4. 补码乘法

(1) 补码一位乘运算规则

设被乘数 $[x]_补 = x_0.x_1x_2\cdots x_n$

乘数 $[y]_补 = y_0.y_1y_2\cdots y_n$

1) 被乘数 x 符号任意,乘数 y 符号为正

$$[x]_补 = x_0.x_1x_2\cdots x_n = 2+x = 2^{n+1}+x \pmod 2$$

$$[y]_补 = 0.y_1y_2\cdots y_n = y$$

则 $$[x]_补 \cdot [y]_补 = [x]_补 \cdot y = (2^{n+1}+x)\cdot y = 2^{n+1}\cdot y+xy$$

由于 $y = 0.y_1y_2\cdots y_n = \sum_{i=1}^{n} y_i 2^{-i}$,则 $2^{n+1}\cdot y = 2\sum_{i=1}^{n} y_i 2^{n-i}$,且 $\sum_{i=1}^{n} y_i 2^{n-i}$是一个大于或等于 1 的正整数,根据模运算的性质,有 $2^{n+1}\cdot y = 2 \pmod 2$,故

$$[x]_补 \cdot [y]_补 = 2^{n+1}\cdot y+xy = 2+xy = [x\cdot y]_补 \qquad \pmod 2$$

即 $$[x\cdot y]_补 = [x]_补 \cdot [y]_补 = [x]_补 \cdot y$$

对照原码乘法式(6.9)和式(6.10)可见,当乘数 y 为正数时,不管被乘数 x 符号如何,都可按原码乘法的规则运算,即

$$\left.\begin{aligned}
&[z_0]_补 = 0\\
&[z_1]_补 = 2^{-1}(y_n[x]_补+[z_0]_补)\\
&[z_2]_补 = 2^{-1}(y_{n-1}[x]_补+[z_1]_补)\\
&\quad\vdots\\
&[z_i]_补 = 2^{-1}(y_{n-i+1}[x]_补+[z_{i-1}]_补)\\
&\quad\vdots\\
&[x\cdot y]_补 = [z_n]_补 = 2^{-1}(y_1[x]_补+[z_{n-1}]_补)
\end{aligned}\right\} \tag{6.11}$$

当然这里的加和移位都必须按补码规则运算。

2) 被乘数 x 符号任意,乘数 y 符号为负

$$[x]_补 = x_0.x_1x_2\cdots x_n$$

$$[y]_补 = 1.y_1y_2\cdots y_n = 2+y \pmod 2$$

则 $$y = [y]_补 - 2 = 1.y_1y_2\cdots y_n - 2 = 0.y_1y_2\cdots y_n - 1$$

$$\begin{aligned} x\cdot y &= x(0.y_1y_2\cdots y_n - 1)\\ &= x(0.y_1y_2\cdots y_n) - x \end{aligned}$$

故 $$[x\cdot y]_补 = [x(0.y_1y_2\cdots y_n)]_补 + [-x]_补$$

将上式 $0.y_1y_2\cdots y_n$ 视为一个正数,正好与上述情况相同。

则 $[x(0.y_1y_2\cdots y_n)]_补 = [x]_补(0.y_1y_2\cdots y_n)$

所以 $$[x\cdot y]_补 = [x]_补(0.y_1y_2\cdots y_n)+[-x]_补 \tag{6.12}$$

由此可得,当乘数为负时是把乘数的补码$[y]_补$去掉符号位,当成一个正数与 $[x]_补$相乘,然

后加上 $[-x]_{补}$ 进行校正，也称校正法，用递推公式表示如下：

$$\left.\begin{aligned}
[z_0]_{补} &= 0 \\
[z_1]_{补} &= 2^{-1}(y_n[x]_{补}+[z_0]_{补}) \\
[z_2]_{补} &= 2^{-1}(y_{n-1}[x]_{补}+[z_1]_{补}) \\
&\vdots \\
[z_i]_{补} &= 2^{-1}(y_{n-i+1}[x]_{补}+[z_{i-1}]_{补}) \\
&\vdots \\
[z_n]_{补} &= 2^{-1}(y_1[x]_{补}+[z_{n-1}]_{补}) \\
[x\cdot y]_{补} &= [z_n]_{补}+[-x]_{补}
\end{aligned}\right\} \tag{6.13}$$

比较式(6.13)与式(6.11)可见，乘数为负的补码乘法与乘数为正时类似，只需最后加上一项校正项 $[-x]_{补}$ 即可。

例 6.19　已知 $[x]_{补}=1.0101$，$[y]_{补}=0.1101$，求 $[x\cdot y]_{补}$。

解：因为乘数 $y>0$，所以按原码一位乘的算法运算，只是在相加和移位时按补码规则进行，如表 6.13 所示。考虑到运算时可能出现绝对值大于 1 的情况（但此刻并不是溢出），故部分积和被乘数取双符号位。

