

② 被加数求和部件： n(n–1)个加法器

绝对值阵列乘法器一次乘法的时间

设与门的传输延迟时间为T_{and}, 加法器的传输延迟时间为T_c，则

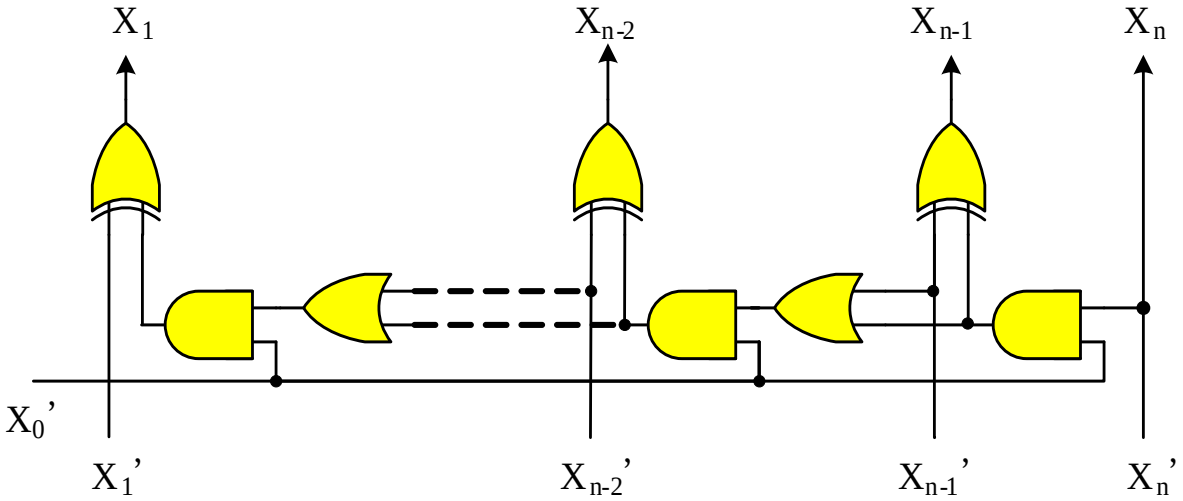
$$T_m=T_{and}+[(n-1)+(n-1)]T_c=T_{and}+(2n-2)T_c \qquad (2.39)$$

带符号的阵列乘法器

① 原码阵列乘法只需另加符号位的处理。

② 补码阵列乘法可对负操作数的补码求补（得绝对值）后相乘，再由乘积符号位决定是否对乘积求补即可。

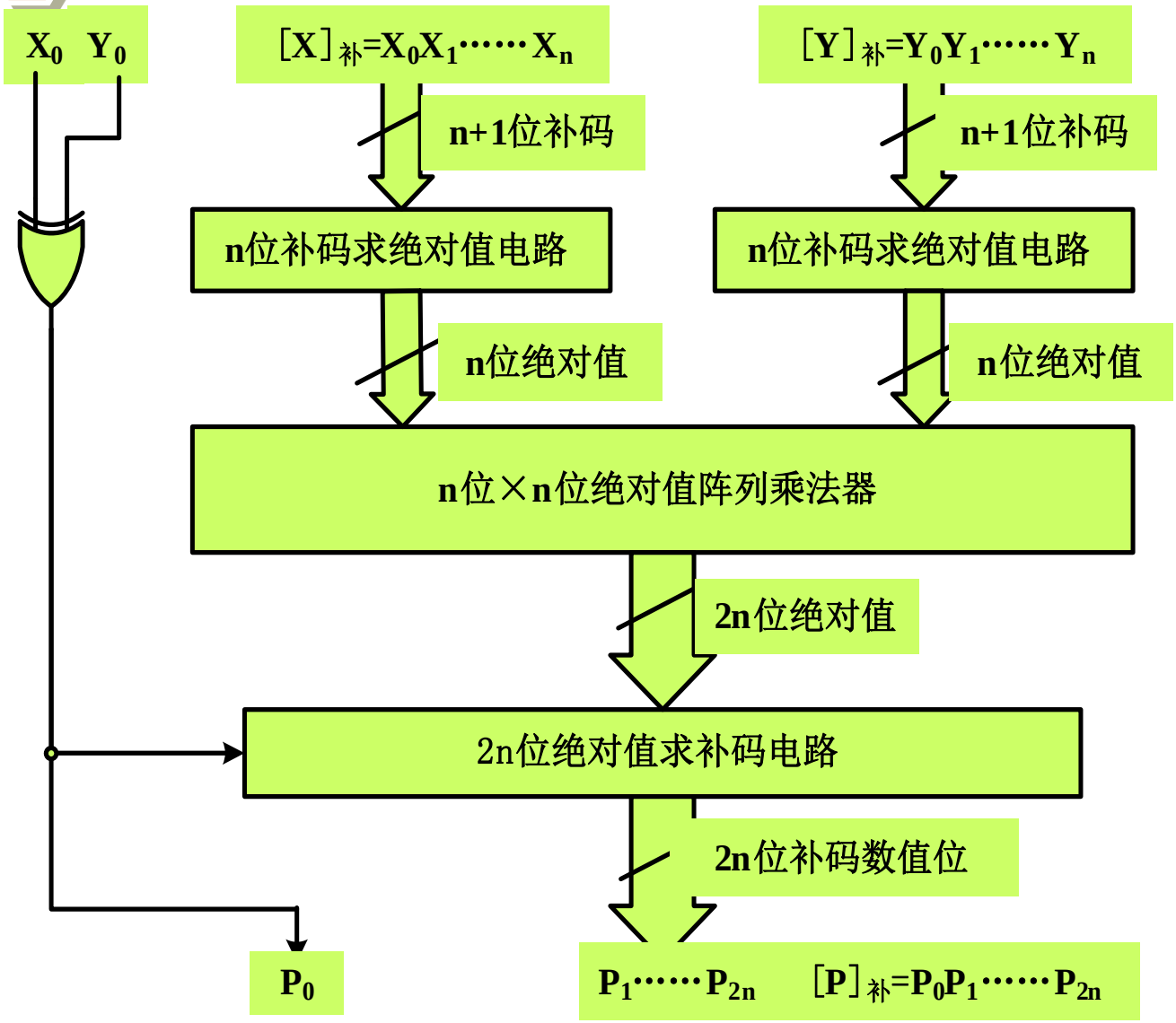
补码阵列乘法器



补码求绝对值电路

先通过一个补码求绝对值的逻辑电路变为绝对值后，送入绝对值阵列乘法器，运算得到积的绝对值，然后再通过一个绝对值求补码的逻辑电路，根据积的符号求出积的补码形式。

补码阵列乘法器


X_0 Y_0
$[X]_{补}=X_0X_1\cdots\cdots X_n$
$[Y]_{补}=Y_0Y_1\cdots\cdots Y_n$
n+1位补码
n+1位补码
n位补码求绝对值电路
n位补码求绝对值电路
n位绝对值
n位绝对值
n位×n位绝对值阵列乘法器
2n位绝对值
2n位绝对值求补码电路
2n位补码数值位
P_0
$P_1\cdots\cdots P_{2n}$ $[P]_{补}=P_0P_1\cdots\cdots P_{2n}$

4.3 定点数除法运算及实现

一 原码除法及实现

二 补码除法及实现

三 阵列除法器（自学）

一、原码除法及实现

1、原码除法算法

❖（1）手工除法算法

- 改进手工算法即可适合机器运算：
  - 计算机通过做减法测试来实现判断：结果大于等于0，表明够减，商1；结果小于0，表明不够减，商0。
  - 计算机将余数左移一位，再直接与不右移的除数相减。

$$
\begin{array}{r|l}
 & 0.1101 \\
\hline
0.1101 & 0.10110 \\
 & 1101 \\
\hline
 & 10010 \\
 & 1101 \\
\hline
 & 10100 \\
 & 1101 \\
\hline
 & 0111
\end{array}
$$

X=+0.1011，Y=-0.1101

X÷Y

商为-0.1101，

余数为+0.00000111

一、原码除法及实现

（2）原码恢复余数算法

- 假设$[X]_{原}=X_S . X_1 X_2 \ldots\ldots X_n$，$[Y]_{原}=Y_S . Y_1 Y_2 \ldots\ldots Y_n$，Q是X÷Y的商，$Q_S$是商的符号，R是X÷Y的余数，$R_S$是余数的符号

- 原码除法运算的规则是：
  - 1. $Q_S = X_S \oplus Y_S$，$R_S = X_S$，$|Q| = |X| \div |Y| - |R| \div |Y|$
  - 2. 余数和被除数、除数均采用双符号位；初始余数为|X|。

一、原码除法及实现

- 原码除法运算的规则是：
  - 3. 每次用余数减去|Y|（通过加上$[-|Y|]_{补}$来实现），若结果的符号位为0，则够减，上商1，余数左移一位；若结果的符号位为1，则不够减，上商0，先加|Y|恢复余数，然后余数左移一位。
  - 4. 循环操作步骤3，共做n+1次，最后一次不左移，但若最后一次上商0，则必须+|Y|恢复余数；若为定点小数除法，余数则为最后计算得到的余数右移n位的值。

例如： X=+0.1011，

Y= － 0.1101

用原码恢复余数算法计算X÷Y。

解：$[X]_{原}$=0.1011

$[Y]_{原}$=1.1101

|X|=0.1011　　|Y|=0.1101

$[-|Y|]_{补}$=11.0011

$Q_S = X_S \oplus Y_S = 1$　　$R_S = 0$

得$[Q]_{原}$=1.1101

$[R]_{原}$=0.00000111

被除数/余数	商Q	操作说明
00.1011	0 0 0 0 0	
+ 11.0011		+[－|Y|]补
11.1110	0 0 0 0 0	R_0<0，上商0
+ 00.1101		+|Y|恢复余数
00.1011		
01.0110	0 0 0 0 0	左移一位
+ 11.0011		+[－|Y|]补
00.1001	0 0 0 0 1	R_1>0，上商1
01.0010	0 0 0 1 0	左移一位
+ 11.0011		+[－|Y|]补
00.0101	0 0 0 1 1	R_2>0，上商1
00.1010	0 0 1 1 0	左移一位
+ 11.0011		+[－|Y|]补
11.1101	0 0 1 1 0	R_3<0，上商0
+ 00.1101		+|Y|恢复余数
00.1010		
01.0100	0 1 1 0 0	左移一位
+ 11.0011		+[－|Y|]补
00.0111	0 1 1 0 1	R_4>0，上商1

一、原码除法及实现

（3）原码不恢复余数算法

- 又称为加减交替法：当某一次求得的差值（余数R_i）为负时，不是恢复它，而是继续求下一位商，但用加上除数（+|Y|）的办法来取代（-|Y|）操作，其他操作不变。
- 其原理证明如下：
  - 在恢复余数除法中，若第i-1次求商的余数为R_{i-1}，下一次求商的余数为R_i，则：

