第3章 运算方法和运算部件

数据在计算机中的处理是通过各种运算来实现的。本章在第2章的基础上，讲解定点数和浮点数的运算方法，并适当介绍一些运算部件的组成及工作原理。

3.1 定点加减法运算

3.1.1 补码加减法运算

从第2章中了解到，补码比其他定点数的机器编码更适合加减法运算，因此，计算机中均采用补码进行加减运算。

设存放数据的寄存器位数为 n 位，x、y 为定点整数。根据补码的数学计算公式，有

$$[x]_{补} = 2^n + x, \quad [y]_{补} = 2^n + y \pmod{2^n}$$

所以

$$\begin{aligned}[x]_{补} + [y]_{补} &= 2^n + x + 2^n + y \\ &= 2^{n+1} + x + y \\ &= 2^n + (x + y) \\ &= [x + y]_{补} \pmod{2^n}\end{aligned}$$

同理

$$\begin{aligned}[x]_{补} - [y]_{补} &= (2^n + x) - (2^n + y) \\ &= x - y \\ &= 2^n + (x - y) \\ &= [x - y]_{补} \\ &= 2^n + 2^n + (x - y) \\ &= (2^n + x) + (2^n + (-y)) \\ &= [x]_{补} + [-y]_{补} \pmod{2^n}\end{aligned}$$

由此可得以下的定点整数补码加、减运算规则：

$$[x]_{补} + [y]_{补} = [x + y]_{补} \pmod{2^n} \tag{3.1}$$

$$[x]_{补} - [y]_{补} = [x]_{补} + [-y]_{补} = [x - y]_{补} \pmod{2^n} \tag{3.2}$$

同理，可得定点小数补码加、减运算规则：

$$[x]_{补} + [y]_{补} = [x + y]_{补} \pmod{2^1} \tag{3.3}$$

$$[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}=[x-y]_{补} \quad (\mathrm{mod}\ 2^1) \tag{3.4}$$

由式(3.1)～式(3.4)可知：两数补码之和、差，等于两数和、差之补码，这完全符合人们的预期。此外，式(3.2)和式(3.4)表明，补码减法运算可以转换为补码加法运算，这样可以简化运算器的设计。

【例 3.1】 设存放数据的寄存器为 8 位，$x=+1010110$，$y=-1001001$，求$[x+y]_{补}$。

解：首先求出 x 和 y 的补码。

$$[x]_{补}=\mathbf{0}1010110 \quad [y]_{补}=\mathbf{1}0110111$$

按式(3.1)，有

$$\begin{array}{rll} & \mathbf{0}\,1\,0\,1\,0\,1\,1\,0 & [x]_{补} \\ + & \mathbf{1}\,0\,1\,1\,0\,1\,1\,1 & [y]_{补} \\ \hline & \mathbf{0}\,0\,0\,0\,1\,1\,0\,1 & [x+y]_{补} \quad (\mathrm{mod}\ 2^8) \end{array}$$

从运算结果来看，最高位上产生了进位 1，但在模 2^8 的作用下，该位不被保留，所以

$$[x+y]_{补}=\mathbf{0}0001101 \quad (\mathrm{mod}\ 2^8)$$

其符号位为 **0**，说明和为正数。

【例 3.2】 设存放数据的寄存器为 8 位，$x=+1010110$，$y=+1101001$，求$[x-y]_{补}$。

解：首先求出 x 和 y 的补码。

$$[x]_{补}=\mathbf{0}1010110 \quad [y]_{补}=\mathbf{0}1101001$$

由式(3.2)可知，要将减法转换为加法，需要求出$[-y]_{补}$。

$$[-y]_{补}=\mathbf{1}0010111$$

由此可得

$$\begin{array}{rll} & \mathbf{0}\,1\,0\,1\,0\,1\,1\,0 & [x]_{补} \\ + & \mathbf{1}\,0\,0\,1\,0\,1\,1\,1 & [-y]_{补} \\ \hline & \mathbf{1}\,1\,1\,0\,1\,1\,0\,1 & [x-y]_{补} \quad (\mathrm{mod}\ 2^8) \end{array}$$

所以

$$[x-y]_{补}=\mathbf{1}1101101 \quad (\mathrm{mod}\ 2^8)$$

从运算结果来看，符号位为 1，说明差为负数。

【例 3.3】 设存放数据的寄存器为 8 位，$x=+1010110$，$y=+1001001$，求$[x+y]_{补}$。

解：首先求出 x 和 y 的补码。

$$[x]_{补}=\mathbf{0}1010110 \quad [y]_{补}=\mathbf{0}1001001$$

按式(3.1)，有

$$\begin{array}{rll} & \mathbf{0}\,1\,0\,1\,0\,1\,1\,0 & [x]_{补} \\ + & \mathbf{0}\,1\,0\,0\,1\,0\,0\,1 & [y]_{补} \\ \hline & \mathbf{1}\,0\,0\,1\,1\,1\,1\,1 & [x+y]_{补} \quad (\mathrm{mod}\ 2^8) \end{array}$$

从运算结果来看，符号位为 1，结果为负数。但由于 x、y 均为正数，其和不可能为负数。究竟是什么原因造成这样的错误呢？

补码是有一定的数据表示范围的，当两个数的补码相加(减)，其和(差)超出特定位数的补码所能表示的数据范围时，称为“溢出”。“溢出”表现为，数的最高有效数字位占据并改变了数的符号位，从而造成数据表示的错误。例 3.3 中，和的符号位实际上已被和的最高有效数字位占据，并且改变了数的正确符号状态，所以和发生了溢出。“溢出”意味着数据表示的

错误，如果无视这种错误，计算机就会产生错误的处理结果。因此，补码加减运算必须检测运算结果的“溢出”状态，并将检测结果反馈给处理器。

下面介绍几种常用的“溢出”检测方法：

(1) 根据运算结果的符号与运算数据的符号之间的关系检测“溢出”。设

$$[x]_{补} = \boldsymbol{x_{n-1}}x_{n-2}\cdots x_1x_0$$
$$[y]_{补} = \boldsymbol{y_{n-1}}y_{n-2}\cdots y_1y_0$$
$$[x+y]_{补} = \boldsymbol{s_{n-1}}s_{n-2}\cdots s_1s_0$$

其中，$\boldsymbol{x_{n-1}}$、$\boldsymbol{y_{n-1}}$和$\boldsymbol{s_{n-1}}$分别为$[x]_{补}$、$[y]_{补}$和$[x+y]_{补}$的符号位。以V表示“溢出”状态，则有

$$V = \overline{(x_{n-1} \oplus y_{n-1})} \wedge (x_{n-1} \oplus s_{n-1}) \tag{3.5}$$

若$V=1$，则有溢出；否则无溢出。

式(3.5)包含以下两方面的含义：

① 不同符号的数相加(或相同符号的数相减)，不会产生溢出；

② 相同符号的数相加(或不同符号的数相减)，如果和(或差)的符号与被加数(或被减数)的符号不同，则产生溢出(如例 3.3)。

这种“溢出”检测方法的检测电路较复杂、延时较大，见图 3.1(a)。

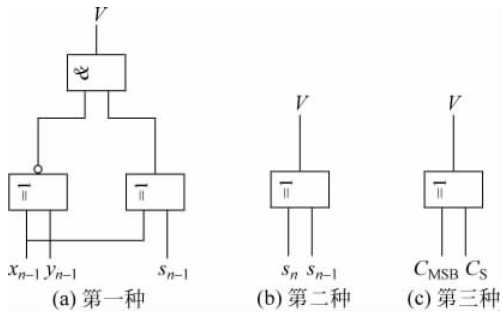


图 3.1　三种“溢出”检测电路

(2) 根据变形补码两个符号位之间的关系检测“溢出”。变形补码是具有两个符号位的补码，正数的变形补码，其两个符号位为 **00**，负数的变形补码，其两个符号位为 **11**。采用变形补码进行加(减)运算时，其两个符号位都参与运算，所得的和(差)也用变形补码表示。

当变形补码产生溢出时，数的最高有效数字位会占据并改变两个符号位中的低位，但两个符号位中的高位不会受到影响。因此，变形补码两个符号位中的高位总能表示数的正确符号。

【例 3.4】 设 $x=+1010110$，$y=+1001001$，用变形补码求$[x+y]_{补}$。

解：首先求出 x 和 y 的变形补码。

$$[x]_{补} = \mathbf{00}1010110 \quad [y]_{补} = \mathbf{00}1001001$$

按式(3.1)，有

$$\begin{array}{rll} & \mathbf{0\,0}\,1\,0\,1\,0\,1\,1\,0 & [x]_{补} \\ + & \mathbf{0\,0}\,1\,0\,0\,1\,0\,0\,1 & [y]_{补} \\ \hline & \mathbf{0\,1}\,0\,0\,1\,1\,1\,1\,1 & [x+y]_{补} \quad (\mathrm{mod}\ 2^9) \end{array}$$

可见，$[x+y]_{补}$的最高有效数字占据并改变了两个符号位中的低位，运算结果产生了溢

出,但其两个符号位中的高位仍为 **0**,指明运算结果为正数。由此,也得到了变形补码检测"溢出"的方法:当运算结果的两个符号位相同时,不产生溢出,否则产生溢出。设$[x+y]_{补}$用变形补码一般表示为

$$[x+y]_{补} = \boldsymbol{s_n s_{n-1}} s_{n-2} \cdots s_1 s_0$$

则有

$$V = s_n \oplus s_{n-1} \tag{3.6}$$

按式(3.6)设计的"溢出"检测电路如图 3.1(b)所示,它比图 3.1(a)简单,延时也少。但由于增加了一个符号位,也就增加了运算电路的复杂程度。

(3) 按补码相加时最高有效数字位产生的进位与符号位产生的进位之间的关系检测"溢出"。设最高有效数字位产生的进位为 C_{MSB},符号位产生的进位为 C_S,则有

$$V = C_{MSB} \oplus C_S \tag{3.7}$$

式(3.7)指出,当 C_{MSB} 和 C_S 相同时,不产生溢出,否则产生溢出(为什么可以这样检测?留给读者来分析)。式(3.7)与式(3.6)同样简单,且这种检测方法采用单符号位,因此,这种检测方法是三种方法中效率最高的。对应的检测电路如图 3.1(c)所示。

如前所述,"溢出"是由于"数的最高有效数字位占据并改变了数的符号位"所引起的,其中强调了"改变了数的符号位"。也就是说,如果仅仅是"数的最高有效数字位占据了数的符号位",而并未"改变了数的符号位",则亦无溢出。如 n 位的定点整数补码可以表示到 -2^{n-1},而定点小数补码可以表示到 -1,就是这个原因。

3.1.2 行波进位补码加法/减法器

由于补码减法可以转换成补码加法进行,因此,补码加法/减法器的主体是加法器。构成加法器的主要器件是全加器,一个全加器是实现带进位的 1 位加法的器件,如图 3.2 所示。

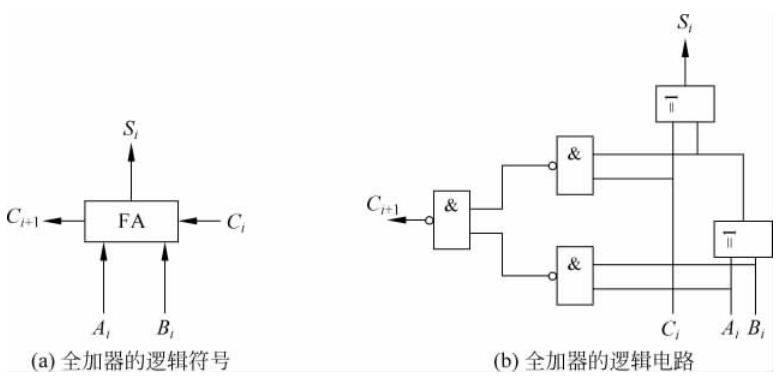


图 3.2 全加器的逻辑符号和逻辑电路

图 3.2(a)中,A_i 和 B_i 是本位上相加的两个 1 位二进制数;C_i 是低位向本位产生的进位;S_i 是本位上 A_i、B_i、C_i 相加所得的和;C_{i+1}是本位向上产生的进位。根据二进制加法运算的特点,有

$$\begin{cases} S_i = A_i \oplus B_i \oplus C_i \\ C_{i+1} = A_iB_i + A_iC_i + B_iC_i = A_iB_i + (A_i \oplus B_i)C_i = \overline{\overline{A_iB_i} \cdot \overline{(A_i \oplus B_i)C_i}} \end{cases} \tag{3.8}$$