表 6.13　例 6.19 的运算过程

部分积	乘数	说　明
00.0000	$110\underline{1}$	初值 $[z_0]_{补}=0$
+ 11.0101		$y_4=1$，$+[x]_{补}$
11.0101		
11.1010	$111\underline{0}$	→1 位，得 $[z_1]_{补}$，乘数同时→1 位
11.1101	$011\underline{1}$	$y_3=0$，→1 位，得 $[z_2]_{补}$，乘数同时→1 位
+ 11.0101		$y_2=1$，$+[x]_{补}$
11.0010	01	
11.1001	$001\underline{1}$	→1 位，得 $[z_3]_{补}$，乘数同时→1 位
+ 11.0101		$y_1=1$，$+[x]_{补}$
10.1110	001	
11.0111	0001	→1 位，得 $[z_4]_{补}$

故　乘积 $[x\cdot y]_{补}=1.01110001$

例 6.20　已知 $[x]_{补}=0.1101$，$[y]_{补}=1.0101$，求 $[x\cdot y]_{补}$。

解：因为乘数 $y<0$，故先不考虑符号位，按原码一位乘的运算规则运算，最后再加上 $[-x]_{补}$，如表 6.14 所示。

表 6.14　例 6.20 的运算过程

部分积	乘数	说　　明
00.0000 + 00.1101	010$\underline{1}$	初值$[z_0]_补=0$ $y_4=1$，$+[x]_补$
00.1101 00.0110 00.0011 + 00.1101	 101$\underline{0}$ 010$\underline{1}$	 →1 位，得$[z_1]_补$，乘数同时→1 位 $y_3=0$，→1 位，得$[z_2]_补$ $y_2=1$，$+[x]_补$
01.0000 00.1000 00.0100 + 11.0011	01 001$\underline{0}$ 0001	 →1 位，得$[z_3]_补$，乘数同时→1 位 $y_1=0$，→1 位，得$[z_4]_补$ $+[-x]_补$进行校正
11.0111	0001	得最后结果$[x\cdot y]_补$

故　乘积 $[x\cdot y]_补=1.01110001$

由以上两例可见，乘积的符号位在运算过程中自然形成，这是补码乘法和原码乘法的重要区别。

上述校正法与乘数的符号有关，虽然可将乘数和被乘数互换，使乘数保持正，不必校正，但当两数均为负时必须校正。总之，实现校正法的控制线路比较复杂。若不考虑操作数符号，用统一的规则进行运算，就可采用比较法。

3）被乘数 x 和乘数 y 符号均为任意

比较法是 Booth 夫妇首先提出来的，故又称 Booth 算法。它的运算规则可由校正法导出。

设 $[x]_补=x_0.x_1x_2\cdots x_n$

　$[y]_补=y_0.y_1y_2\cdots y_n$

按补码乘法校正法规则，其基本算法可用一个统一的公式表示为

$$[x\cdot y]_补=[x]_补(0.y_1y_2\cdots y_n)-[x]_补\cdot y_0 \tag{6.14}$$

当 $y_0=0$ 时，表示乘数 y 为正，无须校正，即

$$[x\cdot y]_补=[x]_补(0.y_1y_2\cdots y_n) \tag{6.15}$$

当 $y_0=1$ 时，表示乘数 y 为负，则

$$[x\cdot y]_补=[x]_补(0.y_1y_2\cdots y_n)-[x]_补 \tag{6.16}$$

比较式(6.12)和式(6.16)，在 mod 2 的前提下，$[-x]_补=-[x]_补$成立①，所以式(6.15)和式(6.16)表达的算法与校正法的结论完全相同，故式(6.14)可以改写为

$$\begin{aligned}
[x\cdot y]_补 &= [x]_补(y_1 2^{-1}+y_2 2^{-2}+\cdots+y_n 2^{-n})-[x]_补\cdot y_0\\
&= [x]_补(-y_0+y_1 2^{-1}+y_2 2^{-2}+\cdots+y_n 2^{-n})\\
&= [x]_补[-y_0+(y_1-y_1 2^{-1})+(y_2 2^{-1}-y_2 2^{-2})+\cdots+(y_n 2^{-(n-1)}-y_n 2^{-n})]\\
&= [x]_补[(y_1-y_0)+(y_2-y_1)2^{-1}+\cdots+(y_n-y_{n-1})2^{-(n-1)}+(0-y_n)2^{-n}]\\
&= [x]_补[(y_1-y_0)+(y_2-y_1)2^{-1}+\cdots+(y_{n+1}-y_n)2^{-n}]
\end{aligned}\tag{6.17}$$

其中，$y_{n+1}=0$。

这样，可得如下递推公式。

$$\left.\begin{aligned}
[z_0]_补 &= 0\\
[z_1]_补 &= 2^{-1}\{[z_0]_补+(y_{n+1}-y_n)[x]_补\}\\
[z_2]_补 &= 2^{-1}\{[z_1]_补+(y_n-y_{n-1})[x]_补\}\\
&\vdots\\
[z_i]_补 &= 2^{-1}\{[z_{i-1}]_补+(y_{n-i+2}-y_{n-i+1})[x]_补\}\\
&\vdots\\
[z_n]_补 &= 2^{-1}\{[z_{n-1}]_补+(y_2-y_1)[x]_补\}\\
[x\cdot y]_补 = [z_{n+1}]_补 &= [z_n]_补+(y_1-y_0)[x]_补
\end{aligned}\right\}\tag{6.18}$$

