第三章　计算机的算术运算

3.1 引言

本章的核心内容

- 四则运算
  - 加减
  - 乘除
  - 溢出处理（当运算生成一个绝对值太大或太小的数时）
- 浮点数
  - 表示方法和运算

3.2 加法和减法

- 计算机中的加/减法都是用补码加/减法实现的。
- 补码加法

$$[X+Y]_{补}=[X]_{补}+[Y]_{补} \quad (\text{mod } M)$$

其中补码是以M为模的。对于纯小数模为2。对于n位（含1位符号位）纯整数，模为2^n。

补码加法运算的规则

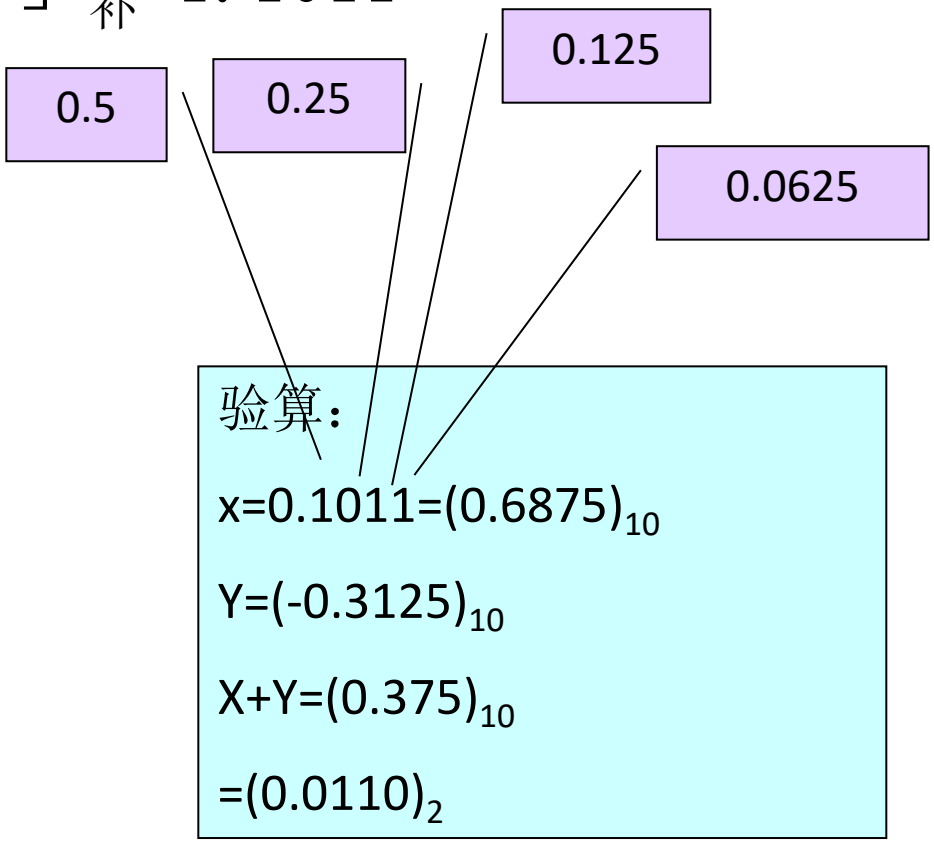
- 两数不管正负，均用补码表示
- 符号位应当作数值参加运算
- 按二进制运算规则，逢2进1
- 符号位相加所产生的进位要丢掉(以M为模)
- 结果为补码(其符号即为和的正确符号)

［例1］ x=+0.1011，y=−0.0101，求x+y=?

解：［x］$_{补}$=0.1011，［y］$_{补}$=1.1011

$$
\begin{array}{rl}
[x]_{补} & = \quad 0.1011 \\
+)\ [y]_{补} & = \quad 1.1011 \\
\hline
[x+y]_{补} & = 1\ 0.0110
\end{array}
$$

丢掉



得［x+y］$_{补}$=0.0110，x+y=+0.0110

［例2］X=-11001，Y=-00011，求X+Y=?

解：［x］$_{补}$=1,00111，［y］$_{补}$=1,11101

$$
\begin{array}{rl}
[x]_{补} = & 1,00111 \\
+)\ [y]_{补} = & 1,11101 \\
\hline
[x+y]_{补} = & 1\ 1,00100
\end{array}
$$

丢掉

验算：

$x=-11001=(-25)_{10}$

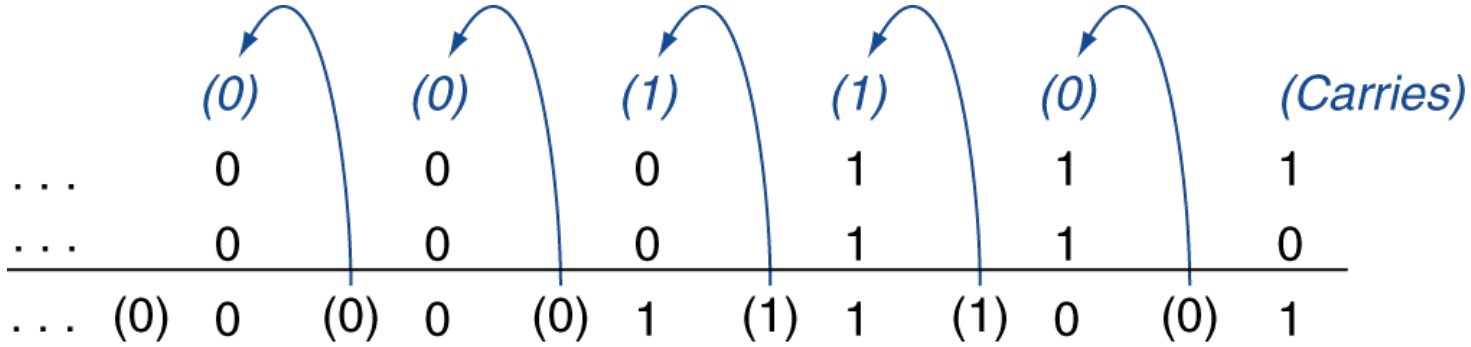
$Y=(-3)_{10}$

$X+Y=(-28)_{10}$

$=(-11100)_{2}$

得到［x+y］$_{补}$=1,00100，结果x+y=-11100

[例3]： 7 + 6



- 可能产生溢出的情况：
  - +ve 和 –ve 不会溢出
  - 两个 +ve
    - 若结果符号位为 1，溢出。（负数）
  - 两个 –ve
    - 若结果符号位为 0，溢出。（正数）

- 补码减法

  根据补码加法公式可推出：

  $[x-y]_{补}=[x+(-y)]_{补}=[x]_{补}+[-y]_{补} \pmod{M}$

  因此，求得$[-y]_{补}$就可以变减法为加法。

- 回顾，变补运算：

  已知$[y]_{补}$求$[-y]_{补}$：

  将$[y]_{补}$连同符号位一起求反，末位加“1”(定点小数中这个“1”是2^{-n})。

［例1］x=-0.1101，y=-0.0110，求x-y=?

解：　［x］$_{补}$=1.0011，

［y］补=1.1010，［-y］补=0.0110

［x］补 = 1.0011

+)［-y］补 = 0.0110

［x-y］补 = 1.1001

［x-y］补=1.1001，x-y=-0.0111

验算：

x=-0.1101

$=(-0.8125)_{10}$

$Y=(-0.375)_{10}$

$X-Y=(-0.4375)_{10}$

$=(-0.0111)_{2}$

- [例2]: 7 – 6 = 7 + (–6)

  +7: 0000 0000 … 0000 0111

  –6: 1111 1111 … 1111 1010

  +1: 0000 0000 … 0000 0001

- 可能产生溢出的情况
  - 两个 +ve 或两个–ve 不会溢出
  - 异号数相减，若结果与被减数相反，则溢出
    - (–ve) – (+ve)
      - 溢出若结果符号位为 0（正数）
    - (+ve) – (-ve)
      - 溢出若结果符号位为 1（负数）

- 加减法的溢出判断的逻辑表达式：

$$V = \overline{X}_f \overline{Y}_f S_f + X_f Y_f \overline{S}_f$$

V=1代表溢出

X_f、Y_f、S_f分别代表相加的两个操作数的符号和结果的符号

- 双符号位的溢出判断方法：

每个操作数采用双符号位（S_{f1}和S_{f2}），称为变形补码。用“00”表示正数，“11”表示负数。两个符号位同时参加运算。

左边的符号位S_{f1}叫做真符，因为它代表了该数真正的符号，双符号位的含义如下：

$S_{f1}S_{f2}$ = 00	结果为正数，无溢出
$S_{f1}S_{f2}$ = 01	结果正溢
$S_{f1}S_{f2}$ = 10	结果负溢
$S_{f1}S_{f2}$ = 11	结果为负数，无溢出

存储时，仍然只存一个符号位，只是运算使用两个符号位。

不是前面的变补运算,变补运算用于减法的减数

[例3.7] x=-1011，y=-1100，求$[x+y]_{补}$=?

解 $[x]_{变形补码}$=11,0101，$[y]_{补}$=11,0100

$$
\begin{array}{rl}
[x]_{补} = & 11,0101 \\
+)\ [y]_{补} = & 11,0100 \\
\hline
[x+y]_{补} = & \underline{1}\,10,1001
\end{array}
$$

丢失

结果的两符号位不同，表示产生溢出(负溢出)。

验算：$x=-(11)_{10}$

$Y=(-12)_{10}$

$X+Y=(-23)_{10}$

5位二进制补码最大负数10000=-16>-23

补码加减运算规则：

(1) 参加运算的两个操作数均用补码表示；

(2) 符号位作为数的一部分参加运算；

(3) 若做加法，则两数直接相加。若做减法，则将被减数与减数的负数的机器数相加；

(4) 运算结果用补码表示；

(5) 符号位的进位为模值，应该丢掉。

无符号整数的溢出

- 无符号整数通常表示内存地址，溢出可忽略。为此，MIPS设置不同的指令：

加法（add）
立即数加法（addi）
减法（sub）

溢出产生异常

无符号加法（addu）
立即数无符号加法（addiu）
无符号减法（subu）

溢出不产生异常

- 异常/中断的定义　P137（旧）P119新

补码定点加减运算的实现（附录c）

$X+(Y \oplus SUB)+C_0$

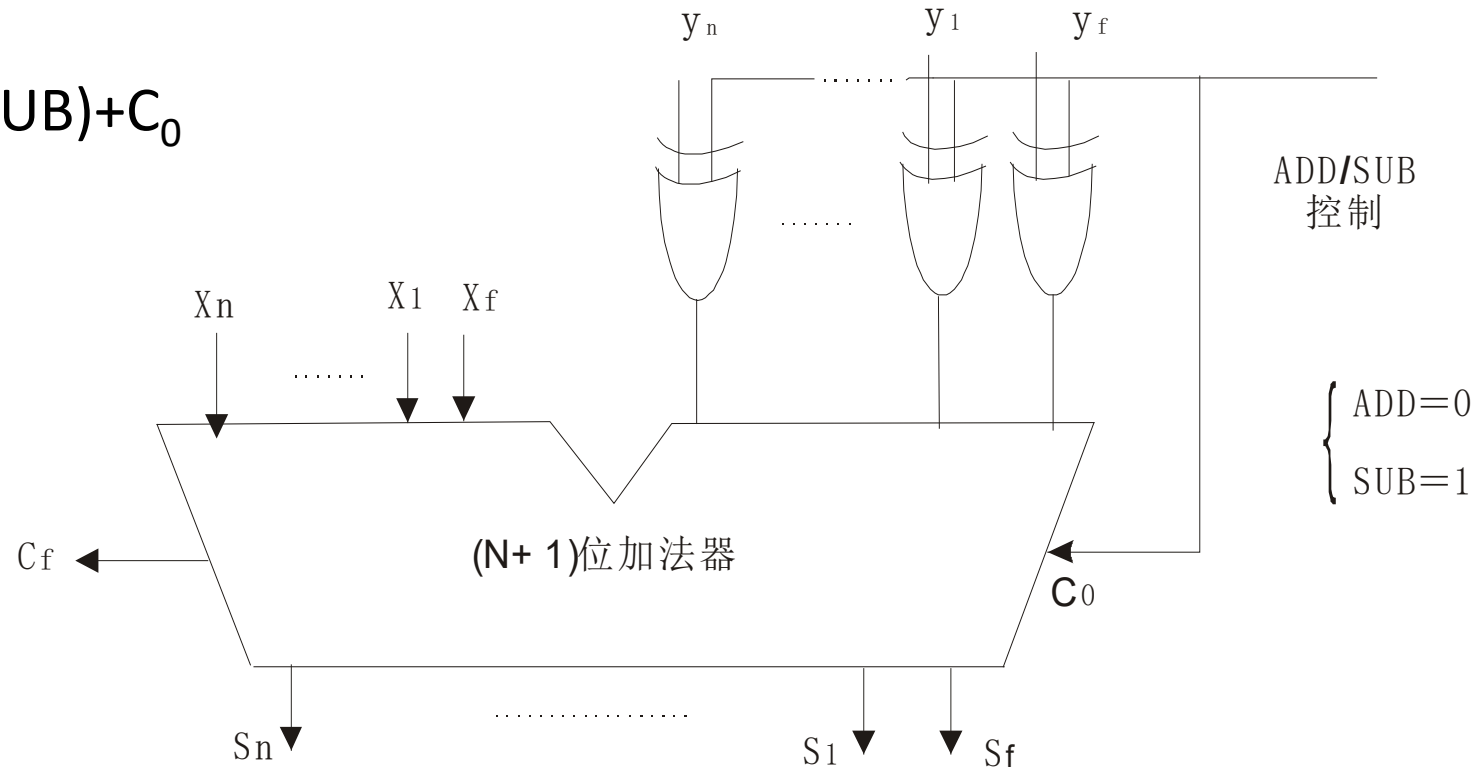


图3-1 n位二进制数补码加/减法器

加法器及其进位系统（补充）

1．全加器

基本的加法单元称为全加器，它要求三个输入量：操作数A_i、B_i和低位传来的进位C_{i-1}，并产生两个输出量：本位的和S_i、向高位的进位C_i。

逻辑表达式为：

$$S_i=A_i \oplus B_i \oplus C_{i-1}$$

$$C_i=A_iB_i+（A_i \oplus B_i）C_{i-1}$$

加法器及其进位系统

表3.4 全加器真值表

Ai	Bi	Ci−1	Si	Ci
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

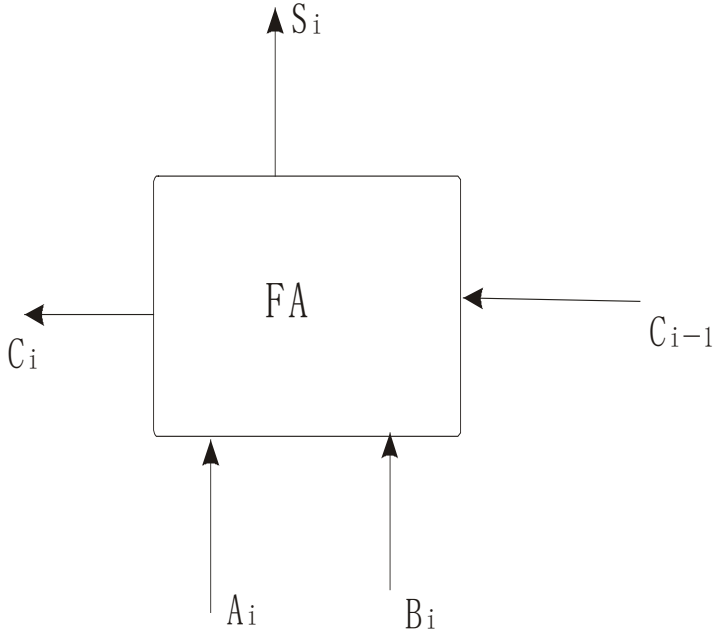


图3-13 全加器的逻辑框图

加法器及其进位系统

2．串行加法器与并行加法器

串行加法器中，只有一个全加器，数据逐位串行送入加法器进行运算。

并行加法器由多个全加器组成，其位数的多少取决于机器的字长，数据的各位同时运算。

1）具有右移功能的寄存器

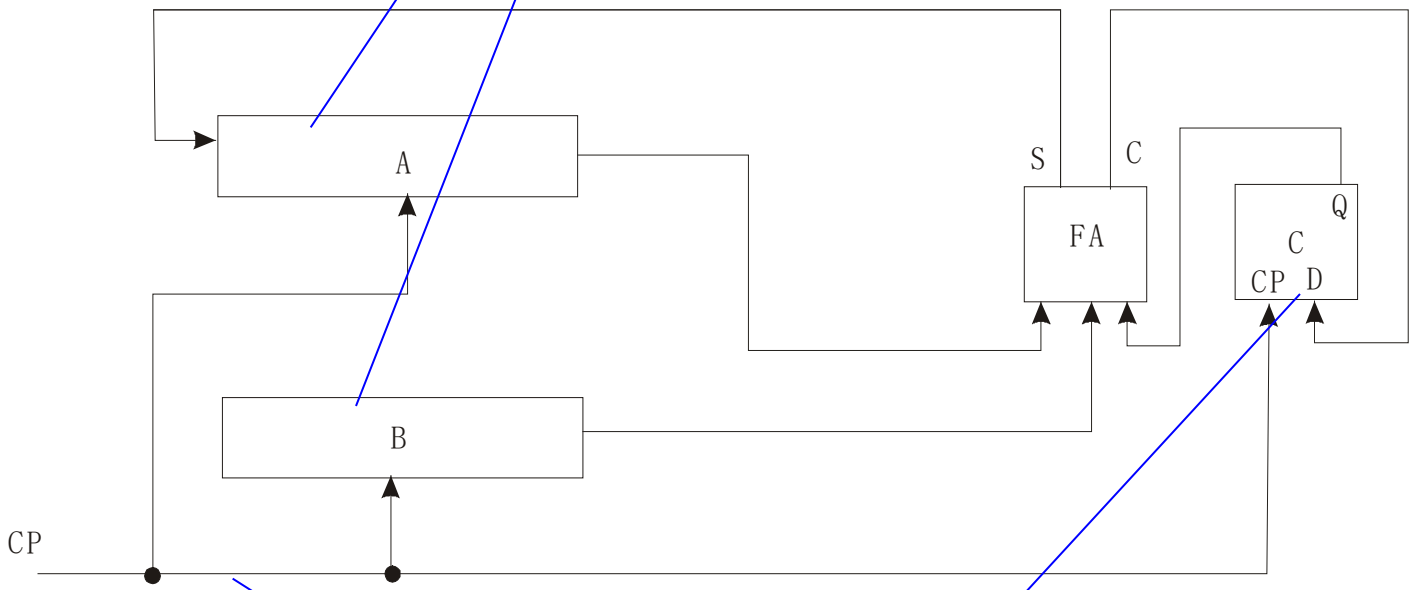


图3-14 串行加法器

2）每个脉冲算一位加法，和回送A，进位暂存与触发器C

加法器及其进位系统

二、进位产生和进位链

并行加法器中的每一个全加器都有一个从低位送来的进位和一个传送给较高位的进位。

各位之间传递进位信号的逻辑线路连接起来构成的进位网络称为进位链。

每一位的进位逻辑表达式为(表3-4推出)：

$$C_i=A_iB_i+(A_i\oplus B_i)C_{i-1}$$

取决于本位参加运算的两个数，而与低位进位无关，因此称A_iB_i为进位产生函数（本次进位产生），用G_i表示。其含义是若本位的两个输入均为1，必然要向高位产生进位。

