第3章 chapter 3

运算方法和运算器

本章详细内容参见图 3-0。

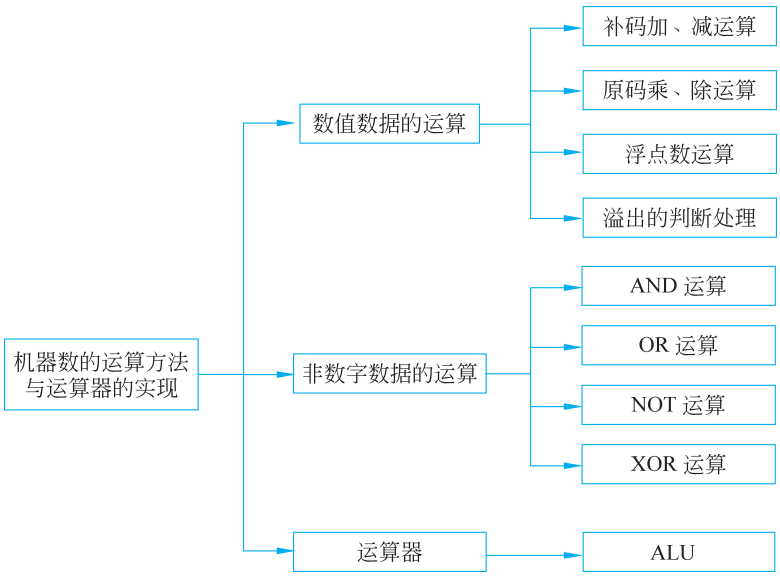


图 3-0　知识点思维图

计算机要对各种信息进行加工和处理，如对数值数据进行加、减、乘、除等数值运算，对非数值数据进行与、或、非等逻辑运算。在计算机中必须有对数据进行处理和加工的部件，这个部件就是运算器。目前，大多数计算机都将运算器和控制器集成在一个芯片上，也就是常说的中央处理器单元(central processing unit，CPU)。

课程思政材料

在计算机内部，所有的数据都以各种不同的编码方式以机器数的形式存在，符号、小数点也都用数字来表示，这样用常规的运算方法来设计运算器对机器数直接进行运算已经无法满足要求。本章重点讨论各种不同编码表示的机器数的运算规律，并以此来设计运算器。

本章主要讨论以下问题。

(1) 定点数的补码运算规律及定点加法器的实现。

(2) 溢出的概念及溢出判别方法。

(3) 定点数的原码运算规律及乘除法器的实现。

(4) 浮点数的运算方法及浮点运算器的实现。

（5）影响运算器速度的主要因素及提高运算器速度的主要方法。

设计运算器必须满足以下要求。

- 保证运算结果正确。
- 结构尽可能简单。
- 最大限度地提高运算速度。

视频：补码加减运算

3.1 定点数的加减法运算

定点数的加减在计算机中通常采用补码进行，运算时连同符号位一起运算，结果也用补码表示。

3.1.1 补码加减法所依据的关系

加法：$[X+Y]_{补}=[X]_{补}+[Y]_{补}$

减法：$[X-Y]_{补}=[X]_{补}+[-Y]_{补}=[X]_{补}+\overline{[Y]}_{补}+1$（最末位）

其中，$[-Y]_{补}$是将$[Y]_{补}$连同符号位一起取反后，再在最低位加1，即$[-Y]_{补}=\overline{[Y]}_{补}+1$（最末位）。也就是说，在计算机内部，减法最终是通过加法来实现的，这对运算器的物理实现将会带来很大的简化。

3.1.2 补码加减法运算规则

补码加减法运算规则如下。

（1）参加运算的操作数用补码表示。

（2）符号位等同于数值一起参加运算。

（3）对于两数相加减的各种情况，计算机都执行加法操作。当操作码为加运算时就直接进行相加，当操作码为减时，将减数连同符号位一起求反、末位加1后再与被减数相加。

（4）结果用补码表示。

以上关系不做证明，仅通过实例来验证。

【例3-1】 已知$X=0.1001$，$Y=-0.0110$，机器字长为5位，用补码运算求$X+Y$。

解：

$$\begin{array}{rl} & [X]_{补}=0.1001 \\ + & [Y]_{补}=1.1010 \\ \hline [X+Y]_{补}= & \boxed{1}\ 0.0011 \end{array}$$

超过机器字长，自然丢失

所以$X+Y=0.0011$。

【例3-2】 已知$X=-0.1001$，$Y=-0.0101$，机器字长为5位，用补码运算求$X+Y$。

解：
$$\begin{aligned}&[X]_{补}=1.0111\\+\quad&[Y]_{补}=1.1011\\\hline&[X+Y]_{补}=\boxed{1}\ 1.0010\end{aligned}$$

超过机器字长，自然丢失

所以 $X+Y=-0.1110$。

【例 3-3】 已知 $X=0.1001$，$Y=0.0110$，机器字长为 5 位，用补码运算求 $X-Y$。

解：
$$\begin{aligned}&[X]_{补}=0.1001\qquad[Y]_{补}=0.0110\\+\quad&[-Y]_{补}=1.1010\\\hline&[X-Y]_{补}=\boxed{1}\ 0.0011\end{aligned}$$

超过机器字长，自然丢失

所以 $X-Y=0.0011$。

【例 3-4】 已知 $X=-0.1001$，$Y=-0.0110$，机器字长为 5 位，用补码运算求 $X-Y$。

解：
$$\begin{aligned}&[X]_{补}=1.0111\qquad[Y]_{补}=1.1010\\+\quad&[-Y]_{补}=0.0110\\\hline&[X-Y]_{补}=1.1101\end{aligned}$$

所以 $X-Y=-0.0011$。

【例 3-5】 已知 $X=+0.1011$，$Y=+0.1001$，机器字长为 5 位，用补码运算求 $X+Y$。

解： $[X]_{补}=0.1011$，$[Y]_{补}=0.1001$
$$\begin{aligned}&[X]_{补}=0.1011\\+\quad&[Y]_{补}=0.1001\\\hline&[X+Y]_{补}=1.0100\end{aligned}$$

所以 $X+Y=-0.1100$。

两个正数相加，结果不可能是负数，显然这结果是错误的。造成错误的原因是 $X+Y=0.1011+0.1001=1.0100>1$，超过了定点小数补码的表示范围，也就是溢出了。

3.1.3 溢出的概念及检测方法

视频：溢出的判断方法

由于计算机字长限制，溢出造成的大型事故是非常多的，例如，1996 年阿丽亚娜 5 火箭第一次发射时就凌空爆炸。事后分析原因时发现，火箭的一段控制程序是直接移植自阿丽亚娜 4 火箭的。而这段程序里的一个需要接收 64 位数据的变量为了节省存储空间使用了 16 位字。当时的程序员仔细计算了阿丽亚娜 4 的代码，分析出这个变量收到的数据无论如何不会超过 16 位。故此采取了这个方案。可是阿丽亚娜 5 接收的数据会超过 16 位，事后却没有人检查出这个问题，于是飞行中变量就溢出了。然后火箭就变成一个几亿欧元的大“烟花”。大国制造，一定要有工匠精神，工作要一丝不苟，否则将会给国家带来不可弥补的损失。