由此可见，开始时 $y_{n+1}=0$，部分积初值$[z_0]_补$为 0，每一步乘法由$(y_{i+1}-y_i)$$(i=1,2,\cdots,n)$决定原部分积加$[x]_补$或加 $[-x]_补$或加 0，再右移一位得新的部分积，以此重复 n 步。第 $n+1$ 步由(y_1-y_0)决定原部分积加$[x]_补$或加$[-x]_补$或加 0，但不移位，即得 $[x\cdot y]_补$。

这里的$(y_{i+1}-y_i)$之差值恰恰与乘数末两位 y_i 及 y_{i+1}的状态对应，对应的操作如表 6.15 所示。当运算至最后一步时，乘积不再右移。这样的运算规则计算机很容易实现。

---

① 证明：$[-x]_补=-[x]_补$　(mod 2)

(1) 若$[x]_补=0.x_1x_2\cdots x_n$

则 $x=0.x_1x_2\cdots x_n$

所以 $-x=-0.x_1x_2\cdots x_n$

故$[-x]_补=1.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n}$(mod 2)　　(a)

又因为 $[x]_补=0.x_1x_2\cdots x_n$

所以 $-[x]_补=-0.x_1x_2\cdots x_n$

$\equiv 2-0.x_1x_2\cdots x_n$(mod 2)

$=1.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n}$　　(b)

比较(a)、(b)两式可得

$[-x]_补=-[x]_补$(mod 2)

证毕

(2)若$[x]_补=1.x_1x_2\cdots x_n$

则 $x=-(0.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n})$

所以 $-x=0.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n}$

故$[-x]_补=0.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n}$(mod 2)　　(c)

又因为 $[x]_补=1.x_1x_2\cdots x_n$

$\equiv -(0.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n})$ (mod 2)

所以 $-[x]_补=0.\bar{x}_1\bar{x}_2\cdots\bar{x}_n+2^{-n}$　　(d)

比较(c)、(d)两式可得

$[-x]_补=-[x]_补$(mod 2)

证毕

表 6.15　$y_i y_{i+1}$的状态对操作的影响

$y_i y_{i+1}$	$y_{i+1}-y_i$	操　作
0 0	0	部分积右移一位
0 1	1	部分积加$[x]_{补}$,再右移一位
1 0	-1	部分积加$[-x]_{补}$,再右移一位
1 1	0	部分积右移一位

应该注意的是,按比较法进行补码乘法时,像补码加、减法一样,符号位也一起参加运算。

例 6.21　已知$[x]_{补}=0.1101$,$[y]_{补}=0.1011$,求$[x\cdot y]_{补}$。

解:表 6.16 列出了例 6.21 的求解过程。

表 6.16　例 6.21 求$[x\cdot y]_{补}$的过程

部　分　积	乘数 y_n	附加位 y_{n+1}	说　明
00.0000 + 11.0011	0101$\underline{1}$	$\underline{0}$	初值$[z_0]_{补}=0$ $y_n y_{n+1}=10$,部分积加$[-x]_{补}$
11.0011 11.1001 11.1100 + 00.1101	 1010$\underline{1}$ 1101$\underline{0}$	 $\underline{1}$ $\underline{1}$	 →1 位,得$[z_1]_{补}$ $y_n y_{n+1}=11$,部分积→1 位,得$[z_2]_{补}$ $y_n y_{n+1}=01$,部分积加$[x]_{补}$
00.1001 00.0100 + 11.0011	11 1110$\underline{1}$	 $\underline{0}$	 →1 位,得$[z_3]_{补}$ $y_n y_{n+1}=10$,部分积加$[-x]_{补}$
11.0111 11.1011 + 00.1101	111 1111$\underline{0}$	 $\underline{1}$	 →1 位,得$[z_4]_{补}$ $y_n y_{n+1}=01$,部分积加$[x]_{补}$
00.1000	1111		最后一步不移位,得$[x\cdot y]_{补}$