按式(3.8)设计的全加器逻辑电路如图 3.2(b)所示。

用全加器构造一个单纯的多位补码加法器是很容易的，只需将多个全加器按进位相联的方式级联起来即可。而这里要构造的是同时具有补码加法和补码减法功能的加法/减法器。由式(3.2)或式(3.4)可知，做补码减法时，$[A]_{补}-[B]_{补}=[A]_{补}+[-B]_{补}$，这与直接的补码加法$[A]_{补}+[B]_{补}$相比，区别只是加数不同：做加法时，加数为$[B]_{补}$，而做减法时，加数为$[-B]_{补}$。因此，只要能根据做加法或做减法分别选择$[B]_{补}$或$[-B]_{补}$作加数，就能实现补码加法/减法器。由于$[-B]_{补}=\overline{[B]_{补}}+1$，因此，可以在做减法时，将$[B]_{补}$按位取反，然后加上 1，求得$[-B]_{补}$；而在做加法时不做这种转换，直接使用$[B]_{补}$。按这种思想构造的一个 n 位的行波进位补码加法/减法器如图 3.3 所示。

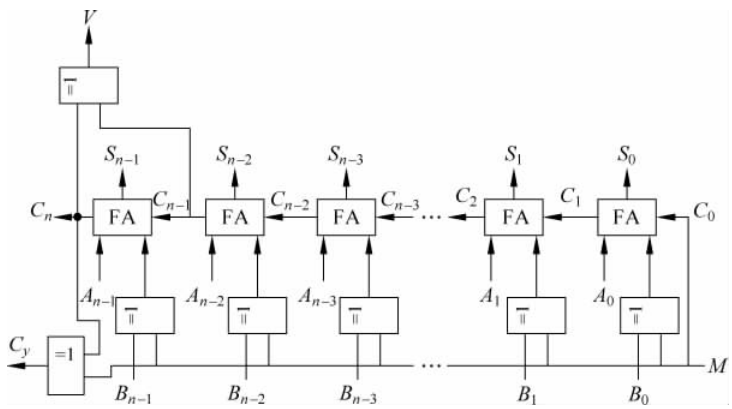


图 3.3　n 位行波进位补码加法/减法器

图 3.3 中，$[A]_{补}$、$[B]_{补}$均为 n 位补码，A_{n-1}和 B_{n-1}分别为$[A]_{补}$和$[B]_{补}$的符号位；$[A]_{补}$作为被加数或被减数，$[B]_{补}$作为加数或减数；M 是方式控制信号，$M=0$ 时，控制异或门将$[B]_{补}$的各位直接送到加法器，且 $C_0=M=0$，所以做的是加法，$M=1$ 时，控制异或门将$[B]_{补}$的各位取反后(即$\overline{[B]_{补}}$)送到加法器，且 $C_0=M=1$，所以，此时加法器实际做的是$[A]_{补}+\overline{[B]_{补}}+1=[A]_{补}+[-B]_{补}=[A]_{补}-[B]_{补}$，即减法。图 3.3 还采用了式(3.7)描述的“溢出”检测方法。

实际的定点运算器中，加法/减法器除了做补码加减运算外，还要做无符号数的加减运算。无符号数是指无符号位的定点数，其每一位都是有效数字，一个 n 位的寄存器可以存放一个 n 位的无符号数。无符号数的加减运算不考虑“溢出”问题，但要考虑最高有效位上产生的进位或借位问题，因为，这关系到无符号数的大小比较和多字长无符号数的加减运算等。图 3.3 中的 Cy，就是在进行无符号数加减运算后，最高有效位上产生的进位或借位状态：$Cy=0$，表示最高有效位上无进位或借位，$Cy=1$，表示最高有效位上有进位或借位(请读者自行分析其产生原理)。

行波进位补码加法/减法器采用的是串行工作方式，即运算从低位向高位逐位进行，因此运算速度比较慢。

3.2 定点乘法运算

3.2.1 原码一位乘法

由于原码的数字部分与数的绝对值是一致的,因此,当两个用原码表示的数相乘时,可以用其数字部分直接相乘,得到乘积的数字部分,而乘积的符号取两个数符号的异或值即可。

为了探求计算机实现原码一位乘法的方法,先来分析一下人工进行二进制乘法运算的过程。例如,$x=1010$,$y=1101$,则 $x \times y$ 的计算过程如下。

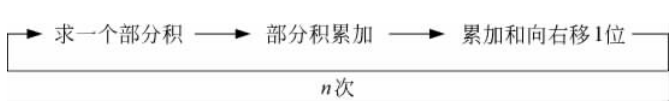
$$
\begin{array}{r}
1\ 0\ 1\ 0 \\
\times\quad 1\ 1\ 0\ 1 \\
\hline
1\ 0\ 1\ 0 \\
0\ 0\ 0\ 0 \\
1\ 0\ 1\ 0 \\
1\ 0\ 1\ 0 \\
\hline
1\ 0\ 0\ 0\ 0\ 0\ 1\ 0
\end{array}
$$

所以 $x \times y=10000010$。以上计算过程可以归纳为两个步骤:

(1) 依次求出各个部分积,并逐个向左偏移 1 位排列;

(2) 对排列好的部分积求和。

这两步工作在计算机中如何完成呢? 首先,求部分积时,可用乘数的 1 位与被乘数的每一位按"逻辑与"运算求得(如用乘数 y 的最低位 1 与被乘数 x 的每一位相"与",得到第一个部分积 1010);其次,计算机不能一次完成多个部分积求和,每次只能对两个数求和,因此,只能采用累加的方式对多个部分积求和;最后,如何实现下一个部分积相对前一次的累加和向左偏移 1 位呢? 实际上,偏移是相对的,为了便于实现,计算机中可以采用部分积不偏移,而将每次得到的部分积累加和向右偏移 1 位,再与下一个部分积累加的方法,来满足计算要求。综合以上因素,可以归纳出计算机执行原码一位乘法的步骤如下:



对于两个 n 位二进制数相乘,以上步骤需要重复 n 次;其中,第一个部分积是与 0 相加。

图 3.4 所示为实现原码一位乘法的逻辑电路框图。在图中,寄存器 R_0、R_1、R_2 均为 n 位寄存器,其中,R_1 和 R_2 分别存放乘数和被乘数的 n 位数字位(符号位另行处理),R_0 的初值为 0,用于存放每次的部分积累加和;"与"逻辑由 n 个"与"门组成,用于将乘数的当前位与被乘数的每一位相"与",求出部分积;n 位加法器用于完成部分积的累加,C 锁存累加产生的最高位进位;计数器用于控制乘法操作步骤的重复次数;控制逻辑用于控制各相关部件的操作;"异或"门用于产生乘积的符号。

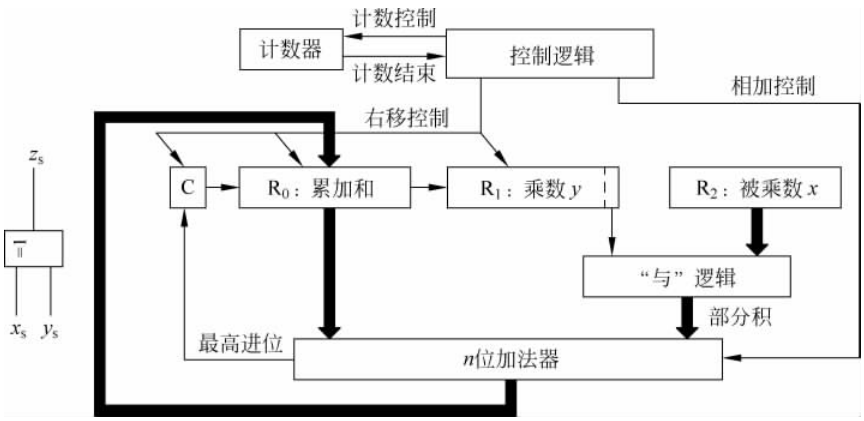


图 3.4　原码一位乘法逻辑电路框图

乘法运算的过程是：

(1) 将乘数 y 和被乘数 x 的数字部分分别置于 R_1 和 R_2 寄存器中，同时将 R_0、C 和计数器清零；

(2) 由 R_1 的最低位(也就是乘数的当前位)与 R_2(被乘数)的每一位相"与"，得到本次的部分积，在控制逻辑的控制下，与 R_0 中的数相加，得到部分积的累加和，并送回 R_0 存储，相加中最高位产生的进位送入 C 锁存；

(3) 控制逻辑发出右移控制信号，使 C、R_0 和 R_1 串联起来向右移动 1 位，即 C 的值移到 R_0 的最高位，R_0 的最低位移到 R_1 的最高位，R_1 的最低位被移出丢弃(因为这一位已用过)；

(4) 控制逻辑控制计数器加 1 计数，若未计到 n，则返回(2)，否则结束乘法运算过程。

乘法结束后，乘积的低位数字部分在 R_1 中，高位数字部分在 R_0 中，积的符号(z_s)则是 x 和 y 的符号(x_s 和 y_s)的"异或"。

3.2.2　补码一位乘法

在计算机中，定点数主要是以补码表示的，因此，必须解决补码乘法问题。

由于负数补码的数字部分与其原码是不同的，所以，补码不能像原码那样用数字部分直接做二进制乘法。解决补码乘法的一种方法是，先将补码转换成原码，然后再按原码乘法(如前面介绍的原码一位乘法)求得乘积的原码，最后再将乘积从原码转换为补码。这种方法原理简单，但操作步骤多、速度慢。显然，更为有效的方法是直接采用补码相乘。

式(3.9)是由布斯(Booth)提出的补码乘法公式，称为"布斯公式"。设有 n 位补码

$$[x]_{补} = x_{n-1}x_{n-2}\cdots x_1x_0$$

$$[y]_{补} = y_{n-1}y_{n-2}\cdots y_1y_0y_{-1}$$

其中，x_{n-1} 和 y_{n-1} 是 $[x]_{补}$ 和 $[y]_{补}$ 的符号位；y_{-1} 是给 $[y]_{补}$ 添加的一个附加位，且 $y_{-1}=0$，则有

$$[x \cdot y]_{补} = [x]_{补} \cdot \sum_{i=n-1}^{0} (y_{i-1} - y_i)2^{i-(n-1)} \tag{3.9}$$

由式(3.9)导出的布斯算法的流程图如图 3.5 所示。图中，A 寄存器的初值为 0；X 和

Y 寄存器分别存放$[x]_{补}$和$[y]_{补}$；Y_{-1}是一个 1 位寄存器(即一个触发器)，作为给 Y 添加的最低附加位，初值为 0；Y_0Y_{-1}即为 Y 的最低位 Y_0 与附加位 Y_{-1}组成的两位二进制序列；i 用于操作步骤计数。

