第 3 章　运算方法与运算器

在计算机中，运算器用于数值运算及加工处理数据，它是由 CPU 中的算术逻辑单元、通用寄存器等部件构成。运算器的结构取决于指令系统、数据的表示方法、运算方法及所选用的硬件。本章的主要内容就是讨论数值运算的方法及具体实现。

3.1　定点数运算

如上一章所述，计算机中常用定点数或浮点数这两种形式来表示数值。它们的运算方法也有所不同，下面将分别予以讨论。本节将描述定点数的四则运算的法则及实现方法。

3.1.1　加减运算

1. 加减运算方法

在前面第 2 章中，已经描述了数值的编码，它们可以用原码、反码、补码、移码等形式表示。原则上讲，对数值的加减运算可以用上述任何一种编码来实现，但是，用得最多最普遍的是采用补码来实现加减法。下面将说明补码加减运算法则及有关问题。

补码加减运算过程中，参加运算的操作数及运算结果均用补码表示。

1）补码加法

补码加法的运算法则为：

$$[X+Y]_{补}=[X]_{补}+[Y]_{补} \tag{3-1}$$

由式(3-1)可以看到，两数和的补码就等于两数补码之和。利用补码求两数之和就变得十分方便。

例 3.1　若两个定点整数 63 和 35，利用补码加法求 63＋35＝？

解：根据题意，用 8 位二进制补码表示 63 和 35：

$$[63]_{补}=00111111$$

$$[35]_{补}=00100011$$

则$[63+35]_{补}=01100010$

例 3.2　若两个定点整数－63 和－35，利用补码加法求－63＋(－35)＝？

解：根据题意，用 8 位二进制补码表示－63 和－35：

$$[-63]_{补}=11000001$$

$$[-35]_{补}=11011101$$

则$[-63+(-35)]_{补}=10011110$

2）补码减法

在数值的补码表示法中，我们注意到，对一个正数求补——即对该数包括符号位在内各位取反再加 1，即可得到该数的负数；若对该负数再求补又可得到原来的正数。也就是说$[[X]_{补}]_{求补}=[-X]_{补}$；$[[-X]_{补}]_{求补}=[X]_{补}$。

补码减法的运算法则为：

$$[X-Y]_{补}=[X]_{补}+[-Y]_{补}=[X]_{补}+[[Y]_{补}]_{求补} \tag{3-2}$$

式(3-2)给大家一个最重要的启示就是用补码做减法时，可用加法来实现；被减数减去减数可化做被减数加上减数求补来完成。这一特征就使得 CPU 的设计者不必在 CPU 中设置减法器，只要设置一个加法器，就既能实现加法运算又能完成减法运算。这也是计算机中普遍地采用补码来实现加减法的原由。