故　$[x\cdot y]_{补}=0.10001111$

由表 6.16 可见,与校正法(参见表 6.13 和表 6.14)相比,Booth 算法的部分积仍取双符号位,乘数因符号位参加运算,故多取 1 位。

例 6.22　已知$[x]_{补}=1.0101$,$[y]_{补}=1.0011$,求$[x\cdot y]_{补}$。

解:表 6.17 列出了例 6.22 的求解过程。

表 6.17　例 6.22 求$[x \cdot y]_{补}$的过程

部分积	乘数 y_n	附加位 y_{n+1}	说　明
00.0000	1001$\underline{1}$	$\underline{0}$	
+ 00.1011			$y_ny_{n+1}=10$，部分积加$[-x]_{补}$
00.1011			
00.0101	1100$\underline{1}$	$\underline{1}$	→1 位，得$[z_1]_{补}$
00.0010	1110$\underline{0}$	$\underline{1}$	$y_ny_{n+1}=11$，部分积→1 位，得$[z_2]_{补}$
+ 11.0101			$y_ny_{n+1}=01$，部分积加$[x]_{补}$
11.0111	11		
11.1011	1111$\underline{0}$	$\underline{0}$	→1 位，得$[z_3]_{补}$
11.1101	1111$\underline{1}$	$\underline{0}$	$y_ny_{n+1}=00$，部分→1 位，得$[z_4]_{补}$
+ 00.1011			$y_ny_{n+1}=10$，部分积加$[-x]_{补}$
00.1000	1111		最后一步不移位，得$[x \cdot y]_{补}$

故 $[x \cdot y]_{补}=0.10001111$

由于比较法的补码乘法运算规则不受乘数符号的约束，因此，控制线路比较简明，在计算机中普遍采用。

(2) 补码比较法(Booth 算法)所需的硬件配置

图 6.9 是实现补码一位乘比较法乘法运算的基本硬件配置框图。

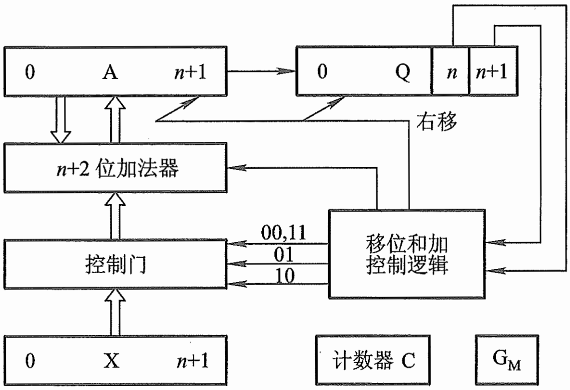

图 6.9　补码比较法运算基本硬件配置

图中 A、X、Q 均为 $n+2$ 位寄存器，其中 X 存放被乘数的补码(含两位符号位)，Q 存放乘数的补码(含最高 1 位符号位和最末 1 位附加位)，移位和加控制逻辑受 Q 寄存器末 2 位乘数控制。当其为 01 时，A、X 内容相加后 A、Q 右移一位；当其为 10 时，A、X 内容相减后 A、Q 右移一位。

计数器 C 用于控制逐位相乘的次数，G_M 为乘法标记。

(3) 补码比较法(Booth 算法)控制流程

补码一位乘比较法的控制流程图如图 6.10 所示。乘法运算前 A 寄存器被清零，作为初始部分积。Q 寄存器末位清零，作为附加位的初态。被乘数的补码存在 X 中(双符号位)，乘数的补码在 Q 高 $n+1$ 位中，计数器 C 存放乘数的位数 n。乘法开始后，根据 Q 寄存器末两位 Q_n、Q_{n+1} 的状态决定部分积与被乘数相加还是相减，或是不加也不减，然后按补码规则进行算术移位，这样重复 n 次。最后，根据 Q 的末两位状态决定部分积是否与被乘数相加(或相减)，或不加也不减，但不必移位，这样便可得到最后结果。补码乘法乘积的符号位在运算中自然形成。

需要说明的是，图中 $(A)-(X)\rightarrow A$ 实际是用加法器实现的，即 $(A)+(\overline{X}+1)\rightarrow A$。同理，Booth 运算规则也适用于整数补码。