    $R_i=2R_{i-1}-|Y|$
  - 如果$R_i>=0$，商的第i位上1，并执行操作：余数左移一位，再减|Y|，得R_{i+1}，则：

    $R_{i+1}=2R_i-|Y|$

一、原码除法及实现

- 如果R_i<0，商的第i位上0，并执行操作：恢复余数（+|Y|），将余数左移一位，再减|Y|，得R_{i+1}。其过程可用公式表示如下：

$$R_{i+1}=2（R_i+|Y|）－|Y|=2R_i+2|Y|－|Y|=2R_i+|Y|$$

- 加减交替法的规则

一、原码除法及实现

- 加减交替法的规则如下：
  - 1. $Q_S = X_S \oplus Y_S$ ，$R_S = X_S$，$|Q| = |X| \div |Y| - |R| \div |Y|$，即符号位单独处理，绝对值参加除法运算。
  - 2. 部分余数和被除数、除数均采用双符号位；初始余数为|X|。
  - 3. 每次用部分余数减去|Y|（通过加上$[-|Y|]_{补}$来实现），若结果的符号位为0，则够减，上商1，部分余数左移一位，然后通过减去|Y|的方法来求下一次的部分余数；若结果的符号位为1，则不够减，上商0，部分余数左移一位，然后通过加上|Y|的方法来求下一次的部分余数。
  - 4. 循环操作步骤3，共做n+1次，最后一次上商后不左移，但若最后一次上商0，则必须+|Y|恢复余数。
  - 5. 若为定点小数除法，余数则为最后计算得到的余数右移n位的值。

例如：X=+0.1011，Y=−0.1101，用原码不恢复余数算法计算X÷Y。

解：$[X]_{原}$=0.1011

$[Y]_{原}$=1.1101

|X|=0.1011

|Y|=0.1101

$[-|Y|]_{补}$=11.0011

$Q_S = X_S \oplus Y_S = 1$

$R_S = 0$

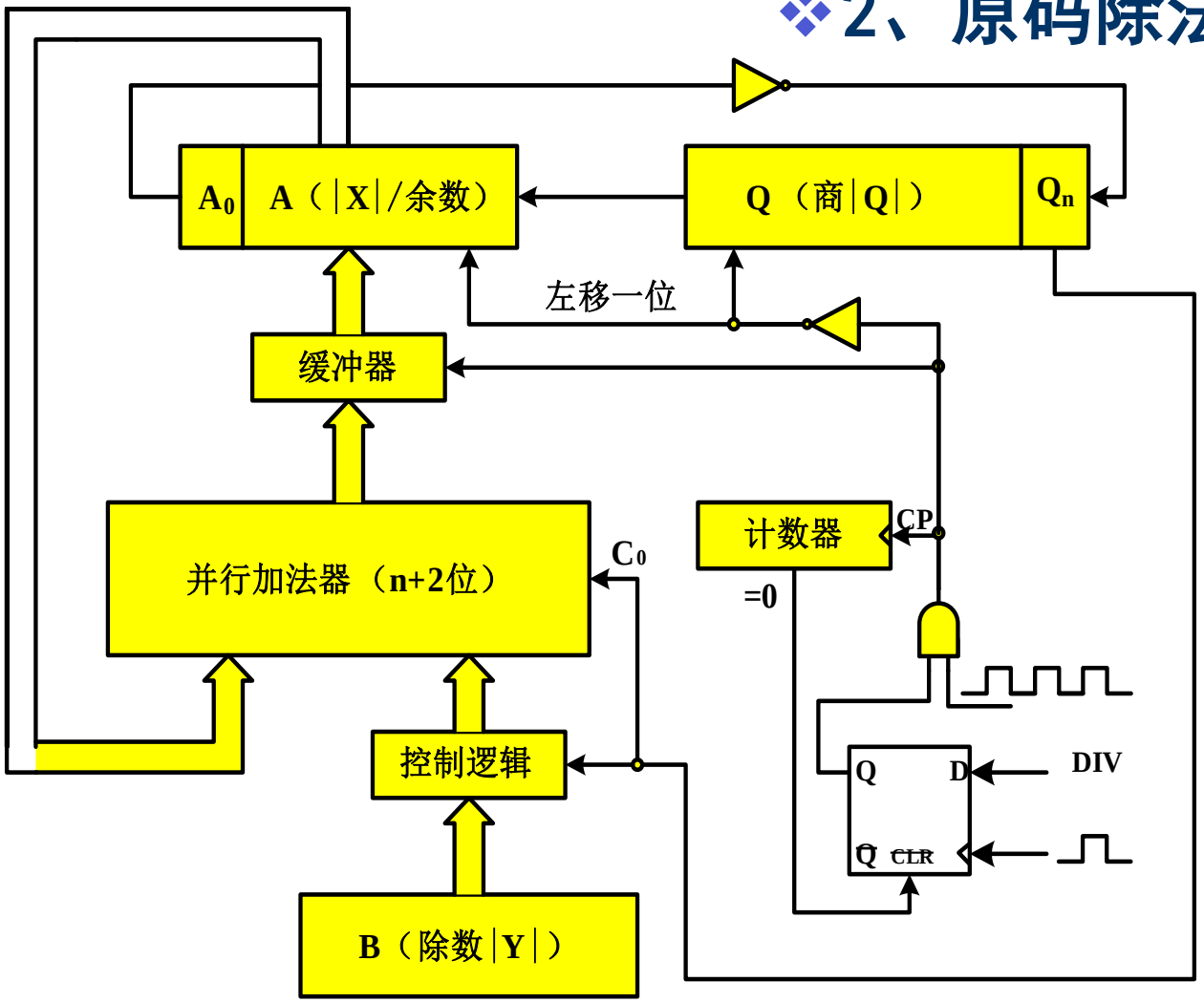
$[Q]_{原}$=1.1101

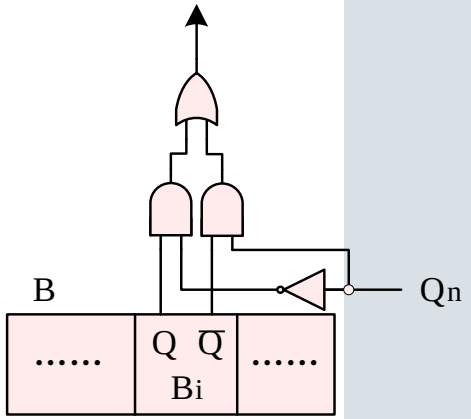
$[R]_{原}$=0.00000111

被除数/余数	商Q	操作说明
00.1011	0 0 0 0 0	
+ 11.0011		+[−|Y|]补
11.1110	0 0 0 0 0	R_0<0，上商0
11.1100	0 0 0 0 0	左移一位
+ 00.1101		+|Y|
00.1001	0 0 0 0 1	R_1>0，上商1
01.0010	0 0 0 1 0	左移一位
+ 11.0011		+[−|Y|]补
00.0101	0 0 0 1 1	R_2>0，上商1
00.1010	0 0 1 1 0	左移一位
+ 11.0011		+[−|Y|]补
11.1101	0 0 1 1 0	R_3<0，上商0
11.1010	0 1 1 0 0	左移一位
+ 00.1101		+|Y|
00.0111	0 1 1 0 1	R_4>0，上商1

一、原码除法及实现

❖2、原码除法的硬件实现

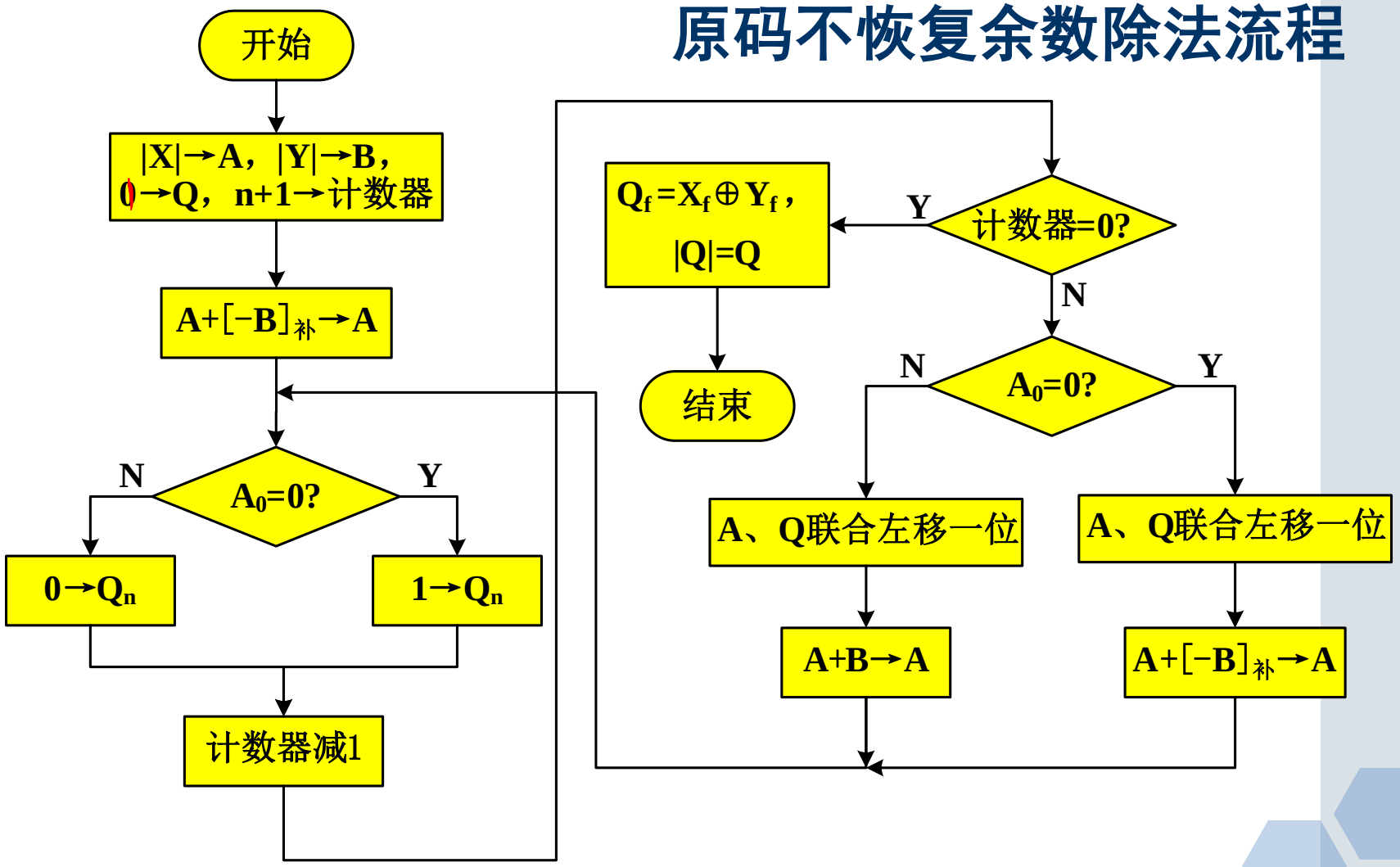




控制电路逻辑

一、原码除法及实现

原码不恢复余数除法流程



一、原码除法及实现

除法总时间

$$T_d = (n+1)T_a + nT_r,$$

式中，T_a为一次加/减法时间
T_r为一次移位时间

若运算和移位合二为一，则

$$T_d = (n+1)T_a$$

二、补码除法及实现

1、补码除法算法

补码不恢复余数除法的规则。假设$[X]_{补}=X_S . X_1 X_2 \ldots\ldots Xn$，$[Y]_{补}=Y_S . Y_1 Y_2 \ldots\ldots Y_n$，Q是X÷Y的商，R是余数，若商采用最末位恒置“1”法（此时最大误差为$\pm 2^{-n}$），则补码除法运算的规则是：