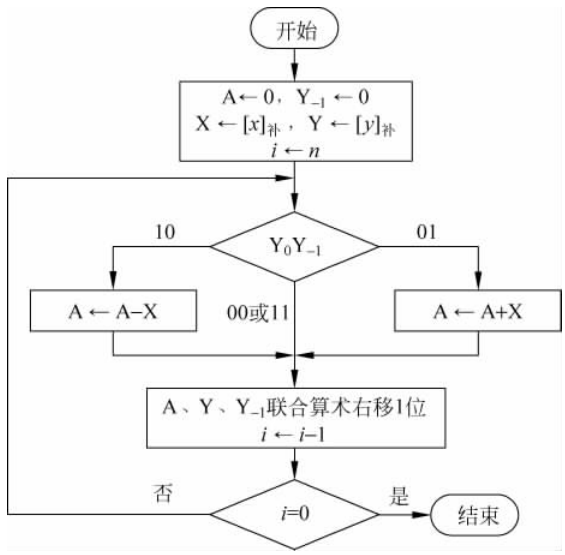


图 3.5　补码乘法的布斯算法流程图

【例 3.5】 设 $x=+101101$，$y=-110010$，用布斯算法求$[x \cdot y]_{补}$。

解：$[x]_{补}=0101101$，$[-x]_{补}=1010011$，$[y]_{补}=1001110$，$Y_{-1}=0$，计算过程如下。

A	Y	Y_0Y_{-1}	说　明
0 0 0 0 0 0 0	1 0 0 1 1 1	**0 0**	A、Y、Y_{-1} 的初始状态，$Y_0Y_{-1}=00$
0 0 0 0 0 0 0	0 1 0 0 1 1	**1 0**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=10$
+ 1 0 1 0 0 1 1			A ← A −X（即 A ← A+$[-x]_{补}$）
1 0 1 0 0 1 1	0 1 0 0 1 1	**1 0**	
1 1 0 1 0 0 1	1 0 1 0 0 1	**1 1**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=11$
1 1 1 0 1 0 0	1 1 0 1 0 0	**1 1**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=11$
1 1 1 1 0 1 0	0 1 1 0 1 0	**0 1**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=01$
+ 0 1 0 1 1 0 1			A ← A+X（即 A ← A+$[x]_{补}$）
0 1 0 0 1 1 1	0 1 1 0 1 0	**0 1**	
0 0 1 0 0 1 1	1 0 1 1 0 1	**0 0**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=00$
0 0 0 1 0 0 1	1 1 0 1 1 0	**1 0**	A、Y、Y_{-1} 算术右移 1 位，$Y_0Y_{-1}=10$
+ 1 0 1 0 0 1 1			A ← A −X（即 A ← A+$[-x]_{补}$）
1 0 1 1 1 0 0	1 1 0 1 1 0	**1 0**	
1 1 0 1 1 1 0	0 1 1 0 1 1	**0 1**	A、Y、Y_{-1} 最后算术右移 1 位

因此，计算结果为

$$[x \cdot y]_{补} = 11011100110110$$

真值为

$$x \cdot y = -100011001010$$

由例 3.5 可知，采用布斯算法对两个 n 位补码相乘，其乘积为 $2n$ 位，其中，乘积的高位部分在 A 寄存器中，低位部分在 Y 寄存器中。算法中的“算术右移”是补码右移的规则，即连同符号位右移一位，符号位补充原来的符号。补码算术右移一位，相当于乘以 2^{-1}。当 Y_0Y_{-1} 为 00 或 11 时，不用做加减运算，只需做算术右移一位，因此，布斯算法能减少加减运算的次数，但其控制比原码一位乘法复杂。

图 3.5 是针对定点整数补码乘法的布斯算法，如为定点小数补码乘法，则最后一步时，不应再做算术右移，而乘积中也不包含 Y 寄存器的最低位 Y_0（或者也可将 Y_0 清零）。这是因为，小数右移后，高位移入的 0 会改变数的大小。

无论是补码一位乘法，还是原码一位乘法，其共同特点是：一次求一个部分积，做一次加法（或减法），再做一次右移，这样的过程需重复 n 次。因此，一位乘法是一种全串行的乘法方法，运算速度慢。

3.2.3　阵列乘法器

大规模集成电路出现后，产生了各种形式的高速阵列乘法器，它们具有一定的并行工作特征，属于并行乘法器。

1. 无符号数阵列乘法器

设 X 和 Y 是两个 n 位无符号二进制数，

$$X = x_{n-1}x_{n-2}\cdots x_1x_0$$

$$Y = y_{n-1}y_{n-2}\cdots y_1y_0$$

若以 X 为被乘数，Y 为乘数，则可以采用如图 3.6 所示的部分积产生电路同时求得 n 个部分积。

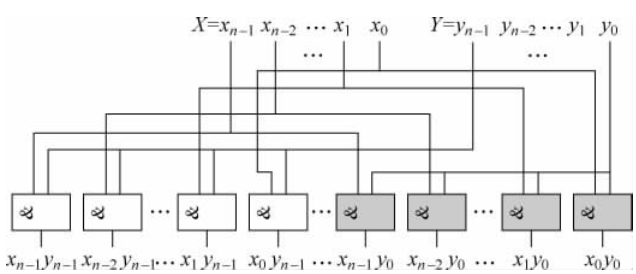


图 3.6　部分积产生电路

相比原码一位乘法要求 n 次部分积，图 3.6 中的部分积产生电路只用了求一次部分积的时间，就求得了 n 个部分积，因此，这个部分积产生电路采用的是并行工作方式，每 n 个“与”门为一组，输出一个部分积，整个电路共使用了 n^2 个“与”门。

对部分积相加则采用一个乘法阵列来完成。图 3.7 所示为实现两个 5 位无符号二进制数相乘的乘法阵列逻辑电路图。可见，乘法阵列是由若干行全加器组成的，由部分积产生电路产生的部分积被安排在相应的行上，每行（最后一行除外）完成一次部分积累加。各行的排列方式按照部分积相加时的要求，后一行相对前一行向左偏移 1 位，使得部分积累加后不用再做右移。除最后一行外，阵列的其他各行并非行波进位加法器，其低位产生的进位不是

进到本行的下一位,而是进到下一行的下一位(图中斜线箭头表示全加器的进位输入或进位输出),这是模仿人工做部分积相加时,对各部分积按列相加,并将进位进到下一列的方式设计的。由于同一行上的各位之间没有进位传递关系,所以,各位可以同时相加,具有并行工作的特点。阵列的最后一行是一个行波进位加法器,用于对最后一次部分积累加产生的和及进位做处理。乘法阵列的输出,即所求的乘积。

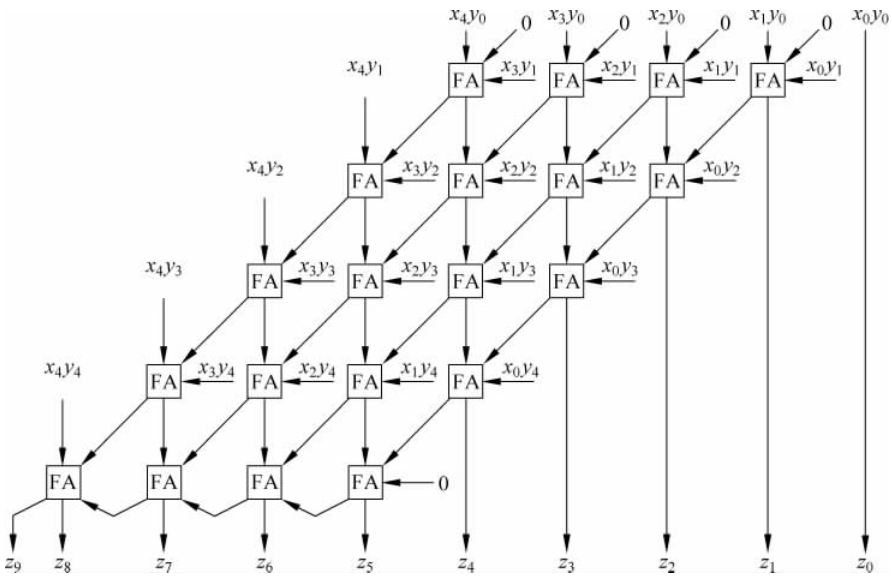


图 3.7 用于两个 5 位无符号数相乘的乘法阵列逻辑电路图

一般地,实现两个 n 位无符号数相乘的乘法阵列,共需 n 行全加器,每行需 $n-1$ 个全加器,故共需 $n(n-1)$个全加器。

将部分积产生电路与乘法阵列相连接,就得到完整的无符号数阵列乘法器,如图 3.8 所示。对 n 位无符号数乘法而言,阵列乘法器的运算速度是一位乘法器的大约 $n/4$ 倍。

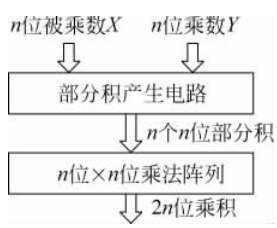


图 3.8 无符号数阵列乘法器组成框图

2. 有符号数阵列乘法器

如有符号数采用原码表示,在相乘时,可将被乘数和乘数的数字部分看作无符号数,直接送入无符号数阵列乘法器进行运算,而乘积的符号则由两数的符号经逻辑“异或”产生。因此,只要在无符号数阵列乘法器的基础上,添加一个符号处理电路,即可得到原码阵列乘法器。

如有符号数采用补码表示，一种方法是，先将被乘数和乘数转换成原码，然后送入原码阵列乘法器进行运算，最后再将乘积由原码转换成补码，即构造一种间接补码阵列乘法器；另一种方法则是直接采用补码相乘，即构造直接补码阵列乘法器。下面仅介绍前一种方法。

如第 2 章中所述，补码与原码之间的互相转换采用的是相同的方法，即正数无须转换；对负数转换时，符号位保持不变，各数字位按位取反，最后再加 1。这种方法要做加 1 运算，需要用到加法电路，所以会增加转换器的硬件复杂度。在此基础上，可以得到一种改进的转换方法：正数无须转换；对负数转换时，符号位保持不变，数字部分从最低位向高位方向寻找第一个“1”，该位“1”及其以右的低位数字保持不变，以左的高位数字则按位取反。改进后的转换方法避免了加 1 操作，降低了转换器的复杂度，如图 3.9 所示，即为一个按此方法设计的 4 位转换器（也称求补器）逻辑电路。

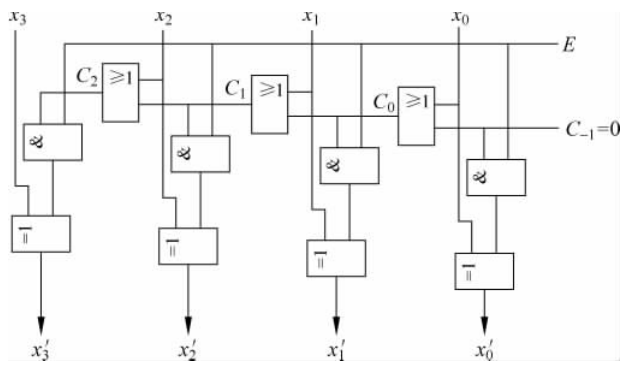


图 3.9　4 位求补器逻辑电路图

在图 3.9 中，$x_3x_2x_1x_0$ 为需要转换的补码（原码）；$x_3'x_2'x_1'x_0'$为转换得到的原码（补码）；E 为转换控制信号，$E=0$ 时，不做转换，即 $x_3x_2x_1x_0=x_3'x_2'x_1'x_0'$；$E=1$ 时，实施转换。对应补码与原码之间的转换要求，正数时无须转换，此时 E 应为 0；负数时需要转换，此时 E 应为 1，因此，E 与被转换的数的符号一致。位数更多的求补器，只需在图 3.9 的基础上，按相同的逻辑关系进行扩展即可。

对这种求补器的使用有两种方式，一是不转换符号位，只转换数字位，这是一般的补码与原码之间的转换；二是符号位与数字位一起转换，此时，求补器的转换结果实际上是被转换数据的绝对值（也就是无符号数）。由于补码与原码的数据表示范围不完全相同，用补码表示的最小数用原码表示不出来，因此，在构造间接补码阵列乘法器时，要按上述第二种方式使用求补器。

图 3.10 所示为实现 n 位补码相乘的间接补码阵列乘法器组成框图，被乘数 X 和乘数 Y 均为 n 位补码，x_{n-1}和 y_{n-1}为其符号位。运算前，x_{n-1}和 y_{n-1}分别控制一个 n 位求补器将 X 和 Y 转换成 n 位无符号数；运算后，则用 x_{n-1}和 y_{n-1}的“异或”输出（即乘积的符号状态），控制输出端的 $2n$ 位求补器将乘积转换成 $2n$ 位补码。

间接补码阵列乘法器由于需要在运算前、后进行求补转换，其工作时间大约比无符号数阵列乘法器多一倍。

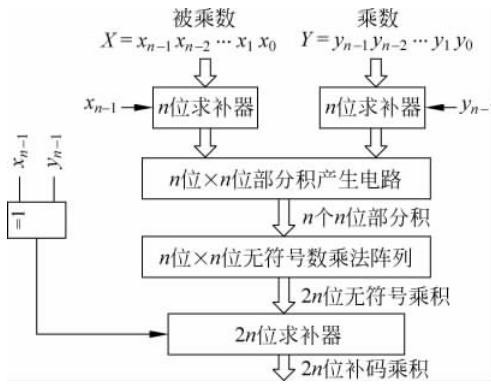


图 3.10 间接补码阵列乘法器组成框图

3.3 定点除法运算

3.3.1 原码一位除法

原码除法以原码表示被除数与除数,以它们的数字部分(相当于绝对值或无符号数)相除,商的符号为两数符号的逻辑“异或”,余数的符号则总是与被除数的符号相同。

计算机中的定点除法运算规定,被除数的位数应是除数位数的两倍。当被除数的每一位都参加运算后,运算即结束。因此,定点除法运算的结果包含两个部分:商和余数。考虑到定点小数除法要求商也应为定点小数(即绝对值小于 1),因此,定点除法(包括定点整数除法和定点小数除法)规定,在被除数与除数以绝对值相除时,若第一次上的商为 1,则除法出错,称为“除法溢出”。