加法器及其进位系统

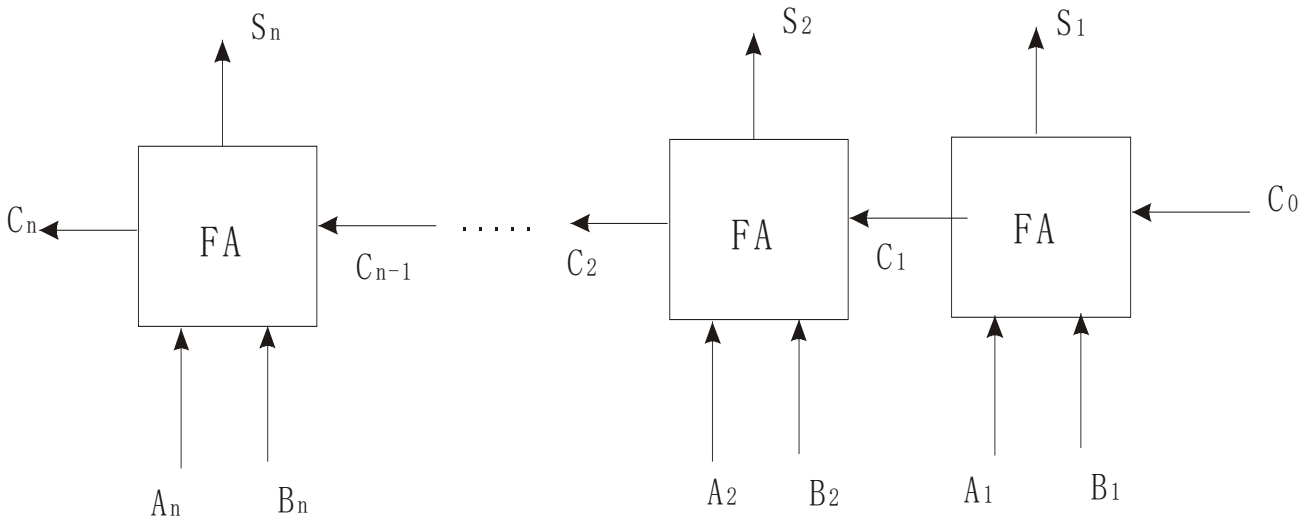
每一位的进位逻辑表达式为：

$$C_i=A_iB_i+(A_i \oplus B_i)C_{i-1}$$

不但与本位的两个数有关，还依赖于低位送来进位，因此称$A_i \oplus B_i$为进位传递函数（低位进位传递），用P_i表示。其含义是当两个输入中有一个为1，低位传来的进位C_{i-1}将超越本位向更高的位传送

所以：$C_i=G_i+P_iC_{i-1}$

1．串行进位方式



其中：

$C_1=G_1+P_1C_0$

$C_2=G_2+P_2C_1$

…

$C_n=G_n+P_nC_{n-1}$

图3-15 串行进位的并行加法器

串行进位链的总延迟时间与字长成正比，字长越长，总延迟时间就越长。

假定，将“与门”、“或门”的延迟时间定为t_y，从上述公式中可看出，每形成一级进位的延迟时间为$2t_y$。在字长为n位的情况下，若不考虑G_i、P_i的形成时间，从$C_0 \rightarrow C_n$的最长延迟时间为$2nt_y$。

2．并行进位方式

并行进位又称之先行进位、同时进位，其特点是各级进位信号同时形成。

$C_1=G_1+P_1C_0$

$C_2=G_2+P_2C_1=G_2+P_2G_1+P_2P_1C_0$

$C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1C_0$

$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0$

Gi、Pi的计算不依赖于低位的进位，若不考虑Gi、Pi的形成时间，从C0→Cn的最长延迟时间仅为2ty，而与字长无关。

但是随着加法器位数的增加，Ci的逻辑表达式会变得越来越长，输入变量不断增加，使电路结构变得很复杂，硬件费用昂贵，而且受到元器件扇入数的限制，所以完全采用并行进位是不现实。

3．分组并行进位方式

这种进位方式是把n位字长分为若干小组，在组内各位之间实行快速进位，在组间既可以采用串行进位方式，也可以采用并行快速进位方式。

（1）单级先行进位方式

实现进位逻辑函数的电路称之为四位先行进位电路（Carry Look Ahead，CLA），其延迟时间是$2t_y$。

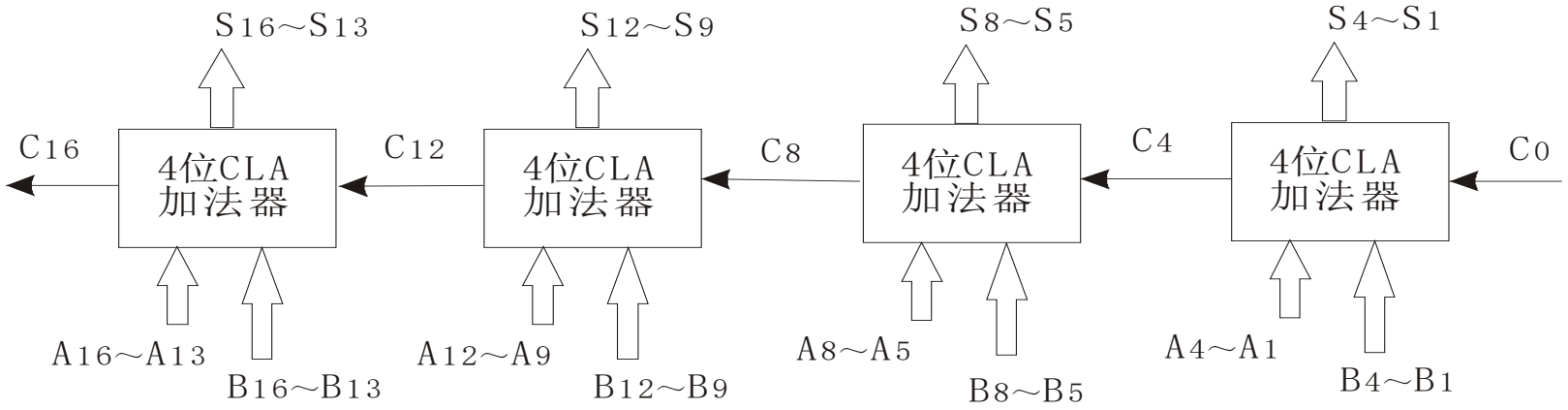


图3-16 16位单级先行进位加法器

若不考虑G_i、P_i的形成时间，从$C_0 \rightarrow C_n$的最长延迟时间为$2mt_y$，其中m为分组的组数。

2）多级先行进位方式

以字长为16位的加法器作为例子，分析两级先行进位加法器的设计方法。第一小组的最高位进位C_4可以变成两个与项相或：

$$C_4=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0= \quad G_1^*+P_1^*C_0$$

其中：$G_1^*=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1$ $\quad P_1^*=P_4P_3P_2P_1$

G_1^*称为组进位产生函数，P_1^*称为组进位传递函数，这两个辅助函数只与P_i、G_i有关。

依次类推，可以得到：

$$C_8= \quad G_2^*+P_2^*C_4 \quad = \quad G_2^*+P_2^*G_1^*+P_2^*P_1^*C_0$$

$$C_{12}= \quad G_3^*+P_3^*G_2^*+P_3^*P_2^*G_1^*+P_3^*P_2^*P_1^*C_0$$

$$C_{16}= \quad G_4^*+P_4^*G_3^*+P_4^*P_3^*G_2^*+P_4^*P_3^*P_2^*G_1^*+P_4^*P_3^*P_2^*P_1^*C_0$$

这些进位用专用电路并行计算，计算公式见上一页ppt

为了要产生组进位函数，须要对原来的CLA电路进行修改：

第1小组内产生G_1^*、P_1^*、C_3、C_2、C_1，不产生C_4；

第2小组内产生G_2^*、P_2^*、C_7、C_6、C_5，不产生C_8；

第3小组内产生G_3^*、P_3^*、C_{11}、C_{10}、C_9，不产生C_{12}；

第4小组内产生G_4^*、P_4^*、C_{15}、C_{14}、C_{13}，不产生C_{16}；

这种电路称为成组先行进位电路(Block Carry Look Ahead，BCLA），其延迟时间是$2t_y$。

用这种四位的BCLA电路以及进位产生/传递电路和求和电路可以构成四位的BCLA加法器。16位的两级先行进位加法器可由四个BCLA加法器和一个CLA电路组成，如图3-17所示。

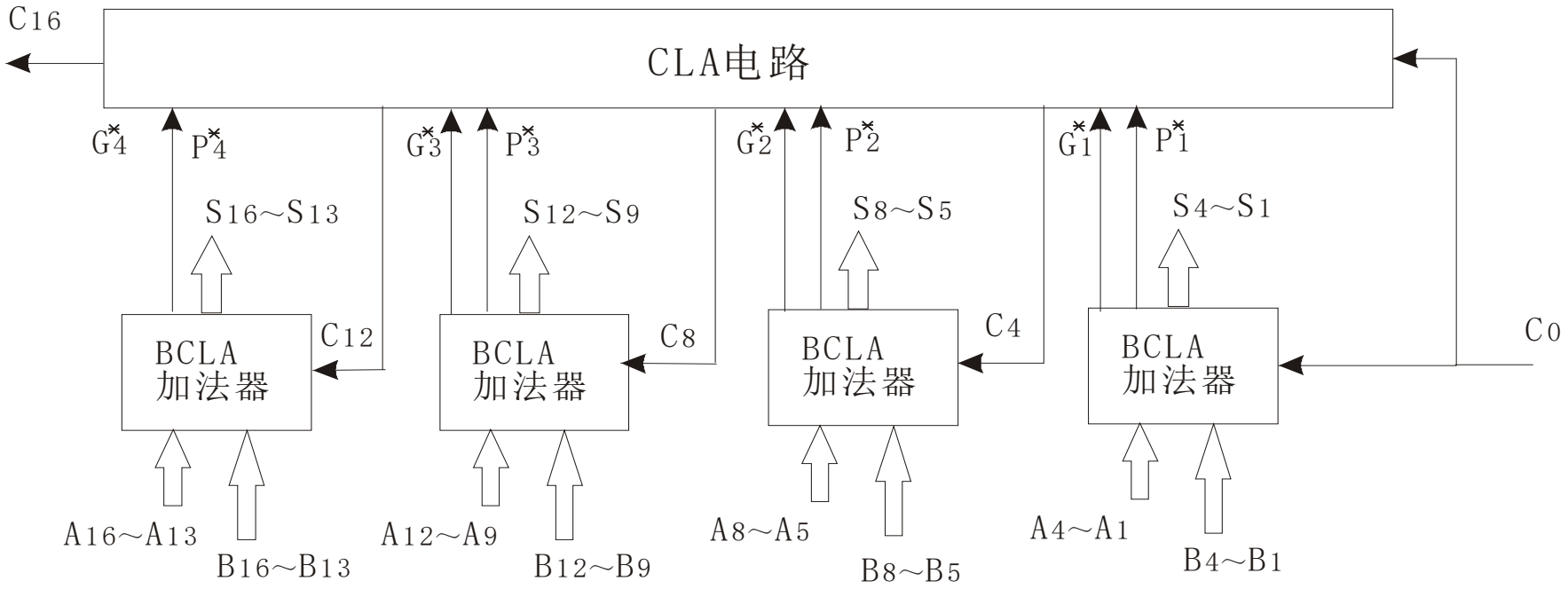


图3-17 16位两级先行进位加法器

若不考虑G_i、P_i的形成时间，C_0经过$2t_y$产生第1小组的C_1、C_2、C_3（由于C0已存在）及所有组进位产生函数G_i^*和组进位传递函数P_i^*（只与AB有关，在BCLA中进行）；

再经过$2t_y$，由CLA电路产生C_4、C_8、C_{12}、C_{16}；（它们的生成公式见前二张胶片，在CLA中。此时每组的组间进位就已产生好了）

再经过$2t_y$后，才能产生第2、3、4小组内的C_5~C_7、C_9~C_{11}、C_{13}~C_{15}。

以典型的四位ALU芯片（SN74181）为例介绍ALU的结构及应用。

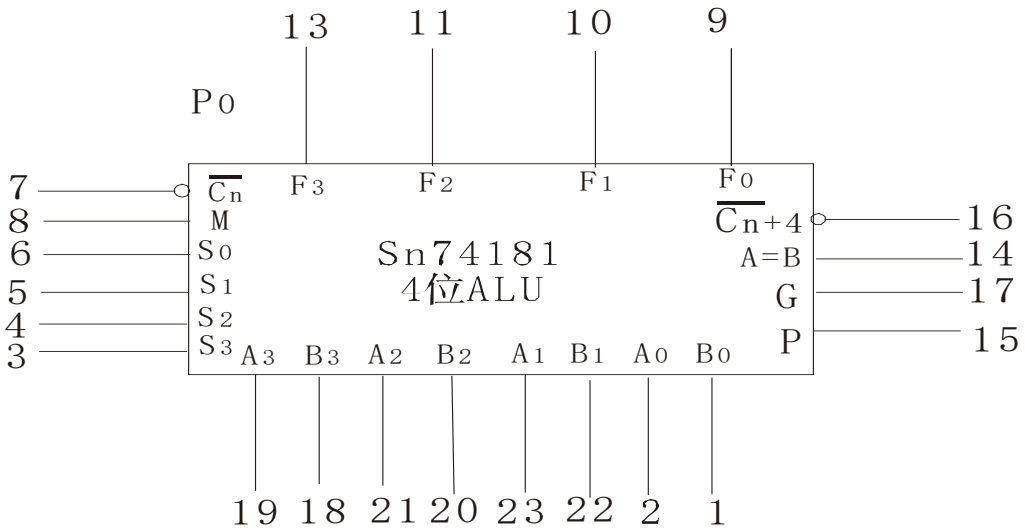


图3-18 74181 ALU芯片方框图

当M=0时，允许组间进位，进行算术操作。

当M=1时，封锁组间进位，进行逻辑操作。

表3.5 SN74181的算术/逻辑运算功能表

工作选择	负逻辑			正逻辑		
$S_3S_2S_1S_0$	逻辑运算（M=1）	算术运算（M=0）Cn=0（无进位）	算术运算（M=0）Cn=1（有进位）	逻辑运算（M=1）	算术运算（M=0）$\overline{Cn}$=1（无进位）	算术运算（M=0）$\overline{Cn}$=0（有进位）
0000	$F=\overline{A}$	F=A减1	F=A	F=A	F=A	F=A加1
0001	$F=\overline{AB}$	F=AB减1	F=AB	$F=\overline{A+B}$	F=A+B	F=（A+B）加1
0010	$F=\overline{A}+B$	$F=A\overline{B}$减1	$F=A\overline{B}$	$F=\overline{AB}$	$F=A+\overline{B}$	$F=(A+\overline{B})$加1
0011	F=1	F=减1	F=0	F=0	F=减1	F=0
0100	$F=\overline{A+B}$	$F=A加(A+\overline{B})$	$F=A加(A+\overline{B})$加1	$F=A\overline{B}$	$F=A加A\overline{B}$	$F=A加A\overline{B}$加1
0101	$F=\overline{B}$	$F=AB加(A+\overline{B})$	$F=AB加(A+\overline{B})$加1	$F=\overline{B}$	$F=(A+B)加A\overline{B}$	$F=(A+B)加A\overline{B}$加1
0110	$F=\overline{A\oplus B}$	F=A减B减1	F=A减B	$F=A\oplus B$	F=A减B减1	F=A减B
0111	$F=A+\overline{B}$	$F=A+\overline{B}$	$F=(A+\overline{B})$加1	$F=A\overline{B}$	$F=A\overline{B}$减1	$F=A\overline{B}$
1000	$F=\overline{A}B$	F=A加（A+B）	F=A加（A+B）加1	$F=\overline{A}+B$	F=A加AB	F=A加AB加1
1001	$F=A\oplus B$	F=A加B	F=A加B加1	$F=\overline{A\oplus B}$	F=A加B	F=A加B加1
1010	F=B	$F=\overline{AB}$加（A+B）	$F=\overline{AB}$加（A+B）加1	F=B	$F=(A+\overline{B})$加AB	$F=(A+\overline{B})$加AB加1
1011	F=A+B	F=A+B	F=（A+B）加1	F=AB	F=AB减1	F=AB
1100	F=0	F=A加A^*	F=A加A^*加1	F=1	F=A加A^*	F=A加A^*加1
1101	$F=\overline{A}B$	F=AB加A	F=AB加A加1	$F=A+\overline{B}$	F=（A+B）加A	F=（A+B）加A加1
1110	F=AB	$F=A\overline{B}$加A	F=$A\overline{B}$加A加1	F=A+B	$F=(A+\overline{B})$加A	$F=(A+\overline{B})$加A加1
1111	F=A	F=A	F=A加1	F=A	F=A减1	F=A