① X和Y以补码形式参加除法运算，商也以补码的形式产生。余数和被除数、除数均采用双符号位。

② 当$[X]_{补}$与$[Y]_{补}$同号时，第一次做$[X]_{补}+[-Y]_{补}$操作，当异号时，第一次做$[X]_{补}+[Y]_{补}$操作，得到第一次的部分余数$[R_0]_{补}$。

二、补码除法及实现

补码除法运算的规则是：

③ **当$[R_i]_{补}$与$[Y]_{补}$同号时，上商1，然后余数先左移一位，加$[-Y]_{补}$得到新余数$[R_{i+1}]_{补}$；当$[R_i]_{补}$与[Y]补异号时，上商0，余数左移一位，加$[Y]_{补}$得到新余数$[R_{i+1}]_{补}$。**

④ **循环操作步骤③，共做n次，得到1位商符和（n-1）位商的补码数值位，最末位采用恒置“1”法。例如n=4位有效数值，连同第②步需做5次加法运算，4次移位操作。**

注：商采用最末位恒置“1”法时，因为商可能有误差$\pm 2^{-n}$，所以此时得到余数也不一定是精确的。

二、补码除法及实现

- [例] x=0.1001，y=−0.1011，用补码加减交替法计算 x÷y。

解： $[x]_{补}=0.1001$，$[y]_{补}=1.0101$，$[-y]_{补}=0.1011$

被除数/余数	商	
00.1001		
+ $[y]_{补}$ 11.0101		
11.1110		
← 11.1100	1	
+ $[-y]_{补}$ 00.1011		
00.0111		
← 00.1110	1.0	
+ $[y]_{补}$ 11.0101		
00.0011		
← 00.0110	1.00	
+ $[y]_{补}$ 11.0101		
11.1011		
← 11.0110	1.001	
+ $[-y]_{补}$ 00.1011		
00.0001		商的末位恒置 1
	1.0011	商左移 1 位，最后一步余数不左移

所以 $[x\div y]_{补}=1.0011$

$[余数]_{补}=0.00000001$

即 $x\div y=-0.1101$ 余数=0.00000001

二、补码除法及实现

- [例] x=-10110000，y=-1101，用补码加减交替法计算 x÷y。

解：$[x]_{补}$=101010000，$[y]_{补}$=10011，$[-y]_{补}$=01101

	被除数/余数	商	
	1101010000		
+ $[-y]_{补}$	001101		
	0000100000		
←	000100000	0	
+ $[y]_{补}$	110011		
	110111000		
←	10111000	01	
+ $[-y]_{补}$	001101		
	11101100		
←	1101100	011	
+ $[-y]_{补}$	001101		
	0000110		
←	000110	0110	
+ $[y]_{补}$	110011		
	111001		商的末位恒置 1
		01101	商左移 1 位，最后一步余数不左移

所以 $[x÷y]_{补}$=01101

$[余数]_{补}$=11001

即 x÷y=1101 余数=-0111

二、补码除法及实现

- 第二种方法（将求商符与求商值的规则统一）的运算规则为：

① X和Y以补码形式参加除法运算，商也以补码的形式产生。余数和被除数、除数均采用双符号位。部分余数初始为$[X]_{补}$，即$[R_0]_{补}=[X]_{补}$。

② 当$[R_i]_{补}$与$[Y]_{补}$同号时，上商1，然后余数先左移一位，加$[-Y]_{补}$得到新余数$[R_{i+1}]_{补}$；当$[R_i]_{补}$与$[Y]_{补}$异号时，上商0，余数左移一位，加$[Y]_{补}$得到新余数$[R_{i+1}]_{补}$。

③ 循环操作步骤②，共做n次，得到1位商符和（n-1）位商的补码数值位，最末位采用恒置“1”法。

注意：在操作完成后，必须将商的符号取反。

二、补码除法及实现

例 X=+0.1011，Y=−0.1101，用补码不恢复余数算法计算X÷Y。

解：$[X]_{补}$=00.1011　$[Y]_{补}$=11.0011　$[-Y]_{补}$=00.1101

第一种方法：

被除数/余数	商Q	操作说明
00.1011	0 0 0 0 0	$[X]_{补}$与$[Y]_{补}$异号
+ 11.0011		$+[Y]_{补}$
11.1110	0 0 0 0 1	$[R_0]_{补}$与$[Y]_{补}$同号，上商1
11.1100	0 0 0 1 0	左移一位
+ 00.1101		$+[-Y]_{补}$
00.1001	0 0 0 1 0	$[R_1]_{补}$与$[Y]_{补}$异号，上商0
01.0010	0 0 1 0 0	左移一位
+ 11.0011		$+[Y]_{补}$
00.0101	0 0 1 0 0	$[R_2]_{补}$与$[Y]_{补}$异号，上商0
00.1010	0 1 0 0 0	左移一位
+ 11.0011		$+[Y]_{补}$
11.1101	0 1 0 0 1	$[R_3]_{补}$与$[Y]_{补}$同号，上商1
11.1010	1 0 0 1 1	左移一位，末位置1

得$[Q]_{补}$=1.0011

Q=−0.1101

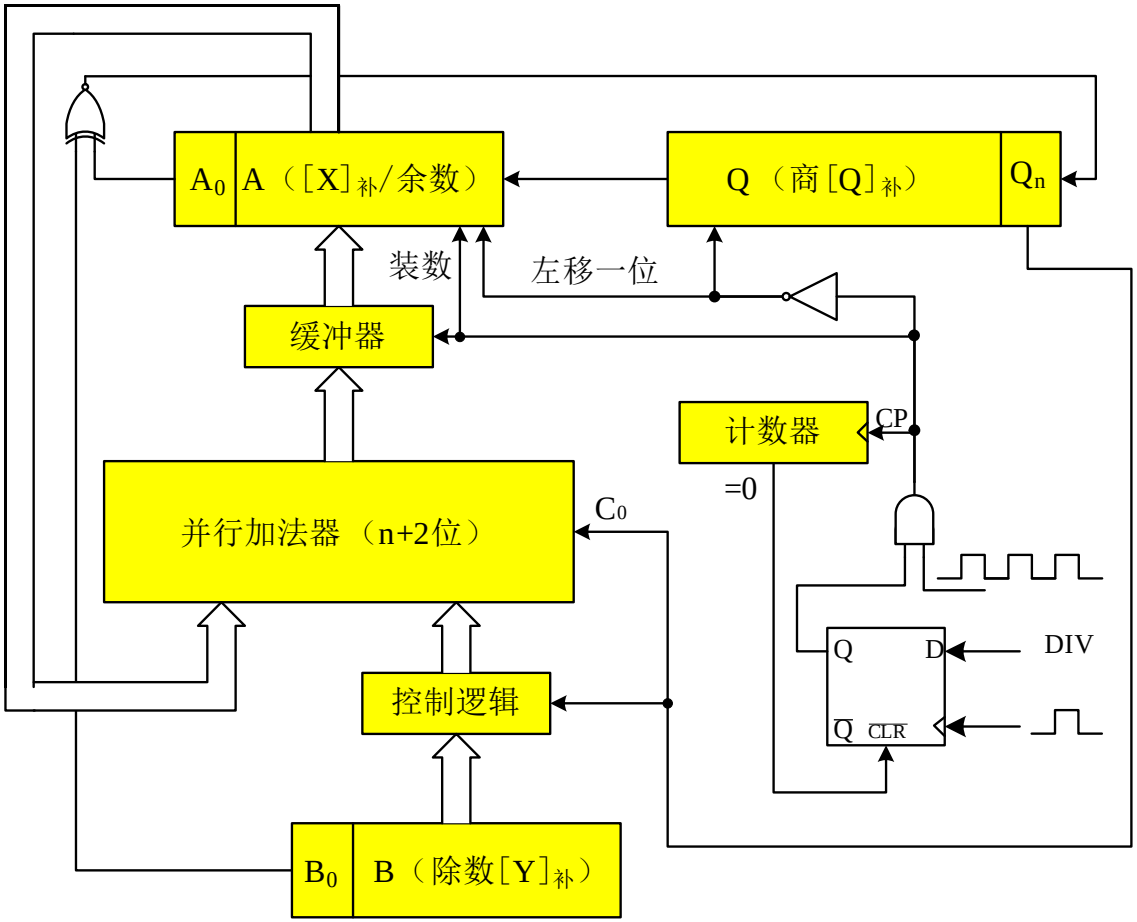
二、补码除法及实现

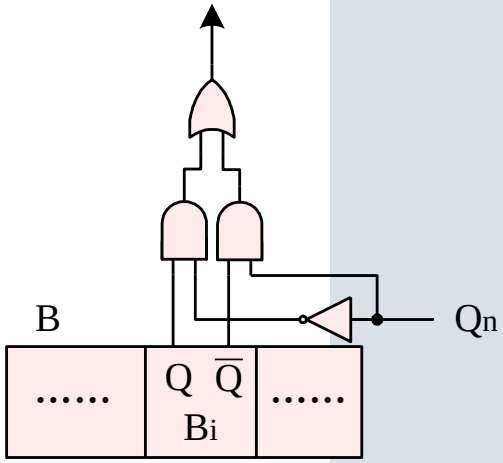
第二种方法：

被除数/余数	商Q	操作说明
00.1011	0 0 0 0 | 0	$[R_0]_{补}$与$[Y]_{补}$异号，上商0
01.0110	0 0 0 | 0 0	左移一位
+ 11.0011		$+[Y]_{补}$
00.1001	0 0 0 | 0 0	$[R_1]_{补}$与$[Y]_{补}$异号，上商0
01.0010	0 0 | 0 0 0	左移一位
+ 11.0011		$+[Y]_{补}$
00.0101	0 0 | 0 0 0	$[R_2]_{补}$与$[Y]_{补}$异号，上商0
00.1010	0 | 0 0 0 0	左移一位
+ 11.0011		$+[Y]_{补}$
11.1101	0 | 0 0 0 1	$[R_3]_{补}$与$[Y]_{补}$同号，上商1
11.1010	| 0 0 0 1 1	左移一位，末位置1