例 3.3 若两个定点整数 63 和 35，利用补码减法求 63－35＝？

解：根据题意，用 8 位二进制补码表示 63 和 35：

$$[63]_{补}=00111111$$

$$[35]_{补}=00100011$$

而$[63-35]_{补}=[63]_{补}+[-35]_{补}$。

同时，$[-35]_{补}=[[35]_{补}]_{求补}=11011101$，从而求出：

$$\begin{array}{r} 00111111 \\ +\ 11011101 \\ \hline 1\ 00011100 \end{array}$$

得到$[63-35]_{补}=00011100$。由于是 8 位二进制运算，在上面相加过程中有一进位 1 而丢弃不用，所得的结果仍是正确的。

综上所述，补码加减运算的规则是如下。

(1) 参加运算的操作数用补码表示；

(2) 符号位参加运算；

(3) 若进行相加，则两个数的补码直接相加；若进行相减运算，则将减数连同符号位一起变反加 1 后与被减数相加；

(4) 运算结果用补码表示。

2. 溢出判断

1) 溢出的概念

从上面的描述中已经明确，只用加法器就可以实现补码加法和补码减法。但在运算的过程中有可能会产生溢出。为了说明溢出的概念及其判别方法，首先来看下面的例子。

例 3.4 若两个定点整数 63 和 85，利用补码加法求 63＋85＝？

解：根据题意，若用 8 位二进制补码表示 63 和 85：

$$[63]_{补}=00111111$$

$$[85]_{补}=01010101$$

$$\begin{array}{r} 00111111 \\ +\ 01010101 \\ \hline 10010100 \end{array}$$

由上面的计算可以看到，两个正数(63 和 85)相加的结果变成一个负数(符号位为 1)，这显然是荒谬的。出现这种错误结果是由于在相加的过程中产生了溢出。原因就在于运算的结果已超出所规定的数值范围。如上所述，规定用 8 位二进制补码来表示带符号的整数，它所能表示的数值范围是＋127～－128。如果运算的结果超出了＋127 则称为上溢出；若超出－128 称为下溢出。

一旦确定了运算的字长和数据的表示方法后，数据的范围也就确定了。只要运算结果

超出所能表示的数据范围，就会发生溢出。发生溢出时，运算结果肯定是错误的。只要发现运算结果产生溢出，就必须采取措施防止溢出发生。最简单有效的方法就是增加补码的二进制编码长度。上面的例子中，只要将数值编码位数增加，如采用16位编码，就一定能防止溢出发生。

值得注意的是：只有当两个同符号的数相加（或者是相异符号数相减）时，运算结果才有可能溢出。而在异符号的数相加（或者是同符号数相减）时，永远不会产生溢出。下面再举两例说明。

例 3.5 设正整数 X＝＋1000001，Y＝ ＋1000011，若用 8 位补码表示，则$[X]_{补}$＝01000001，$[Y]_{补}$＝01000011，求$[X+Y]_{补}$。

解：计算$[X]_{补}+[Y]_{补}$

$$\begin{array}{rr} & 0\ 1000001 \\ + & 0\ 1000011 \\ \hline & 1\ 0000100 \end{array}$$

两个正数相加的结果为一个负数，结果显然是错误的。产生错误的原因就是溢出。

例 3.6 设负整数 X＝－1111000，Y＝－10010，若用 8 位补码表示，则$[X]_{补}$＝10001000，$[Y]_{补}$＝11101110，求$[X+Y]_{补}$。

解：计算$[X]_{补}+[Y]_{补}$

$$\begin{array}{rr} & 1\ 0001000 \\ + & 1\ 1101110 \\ \hline & 0\ 1110110 \end{array}$$

两个负数相加，结果为一个正数，显然也是错误的。

在上面的举例中，所用的都是定点整数。上面的概念和结论对于定点纯小数是完全一样的。但需注意所规的定点纯小数的数值范围。

2）溢出的判定

常用的溢出检测机制主要有进位判决法和双符号位判决法等如下几种方法。

（1）双符号位判决法

若采用两位表示符号，即 00 表示正号、11 表示负号，一旦发生溢出，则两个符号位就一定不一致，利用判别两个符号位是否一致便可以判定是否发生了溢出。

若运算结果两符号分别用 S_2S_1 表示，则判别溢出的逻辑表示式为：

$$VF = S_2 \oplus S_1 \tag{3-3}$$

例 3.7 设两正整数 X＝＋1000001，Y＝＋1000011，若用双符号位的 8 位补码表示，则$[X]_{补}$＝00 1000001，$[Y]_{补}$＝00 1000011，求$[X+Y]_{补}$。

解：计算$[X]_{补}+[Y]_{补}$

$$\begin{array}{rr} & 00\ 1000001 \\ + & 00\ 1000011 \\ \hline & 01\ 0000100 \end{array}$$