下面以人工进行无符号数除法运算为例,来探寻计算机做无符号数除法的基本过程。

设被除数 x 和除数 y 分别为

$$x = 11001, \quad y = 111$$

为了使被除数的位数达到除数位数的两倍,在 x 的最高位添 1 个 0,得

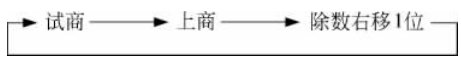
$$x = 011001$$

除法运算过程如下:

```
        0011          商
111 ) 011001
    - 111             首次试商，不够减，上商 0，不减除数
      011001 ——  r0   部分余数 r0
    - 0111            除数右移 1 位试商，不够减，上商 0，不减除数
      011001 ——  r1   部分余数 r1
    - 00111           除数右移 1 位试商，够减，上商 1，减除数
      001011 ——  r2   部分余数 r2
    - 000111          除数右移 1 位试商，够减，上商 1，减除数
      000100 ——  r3   除法结束，部分余数 r3 也就是最终的余数
```

第一次上商为 0，故未产生除法溢出，运算结果正确，即商＝011，余数＝100。

以上除法运算过程可以描述为：重复进行以下操作



直到被除数的每一位均参与了运算为止。此过程中，“上商”和“除数右移”很容易在计算机中实现，只有“试商”是比较复杂的。

人在做减除数试商时，首先通过心算判断是否够减，如够减，则减除数得新的部分余数，同时上商 1；如不够减，则不减除数，保持原来的部分余数不变，并上商 0。但是，计算机不具备类似人的心算能力，它在做试商时，只能先减去除数，再判断差的符号，若符号为 0（表示差为正数），则够减，上商 1，差即为正确的部分余数；若符号为 1（表示差为负数），则不够减，上商 0，而差是错误的部分余数。对错误的部分余数的不同处理方式，形成了“恢复余数法”和“加减交替法”两种原码一位除法的方法。

恢复余数法的思想是：当减除数试商得到部分余数为正时，上商 1，部分余数正确；当出现部分余数为负时，上商 0，然后在错误的部分余数上再加上除数，使部分余数恢复正常，以便继续后面的运算。

加减交替法的思想是：首次试商采用减除数试商，当部分余数为正时，上商 1，下次试商采用减除数试商；当部分余数为负时，上商 0，不恢复余数，但在下次试商时采用加除数试商。由于试商时，既有减除数试商，又有加除数试商，因此，这种方法被称为“加减交替法”，也称为“不恢复余数法”。

以上两种方法都要判断部分余数的符号，因此，需要给被除数和除数的数字部分都添加一个符号位，并设置为正号(0)。除法运算时，相当于两个正数相除，而商和余数的符号另行处理。

【例 3.6】 设$[x]_{原}=\mathbf{0}.101001$，$[y]_{原}=\mathbf{1}.111$。用恢复余数法求 $x\div y$。

解：分别以 x' 和 y' 表示 x 和 y 的数字部分，并添加正号 **0**，则有

$$x'=\mathbf{0}.101001,\quad y'=\mathbf{0}.111$$

考虑到计算机把减法变成加法来计算的特点，减除数 y' 试商时，实际上是在加$[-y']_{补}$，因此给出$[-y']_{补}=\mathbf{1}.001$。运算过程如下：

0.101001	
+ **1**.001	减 y' 试商
1.110001	符号为 **1**，部分余数为负，上商 0
+ **0**.111	加 y'，恢复余数
0.101001	
+ **1**.1001	y' 右移 1 位，减 y' 试商
0.001101	符号为 **0**，部分余数为正，上商 1
+ **1**.11001	y' 右移 1 位，减 y' 试商
1.111111	符号为 **1**，部分余数为负，上商 0
+ **0**.00111	加 y'，恢复余数
0.001101	
+ **1**.111001	y' 右移 1 位，减 y' 试商
0.000110	符号为 **0**，部分余数为正，上商 1，运算结束

第一次上商为0,故未产生除法溢出,运算结果正确。商的数字部分为0.101,余数的数字部分为0.000110。由于x与y异号,且x为正,故得商=−0.101,余数=+0.000110,表示成原码为$[商]_{原}=\mathbf{1}.101$,$[余数]_{原}=\mathbf{0}.000110$。

由例3.6可知,恢复余数法要增加恢复余数的运算步骤,所以运算速度慢,且运算步数不确定。

【例3.7】 仍设$[x]_{原}=\mathbf{0}.101001$,$[y]_{原}=\mathbf{1}.111$。用加减交替法求$x\div y$。

解:分别以x'和y'表示x和y的数字部分,并添加正号**0**,则有

$$x'=\mathbf{0}.101001,\quad y'=\mathbf{0}.111$$

考虑到计算机把减法变成加法来计算的特点,减除数y'试商时,实际上是在加$[-y']_{补}$,因此给出$[-y']_{补}=\mathbf{1}.001$。运算过程如下:

0.101001	
+ **1.001**	首次试商,采用减y'试商
1.110001	符号为**1**,部分余数为负,上商0
+ **0.0111**	不恢复余数,y'右移1位,加y'试商
0.001101	符号为**0**,部分余数为正,上商1
+ **1.11001**	y'右移1位,减y'试商
1.111111	符号为**1**,部分余数为负,上商0
+ **0.000111**	不恢复余数,y'右移1位,加y'试商
0.000110	符号为**0**,部分余数为正,上商1,运算结束

可见,运算结果与例3.6完全相同。以上运算过程中,加粗的数字为部分余数和除数的符号位及符号扩展位(符号扩展位与符号位的状态是一致的,它是补码表示中的一种特殊现象,当将补码转换成原码后,符号扩展位将被全部转换为0)。从本例中还可以发现,无论是减除数试商,还是加除数试商,实际均为两个异号的数相加。试商时,符号位上相加产生的和及向上进位状态,将一直传递到最高扩展符号位,而最高扩展符号位的向上进位状态,与本次应上的商一致。

由于加减交替法避免了恢复余数,运算步骤少,且步数确定,因此运算效率较高,更适合计算机使用。

【例3.8】 设$[x]_{原}=\mathbf{0}0111$,$[y]_{原}=\mathbf{0}11$。用加减交替法求$x\div y$。

解:分别以x'和y'表示x和y的数字部分,并添加正号**0**,则有

$$x'=\mathbf{0}0111,\quad y'=\mathbf{0}11$$

并且$[-y']_{补}=101$。

下面是运算过程:

00111	
+ **1**01	首次试商,采用减y'试商
11011	符号为**1**,部分余数为负,上商0
+ **0**011	不恢复余数,y'右移1位,加y'试商
00001	符号为**0**,部分余数为正,上商1
+ **1**1101	y'右移1位,减y'试商
11110	符号为**1**,部分余数为负,上商0,运算结束

第一次上商为0，故未产生除法溢出，运算结果正确。商的数字部分为10，且由于 x 与 y 同号，故商为正，$[商]_{原}=\mathbf{0}10$。由于最后一次部分余数为负，故还需做一次恢复余数的运算，即再加上 **000**11，得正确的部分余数为 **000**01。因为是整数除法，且 x 为正，故最终余数为+01，$[余数]_{原}=\mathbf{0}01$。

可见，当加减交替法产生的最后一次部分余数为负时，也需要做恢复余数运算。有时，这种恢复余数运算可能要做多次，才能得到正确的余数。这给运算带来了不利的影响。实际上，恢复余数是一个回溯过程，即回溯到最近一次出现的正余数。一种较好的解决方法是：利用一个寄存器，其初始内容为被除数，此后，在除法运算过程中，每当部分余数为正，即将该部分余数存入此寄存器。运算结束时，该寄存器的内容即为正确的余数。

此外，整数相除时，所得的商和余数的位数均与除数一致；小数相除时，所得的商与除数具有相同的位数，而余数的位数则与被除数一致。

如何证明加减交替法的正确性呢？首先，恢复余数法的正确性是不用怀疑的。下面通过将加减交替法与恢复余数法对比的方式，来证明加减交替法的正确性。

设除数为 y，运算过程中采用除数右移方式。当部分余数为正时，两种方法的处理过程是相同的；若第 i 次试商所得的部分余数 r_i 为负，对恢复余数法，处理过程是：

(1) 上商0；

(2) 恢复余数：r_i+y；

(3) 除数右移：$y\times 2^{-1}$(右移1位相当于除以2)；

(4) 第 $i+1$ 次减除数试商，得 $r_{i+1}=(r_i+y)-y\times 2^{-1}=r_i+y\times 2^{-1}$。

对加减交替法，处理过程是：

(1) 上商0；

(2) 除数右移：$y\times 2^{-1}$；

(3) 第 $i+1$ 次采用加除数试商，得 $r_{i+1}=r_i+y\times 2^{-1}$。

由此可见，加减交替法与恢复余数法的最终运算结果是完全相同的。因此，加减交替法是正确的。

3.3.2　补码一位除法

补码除法采用补码直接相除，即从 $[x]_{补}$ 和 $[y]_{补}$ 直接求 $[x\div y]_{补}$。下面介绍补码加减交替法，同样要求被除数的数字部分位数是除数的两倍。

在不产生除法溢出的前提下，补码加减交替法的运算规则如下(证明略)：

(1) 第一次试商时，若被除数与除数同号，则减除数试商，否则加除数试商；若试商所得部分余数与除数同号，则上商1，否则上商0。这是第一次上的商，也就是最终商的符号位(如果在被除数与除数同号时，商的符号位为1，或在被除数与除数异号时，商的符号位为0，均产生除法溢出)。

(2) 求商的数字部分时，先将除数右移1位，如果上次上商1，则减除数试商，否则加除数试商；若试商所得部分余数与除数同号，则上商1，否则上商0。重复(2)，直到被除数每一位均参加运算为止。

(3) 当除不尽时，若商为负，则需对商最低位加1修正；若商为正，则无须修正。当正好

除尽时,若除数为负,则需对商最低位加 1 修正;若除数为正,则无须修正。

(4) 若最后一次试商所得部分余数与被除数同号,则余数正确,否则需要恢复余数。恢复余数是一个余数回溯的过程,可能需要连续多次回溯,才能将余数恢复到与被除数同号。

【例 3.9】 设$[x]_{补}=\mathbf{1}.0111$,$[y]_{补}=\mathbf{0}.1101$。用补码加减交替法求$[x\div y]_{补}$。

解:将被除数的数字位数扩展到除数的两倍,有

$$[x]_{补}=\mathbf{1}.01110000$$

并按加减交替法的需要,给出$[-y]_{补}=\mathbf{1}.0011$。

运算过程如下:

1.01110000	
+ **0**.1101	x 与 y 异号,第一次加除数试商
0.**0**1000000	部分余数与除数同号,上商 1,即最终商的符号位为 1
+ **1**.**1**0011	除数右移 1 位,上次上商 1,本次减除数试商
1.**11**011000	部分余数与除数异号,上商 0
+ **0**.**00**1101	除数右移 1 位,上次上商 0,本次加除数试商
0.**000**01100	部分余数与除数同号,上商 1
+ **1**.**111**0011	除数右移 1 位,上次上商 1,本次减除数试商
1.**1111**0010	部分余数与除数异号,上商 0
+ **0**.**0000**1101	除数右移 1 位,上次上商 0,本次加除数试商
1.11111111	部分余数与除数异号,上商 0,运算结束

最后一次部分余数与被除数同号,余数正确;余数不为 0,未除尽,且商为负,商最低位需加 1 修正,即 1.0100+1=1.0101,故得

$$[\text{商}]_{补}=\mathbf{1}.0101,\quad [\text{余数}]_{补}=\mathbf{1}.11111111$$

【例 3.10】 设$[x]_{补}=\mathbf{0}.0100$,$[y]_{补}=\mathbf{0}.1000$。用补码加减交替法求$[x\div y]_{补}$。