1. 溢出的概念

加法器或寄存器由多少个二进制位组成，这通常称为定点运算器的字长（或运算器一次处理二进制数的位数）。当选定了运算器字长及数据表示格式后，所能表示的数据范围也就确定了。计算机执行算术运算所产生的结果超出机器数所能表示的数的范围称为溢出。只有符号相同的两数进行加法运算才有可能产生溢出。计算机进行数值运算时，必须对运算结果是否产生溢出进行判断，以保证运算结果的正确性。


视频：逻辑代数复习

2. 溢出的检测方法

1）采用一位符号位判断

设 A 和 B 是两个操作数，S 为运算结果

$$A = a_n a_{n-1} \cdots a_0$$

$$B = b_n b_{n-1} \cdots b_0$$

$$S = A + B = s_n s_{n-1} \cdots s_0$$

式中，a_n、b_n、s_n 分别代表两个操作数和结果的符号。当 $a_n=1$，$b_n=1$ 而 $s_n=0$（两个负数相加，结果是正数，说明产生溢出，且为负溢出），或当 $a_n=0$，$b_n=0$ 而 $s_n=1$（两个正数相加，结果是负数，说明也产生了溢出，且为正溢出），真值表如表 3-1 所示。

表 3-1　采用一位符号位判断的真值表

a_n	b_n	s_n	OVER
0	0	1	1
1	1	0	1

溢出位 OVER$=\bar{a}_n\bar{b}_n s_n + a_n b_n \bar{s}_n$。图 3-1 所示为溢出判断电路。

2）用 c_{n-1} 和 c_n 判断

c_{n-1} 表示最高数值位向符号位的进位，c_n 表示符号本身产生的进位，在进行补码加法运算时，如果最高数值位和符号位不同时产生进位，则结果产生溢出，其真值表如表 3-2 所示。溢出位 OVER$=c_{n-1} \oplus c_n$。图 3-2 所示为溢出判断电路。

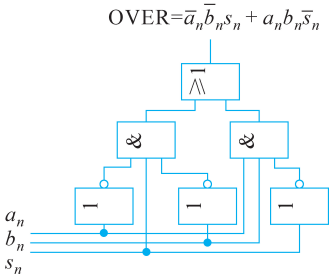


图 3-1　溢出判断电路 1

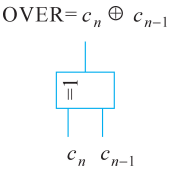


图 3-2　溢出判断电路 2

表 3-2 用 c_{n-1} 和 c_n 判断的真值表

c_n	c_{n-1}	OVER
0	1	1(这是两个正数相加,结果是负数的情况)
1	0	1(这是两个负数相加,结果是正数的情况)

3) 采用双符号位(也称变形补码)判断

用 s_{f1} 和 s_{f2} 表示最高符号位和第二个符号位,00 表示正数,11 表示负数。两个正数(符号位为 00)相加,结果的符号位为 01,表示有正溢出;两个负数(符号位为 11)相加,结果的符号位为 10,表示有负溢出,真值表如表 3-3 所示。溢出位 OVER $=s_{f1}\oplus s_{f2}$。图 3-3 所示为溢出判断电路。

表 3-3 采用双符号位判断的真值表

s_{f1}	s_{f2}	OVER
1	0	1
0	1	1

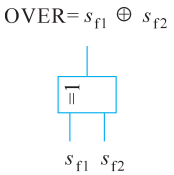


图 3-3 溢出判断电路 3

根据上面的讨论,对例 3-5,运用溢出判断公式,有

$$\text{OVER}=\bar{a}_n\bar{b}_n s_n+a_n b_n\bar{s}_n=1\cdot 1\cdot 1+0\cdot 0\cdot 0=1$$

或 OVER $=c_{n-1}\oplus c_n=1\oplus 0=1$,结果溢出。

【例 3-6】 $X=-0.1101$, $Y=-0.1011$,机器字长 6 位,求 $X+Y$。采用变形补码。

解:

$$[X]_{补}=11.0011$$
$$+\quad [Y]_{补}=11.0101$$
$$[X+Y]_{补}=110.1000$$

自然丢失

结果的符号位是 10, OVER $=s_{f1}\oplus s_{f2}=1\oplus 0=1$ 溢出,且负溢出。

为了保证运算结果的正确,必须对运算结果进行溢出检测,并把检测的结果保存在标志寄存器 FLAG(或程序状态字寄存器 PSW)中,然后采用指令对溢出标志位进行检测。标志寄存器 FLAG 中除了溢出标志外,还有一些其他标志位,如进位标志、符号标志、结果为零标志等。

3.2 二进制加法器

视频:半加器和全加器

3.2.1 半加器

两个一位二进制数 A_i、B_i 相加(不考虑低位的进位),称为半加。实现半加操作的电路称为半加器。根据二进制运算规则得到表 3-4 所示的半加器真值表。其中 A_i、B_i 为参加运算的操作数,S_i 为和,C_i 为进位位。

表 3-4　半加器真值表

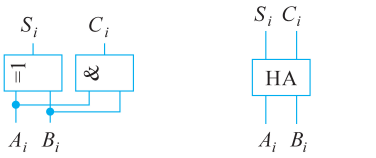
A_i	B_i	S_i	C_i	A_i	B_i	S_i	C_i
0	0	0	0	1	0	1	0
0	1	1	0	1	1	0	1

根据真值表可以写出逻辑表达式：

$$S_i = \overline{A}_i B_i + A_i \overline{B}_i = A_i \oplus B_i$$

$$C_i = A_i B_i$$

根据逻辑表达式，用门电路实现如图 3-4 所示的半加器逻辑电路，并用逻辑符号表示。



(a) 半加器逻辑电路　(b) 半加器逻辑符号

图 3-4　半加器

3.2.2　全加器

在实现多位二进制数相加时，不仅考虑本位，还要考虑从低位来的进位。考虑低位的进位加法运算就是全加运算，实现全加运算的电路称为全加器。根据二进制运算规则可以得到表 3-5 所示的全加器真值表。其中 A_i、B_i 为参加运算的操作数，S_i 为和，C_i 为本位向高位进位位，C_{i-1} 为从低位来的进位。

