① 机器数为正时，不论是左移还是右移，添补代码均为 0。

② 由于负数的原码数值部分与真值相同，故在移位时只要使符号位不变，其空位均添 0 即可。

③ 由于负数的反码各位除符号位外与负数的原码正好相反，故移位后所添的代码应与原码相反，即全部添 1。

④ 分析任意负数的补码可发现，当对其由低位向高位找到第一个“1”时，在此“1”左边的各位均与对应的反码相同，而在此“1”右边的各位（包括此“1”在内）均与对应的原码相同。故负数的补码左移时，因空位出现在低位，则添补的代码与原码相同，即添 0；右移时因空位出现在高位，则添补的代码应与反码相同，即添 1。

例 6.7　设机器数字长为 8 位（含 1 位符号位），若 $A=\pm 26$，写出三种机器数左、右移一位和两位后的表示形式及对应的真值，并分析结果的正确性。

解：（1）$A=+26=(+11010)_{二}$

则 $[A]_{原}=[A]_{补}=[A]_{反}=0,0011010$

移位结果如表 6.5 所示。

表 6.5　对 $A=+26$ 移位后的结果

移位操作	机　器　数	对应的真值
	$[A]_{原}=[A]_{补}=[A]_{反}$	
移位前	0，0011010	+26
左移一位	0，0110100	+52
左移两位	0，1101000	+104
右移一位	0，0001101	+13
右移两位	0，0000110	+6

可见，对于正数，三种机器数移位后符号位均不变，左移时最高数位丢 1，结果出错；右移时最低数位丢 1，影响精度。

（2）$A=-26=(-11010)_{二}$

三种机器数移位结果如表 6.6 所示。

表 6.6　对 $A=-26$ 移位后的结果

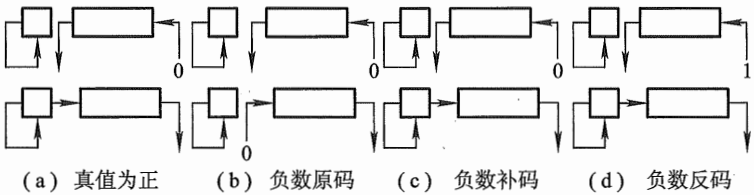
移位操作	机　器　数		对应的真值
移位前	原码	1，0011010	−26
左移一位		1，0110100	−52
左移两位		1，1101000	−104
右移一位		1，0001101	−13
右移两位		1，0000110	−6

续表

移位操作	机　器　数		对应的真值
移位前	补码	1，1100110	-26
左移一位		1，1001100	-52
左移两位		1，0011000	-104
右移一位		1，1110011	-13
右移两位		1，1111001	-7
移位前	反码	1，1100101	-26
左移一位		1，1001011	-52
左移两位		1，0010111	-104
右移一位		1，1110010	-13
右移两位		1，1111001	-6

可见，对于负数，三种机器数算术移位后符号位均不变。负数的原码左移时，高位丢 1，结果出错；右移时，低位丢 1，影响精度。负数的补码左移时，高位丢 0，结果出错；右移时，低位丢 1，影响精度。负数的反码左移时，高位丢 0，结果出错；右移时，低位丢 0，影响精度。

图 6.3 示意了机器中实现算术左移和右移操作的硬件框图。其中，图 6.3(a)为真值为正的三种机器数的移位操作；图 6.3(b)为负数原码的移位操作；图 6.3(c)为负数补码的移位操作；图 6.3(d)为负数反码的移位操作。



(a) 真值为正　(b) 负数原码　(c) 负数补码　(d) 负数反码

图 6.3　实现算术左移和右移操作的硬件示意图

3. 算术移位和逻辑移位的区别

有符号数的移位称为算术移位，无符号数的移位称为逻辑移位。逻辑移位的规则是：逻辑左移时，高位移丢，低位添 0；逻辑右移时，低位移丢，高位添 0。例如，寄存器内容为 01010011，逻辑左移为 10100110，算术左移为 00100110(最高数位“1”移丢)。又如，寄存器内容为 10110010，逻辑右移为 01011001，若将其视为补码，算术右移为 11011001。显然，两种移位的结果是不同的。上例中为了避免算术左移时最高数位丢 1，可采用带进位(C_y)的移位，其示意图如图 6.4 所示。算术左移时，符号位移至 C_y，最高数位就可避免移丢。

