第 3 章　运算方法与运算器

计算机中的运算包括算术运算和逻辑运算两大类。算术运算是指带符号数的加法、减法、乘法和除法运算。由于在计算机中数值有定点和浮点两种表示方式，因此算术运算应有定点数的算术运算和浮点数的算术运算之分。逻辑运算是指不考虑进位的“位对位”的运算，参加逻辑运算的操作数，常被称作逻辑数。一般来说，逻辑数是不带符号的整数，广义的逻辑运算可定义为非算术运算。本章主要讨论各种运算的运算方法及其实现。运算器就是计算机中完成各种运算的一个必不可少的重要部件。

3.1　定点加、减法运算及其实现

在计算机中进行定点加、减法运算基本上都是采用补码，极少数机器中也可采用反码，但是绝对没有采用原码进行加、减法运算的。原因很简单，第 2 章中已经说到，如果采用原码进行加、减法运算，操作太复杂。单就加法运算而言，不仅要有一个加法器，而且还要有减法器，不仅要对数码位进行加、减法运算，还要对符号位进行特殊处理。因此，本节中只讨论最常用的补码加、减法运算。

3.1.1　补码加、减法运算方法

对于补码加、减法运算需要证明如下的公式

$$[X+Y]_{补} = [X]_{补} + [Y]_{补}$$
$$[X-Y]_{补} = [X]_{补} + [-Y]_{补}$$

下面分 4 种情况证明：

(1) $X>0, Y>0$，显然 $X+Y>0$。

根据定点小数的补码定义

$$[X]_{补} = X$$
$$[Y]_{补} = Y$$

所以

$$[X]_{补} + [Y]_{补} = X+Y = [X+Y]_{补}$$

(2) $X>0, Y<0$。

根据定点小数的补码定义

$$[X]_{补} = X$$
$$[Y]_{补} = 2+Y$$
$$[X]_{补} + [Y]_{补} = 2+X+Y$$

那么，如果 $X+Y>0$，则

$$[X]_{补}+[Y]_{补}=X+Y=[X+Y]_{补} \quad (\text{mod } 2)$$

如果 $X+Y<0$,则

$$[X]_{补}+[Y]_{补}=2+(X+Y)=[X+Y]_{补} \quad (\text{mod } 2)$$

(3) $X<0, Y>0$。

证明同(2)。

(4) $X<0, Y<0$,显然 $X+Y<0$。

根据定点小数的补码定义

$$[X]_{补}=2+X$$
$$[Y]_{补}=2+Y$$

则

$$[X]_{补}+[Y]_{补}=2+X+2+Y=2+(2+X+Y)=2+X+Y=[X+Y]_{补} \quad (\text{mod } 2)$$

所以

$$[X+Y]_{补}=[X]_{补}+[Y]_{补}$$

对于$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$的证明则更简单:因为$[X-Y]_{补}=[X+(-Y)]_{补}$,利用补码加法的公式得,$[X+(-Y)]_{补}=[X]_{补}+[-Y]_{补}$。

以上是根据定点小数的定义所作的证明。对于定点整数来说,证明方法完全相同,只是整数的补码以 2^{n+1} 为模,不需要另行证明。

例 1 已知:$X=(+0.125)_{10}$,$Y=(+0.625)_{10}$,求:$X+Y=?$ $X-Y=?$

解:$X=+0.0010$ $Y=+0.1010$

$[X]_{原}=0.0010$ $[Y]_{原}=0.1010$

$[X]_{补}=0.0010$ $[Y]_{补}=0.1010$

$[-Y]_{补}=1.0110$

$$\begin{array}{r} [X]_{补}=0.0010 \\ +\quad [Y]_{补}=0.1010 \\ \hline [X+Y]_{补}=0.1100 \end{array}$$

所以

$$X+Y=+0.1100=(+0.75)_{10}$$

$$\begin{array}{r} [X]_{补}=0.0010 \\ +\quad [-Y]_{补}=1.0110 \\ \hline [X-Y]_{补}=1.1000 \end{array}$$