式中，由于结果的 S_2 和 S_1 不一致，$VF=S_2 \oplus S_1=1$，溢出发生。

（2）进位判决法

若 C_{n-1} 为最高数值位向最高位（符号位）的进位，C_n 表示符号位的进位，则判别溢出的逻辑表示式为：

$$VF = C_{n-1} \oplus C_n \tag{3-4}$$

溢出判定表如表 3.1 所示。

表 3.1　溢出判定表

C_{n-1}	C_n	$C_{n-1} \oplus C_n$	C_{n-1}	C_n	$C_{n-1} \oplus C_n$
0	0	0	1	0	1
0	1	1	1	1	0

在例 3.6 的运算过程中，C_{n-1} 为 0 而 C_n 为 1，故 $C_{n-1} \oplus C_n = 1$，表示运算结果有溢出。

(3) 根据运算结果的符号位和进位标志判别

该方法适用于两同号数求和或异号数求差时判别溢出。溢出的逻辑表达式为：

$$VF = SF \oplus CF \tag{3-5}$$

式中 SF 和 CF 分别是运算结果的符号标志和进位标志。

在使用式(3-5)判别溢出时，一定要注意该判别方法的适用条件，否则，将会出错。

(4) 根据运算前后的符号位进行判别

若用 X_s、Y_s、Z_s 分别表示两个操作数及运算结果的符号位，当两同号数求和或异号数求差时，就有可能发生溢出。溢出是否发生可根据运算前后的符号位进行判别，其逻辑表达式为：

$$VF = X_s \cdot Y_s \cdot \overline{Z}_s + \overline{X}_s \cdot \overline{Y}_s \cdot Z_s \tag{3-6}$$

在 CPU 中，进行定点算术运算是否发生溢出，通常是由 CPU 中的硬件逻辑电路进行检测。一旦溢出发生，则会在 CPU 中的标志寄存器中建立溢出标志，或者产生溢出中断。

3. 一位全加器的实现

设一位全加器的输入分别为 X_i 和 Y_i，低一位对该位的进位为 C_i。全加器的结果和向高一位的进位分别用 Z_i 和 C_{i+1} 表示。则一位全加器所实现的逻辑表达式如下。

$$Z_i = X_i \oplus Y_i \oplus C_i \tag{3-7}$$

$$C_{i+1} = (X_i \cdot Y_i) + (X_i + Y_i) \cdot C_i \tag{3-8}$$

实现上述逻辑功能的一位全加器的逻辑电路及其框图分别如图 3.1(a)和(b)所示。

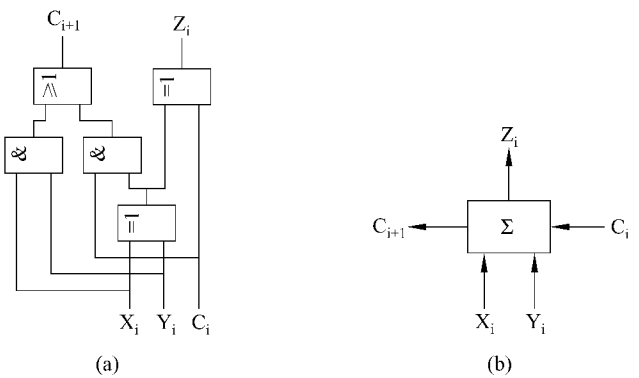

图 3.1　一位全加器逻辑及其框图

若令 $G_i = X_i \cdot Y_i$，$P_i = X_i + Y_i$，则式(3-8)可写为：

$$C_{i+1} = G_i + P_i \cdot C_i \tag{3-9}$$

人们将 G_i 称为生成进位的产生函数，而将 P_i 称为进位的传递函数。

4. n 位加法器的实现

1) 行波进位加法器

利用上述 n 个全加器串在一起工作，可以构成 n 位加法器。同时，前面已经提到，补码加减运算，用加法器便可实现。图 3.2 所给出的就是用 n 个一位全加器及门电路构成 n 位补码加法/减法器的框图。