表 3-5　全加器真值表

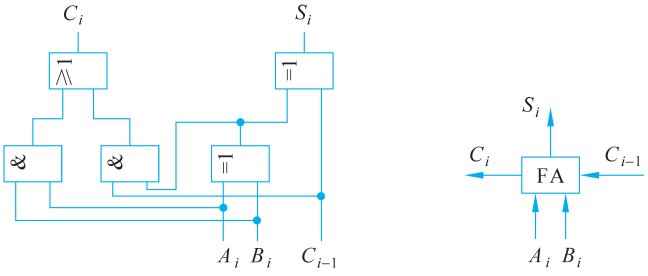
A_i	B_i	C_{i-1}	S_i	C_i	A_i	B_i	C_{i-1}	S_i	C_i	A_i	B_i	C_{i-1}	S_i	C_i
0	0	0	0	0	1	1	0	0	1	1	0	1	0	1
0	1	0	1	0	0	0	1	1	0	1	1	1	1	1
1	0	0	1	0	0	1	1	0	1					

根据真值表可以写出逻辑表达式并进行化简：

$$S_i = \overline{A}_i B_i \overline{C}_{i-1} + A_i \overline{B}_i \overline{C}_{i-1} + \overline{A}_i \overline{B}_i C_{i-1} + A_i B_i C_{i-1} = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i B_i \overline{C}_{i-1} + \overline{A}_i B_i C_{i-1} + A_i \overline{B}_i C_{i-1} + A_i B_i C_{i-1} = A_i B_i + (A_i \oplus B_i) C_{i-1}$$

根据逻辑表达式，用门电路实现如图 3-5 所示的全加器逻辑电路，并用逻辑符号表示。



(a) 全加器逻辑电路图　(b) 全加器逻辑符号

图 3-5　全加器

一位全加器只能实现一位二进制数的加法，另外门电路都有延迟，这个延迟影响了整

个加法器的运算速度。假设一级门的延迟时间是 1ty，则产生 S_i 要两级门延时时间 2ty，产生进位 C_i 要三级门延时时间 3ty（为了简化讨论，假设所有门电路的延时时间相同）。

3.2.3 加法器


视频：串行进位加法器

一位全加器只能实现一位二进制数的加法，要实现 n 位数相加就要组成加法器，最基本的加法器就是串行加法器和并行加法器。

1. 串行加法器

如果 n 位字长的加法器仅有一位全加器，使用移位寄存器从低位到高位串行地提供操作数，分 n 步进行相加，称为串行加法器。图 3-6 所示为串行加法器框图。移位寄存器 A 中存放一个操作数 A，并保存最后的运算结果，移位寄存器 B 中存放另一个操作数 B。移位寄存器在移位脉冲的控制下，将低位送到全加器的输入端进行加法运算。串行加法器每步操作产生一位和，并串行地送入结果寄存器中（通常用累加器，在图 3-6 中的移位寄存器 A），进位信号则用一位 D 触发器寄存并参与下一位运算。由于要完成两个 n 位数的加法运算需要进行 n 步的一位加法和 n 次的移位，速度太慢，现代计算机几乎不再采用。

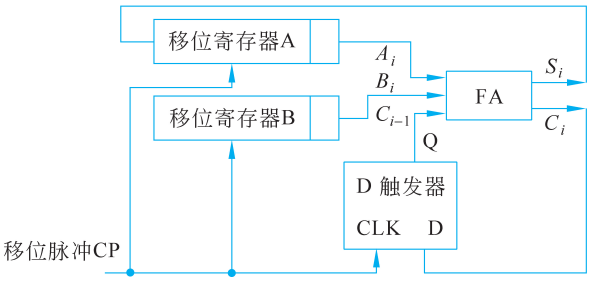


图 3-6 串行加法器框图

2. 并行加法器

并行加法器的全加器个数和操作数位数相同，能同时对所有位求和。下面讨论组成 n 位并行加法器的几种方法，主要讨论并行加法器采用的几种进位结构，以及不同的进位结构对运算速度的影响。在这些结构中，最基本的是串行进位和并行进位。在此基础上可将整个加法器分组、分级，对组内、组间、级间分别采用串行和并行进位。

1）进位公式分析

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

令

$$G_i = A_iB_i$$

$$P_i = A_i \oplus B_i$$

其中 G_i 为全加器第 i 位的进位产生函数（carry generate function），其含义是若本位两个输入均为 1，则向高位产生进位，与低位进位无关，所以又称本位进位；P_i 为全加器第 i 位

的进位传递函数(carry propagate function)，其含义是当 $P_i=1$ 时，若低位有进位，则本位将产生进位，也就说，$P_i=1$ 时，低位传送过来的进位能越过本位向更高位传送，P_i 又称进位传送条件，P_iC_{i-1} 称为传送进位。将 G_i、P_i 代入上式，得

$$C_i=G_i+P_iC_{i-1}$$

2）串行进位加法器

将 n 个全加器的进位位按照串行串联方法，从低到高逐位依次连接起来(低位全加器的进位输出连接到高位全加器的进位输入)就可以得到 n 位串行进位加法器。图 3-7 所示为四位串行进位加法器。

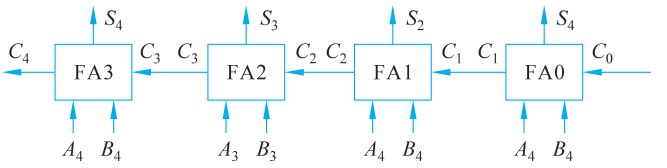


图 3-7　四位串行进位加法器

四位串行进位加法器

用这种方法组成的加法器，从最低位来的进位信号是一位一位逐位串行向高位传播的，所以又称为行波进位加法器，在行波进位加法器中，进位信号的逻辑表达式为

$$C_1=G_1+P_1C_0$$
$$C_2=G_2+P_2C_1$$
$$C_3=G_3+P_3C_2$$
$$\vdots$$
$$C_n=G_n+P_nC_{n-1}$$

后一级的进位直接依赖前一级，进位是逐级形成的，假设一级门的延迟时间是 1ty，最长进位延迟时间为(2n+1)ty，形成最后和的时间是 2$n$ty，与 n 成正比。可见，行波进位加法器的加法速度比较低。通常 CPU 工作的周期是以 ALU 的工作时间为依据，加法器的延迟时间越长，CPU 的工作速度就越低，因此，提高加法器的速度有重要的意义。

视频：先行进位加法器

3）先行进位加法器

提高加法器的运算速度的关键因素有两个：一个是减小门电路的延迟时间，门电路的延迟时间的减少受限于半导体集成电路的制造工艺和技术；另一个是电路结构造成的进位信号的延迟，高位的和必须等到低位的进位信号产生后才能完成。因此，改变加法器的电路结构、消除行波进位中的进位逐位串行传播、让各位进位独立地同时形成是提高运算速度有效方法。下面以四位加法器为例，将进位信号用 G_i、P_i 表示成以下形式：

$$C_1=G_1+P_1C_0$$
$$C_2=G_2+P_2C_1=G_2+P_2(G_1+P_1C_0)$$
$$C_3=G_3+P_3C_2=G_3+P_3(G_2+P_2(G_1+P_1C_0))$$
$$C_4=G_4+P_4C_3=G_4+P_4(G_3+P_3(G_2+P_2(G_1+P_1C_0)))$$