*：A加A=2A，算术左移一位

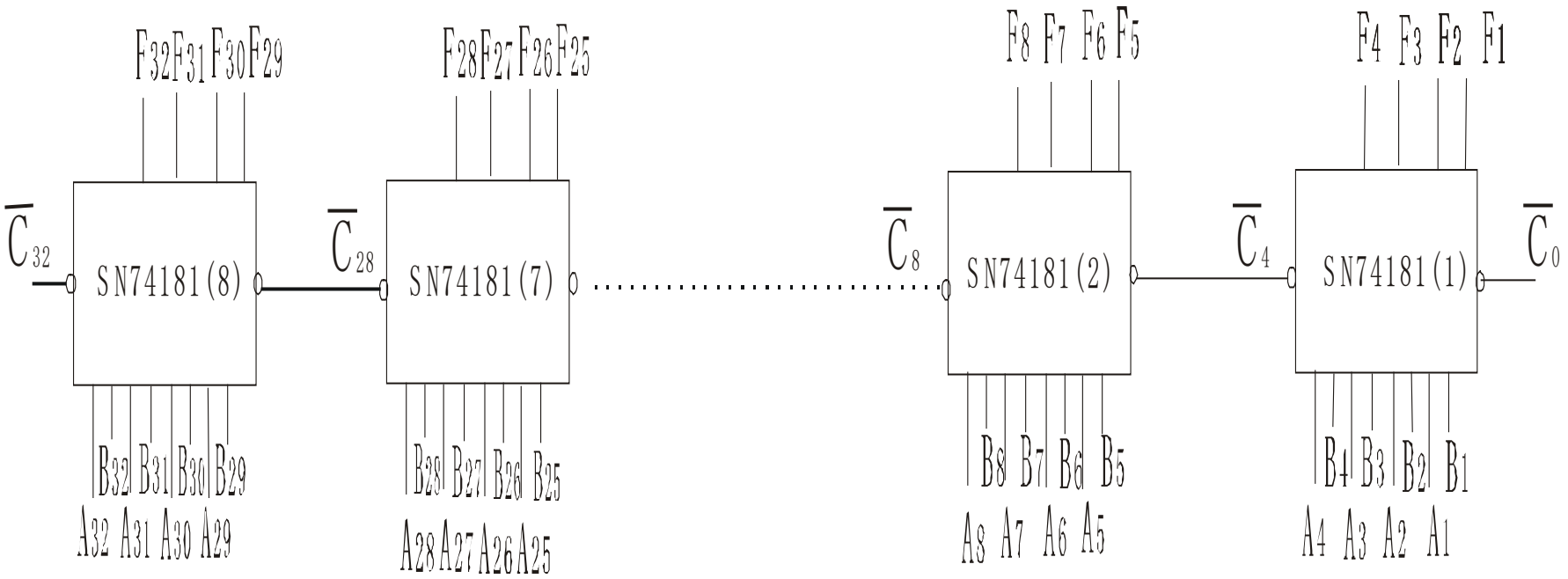


图3-19 32位行波进位方式的ALU

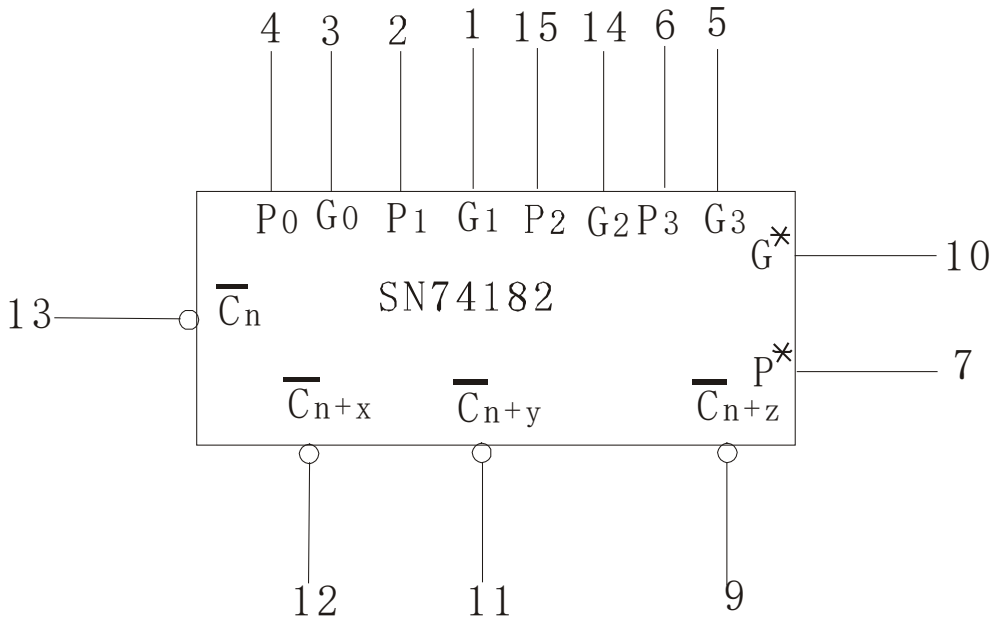


图3-20 74182芯片方框图

先行进位部件

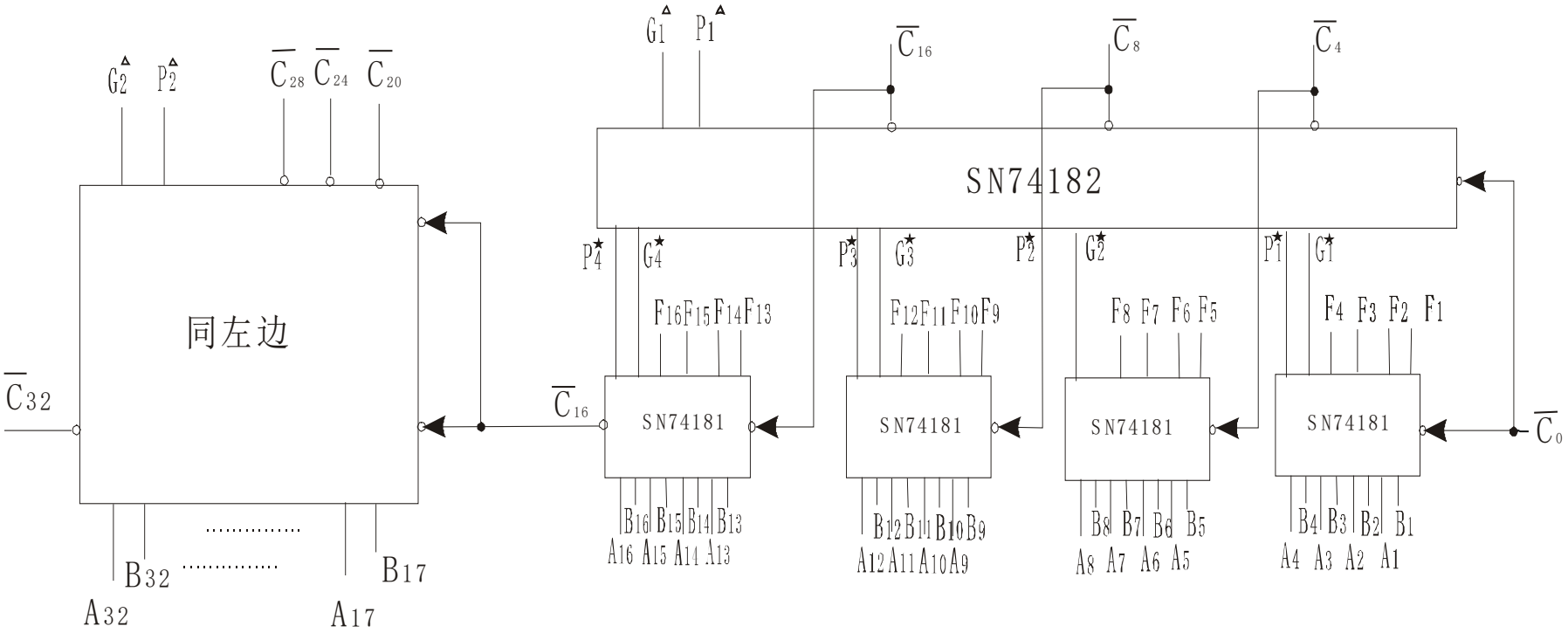


图3-21 32位两级行波ALU

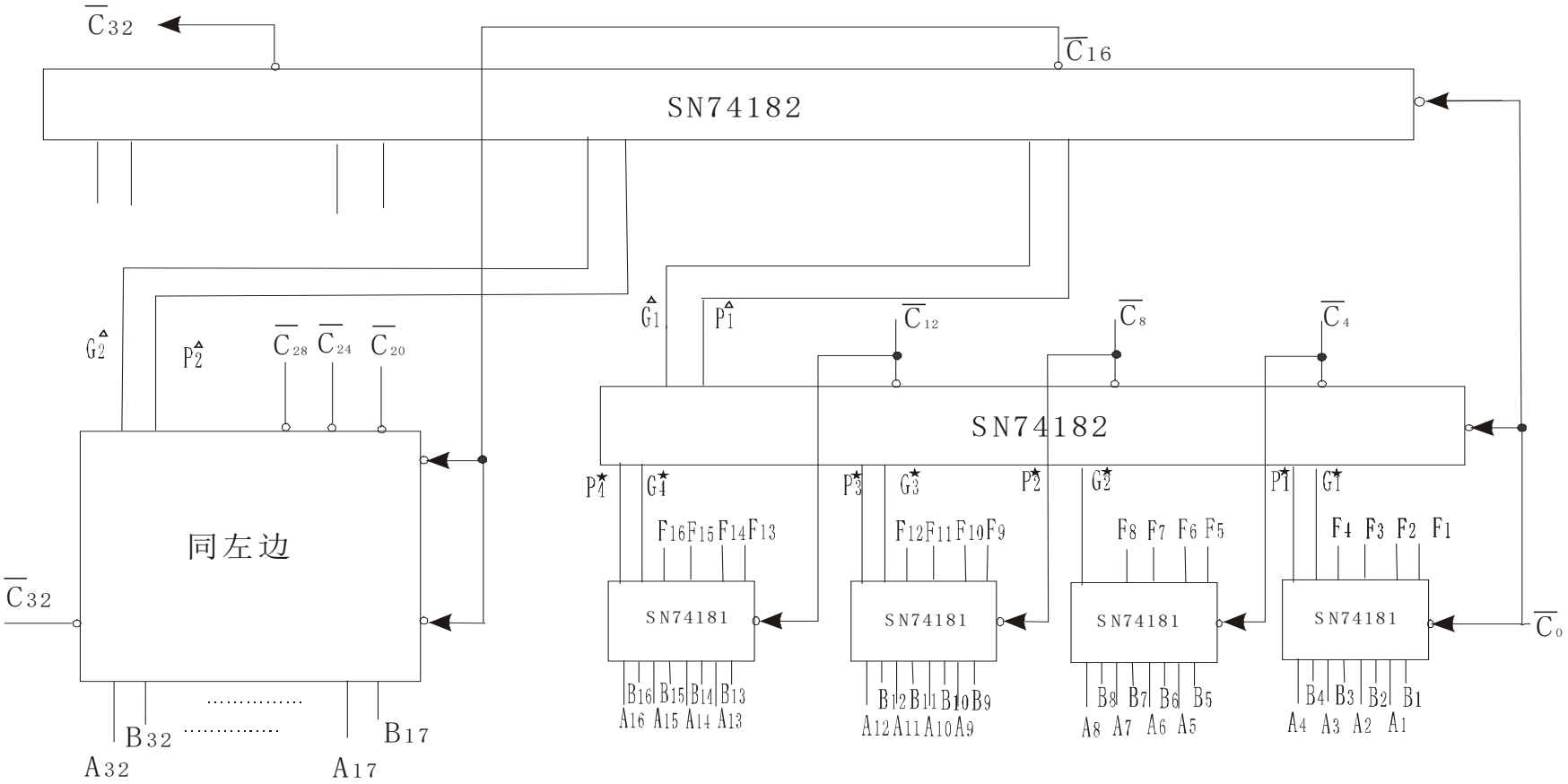


图3-22 32位三级并行进位ALU

3.2.1 多媒体算数运算

- 桌面处理器一般都有显卡，颜色2^8*3。像素位置256*256（2^8）。音频8位或16位。所以，处理器需对8位、16位数据进行支持。（新教材无，答案3）
- 饱和：计算结果溢出，则结果设置为最大正数（或最小负数）：音量到最大再同向旋转音量停留在最大值。

溢出判断代码

- MIPS无测试溢出分支指令，可用一段程序判断溢出。
- 有符号加法：

```
addu $t0,$t1,$t2
xor $t3,$t1,$t2
slt $t3,$t3,$0
bne $t3,$0,No_overflow
xor $t3,$t0,$t1

slt $t3,$t3,$0
bne $t3,$0,Overflow
```

溢出判断代码

- 异号相加不会溢出。设$t1为+（最高位0），$t2为-（最高位1）

```
addu $t0,$t1,$t2  #t0=t1+t2，做无符号加法（最高位也相加）
xor $t3,$t1,$t2 #最高位0与1异或，$t3最高位为1，负数（其余位无关紧要）
slt $t3,$t3,$0  # t3<0为true，t3=1
bne $t3,$0,No_overflow  #t3=1≠0，转到No_overflow处理

xor $t3,$t0,$t1
slt $t3,$t3,$0
bne $t3,$0,Overflow
```

溢出判断代码

- 同号相加会溢出。设$t1为+（最高位0），$t2为+（最高位0）分析+++ = -的情况，用t0记和，t3判断符号位。

```
addu $t0,$t1,$t2  #t0=t1+t2，做无符号加法（最高位也相加）
xor $t3,$t1,$t2 #最高位0与0异或，$t3最高位为0，正数（其余位无关紧要）
slt $t3,$t3,$0  # t3<0为false，t3=0
bne $t3,$0,No_overflow  #0≠0为false，不转No_overflow，继续下面

xor $t3,$t0,$t1  #和t0与操作数异或，若正+正=负，溢出，
                 #则t0最高位为1，异或后$t3最高位为1，负数
slt $t3,$t3,$0      #$t3<0为true，t3=1
bne $t3,$0,Overflow  # t3≠ 0为true，转到overflow处理
```

溢出判断代码

- 同理分析+ + + = +不溢出。
- 设$t1为+（最高位0），$t2为+（最高位0）分析+++ = +的情况，用t0记和，t3判断符号位。

```
addu $t0,$t1,$t2  #t0=t1+t2，做无符号加法（最高位也相加）
xor $t3,$t1,$t2 #最高位0与0异或，$t3最高位为0，正数（其余位无关紧要）
slt $t3,$t3,$0  # t3<0为false，t3=0
bne $t3,$0,No_overflow  #0≠0为false，不转No_overflow，继续下面

xor $t3,$t0,$t1  #若不溢出，则正+正=正，
                   #则t0最高位为0，异或后$t3最高位为0，正数
slt $t3,$t3,$0      #$t3<0为false，t3=0
bne $t3,$0,Overflow  # t3≠ 0为false，转到no-overflow处理
```

- 同理分析- + - 等其他情况。

溢出判断代码

- 无符号数加法：无符号数相加在和大于111…1=2^{32}-1时溢出。（因为只有32位）

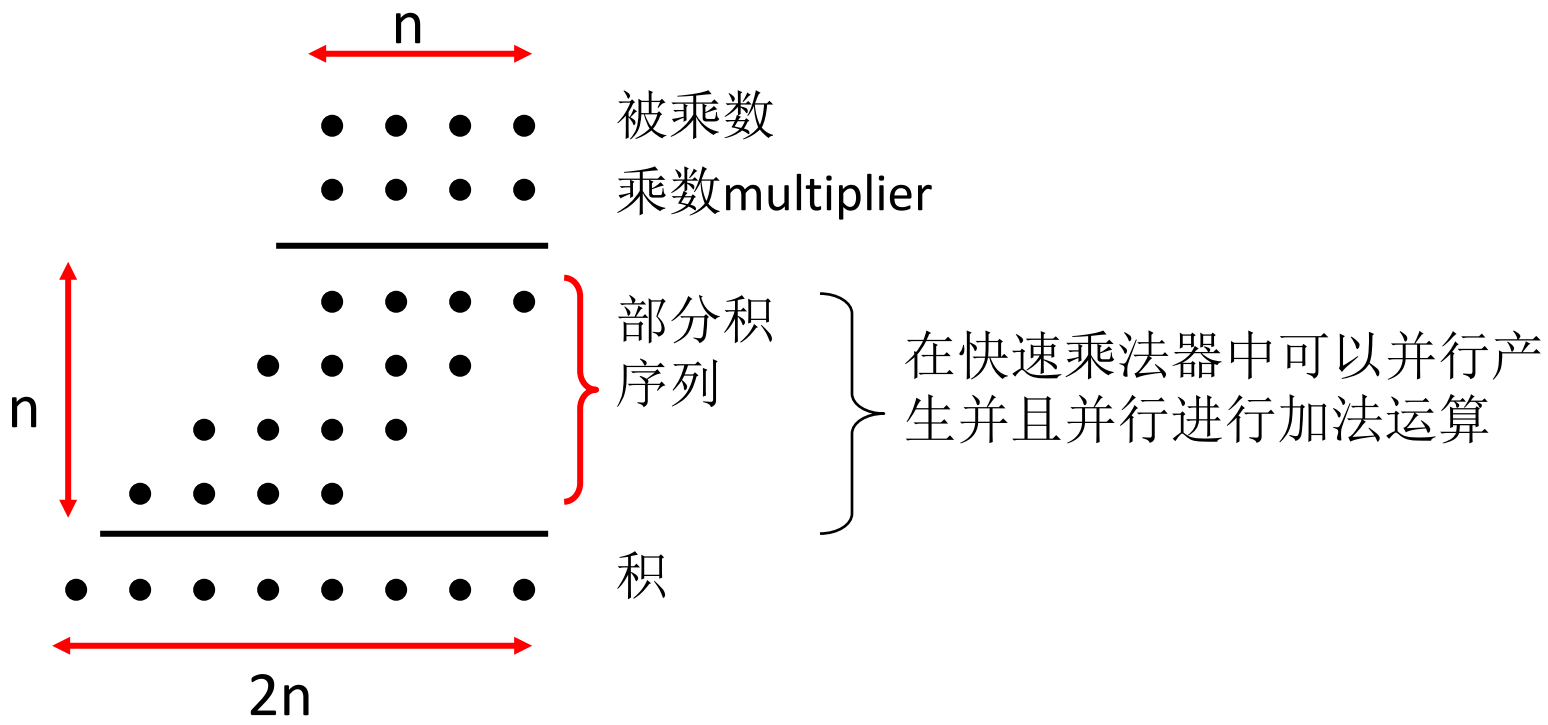
```
addu $t0,$t1,$t2 # t0=t1+t2
nor $t3,$t1,$0  # $t3 = Not$t1 = 2^32-$t1-1

sltu $t3,$t3,$t2 # t3<t2，则t3=1
                 # →2^32-$t1-1 < t2 (t3=1此时)
                 #→2^32-1 < t1 + t2 溢出 (t3=1此时)

bne $t3,$0,Overflow #t3=1≠0为true，转到overflow处理
```