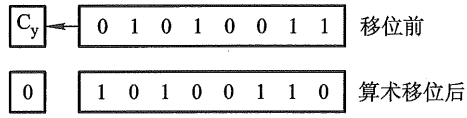


图 6.4　用带进位的移位实现算术左移

6.3.2　加法与减法运算

加减法运算是计算机中最基本的运算，因减法运算可看作被减数加上一个减数的负值，即 $A-B=A+(-B)$，故在此将机器中的减法运算和加法运算合在一起讨论。现代计算机中都采用补码作加减法运算。

1. 补码加减运算的基本公式

补码加法的基本公式如下：

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

小数　　$[A]_{补}+[B]_{补}=[A+B]_{补}(\bmod 2)$

即补码表示的两个数在进行加法运算时，可以把符号位与数值位同等处理，只要结果不超出机器能表示的数值范围，运算后的结果按 2^{n+1} 取模（对于整数）或按 2 取模（对于小数），就能得到本次加法的运算结果。

读者可根据补码定义，按两个操作数的四种正负组合情况加以证明。

对于减法，因 $A-B=A+(-B)$

则 $[A-B]_{补}=[A+(-B)]_{补}$

由补码加法基本公式可得

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

小数　　$[A-B]_{补}=[A]_{补}+[-B]_{补}(\bmod 2)$

因此，若机器数采用补码，当求 $A-B$ 时，只需先求 $[-B]_{补}$（称 $[-B]_{补}$ 为“求补”后的减数），就可按补码加法规则进行运算。而 $[-B]_{补}$ 由 $[B]_{补}$ 连同符号位在内，每位取反，末位加 1 而得。

例 6.8　已知 $A=0.1011$，$B=-0.0101$，求 $[A+B]_{补}$。

解：因为　$A=0.1011$，$B=-0.0101$

所以　　$[A]_{补}=0.1011$，$[B]_{补}=1.1011$

则　　$[A]_{补}+[B]_{补}=0.1011$

$$\begin{array}{r} +1.1011 \\ \hline \boxed{1}\ \ 0.0110=[A+B]_{补} \end{array}$$

丢掉

按模2的意义，最左边的1丢掉，故$[A+B]_{补}=0.0110$，结果正确。

例6.9　已知$A=-1001$，$B=-0101$，求$[A+B]_{补}$。

解：因为　$A=-1001$，$B=-0101$

所以　　$[A]_{补}=1,0111$，$[B]_{补}=1,1011$

则　　$[A]_{补}+[B]_{补}=1,0111$

$$\begin{array}{r} +1,1011 \\ \hline \boxed{1}\ \ 1,0010=[A+B]_{补} \end{array}$$

丢掉←

按模2^{4+1}的意义，最左边的1丢掉。

例6.10　设机器数字长为8位(含1位符号位)，若$A=+15$，$B=+24$，求$[A-B]_{补}$并还原成真值。

解：因为　$A=+15=+0001111$，$B=+24=+0011000$

所以　　$[A]_{补}=0,0001111$，$[B]_{补}=0,0011000$，$[-B]_{补}=1,1101000$

则　　$[A-B]_{补}=[A]_{补}+[-B]_{补}=0,0001111$

$$\begin{array}{r} +1,1101000 \\ \hline 1,1110111 \end{array}$$

所以　　$[A-B]_{补}=1,1110111$

故　　$A-B=-0001001=-9$

可见，不论操作数是正还是负，在做补码加减法时，只需将符号位和数值部分一起参加运算，并且将符号位产生的进位自然丢掉即可。

例6.11　设机器数字长为8位，其中1位为符号位，令$A=-93$，$B=+45$，求$[A-B]_{补}$。

解：由$A=-93=-1011101$，得$[A]_{补}=1,0100011$

由$B=+45=+0101101$，得$[B]_{补}=0,0101101$，$[-B]_{补}=1,1010011$

$$\begin{array}{r} [A]_{补}= \quad 1,0100011 \\ +[-B]_{补}= \quad 1,1010011 \\ \hline [A]_{补}+[-B]_{补}=\boxed{1}\ 0,1110110=[A-B]_{补} \end{array}$$

丢掉←

按模2^{n+1}的意义，最左边的“1”自然丢掉，故$[A-B]_{补}=0,1110110$，还原成真值得$A-B=118$，结果出错，这是因为$A-B=-138$超出了机器字长所能表示的范围。在计算机中，这种超出机器字长的现象叫溢出。为此，在补码定点加减运算过程中，必须对结果是否溢出做出明确的判断。