Q的符号取反，得$[Q]_{补}$=1.0011。

二、补码除法及实现



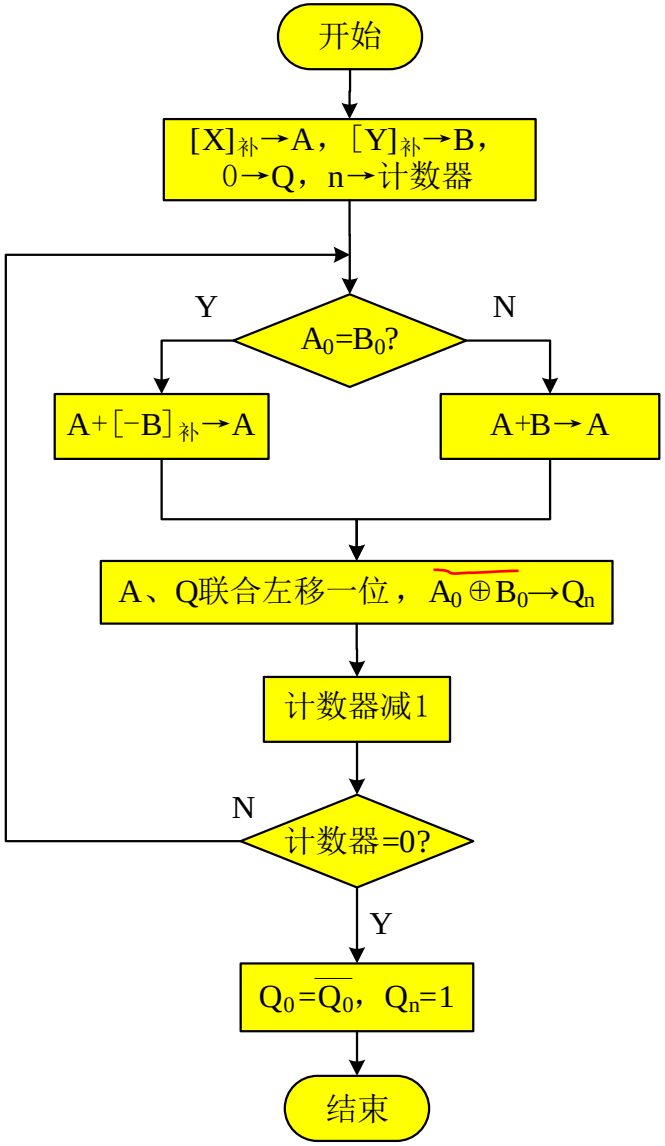


控制电路逻辑

补码不恢复余数除法的电路框图（第二种方法）

二、补码除法及实现

补码不恢复余数除法流程



4.4 定点运算器的组成与结构

一 定点运算器的组成

二 定点运算器的内部总线结构与通路

三 标志寄存器

一、定点运算器的组成

- 基本组成包括：
  - 算术逻辑运算单元ALU：核心部件
  - 暂存器：用来存放参与计算的数据及运算结果，它只对硬件设计者可见，即只被控制器硬件逻辑控制或微程序所访问，是程序员不可见的。
  - 通用寄存器堆：用于存放程序中用到的数据，它可以被软件设计者所访问。
  - 标志寄存器：用于记录运算器上次运算结果的状态，例如结果是否为0、是否有进位、是否溢出等，可以被软件设计者所访问。
  - 内部总线：用于连接各个部件的信息通道。
  - 其他可选电路：如多路选择器、移位器、三态缓冲门等。

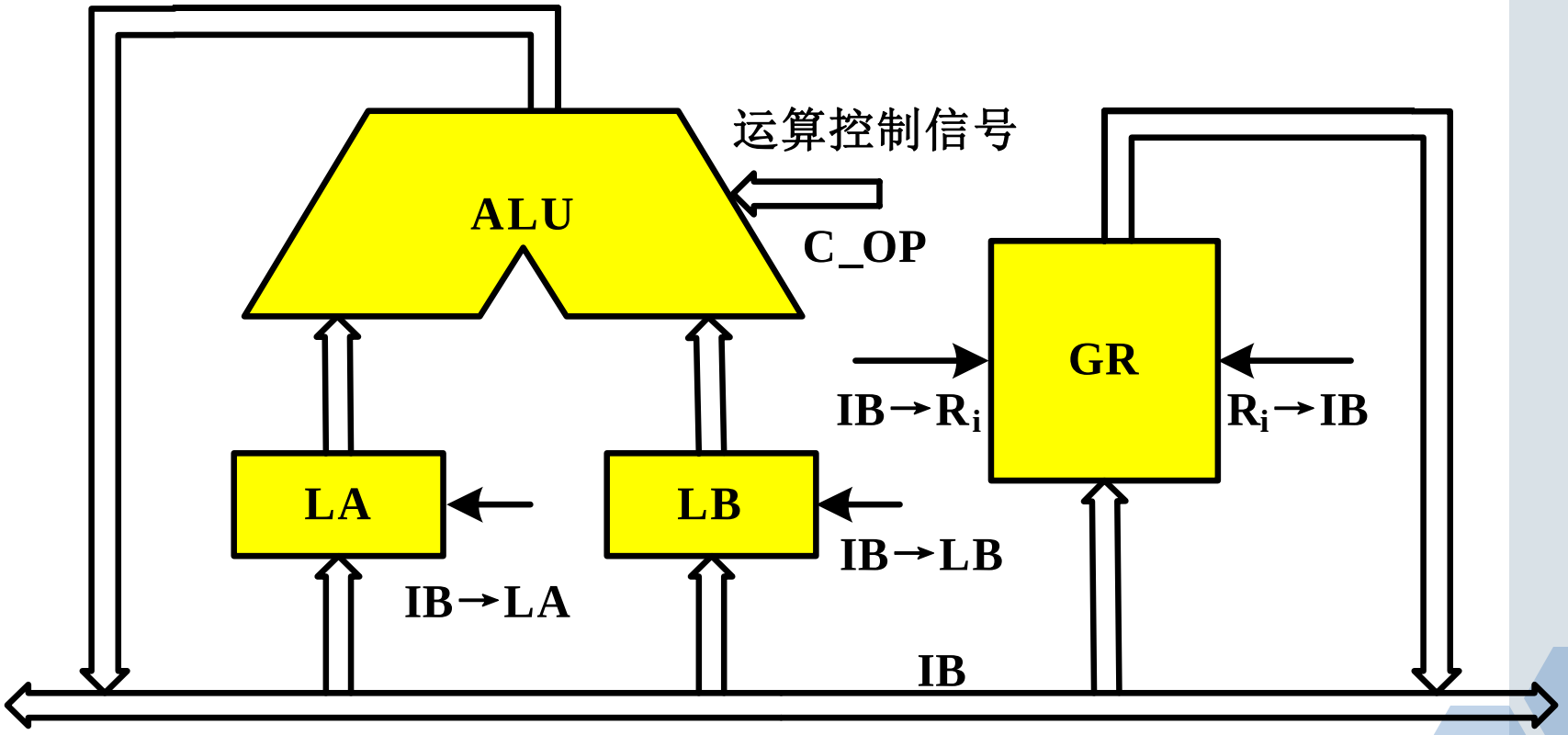
一、定点运算器的组成

- 设计定点运算器，如何确定各部件的功能和组织方式是关键，这取决于以下几个方面：
  - 指令系统
  - 机器字长
  - 机器数及其运算原理
  - 体系结构