作业

- 3.6　3.7

3.3 乘法

- 二进制乘法手算过程：（无符号）

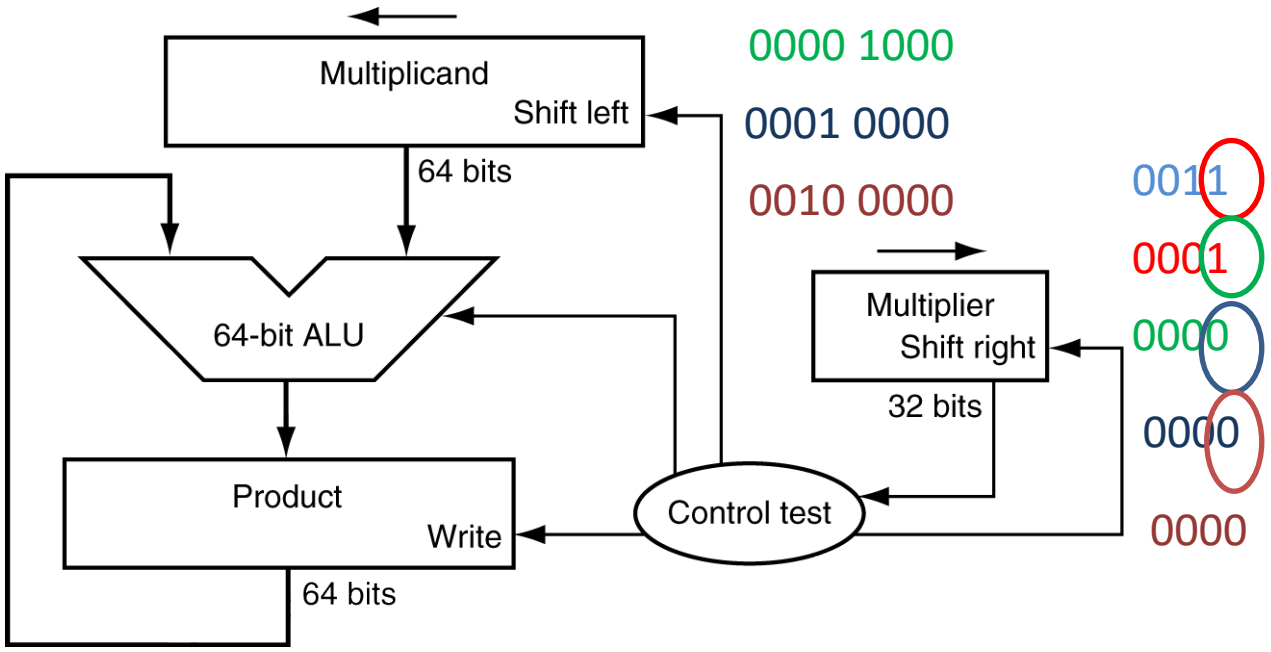


从上述算法分析：

1）积的位数：忽略符号位，若被乘数的位数为n，乘数的位数为m，则积的位数为n+m

2）当乘数位为1，只需要将被乘数复制到合适位置。

3）当乘数位为0，将0复制到合适位置。



0000 0010

0000 0100 左移

0000 1000

0001 0000

0010 0000

0011

0001 右移

0000

0000

0000

0000 0000

+ 0000 0010 被乘数

0000 0010

+ 0000 0100

0000 0110

+ 0000 0000

0000 0110

+ 0000 0000

0000 0110

以两个4位数相乘为例：

1）共做4次迭代

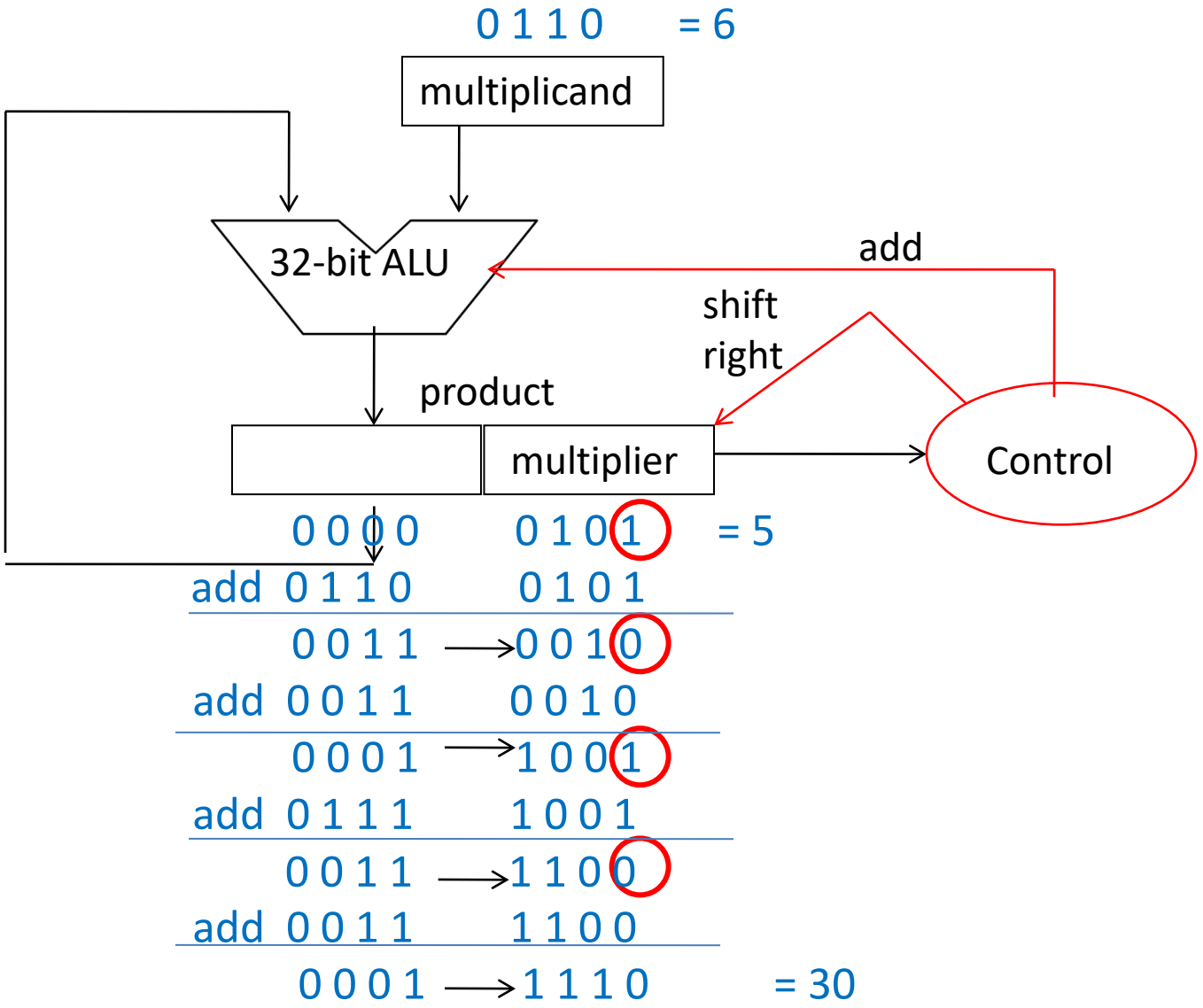
2）每次迭代，测试加法

左移（观察前页图可知每个部分积相当于被乘数左移后*乘数判断位）

右移（已经判断过的位移出）共三步。

所以，若每部一个时钟周期，共3*字长个时钟周期

- 以上一步部分积右移代替被乘数（可能加到部分积上形成新部分积）左移。
- 减少一步移位（测试加法、右移两步完成）（65位 P141， P123）



有符号乘法

- 被乘数、乘数取绝对值参与运算。
- 积的符号位为被乘数乘数符号位异或。

该算法实际上是：原码一位乘法算法。

很多教材介绍补码一位乘法（符号位参与运算）。

原码一位乘法小结*

原码一位乘法的运算规则

被乘数：$X=x_f.x_1x_2\cdots x_n$

乘数：　$Y=y_f.y_1y_2\cdots y_n$

积：　　$C=c_f.c_1c_2\cdots c_n$

式中，x_f，y_f,c_f为符号位。则

$$c_f = x_f \oplus y_f$$

$$|C|=|X|\bullet|Y|$$

P141图3-6　65位，多出的1位为了保留进位

求|C|的运算规则如下：

（1） 被乘数和乘数均取绝对值参加运算，符号位单独考虑。

（2） 被乘数取双符号，部分积的长度与被乘数相同，初值P_0为0。

（3） 从乘数的最低位y_n开始判断：

①若y_{n-i+1}=1，则部分积Pi加被乘数|x|，然后右移一位；

②若y_{n-i+1}=0，则部分积Pi加上0，然后右移一位。

移位时连同符号位一起移位

（4）重复步骤（3），判断n次。最终结果C=Cf.Pn

[例] x = 0.1101，y = -0.1011，求$[x \cdot y]_{原}$=?

被乘数、乘数取绝对值参与运算，所以，|y| = 0.1011

被乘数用双符号位，|x| = 00.1101

迭代	步骤	部分积	乘数
0	初始值	00.0000	0.1011
1	y_4=1，加｜x｜	+00.1101	
		00.1101	0.1011
	右移	00.0110	10.101
2	y_3=1，加｜x｜	+00.1101	
		01.0011	10.101
	右移	00.1001	110.10
3	y_2=0, 加0	+00.0000	
		00.1001	110.10
	右移	00.0100	1110.1
4	y_1=1，加｜x｜	+00.1101	
		01.0001	1110.1
	右移	00.1000	11110

由于　　$C_f = x_f \oplus y_f = 0 \oplus 1 = 1$

所以　　$[x \cdot y]_{原} = 1.10001111$

从例中可见：

(1) 字长n+1位数参加运算要做n次相加和移位操作，计算机中可利用计数器计数加以控制；

(2) 用相加和移位操作实现了乘法运算。

补码一位乘法*

1．补码一位乘法的原理

被乘数 $[X]_{补} = x_0.x_1x_2\cdots x_n$,

乘数 $[Y]_{补} = y_0.y_1y_2\cdots y_n$

积 $[C]_{补} = c_0.c_1c_2\cdots c_n$

(1) X正负任意数，Y为正数

$[X]_{补} = 2+X$

$= 2^{n} \cdot 2+X \pmod 2$

$= 2^{n+1}+X \pmod 2$

$[Y]_{补} = Y$

则：$[x]_{补} \cdot [y]_{补} = 2^{n+1} \cdot Y + X \cdot Y$

$= 2^{n+1} \cdot (y_0 \cdot y_1 y_2 \cdots y_n) + X \cdot Y$

$= 2 \cdot (y_0 y_1 y_2 \cdots y_n) + X \cdot Y$

$= 2 \cdot (y_1 y_2 \cdots y_n) + X \cdot Y$

$= 2 + X \cdot Y$

$= [x \cdot y]_{补}$

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

$Y>0, y_0=0$

$y_1 y_2 \cdots y_n$为整数。根据模的运算性质：
$2 \times (y_1 y_2 \cdots y_n) = 2$

(2) X正负任意数，Y为负数

$[X]_{补}=x_0.x_1x_2\cdots x_n$

$[Y]_{补}=1.y_1y_2\cdots y_n=2+y$

$\therefore Y=[Y]_{补}-2$

$=1.y_1y_2\cdots y_n-2$

$=0.y_1y_2\cdots y_n-1$

$\therefore\ X\cdot Y=X\cdot(0.y_1y_2\cdots y_n)-X$

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

因为 $(0.y_1y_2\cdots y_n)>0$

$\therefore[X\cdot Y]_{补}=[X]_{补}\cdot(0.y_1y_2\cdots y_n)+[-X]_{补}$

(3) X和Y正负任意

将(1)和(2)两种情况综合起来，得：

$$[X\cdot Y]_{补}=[X]_{补}\cdot(0.y_1y_2\cdots y_n)+[-X]_{补}\times y_0$$

$$=[X]_{补}(-y_0+0.y_1y_2\cdots y_n)$$

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

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

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

$$=[X]_{补}\cdot\sum_{i=1}^{n}(y_{i+1}-y_i)2^{-i}$$

乘数的最低位为y_n，在其后面添加1位y_{n+1}，其值为0，按机器执行顺序求出每一步的部分积。

写成递推公式如下：

右移

$$[P_0]_{补} = 0$$

$$[P_1]_{补} = 2^{-1}\{[P_0]_{补} + (y_{n+1} - y_n)[x]_{补}\} \qquad (y_{n+1}=0)$$

$$[P_2]_{补} = 2^{-1}\{[P_1]_{补} + (y_n - y_{n-1})[x]_{补}\}$$

…

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

判断+-$[X]_{补}$

…

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

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

请注意：y_0是乘数y的符号位，y_{n+1}是附加位。

2．补码一位乘法的运算规则（Booth算法）

(1) 符号位参加运算，运算的数均以补码表示；

(2) 被乘数取双符号位参加运算，部分积初值为0；

(3) 乘数取单符号位以决定最后一步是否需要校正，即是否加$[-x]_{补}$；

(4) 乘数末位增设附加位y_{n+1}且初值为0；

(5) 由于求得一次部分积要右移一位，y_n与y_{n+1}位就构成了各步运算的判断位。从而可以得到算法如表3.1所示。

表3.1 补码一位乘法的算法

yn（高位）	Yn+1（低位）	yn+1-yn	操作
0	0	0	部分积右移一位
0	1	1	部分积加**[x]**补右移一位
1	0	−1	部分积加**[-x]**补右移一位
1	1	0	部分积右移一位

(6) 按照上述算法进行n+1步操作，但第n+1步不再移位，仅根据y_0与y_1的比较结果作相应的运算即可。

(7) 右移时，部分积>0，最高位补0，<0，最高位补1。

例：$[x]_{补}$ = 1.0101，$[y]_{补}$ = 1.0011，求$[x \cdot y]_{补}$=?