2. 溢出判断

补码定点加减运算判断溢出有两种方法。

(1) 用一位符号位判断溢出

对于加法,只有在正数加正数和负数加负数两种情况下才可能出现溢出,符号不同的两个数相加是不会溢出的。

对于减法,只有在正数减负数或负数减正数两种情况下才可能出现溢出,符号相同的两个数相减是不会溢出的。

下面以机器字长为 4 位(含 1 位符号位)为例,说明机器是如何判断溢出的。

机器字长为 4 位的补码所对应的真值范围为 $-8 \sim +7$,运算结果一旦超过这个范围即为溢出。表 6.7 列出了四种溢出情况。

由于减法运算在机器中是用加法器实现的,因此可得出如下结论:不论是作加法还是减法,只要实际参加操作的两个数(减法时即为被减数和"求补"以后的减数)符号相同,结果又与原操作数的符号不同,即为溢出。

表 6.7　补码定点运算溢出判断举例

真　值	补码运算
$A=5$ $+B=4$ $A+B=9>7$　溢出	$[A]_{补}=0,101$ $+[B]_{补}=0,100$ $[A+B]_{补}=1,001$
$A=-5$ $+B=-4$ $A+B=-9<-8$　溢出	$[A]_{补}=1,011$ $+[B]_{补}=1,100$ $[A+B]_{补}=10,111$
$A=5$ $-B=-4$ $A-B=9>7$　溢出	$[A]_{补}=0,101$ $+[-B]_{补}=0,100$ $[A-B]_{补}=1,001$
$A=-5$ $-B=+4$ $A-B=-9<-8$　溢出	$[A]_{补}=1,011$ $+[-B]_{补}=1,100$ $[A-B]_{补}=10,111$

例 6.12　已知 $A=-\dfrac{11}{16}, B=-\dfrac{7}{16}$,求 $[A+B]_{补}$。

解:由 $A=-\dfrac{11}{16}=-0.1011, B=-\dfrac{7}{16}=-0.0111$

得　$[A]_{补}=1.0101, [B]_{补}=1.1001$

所以

$$
\begin{aligned}
[A+B]_{补} &= [A]_{补} = 1.0101 \\
&+[B]_{补} = 1.1001 \\
\hline
&\boxed{1}\ 0.1110
\end{aligned}
$$

丢掉

两操作数符号均为1，结果的符号为0，故为溢出。

例 6.13　已知 $A=-0.1000, B=-0.1000$，求 $[A+B]_{补}$。

解：由 $A=-0.1000, B=-0.1000$

得　　$[A]_{补}=1.1000, [B]_{补}=1.1000$

所以

$$\begin{aligned}[A+B]_{补}&=[A]_{补}=1.1000\\&\quad+[B]_{补}=1.1000\\\hline&\quad\boxed{1}\ 1.0000\end{aligned}$$

丢掉←

结果的符号同原操作数符号，故未溢出。

由 $[A+B]_{补}=1.0000$，得 $A+B=-1$，由此可见，用补码表示定点小数时，它能表示-1的值。

计算机中采用1位符号位判断时，为了节省时间，通常用符号位产生的进位与最高有效位产生的进位异或操作后，按其结果进行判断。若异或结果为1，即为溢出；异或结果为0，则无溢出。例6.12中符号位有进位，最高有效位无进位，即 $1\oplus 0=1$，故溢出。例6.13中符号位有进位，最高有效位也有进位，即 $1\oplus 1=0$，故无溢出。

(2) 用两位符号位判断溢出

在6.1.2节中已提到过2位符号位的补码，即变形补码，它是以4为模的，其定义为

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

在用变形补码作加法时，2位符号位要连同数值部分一起参加运算，而且高位符号位产生的进位自动丢失，便可得正确结果，即

$$[x]_{补'}+[y]_{补'}=[x+y]_{补'}\quad(\mathrm{mod}\ 4)$$

变形补码判断溢出的原则是：当2位符号位不同时，表示溢出，否则；无溢出。不论是否发生溢出，高位(第1位)符号位永远代表真正的符号。

例 6.14　设 $x=+\dfrac{11}{16}, y=+\dfrac{3}{16}$，试用变形补码计算 $x+y$。

解：因为　$x=+\dfrac{11}{16}=0.1011, y=+\dfrac{3}{16}=0.0011$

所以　　$[x]_{补'}=00.1011, [y]_{补'}=00.0011$

则

$$\begin{aligned}[x]_{补'}+[y]_{补'}&=00.1011\\&\quad+00.0011\\\hline&\quad 00.1110\end{aligned}$$