所以

$$[X-Y]_{补}=1.1000$$
$$[X-Y]_{原}=1.1000$$
$$X-Y=-0.1000=(-0.5)_{10}$$

例 2 已知:$X=-0.0625$,$Y=-0.75$,求:$X+Y=?$ $X-Y=?$

解:$X=-0.0001$ $Y=-0.1100$

$[X]_{原}=1.0001$ $[Y]_{原}=1.1100$

$[X]_{补}=1.1111$ $[Y]_{补}=1.0100$

$[-Y]_{补}=0.1100$

$$
\begin{array}{rl}
 & [X]_{补}=1.1111 \\
+ & [Y]_{补}=1.0100 \\
\hline
 & [X+Y]_{补}=1.0011 \\
 & [X+Y]_{原}=1.1101
\end{array}
$$

所以

$$X+Y=-0.1101=(-0.8125)_{10}$$

$$
\begin{array}{rl}
 & [X]_{补}=1.1111 \\
+ & [-Y]_{补}=0.1100 \\
\hline
 & [X-Y]_{补}=0.1011 \\
 & [X-Y]_{原}=0.1011
\end{array}
$$

所以

$$X-Y=+0.1011=(+0.6875)_{10}$$

例 3　已知：$X=+0.25, Y=-0.625$，求：$X+Y=?$　$X-Y=?$

解：$X=+0.0100$　　$Y=-0.1010$

$[X]_{原}=0.0100$　　$[Y]_{原}=1.1010$

$[X]_{补}=0.0100$　　$[Y]_{补}=1.0110$

$[-Y]_{补}=0.1010$

$$
\begin{array}{rl}
 & [X]_{补}=0.0100 \\
+ & [Y]_{补}=1.0110 \\
\hline
 & [X+Y]_{补}=1.1011
\end{array}
$$

所以

$$
\begin{array}{l}
[X+Y]_{原}=1.0110 \\
X+Y=-0.0110=(-0.375)_{10}
\end{array}
$$

$$
\begin{array}{rl}
 & [X]_{补}=0.0100 \\
+ & [-Y]_{补}=0.1010 \\
\hline
 & [X-Y]_{补}=0.1110
\end{array}
$$

所以

$$X-Y=+0.1110=(+0.875)_{10}$$

例 4　已知：$X=-0.125, Y=+0.5625$，求：$X+Y=?$　$X-Y=?$

解：$X=-0.0010$　　$Y=+0.1001$

$[X]_{原}=1.0010$　　$[Y]_{原}=0.1001=[Y]_{补}$

$[X]_{补}=1.1110$　　$[-Y]_{补}=1.0111$

$$
\begin{array}{rl}
 & [X]_{补}=1.1110 \\
+ & [Y]_{补}=0.1001 \\
\hline
 & [X+Y]_{补}=0.0111
\end{array}
$$

所以

$$X+Y=+0.0111=(+0.4375)_{10}$$

$$\begin{array}{rl} & [X]_{补}=1.1110 \\ + & [-Y]_{补}=1.0111 \\ \hline & [X-Y]_{补}=1.0101 \end{array}$$

所以

$$[X-Y]_{原}=1.1011$$

$$X-Y=(-0.6875)_{10}$$

例 5 已知：$X=+25, Y=-5$，求：$X+Y=?$ $X-Y=?$

解：$X=+11001$ $\quad Y=-00101$

$[X]_{原}=011001$ $\quad [Y]_{原}=100101$

$[X]_{补}=011001$ $\quad [Y]_{补}=111011$

$[-Y]_{补}=000101$

$$\begin{array}{rl} & [X]_{补}=011001 \\ + & [Y]_{补}=111011 \\ \hline & [X+Y]_{补}=010100 \end{array}$$

所以

$$X+Y=+010100=(+20)_{10}$$

$$\begin{array}{rl} & [X]_{补}=011001 \\ + & [-Y]_{补}=000101 \\ \hline & [X-Y]_{补}=011110 \end{array}$$

所以

$$X-Y=+11110=(+30)_{10}$$

例 6 已知：$X=-9, Y=+20$，求：$X+Y=?$ $X-Y=?$

解：$X=-01001$ $\quad Y=+10100$

$[X]_{原}=101001$ $\quad [Y]_{原}=010100=[Y]_{补}$

$[X]_{补}=110111$ $\quad [-Y]_{补}=101100$

$$\begin{array}{rl} & [X]_{补}=110111 \\ + & [Y]_{补}=010100 \\ \hline & [X+Y]_{补}=001011 \end{array}$$