解：$[-x]_{补}$ = 0.1011 = 00.1011

迭代	步骤	部分积	乘数
0	初值	00.0000	1.0011 0
1	$y_n y_{n+1}$=10，加$[-x]_{补}$	+00.1011	
		00.1011	1.0011 0
	右移	00.0101	11.001 1
2	$y_n y_{n+1}$=11，加0	+00.0000	
		00.0101	11.001 1
	右移	00.0010	111.00 1
3	$y_n y_{n+1}$=01，加$[x]_{补}$	+11.0101	
		11.0111	111.00 1
	右移（补码算数移位，补入符号位）	11.1011	1111.0 0
4	$y_n y_{n+1}$=00,加0	+00.0000	
		11.1011	
	右移	11.1101	1111 1. 0
5	$y_n y_{n+1}$=10，加$[-x]_{补}$	+00.1011	
	最后一步不移位	00.1000	1111

故得　$[x \cdot y]_{补}=0.10001111$

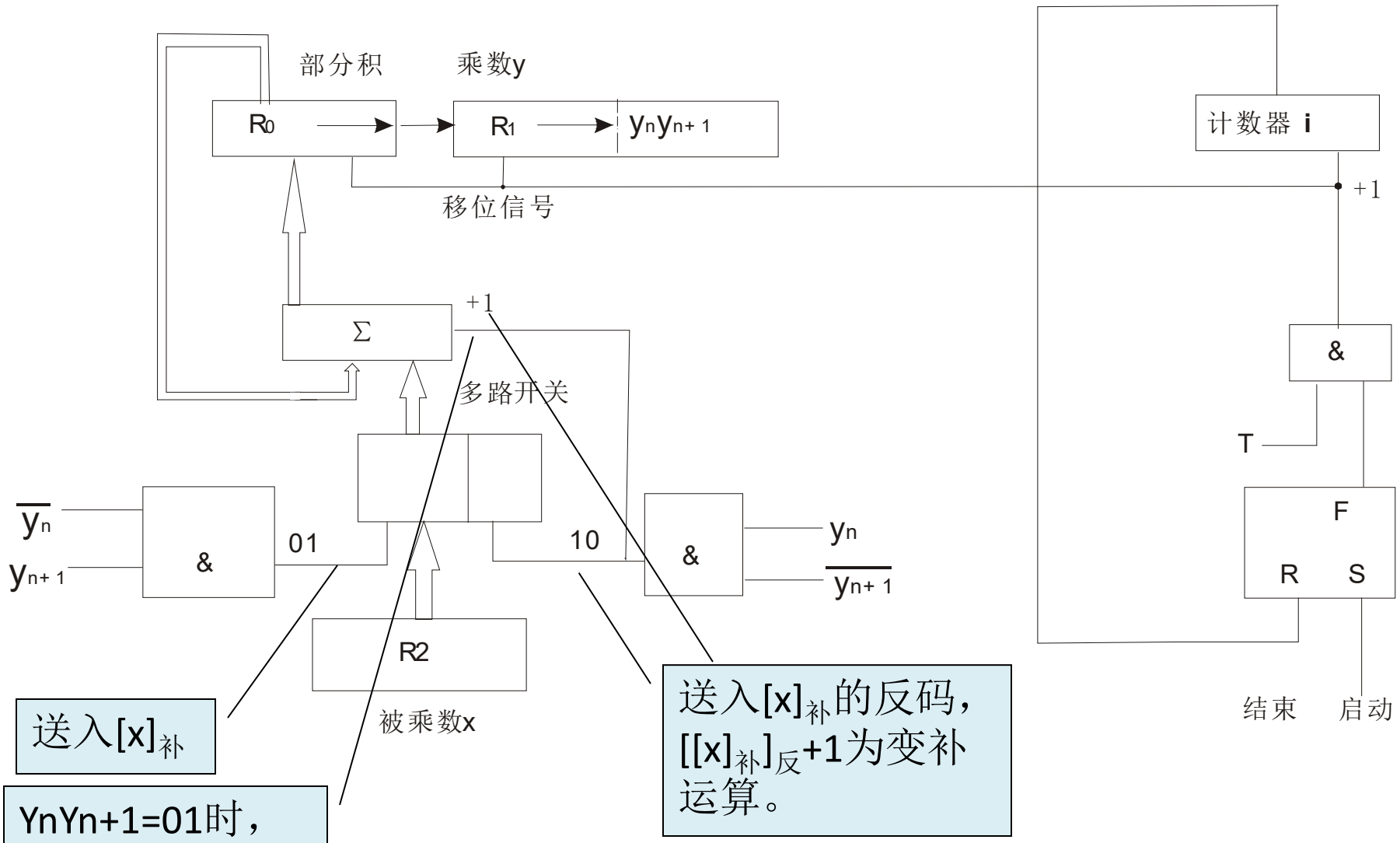
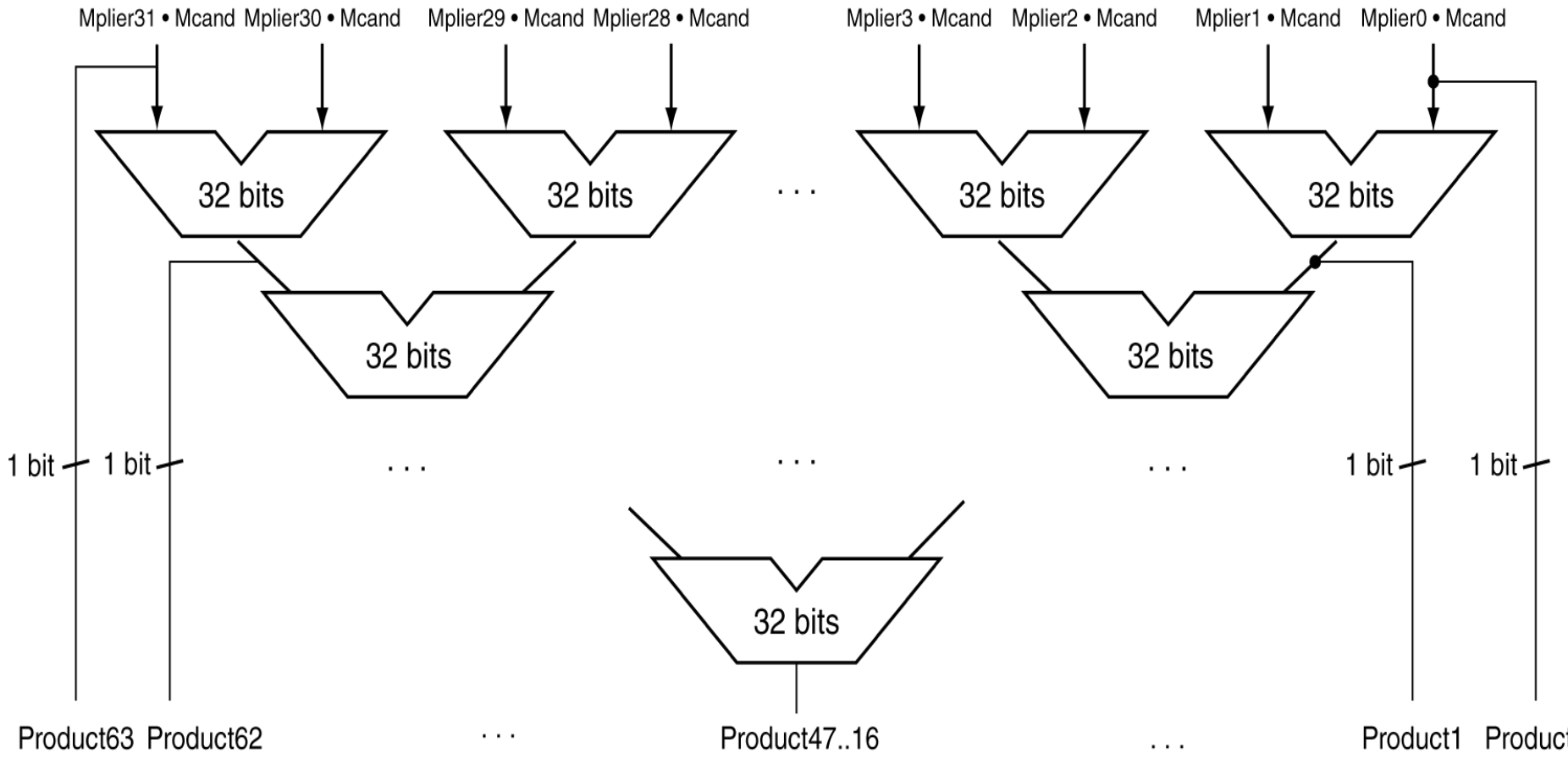
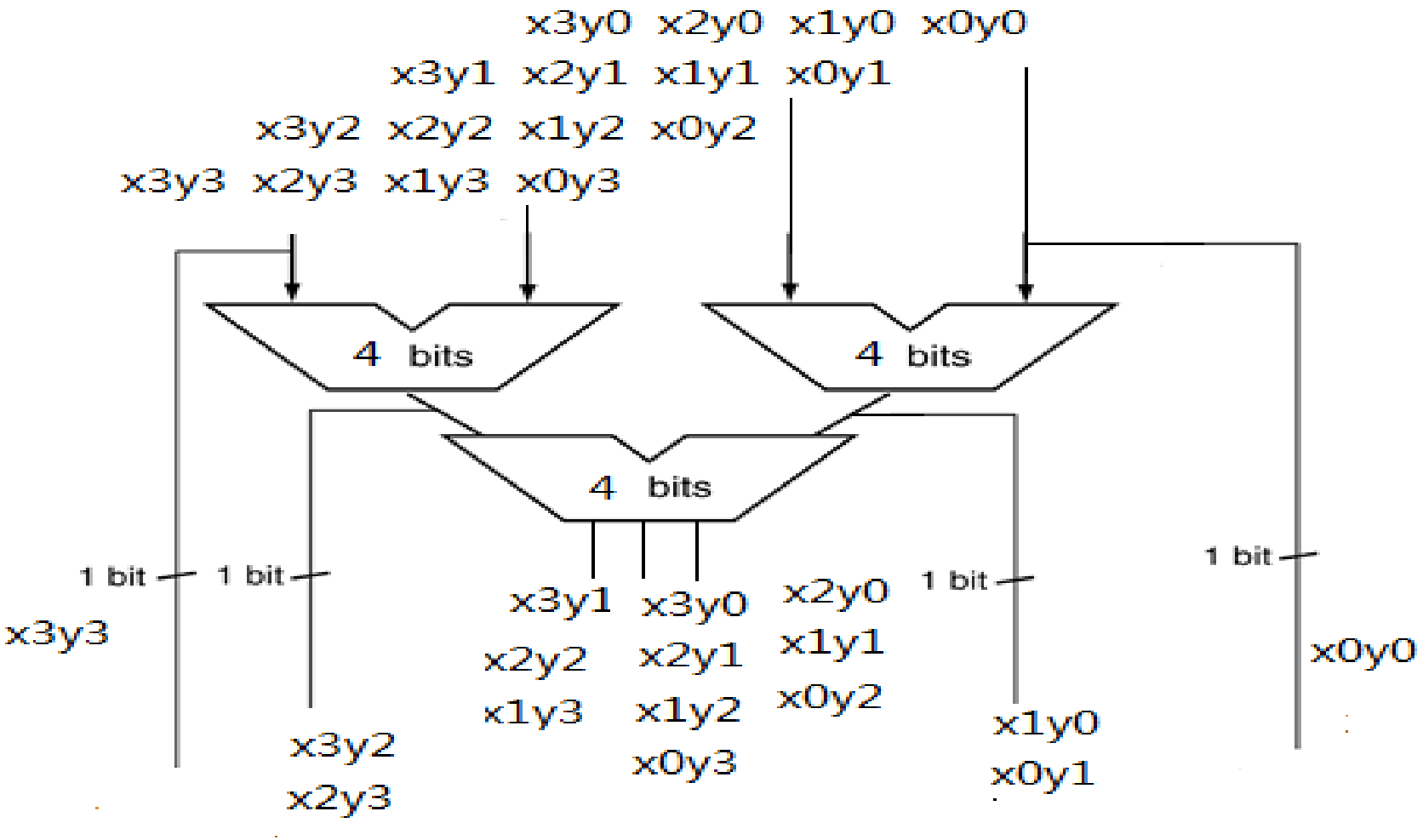


图3-5 补码一位乘法的逻辑结构图

*

快速乘法器


Mplier31 • Mcand
Mplier30 • Mcand
Mplier29 • Mcand
Mplier28 • Mcand
Mplier3 • Mcand
Mplier2 • Mcand
Mplier1 • Mcand
Mplier0 • Mcand
32 bits
32 bits
32 bits
32 bits
32 bits
32 bits
32 bits
1 bit
1 bit
1 bit
1 bit
Product63
Product62
Product47..16
Product1
Product0



将3个加法器组成并行树。等待2次（$\log_2 4=2$）加法时间。

必须有先行进位计算？

阵列乘法器

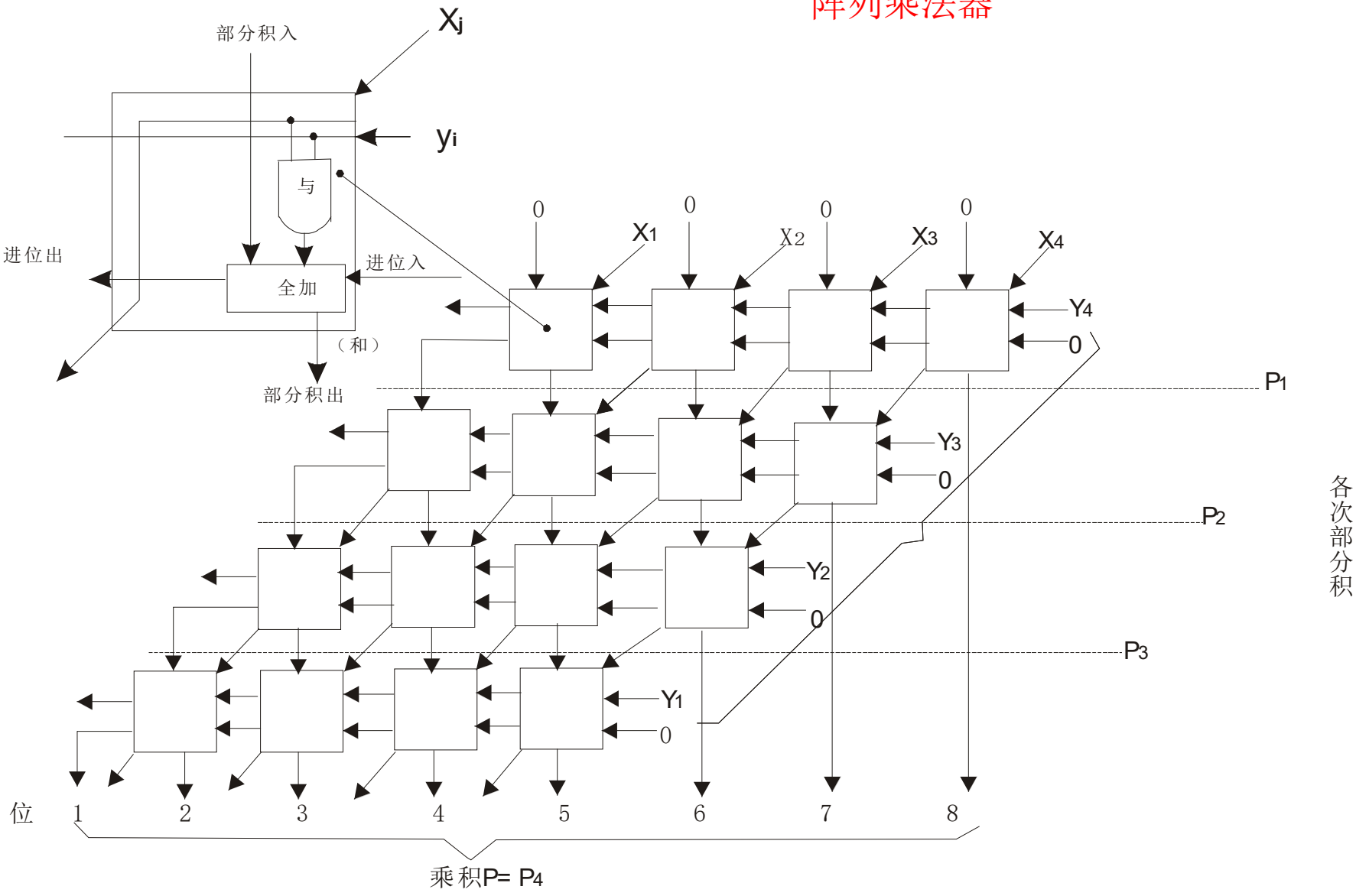


图3-6 阵列乘法器

MIPS中的乘法

- 两个32-bit 寄存器用于存储积
  - HI: 高 32 bits
  - LO: 低 32-bits
- 指令
  - `mult rs, rt  /  multu rs, rt` (P518)
    - 64-bit 积存于 HI/LO
  - `mfhi rd  /  mflo rd` (P527)
    - Move from HI/LO to rd
    - 可以用于检测HI 的值以便判断溢出（积超过32位）
  - `mul rd, rs, rt` (P518)
    - rs和rt积的低32位直接存入rd