故　$[x+y]_{补'}=00.1110$

$x+y=0.1110$

例 6.15　设 $x=+\frac{11}{16}, y=+\frac{7}{16}$，试用变形补码计算 $x+y$。

解：因为　$x=+\frac{11}{16}=0.1011, y=+\frac{7}{16}=0.0111$

所以　　$[x]_{补'}=00.1011, [y]_{补'}=00.0111$

则　　$[x]_{补'}+[y]_{补'}=00.1011$

$$\begin{array}{r} +00.0111 \\ \hline 01.0010 \end{array}$$

第 1 位符号位 → 01 为溢出

此时，符号位为“01”，表示溢出，又因第 1 位符号位为“0”，表示结果的真正符号为正，故“01”表示正溢出。

例 6.16　设 $x=-\frac{11}{16}, y=-\frac{7}{16}$，用变形补码计算 $x+y$。

解：因为　$x=-\frac{11}{16}=-0.1011, y=-\frac{7}{16}=-0.0111$

所以　　$[x]_{补'}=11.0101, [y]_{补'}=11.1001$

则

$$[x]_{补'}+[y]_{补'}=11.0101$$

$$\begin{array}{r} +11.1001 \\ \hline 1\,10.1110 \end{array}$$

丢掉 ← 1

符号位为“10”，表示溢出。由于第 1 位符号位为 1，则表示负溢出。

上述结论对于整数也同样适用。在浮点机中，当阶码用两位符号位表示时，判断溢出的原则与小数的完全相同。

这里需要说明一点，采用双符号位方案时，寄存器或主存中的操作数只需保存一位符号位即可。因为任何正确的数，两个符号位的值总是相同的，而双符号位在加法器中又是必要的，故在相加时，寄存器中一位符号的值要同时送到加法器的两位符号位的输入端。

3. 补码定点加减法所需的硬件配置

图 6.5 是实现补码定点加减法的基本硬件配置框图。

图中寄存器 A、X、加法器的位数相等，其中 A 存放被加数（或被减数）的补码，X 存放加数（或减数）的补码。当作减法时，由“求补控制逻辑”将 $\overline{X}$ 送至加法器，并使加法器的最末位外来进位为 1，以达到对减数求补的目的。运算结果溢出时，通过溢出判断电路置“1”溢出标记 V。G_A 为加法标记，G_S 为减法标记。

4. 补码加减运算控制流程

补码加减运算控制流程如图 6.6 所示。

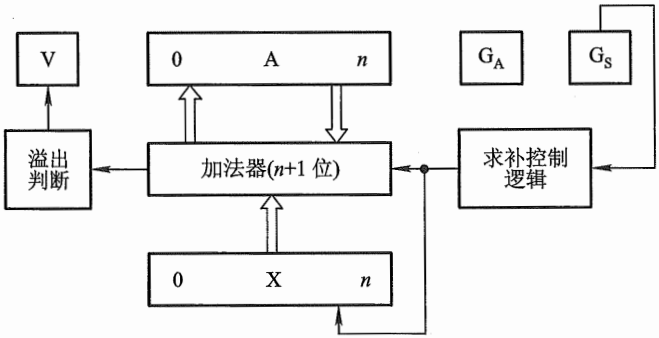


图 6.5　补码定点加减法硬件配置

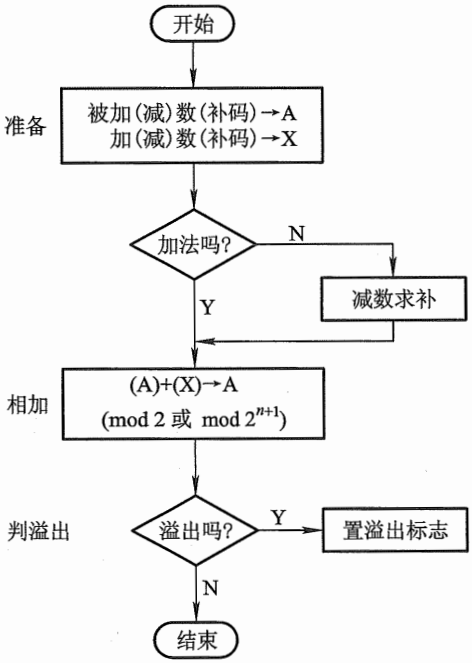


图 6.6　补码加减运算控制流程

由图可见,加(减)法运算前,被加(减)数的补码在 A 中,加(减)数的补码在 X 中。若是加法,直接完成(A)+(X)→A(mod 2 或 mod 2^{n+1})的运算;若是减法,则需对减数求补,再和 A 寄存器的内容相加,结果送 A。最后完成溢出判断。