展开整理，得

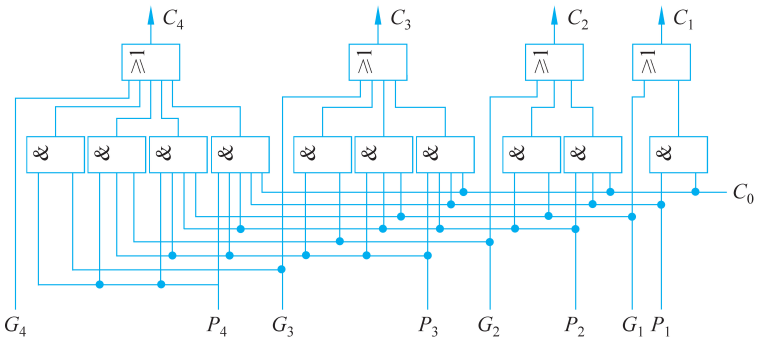
$$C_1 = G_1 + P_1C_0$$

$$C_2 = G_2 + P_2G_1 + P_2P_1C_0$$

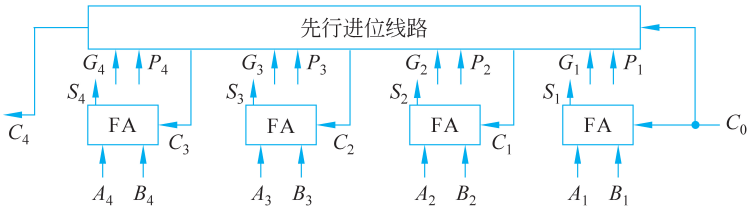
$$C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0$$

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

由以上公式可知，四个进位输出信号仅由进位产生函数 G_i、进位传递 P_i 和最低进位 C_0 决定，与各自低位进位位无关。图 3-8 所示为四位先行进位电路图和四位先行进位加法器。



(a) 先行进位产生电路



(b) 四位先行进位加法

图 3-8 四位先行进位加法器

输入信号 A_i、B_i、C_0 是同时提供的，这样，所有的进位延迟时间都是 3ty，因为进位信号同时产生，所以称为同时进位或并行进位(carry look ahead，CLA)，把这种进位信号由逻辑线路产生，再送去求和而构成的加法器，称为先行进位加法器。

理论上讲，这种先行进位加法器可以扩充到 n 位字长，但是，当加法器位数增加时，进位函数 C_{i+1} 会变得越来越复杂(n 位字长的加法器，最高进位位需要一个 $n+1$ 位输入的或门和 $n+1$ 位输入的与门电路实现)，这在早期的电路实现是比较困难的。通常的做法是采用分组的方法，在保证进位快速性的同时，减少电路的复杂性。如以四位先行进位加法器为一组，在组间分别采用串行进位或并行进位。

十六位行组间串行进位，组内并行进位加法器

4) 组间行波进位加法器

把四个先行进位加法器的组间按照如图 3-9 所示串联起

来，组成一个16位的行波进位加法器。在这种结构中，由于组间进位C_4、C_8、C_{12}、C_{16}仍然是串行产生的，最高进位的产生时间为$4\times(3ty)=12ty$。采用这种结构，在大大地缩短了进位延迟时间的同时，兼顾了电路设计的复杂性。如果还需要进一步提高速度，可以采用两级先行进位结构。这里不再讨论。

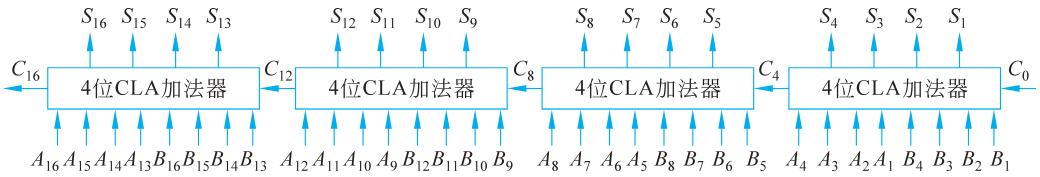


图3-9　16位行波进位加法器

视频：补码加减运算器＋移位操作

3. 实现补码加减运算的逻辑电路

图3-10所示为实现补码加减运算的逻辑电路。图中的FA代表加法器，A、B代表两个寄存器，其功能是存放参加运算的数据（补码）。A寄存器还用来保存运算结果。$A\rightarrow FA$、$B\rightarrow FA$、$\overline{B}\rightarrow FA$以及$FA\rightarrow A$分别表示送加法器两个输入端及运算结果回送A的控制信号。$1\rightarrow C_0$为加法器的最低进位信号。

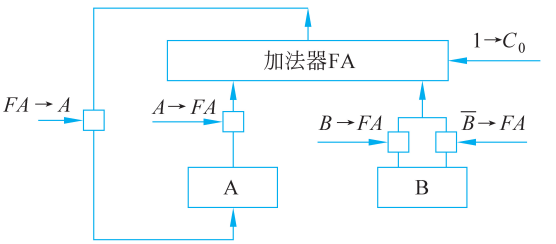


图3-10　实现补码加减运算的逻辑电路

补码加减运算的逻辑电路

假设参加运算的数已送到A、B寄存器，实现加法运算时，控制器产生$A\rightarrow FA$及$B\rightarrow FA$两个控制信号则将$[A]_{补}$和$[B]_{补}$送到加法器FA的两个输入端，FA完成$[A]_{补}+[B]_{补}$的加法过程，然后通过$FA\rightarrow A$控制信号将加法运算结果存入A寄存器。如果实现减法运算，控制器产生$A\rightarrow FA$及$\overline{B}\rightarrow FA$以及$1\rightarrow C_0$控制信号，则将$[A]_{补}$和$\overline{[B]_{补}}$送到加法器FA的两个输入端，1送$C_0$，FA完成$[A]_{补}+[-B]_{补}$的加法过程，然后通过$FA\rightarrow A$控制信号将减法运算结果存入A寄存器。

3.2.4　十进制加法器

在计算机中，处理十进制数有两种常用的方法：一种方法是先将输入的十进制数转换为二进制数，在计算机中进行二进制运算，再将运算结果转换为十进制数，这种方法适用于数据不太多而计算量大的场合；另一种方法是采用BCD码进行十进制运算，这种方法适用于数据量多而计算较简单的场合。目前，许多通用计算机都采用第二种处理方法。

3.3 定点数的乘除法运算

乘除法运算是计算机的基本运算之一。各类计算机实现乘除法运算的方法可能不同，但是不管采取什么方法，乘除法的基本原理是相同的，即通过多次相加和移位操作来实现乘除法运算；或直接采用硬件，用阵列乘法器与阵列除法器来实现。

3.3.1 移位操作

移位操作是计算机进行算术运算和逻辑运算不可缺少的基本操作，利用移位也可以进行寄存器中信息的串行传送。因此，几乎所有计算机的指令系统中都设置有各类移位操作指令。移位操作可以分为三大类：逻辑移位、循环移位和算术移位。图 3-11 所示为三种移位操作。