解:将被除数的数字位数扩展到除数的两倍,有

$$[x]_{补}=\mathbf{0}.01000000$$

并按加减交替法的需要,给出$[-y]_{补}=\mathbf{1}.1000$。

运算过程如下:

0.01000000	
+ **1**.1000	x 与 y 同号,第一次减除数试商
1.**11**000000	部分余数与除数异号,上商 0,即最终商的符号位为 0
+ **0**.**0**1000	除数右移 1 位,上次上商 0,本次加除数试商
0.**00**000000	部分余数与除数同号,上商 1
+ **1**.**111**000	除数右移 1 位,上次上商 1,本次减除数试商
1.**111**00000	部分余数与除数异号,上商 0
+ **0**.**000**1000	除数右移 1 位,上次上商 0,本次加除数试商
1.**1111**0000	部分余数与除数异号,上商 0
+ **0**.**0000**1000	除数右移 1 位,上次上商 0,本次加除数试商
1.11111000	部分余数与除数异号,上商 0,运算结束

最后一次部分余数与被除数异号，余数错误，需要恢复余数，整个过程需要三次回溯，才能将余数恢复到与被除数同号，即

$$\begin{aligned}&(((1.11111000-0.00001000)-0.0001000)-1.111000)\\=&(((1.11111000+1.11111000)+1.1111000)+0.001000)\\=&\ 0.00000000\end{aligned}$$

可见，最终余数为 0，说明除尽，并且除数为正，商无须修正，故得

$$[\text{商}]_{\text{补}}=\mathbf{0}.1000,\quad [\text{余数}]_{\text{补}}=\mathbf{0}.00000000$$

【例 3.11】 设$[x]_{\text{补}}=\mathbf{1}.1100$，$[y]_{\text{补}}=\mathbf{0}.1000$。用补码加减交替法求$[x\div y]_{\text{补}}$。

解：将被除数的数字位数扩展到除数的两倍，有

$$[x]_{\text{补}}=\mathbf{1}.11000000$$

并按加减交替法的需要，给出$[-y]_{\text{补}}=\mathbf{1}.1000$。

运算过程如下：

1.11000000	
+ **0.1000**	x 与 y 异号，第一次加除数试商
0.01000000	部分余数与除数同号，上商 1，即最终商的符号位为 1
+ **1.11000**	除数右移 1 位，上次上商 1，本次减除数试商
0.00000000	部分余数与除数同号，上商 1
+ **1.111000**	除数右移 1 位，上次上商 1，本次减除数试商
1.11100000	部分余数与除数异号，上商 0
+ **0.0001000**	除数右移 1 位，上次上商 0，本次加除数试商
1.11110000	部分余数与除数异号，上商 0
+ **0.0001000**	除数右移 1 位，上次上商 0，本次加除数试商
1.11111000	部分余数与除数异号，上商 0，运算结束

最后一次部分余数与被除数同号，余数正确；余数不为 0，未除尽，且商为负，需要在最低位加 1 修正，即 **1**.1000＋1＝**1**.1001，故得

$$[\text{商}]_{\text{补}}=\mathbf{1}.1001,\quad [\text{余数}]_{\text{补}}=\mathbf{1}.11111000$$

实际上，例 3.11 与例 3.10 中的除数相等，且被除数是绝对值相等的，但由于被除数的符号相反，所以，最终的商和余数的绝对值发生了明显的改变。这也是补码除法与原码除法之间的差异。

无论以上哪种一位除法算法，其运算过程都是重复地交替进行加减运算与移位操作，因此，一位除法器的组成与一位乘法器是相似的。在一位除法器中，由于被除数位数是除数位数的两倍，除数用一个寄存器存放，而被除数需用两个寄存器存放（这两个寄存器也用于存放运算过程中的部分余数）。为了便于实现，采用部分余数左移来取代除数右移，且每次上的商也随部分余数左移，从最低位进入存放被除数的寄存器。当运算结束时，原用于存放被除数的两个寄存器中，低位寄存器中存放的是商，而高位寄存器中存放的是余数（对小数除法，还需将余数部分再右移 n 位，n 是除数的数字位数）。一位除法器的主要缺点，就是运算速度慢，控制复杂。

3.3.3 阵列除法器

阵列除法器建立在大规模集成电路的基础上,它通过合理的硬件设计,简化了运算控制,提高了运算速度。下面以原码加减交替法阵列除法器为例,说明阵列除法器的组成及工作原理。

原码加减交替法阵列除法器要求被除数与除数均以正数表示,商和余数的符号另行处理。由于试商时,既有减除数试商,也有加除数试商,因此,每次试商都要用到加/减法器。构成加/减法器的基本电路称为可控加/减(CAS)单元,如图 3.11 所示。

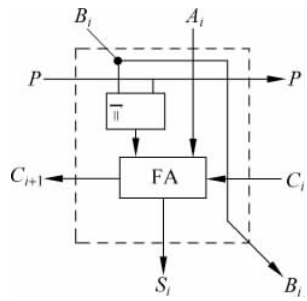


图 3.11 可控加/减(CAS)单元

一个 CAS 的功能,其实是与行波进位加/减法器(图 3.3)的一个 1 位单元电路相同的。其中,A 是被加数(或被减数);B 是加数(或减数);P 是控制加/减的控制信号,$P=0$,做加法,$P=1$,做减法。

设被除数的数字位数为 6 位,除数的数字位数为 3 位,则对应规模的阵列除法器如图 3.12 所示。图中,被除数 x 和除数 y 都添加了一个正号 0;阵列的每行都是一个 4 位行波进位加/减法器,用于完成一次减(或加)除数试商。由于第一次试商必为减除数试商,因此,第一行的加/减控制信号 $P=1$。阵列各行的排列方式,体现了每次除数右移 1 位的特点,因此,运算过程中不用再做右移操作。每次上商的产生,以及各次试商时的加/减控制,是阵列除法器的技术关键。图中以每行加/减法器的最高进位输出作为每次上的商,并以本次上的商作为下次试商时的加/减控制,这种设计原理,读者可以结合例 3.7 来分析和理解。部分余数的最高位是符号位,如果最后一次产生的部分余数为负,则需要恢复余数;如为小数除法,还需将阵列产生的余数再右移 n 位(n 为除数的数字位数)。当第一行所上的商为 1 时,产生除法溢出,因此,第一行的最高进位输出可以作为“除法溢出”的状态标志。

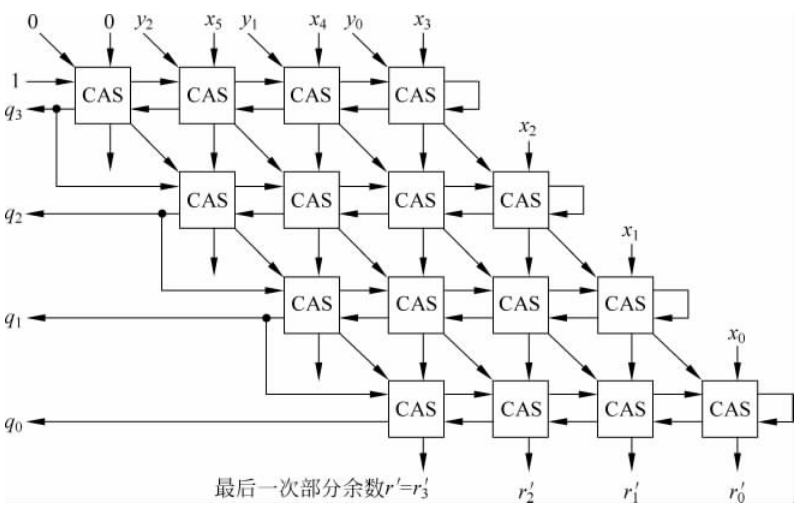


图 3.12 加/减交替法阵列除法器逻辑电路图

仿照间接补码阵列乘法器(见图 3.10)的设计方法,添加算前求补器、算后求补器和符号处理逻辑,就能构成间接补码阵列除法器。

3.4　定点运算器的组成与结构

定点运算器不仅要有定点算术运算的功能,还要具有逻辑运算、移位操作等功能。

3.4.1　逻辑运算与移位操作

逻辑运算是计算机进行判断、实现控制等操作的重要手段。计算机中的逻辑运算,主要是指“逻辑非”、“逻辑加”、“逻辑乘”、“逻辑异”4 种基本运算(详见 2.2 节)。

移位操作是计算机中的一类特殊操作,起着其他操作无法替代的作用。移位操作是通过具有移位功能的寄存器(称为移位寄存器)来完成的。常见的移位操作有:左移、右移、循环左移、循环右移、带进位循环左移、带进位循环右移等。其中,左移和右移又有逻辑移位和算术移位之分。逻辑移位总是在移空位(左移后,最低位成为移空位;右移后,最高位成为移空位)上补 0,可用于单纯的移位操作,也可用于对无符号数乘以 2(左移 1 位)或除以 2(右移 1 位)的运算。算术移位用于对以补码表示的数进行移位,算术左移 1 位后,移空位上补 0,若符号位未因移位而改变,则相当于对原数乘以 2;算术右移则规定移空位上复制原来的最高位(即保持符号位不变),因此,算术右移 1 位,相当于对原数除以 2。

移位操作通常还要涉及进位/借位标志触发器,移位操作时,从移位寄存器中移出的位,总是被移入该标志触发器。图 3.13 描述了各类移位操作的功能,其中,C 表示进位/借位标志触发器;R 表示移位寄存器。

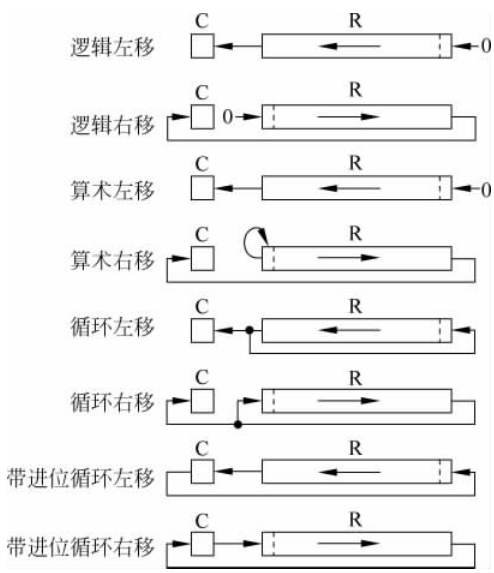


图 3.13　移位操作的种类及功能

3.4.2 算术逻辑单元的功能设计

算术逻辑单元(ALU)是组成运算器的核心器件,其主要功能是执行定点数算术加/减法运算及各种逻辑运算。早期的一位乘法器和一位除法器也以 ALU 为主,配合移位寄存器等辅助电路构成。现在的乘/除运算均采用高速的阵列乘法器和阵列除法器来完成,不再通过 ALU 来执行。

ALU 的核心是加法器,无论是算术运算,还是逻辑运算,最终均通过加法器求得结果。显然,如果直接将两个运算数据送入加法器,则只能求出两数的和,实现单一的加法运算。要使一个加法器能完成多种不同的运算功能,不能直接将运算数据送入加法器,而应先对运算数据进行各种函数变换,产生不同的变换结果送入加法器,从而使运算结果发生变化,以实现不同的运算功能。

下面以一种 4 位多功能 ALU 芯片 74181 为例,说明多功能 ALU 的设计思想。图 3.14 为 74181ALU 的一位单元逻辑结构图。图中,A_i 和 B_i 表示原始运算数据 A 和 B 的第 i 位($i=0,1,2,3$),A_i 和 B_i 经函数变换后产生 X_i 和 Y_i;送入全加器相加的是 X_i 和 Y_i,而不直接是 A_i 和 B_i;S_0、S_1、S_2、S_3 为函数发生器的控制信号,用于控制函数发生器做不同的函数变换;4 个控制信号可以形成 16 种不同的控制组合,产生 16 种不同的变换结果,相当于使 ALU 具有了 16 种不同的算术或逻辑运算功能。

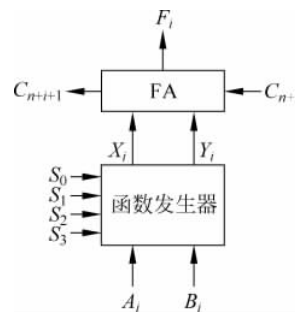


图 3.14 74181ALU 一位单元逻辑结构图

图 3.14 中,全加器的输出 F_i 和 C_{n+i+1} 的逻辑表达式为

$$\begin{cases} F_i = X_i \oplus Y_i \oplus C_{n+i} \\ C_{n+i+1} = X_iY_i + X_iC_{n+i} + Y_iC_{n+i} \end{cases} \tag{3.10}$$