6.3.3　乘法运算

在计算机中，乘法运算是一种很重要的运算，有的机器由硬件乘法器直接完成乘法运算，有的机器内没有乘法器，但可以按机器做乘法运算的方法，用软件编程实现。因此，学习乘法运算方法不仅有助于乘法器的设计，也有助于乘法编程。

下面从分析笔算乘法入手，介绍机器中用到的几种乘法运算方法。

1. 分析笔算乘法

设 $A=0.1101$，$B=0.1011$，求 $A\times B$。

笔算乘法时，乘积的符号由两数符号心算而得：正正得正。其数值部分的运算如下：

$$
\begin{array}{rl}
 & 0.1101 \\
\times & 0.1011 \\
\hline
 & 1101 \quad \cdots\cdots A\times 2^{0} \quad A\text{ 不移位} \\
 & 1101 \quad \cdots\cdots A\times 2^{1} \quad A\text{ 左移 1 位} \\
 & 0000 \quad \cdots\cdots 0\times 2^{2} \quad 0\text{ 左移 2 位} \\
 & 1101 \quad \cdots\cdots A\times 2^{3} \quad A\text{ 左移 3 位} \\
\hline
 & 0.10001111
\end{array}
$$

所以　$A\times B=+0.10001111$

可见，这里包含着被乘数 A 的多次左移，以及 4 个位积的相加运算。

若计算机完全模仿笔算乘法步骤，将会有两大困难：其一，将 4 个位积一次相加，机器难以实现；其二，乘积位数增长了一倍，这将造成器材的浪费和运算时间的增加。为此，对笔算乘法进行改进。

2. 笔算乘法的改进

$$
\begin{aligned}
A\cdot B &= A\cdot 0.1011 \\
&= 0.1A+0.00A+0.001A+0.0001A \\
&= 0.1A+0.00A+0.001(A+0.1A) \\
&= 0.1A+0.01[0A+0.1(A+0.1A)] \\
&= 0.1\{A+0.1[0A+0.1(A+0.1A)]\} \\
&= 2^{-1}\{A+2^{-1}[0A+2^{-1}(A+2^{-1}A)]\} \\
&= 2^{-1}\{A+2^{-1}[0A+2^{-1}(A+2^{-1}(A+0))]\}
\end{aligned}
\tag{6.8}
$$

由式(6.8)可见，两数相乘的过程，可视为加法和移位(乘 2^{-1} 相当于做一位右移)两种运算，这对计算机来说是非常容易实现的。

从初始值为 0 开始，对式(6.8)作分步运算，则

第一步：被乘数加零　　$A+0=0.1101+0.0000=0.1101$

第二步：右移一位，得新的部分积　　$2^{-1}(A+0)=0.01101$

第三步：被乘数加部分积　　$A+2^{-1}(A+0)=0.1101+0.01101=1.00111$

第四步：右移一位，得新的部分积　　$2^{-1}[A+2^{-1}(A+0)]=0.100111$

第五步：　　$0\cdot A+2^{-1}[A+2^{-1}(A+0)]=0.100111$

第六步：　　$2^{-1}\{0\cdot A+2^{-1}[A+2^{-1}(A+0)]\}=0.0100111$

第七步：　　$A+2^{-1}\{0\cdot A+2^{-1}[A+2^{-1}(A+0)]\}=1.0001111$

第八步：　　$2^{-1}\{A+2^{-1}[0\cdot A+2^{-1}(A+2^{-1}(A+0))]\}=0.10001111$

表 6.8 列出了式(6.8)的全部运算过程。

表 6.8　式(6.8)的运算过程

部 分 积	乘 数	说　明
0.0000 + 0.1101	1011	初始条件，部分积为 0 乘数为 1，加被乘数
0.1101 0.0110 + 0.1101	1101	→1 位，形成新的部分积；乘数同时→1 位 乘数为 1，加被乘数
1.0011 0.1001 + 0.0000	1 1110	→1 位，形成新的部分积；乘数同时→1 位 乘数为 0，加上 0
0.1001 0.0100 + 0.1101	11 1111	→1 位，形成新的部分积；乘数同时→1 位 乘数为 1，加被乘数
1.0001 0.1000	111 1111	→1 位，形成最终结果