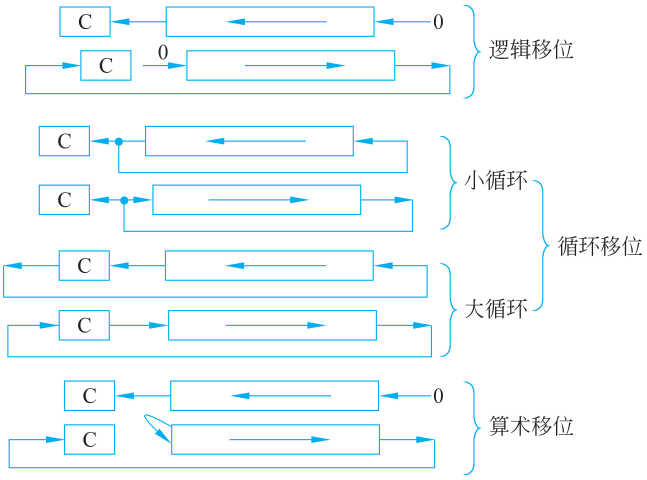


图 3-11 三种移位操作示意图

1. 逻辑移位

逻辑移位只有数码位置的变化而无数量的变化，左移低位补 0，右移高位补 0。在逻辑移位指令中，左移高位进进位位，右移低位进进位位。

2. 循环移位

循环移位是指寄存器两端触发器有移位通道，形成闭合的移位环路。分不带进位位循环（小循环，进位位不参与循环）和带进位位循环（大循环，进位位参与循环）。

3. 算术移位

算术移位是数的符号位不变而数量发生变化。左移一位相当于乘以 2，右移一位相当于除以 2。数的编码方法不同，算术移位的规则也不同。

正数：当机器数是正数时，原码、补码、反码的符号位为0，数值部分都与真值相等。所以不论左移、右移，移位后的空位均补0。

负数：负数的原码移位后，符号位不变，空位补0，负数的补码左移空位补0，右移空位补1，负数的反码移位后空位补1。


视频：乘法运算器1(加移位)

3.3.2 原码一位乘法

在计算机中，乘法一般采用原码进行。原码实现乘法运算非常简单，原因是乘积的符号位与积是分开考虑的。积的符号位为相乘两数的符号的“异或”值（同号得正，异号得负），其数值部分为两数的绝对值之积，即原码的数值部分之积。以定点小数为例，设

$$[X]_{原}=x_s x_1 x_2 \cdots x_n$$

$$[Y]_{原}=y_s y_1 y_2 \cdots y_n$$

$$[X]_{原}\times[Y]_{原}=x_s \oplus y_s (0.x_1 x_2 \cdots x_n)(0.y_1 y_2 \cdots y_n)$$

1. 运算方法

原码一位乘法是从手工算法演变而来的。下面举例说明。

【例3-7】 $X=0.1101$ $Y=0.1011$

$$\begin{array}{r} 0.1\ 1\ 0\ 1 \\ \times\quad 0.1\ 0\ 1\ 1 \\ \hline 1\ 1\ 0\ 1 \\ 1\ 1\ 0\ 1 \\ 0\ 0\ 0\ 0 \\ 1\ 1\ 0\ 1 \\ \hline XY=0.1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array}$$

在上述手工计算过程中，两个二进制数的乘法有以下特点。

(1) 用乘数 Y 的每一位依次乘以被乘数 X 得 $X\times y_i$。若 $y_i=0$，得0，若 $y_i=1$ 得 X。

(2) 把(1)中求的各项结果 $X\times y_i$ 在空间上向左错位排列，即逐次左移。可以表示为 $X\times y_i\times 2^{-i}$。

(3) 对(2)求得的结果求和 $\sum X\times y_i\times 2^{-i}$。这就是两个正数的积。

计算机在进行这一计算过程时应解决：

(1) ALU只有两个输入端，通常一次只能完成对两个数的求和操作，整个乘法操作要执行多次加法操作才能完成。

(2) 在手工计算时，各加数逐位左移，这样两个 n 位数相乘，就需要 $2n$ 位加法器。但每一次相加时，因为加数左移，最低一位被加数实际上是加0，即不加任何数。因此只

要将它移到另一个寄存器寄存起来而不需要用加法器。这样就可以将被加数右移代替加数左移，加法器的位数不会增加。

计算机在进行乘法运算时采用以下方法。

(1) 将乘数 Y 的一位乘以被乘数得 $X\times y_i$ 后，与前面所得到的部分结果累加获得新的部分积 P_i，这样只要设置一个寄存器保存部分积，初始化时该部分积寄存器为 0。减少了保存每次相乘结果 $X\times y_i$ 的开销。

(2) 在每次求得 $X\times y_i$ 后，不是将它左移与前一次部分积 P_{i-1} 相加，而是将部分积 P_{i-1} 右移一位与 $X\times y_i$ 相加。这是因为加法运算始终对部分积中的高 n 进行的。因此，只用 n 位的加法器(与一位进位位共 $n+1$ 位)就可以实现两个 n 位数的相乘。

(3) 对乘数中的“$y_i=1$”的位，执行部分积加被乘数绝对值的操作，得到新的部分积，然后将新的部分积右移一位。对乘数中的“$y_i=0$”的位，执行部分积加 0 并右移一位(也可以采用部分积直接右移，这样节省了部分积的生成时间)。以上过程的理论根据为

$$X\times Y=X\times(0.y_1y_2\cdots y_n)=X\times y_1\times 2^{-1}+X\times y_2\times 2^{-2}+\cdots+X\times y_n\times 2^{-n}$$
$$=2^{-1}\{2^{-1}[2^{-1}\cdots 2^{-1}(2^{-1}(0+X\times y_n)+X\times y_{n-1})+\cdots+X\times y_2]+X\times y_1\}$$

设 $P_0=0$，则

$$P_1=2^{-1}(P_0+X\times y_n)$$
$$P_2=2^{-1}(P_1+X\times y_{n-1})$$
$$\vdots$$
$$P_i=2^{-1}(P_{i-1}+X\times y_{n-(i-1)})$$
$$\vdots$$
$$P_{n-1}=2^{-1}(P_{n-2}+X\times y_2)$$
$$P_n=2^{-1}(P_{n-1}+X\times y_1)$$

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

乘法的数值部分通过多次累加和移位完成。每累加一次右移一位，所得结果为部分积，最后一次部分积即为乘积的数值部分。所以，欲求 $X\times Y$，需要设置三个寄存器 A、B、C。寄存器 A 用于保留部分积高位。乘法开始时，令其初值为 0，然后每乘一位乘数与前一次部分积相加并右移一位，得到一个新的部分积。寄存器 B 用于存放被乘数的绝对值。寄存器 C 用于存放乘数的绝对值，因为乘数每乘一位，此位代码就不再使用了，所以可以用乘数寄存器与部分积寄存器联合右移存放逐次增加的部分积低位。乘法结束时，A 中存放乘积的高位，C 中存放乘积的低位。