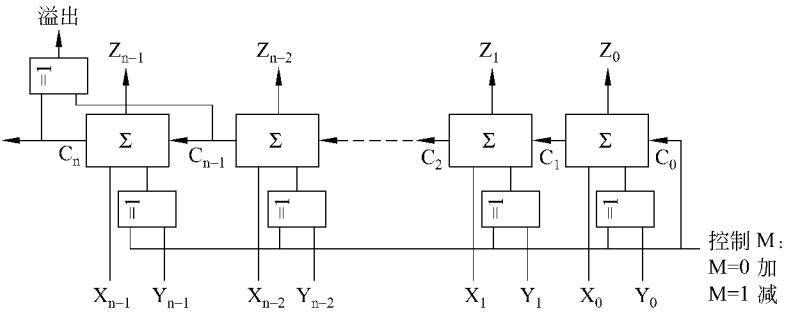

图 3.2　行波进位的 n 位加法/减法器

根据图 3.2，我们注意到：

① 在此加法器中，进位是逐位形成的。如图 3.1 所示，进位的产生是由三级门形成的，是需要花时间的。因为，任何一级门必定有延时。而图 3.2 所示的 n 位加法器中，高位运算必须等低运算的进位产生后方能进行。进位是由低到高逐位产生，一位接一位向高位传递。故这种加法器通常称为行波进位加法器。若一位全加器的进位延时为 Δt，则上述 n 位加法器的延时就是 $n\Delta t$。也就是说完成 n 位加法需要 $n\Delta t$。应当明确，设计运算器(或 CPU)所追求的最重要的指标之一是速度。显然，行波进位加法器的速度是难以令人满意的。

② 在图 3.2 中，利用增加异或门和控制信号 M：当 M 为 0 时，实现加法器的功能；当 M 为 1 时，可将操作数 Y 的各位其反并在 C_0(最低位)上加上 M(此时是 1)，从而实现求补功能。然后再做加法——完成减法的功能。

③ 图 3.2 中，利用异或门完成式(3-4)所示的溢出判别功能。该异或门的输入刚好是 C_{n-1} 和 C_n。

2) 先行进位加法器

如上所述，行波进位加法器尽管简单，但其致命的缺点就在于当加法器的位数增加时，由于进位而造成的加法速度的降低使人不能忍受。是否有加快进位速度的方法呢？答案是肯定的。

首先，分析式(3-9)，其表示式为：$C_{i+1} = G_i + P_i \cdot C_i$。从式中可知，只要有输入 X_i 和 Y_i 就能求出 G_i 和 P_i，在已知输入 C_i 的情况下，便可以获得 C_{i+1}。

那么，在有输入 X_{i+1} 和 Y_{i+1} 和 C_{i+1} 的情况下，便可以获得 C_{i+2}。以此类推，在输入存在的情况下，便可以求出 C_{i+3}、C_{i+4}……下面仅将 4 个进位的产生逻辑表示式写在下面：

$$C_{i+1} = G_i + P_i C_i \tag{3-10}$$

$$C_{i+2} = G_{i+1} + P_{i+1} C_{i+1} = G_{i+1} + P_{i+1} G_i + P_{i+1} P_i C_i \tag{3-11}$$

$$C_{i+3} = G_{i+2} + P_{i+2} C_{i+2} = G_{i+2} + P_{i+2} G_{i+1} + P_{i+2} P_{i+1} G_i + P_{i+2} P_{i+1} P_i C_i \tag{3-12}$$

$$C_{i+4} = G_{i+3} + P_{i+3} C_{i+3} = G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i + P_{i+3} P_{i+2} P_{i+1} P_i C_i = G^*_{i+3} + P^*_{i+3} C_i \tag{3-13}$$