上述运算过程可归纳如下：

① 乘法运算可用移位和加法来实现，两个 4 位数相乘，总共需要进行 4 次加法运算和 4 次移位。

② 由乘数的末位值确定被乘数是否与原部分积相加，然后右移一位，形成新的部分积；同时，乘数也右移一位，由次低位作新的末位，空出最高位放部分积的最低位。

③ 每次做加法时，被乘数仅仅与原部分积的高位相加，其低位被移至乘数所空出的高位位置。

计算机很容易实现这种运算规则。用一个寄存器存放被乘数，一个寄存器存放乘积的高位，另一个寄存器存放乘数及乘积的低位，再配上加法器及其他相应电路，就可组成乘法器。又因加法只在部分积的高位进行，故不但节省了器材，而且还缩短了运算时间。

3. 原码乘法

由于原码表示与真值极为相似，只差一个符号，而乘积的符号又可通过两数符号的逻辑异或

求得，因此，上述讨论的结果可以直接用于原码一位乘，只需加上符号位处理即可。

(1) 原码一位乘运算规则

以小数为例：

设　$[x]_{原}=x_0.x_1x_2\cdots x_n$

　　$[y]_{原}=y_0.y_1y_2\cdots y_n$

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

式中，$0.x_1x_2\cdots x_n$ 为 x 的绝对值，记作 x^*；$0.y_1y_2\cdots y_n$ 为 y 的绝对值，记作 y^*。

原码一位乘的运算规则如下：

① 乘积的符号位由两原码符号位异或运算结果决定。

② 乘积的数值部分由两数绝对值相乘，其通式为

$$\left.\begin{aligned}x^*\cdot y^* &= x^*(0.y_1y_2\cdots y_n)\\ &= x^*(y_12^{-1}+y_22^{-2}+\cdots+y_n2^{-n})\\ &= \underbrace{2^{-1}(y_1x^*+\underbrace{2^{-1}(y_2x^*+2^{-1}(\cdots+\underbrace{2^{-1}(y_{n-1}x^*+\underbrace{2^{-1}(y_nx^*+\overset{z_0}{0})}_{z_1})}_{z_2}\cdots))}_{z_{n-1}})}_{z_n}\end{aligned}\right\}\quad(6.9)$$

再令 z_i 表示第 i 次部分积，式(6.9)可写成如下递推公式。

$$\left.\begin{aligned}z_0&=0\\ z_1&=2^{-1}(y_n\cdot x^*+z_0)\\ z_2&=2^{-1}(y_{n-1}\cdot x^*+z_1)\\ &\vdots\\ z_i&=2^{-1}(y_{n-i+1}\cdot x^*+z_{i-1})\\ &\vdots\\ z_n&=2^{-1}(y_1\cdot x^*+z_{n-1})\end{aligned}\right\}\quad(6.10)$$

例 6.17　已知 $x=-0.1110$，$y=-0.1101$，求 $[x\cdot y]_{原}$。

解：因为　$x=-0.1110$

所以 $[x]_{原}=1.1110$，$x^*=0.1110$（为绝对值），$x_0=1$

又因为　$y=-0.1101$

所以 $[y]_{原}=1.1101$，$y^*=0.1101$（为绝对值），$y_0=1$

按原码一位乘运算规则，$[x\cdot y]_{原}$ 的数值部分计算如表 6.9 所示。

表 6.9　例 6.17 数值部分的计算

部　分　积	乘　　数	说　　明
0.0000	1101	开始部分积 $z_0=0$
+0.1110		乘数为 1,加上 x^*
0.1110		
0.0111	0110	→1 位得 z_1,乘数同时→1 位
+0.0000		乘数为 0,加上 0
0.0111	0	
0.0011	1011	→1 位得 z_2,乘数同时→1 位
+0.1110		乘数为 1,加上 x^*
1.0001	10	
0.1000	1101	→1 位得 z_3,乘数同时→1 位
+0.1110		乘数为 1,加上 x^*
1.0110	110	
0.1011	0110	→1 位得 z_4,乘数已全部移出