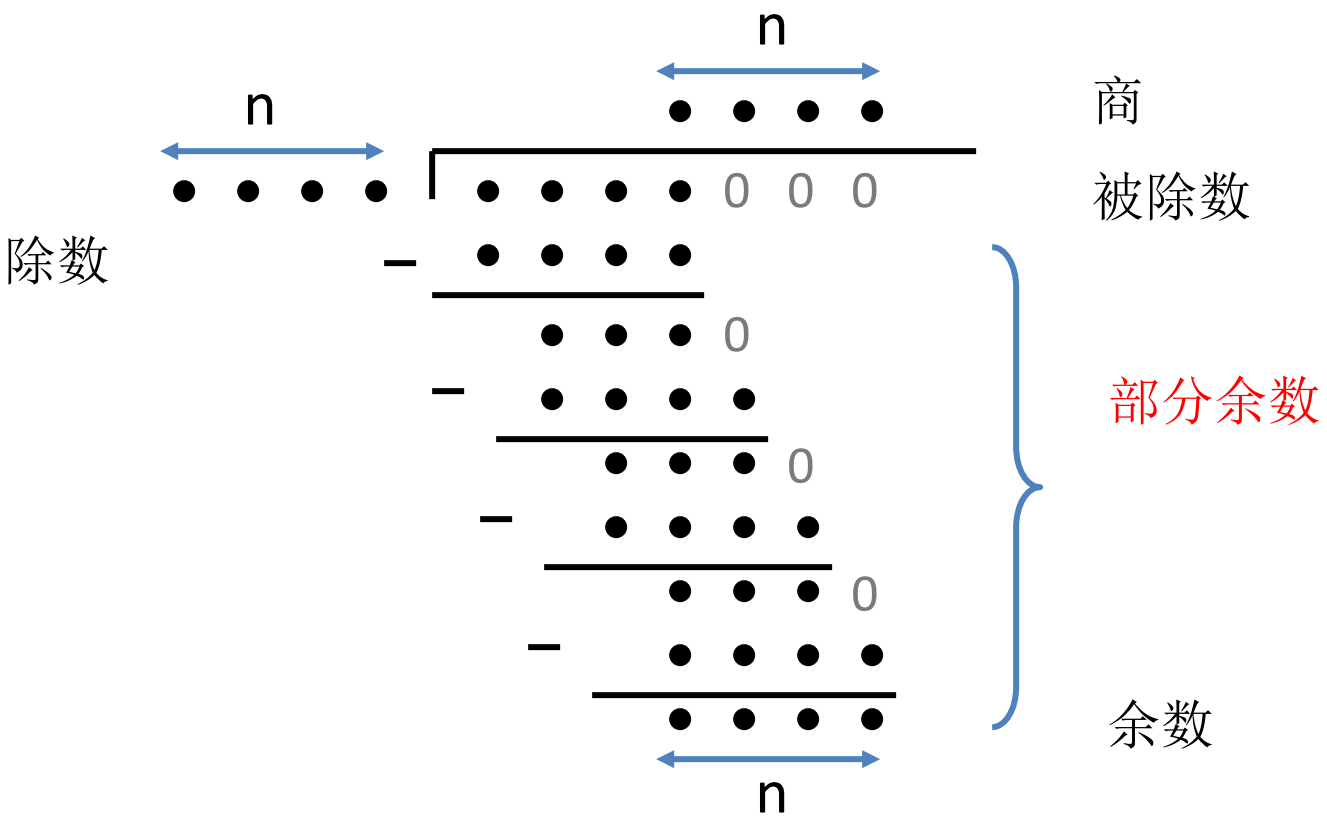
作业

- 3.12 3.13

3.4 除法

除法每次测试最大能减掉多少，并以此产生商。
二进制只有0或1两种选择。

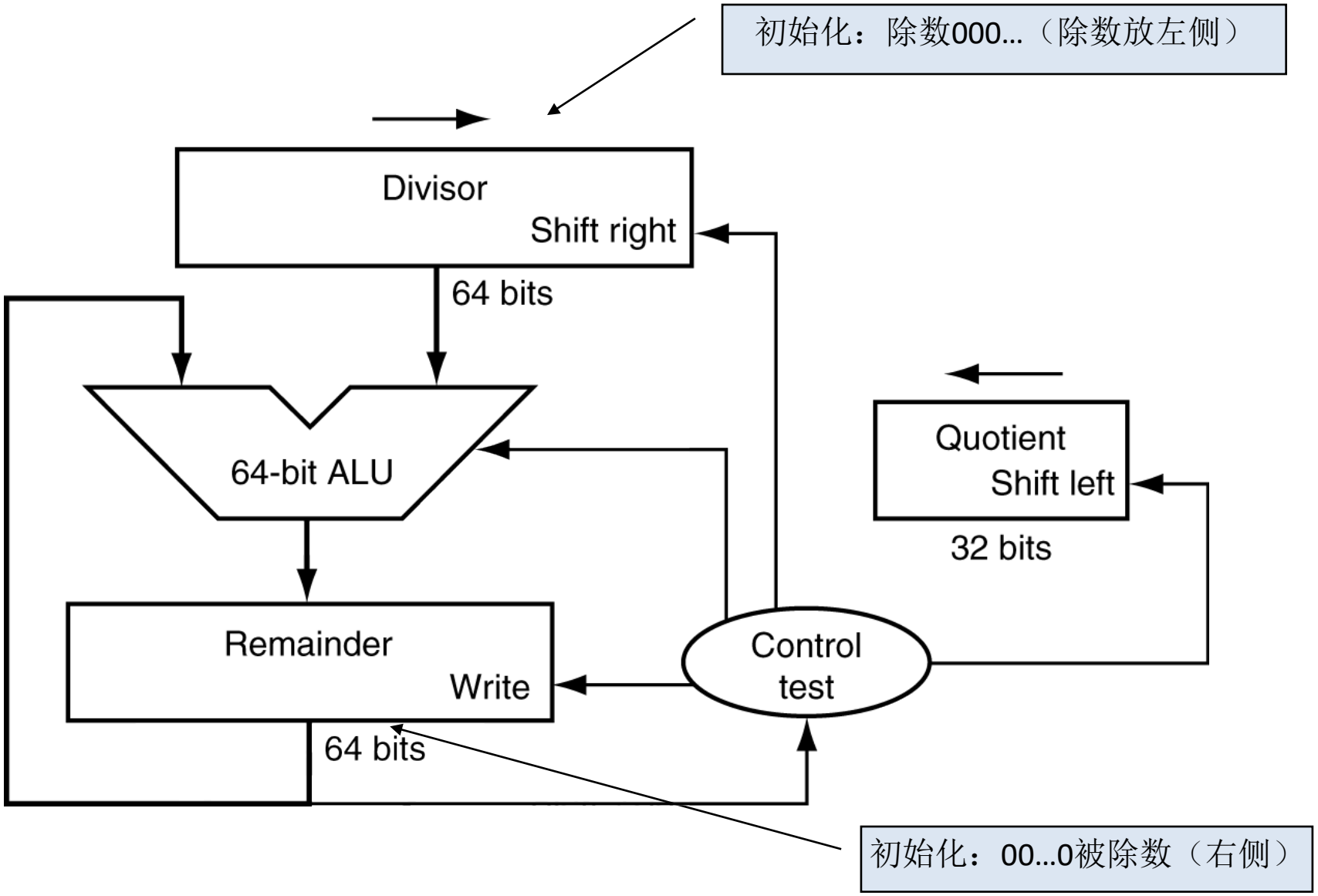
被除数 = 商 x 除数 + 余数



对位关系：除数右移

商：产生最后一位，左移

整数除法


初始化：除数000...（除数放左侧）
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
64 bits
Control
test
初始化：00...0被除数（右侧）

例：使用4位数据计算7/2。

7——0111——0000 0111 （因为4*4位为8位，所以被除数要补足位数。注：纯小数不是这样）
2——0010——0010 0000 （补位，对齐）

溢出检测：
8位被除数除以4位除数，商有效位不得超过4位。

被除数高4位 - 除数 >= 0 则溢出。

迭代	步骤说明	商	除数	余数
0	初值	0000	0010 0000	0000 0111
1	余=余-除			1110 0111
	余<0，恢复余数，商左移，商上0	0000		0000 0111
	右移除数		0001 0000	
2	同上			1111 0111
		0000		0000 0111
			0000 1000	
3	同上			1111 1111
		0000		0000 0111
			0000 0100	
4	余=余-除			0000 0011
	余>0，商左移，商上1	0001		
	右移除数		0000 0010	
5	同上			0000 0001
		0011		
			0000 0001	

- 余数左移代替除数右移——

  除数寄存器位数减少
- 余数、被除数、商合用一个左移寄存器——

  初始化：

每左移一次，上一位商

步骤

1）余数寄存器左移（假设商上0）

余数　　被	除数	商的0

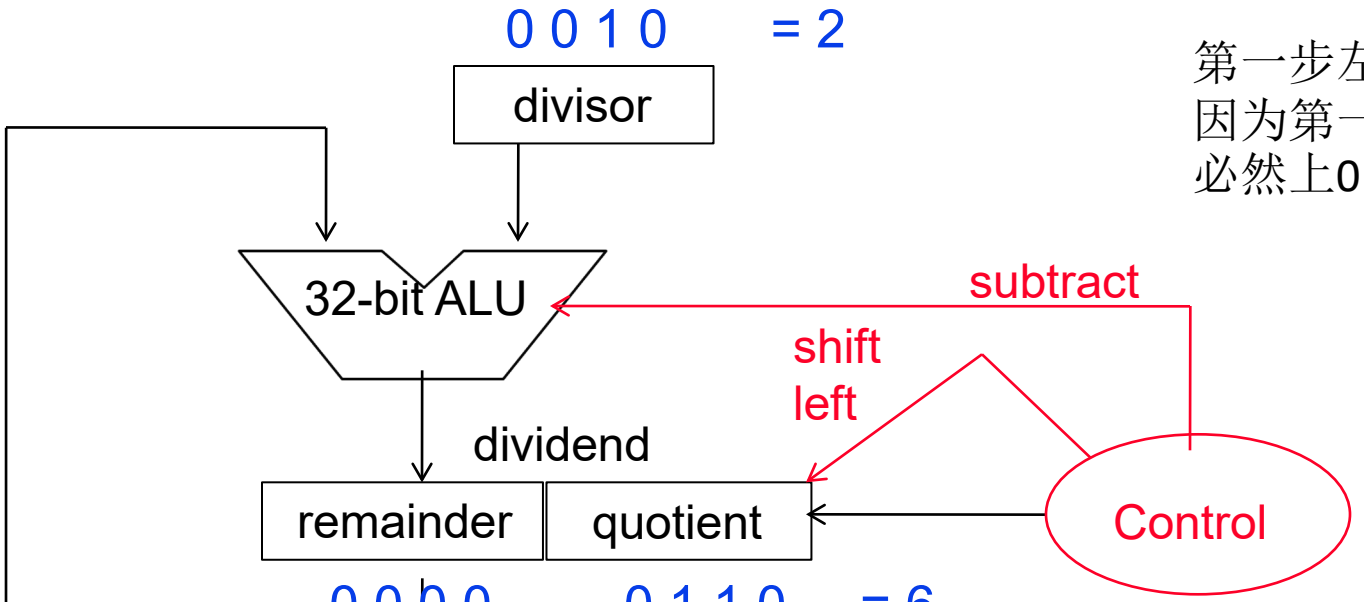
2）新余数 = 余数（高32位） – 除数

3）<0 不够减，恢复余数，转4）

>=0够减，改商位为1，转4）

4）循环次数32，否，转1）。是，结束。

Left Shift and Subtract Division Hardware



第一步左移，
因为第一步商
必然上0

	0 0 0 0	0 1 1 0	= 6
	0 0 0 0	1 1 0 0	
sub	1 1 1 0	1 1 0 0	rem neg, so ‘ient bit = 0
	0 0 0 0	1 1 0 0	restore remainder
	0 0 0 1	1 0 0 0	
sub	1 1 1 1	1 1 0 0	rem neg, so ‘ient bit = 0
	0 0 0 1	1 0 0 0	restore remainder
	0 0 1 1	0 0 0 0	
sub	0 0 0 1	0 0 0 1	rem pos, so ‘ient bit = 1
	0 0 1 0	0 0 1 0	
sub	0 0 0 0	0 0 1 1	rem pos, so ‘ient bit = 1

= 3 with 0 remainder

有符号除法

- 取绝对值参与运算
- 结果符号位为操作数符号位异或
- 非0余数的符号位与被除数相同

MIPS 除法指令

- 使用 HI/LO 两个寄存器存储结果
  - HI: 32-bit 余数
  - LO: 32-bit 商
- 指令
  - `div rs, rt  /  divu rs, rt`
  - 无溢出或被0除检测
    - 如需检测有软件完成
  - 使用 `mfhi,mflo` 来访问结果

原码一位除法*

设：被除数$[X]_{原}=x_f.x_1x_2\cdots x_n$

除数$[Y]_{原}=y_f.y_1y_2\cdots y_n$

则商的符号$Q_f=x_f\oplus y_f$

商的数值$|Q|=|X|/|Y|$

纯小数除法运算前，先要判断是否会产生溢出:

被除数-除数 >0

即商大于1，纯小数溢出，就表示溢出，此时除法就不进行，并由程序进行处理。

此外，还需判断除数不为0。

原码恢复余数除法

一、原码恢复余数除法

符号位单独处理，$Q_f=x_f \oplus y_f$，两个正数参加运算。

(1)余数=|被除（余）数|-|除数|

(2) >0 上 1

<0 上 0 且 +除数（称为恢复余数）

商上0时，由于在比较时已减去了除数，而商又上0，此时正确的余数应是|被除数|-0，因此需要加上除数，将余数还原成原来的数值再进行后面的运算。

(3)余数和商左移一位,返回(1)直到商的位数与除数相同。

(4)左移时末位补0。

［例］［x］$_{原}$=1.1001，［y］$_{原}$=0.1011，求[x/y]$_{原}$=?

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

迭代	步骤	余数	商
0	初始值	00.1001	0.0000
1	(x-y)比较，即+[-|y|]$_{补}$	11.0101	
	余数r_0＜0，商上0	11.1110	0.0000
	加y恢复余数	00.1001	0.0000
	左移	01.0010	0.0000
2	减y比较	11.0101	
	r_1＞0，商上1	00.0111	0.0001
	左移	00.1110	0.0010
3	减y比较	11.0101	
	r_2＞0，商上1	00.0011	0.0011
	左移	00.0110	0.0110
4	减y比较	11.0101	
	r_3＜0，商上0	11.1011	0.0110
	恢复余数	00.0110	0.0110

	左移	00.1100	0.1100
5	减y比较	11.0101	
	r_4＞0，商上1	00.0001	0.1101

$[Q]_{原}=2^0(x_f \oplus y_f)+|[Q]_{原}|=1.1101$

$[r_4]_{原}=2^0x_f+|[r_4]_{原}|=1.0001$

$[r_4]_{原真}=2^0x_f+2^{-4}[r4]_{原}=1.0000\ 0001$

余数与被除数同号

原码不恢复余数法

二、原码不恢复余数法

1. 原码不恢复余数法的原理:不恢复余数法的特点是不够减时不必恢复余数，而根据余数的符号作相应处理就可继续往下运算，因此运算步数固定、控制简单，提高了运算速度。

原码不恢复余数法

推证：设R_i是第i步的假余数，Ri<0，则

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

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

$=2R_i+|Y|$

将假余数当作真余数左移一位，再+|Y|即可得下一步余数。

不恢复余数法就是采用这种方法，所以又称为加减交替法。

原码不恢复余数法

2. 原码不恢复余数法求|Q|的运算规则如下：

符号位单独处理。

(1)被除数和除数取绝对值且为双符号位参与运算，并要求|x|<|y|

(2) 先用被除（余）数减去除数。

·当新余数为正时，商上1，余数左移一位，再减去除数。

·当新余数为负时，商上0，余数左移一位，再加上除数。

(3) 重复(2)，左移时末位补0

恒置1法，重复n次上商操作，最后一位商置1；

校正法，重复n+1次上商操作，第(n+1)次不左移，且当第n+1步余数为负时，需加上|y|得到第n+1步正确的余数。

无论哪种方法$[r_n]_{原真}= 2^{-n} |[r_n]|+x_f.0...0$(余数与被除数同号)。

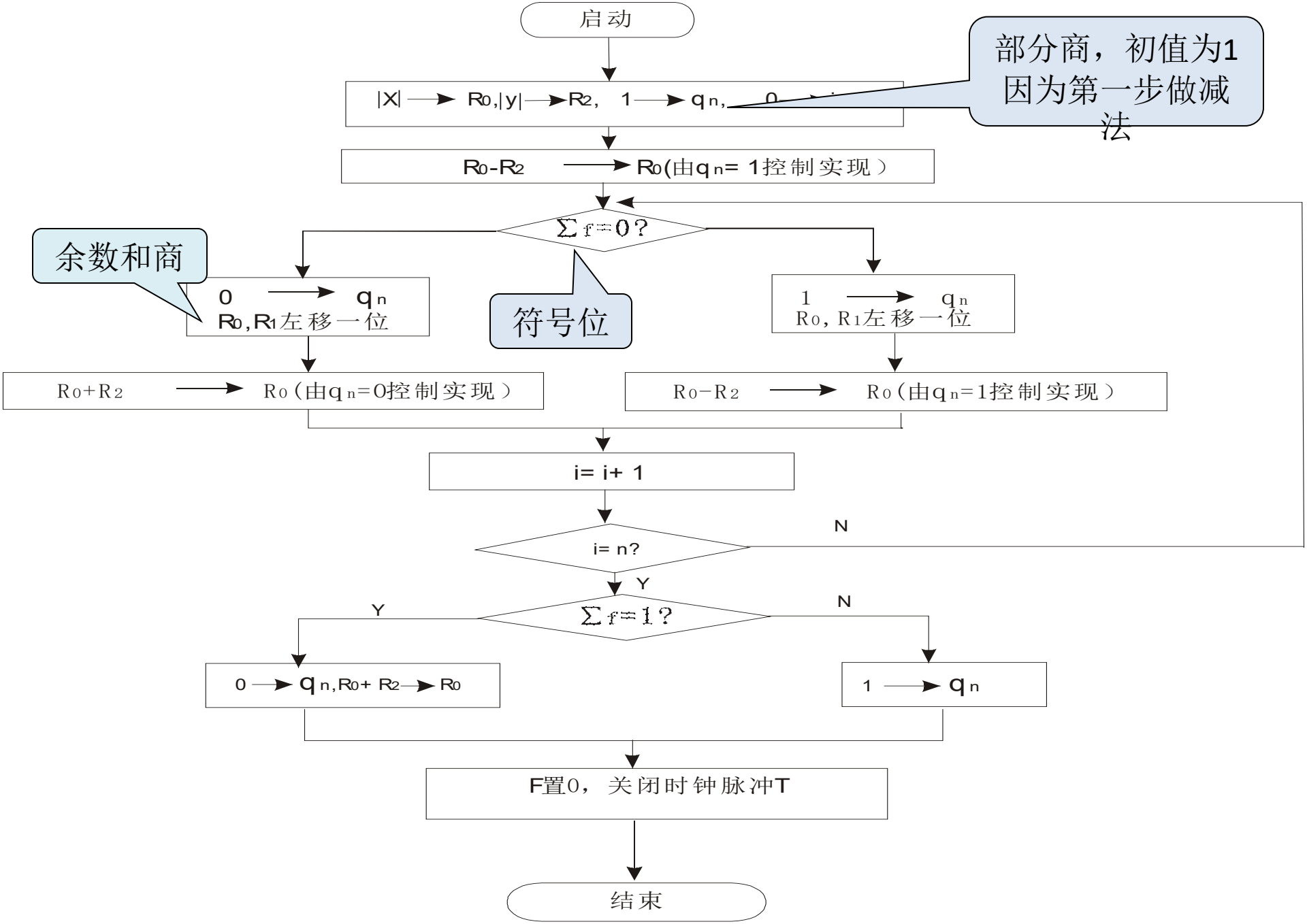


图3-7 原码不恢复余数法算法流程图

[例] $[x]_{原}$=1.1001， $[y]_{原}$=0.1011

用不恢复余数法求$[x/y]_{原}=[Q]_{原}$=?