函数发生器的输出与输入之间的函数关系如表 3.1 所示。

表 3.1 函数发生器的输出与输入之间的函数关系

S_0	S_1	Y_i	S_2	S_3	X_i
0	0	$\overline{A_i}$	0	0	1
0	1	$\overline{A_i}B_i$	0	1	$\overline{A_i}+\overline{B_i}$
1	0	$\overline{A_i}\,\overline{B_i}$	1	0	$\overline{A_i}+B_i$
1	1	0	1	1	$\overline{A_i}$

由表 3.1 可得 X_i 和 Y_i 的逻辑表达式如下:

$$X_i = \overline{S_2}\,\overline{S_3} + \overline{S_2}S_3(\overline{A_i} + \overline{B_i}) + S_2\,\overline{S_3}(\overline{A_i} + B_i) + S_2S_3\,\overline{A_i}$$

$$Y_i = \overline{S_0}\,\overline{S_1}\,\overline{A_i} + \overline{S_0}S_1\,\overline{A_i}B_i + S_0\,\overline{S_1}\,\overline{A_i}\,\overline{B_i}$$

进一步化简,然后代入式(3.10),得到 74181ALU 一位单元的逻辑表达式如下:

$$\begin{cases} X_i = \overline{S_3A_iB_i + S_2A_i\,\overline{B_i}} \\ Y_i = \overline{A_i + S_0B_i + S_1\,\overline{B_i}} \\ F_i = X_i \oplus Y_i \oplus C_{n+i} \\ C_{n+i+1} = Y_i + X_iC_{n+i} \end{cases} \tag{3.11}$$

当 $i=0$ 时，C_n（即 C_{n+0}）是 ALU 的最低进位输入；当 $i=3$ 时，C_{n+4}（即 C_{n+3+1}）是 ALU 的最高进位输出。

74181ALU 采用了先行进位技术，使得 4 位之间的进位同时产生。下面是用于实现先行进位的逻辑表达式：

$$\begin{cases} C_{n+1} = Y_0 + X_0 C_n \\ C_{n+2} = Y_1 + X_1 C_{n+1} = Y_1 + Y_0 X_1 + X_0 X_1 C_n \\ C_{n+3} = Y_2 + X_2 C_{n+2} = Y_2 + Y_1 X_2 + Y_0 X_1 X_2 + X_0 X_1 X_2 C_n \\ C_{n+4} = Y_3 + X_3 C_{n+3} = Y_3 + Y_2 X_3 + Y_1 X_2 X_3 + Y_0 X_1 X_2 X_3 + X_0 X_1 X_2 X_3 C_n \end{cases} \tag{3.12}$$

这些表达式中，X_i、Y_i（$i=0,1,2,3$）及 C_n 都是在加法运算前就确定的已知量，因此，各个进位都可以直接通过这些已知量先行求得，无须如行波进位那样逐位向上传递。而且，这些进位逻辑表达式的实现电路都具有相同的延时。所以，各个进位可以同时产生，每一位上的加法运算可以同时进行，这就使得 ALU 具有了并行运算的能力，提高了运算速度。

若设

$$G = Y_3 + Y_2 X_3 + Y_1 X_2 X_3 + Y_0 X_1 X_2 X_3$$

$$P = X_0 X_1 X_2 X_3$$

则 C_{n+4} 的逻辑表达式可表示为

$$C_{n+4} = G + PC_n \tag{3.13}$$

其中，G、P 两个信号也是 74181 芯片的输出信号，G 称为进位发生输出，P 称为进位传送输出。

图 3.15 所示为采用正逻辑操作数（即高电平表示 1，低电平表示 0）的 74181ALU 芯片的方框图。图中，$A_3 \sim A_0$、$B_3 \sim B_0$ 为两个 4 位的操作数（输入信号）；$F_3 \sim F_0$ 为 4 位运算结果（输出信号）；M 为运算类型选择信号（输入信号），$M=0$，做算术运算，$M=1$，做逻辑运算；$S_3 \sim S_0$ 为函数发生器的控制信号（输入信号），用于控制函数发生器做不同的函数变换，从而实现不同的运算功能；C_n 为最低进位输入信号，是低电平有效的信号，即 $C_n=0$ 时，表示最低有进位输入，加法器在相加时，会多加上 1，$C_n=1$ 时，表示最低无进位输入；C_{n+4} 为最高进位输出信号，也是低电平有效的信号，$C_{n+4}=0$，表示最高位有进位输出，$C_{n+4}=1$，表示最高位无进位输出；G 和 P 为输出信号，其逻辑表达式如上所述，其作用在下面介绍；$A=B$ 为输出信号，用于指出 A 与 B 是否相等。

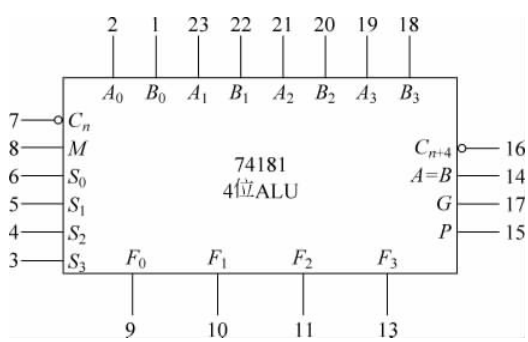


图 3.15　74181ALU 的方框图

表 3.2 所列为采用正逻辑操作数的 74181ALU 的运算功能。其中,“+”是指逻辑加;“AB”是指 A 与 B 的逻辑乘(其余类推);“加”和“减”是指算术加和算术减。算术运算部分,$C_n=1$ 表示最低位无进位输入;若 $C_n=0$,则表示最低位有进位输入,表中所列算术运算都要再加上 1。此外,减法按补码方法进行。

表 3.2 采用正逻辑操作数的 74181ALU 运算功能表

运算功能选择				运算类型选择	
S_3	S_2	S_1	S_0	逻辑运算:$M=1$	算术运算:$M=0, C_n=1$
0	0	0	0	$\overline{A}$	A
0	0	0	1	$\overline{A+B}$	$A+B$
0	0	1	0	$\overline{A}B$	$A+\overline{B}$
0	0	1	1	逻辑 0	减 1
0	1	0	0	$\overline{AB}$	A 加 $A\overline{B}$
0	1	0	1	$\overline{B}$	$(A+B)$ 加 $A\overline{B}$
0	1	1	0	$A\oplus B$	A 减 B 减 1
0	1	1	1	$A\overline{B}$	$A\overline{B}$ 减 1
1	0	0	0	$\overline{A}+B$	A 加 AB
1	0	0	1	$\overline{A\oplus B}$	A 加 B
1	0	1	0	B	$(A+\overline{B})$ 加 AB
1	0	1	1	AB	AB 减 1
1	1	0	0	逻辑 1	A 加 A
1	1	0	1	$A+\overline{B}$	$(A+B)$ 加 A
1	1	1	0	$A+B$	$(A+\overline{B})$ 加 A
1	1	1	1	A	A 减 1

当需要用多个 74181 芯片组成一个位数更多的 ALU 时,利用 C_n 及各个芯片输出的 G、P 信号,可以构成芯片之间的先行进位逻辑。如采用 4 个 74181 芯片组成一个 16 位 ALU 时,各个芯片的最高进位的逻辑表达式如下:

$$\begin{cases} C_{n+4} = G_0 + P_0 C_n \\ C_{n+8} = G_1 + P_1 C_{n+4} = G_1 + G_0 P_1 + P_0 P_1 C_n \\ C_{n+12} = G_2 + P_2 C_{n+8} = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_0 P_1 P_2 C_n \\ C_{n+16} = G_3 + P_3 C_{n+12} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_0 P_1 P_2 P_3 C_n \end{cases} \tag{3.14}$$

若令

$$G^* = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$
$$P^* = P_0 P_1 P_2 P_3$$

则

$$C_{n+16} = G^* + P^* C_n \tag{3.15}$$

可见,式(3.14)与式(3.12)所对应的逻辑电路具有相同的结构。由于各个芯片的 G、P 信号都是由 X 和 Y 生成的,所以也都是已知量,因此,各个芯片的最高进位可以先行同时产生,从而使各个芯片能够同时进行运算,实现了芯片间的并行运算。把芯片间和芯片内均采用先行进位技术的 ALU,称为全先行进位 ALU。

74182 芯片是专门与 74181ALU 芯片配套使用的先行进位部件(CLA),它按照式(3.14)和式(3.15)设计,可以为 4 个 74181ALU 芯片提供芯片间的先行进位支持,从而构成一个 16 位全先行进位 ALU,如图 3.16 所示。

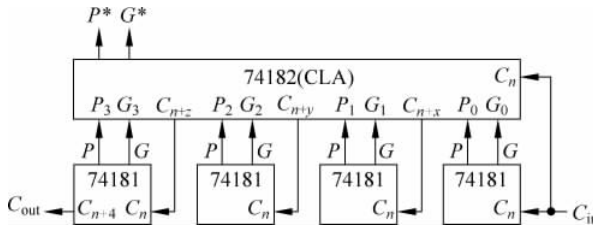


图 3.16　16 位全先行进位 ALU 逻辑方框图

图 3.16 中,C_{n+x}、C_{n+y}、C_{n+z}分别对应式(3.14)中的 C_{n+4}、C_{n+8}、C_{n+12};C_{in}是 16 位 ALU 的最低进位输入,C_{out}是最高进位输出(即 C_{n+16})。若以 16 位全先行进位 ALU 作为一个模块,则 74182CLA 的输出信号 G^* 和 P^* 可以看作一个模块的进位发生输出和进位传送输出;当需要用 4 个模块组成一个 64 位的 ALU 时,可将 C_{in}与 4 个模块的 G^* 和 P^* 输入一个 74182CLA,这样,就可以形成模块之间的先行进位。

先行进位技术可以极大地提高 ALU 的运算速度。当然,这是以增加硬件复杂度及硬件成本为代价的。

3.4.3　定点运算器的基本结构

定点运算器由 ALU、阵列乘法器、阵列除法器、通用寄存器、专用寄存器、缓冲寄存器、多路开关、三态缓冲器、数据总线等组成。其中,通用寄存器用来暂存 ALU 的运算数据或运算结果;专用寄存器包括用于移位操作的移位寄存器、用于记录各种状态标志的状态标志寄存器等;缓冲寄存器通常被置于 ALU 的输入端或输出端,用于对数据的输入或输出进行缓冲;多路开关用于在多路输入数据中选择一路送往 ALU;三态缓冲器用于控制数据通路的通、断及数据传送的方向;数据总线是连接运算器内部各组成部件的数据通路,用于在各部件之间传送数据。

运算器的设计,主要是围绕着 ALU 和寄存器同数据总线之间如何传送操作数和运算结果(即数据通路的设计)而进行的。对数据总线的控制主要采用三态缓冲器。图 3.17 所示为双向传输总线的两种常见的实现方式。

图 3.17(a)采用双向缓冲器控制数据的双向传输,总线两端可以连接任何需要发送和接收数据的部件。图 3.17(b)是专门与寄存器相连的双向传输总线,数据的发送端和接收端均为寄存器。图中的 4 位寄存器由 4 个 DE 触发器组成,其中,D 为数据输入端;E 为输入允许控制端(高电平允许输入);Q 为数据输出端。当 E=1 时,发送三态门被禁止,在时钟脉冲到来时,D 端的数据被锁存到寄存器中;当 E=0 时,只要控制发送的脉冲信号到来,发送三态门就被允许,寄存器中的数据就被输出到总线上。

根据不同的性能要求,运算器大体有单总线结构、双总线结构和三总线结构三种结构形式,如图 3.18 所示。为了重点突出总线设计上的特点,图中的运算器结构做了适当简化,只描述了其中的核心部分(即 ALU 和寄存器)与总线的连接关系。