即 $x^* \cdot y^* = 0.10110110$

乘积的符号位为 $x_0 \oplus y_0 = 1 \oplus 1 = 0$

故　$[x \cdot y]_{原} = 0.10110110$

值得注意的是,这里部分积取 $n+1$ 位,以便存放乘法过程中绝对值大于或等于 1 的值。此外,由于乘积的数值部分是两数绝对值相乘的结果,故原码一位乘法运算过程中的右移操作均为逻辑右移。

(2) 原码一位乘所需的硬件配置

图 6.7 是实现原码一位乘运算的基本硬件配置框图。

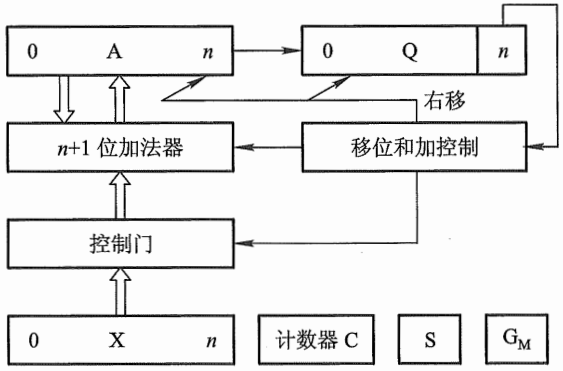


图 6.7　原码一位乘运算基本配置

图中 A、X、Q 均为 $n+1$ 位的寄存器，其中 X 存放被乘数的原码，Q 存放乘数的原码。移位和加控制电路受末位乘数 Q_n 的控制（当 $Q_n=1$ 时，A 和 X 内容相加后，A、Q 右移一位；当 $Q_n=0$ 时，只作 A、Q 右移一位的操作）。计数器 C 用于控制逐位相乘的次数。S 存放乘积的符号。G_M 为乘法标记。

（3）原码一位乘控制流程

原码一位乘控制流程如图 6.8 所示。

乘法运算前，A 寄存器被清零，作为初始部分积，被乘数原码在 X 中，乘数原码在 Q 中，计数器 C 中存放乘数的位数 n。乘法开始后，首先通过异或运算，求出乘积的符号并存于 S，接着将被乘数和乘数从原码形式变为绝对值。然后根据 Q_n 的状态决定部分积是否加上被乘数，再逻辑右移一位，重复 n 次，即得运算结果。

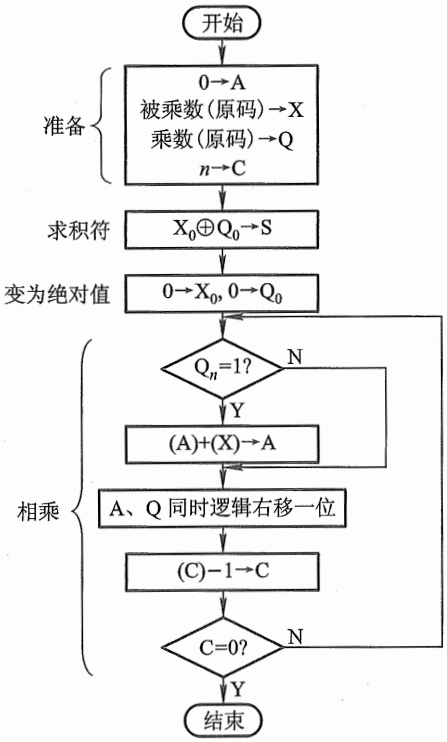


图 6.8　原码一位乘控制流程

上述讨论的运算规则同样可用于整数原码。为了区别于小数乘法，书写上可将表 6.9 中的“.”改为“,”。

为了提高乘法速度，可采用原码两位乘。

（4）原码两位乘

原码两位乘与原码一位乘一样，符号位的运算和数值部分是分开进行的，但原码两位乘是用

两位乘数的状态来决定新的部分积如何形成，因此可提高运算速度。

两位乘数共有四种状态，对应这四种状态可得表 6.10。

表 6.10　两位乘数所对应的新的部分积

乘数 $y_{n-1}y_n$	新的部分积
0 0	新部分积等于原部分积右移两位
0 1	新部分积等于原部分积加被乘数后右移两位
1 0	新部分积等于原部分积加 2 倍被乘数后右移两位
1 1	新部分积等于原部分积加 3 倍被乘数后右移两位