$X=0.1101$，$Y=0.1011$，计算机进行 $X\times Y$ 操作过程如下。

	部分积 A	乘数 C	操作说明
	0 0 0 0 0	1 0 1 1	$y_i=1$，部分积加被乘数绝对值
+	0 1 1 0 1		
	0 1 1 0 1		
→	0 0 1 1 0	1 1 0 1	部分积与乘数一起右移一位
+	0 1 1 0 1		$y_i=1$，部分积加被乘数绝对值
	1 0 0 1 1		
→	0 1 0 0 1	1 1 1 0	部分积与乘数一起右移一位
+	0 0 0 0 0		$y_i=0$，部分积加 0
	0 1 0 0 1		
→	0 0 1 0 0	1 1 1 1	部分积与乘数一起右移一位
+	0 1 1 0 1		$y_i=1$，部分积加被乘数绝对值
	1 0 0 0 1		
→	0 1 0 0 0	1 1 1 1	部分积与乘数一起右移一位

积的符号位=0⊕0=0，所以 $XY=0.1\ 0\ 0\ 0\ 1\ 1\ 1\ 1$

2. 原理框图

实现原码一位乘法的程序流程图和逻辑电路原理框图如图 3-12 和图 3-13 所示。图中，CNT 表示计数器，其初始值 CNT=n。

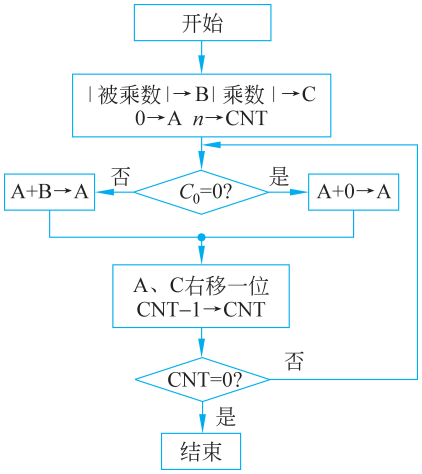


图 3-12 原码一位乘法流程图

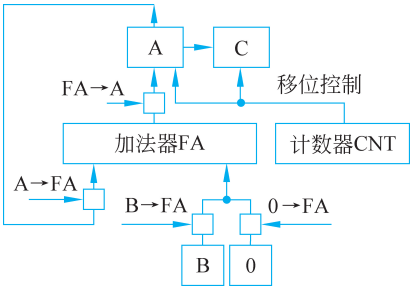


图 3-13 实现原码一位乘法运算的逻辑电路

乘法步骤：

(1) 初始化，被乘数→B，乘数→C，0→A，n→CNT。

(2) 若 $C_0=1$(C 的最末位，即 y_i)，A+B→A；否则 $C_0=0$ 时，A+0→A。

(3) A、C 联合右移一位。

(4) CNT−1→CNT，若 CNT≠0，按照(2)、(3)步循环，直至 CNT=0。

乘积为 $2n$ 位，高位在 A 中，低位在 C 中，符号位由异或门决定并冠在乘积最高位之前。

3.3.3 阵列乘法器


视频：乘法运算器2(阵列乘法器)

用加、移位的方法来实现乘法运算，对于两个 n 位数的乘法运算需要执行 n 次加和 n 次右移，随着数值位的增加，运算速度比较慢，高速乘法运算器都不采用这种方法。在前面讨论的用手算实现两个定点数绝对值相乘的执行过程中，在计算机内，可以用组合逻辑电路实现，图 3-14 所示为乘法器。

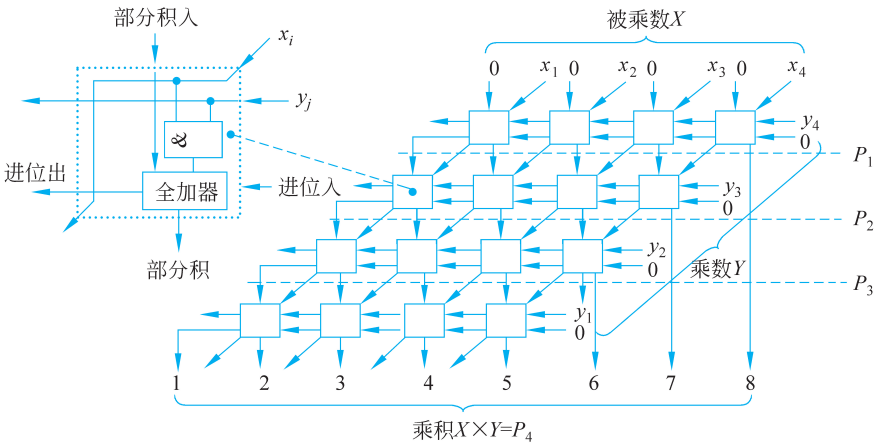


图 3-14 直接实现定点乘法的阵列乘积器

由于该乘法器呈现阵列结构的形式，故称为阵列乘法器(array multiplier)。图 3-14 中实现了 $X\times Y$ 的乘法运算，即

$$X\times Y=\sum(X\times y_j)2^j=\sum\left(\sum(x_i\times y_j)2^i\right)2^j$$

式中的一位乘积 $x_i\times y_j$ 可以用一个两输入端的与门实现。每一次加法操作用一个全加器实现。2^i 和 2^j 的因子所蕴含的移位是由全加器空间错位来实现的。与门和全加器的功能可以用一个单元组合起来，在图 3-14 中用一个方框表示。

在阵列乘法器中，每一行由乘数的每一数位 y_j 控制，即为 $x_i\times y_j\times 2^i$，而每一斜列则由被乘数的每一数位 x_i 控制，即为 $x_i\times y_j\times 2^j$。这种阵列乘法器的元件代价随 n 平方增加(n 是数据的位数)。

阵列乘法器的组织结构规则性强，标准化程度高。适合用超大规模集成电路实现，且能获得很高的运算速度。

3.3.4 定点除法运算


视频：原码除法运算1

1. 原码除法运算

除法运算比乘法运算要复杂，它也是由手算演变来的。

1) 手算过程

下面以 0.10011101÷0.1011 两个定点小数为例说明手算的一般过程，如图 3-15 所示。

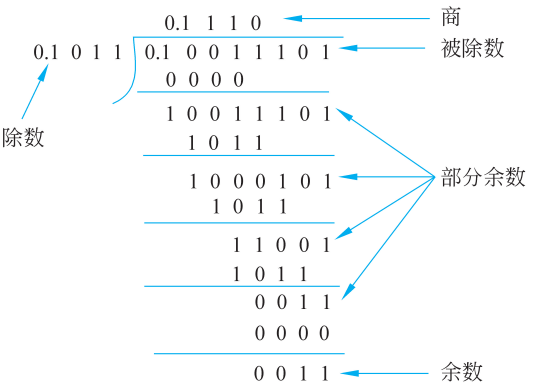


图 3-15　手工除法

（1）被除数与除数进行比较（试商），决定上商。若被除数小，上商为 0，否则上商为 1。并得到部分余数。

（2）将除数右移，再与上一步所得的部分余数比较，决定上商，并得到新的部分余数。

（3）重复执行第（2）步，直到得到的商的位数足够为止。

原码一位除法运算与原码一位乘法运算一样，符号位和数值位是分别处理的，商的符号位为相除两数的符号位“异或”（同号得正，异号得负）。商的数值位为两数绝对值之商。

2）为方便计算机实现采用的方法

（1）比较运算用减法（用补码加法）来实现，根据减法结果的符号位的正负来判断两数的大小，结果为正，上商为 1，结果为负，上商为 0。

（2）进行减法运算时，加法器中两数对齐是用部分余数左移实现，并与除数比较，以代替手算时的除数右移与部分余数比较。左移出界的部分余数的高位都是 0，对运算不会产生任何影响。