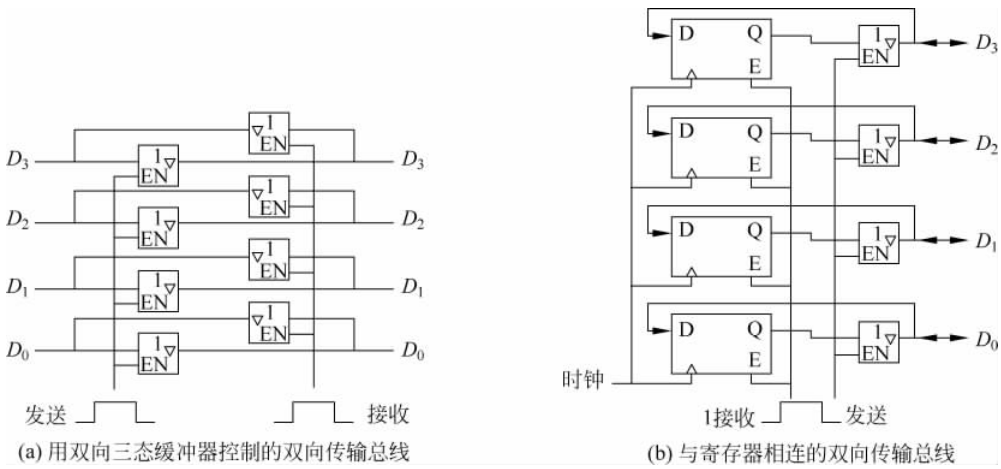


图 3.17 用三态缓冲器控制的单向传输总线和双向传输总线

如图 3.18(a)所示,在单总线结构的运算器中,所有需要参与数据传输与处理的部件都连接在同一套总线上,通过这一套总线,完成相互间的数据传输。由于在同一套总线上,一次只能传输一个数据,所以,多个数据的传输必须控制在不同的时间进行。对 ALU 而言,所需的两个操作数不能同时通过总线传输,只能分两次传输。由于 ALU 需要等两个操作数都到齐后才能进行运算,而 ALU 本身又没有数据暂存能力,因此,要在 ALU 的输入端设置两个缓冲寄存器 A 和 B。当第一个操作数出现在总线上时,先将其存入 A 缓冲寄存器,然后再将第二个操作数送上总线,并存入 B 缓冲寄存器,ALU 所需的操作数实际上是由 A 和 B 两个缓冲寄存器提供的。ALU 的输出受三态缓冲器的控制,其运算结果通过三态缓冲器输出到总线,并传输到指定的目标寄存器或主存单元。单总线结构运算器的结构简单,控制方便,其主要缺点是数据传输效率低。

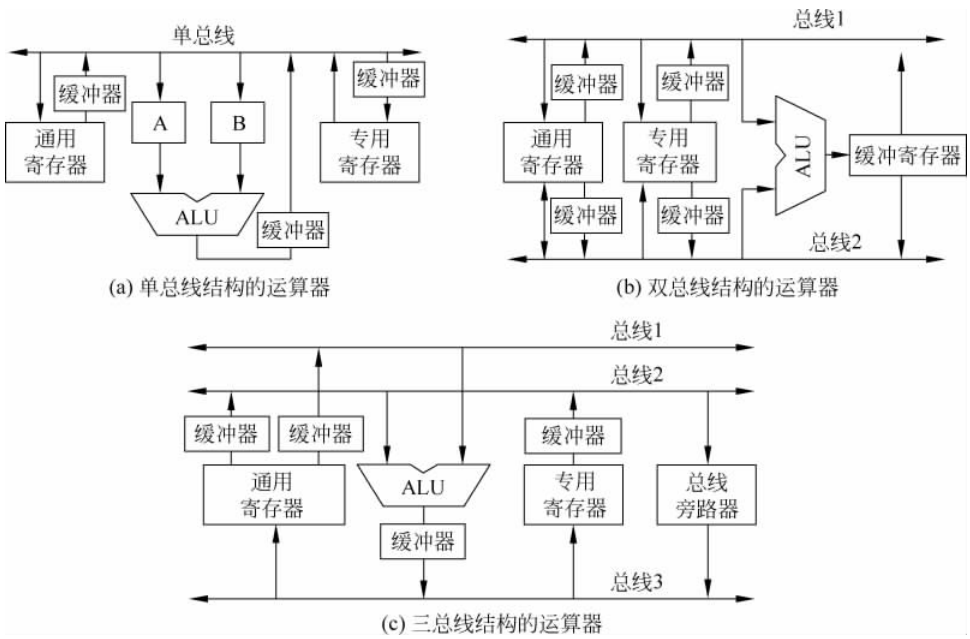


图 3.18 运算器的三种基本结构形式

图 3.18(b)所示为双总线结构的运算器。这种结构中，通用寄存器和专用寄存器既可通过总线 1 传输数据，也可通过总线 2 传输数据；ALU 所需的两个操作数分别通过总线 1 和总线 2，同时送到 ALU 的数据输入端，并在整个运算期间保持在总线上，故无须在 ALU 的输入端设置缓冲寄存器；ALU 的运算结果不能直接输出到任一总线上，因为此时操作数尚未从总线上撤销，故应先将运算结果存入缓冲寄存器，待操作数从总线上撤销后(即关闭了提供操作数的寄存器的输出三态门后)，再输出到总线上。显然，双总线结构比单总线结构具有更高的数据传输效率和灵活性，但控制也较为复杂。

在图 3.18(c)所示的三总线结构的运算器中，通用寄存器只能从总线 3 输入，但可以向总线 1 和总线 2 输出，这使它可以同时输出两个数据；ALU 的两个输入端分别与总线 1 和总线 2 相连，使它可以同时从这两套总线获得两个操作数，ALU 的运算结果则在三态缓冲器的控制下输出到总线 3；总线旁路器实际上就是一个三态缓冲器，它将总线 2 和总线 3 连接起来，其作用是实现单纯的数据传送(不做数据处理)，如将数据从一个寄存器传送到另一个寄存器。三总线结构对 ALU 提供了最好的数据传输支持，因此运算器的工作速度也最快，同时，运算器的硬件结构及其控制也是最复杂的。

以上三种结构的运算器在数据传输效率上的差异，主要表现在 ALU 与寄存器之间的数据传输上。同样是传送两个操作数和一个运算结果，对单总线结构的运算器，共需要三次访问寄存器的时间；对双总线结构的运算器，需要两次访问寄存器的时间再加上运算结果在缓冲寄存器中的等待时间；对三总线结构的运算器，则只需两次访问寄存器的时间。如果操作数取自主存，则三种结构的运算器在数据传输效率上没有多少差别，因为，运算器与主存之间只有一套数据总线，一次只能传输一个数据。

需要强调的是：

(1) ALU 无数据暂存能力。

(2) 要避免总线上的数据冲突。这就是各种器件在向总线输出数据时，要受到三态缓冲器控制的原因。

(3) 一次运算过程是分成多个步骤来完成的，如传送操作数、计算、传送运算结果等。每个步骤都是在控制器所发出的控制信号的控制下进行的。

3.5 浮点运算

浮点运算需要同时涉及尾数和阶码的运算，此外，运算过程中还需要对浮点数做规格化处理等，因此，浮点运算要比定点运算复杂得多。

3.5.1 浮点加法、减法运算

设有两个规格化浮点数 x 和 y，分别表示为

$$x = 2^{E_x} \cdot M_x$$

$$y = 2^{E_y} \cdot M_y$$

其中，E_x 和 E_y 分别为 x 和 y 的阶码；M_x 和 M_y 分别为 x 和 y 的尾数。显然，计算 $x \pm y$ 的基础是 $E_x = E_y$。如果 $E_x \neq E_y$，则需要先将它们调整为相等，这个操作称为“对阶”。

计算机中,浮点加减运算的过程大体分为以下几步:0操作数检查;比较阶码大小并完成对阶;尾数相加或相减;对结果进行处理,包括规格化、舍入处理和溢出处理。下面对每一步的操作进行详细说明。

(1) 0操作数检查。即检查 x 或 y 是否为0,只要其中任何一个为0,则可直接给出最终的运算结果,无须经过整个运算过程。这样做的目的,是为了尽可能减少运算的时间。

(2) 比较阶码大小并完成对阶。从数学角度来看,对阶时,既可以将较大的阶码调小,也可以将较小的阶码调大。当然,在将较大的阶码调小时,尾数需要相应地向左移位,而在将较小的阶码调大时,尾数需要相应地向右移位。由于浮点数的尾数是规格化的定点小数,向左移位后,其有效数字的高位部分将会丢失,从而造成很大的数据误差,因此,对阶时,不能采用将较大的阶码调小的方式,只能采用将较小的阶码调大的方式,这称为"小阶向大阶看齐"。

如何判断阶码之间的大小及差值呢?计算机采用求阶差(即将两数的阶码相减)来解决这个问题。求出阶差后,根据阶差的符号状态,可以判断出阶码之间的大小关系,而阶差的绝对值就是小阶与大阶之间的差值,只要将小阶加上这个差值,同时,以该差值作为移位次数,将对应的尾数向右移动若干位,即可实现对阶。对阶时所做的尾数右移,会将尾数有效数字的低位部分移出去,但在浮点运算器中,有专门的寄存器接收这些被移出的数据位,留待后面处理,这些数据位被称为"保护位"。

(3) 尾数相加或相减。尾数运算即为一般的定点补码加减运算,通常采用双符号位的变形补码进行运算。如设

$$[M_x]_{补} = 11.0011010,\quad [M_y]_{补} = 11.1010011$$

则 $[M_x]_{补}+[M_y]_{补}$ 为

$$\begin{array}{r} 11.0011010 \\ +\ 11.1010011 \\ \hline 10.1101101 \end{array}$$

如果单就补码加法运算而言,上面的尾数相加已产生溢出,但在浮点运算中,尾数溢出并不意味着运算结果出错,可以在后面的步骤中进行处理。

(4) 运算结果规格化。尾数运算的结果可能出现非规格化状态(包括溢出状态),需要重新规格化。规格化时,对尾数左移称为向左规格化(简称"左规");反之,称为向右规格化(简称"右规")。尾数移位的同时,阶码也必须做相应的修改,即尾数左移1位,阶码减1,尾数右移1位,阶码加1。

需要特别注意的是,尾数是用变形补码表示的,在判断其是否规格化时,不能采用对真值或原码的判断方法。具体判断方法是:当尾数未溢出(即两个符号位一致)时,若尾数的符号位与最高有效数字位不同,则已规格化,反之,则未规格化;当尾数溢出(即两个符号位不一致)时,则必为非规格化状态,此时,尾数的最高符号位代表尾数的真实符号。

例如,设尾数 M_1、M_2 和 M_3 的变形补码表示分别为

$$[M_1]_{补} = \mathbf{11}.0011010$$
$$[M_2]_{补} = \mathbf{11}.1010011$$
$$[M_3]_{补} = \mathbf{10}.1101101$$

其中,$[M_1]_{补}$ 和 $[M_2]_{补}$ 未溢出,$[M_1]_{补}$ 已规格化,$[M_2]_{补}$ 未规格化;$[M_3]_{补}$ 溢出,故未规格化。

在确认尾数未规格化后，就要对其进行规格化处理。规格化处理的原则是：如尾数未溢出，则进行左规处理，直至满足规格化要求为止；如尾数溢出，则做右规处理，只需将尾数算术右移 1 位，阶码加 1 即可。如上例中，$[M_2]_{补}$需左规处理，将$[M_2]_{补}$左移 1 位（同时将阶码减 1）后，得$[M_2]_{补}=\mathbf{11}.0100110$，已满足规格化的要求；$[M_3]_{补}$需右规处理，将$[M_3]_{补}$算术右移 1 位（同时将阶码加 1）后，得$[M_3]_{补}=\mathbf{11}.0110110\ (1)$，已满足规格化的要求（括号中的数字 1 为右移后产生的保护位，留待下一步处理）。

（5）舍入处理。在对阶或规格化处理过程中，由于尾数右移，会使尾数的低位部分被移出，形成保护位，对保护位所做的处理，称为舍入处理。对舍入处理方法的选择，将会影响到运算结果的精度。