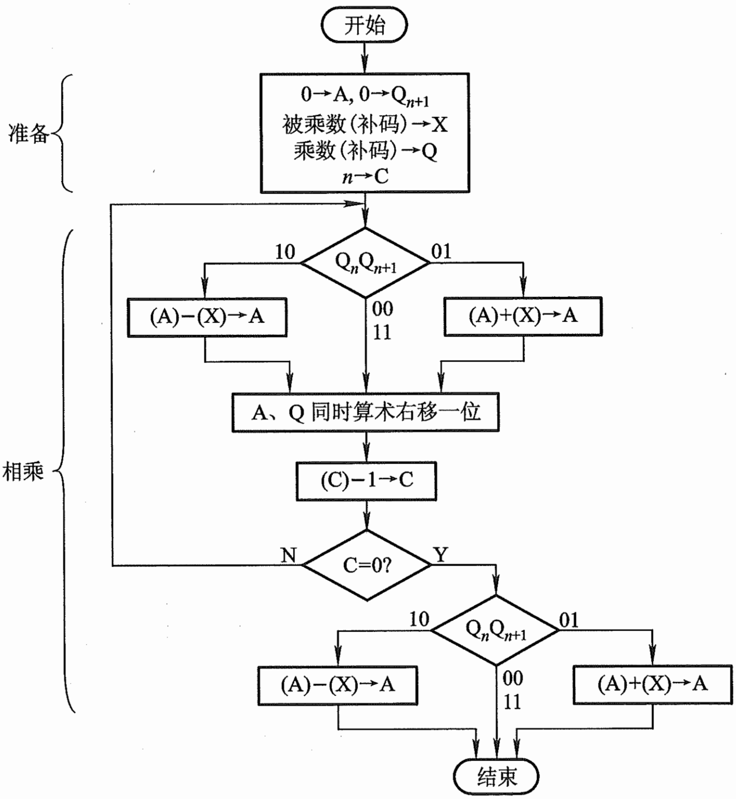

图 6.10　补码一位乘比较法控制流程图

为了提高乘法的运算速度，可采用补码两位乘。

(4) 补码两位乘

补码两位乘运算规则是根据补码一位乘的规则，把比较 y_ny_{n+1} 的状态应执行的操作和比较

$y_{n-1}y_n$ 的状态应执行的操作合并成一步得出的。

例如，$y_{n-1}y_ny_{n+1}$ 为 011，则第一步由 $y_ny_{n+1}=11$ 得出只作右移，即 $2^{-1}[z_i]_补$，第二步由 $y_{n-1}y_n=01$得出需作 $2^{-1}\{2^{-1}[z_i]_补+[x]_补\}$ 的操作，可改写为 $2^{-2}\{[z_i]_补+2[x]_补\}$，即最后结论为当 $y_{n-1}y_ny_{n+1}$为 011 时，完成 $2^{-2}\{[z_i]_补+2[x]_补\}$操作，同理可分析其余 7 种情况。表 6.18 列出了补码两位乘的运算规则。

表 6.18　补码两位乘的运算规则

判断位 $y_{n-1}y_ny_{n+1}$	操作内容
0 0 0	$[z_{i+1}]_补=2^{-2}[z_i]_补$
0 0 1	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+[x]_补\}$
0 1 0	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+[x]_补\}$
0 1 1	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+2[x]_补\}$
1 0 0	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+2[-x]_补\}$
1 0 1	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+[-x]_补\}$
1 1 0	$[z_{i+1}]_补=2^{-2}\{[z_i]_补+[-x]_补\}$
1 1 1	$[z_{i+1}]_补=2^{-2}[z_i]_补$

由表 6.18 可见，操作中出现加 $2[x]_补$和加 $2[-x]_补$，故除右移两位的操作外，还有被乘数左移一位的操作；而加 $2[x]_补$和加 $2[-x]_补$都可能因溢出而侵占双符号位，故部分积和被乘数采用 3 位符号位。

例 6.23　已知$[x]_补=0.0101$，$[y]_补=1.0101$，求$[x\cdot y]_补$。

解：表 6.19 列出了此例的求解过程。其中，乘数取 2 位符号位，外加 1 位附加位（初态为 0），即 11.01010，$[-x]_补=1.1011$ 取 3 位符号位为 111.1011。

表 6.19　例 6.23 补码两位乘求$[x\cdot y]_补$的过程

部　分　积	乘　　数	说　　明
000.0000 + 000.0101	1101$\underline{010}$	 判断位为 010，加$[x]_补$
000.0101 000.0001 + 000.0101	 0111$\underline{010}$	 →2 位 判断位为 010，加$[x]_补$
000.0110 000.0001 + 111.1011	01 1001$\underline{110}$	 →2 位 判断位为 110，加$[-x]_补$
111.1100	1001	最后一步不移位，得$[x\cdot y]_补$

故　$[x \cdot y]_{补} = 1.11001001$

由表 6.19 可见，与补码一位乘相比（参见表 6.16 和表 6.17），补码两位乘的部分积多取 1 位符号位（共 3 位），乘数也多取 1 位符号位（共 2 位），这是由于乘数每次右移 2 位，且用 3 位判断，故采用双符号位更便于硬件实现。可见，当乘数数值位为偶数时，乘数取 2 位符号位，共需作 $n/2$ 次移位，最多作 $n/2+1$ 次加法，最后一步不移位；当 n 为奇数时，可补 0 变为偶数位，以简化逻辑操作。也可对乘数取 1 位符号位，此时共进行 $n/2+1$ 次加法运算和 $n/2+1$ 次移位（最后一步移一位）。

对于整数补码乘法，其过程与小数补码乘法完全相同。为了区别于小数乘法，在书写上将符号位和数值位中间的“.”改为“,”即可。

6.3.4 除法运算

1. 分析笔算除法

以小数为例，设 $x = -0.1011$，$y = 0.1101$，求 x/y。

笔算除法时，商的符号心算而得：负正得负。其数值部分的运算如下面的竖式所示。

$$
\begin{array}{r|ll}
 & 0.1101 & \\
\hline
0.1101 & 0.10110 & \\
 & \underline{0.01101} & 2^{-1} \cdot y \\
 & 0.010010 & \\
 & \underline{0.001101} & 2^{-2} \cdot y \\
 & 0.00010100 & \\
 & \underline{0.00001101} & 2^{-4} \cdot y \\
 & 0.00000111 &
\end{array}
$$