表中 2 倍被乘数可通过将被乘数左移一位实现，但 3 倍被乘数的获得较难。此刻可将 3 视为 4-1(11=100-1)，即把乘以 3 分两步完成，第一步先完成减 1 倍被乘数的操作，第二步完成加 4 倍被乘数的操作。而加 4 倍被乘数的操作实际上是由比“11”高的两位乘数代替完成的，可看作是在高两位乘数上加“1”。这个“1”可暂存在 C_j 触发器中。机器完成 C_j 置“1”，即意味着对高两位乘数加 1，也即要求高两位乘数代替本两位乘数“11”来完成加 4 倍被乘数的操作。由此可得原码两位乘的运算规则如表 6.11 所示。

表 6.11　原码两位乘的运算规则

乘数判断位 $y_{n-1}y_n$	标志位 C_j	操 作 内 容
0 0	0	$z\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 保持“0”
0 1	0	$z+x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 保持“0”
1 0	0	$z+2x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 保持“0”
1 1	0	$z-x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 置“1”
0 0	1	$z+x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 置“0”
0 1	1	$z+2x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 置“0”
1 0	1	$z-x^*\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 保持“1”
1 1	1	$z\rightarrow$2 位，$y^*\rightarrow$2 位，C_j 保持“1”

表中 z 表示原有部分积，x^* 表示被乘数的绝对值，y^* 表示乘数的绝对值，$\rightarrow$2 表示右移两位，当进行 $-x^*$ 运算时，一般都采用加 $[-x^*]_{补}$ 来实现。这样，参与原码两位乘运算的操作数是绝对值的补码，因此运算中右移两位的操作也必须按补码右移规则完成。尤其应注意的是，乘法过程中可能要加 2 倍被乘数，即 $+[2x^*]_{补}$，使部分积的绝对值大于 2。为此，只有对部分积取 3 位符号位，且以最高符号位作为真正的符号位，才能保证运算过程正确无误。

此外，为了统一用两位乘数和一位 C_j 共同配合管理全部操作，与原码一位乘不同的是，需在乘数（当乘数位数为偶数时）的最高位前增加两个 0。这样，当乘数最高两个有效位出现“11”时，需将 C_j 置“1”，再与所添补的两个 0 结合呈 001 状态，以完成加 x^* 的操作（此步不必移位）。

例 6.18　设 $x=0.111111$，$y=-0.111001$，用原码两位乘求 $[x\cdot y]_{原}$。

解：① 数值部分的计算如表 6.12 所示，其中

$$x^*=0.111111,[-x^*]_{补}=1.000001,2x^*=1.111110,y^*=0.111001$$

表 6.12　例 6.18 原码两位乘数值部分的运算过程

部　分　积	乘数 y^*	C_j	说　　明
000.000000 +000.111111	00111001	0	开始，部分积为 0，$C_j=0$ 根据 $y_{n-1}y_nC_j=010$，加 x^*，保持 $C_j=0$
000.111111 000.001111 +001.111110	11001110	0	→2 位，得新的部分积，乘数同时→2 位 根据“100”加 2 x^*，保持 $C_j=0$
010.001101 000.100011 +111.000001	11 01110011	0	→2 位，得新的部分积，乘数同时→2 位 根据“110”减 x^*（即加 $[-x^*]_{补}$），C_j 置“1”
111.100100 111.111001 +000.111111	0111 00011100	1	→2 位，得新的部分积，乘数同时→2 位 根据“001”加 x^*，C_j 置“0”
000.111000	000111		形成最终结果

② 乘积符号的确定：

$$x_0\oplus y_0=0\oplus 1=1$$

故　　$[x\cdot y]_{原}=1.111000000111$

不难理解，当乘数为偶数时，需做 $n/2$ 次移位，最多做 $n/2+1$ 次加法。当乘数为奇数时，乘数高位前可只增加一个“0”，此时需做 $n/2+1$ 次移位（最后一步移一位），最多需做 $n/2+1$ 次加法。

虽然两位乘法可提高乘法速度，但它仍基于重复相加和移位的思想，而且随着乘数位数的增加，重复次数增多，仍然影响乘法速度的进一步提高。采用并行阵列乘法器可大大提高乘法速度。有关阵列乘法器的内容可参见附录 6B。

原码乘法实现比较容易，但由于机器都采用补码做加减运算，倘若做乘法前再将补码转换成原码，相乘之后又要将负积的原码变为补码形式，这样增添了许多操作步骤，反而使运算复杂。为此，有不少机器直接用补码相乘，机器里配置实现补码乘法的乘法器，避免了码制的转换，提高了机器效率。