所以

$$X+Y=+01011=(+11)_{10}$$

$$\begin{array}{rl} & [X]_{补}=110111 \\ + & [-Y]_{补}=101100 \\ \hline & [X-Y]_{补}=100011 \end{array}$$

所以

$$[X-Y]_{原}=111101$$

$$X-Y=-11101=(-29)_{10}$$

从上述的例子可得出如下的重要结论：

(1) 采用补码进行加、减法运算可变减法为加法，使得运算器中只需要设置一个加法器，便可完成加、减法运算。

(2) 补码加法运算中，符号位像数码位一样参加加法运算，能自然得到运算结果的正确符号。

(3) 定点小数的补码加法运算以 2 为模；定点整数的补码加法运算以 $2n+1$ 为模。即符号位向更高位的进位自然丢失，并不影响运算结果的正确性。

3.1.2　定点加、减法运算中的溢出问题

所谓“运算溢出”，是指运算结果大于机器所能表示的最大正数或者小于机器所能表示的最小负数。这就是说，运算溢出只对带符号数的运算有效。

下面举例说明补码加法运算中什么情况下会产生运算溢出。

例 1　两个正数相加。

$$\begin{array}{rr} & [X]_{补}=0.1010 \\ + & [Y]_{补}=0.1001 \\ \hline & [X+Y]_{补}=1.0011 \end{array}$$

例 2　两个负数相加。

$$\begin{array}{rr} & [X]_{补}=1.0001 \\ + & [Y]_{补}=1.0101 \\ \hline & [X+Y]_{补}=0.0110 \end{array}$$

从上面两个例子中可看出，两个正数相加结果为负数，两个负数相加结果为正数，这显然是错误的，造成这种错误的原因是由于运算产生子溢出。除此之外，正数减负数，或负数减正数，也可能产生运算溢出。溢出是一种错误，计算机在运算过程中必须能发现这种溢出现象，并进行必要的处理，否则将带来严重后果。常用的判定溢出方法有以下两种。

1. 采用变形补码判定溢出

“变形补码”是采用 2 位符号位的补码，常记作$[X]'_{补}$。

上述两个例子中，如果采用变形补码来进行加法运算，则：

例 1

$$\begin{array}{rr} & [X]'_{补}=00.1010 \\ + & [Y]'_{补}=00.1001 \\ \hline & [X+Y]'_{补}=01.0011 \end{array}$$

例 2

$$\begin{array}{rr} & [X]'_{补}=11.0001 \\ + & [Y]'_{补}=11.0111 \\ \hline & [X+Y]'_{补}=10.1000 \end{array}$$

例 1 中,运算结果的两位符号位为 01,表示产生了正溢出。例 2 中,运算结果的两位符号位为 10,表示产生了负溢出。

不溢出的情况分别为:

$$\begin{array}{r} [X]'_{补}=00.0101 \\ +\quad [Y]'_{补}=00.0111 \\ \hline [X+Y]'_{补}=00.1100 \end{array} \qquad \begin{array}{r} [X]'_{补}=11.1011 \\ +\quad [Y]'_{补}=11.0111 \\ \hline [X+Y]'_{补}=11.0010 \end{array}$$

这就是说,采用变形补码进行加、减法运算时,运算结果的两位符号位应当相同。若两位符号位同时为 00,表示结果是一个正数;反之,若两位符号位同时为 11,表示结果是一个负数。若两位符号位不同,则表示运算产生了溢出,且左边一位表示结果的正确符号,因此两位符号位为 01,表示运算结果的正确符号应该为正,即产生了正溢出;与此相反,若两位符号位为 10,则表示运算结果的正确符号应该为负,即产生了负溢出。

这种判定溢出的办法简单,而且容易实现,只需要在两位符号位上增设一个半加器就行了,但是运算器要增加一位字长,或者要降低一位运算精度。

2. 利用符号位的进位信号判定溢出

对于带符号数,最高位是符号位,如果将最高数码位向符号位的进位叫做“进位入”,记作 C_{n-1};将符号位向更高位的进位叫做“进位出”,记作 C_n。

还是用上述的两个例子来看。

$$\begin{array}{r} [X]_{补}=0.1010 \\ +\quad [Y]_{补}=0.1001 \\ \hline [X+Y]_{补}=1.0011 \end{array} \qquad \begin{array}{r} [X]_{补}=1.0001 \\ +\quad [Y]_{补}=1.01111 \\ \hline 0.1000 \end{array}$$