所以　　商 $x/y = -0.1101$，余数 $= 0.00000111$

其特点可归纳如下：

① 每次上商都是由心算来比较余数（被除数）和除数的大小，确定商为“1”还是“0”。

② 每做一次减法，总是保持余数不动，低位补 0，再减去右移后的除数。

③ 上商的位置不固定。

④ 商符单独处理。

如果将上述规则完全照搬到计算机内，实现起来有一定困难，主要问题如下：

① 机器不能“心算”上商，必须通过比较被除数（或余数）和除数绝对值的大小来确定商值，即 $|x| - |y|$，若差为正（够减）上商 1，差为负（不够减）上商 0。

② 按照每次减法总是保持余数不动低位补 0，再减去右移后的除数这一规则，则要求加法器的位数必须为除数的两倍。仔细分析发现，右移除数可以用左移余数的方法代替，其运算结果是一样的，但对线路结构更有利。不过此刻所得到的余数不是真正的余数，只有将它乘上 2^{-n} 才是

真正的余数。

③ 笔算求商时是从高位向低位逐位求的，而要求机器把每位商直接写到寄存器的不同位置也是不可取的。计算机可将每一位商直接写到寄存器的最低位，并把原来的部分商左移一位，这样更有利于硬件实现。

综上所述便可得原码除法运算规则。

2. 原码除法

原码除法和原码乘法一样，符号位是单独处理的，下面以小数为例。

设
$$[x]_{原}=x_0.x_1x_2\cdots x_n$$
$$[y]_{原}=y_0.y_1y_2\cdots y_n$$

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

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^*。

即商符由两数符号位进行异或运算求得，商值由两数绝对值相除（x^*/y^*）求得。

小数定点除法对被除数和除数有一定的约束，即必须满足下列条件：

$$0<|被除数|\leqslant|除数|$$

实现除法运算时，还应避免除数为 0 或被除数为 0。前者结果为无限大，不能用机器的有限位数表示；后者结果总是 0，这个除法操作没有意义，浪费了机器时间。商的位数一般与操作数的位数相同。

原码除法中由于对余数的处理不同，又可分为恢复余数法和不恢复余数法（加减交替法）两种。

（1）恢复余数法

恢复余数法的特点是：当余数为负时，需加上除数，将其恢复成原来的余数。

由上所述，商值的确定是通过比较被除数和除数的绝对值大小，即 x^*-y^* 实现的，而计算机内只设加法器，故需将 x^*-y^* 操作变为 $[x^*]_{补}+[-y^*]_{补}$ 的操作。

例 6.24　已知 $x=-0.1011, y=-0.1101$，求 $\left[\frac{x}{y}\right]_{原}$。

解：由 $x=-0.1011, y=-0.1101$

得　$[x]_{原}=1.1011, x^*=0.1011$

$[y]_{原}=1.1101, y^*=0.1101, [-y^*]_{补}=1.0011$

表 6.20 列出了例 6.24 商值的求解过程。

表 6.20　例 6.24 恢复余数法的求解过程

被除数（余数）	商	说　明
0.1011	0.0000	
+ 1.0011		$+[-y^*]_{补}$（减去除数）

续表

被除数(余数)	商	说明
1.1110 + 0.1101	0	余数为负,上商“0” 恢复余数+$[y^*]_{补}$
0.1011 1.0110 + 1.0011	0	被恢复的被除数 ←1 位 +$[-y^*]_{补}$(减去除数)
0.1001 1.0010 + 1.0011	01 01	余数为正,上商“1” ←1 位 +$[-y^*]_{补}$(减去除数)
0.0101 0.1010 + 1.0011	011 011	余数为正,上商“1” ←1 位 +$[-y^*]_{补}$(减去除数)
1.1101 + 0.1101	0110	余数为负,上商“0” 恢复余数+$[y^*]_{补}$
0.1010 1.0100 + 1.0011	0110	被恢复的余数 ←1 位 +$[-y^*]_{补}$(减去除数)
0.0111	01101	余数为正,上商“1”

故　商值为 0.1101

商的符号位为

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

所以

$$\left[\frac{x}{y}\right]_{原} = 0.1101$$

由此例可见,共左移(逻辑左移)4 次,上商 5 次,第一次上的商在商的整数位上,这对小数除法而言,可用它作溢出判断。即当该位为“1”时,表示此除法溢出,不能进行,应由程序进行处理;当该位为“0”时,说明除法合法,可以进行。