解：$[|x|]_{原}$=00.1001，

$[|y|]_{原}$=00.1011，

$[-|y|]_{补}$=11.0101

运算过程见下页，

$[Q]_{原}$=1.1101

$r_4>0$所以是真余数

所以$[r4]_{原真}=2^0x_f+|[r_4]|2^{-4}$=1.0000 0001

若采用恒置1法，$[Q]_{原}$=1.1101，但余数 r4=11.0110为假余数。

r4=11.0110+00.1011=00.0001

所以$[r4]_{原真}$=1.0000 0001

迭代	步骤	余数	商
0	初值	00.1001	0.0000
1	(x-y)比较，第一次都是减法，$+[-\|y\|]_{补}$	11.0101	
	$r_0<0$，商上0	11.1110	0.0000
	左移一位	11.1100	0.000
2	余数为负，加$\|y\|$比较，$+[\|y\|]_{原}$	00.1011	
	$r_1>0$，商上1	00.0111	0.0001
	左移一位	00.1110	0.001
3	余数为正，减$\|y\|$比较，$+[-\|y\|]_{补}$	11.0101	
	$r_2>0$，商上1	00.0011	0.0011
	左移一位	00.0110	0.011
4	余数为正，减$\|y\|$比较	11.0101	
	$r_3<0$，商上0	11.1011	0.0110
	左移一位，若采用“恒置1法”，末位恒置1，结束	11.0110	0.110
5	余数为负，加$\|y\|$比较	00.1011	
	$r_4>0$，商上1	00.0001	0.1101

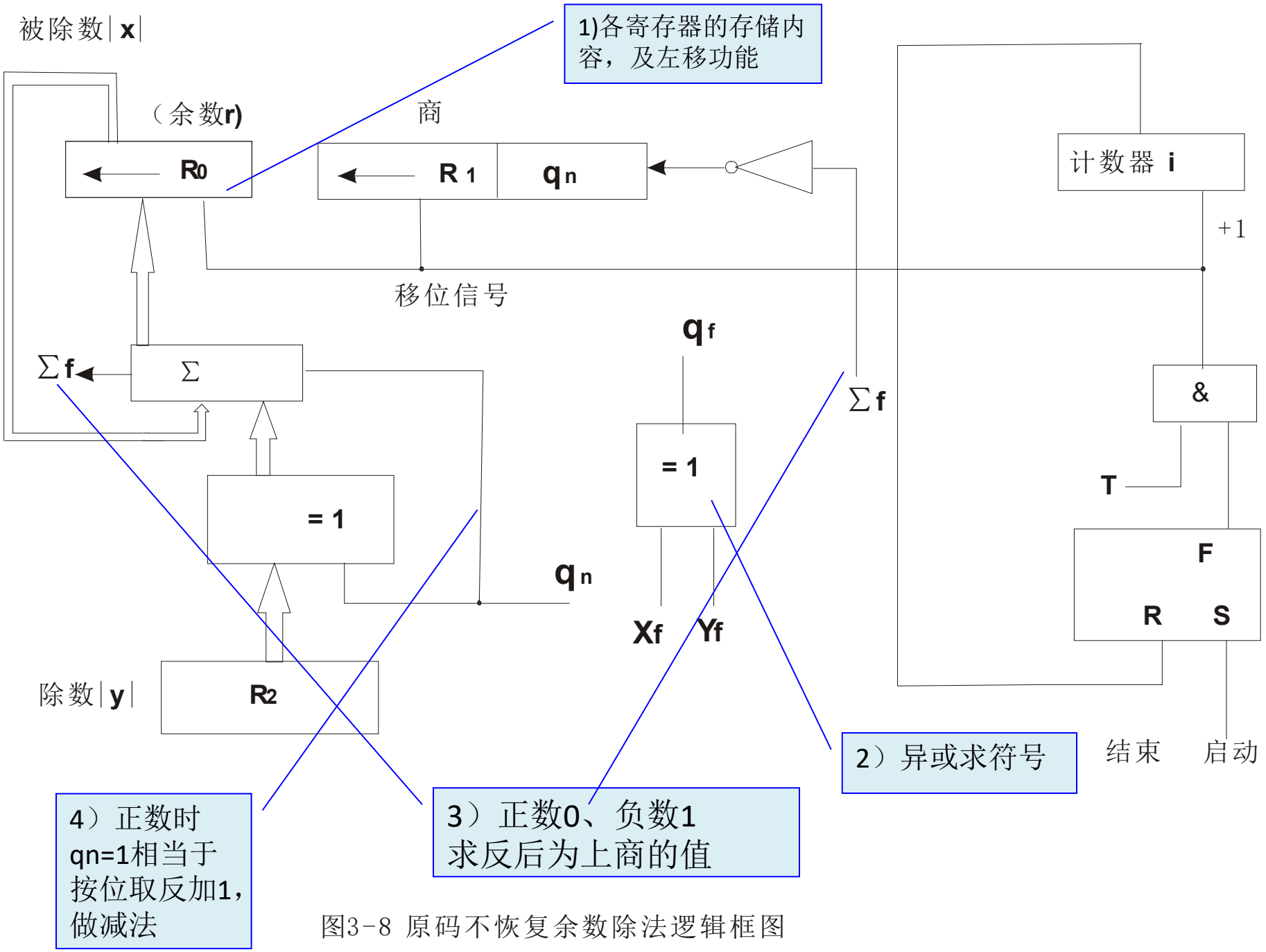


图3-8 原码不恢复余数除法逻辑框图

作业

- 3.19

3.5 浮点运算

规格化浮点数

- 科学记数法

  -2.34×10^{56} （规格化）

  $+0.002 \times 10^{-4}$

  $+987.02 \times 10^{9}$

- 二进制中的规格化数：

  $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

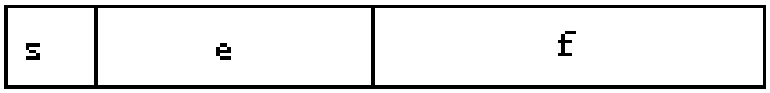
  没有前导0，且小数点左边只有一位非0数。

IEEE 754

- IEEE754中浮点数标准

1）IEEE标准委员会1985年3月批准的。内容包括：浮点数表示形式、类型、定义、舍入方式、例外处理等。

2）IEEE754浮点数的格式：尾数用原码，指数用移码。格式如下：



对于单精度字长32，s为符号位，阶码e为8位，f为23位，尾数最高位隐藏。

(1)若e=（1111 1111）$_2$=255且f≠0，则无论s取何值，浮点数v为NaN（非数）。

(2)若e=255且f=0，则v=$(-1)^s*\infty$，为±∞。

(3)若0<e<255, 则v=$(-1)^s*2^{e-127}*(1.f)$。

即：$v = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$

(4)若e=0且f ≠0，则v=$(-1)^s*2^{e-126}*$

(5)若e=0且f =0，则v=$(-1)^s*0$，即

对于双精度字长64，阶码e位，尾数最高位隐藏。相应25值改为2047、1023、1022。

国内教材讲的规格化形式是：移码偏移量为128，尾数规格化为0.1f….

IEEE754规格化为：1.f的形式，移码偏移量127

e总是无符号数（正），最小的负数是0，最大的正数是111…1

格式定义的原因1,2P133

例： $0.15625=(0.00101)_2$

$0.00101 \quad 0.101*2^{-2}$ 规格化

$1.01*2^{-3}$ IEEE规格化

阶码偏移量为127，所以：

$e = 127-3 = 124 = (0111\ 1100)_2$

f = 010…0

所以，IEEE754单精度浮点数为：

0 0111 1100 010…0

例：　-5=-101

数符 s = 1

IEEE规格化$1.01*2^2$

e=127+2=129=$(1000\ 0001)_2$

f = 010…0

IEEE754单精度浮点数：

1 1000 0001 010…0

例：　IEEE754单精度浮点数用十进制表示。

1 1000 0001 0010…0

S=1 e=129 f=$(0.001)_2$=1/8=0.125

代入公式（3）可得-4.5

例： –0.75

$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$

s = 1

f = $1000\ldots00_2$

e = –1 + Bias

Single: –1 + 127 = 126 = 01111110_2

Double: –1 + 1023 = 1022 = 01111111110_2

Single: 10111111001000…00

Double: 10111111111101000…00

单精度浮点数表示范围

- Exponent8位：00000000 和11111111保留，见(1)(2)(4)(5)
- 最小值
  - exponent: 00000001
    ⇒ exponent - Bias = 1 – 127 = –126
  - Fraction: 000…00 ⇒ 实际值 = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- 最大值
  - exponent: 11111110
    ⇒ exponent - Bias = 254 – 127 = +127
  - Fraction: 111…11 ⇒ 实际值 ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

双精度浮点数表示范围

- Exponent 0000…00 和 1111…11 保留
- 最小值
  - Exponent: 00000000001
    ⇒ exponent – bias = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ 实际值 = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- 最大值
  - Exponent: 11111111110
    ⇒ exponent – bias = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ 实际值 ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

浮点数溢出

浮点数的溢出是对规格化的阶码进行判断：

阶码大于机器的最大阶码（指数域放不下），称为上溢，此时机器不能再继续运算，一般进行中断处理。

阶码小于最小阶码（绝对值太大），称为下溢，把浮点数各位强迫为0，机器仍可继续运行。

作业

3.23　3.24

浮点数的加法

- 以4位十进制尾数，2位指数为例：

  $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

  1. 对阶：小阶向大阶看齐

     $9.999 \times 10^{1} + 0.016 \times 10^{1}$

  2. 尾数（有效数）相加

     $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

  3. 规格化、判断溢出（指数不超过表示范围）

     1.0015×10^{2}

  4. 舍入（按尾数的字长，四舍五入）

     1.002×10^{2}　　重新规格化（若需要）

浮点数的加法

- 例2：求4位二进制数的和：$(0.5_{10} + -0.4375_{10})$

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

1. 对阶

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. 尾数加

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

3. 规格化、判溢出

   $1.000_2 \times 2^{-4}$，无溢出

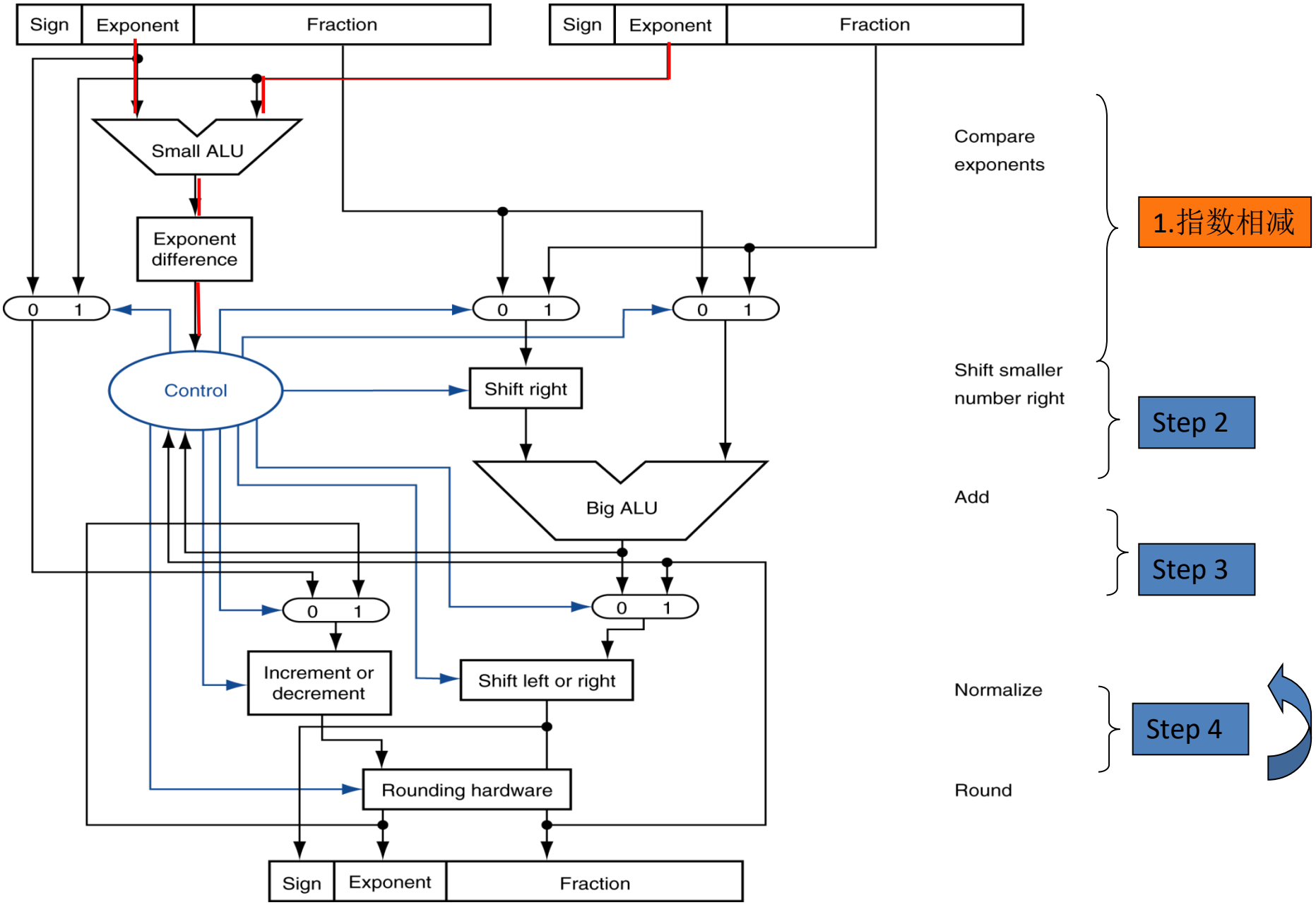
4. 舍入（无变化）

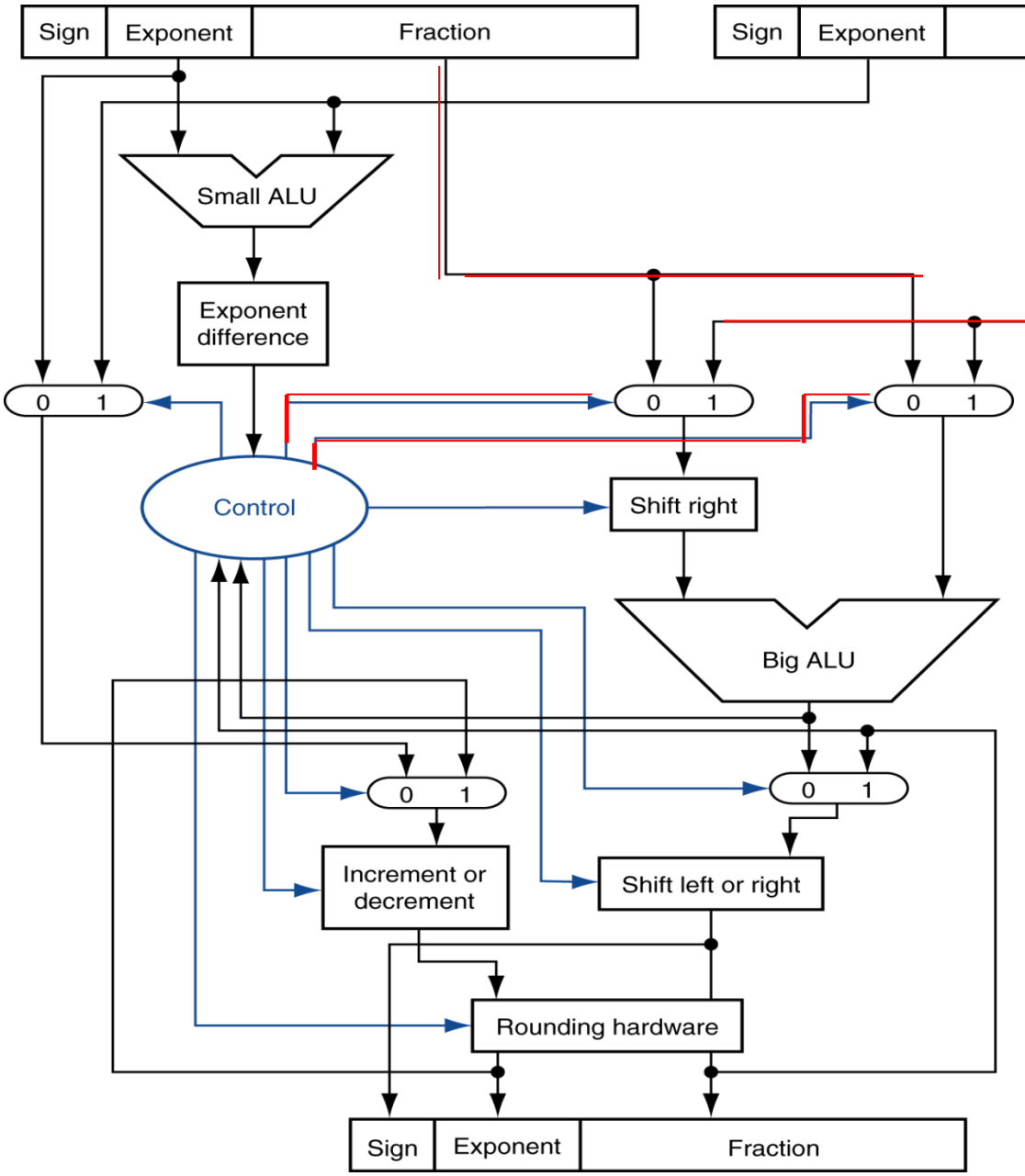
   $1.000_2 \times 2^{-4} = 0.0625$

浮点加法器的实现

- 比整数加法器复杂得多
- 若用一个时钟周期完成浮点加法会使时钟周期变得很长。

  所以，一般都用几个时钟周期来完成。


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
1.指数相减
Shift smaller number right
Step 2
Add
Step 3
Normalize
Step 4
Round