$[X+Y]_{补}=1.0011$ 的进位:$C_{n-1}=1$,$C_n=0$

0.1000 的进位:$C_{n-1}=0$,$C_n=1$

例 1 中:$C_n=0$,$C_{n-1}=1$,表示产生了正溢出。

例 2 中:$C_n=1$,$C_{n-1}=0$,表示产生了负溢出。

只有 $C_n=C_{n-1}=0$ 或 $C_n=C_{n-1}=1$,才表示运算未溢出。

因此可以由 $C_n \neq C_{n-1}$ 来判定运算是否产生了溢出,这种判定溢出方式不降低运算精度,只需增设一个半加器来产生溢出标志,即

$$\text{OF(溢出标志)} = C_n \oplus C_{n-1}$$

请注意一个问题,机器本身无法知道加、减法运算的操作数是带符号数还是无符号数,而总是把它当成带符号的操作数处理,并判定是否产生了溢出。溢出标志当前是否有效,应由程序员自己来判定。也就是说,如果当前进行的是两个无符号数的加减法运算,则尽管 OF=1,也不表示运算产生了溢出。

3.1.3 补码加、减法运算的实现

如 3.1.2 节所述,采用补码进行加、减法运算只需要设置一个加法器。采用串行进位方

式的 n 位加法器的逻辑结构如图 3.1 所示。

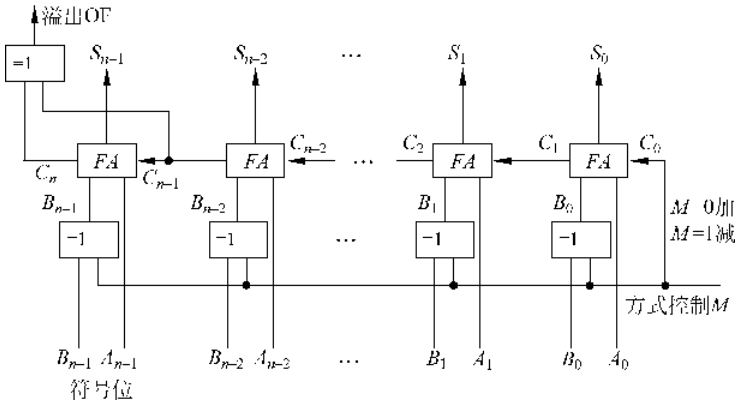


图 3.1　串行进位方式的 n 位加法器逻辑结构

从图 3.1 中可看出，采用串行进位方式的 n 位并行加法器的主体是 n 个全加器（FA_{n-1}～FA_0），其进位信号 C_i 从低位向高位逐位串行传送，两个 n 位长的操作数的补码（$A_{n-1}A_{n-2}\cdots A_1A_0$ 和 $B_{n-1}B_{n-2}\cdots B_1B_0$）连同向最低位的进位信号（$C_{-1}$）一起同时送到 n 位全加器的输入端，经过一定的运算时间，最后可得到 n 位的运算结果（$S_{n-1}S_{n-2}\cdots S_1S_0$），其中最高位是符号位，低端的（$n-1$）位是其数码位。图中左上方的半加器用来判定加减法运算是否产生了溢出。

图 3.1 中右下方的 M 是方式控制信号，当 $M=0$ 时，操作数 B 经半加器延迟后直送各位全加器的左输入端，并且 $C_{-1}=0$，完成加法运算功能（$[S]_{补}=[A]_{补}+[B]_{补}=[A+B]_{补}$）；当 $M=1$ 时，操作数 B 经半加器取反后送各位全加器的左输入端，并且 $C_{-1}=1$，完成减法运算功能（$[S]_{补}=[A]_{补}+[\overline{B}]_{补}+1=[A_i]+[-B_{补}]=[A-B]_{补}$）。