式(3-13)中，

$$G^*_{i+3} = G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i$$

$$P^*_{i+3} = P_{i+3} P_{i+2} P_{i+1} P_i$$

由上面的逻辑表达式(3-10)～式(3-13)可以看到，利用输入信号 X_i、X_{i+1}、X_{i+2}、X_{i+3} 和 Y_i、Y_{i+1}、Y_{i+2}、Y_{i+3} 以及 C_i，通过与或逻辑电路的组合就可以同时将上面 4 个信号产生出来。然后，将这些先行形成的进位信号并行地加到加法器上，则加法器就不必等待进位产生，从而大大地提高了加法器的速度。也就是说在进行加法运算之前，各位加法器所需要的进位已经产生出来。这将加快加法运算的速度，这就是先行进位的来由。

根据上面的分析，具体用与或组合逻辑构成的 4 位先行进位产生电路如图 3.3 所示。

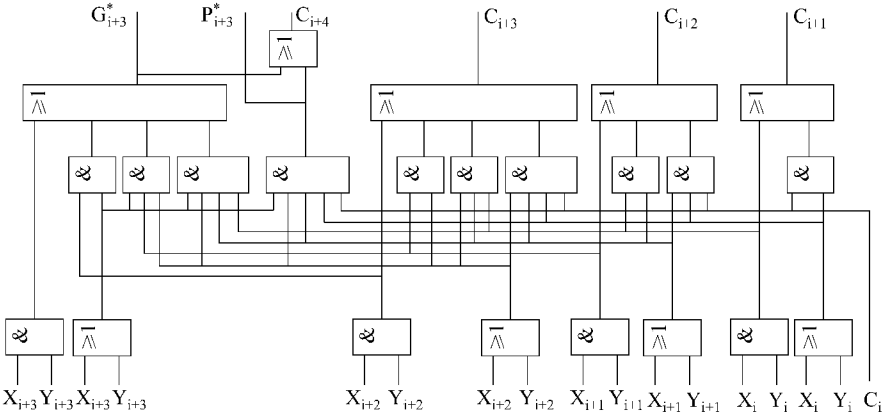

图 3.3　4 位先行进位链电路

由图 3.3 的逻辑图可以看到，图 3.3 的下方是输入信号 X_i、X_{i+1}、X_{i+2}、X_{i+3} 和 Y_i、Y_{i+1}、Y_{i+2}、Y_{i+3} 以及 C_i。这些信号经过三级门便可以得到所有 4 位加法器所需要的进位信号 C_{i+1}、C_{i+2}、C_{i+3}、C_{i+4}。三级门的延时要比多级行波进位小得多。

以上我们分析了 4 位先行进位的产生电路。显然，随着位数的增加，这种电路会越来越复杂。因此，在设计加法器时会将多位加法分组，例如以 4 位为一组。将各组串联在一起，即组内采用先行进位方式而组与组之间采用行波进位。这时还会用到前面已经出现的 G^*_{i+3} 和 P^*_{i+3} 信号。有关此问题的细节将在后面讨论。

5. BCD 数加法器

1）概述

在人们的日常生活中更多的是使用十进制数，而计算机的工作是用二进制数。为了将它们联系在一起，计算机中经常采用二进制编码表示的十进制数，即 BCD 数。

BCD 数规定用 4 位二进制编码表示一位十进制数。4 位二进制编码共有从 0000 到

1111 十六种状态，可以取其中十种状态来表示十进制数的 0 到 9 十个数字。在取十种状态时，可用不同的权值、与十进制真值多 3 或相邻十进制数的编码仅改变一位等方法来定义多种形式的 BCD 码。表 3.2 列出了常用的几种 BCD 码。

表 3.2　几种常用的 BCD 码

十 进 制 数	8421 码	2421 码	余 3 码	格雷码
0	0000	0000	0011	0000
1	0001	0001	0100	0001
2	0010	0010	0101	0011
3	0011	0011	0110	0010
4	0100	0100	0111	0110
5	0101	0101	1000	1110
6	0110	1100	1001	1010
7	0111	1101	1010	1000
8	1000	1110	1011	1100
9	1001	1111	1100	0100