二、定点运算器的总线结构

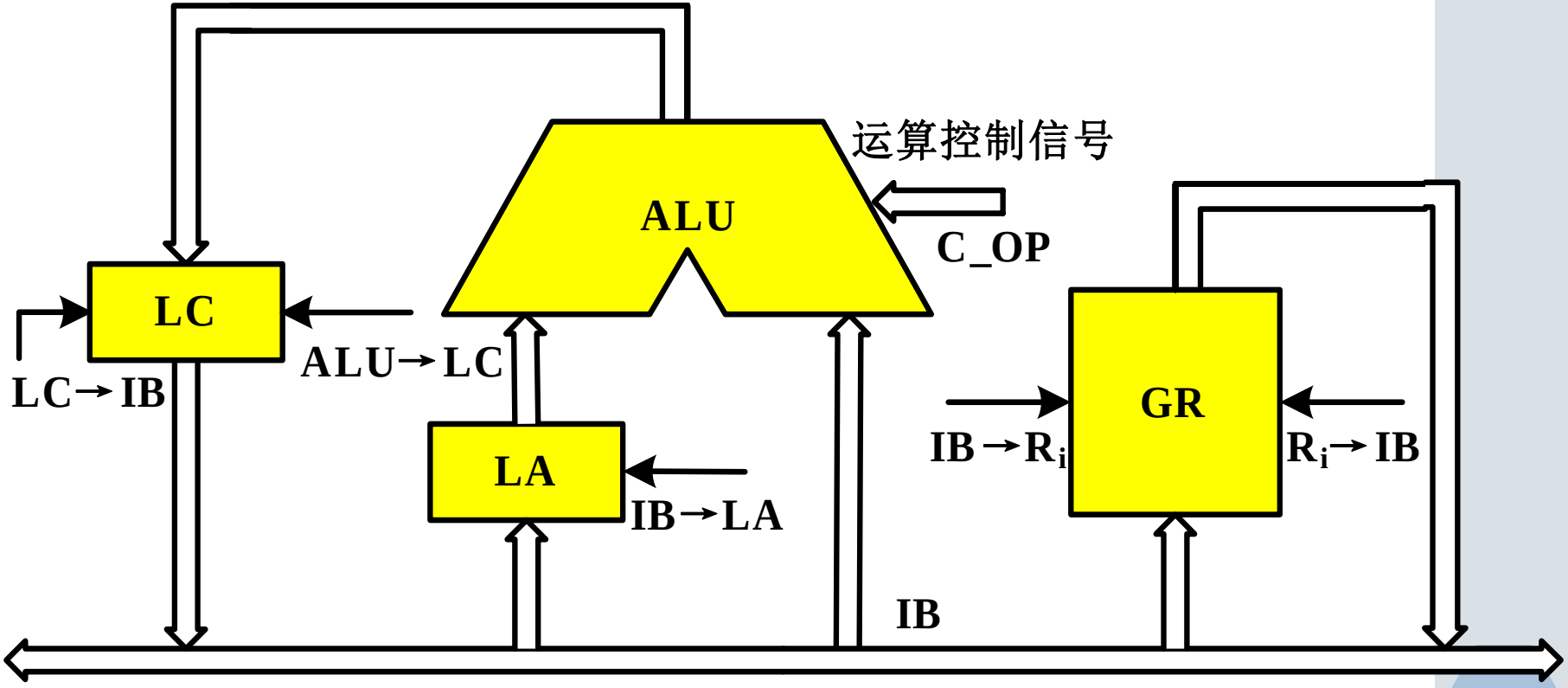
❖ 1、单总线结构

▪ 单总线运算器的结构形式1



二、定点运算器的总线结构

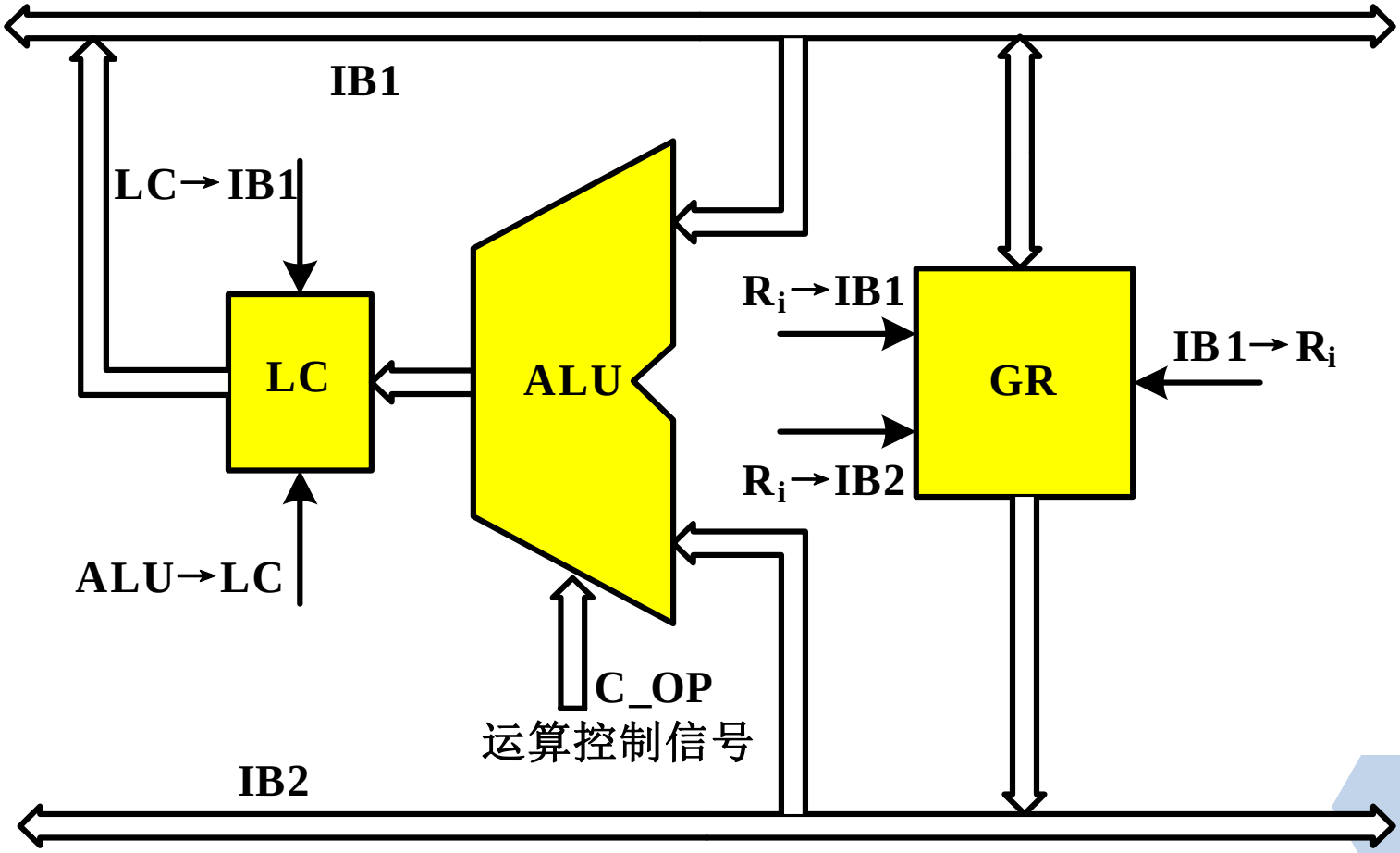
- 单总线运算器的结构形式2



二、定点运算器的总线结构

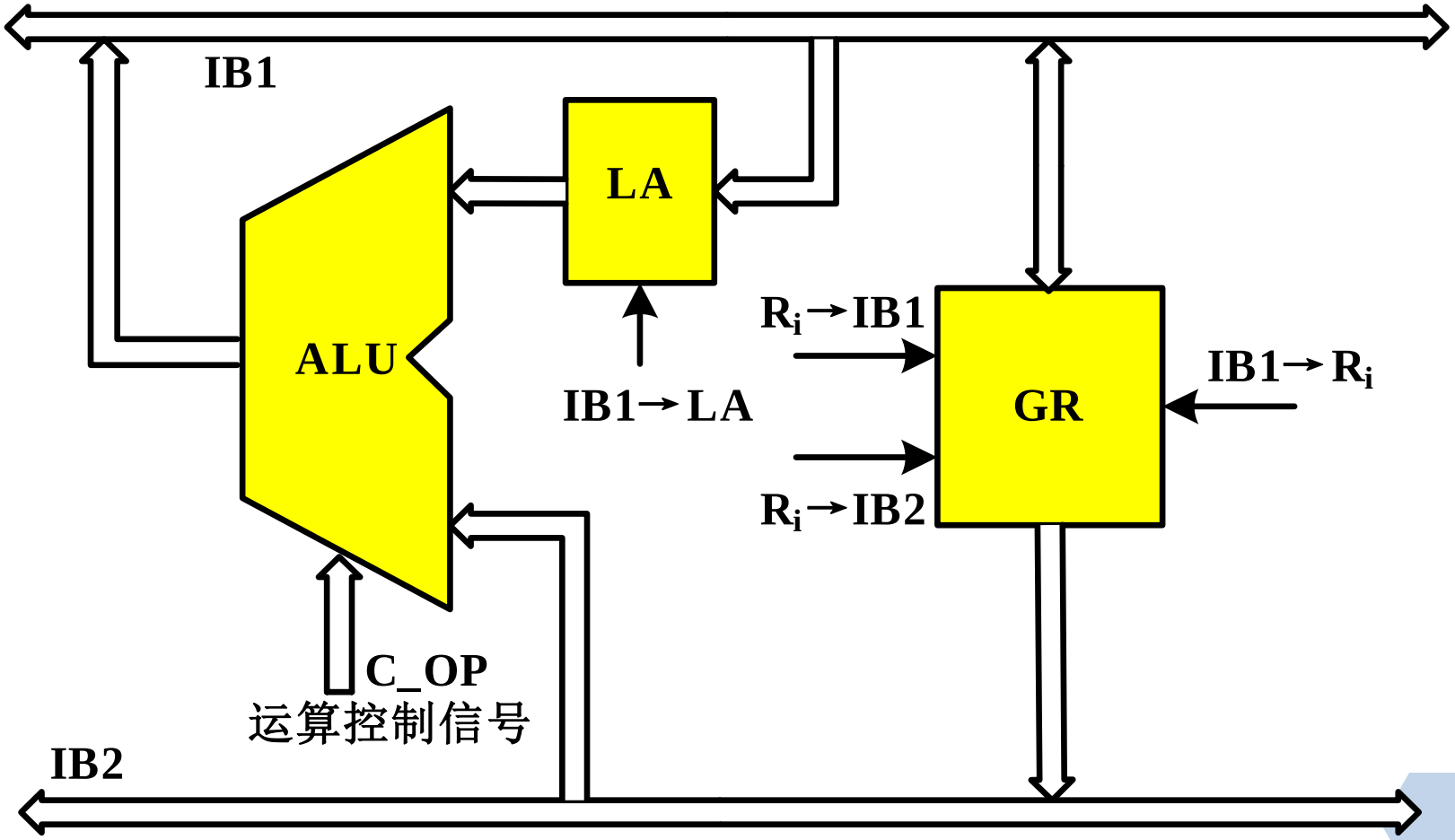
❖ 2、双总线结构

▪ 双总线运算器的结构形式1



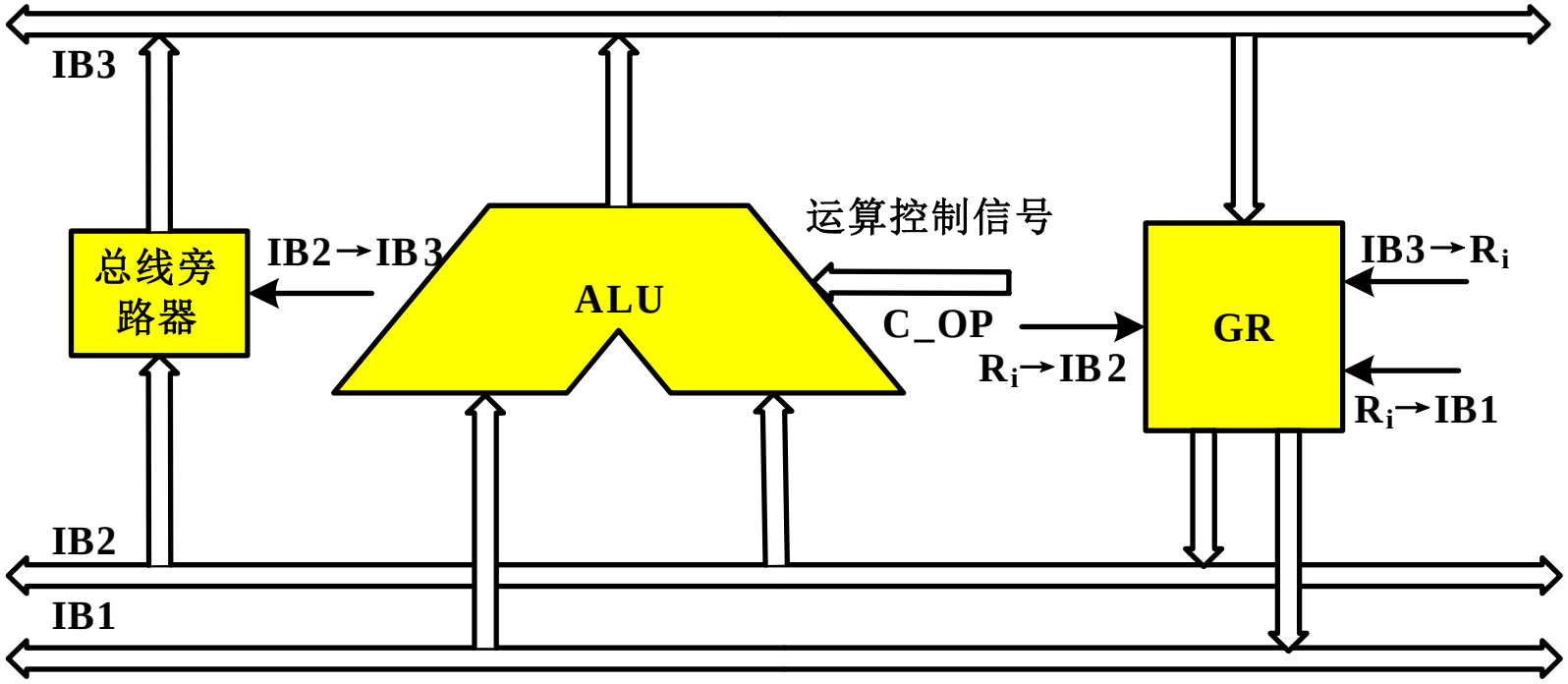
二、定点运算器的总线结构

- 双总线运算器的结构形式2



二、定点运算器的总线结构

3、三总线结构



最后必须指出的是，在分析某一种运算器的运算过程和通路时，一个基本的原则就是在一个CPU周期（一步）内，某条总线上的数据必须是唯一的，且不能保留（至下一个CPU周期）。