1. 全加器的结构

一位全加器（FA_i）可有多种不同的实现方案。从原理上说，凡是能满足表 3.1 所示真值表的逻辑电路均可称作全加器。

表 3.1　全加器真值表

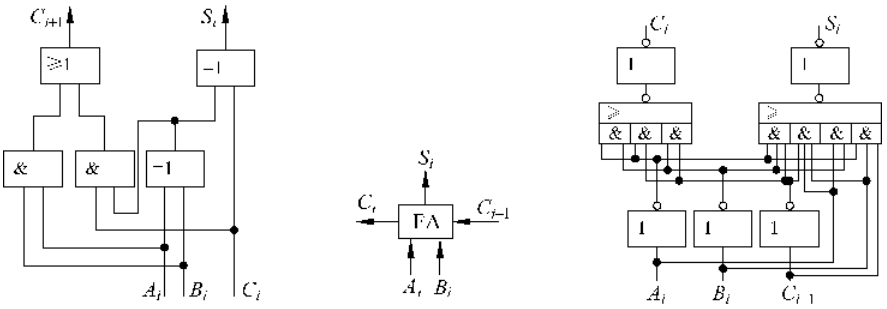
A_i	B_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

根据真值表,可写出一位全加器的逻辑表达式

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

$$C_i = A_iB_i = B_iC_{i-1} + A_iC_{i-1}$$

两种比较常用的全加器逻辑电路图如图 3.2 所示。



(a) 逻辑图 (b) 符号表示 (c) 逻辑图

图 3.2 一位全加器逻辑电路图

采用这两种逻辑结构的全加器,使用的逻辑器件有非门(反相器)、与非门、与或非门和半加器(异或门)。为了简单起见,假定前 3 种逻辑门的平均延迟时间均为 $1t_y$,半加器的延迟时间为 $3t_y$。于是便可计算出每位全加器完成一位全加运算所需要的延迟时间,进而可计算出采用两种方案的 n 位串行进位加法器的总延迟时间。

对于方案(a)的全加器,从 A_i,B_i 和 C_{i-1} 输入到输出 S_i 需要 $6t_y$;从 A_i,B_i 和 C_{i-1} 输入到输出 C_i 需要 $5t_y$。但是如果 A_i,B_i 早已输入,以 C_{i-1} 输入到输出 C_i 则只需要 $2t_y$。

对于方案(b)的全加器,从 A_i,B_i 和 C_{i-1} 输入到输出 S_i 和 C_i 均需要 $2t_y$。

2. 计算整个加法器的运算时间

采用串行进位方式的 n 位加法器(如图 3.1 所示)中,如果各位全加器采用图 3.2(a)的方案来实现,那么对于最低位(FA_0)来说,从 $A_{n-1}\sim A_0$,$B_{n-1}\sim B_0$ 和 C_{-1} 输入到输出全加和 S_0 需要 $6t_y$,而到输出进位信号 C_0,则需要 $5t_y$。至于高端的各位情况就有所不同。因为由 $B_{n-1}\sim B_1$ 进入半加器的延迟时间与 B_0 是覆盖的,而且在全加器(FA_i)内部 A_iB_i 进入的半加器的延迟时间也是相互覆盖的,所以对于高端的各位来说,每位的延迟时间只需要考虑从 C_{i-1} 至 C_i 所需要的 $2t_y$ 的延迟时间,最后再加上产生溢出标志的半加器的延迟时间,即为整个加法器的总延迟时间,如果用 t_a 来表示,则

$$t_a = 3t_y + 3t_y + n \times 2t_y + 3t_y = (2n+9)t_y$$

而如果各位全加器采用图 3.2(b)的方案来实现,因为每一位全加器从输入到输出 S_i 和 C_i 所需的延迟时间均为 $2t_y$,则整个加法器的总延迟时间,如果用 t_b 表示,则

$$t_b = 3t_y + n \times 2t_y + 3t_y = (2n+6)t_y$$

从 t_a 和 t_b 可看出,不管是采用方案(a)还是方案(b)的全加器电路,整个加法器的总延迟时间,主要取决于 n 的大小,而与各位全加器本身的延迟时间关系不大,只是相差 $3t_y$。n 越大(即加法器字长越大)所需的总延迟时间就越长,即加法运算速度越慢,原因就在于其进位信号是从最低位开始逐位向高位串行传送。从而可得出重要结论:提高加法器运算速度