2）8421 BCD 码

(1) 定义

在上述几种 BCD 码中，计算机里应用最广泛的是 8421 BCD 码。8421 是指在表示十进制数的二进制编码各位的权值，从最高位开始，各位的权值依次为 8、4、2、1。知道了权值的这种规定，则很容易从这种 BCD 码得到其表示的十进制数。

从表 3.2 可以看到，8421 BCD 码只利用了 4 位二进制编码的 0000 到 1001 这十种来表示十进制数的 0 到 9。剩余的 6 种：1010，1011，1100，1101，1110，1111 对用于表示十进制数来说是非法的。一旦在定义的 BCD 运算中出现这 6 种编码，结果一定是错误的。

在使用 BCD 码时，若用一个字节(八位二进制数)表示两位 BCD 数，即高 4 位表示一位 BCD 数，低 4 位表示一位 BCD 数。则此字节所表示的数称为压缩 BCD 数；若一个字节只表示一位 BCD 数，即高 4 位全为 0，只用低 4 位表示一位 BCD 数，则此字节所表示的数称为非压缩 BCD 数。

(2) 加法运算

8421 BCD 码加法运算也可以用多位全加器实现，在运算过程中有可能产生上面提到的错误结果。为了说明这一问题，请看下面的举例。

例 3.8　a. 计算压缩 BCD 数 46＋32＝？

b. 计算压缩 BCD 数 46＋67＝？

解：两题分别相加如下：

a：
$$\begin{array}{rr} & 0100\ 0110 \\ + & 0011\ 0010 \\ \hline & 0111\ 1000 \end{array}$$

b：
$$\begin{array}{rr} & 0100\ 0110 \\ + & 0110\ 0111 \\ \hline & 1010\ 1101 \end{array}$$

由该例可以看到，在 a 的情况下，两个 BCD 数相加结果是正确的。在 b 的情况下，两个 BCD 数相加结果是错误的。

(3) 校正

为了保证 BCD 数加法运算结果的正确，就必须进行校正。对于一个字节的压缩 BCD

数加法进行校正的法则是：

① 运算中低 4 位相加的结果＞9 或有由 bit3 向 bit4 的进位，则结果加 06H；

② 运算中高 4 位相加(包括由 bit3 向 bit4 的进位)的结果＞9 或有由 bit7 向更高位的进位，则结果加 60H，同时进位为 1 且应看做是相加结果的最高位；

③ 若高低 4 位均不满足上述条件，如例 3.8 中的 a 式，则在校正时加 00H；若同时满足 ab 两项条件，则在校正时结果加 66H。

以上讨论的是一个字节的压缩 BCD 数加法进行校正的法则。其他运算(减法、乘法、除法)同样需要校正，才能保证结果总是正确。同样，非压缩 BCD 的运算也需要校正。上面讨论的是 8421 BCD 码的校正问题，而其他类型的 BCD 码也同样会存在类似的题。所有这一切此处不再说明。