Compare exponents

Shift smaller number right

Add

Normalize

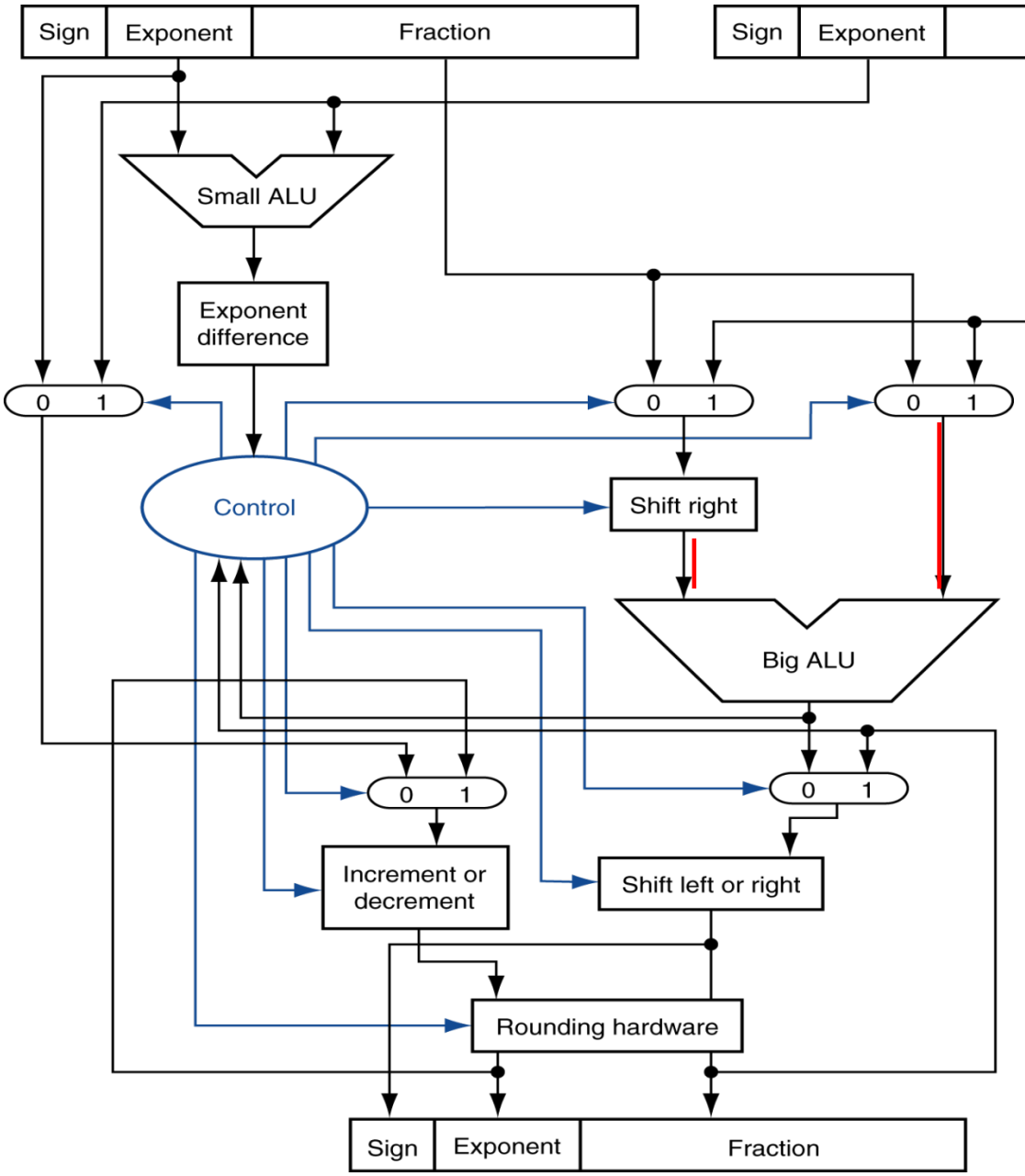
Round

1.指数相减

2.将较小指数的尾数放到左侧左侧，右移，对阶

Step 3

Step 4



Compare exponents

Shift smaller number right

Add

Normalize

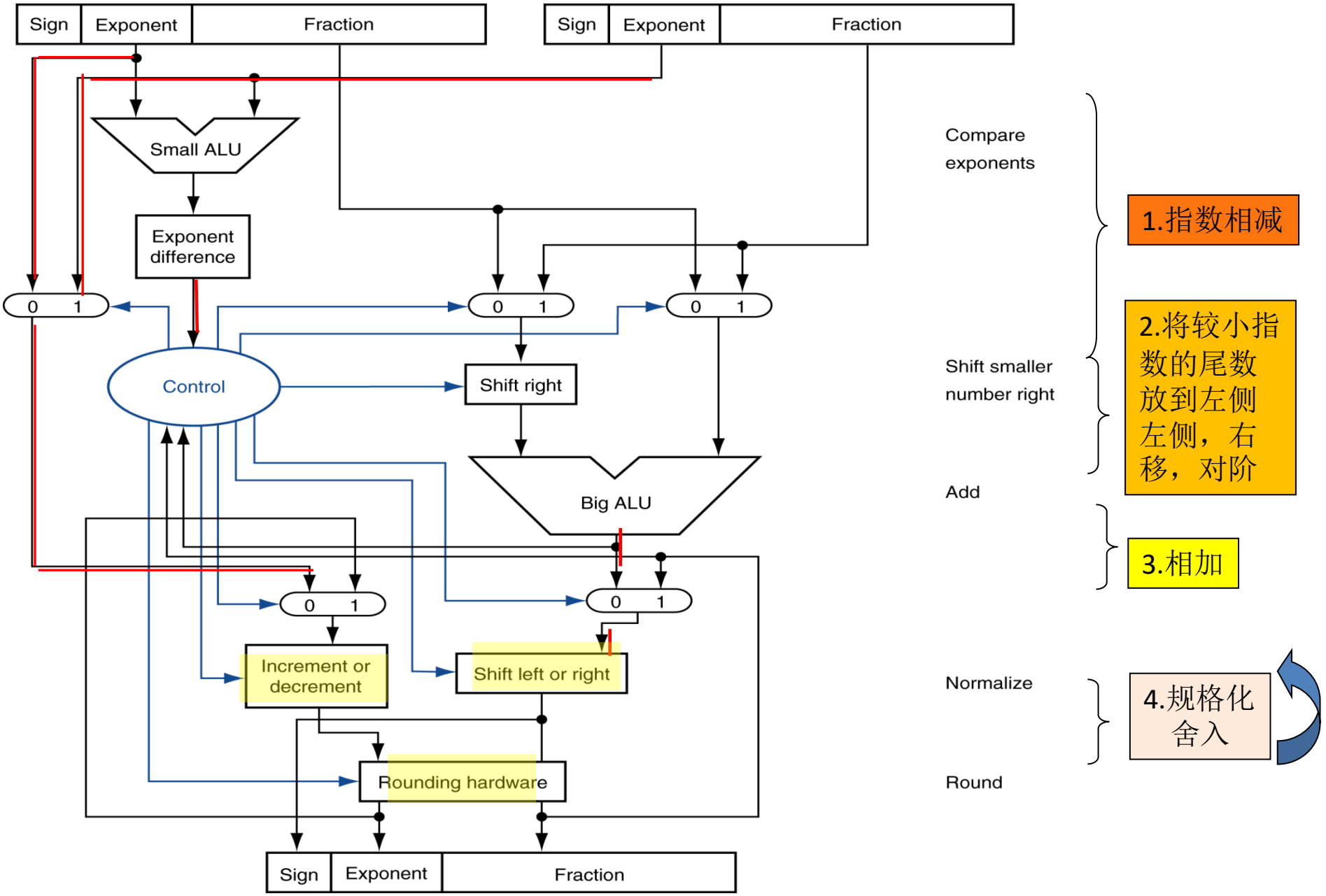
Round

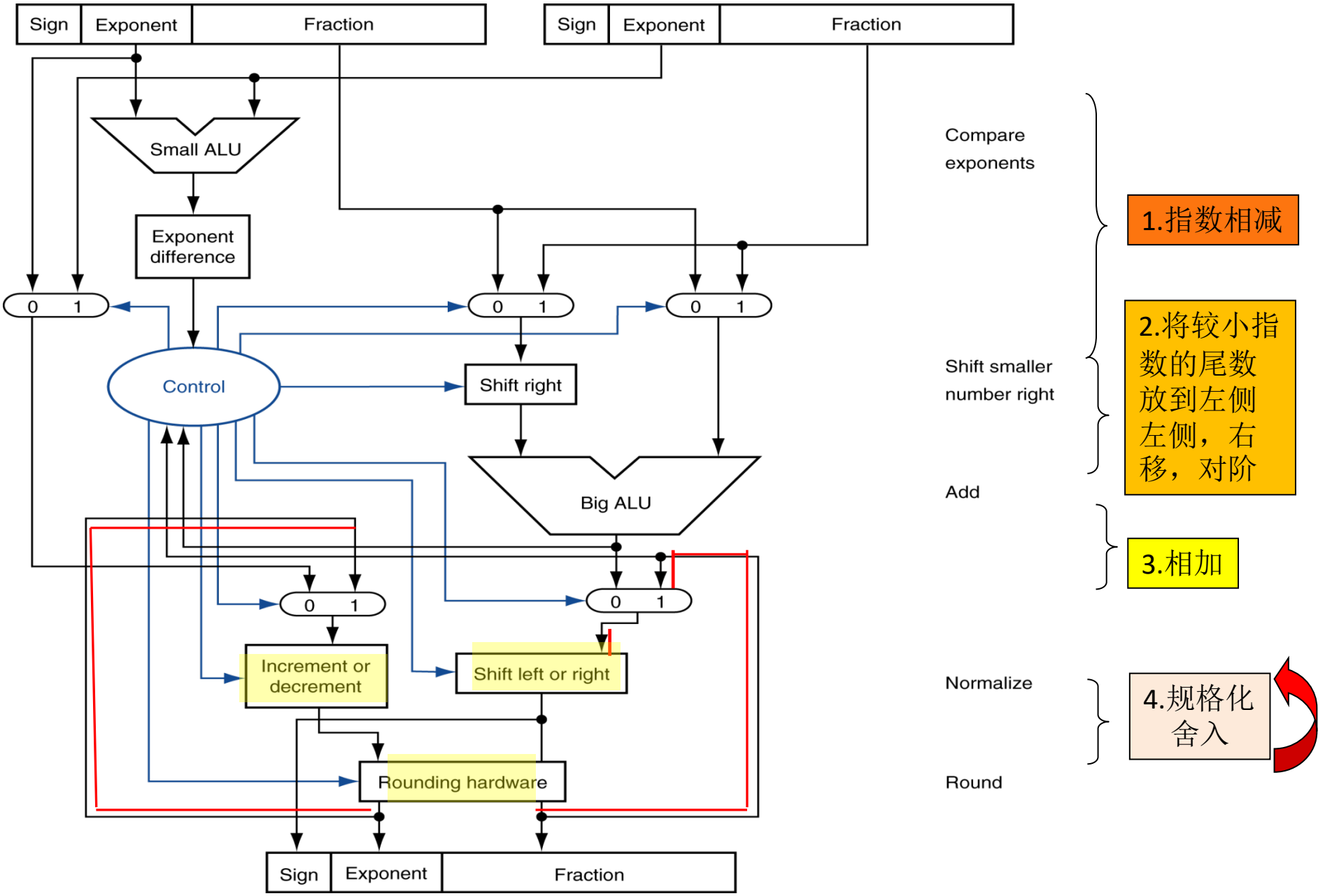
1.指数相减

2.将较小指数的尾数放到左侧左侧，右移，对阶

3.相加

Step 4


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
1.指数相减
2.将较小指数的尾数放到左侧左侧，右移，对阶
3.相加
4.规格化舍入


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
1.指数相减
2.将较小指数的尾数放到左侧左侧，右移，对阶
3.相加
4.规格化舍入

浮点数乘法

- 以4位十进制尾数，2位指数为例：

  $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

  1. 指数相加

     对于有偏阶的指数，要减去一个偏阶

     新的指数 = 10 + –5 = 5

  2. 尾数相乘

     $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$

  3. 规格化、判溢出

     1.0212×10^{6}

  4. 舍入（若需要再次规格化）

     1.021×10^{6}

  5. 依据操作数的符号位确定结果符号位

     $+1.021 \times 10^{6}$

浮点数乘法

- 例：用4位二进制求：(0.5 × –0.4375)

  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$

  1. 指数相加

  无偏阶时： $-1 + -2 = -3$

  有偏阶时： $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

  2. 尾数相乘

  $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$

  3.规格化、判溢出

  $1.110_2 \times 2^{-3}$ （无变化）

  4.舍入（若需要再次规格化）

  $1.110_2 \times 2^{-3}$ （无变化）

  5. 确定符号位: +ve × –ve ⇒ –ve

  $-1.110_2 \times 2^{-3} = -0.21875$

浮点数乘法

- 浮点乘法器与浮点加法器非常相似，只需将尾数加法器换成乘法器即可。
- 通常也需要几个时钟周期完成。

浮点数指令

- 专门的浮点寄存器
  - 32位单精度： $f0, $f1, … $f31
  - 成对使用用于双精度： $f0/$f1, $f2/$f3, …（偶数号做为其名称）
  - P157浮点寄存器编号0-31，P528协处理器
- 浮点指令仅操作于浮点寄存器
- FP存取指令

lwc1, ldc1, swc1, sdc1

例： ldc1 $f8, 32($sp)

浮点数指令

- 单精度算数指令
  add.s, sub.s, mul.s, div.s

  例：add.s $f0, $f1, $f6

- 双精度算数指令
  add.d, sub.d, mul.d, div.d

  例：mul.d $f4, $f4, $f6

- 单双精度比较指令
  c.*xx*.s, c.*xx*.d (*xx* 表示eq, lt, le, …)设置或清除浮点标志位

  例：c.lt.s $f3, $f4
- 浮点数分支指令：bc1t, bc1f

  例：bc1t TargetLabel

  P158 表 cond cc解释

算术精确性

- 如果运算过程中，中间结果都被截短成准确的位数，就无法舍入了
- IEEE754在中间计算时，右边多保留两位：保护位、舍入位。
- 当舍入位右边有非0数时，粘贴位被置1

算术精确性

例：3位十进制尾数，说明保护位、舍入位用途。

$2.56 \times 10^0 + 2.34 \times 10^2$

对阶：$0.0256 + 2.34 = 2.3656$

舍入：2.37×10^2

乘法对舍入位的需要：精解

若无保护位、舍入位：

对阶：$0.02 + 2.34 = 2.36$

误差0.01

算术精确性

1. 舍入到最接近,在一样接近的情况下偶数优先（这是默认的舍入方式）：会将结果舍入为最接近且可以表示的值,但是当存在两个数一样接近的时候，则取其中的偶数（在二进制中是以0结尾的）

2.向+∞方向舍入：会将结果朝正无限大的方向舍入。
例如： ceil(1.324) = 2。Ceil(-1.324) = -1

3. 向-∞方向舍入： 会将结果朝负无限大的方向舍入。
例如：floor(1.324) = 1，floor(-1.324) = -2

4.向0方向（截断）舍入：
例如：(int) 1.324 = 1，(int) -1.324 = -1

算术精确性

- 粘贴位用于记住舍入位之后有丢掉的1，以便更精确。
- 例：3位尾数，$5.01\times10^{-1} + 2.34\times10^{2}$

  对阶：0.0050 + 2.34 = 2.345

  向最接近的偶数舍入：2.34

  （2.35一样接近）

  有粘贴位时： 0.00501 + 2.34 = 2.3451

  则最接近的只有2.35

其他

- 混合乘加指令：减少舍入次数，增加精度
- 非规格化数的用处。

- 计算机算术和真实世界的算术是有差异的。

余-N码

- P178　3.11.2

余-N码：十进制数+N再转换成二进制就是该十进制数的余-N码。

例：余3码

十进制　余3码

0　0011

1　0100

2　0101…

IEEE754　余-127码

例：$-1278*10^3$的NVIDIA表示

$-1278*10^3 = -1278000$

$= -100111000000000110000$

$= -1.00111000000000110000*2^{20}$

Exponent = 20+16 = 36

NVIDIA指数位5位最大11111 = 31

所以溢出

S符号位	指数位 5 位	尾数 10 位

例：$2.3109375*10^1$的NVIDIA表示

$2.3109375*10^1 = 23.109375$

$= 10111.000111$

$= 1.0111000111*2^4$

Exponent = 4+16 = 20

NVIDIA指数位5位最大10100 = 20

所以0 10100 0111000111

十进制小数转二进制，*2，看进位，有进位减掉，再乘

作业

- 3.27

- 原
- 3.11.4 a用IEEE754 b用NVIDIA

不必写出浮点数的全部各位，实际上只是尾数IEEE754为23位+保护位G+舍入位R+粘贴位S，而NVIDIA为10位+G+R+S

- 3.12.1 a用IEEE754 b用NVIDIA

3.11.4b

$2.3109375*10^{1} = 1.0111000111*2^{4}$

$6.391601562*10^{-1} = 0.6391601562$

$= 0.1010001110\ 0111$

规格化 (可省略) $= 1.0100011100111*2^{-1}$

对阶 $= 0.00001010001110011100111*2^{4}$

1.0111000111 000 （GRS）

\+ 0.0000101000 111

1.0111101111 111

由于GRS为111，说明该数介于1.0111101111和1.0111110000之间，若为GRS为100则在当中，现在更偏向大的数，所以取1.0111110000

$2.3109375*10^{1} + 6.391601562*10^{-1} = 1.0111110000*2^{4}$ = 10111.11 = 23.75

3.6 结合律

- 整数加法符合结合律，浮点数加法不符合结合律。

3.8 陷阱与谬误

- 左移乘2（正确），右移除2（仅对无符号数成立）
- 对于有符号数
  - 算数右移，补入符号位
  - 例： –5 / 4（不够精确）
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - 逻辑右移，补入0 （完全不对）
    - 11111011_2 >> 2 = 00111110_2 = +62
- 浮点数的精度

本章小结

- 下周小测验（因为有期中考试，所以不考了）
- 下周交作业