的关键在于加快进位信号的传送速度，而不在于全加器电路方案的选择。

3.2　定点乘法运算及其实现

定点乘法运算与定点加、减法运算不同，可采用原码进行，原码乘法又有原码一位乘法和原码两位乘法。还可采用补码进行，补码乘法也有补码一位乘法和补码两位乘法之分，更进一步提高乘法运算速度还可采用阵列乘法器来实现。

3.2.1　原码一位乘法

首先要给原码乘法下个定义："符号位单独运算，将两个操作数的数码位相乘，最后给乘积冠以正确符号。"从最低位开始，每次取一位乘数与被乘数相乘，最后累加结果，称做"原码一位乘法"。

原码一位乘法的操作过程与十进制乘法运算的过程很类似，下面通过一个具体例子来说明。

例如，已知 $X=+0.1101$，$Y=-0.1011$，求 $Z=X\times Y$。

解：

$$[X]_{原}=0.1101$$

$$[Y]_{原}=1.1101$$

先进行符号位运算，$Z_f=X_f\oplus Y_f=0\oplus 1=1$。

后将两数码位的原码相乘，操作过程如下

$$\begin{array}{r}
0.1101 \\
\times\ 0.1011 \\
\hline
1101 \quad \cdots\cdots \text{第①次部分积} \\
1101 \quad \cdots\cdots \text{第②次部分积} \\
0000 \quad \cdots\cdots \text{第③次部分积} \\
+1101 \quad \cdots\cdots \text{第④次部分积} \\
\hline
0.10001111 \quad \cdots\cdots \text{乘积(4 次部分积之和)}
\end{array}$$

这是二进制乘法的手算过程。这一过程如果在计算机中实现，存在两个问题：其一是两个 n 位数相乘，需要 $2n$ 位的加法器，这不合算；其二是 n 次部分积一次累加，实现有困难。这两个问题只需要操作上稍作改动就可以得到满意的解决。在计算机中实现上述乘法过程的具体做法有两种：一种是串行方式，每得到一次部分积，立即与上次部分积相加，然后将结果右移一位，待 n 次的"相加右移"操作结束，乘法运算过程也告结束；另一种是并行方式，利用全加器排成阵列，实现部分积并行快速计算。在本节主要介绍串行方式。

用上面这个例子，在计算机内实现原码一位乘法的操作过程如下

	部分积	$Y_0Y_1Y_2Y_3Y_4$	
	0.0000	0. 1 0 1 1	$Y_4=1$
$+X$	0.1101		
	0.1101	0 1 0 1 1	
	0.0110	1 0 1 0 1	$Y_3=1$
$+X$	0.1101		
	1.0011	1 0 1 0 1	
	0.1001	1 1 0 1 0	$Y_2=0$
$+0$	0.0000		
	0.1001	1 1 0 1 0	
	0.0100	1 1 1 0 1	$Y_1=1$
$+X$	0.1101		
	1.0001	1 1 1 0 1	
	0.1000	1 1 1 1 0	

可以看出,上例中经过 4 次相加(+X 或+0)右移的操作,在粗黑线左方可得到正确的运算结果,最后应给它冠以正确的符号 1,所以

$$[Z]_{原} = 1.10001111$$

$$Z = -0.1000111$$

从上述过程来看,两个 n 位带符号数相乘需要一个 $n+1$ 位加法器。并且需要两个 $n+1$ 位寄存器,操作前分别存放部分积和乘数 Y,操作后分别存放最后乘积的高 n 位和低 n 位,并要求这两个寄存器能连接起来一起进行右移操作。