三、标志寄存器

- 标志寄存器用来保存ALU操作结果的某些状态，这种状态可作为外界对操作结果进行分析的一个依据，也可以用于判断程序是否要转移的条件，该寄存器通常也称为状态寄存器。
- 依据功能上的差别，不同的CPU，其标志寄存器中包含的标志也不尽相同。

三、标志寄存器

- 一般标志寄存器中包含了最基本的5种运算结果标志：
  - ZF　结果为零标志（zero flag bit）：记录运算结果是否为零的状态，运算结果为0（全零）时ZF置1，否则ZF置0。
  - CF　进位/借位标志位（carry flag bit）：记录最高位产生的进位C，加法运算时C=1则CF置1（表示有进位），否则置0；减法运算时C=0则CF置1（表示不够减，有借位），否则置0。CF标志只对无符号数运算有意义。
  - OF　溢出标志（overflow flag bit）：用于反映有符号数加减运算所得结果是否溢出。此时OF标志位为1，否则置0。OF标志只对带符号数运算有意义。

三、标志寄存器

- 一般标志寄存器中包含了最基本的5种运算结果标志：
  - SF　符号标志（sign flag bit），记录运算结果的符号，它与运算结果的最高位相同。在现代微机中，有符号数采用补码表示法，所以，SF也就反映运算结果的正负号。运算结果为正数时，SF置为0，否则其值为1。
  - PF　奇偶标志（parity flag bit），用于反映运算结果中“1”的个数的奇偶性，当结果操作数中“1”的个数为偶数时置1，否则置0。

- 每条指令执行完成后，是否要修改标志寄存器各个标志的值，取决于硬件设计者对指令功能的设计与控制。

三、标志寄存器（举例）

- 例 通过以下两条指令将80H加80H后送到AL，试写出运算结果及其标志位。
- MOV　　AL，80H
- ADD　　AL，80H
- ADD指令运算结束后：（AL）=00H；
- ZF=1：因为运算结果为全零；
- CF=1：因为加法运算的最高位产生了进位，表明无符号数加运算发生溢出。实际上是：把操作数80H和80H均当作无符号数128和128，做加法运算的正确结果为256，超出了8位无符号数的表示范围（0~255）。
- OF=1：因为C1⊕Cf=1，按照单符号判溢方法，表明有符号数运算发生溢出。实际上是：把操作数80H和80H均当作有符号数（补码）-128（-2^7）和-128，做加法运算的正确结果为-256，超出了8位补码机器数的表示范围（-128~+127）。
- SF=0：因为运算结果的最高位为0。显然，因为OF=1（发生了溢出），所以SF是错误的。
- PF=1：结果中“1”的个数为0个，所以PF=1。

4.5 浮点运算及运算器

一 浮点加减运算

二 浮点乘法运算

三 浮点除法运算

四 浮点运算器

一、浮点加减运算

- 假设两个浮点数X和Y

$$X = M_X \times 2^{E_X} \qquad Y = M_Y \times 2^{E_Y}$$

- 则必须保证X和Y的阶码（指数）是相同的，然后对尾数做加减运算。

$$Z = X \pm Y = (M_X \bullet 2^{(E_X - E_Y)} \pm M_Y) \times 2^{E_Y}$$

$$E_X \le E_Y$$

浮点加减运算步骤

- （1）0操作数检查：以尽可能的简化操作。
- （2）对阶：将参加浮点加减运算的两个操作数的阶码变为相等的过程。原则是小阶对向大阶
  - 求阶差 $\Delta E=E_X-E_Y$，若 $\Delta E\neq 0$，即 $E_X\neq E_Y$ 时需要对阶。
  - 若 $\Delta E>0$，则 $E_X>E_Y$，M_Y 每右移一位，E_Y+1，直至 $E_Y=E_X$ 。
  - 若 $\Delta E<0$，则 $E_X<E_Y$，M_X 每右移一位，E_X+1，直至 $E_X=E_Y$ 。
- （3）尾数相加减

浮点加减运算步骤

- （4）**结果规格化**：尾数运算的结果可能出现两种非规格化情况：
  - A、尾数溢出：需要右规（1次），即尾数右移1位，阶码＋1
  - B、|尾数| $<2^{-1}$：需要左规，即尾数左移1位，阶码－1，左规可能多次，直到尾数变为规格化形式。
  - 注意：在右规和左规的过程中，由于阶码的增减，阶码可能出现正溢出（右规时）和负溢出（左规时）两种情况，必须报告溢出。
- （5）**舍入**：可采用截断法、0舍1入法、末位恒置1。

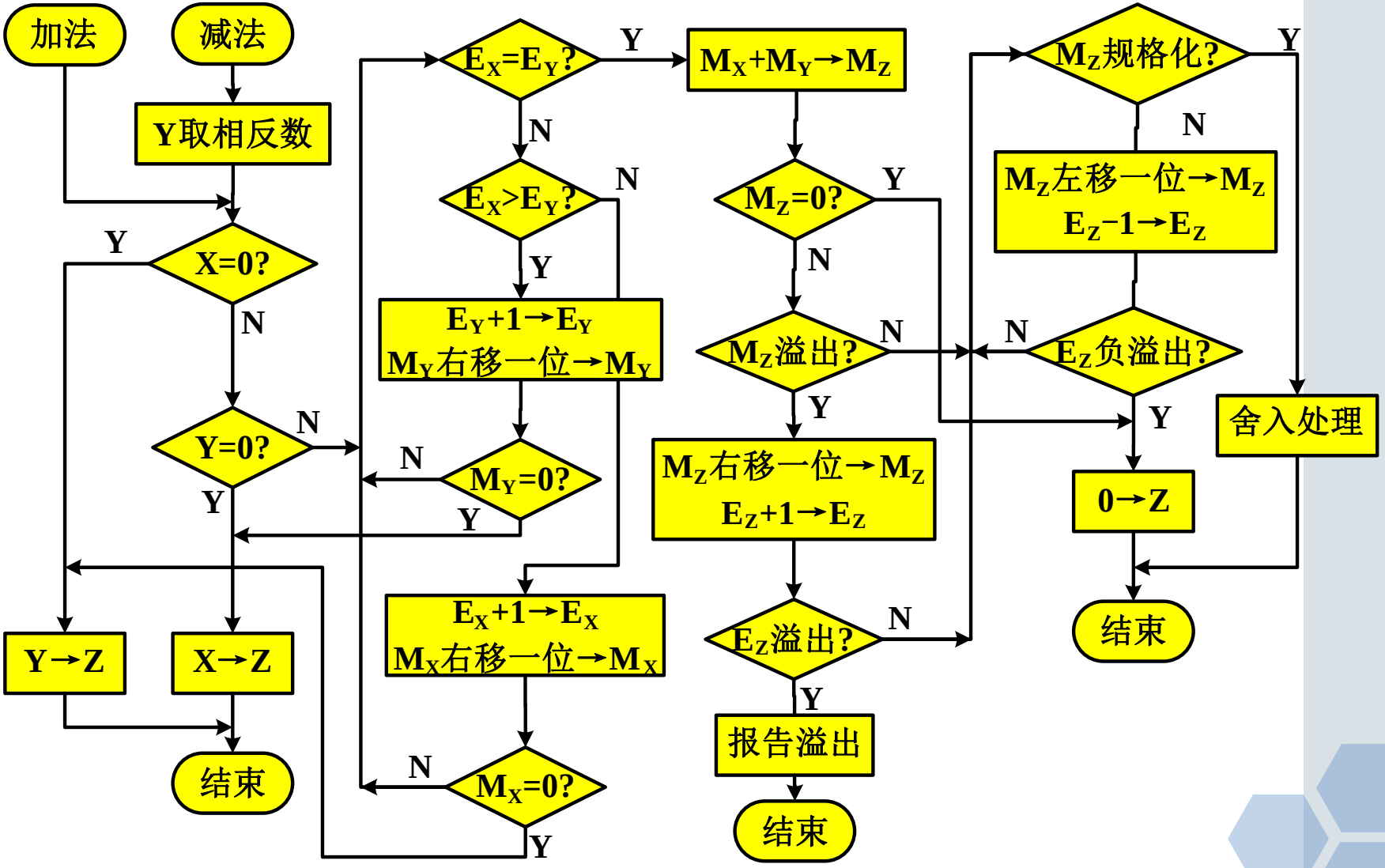
一、浮点加减运算

- IEEE 754标准规定了4种可选的舍入模式：
  - 向上舍入（总是朝+∞）：为正数时，只要多余位不全为0，就向最低有效位进1；为负数时，则采用简单的截断法。
  - 向下舍入（总是朝－∞）：为正数时，只要多余位不全为0，就简单地截尾；为负数时，则向最低有效位进1。
  - 向0舍入：即朝数轴的原点方向舍入，就是无论正数还是负数，都采用简单的截尾，从而使得绝对值总是变小。这种方法容易累积误差。

一、浮点加减运算

- 就近舍入：即舍入到最接近的数，就是通常的“四舍五入”。当多余位的值超出它们量程（即最低有效位的权值）的一半，则向最低有效位进1；当小于一半，则截尾（即舍去）；当等于一半（中点），则若最低有效位为0（偶数）就截尾，若最低有效位为1（奇数）就进1，以使得最低有效位总是为0。在中点的这种舍入方法很公平：一半的时间里向上舍入，在另一半的时间里向下舍入，它也不容易累积误差，所以被广泛使用。