在设计 CPU 的时候，会在其指令系统中设置一些 BCD 数各种运算的校正指令，CPU 执行指令时会检测上述法则，根据不同的状态在结果上加上应加的数值。

8421 BCD 加法器及其校正也可以用硬件实现，图 3.4(a)所示的就是带有硬件校正电路的一位 8421 BCD 加法器。图 3.4(b)便是一位 BCD 加法器的框图。

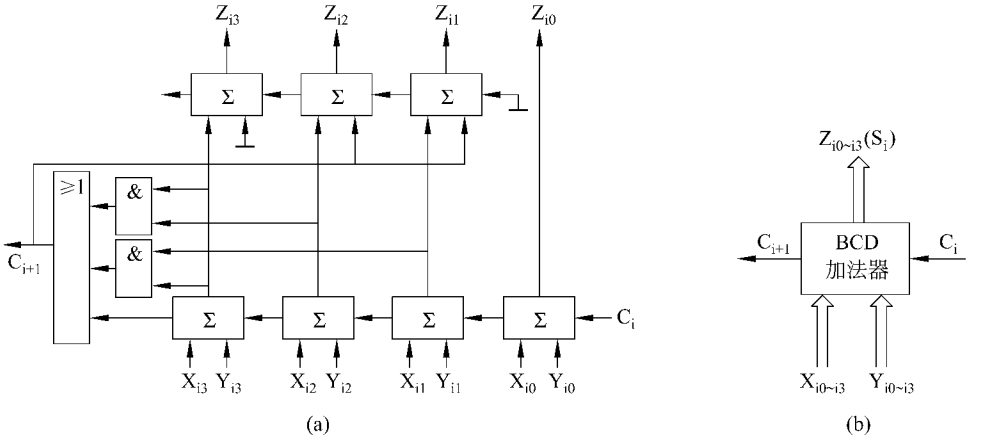

图 3.4　一位 8421 BCD 加法器

同样，将多个一位 BCD 加法器串联起来，构成多位行波进位的 BCD 加法器，其框图如图 3.5 所示。

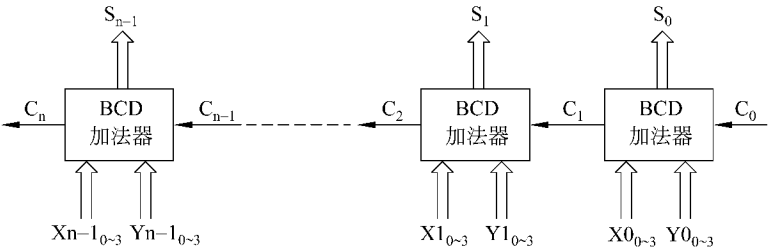

图 3.5　n 位行波进位 BCD 加法器框图

有关其他类型 BCD 码运算器的硬件实现，这里不再说明。

6. 移码加减运算

1）运算法则

关于移码的定义及其特点，在第 2 章中已经做了介绍。由于移码多用在浮点数中表示阶码，因此，在这里仅就定点整数移码的加减运算加以说明。

定点整数移码的加减运算的法则是：

① 对两移码求和差时，首先对该两移码求和差；② 然后，对结果进行修正——将结果的符号取反。这样，就可以得到正确的结果。下面举例说明。

例 3.9 用 8 位移码表示十进制数 57 和－35。并且求移码的和与差。

解：十进制数的 57 和－35 用移码表示如下：

$$[57]_{移}=10111001$$

$$[-35]_{移}=01011101$$

求两者之和：

① 首先求$[57]_{移}+[-35]_{移}=10111001+01011101=00010110$；

② 再将符号位取反，从而得到$[57+(-35)]_{移}=10010110$。

求两者之差：

① 首先求$[57]_{移}-[-35]_{移}=[57]_{移}+[[-35]_{移}]_{补}=10111001+10100011=01011100$；

② 再将符号位取反，从而得到$[57-(-35)]_{移}=11011100$。

2）移码运算应注意的问题

① 对移码运算的结果需要加以修正，修正量为 2^n，即对结果的符号位取反后才是移码形式的正确结果。

② 移码表示中，0 有唯一的编码为 1000…00。

当出现 000…00 时（表示-2^n），属于浮点数下溢，下溢按机器零处理。

3.1.2 乘法运算

在一些简单的计算机中，乘法运算可以完全用软件来实现。利用计算机所设置加法、移位等指令，编写一段程序完成两数相乘。这样做 CPU 的硬件结构简单，但实现乘法所用的时间很长，速度很慢。另一种情况是在 ALU 等硬件的基础上，再适当增加一些硬件构成乘法器。依据这种思路实现乘法，硬件要复杂一些，但速度也比较快。速度最快的是全部由硬件来实现的阵列乘法器。这种方法硬件就更加复杂。可见，可以用硬件来换取速度。前一种方法是软件设计问题，这里不再说明。本小节将讨论后面两种实现乘法的方法。