舍入处理的方法很多，选择时主要考虑以下三方面的因素：①本身的误差要小；②积累误差要小；③容易实现。下面介绍三种舍入处理的方法。

恒舍法：也称截断法，是一种最容易实现的舍入处理方法。其做法就是直接舍去保护位。这种方法最大的缺点，就是积累误差很大，不宜用在运算精度要求比较高的场合。

恒置 1 法：这种方法也称为恒置法或冯·诺依曼法，其实现的容易程度仅次于恒舍法。其做法是，不论保护位中的数字是什么，总是将尾数有效数字的最低位置为 1。

恒置 1 法无论在正数区还是负数区的积累误差都比较小，而且绝对值相等，符号相反，正好能达到平衡。因此，恒置 1 法目前被广泛应用于各种计算机系统中。

0 舍 1 入法：这是对应于十进制的“4 舍 5 入法”的一种舍入处理方法。

在尾数以真值或原码表示时，0 舍 1 入法的规则是：如果保护位中的最高位为 0，则将保护位舍去，否则向尾数的最低有效位进 1（即加上 1）。

在尾数以补码表示时，对于正数，仍按上面针对原码的规则处理；对于负数，则需将 0 舍 1 入法修改为：当保护位中的最高位为 0，或保护位中的最高位为 1，但其余各位均为 0 时，作“舍”处理；只有在保护位中的最高位为 1，且其余位不全为 0 时，才做“入”处理。

0 舍 1 入法与恒置法相比，其主要优点是精度更高，正、负数区的积累误差更小，且能达到平衡。但 0 舍 1 入法实现起来比较困难，这是因为，0 舍 1 入法要做较多的判断，才能决定“舍”或“入”，而且，在做了“入”处理后，可能会再次导致对尾数的规格化和舍入处理。因此，0 舍 1 入法很少实际用于浮点运算器。

（6）溢出处理。只有阶码溢出，浮点数才会溢出。因此，最后还要判断阶码是否溢出。若阶码下溢，机器自动将运算结果当作 0（即机器零）；若阶码上溢，则需报告运算错误。

由于对浮点数有规格化表示的要求，所以，无论是运算数据，还是运算结果，都必须是规格化的；只有在运算过程中，允许暂时出现非规格化现象。

【例 3.12】 设 $x=2^{010}\times 0.11011011$，$y=2^{100}\times(-0.10101100)$，按浮点运算步骤，求 $x+y$。舍入处理采用 0 舍 1 入法。

解：为方便人工计算，设浮点数格式为：阶码 5 位，用双符号补码（即变形补码）表示，以便判断阶码是否溢出；尾数数字部分 8 位，用双符号补码表示，便于规格化处理。

x、y 均不为 0，且均已符合规格化要求，将其表示为浮点格式，有

$$[x]_{浮}=\mathbf{00}010,\mathbf{00}.11011011$$

$$[y]_{浮}=\mathbf{00}100,\mathbf{11}.01010100$$

(1) 求阶差并对阶。

$$[E_x]_{补}-[E_y]_{补}=[E_x]_{补}+[-E_y]_{补}=\mathbf{00}010+\mathbf{11}100=\mathbf{11}110=(-2)_{10}$$

所以,$E_x<E_y$,E_x 应向 E_y 看齐,即 E_x 加上 2,M_x 右移 2 位,得

$$[x]_{浮}=\mathbf{00}100,\mathbf{00}.00110110\ (11)$$

括弧中的 11 为保护位。

(2) 尾数相加。

尾数相加时,保护位也参与

$$\begin{array}{r} \mathbf{0\,0}.\,0\,0\,1\,1\,0\,1\,1\,0\,(11)\\ +\quad \mathbf{1\,1}.\,0\,1\,0\,1\,0\,1\,0\,0\\ \hline \mathbf{1\,1}.\,1\,0\,0\,0\,1\,0\,1\,0\,(11)\end{array}$$

(3) 规格化处理。

尾数运算结果的符号位与最高有效数字位相同,所以未规格化,应执行左规处理,即尾数左移 1 位(保护位一起移动),同时,阶码减 1,得

$$\mathbf{00}011,\quad \mathbf{11}.00010101\ (10)$$

(4) 舍入处理。

由于尾数是负数的补码,且保护位为 10,按 0 舍 1 入法,应做“舍”处理,结果为

$$\mathbf{00}011,\quad \mathbf{11}.00010101$$

(5) 判溢出。

由于阶码的两个符号位相同(为 **00**),所以阶码未溢出,运算结果正确,即

$$[x+y]_{浮}=\mathbf{00}011,\mathbf{11}.00010101$$

$$x+y=2^{011}\times(-0.11101011)$$

【例 3.13】 设 $x=2^{-101}\times(-0.11001101)$,$y=2^{-011}\times(-0.01011010)$,按浮点运算步骤,求 $x+y$。舍入处理采用 0 舍 1 入法。

解:为方便人工计算,设浮点数格式为:阶码 5 位,用双符号补码(即变形补码)表示,以便判断阶码是否溢出;尾数数字部分 8 位,用双符号补码表示,便于规格化处理。

x、y 均不为 0。由于 y 未规格化,应先将其规格化,即尾数左移 1 位,指数减 1,得

$$y=2^{-100}\times(-0.10110100)$$

于是有

$$[x]_{浮}=\mathbf{11}011,\mathbf{11}.00110011$$

$$[y]_{浮}=\mathbf{11}100,\mathbf{11}.01001100$$

(1) 求阶差并对阶。

$$[E_x]_{补}-[E_y]_{补}=[E_x]_{补}+[-E_y]_{补}=\mathbf{11}011+\mathbf{00}100=\mathbf{11}111=(-1)_{10}$$

所以,$E_x<E_y$,E_x 应向 E_y 看齐,即 E_x 加上 1,M_x 右移 1 位,得

$$[x]_{浮}=\mathbf{11}100,\mathbf{11}.10011001\ (1)$$

括弧中的 1 为保护位。

(2) 尾数相加。

$$\begin{array}{r} \mathbf{1\,1}.\,1\,0\,0\,1\,1\,0\,0\,1\,(1)\\ +\quad \mathbf{1\,1}.\,0\,1\,0\,0\,1\,1\,0\,0\\ \hline \mathbf{1\,0}.\,1\,1\,1\,0\,0\,1\,0\,1\,(1)\end{array}$$

(3) 规格化处理。

尾数运算结果的两个符号位不一致，所以未规格化，应做右规处理，即尾数算术右移 1 位，阶码加 1，得

$$\mathbf{11}101,\quad \mathbf{11}.01110010\ (11)$$

规格化处理后，形成的保护位为 11。

(4) 舍入处理。

由于尾数是负数的补码，且保护位为 11，按 0 舍 1 入法，应做“入”处理，即

$$\begin{array}{r} \mathbf{1\,1}.\,0\,1\,1\,1\,0\,0\,1\,0 \\ +\qquad\qquad\qquad\qquad 1 \\ \hline \mathbf{1\,1}.\,0\,1\,1\,1\,0\,0\,1\,1 \end{array}$$

所以，舍入处理后的结果为 **11**101，**11**.01110011。

(5) 判溢出。

由于阶码的两个符号位相同(为 **11**)，所以阶码未溢出，运算结果正确，即

$$[x+y]_{浮} = \mathbf{11}101,\mathbf{11}.01110011$$

$$x+y = 2^{-011}\times(-0.10001101)$$

3.5.2　浮点乘法、除法运算

设有两个规格化浮点数 x 和 y，分别表示为

$$x = 2^{E_x}\cdot M_x$$

$$y = 2^{E_y}\cdot M_y$$

其中，E_x 和 E_y 分别为 x 和 y 的阶码；M_x 和 M_y 分别为 x 和 y 的尾数，则浮点乘法和除法运算的规则是

$$x\times y = 2^{E_x+E_y}\cdot(M_x\times M_y)$$

$$x\div y = 2^{E_x-E_y}\cdot(M_x\div M_y)$$

也就是说，对浮点乘法，积的阶码是被乘数与乘数的阶码之和，积的尾数是被乘数的尾数与乘数的尾数相乘之积；对浮点除法，商的阶码是被除数与除数的阶码之差，商的尾数是被除数的尾数与除数的尾数相除之商。

浮点乘法与除法运算涉及尾数相乘与相除、阶码相加与相减，这实际上都是定点数之间的运算。考虑到实际的计算机中，浮点数的阶码较多采用移码表示，下面介绍移码加减运算的方法。

(1) 直接移码加、减法。运算规则如下：

$$[a]_{移}\pm[b]_{移} = [a\pm b]_{补}$$

由于移码与补码仅符号相反，因此，在未产生溢出时，只要将运算结果$[a\pm b]_{补}$的符号取反，即可得到用移码表示的运算结果$[a\pm b]_{移}$(读者可以从移码和补码的数学定义来推导上面的运算规则)。

为了便于溢出判断，可采用双符号位移码进行运算，且规定最高符号位恒取 0，则溢出判断规则是：当运算结果$[a\pm b]_{补}$的两个符号不同时，无溢出；当运算结果$[a\pm b]_{补}$的两个符号相同时，有溢出。且当$[a\pm b]_{补}$的两个符号为 00 时，产生下溢，为 11 时，产生上溢。

(2) 移码和补码混合加、减法。运算规则如下:

$$[a]_{移}+[b]_{补}=[a+b]_{移}$$

$$[a]_{移}+[-b]_{补}=[a-b]_{移}$$

用这种方法进行阶码的加减运算时,虽然可以直接得到用移码表示的运算结果,但在运算前,需要将加数或减数从移码转换成补码。

为便于判断溢出,可对运算数采用双符号位表示,且移码的双符号位中,最高符号位恒取0。当运算结果的最高符号位为1时,发生溢出,此时,若较低的符号位为0,则发生上溢,否则发生下溢。而当运算结果的最高符号位为0时,未溢出,其较低的符号位即为运算结果的实际符号位。

下面列出浮点乘、除法运算的步骤:

(1) 0操作数检查;

(2) 阶码相加、减,并作溢出判断;

(3) 尾数相乘、除;

(4) 规格化处理;

(5) 舍入处理;

(6) 溢出判断。

3.5.3 浮点运算部件

浮点运算涉及阶码运算和尾数运算,阶码运算只做加减运算,而尾数运算要做加、减、乘、除4种运算。无论阶码运算还是尾数运算,实际均为定点运算,因此,浮点运算部件的核心,实际上是两个定点运算部件。其中,阶码运算部件用来完成阶码加、减,以及对阶或规格化时对阶码的调整,此外,还要控制对阶时尾数右移的次数;尾数运算部件则用来完成尾数的四则运算,以及尾数的规格化处理等。

一些较早的计算机系统中,浮点运算部件是一个可选的部件,如果选用,就可以通过执行浮点指令来完成浮点运算,运算速度快;如果不用,也可以用定点指令编程,在定点运算器上模拟浮点运算,但运算速度慢。现在的计算机系统中,浮点运算部件已被集成到CPU芯片中,且采用流水式并行工作方式,因此大大提高了浮点运算的速度。

习题

1. 用变形补码计算 $x+y$,指出运算结果是否溢出,如未溢出,则给出其真值表示。

(1) $x=+10011, y=+01110$ (2) $x=-10101, y=+11101$ (3) $x=-01101, y=-10011$

2. 用变形补码计算 $x-y$,指出运算结果是否溢出,如未溢出,则给出其真值表示。

(1) $x=+10001, y=+11011$ (2) $x=+10010, y=-01111$ (3) $x=-10000, y=+10100$

3. 设 $x=+110101, y=-101011$,用补码一位乘法(布斯算法)计算$[x\cdot y]_{补}$。

4. 设 $x=+1100101, y=-1011$,用原码一位除法之加减交替法计算$[x\div y]_{原}$。

5. 设 $x=-0.100111$，$y=+0.1011$，用补码一位除法之加减交替法计算$[x\div y]_{补}$。

6. 设浮点数格式为：阶码 3 位数字位，尾数 8 位数字位，且都用补码表示。按浮点运算步骤，计算 $x+y$ 和 $x-y$。舍入处理采用 0 舍 1 入法。

(1) $x=2^{100}\times 0.11001010$，$y=2^{011}\times(-0.11001011)$

(2) $x=2^{-010}\times(-0.01101011)$，$y=2^{-100}\times(-0.11010101)$

(3) $x=2^{-001}\times 0.10010110$，$y=2^{010}\times(-0.01110010)$

7. 要求按式(3.12)设计出 4 位 ALU 的先行进位逻辑电路。

8. 设计一个串行加法器，能够计算两个 n 位数据之和。设计要求是：用于相加的两个 n 位数分别存放于 A 和 B 寄存器中，和存入 A 寄存器，且只允许使用一个全加器来对 n 位数据逐位进行计算。试画出该串行加法器的逻辑电路框图。