上述原码一位乘法的操作过程可用流程图描述,如图 3.3 所示。

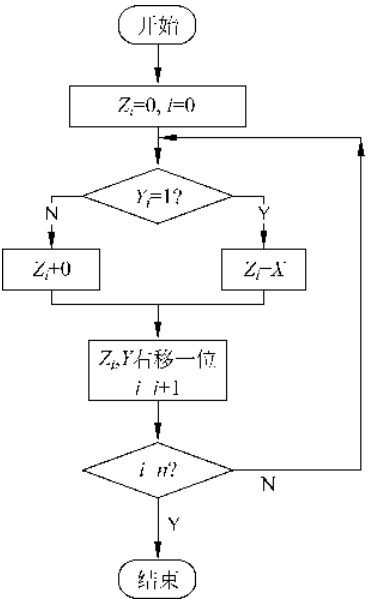


图 3.3 原码一位乘法操作流程图

实现原码一位乘法的乘法器结构框图如图 3.4 所示。

从图 3.4 中可以看出,组成该乘法器的主体是一个 $n+1$ 位的加法器,$n+1$ 位寄存器 R_0 用来存放各次部分积和累加运算的结果,最后存放的是乘积的高 $n+1$ 位;R_1 寄存器也是 $n+1$ 位,用来存放乘数(Y),最后在 R_1 的高 n 位上存放的是乘积的低 n 位;R_0 和 R_1 寄存器连接起来右移,R_1 寄存器的最低位(Y_0)总是给出参加本次运算的一位乘数;$n+1$ 位的寄存器 R_2 用来存放被乘数 X。计数器中事先存放计数值($-n$)。每进行一次"相加右移"操作,计数器+1,待计数器内容为 0,将触发器 C_x 清 0,一次原码一位乘法运算结束。图中部的半加器专用于符号位的运算。

如果 $n+1$ 位加法器进行一次加法运算的时间为 t_a,两个寄存器一次右移操作的时间为 t_s,那么原码一位乘

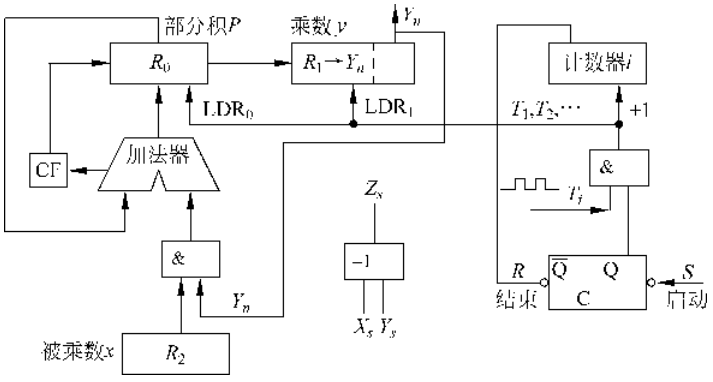


图 3.4　原码一位乘法的乘法器结构框图

法所需要的时间为 $t_m = n(t_a + t_s)$。

显然，提高加法运算的速度（t_a 减小）对于加快乘法运算速度无疑是有效的。

关于原码两位乘法，基本原理类似，只是从最低位开始，每次取两位乘数与被乘数相乘，得到一次部分积，与上次部分积相加后右移 2 位。显然，两个 n 位数相乘，采用原码两位乘法时，只需要进行 $n/2$ 次“相加右移”的操作，因此，乘法运算速度可提高一倍。有关原码两位乘法的具体操作过程不再详细讨论。

3.2.2　补码一位乘法

原码乘法存在两个明显的缺点：其一是符号位需要单独运算；其二是最后要根据符号位运算的结果给乘积冠以正确符号。尤其是对于采用补码存储的计算机，从存储器或寄存器中取得的操作数均为补码，需要先变换成原码才能进行乘法运算，乘积又要变换成补码才能存储起来。这正是需要推出补码乘法的原因。

“补码乘法”是指采用操作数的补码进行乘法运算，最后乘积仍为补码，能自然得到乘积的正确符号。从乘数的最低位开始，每次取一位乘数与被乘数相乘，经过（$n+1$）次“相加右移”操作完成乘法运算的过程被称作“补码一位乘法”。

实际中较常用的“补码一位乘法”是由英国的 Booth 夫妇提出来的“比较法”，所以又可称作“Booth 法”。

下面来讨论“比较法”的由来及其操作过程。

1. 补码与真值之间的关系

根据补码定义：

$$\text{对于正数}\quad [Y]_{补} = Y = 0.Y_1Y_2\cdots Y_n \qquad (Y_0 = 0)$$

$$\text{对于负数}\quad [Y]_{补} = 2 + Y = 1.Y_1Y_2\cdots Y_n \qquad (Y_0 = 1)$$

所以

$$\begin{aligned} Y &= 1.Y_1Y_2\cdots Y_{n-2} = 1 + 0.Y_1Y_2\cdots Y_{n-2} \\ &= -1 + 0.Y_1Y_2\cdots Y_n = -Y_0 + 0.Y_1Y_2\cdots Y_n \end{aligned}$$