1. 原码一位乘法运算

1）原码一位乘法的法则

假定被乘数 X 和乘数 Y 为用原码表示的纯小数（下面的讨论同样适用于纯整数），分别为：

$$[X]_{原}=X_0.X_{-1}X_{-2}\cdots X_{-(n-1)}$$

$$[Y]_{原}=Y_0.Y_{-1}Y_{-2}\cdots Y_{-(n-1)}$$

其中 X_0、Y_0 是它们的符号位。

同时，假定乘积为：

$$[Z]_{原}=Z_0.Z_{-1}Z_{-2}\cdots Z_{-(2n-1)}$$

原码一位乘法的法则是：

① 乘积的符号为被乘数的符号位与乘数的符号位相异或；

② 乘积的绝对值为被乘数的绝对值与乘数的绝对值之积。即

$$[X]_{原} \times [Y]_{原} = (X_0 \oplus Y_0)(|X| \times |Y|) \tag{3-14}$$

可见，原码一位乘法可以分别求出积的符号和两乘数绝对值之积，然后拼接在一起，构成乘积。

2）原码一位乘法的实现思路

(1) 手工乘法运算

为了求得绝对值之积，先举例来讨论用手算的过程。

例 3.10 若$[X]_{原}=0.1101$，$[Y]_{原}=1.1011$，求两者之积。

解：乘积的符号为 $0 \oplus 1=1$

手算过程如下：

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ 1101 \\ \hline .10001111 \end{array}$$

在上面计算结果小数点左边加上乘积的符号 1，即为两者原码积。

在上面的算式中，由于是手算，我们一次即可把经过左移的 4 个被乘数加起来，立即便可获得结果。但在 CPU 中，由于其内部只有一个 ALU，一次只能加一个经左移的(最低位时不左移)被乘数。要加 4 次才能获得结果。我们将每次相加的结果称为部分积。

另外，还需要强调的是在上面手算过程中，根据乘数各位权值不同，每一次使被乘数左移一位加到部分积上。但在实际构成乘法器时，这样做比校麻烦。因此，可以采用被乘数位置不动而使部分积右移。这样做实现起来就更加容易。

(2) 思路流程

根据上面的分析，可将求绝对值之积的思路描述如图 3.6 所示。

设置一个寄存器 D，开始置 0，运算的中间过程放部分积，最后放结果。用一个寄存器 A，开始放乘数，并与 D 连在一起在运算中间放部分积，最后放结果。从乘数的最低位开始，若为 1，则加被乘数到部分积(D 连 A)。若为 0，则部分积加 0。部分积右移 1 位；再检测乘数的次低位，若为 1，则加被乘数到部分积(D 连 A)。若为 0，则部分积加 0。部分积右移 1 位；以此类推，直到乘数各位检测完。

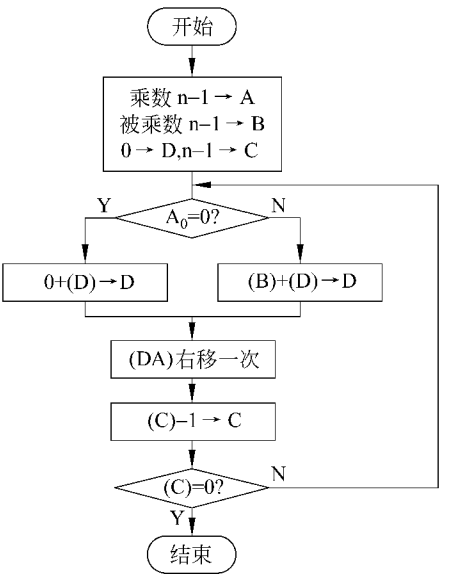

图 3.6 绝对值乘法思路框图