浮点加减运算流程


加法
减法
Y取相反数
X=0?
Y
N
Y=0?
N
Y
Y→Z
X→Z
结束
$E_X=E_Y$?
Y
N
$E_X>E_Y$?
N
Y
$E_Y+1\rightarrow E_Y$
M_Y右移一位→M_Y
M_Y=0?
N
Y
$E_X+1\rightarrow E_X$
M_X右移一位→M_X
M_X=0?
N
Y
$M_X+M_Y\rightarrow M_Z$
M_Z=0?
Y
N
M_Z溢出?
N
Y
M_Z右移一位→M_Z
$E_Z+1\rightarrow E_Z$
E_Z溢出?
N
Y
报告溢出
结束
M_Z规格化?
Y
N
M_Z左移一位→M_Z
$E_Z-1\rightarrow E_Z$
E_Z负溢出?
N
Y
0→Z
结束
舍入处理

一、浮点加减运算

举例：12位浮点数，阶码4位，包含1位阶符，尾数8位，包含1位数符，用补码表示，阶码在前，尾数（包括数符）在后，已知：

$X=(-0.1001011)\times 2^{001}$　　$Y=0.1100101\times 2^{-010}$

求Z=X+Y。

- 解：$[X]_{浮}$ = 00，001　11.0110101　　$[Y]_{浮}$ = 11，110　00.1100101
- （1）对阶
  - $\Delta E=E_X-E_Y=[E_X]_{补}+[-E_Y]_{补}$ = 00，001+00，010 = 00，011
  - $\Delta E=3>0$，将M_Y右移3位，E_Y加3：
  - $[Y]_{浮}$ = 00，001　00.0001100　（101）

一、浮点加减运算

- （2）尾数相加：　$[M_Z]_{补}$ = 11.1000001（101）
- （3）结果规格化：左规一位，无溢出：
  - $[M_Z]_{补}$ = 11.0000011（01）
  - $[E_Z]_{补}$ = 00，001 + 11，111= 00，000
- （4）舍入：按照0舍1入法，尾数多余位舍去
- 结果为：$[Z]_{浮}$ = 0，000　1.0000011

二、浮点乘法运算

- 假设两个浮点数X和Y：

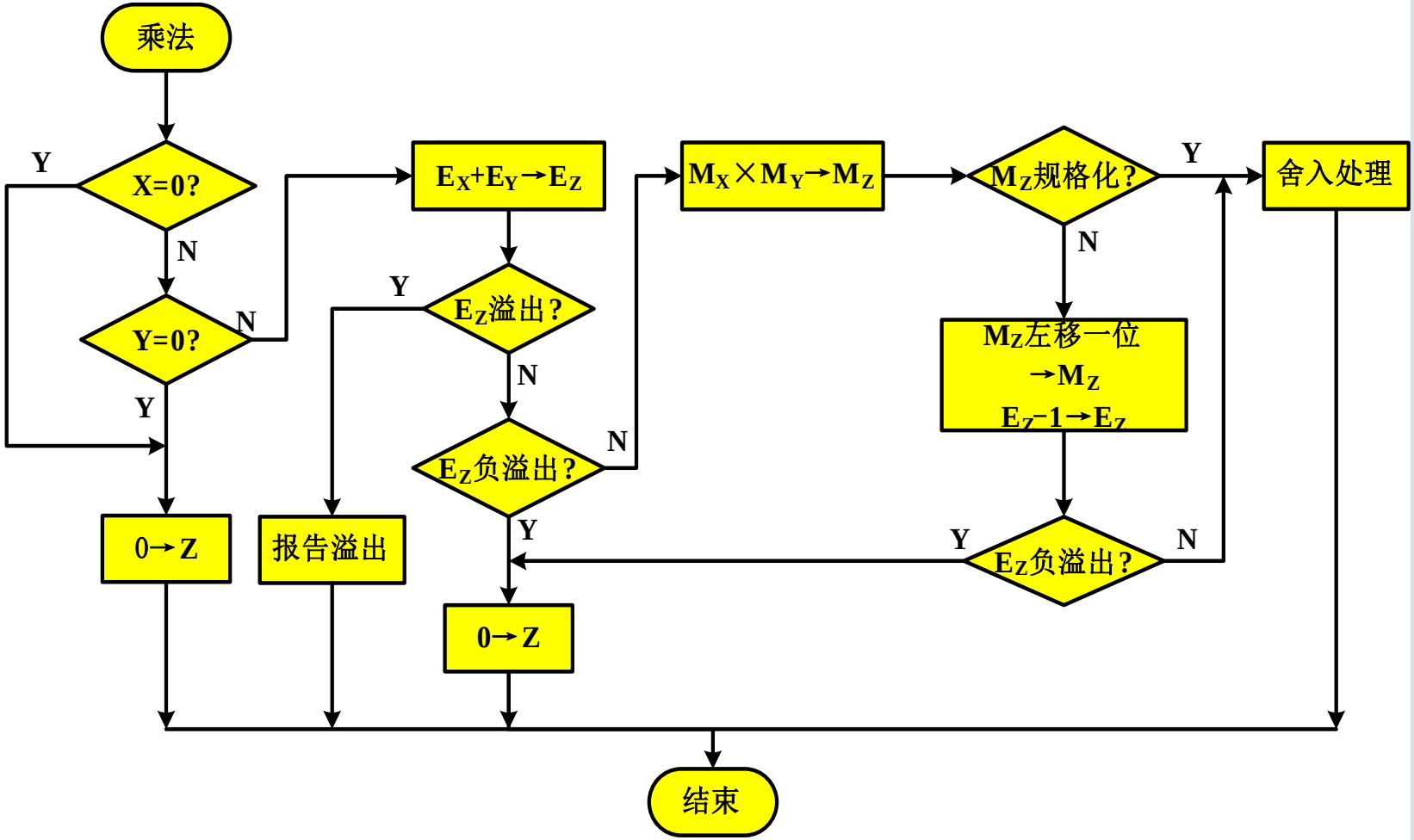
$$X = M_X \times 2^{E_X} \qquad Y = M_Y \times 2^{E_Y}$$

$$Z = X \times Y = (M_X \bullet M_Y) \times 2^{(E_X + E_Y)}$$

浮点乘法运算步骤

- （1）0操作数检查
- （2）阶码相加：阶码相加可以采用补码或者移码的定点整数加法，同时对相加结果判溢，一旦发生正溢出，则需报告溢出，若发生负溢出，则将结果置为机器零。
- （3）尾数相乘
- （4）结果规格化：可能需要左规1位
- （5）舍入处理：尾数相乘的结果长度是尾数长度的两倍，必须对低位舍入。

浮点数乘法运算流程


乘法
X=0?
Y
N
Y=0?
N
Y
0→Z
$E_X+E_Y→E_Z$
Y
E_Z溢出?
N
报告溢出
N
E_Z负溢出?
Y
0→Z
$M_X×M_Y→M_Z$
M_Z规格化?
Y
N
M_Z左移一位
→M_Z
$E_Z-1→E_Z$
Y
E_Z负溢出?
N
舍入处理
结束

二、浮点乘法运算（举例）

- 一浮点数表示格式为：10位浮点数，阶码4位，包含1位阶符，用移码表示，尾数6位，包含1位数符，用补码表示，阶码在前，尾数（包括数符）在后，已知：

X=（-0.11001）×2^{011}　　Y=0.10011×2^{-001}，求Z=X·Y。要求阶码用移码计算，尾数用补码Booth算法计算。

- 解：按照浮点数的格式分别写出它们的表示形式，为计算方便，阶码采用双符号位移码，尾数采用双符号位补码：
- $[X]_{浮}$ = 01，011　　11.00111
- $[Y]_{浮}$ = 00，111　　00.10011

二、浮点乘法运算（举例）

1. 阶码相加

$[EZ]_{移} = [EX]_{移} + [EY]_{补} =$ 01，011 + 11，111 = 01，010

结果无溢出，$[EZ]_{移}$ =1，010。

2. 尾数相乘

采用补码Booth算法计算 $[MX \cdot MY]_{补}$，首先写出下例数据：

$[MX]_{补} = 11.00111$

$[MY]_{补} = 0.10011$

$[-MX]_{补} = 00.11001$

$[MZ]_{补} = 1.10001 \quad 00101$

部分积	乘数Y ($Y_n Y_{n+1}$)	操作说明
00.00000	0.1 0 0 1 1 0	Y_5Y_6=10，+[-X]补
+ 00.11001		
00.11001		
00.01100	1 0.1 0 0 1 1	右移一位
+ 00.00000		Y_4Y_5=11，+0
00.01100		
00.00110	0 1 0.1 0 0 1	右移一位
+ 11.00111		Y_3Y_4=01，+[X]补
11.01101		
11.10110	1 0 1 0.1 0 0	右移一位
+ 00.00000		Y_2Y_3=00，+0
11.10110		
11.11011	0 1 0 1 0.1 0	右移一位
+ 00.11001		Y_1Y_2=10，+[-X]补
00.10100		
00.01010	0 0 1 0 1 0.1	右移一位
+ 11.00111		Y_0Y_1=01，+[X]补
11.10001	0 0 1 0 1	