（3）在计算机中设置商值寄存器。每一次上商都写到最低位，然后将部分余数同商一起左移，腾空商值寄存器的最低位，以备上新的商。

由于采用部分余数减除数的方法比较二者大小，当减法结果为负，即上商为 0 时，破坏了部分余数。为了保证运算结果的正确，可以采用两种方法：第一种方法是将减去的余数再加回去，恢复原来的部分余数，这种方法称为恢复余数法；第二种方法是在这一步中多减去的除数在下一步运算时补回来，这种方法称为不恢复余数法，又称为加减交替法。

2. 恢复余数法

若已知两个定点小数 X 和 Y，$X=(0.x_1x_2\cdots x_n)$，$Y=(0.y_1y_2\cdots y_n)$，求 X/Y 的商和余数的方法如下。

（1）$R_1=X-Y$。若 $R_1<0$，则上为商 $q_0=0$，同时恢复余数 $R_1=R_1+Y$；否则，上商 $q_0=1$。说明商大于 1。根据计算机中的定点小数表示规定，超过了数的表示范围，做溢出处理。

(2) 若已求得第 i 次的部分余数 R_i，则第 $i+1$ 次的部分余数 $R_{i+1}=2R_i-Y$。若 $R_{i+1}<0$，则上商 $q_i=0$，同时恢复余数 $R_{i+1}=R_{i+1}+Y$；否则，上商 $q_i=1$。

(3) 不断循环(2)，直到求得所需要的商的位数($n+1$)。

其中，R_i 为部分余数(开始时为被除数 X)。

【例 3-8】 已知 $X=0.1011$，$Y=-0.1101$。用恢复余数法求 $X\div Y$。

因为原码除法是将符号位和商分开考虑的，商是直接用两个数字的绝对值相除，试商时，采用部分余数减除数是采用补码相加来实现的。先求两个数绝对值的补码。$|X|=0.1011$，$|Y|=0.1101$，$[|X|]_{补}=00.1011$，$[|Y|]_{补}=00.1101$，$[-|Y|]_{补}=11.0011$。采用恢复余数，$-|Y|$ 可以用 $+[-|Y|]_{补}$ 来实现。采用双符号位(防止左移时部分余数会改变符号位产生溢出)。在计算机中隐含了小数点，运算过程如图 3-16 所示。

	余数 R	商 Q	操作说明
	0 0 1 0 1 1	0 0 0 0 0	开始 $R_0=X$
+	1 1 0 0 1 1		$R_1=X-Y$
	1 1 1 1 1 0	0 0 0 0 0	$R_1<0$，则上商 $q_0=0$
+	0 0 1 1 0 1		恢复余数 $R_1=R_1+Y$
	0 0 1 0 1 1		得 R_1
←	0 1 0 1 1 0	0 0 0 0 0	部分余数同商一起左移 $2R_1$
+	1 1 0 0 1 1		$2R_1-Y$
	0 0 1 0 0 1	0 0 0 0 1	$R_2>0$，则上商 $q_1=1$
←	0 1 0 0 1 0	0 0 0 1 0	部分余数同商一起左移 $2R_2$
+	1 1 0 0 1 1		$2R_2-Y$
	0 0 0 1 0 1	0 0 0 1 1	$R_3>0$，则上商 $q_2=1$
←	0 0 1 0 1 0	0 0 1 1 0	部分余数同商一起左移 $2R_3$
+	1 1 0 0 1 1		$2R_3-Y$
	1 1 1 1 0 1	0 0 1 1 0	$R_4<0$，则上商 $q_3=0$
+	0 0 1 1 0 1		恢复余数 $R_4=R_4+Y$
	0 0 1 0 1 0		得 R_4
←	0 1 0 1 0 0	0 1 1 0 0	部分余数同商一起左移 $2R_4$
+	1 1 0 0 1 1		$2R_4-Y$
	0 0 0 1 1 1	0 1 1 0 1	$R_5>0$，则上商 $q_4=1$