在恢复余数法中,每当余数为负时,都需恢复余数,这就延长了机器除法的时间,操作也很不规则,对线路结构不利。加减交替法可克服这些缺点。

(2) 加减交替法

加减交替法又称不恢复余数法,可以认为它是恢复余数法的一种改进算法。

分析原码恢复余数法得知：

当余数 $R_i>0$ 时，可上商"1"，再对 R_i 左移一位后减除数，即 $2R_i-y^*$。

当余数 $R_i<0$ 时，可上商"0"，然后先做 R_i+y^*，即完成恢复余数的运算，再做 $2(R_i+y^*)-y^*$，即 $2R_i+y^*$。

可见，原码恢复余数法可归纳如下：

当 $R_i>0$，商上"1"，做 $2R_i-y^*$ 的运算。

当 $R_i<0$，商上"0"，做 $2R_i+y^*$ 的运算。

这里已经看不出余数的恢复问题了，而只是做加 y^* 或减 y^*，因此，一般将其称为加减交替法或不恢复余数法。

例 6.25　已知 $x=-0.1011, y=0.1101$，求 $\left[\frac{x}{y}\right]_{原}$。

解： 由 $x=-0.1011, y=0.1101$

得　$[x]_{原}=1.1011, x^*=0.1011$

$[y]_{原}=0.1101, y^*=0.1101, [-y^*]_{补}=1.0011$

表 6.21 列出了此例商值的求解过程。

表 6.21　例 6.25 加减交替法的求解过程

被除数（余数）	商	说　明
0.1011	0.0000	
+1.0011		$+[-y^*]_{补}$（减除数）
1.1110	0	余数为负，上商"0"
1.1100	0	←1 位
+0.1101		$+[y^*]_{补}$（加除数）
0.1001	01	余数为正，上商"1"
1.0010	01	←1 位
+1.0011		$+[-y^*]_{补}$（减除数）
0.0101	011	余数为正，上商"1"
0.1010	011	←1 位
+1.0011		$+[-y^*]_{补}$（减除数）
1.1101	0110	余数为负，上商"0"
1.1010	0110	←1 位
+0.1101		$+[y^*]_{补}$（加除数）
0.0111	01101	余数为正，上商"1"

商的符号位为

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

所以
$$\left[\frac{x}{y}\right]_{原} = 1.1101$$

分析此例可见,n 位小数的除法共上商 $n+1$ 次(第一次商用来判断是否溢出),左移(逻辑左移)n 次,可用移位次数判断除法是否结束。倘若比例因子选择恰当,除法结果不溢出,则第一次商肯定是0。如果省去这位商,只需上商 n 次即可,此时除法运算一开始应将被除数左移一位减去除数,然后再根据余数上商。读者可以自己练习。

需要说明一点,表6.21中操作数也可采用双符号位,此时移位操作可按算术左移处理,最高符号位是真正的符号,次高位符号位在移位时可被第一数值位占用。

(3) 原码加减交替法所需的硬件配置

图6.11是实现原码加减交替法运算的基本硬件配置框图。

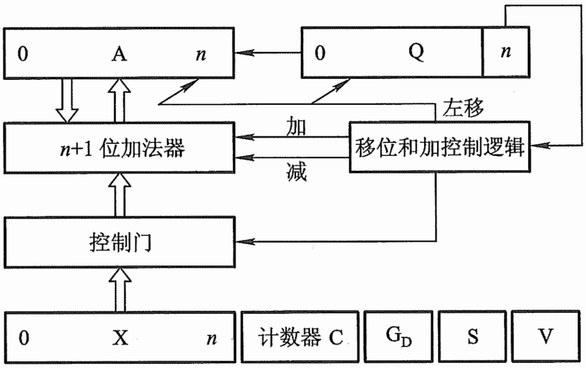

图6.11 原码加减交替法运算的基本硬件配置

图中A、X、Q均为 $n+1$ 位寄存器,其中A存放被除数的原码,X存放除数的原码。移位和加控制逻辑受Q的末位 Q_n 控制($Q_n=1$ 做减法,$Q_n=0$ 做加法),计数器C用于控制逐位相除的次数 n,G_D 为除法标记,V为溢出标记,S为商符。

(4) 原码加减交替法控制流程

图6.12为原码加减交替法控制流程图。

除法开始前,Q寄存器被清零,准备接收商,被除数的原码放在A中,除数的原码放在X中,计数器C中存放除数的位数 n。除法开始后,首先通过异或运算求出商符,并存于S。接着将被除数和除数变为绝对值,然后开始用第一次上商判断是否溢出。若溢出,则置溢出标记V为1,停止运算,进行中断处理,重新选择比例因子;若无溢出,则先上商,接着A、Q同时左移一位,然后再根据上一次商值的状态,决定是加还是减除数,这样重复 n 次后,再上最后一次商(共上商 $n+1$次),即得运算结果。