二、浮点乘法运算（举例）

3. 结果规格化

MZ左规一次得：$[MZ]_{补}$= 1.00010　01010

EZ减1得：

$[EZ]_{移}$ = 01，010 + 11，111 =01，001

4. 舍入

对尾数MZ进行0舍1入，最后得

$[Z]_{浮}$ = 1，001　1.00010

三、浮点除法运算

- 假设两个浮点数X和Y：

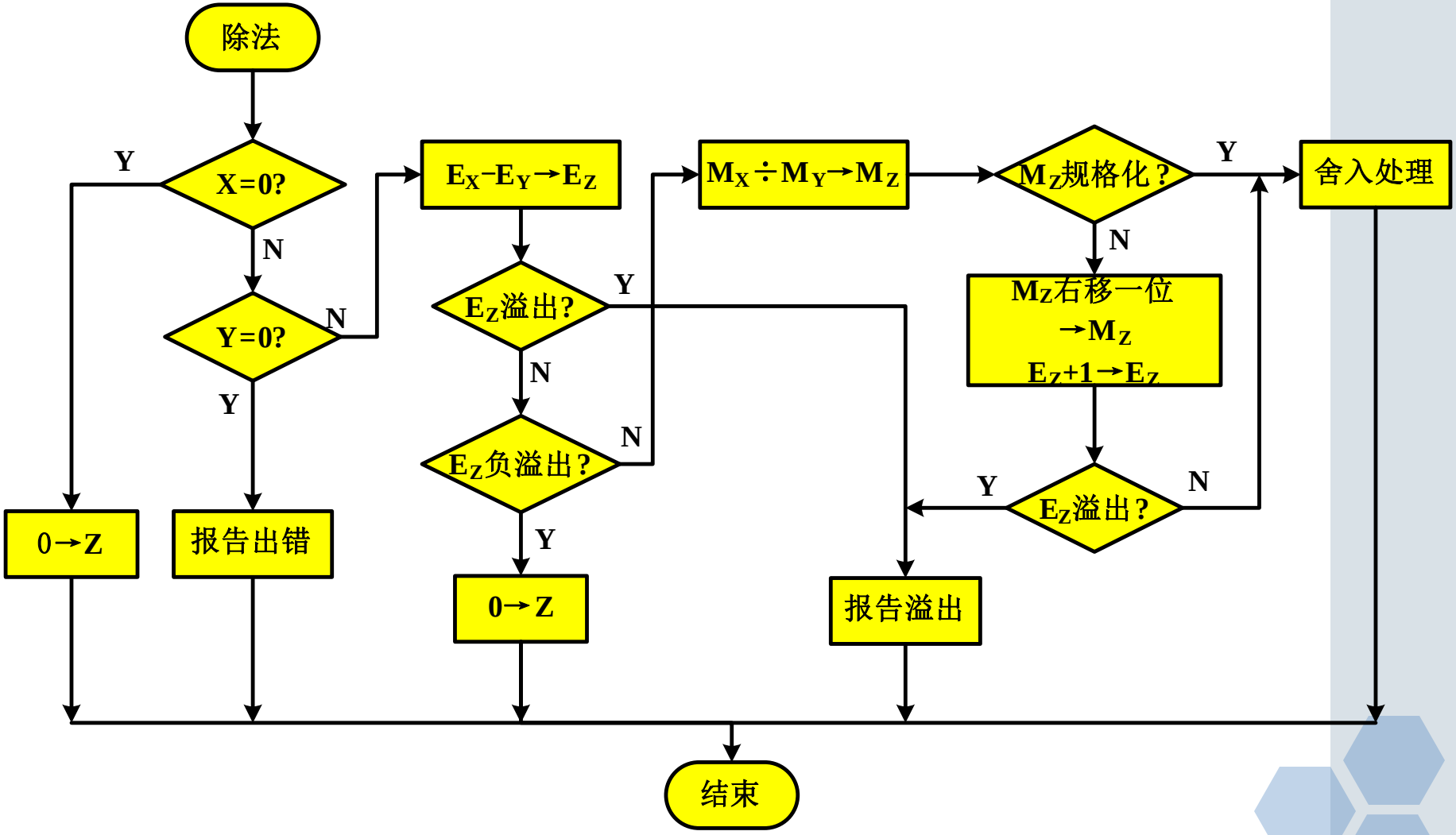
$$X = M_X \times 2^{E_X} \qquad Y = M_Y \times 2^{E_Y}$$

$$Z = X \div Y = (M_X \div M_Y) \times 2^{(E_X - E_Y)}$$

浮点数除法运算步骤

- （1）0操作数检查
  - 当除数为0，则报告除法出错，或者结果（商）无穷大；当被除数为0，则商为0。
- （2）阶码相减
  - 阶码相减的结果也可能溢出，若发生正溢出，则需报告浮点数溢出，若发生负溢出，则将结果置为机器零。
- （3）尾数相除
- （4）结果规格化
- （5）舍入处理

浮点数除法运算流程


除法
X=0?
Y
N
Y=0?
N
Y
0→Z
报告出错
EX−EY→EZ
EZ溢出?
Y
N
EZ负溢出?
N
Y
0→Z
MX÷MY→MZ
MZ规格化?
Y
N
MZ右移一位
→MZ
EZ+1→EZ
EZ溢出?
Y
N
报告溢出
舍入处理
结束

三、浮点除法运算（举例）

- 一浮点数表示格式为：10位浮点数，阶码4位，包含1位阶符，尾数6位，包含1位数符，用补码表示，阶码在前，尾数（包括数符）在后，已知：

X=（-0.11001）×2^{011}　　Y=0.10011×2^{-001}求Z=X÷Y。要求阶码用移码计算，尾数用原码加减交替除法计算。

- 解：按照浮点数的格式分别写出它们的表示形式为：

$[X]_{浮}$ = 1，011　1.00111　$[Y]_{浮}$ = 0，111　0.10011

1. 阶码相减

$[EZ]_{移}$ = $[EX]_{移}$ +$[-EY]_{补}$ = 01，011 + 00，001 = 01，100

三、浮点除法运算（举例）

2. 尾数相除

采用原码加减交替法计算 |MX| ÷|MY|，首先写出下例数据：

|MX| = 00.11001

|MY| = 00.10011

$[-|MY|]_{补}$ = 11.01101

|MZ| = |MX|÷|MY|

= 1.01010

被除数 余数	商Q	操作说明
00.11001	0 0 0 0 0 | 0	
+ 11.01101		+$[-\|M_Y\|]_{补}$
00.00110	0 0 0 0 | 0 1	R_0>0，上商1
00.01100	0 0 0 | 0 1.0	左移一位
+ 11.01101		+$[-\|M_Y\|]_{补}$
11.11001	0 0 0 | 0 1.0	R_1<0，上商0
11.10010	0 0 | 0 1.0 0	左移一位
+ 00.10011		+$\|M_Y\|$
00.00101	0 0 | 0 1.0 1	R_2>0，上商1
00.01010	0 | 0 1.0 1 0	左移一位
+ 11.01101		+$[-\|M_Y\|]_{补}$
11.10111	0 | 0 1.0 1 0	R_3<0，上商0
11.01110	| 0 1.0 1 0 0	左移一位
+ 00.10011		+$\|M_Y\|$
00.00001	| 0 1.0 1 0 1	R_4>0，上商1
00.00010	1.0 1 0 1 0	左移一位
+ 11.01101		+$[-\|M_Y\|]_{补}$
11.01111	1.0 1 0 1 0	R_5<0，上商0
+ 00.10011		+$\|M_Y\|$恢复余数
00.00010		

三、浮点除法运算（举例）

3. 结果规格化

由于|MX| >|MY|，所以|MZ|>1，必须右规一位，得|MZ| = 0.10101　0

EZ加1得：[EZ]移 = 01，100 + 00，001 = 01，101

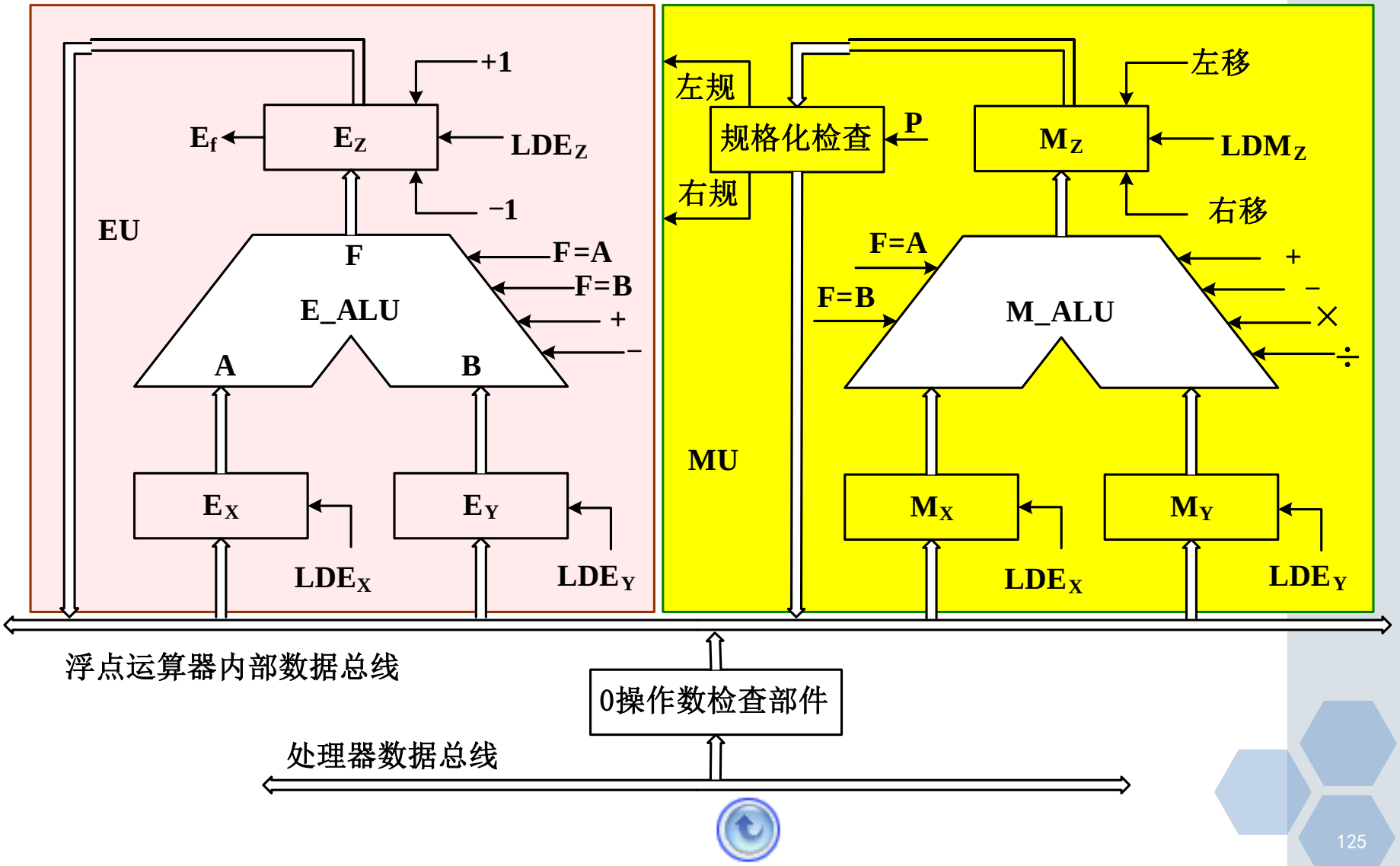
4. 舍入

对|MZ|进行0舍1入，得|MZ| = 0.10101

[MZ]原 = 1.10101　[MZ]补 = 1.01011

最后：[Z]浮 = 1，101　1.01011

四、浮点运算器


EU
+1
−1
E_f
E_Z
LDE_Z
F
F=A
F=B
+
−
E_ALU
A
B
E_X
E_Y
LDE_X
LDE_Y
MU
左规
右规
规格化检查
P
左移
右移
M_Z
LDM_Z
F=A
F=B
M_ALU
+
−
×
÷
M_X
M_Y
LDE_X
LDE_Y
浮点运算器内部数据总线
0操作数检查部件
处理器数据总线

本章小结

- 定点机器数的加减法运算通常通过补码来实现。补码的加减运算规则使得计算机中的减法转化为加法来运算，方便了硬件设计。
- 定点机器数的乘、除法运算则可以采用原码和补码来实现。乘、除法器件可以采用基于串行乘、除法算法的乘法器，也可以采用高速的阵列乘法器。
- 浮点数的表示和运算均基于定点数的表示和运算。浮点运算器由阶码运算部件和尾数运算部件两部分构成。
- 本章重点为定点数和浮点数的运算方法。

The End!