图 3-16 原码一位恢复余数法

商的符号=0⊕1=1。所以 $X/Y=-0.1101$(商)，余数为 0.0111×2^{-4}。

实现两个正数的除法的逻辑电路如图 3-17 所示。图 3-18 所示为恢复余数算法的流程图。其中寄存器、加法器、计数器的功能如下。

寄存器 R：初始值为被除数，除法运算过程中存放部分余数，除法结束时存放余数。

寄存器 Y：存放除数。

寄存器 Q：初始值为 0，除法运算过程中，与寄存器 R 一起左移，空出的最低位不断上商。除法结束时，存放商的数值位。

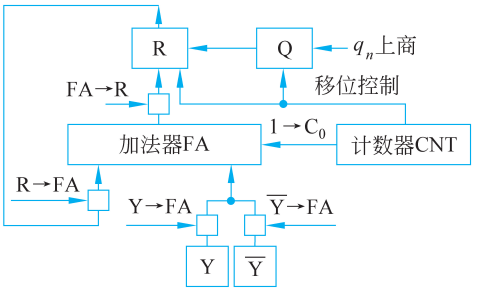


图 3-17　实现原码一位除法运算的逻辑电路

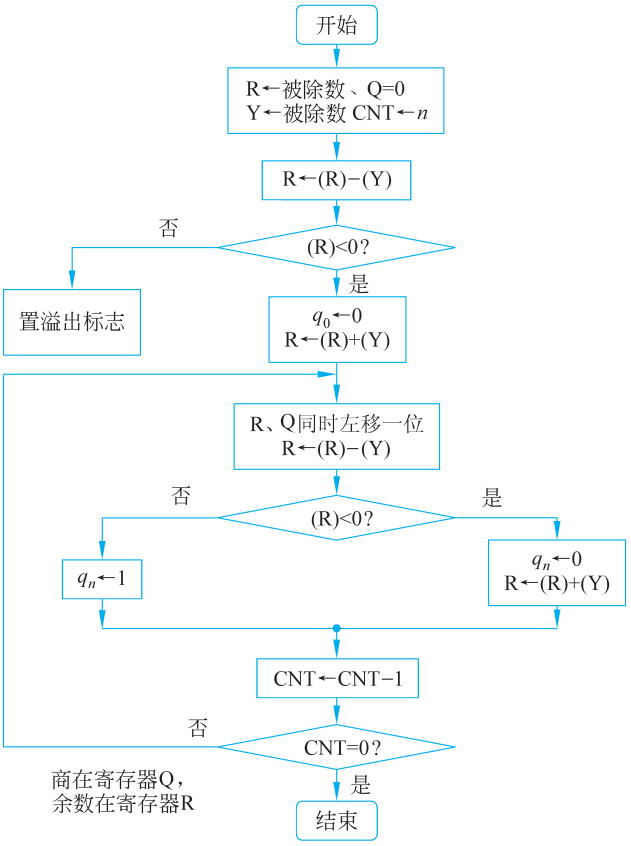


图 3-18　恢复余数算法的流程图

加法器 FA：用补码相加的方法实现部分余数与除数的比较。

计数器 CNT：控制除法的循环次数。初始值为数据的位数，每循环一次，自动减 1。当 CNT＝0 时，控制除法运算的结束。

视频：除法运算器 2(不恢复余数法)

3. 不恢复余数法(加减交替法)

在恢复余数的算法中，当部分余数为负时，要恢复余数，这时要多做一次加法运算操

作(+Y),这样不仅降低了运算速度,而且还使控制线路变得复杂。因此,在计算机中,很少采用恢复余数的算法,而普遍采用加减交替法。实现原理为:

在恢复余数的除法中,第 i 次的余数 R_i,若 $R_i \geqslant 0$,则第 $i+1$ 次的部分余数 $R_{i+1}=2R_i-Y$。

若 $R_i<0$(记位 R_i'),则上商为 $q_i=0$,同时恢复余数 $R_1=R_i'+Y$,则 $R_{i+1}=2R_i-Y=2(R_i'+Y)-Y=2R_i'+Y$。所以当第 i 次的部分余数为负时,可以省略恢复余数这一步操作,而在下一次求部分余数时将减除数操作改为加除数操作,这样就节省了一次恢复余数的运算。对两个正数采用不恢复余数的算法的一般步骤归纳于下。

(1) $R_1=X-Y$。若 $R_1<0$,则上商为 $q_0=0$,同时恢复余数 $R_1=R_1+Y$(实际操作时,是在下一步操作中执行 $R_2=2R_1+Y$)。否则,上商$q_0=1$。说明商大于 0。根据计算机的定点小数表示规定,超过了数的表示范围,做溢出处理。

(2) 如果已求得第 i 次的部分余数R_i,若 $R_i<0$,则上商为 $q_{i-1}=0$, $R_{i+1}=2R_i+Y$。上次运算中多减的 Y 在这次运算中补回来了,否则,上商 $q_{i-1}=1$,$R_{i+1}=2R_i-Y$。

(3) 不断循环(2),直到求得所需要的商的位数($n+1$)。

注意,在运算到最后一步时,如果部分余数是负的,要进行一次恢复余数。

【例 3-9】 已知 $X=0.1011$,$Y=-0.1101$。用不恢复余数法求 X/Y。

$|X|=0.1011$,$|Y|=0.1101$, $[|X|]_{补}=00.1011$,$[|Y|]_{补}=00.1101$,$[-|Y|]_{补}=11.0011$。采用不恢复余数。$-Y$ 可以用$+[-|Y|]_{补}$来实现。采用双符号位(防止左移时部分余数会改变符号位产生溢出)。运算过程如图 3-19 所示。

	余数 R	商 Q	操作说明
	00 1011	00000	开始 $R_0=X$
+	11 0011		$R_1=X-Y$
	11 1110	00000	$R_1<0$,则上商 $q_0=0$
←	11 1100	00000	部分余数和商一起左移 $2R_1$
+	00 1101		$2R_1+Y$
	00 1001	0000$\underline{1}$	$R_2>0$,则上商 $q_1=1$
←	01 0010	000$\underline{1}$0	部分余数和商一起左移 $2R_2$
+	11 0011		$2R_2-Y$
	00 0101	000$\underline{11}$	$R_3>0$,则上商 $q_2=1$
←	00 1010	00110	部分余数同商一起左移 $2R_3$
+	11 0011		$2R_3-Y$
	11 1101	0$\underline{1}\underline{1}\underline{1}\underline{0}$	$R_4<0$,则上商 $q_3=0$
←	11 1010	$\underline{0}\underline{1}\underline{1}\underline{0}\underline{0}$	部分余数同商一起左移 $2R_4$
+	00 1101		$2R_4+Y$
	00 0111	$\underline{0}\underline{1}\underline{1}\underline{0}\underline{1}$	$R_5>0$,则上商 $q_4=1$

图 3-19 原码一位不恢复余数法

商的符号=0⊕1=1,所以 $X/Y=-0.1101$(商),余数为 0.0111×2^{-4}。

从上面的计算过程可以看出，在不恢复余数除法运算中，每一位商的计算或是减法或是加法，而不是两者都要。

视频：逻辑运算＋集成的算术/逻辑运算器

3.4 逻辑运算

在计算机中，对非数值数据的处理主要涉及逻辑运算，逻辑运算主要包括逻辑与、或、非和异或操作。逻辑运算的结果只与本位逻辑数有关，不受其他位的影响。

3.4.1 逻辑与

逻辑与的运算符为∧（或直接用·表示）；运算关系为1∧1＝1，1∧0＝0，0∧1＝0，0∧0＝0；逻辑与的运算口诀为“全1得1，见0得0”。所以逻辑与运算又称为逻辑乘。逻辑与运算器可以直接用与门电路实现，如图3-20所示。

3.4.2 逻辑或

逻辑或的运算符为∨（或直接用＋表示）；运算关系为1∨1＝1，1∨0＝1，0∨1＝1，0∨0＝0；逻辑或的运算口诀为“见1得1，全0得0”。所以逻辑或运算又称为逻辑加。逻辑或运算器可以直接用或门电路实现，如图3-21所示。

图3-20 逻辑与电路

图3-21 逻辑或电路

3.4.3 逻辑异或

逻辑异或的运算符为⊕；运算关系为1⊕1＝0，1⊕0＝1，0⊕1＝1，0⊕0＝0；逻辑异或的运算口诀为“相同得0，不同得1”。逻辑异或运算器可以直接用异或门电路实现，如图3-22所示。

3.4.4 逻辑非

逻辑非的运算符为¯，只有一个操作数，运算关系为$\overline{1}=0$，$\overline{0}=1$。所以逻辑非运算又被称为取反运算。逻辑非运算器可以直接用非门电路（反向器）实现，如图3-23所示。

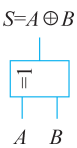


图3-22 逻辑异或电路

图3-23 反向器逻辑电路