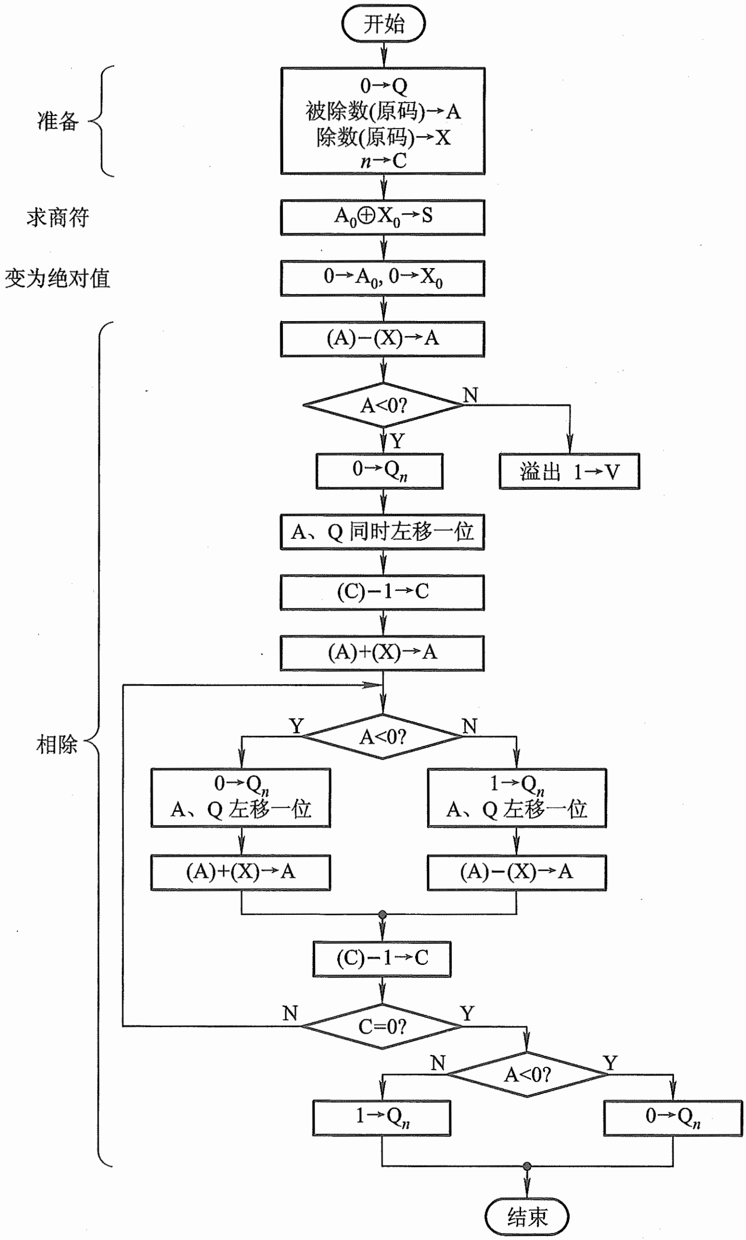

图 6.12　原码加减交替法控制流程图

对于整数除法，要求满足以下条件：

$$0 < |除数| \leqslant |被除数|$$

因为这样才能得到整数商。通常在做整数除法前，先要对这个条件进行判断，若不满足上述

条件,机器发出出错信号,程序要重新设定比例因子。

上述讨论的小数除法完全适用于整数除法,只是整数除法的被除数位数可以是除数的两倍,且要求被除数的高 n 位要比除数(n 位)小,否则即为溢出。如果被除数和除数的位数都是单字长,则要在被除数前面加上一个字的 0,从而扩展成双倍字长再进行运算。

为了提高除法速度,可采用阵列除法器,有关内容参见附录 6B。

3. 补码除法

与补码乘法类似,也可以用补码完成除法操作。补码除法也分恢复余数法和加减交替法,后者用得较多,在此只讨论加减交替法。

(1) 补码加减交替法运算规则

补码除法的符号位和数值部分是一起参加运算的,因此在算法上不像原码除法那样直观,主要需要解决 3 个问题:① 如何确定商值;② 如何形成商符;③ 如何获得新的余数。

① 欲确定商值,必须先比较被除数和除数的大小,然后才能求得商值。

- 比较被除数(余数)和除数的大小

补码除法的操作数均为补码,其符号又是任意的,因此要比较被除数$[x]_{补}$和除数 $[y]_{补}$的大小就不能简单地用 $[x]_{补}$减去 $[y]_{补}$。实质上比较 $[x]_{补}$和 $[y]_{补}$的大小就是比较它们所对应的绝对值的大小。同样在求商的过程中,比较余数 $[R_i]_{补}$与除数 $[y]_{补}$的大小,也是比较它们所对应的绝对值的大小。这种比较的算法可归纳为以下两点。

第一,当被除数与除数同号时,做减法,若得到的余数与除数同号,表示“够减”,否则表示“不够减”。

第二,当被除数与除数异号时,做加法,若得到的余数与除数异号,表示“够减”,否则表示“不够减”。

此算法如表 6.22 所示。

表 6.22 比较算法表

比较$[x]_{补}$与$[y]_{补}$的符号	求余数	比较$[R_i]_{补}$与$[y]_{补}$的符号
同号	$[x]_{补}-[y]_{补}$	同号,表示“够减”
异号	$[x]_{补}+[y]_{补}$	异号,表示“够减”

- 商值的确定

补码除法的商也是用补码表示的,如果约定商的末位用“恒置 1”的舍入规则,那么除末位商外,其余各位的商值对正商和负商而言,上商规则是不同的。因为在负商的情况下,除末位商以外,其余任何一位的商与真值都正好相反。因此,上商的算法可归纳为以下两点。

第一,如果$[x]_{补}$与$[y]_{补}$同号,商为正,则“够减”时上商“1”,“不够减”时上商“0”(按原码规则上商)。