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。本章将在第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

$$[x]_{补} = 2^n + x, \quad [y]_{补} = 2^n + y \quad (\mathrm{mod}\ 2^n)$$

所以

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

同理

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

由此可得如下的定点整数补码加、减运算规则：

$$[x]_{补} + [y]_{补} = [x + y]_{补} \quad (\mathrm{mod}\ 2^n) \tag{3.1}$$

$$[x]_{补} - [y]_{补} = [x]_{补} + [-y]_{补} = [x - y]_{补} \quad (\mathrm{mod}\ 2^n) \tag{3.2}$$

同理，可得定点小数补码加、减运算规则：

$$[x]_{补}+[y]_{补}=[x+y]_{补}\quad(\mathrm{mod}\ 2^1)\tag{3.3}$$

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

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；这完全符合我们的预期。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

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

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

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

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

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

从运算结果来看，符号位为 1，说明为负数。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？

我们知道，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。在例 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

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

其中，x_{n-1}、y_{n-1}和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)包含如下两方面的含义：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(见例 3.3)。

这种“溢出“检测方法的检测电路较复杂、延时较大，如图 3.1(a)所示。

(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

$$\begin{array}{rl} \mathbf{0\,0}\,1\,0\,1\,0\,1\,1\,0 & [x]_{补} \\ +\ \mathbf{0\,0}\,1\,0\,0\,1\,0\,0\,1 & [y]_{补} \\ \hline \mathbf{0\,1}\,0\,0\,1\,1\,1\,1\,1 & [x+y]_{补} \pmod{2^9} \end{array}$$

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

$$[x+y]_{补} = s_ns_{n-1}s_{n-2}\cdots s_1s_0$$

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？读者自己分析。

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。

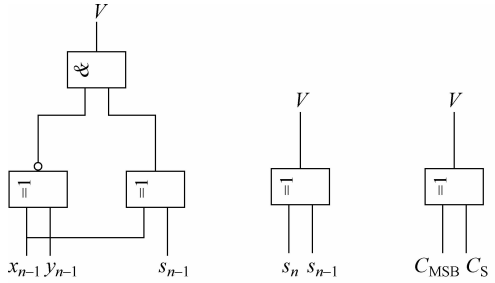


图 3.1　三种“溢出”检测电路

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。

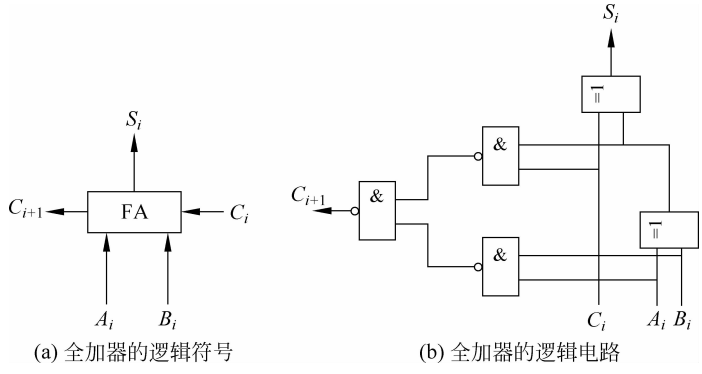


图 3.2　全加器的逻辑符号和逻辑电路

在图 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

$$\left.\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \\ C_{i+1} &= A_iB_i + A_iC_i + B_iC_i = A_iB_i + (A_i \oplus B_i)C_i = \overline{\overline{A_iB_i} \cdot \overline{(A_i \oplus B_i)C_i}} \end{aligned}\right\} \quad (3.8)$$

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。而我们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$[B]_{补}$；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。

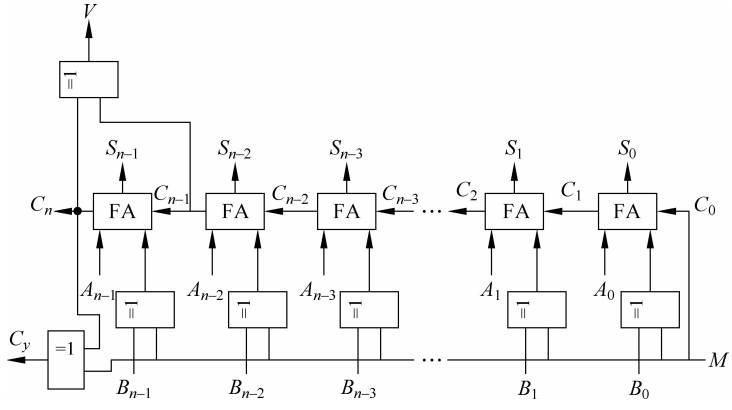


图 3.3　n 位行波进位补码加法/减法器

在图 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：

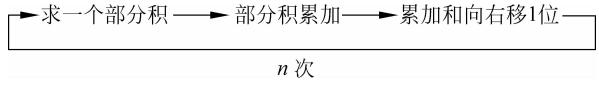
$$
\begin{array}{r}
1\ 0\ 1\ 0 \\
\times\ 1\ 1\ 0\ 1 \\
\hline
1\ 0\ 1\ 0 \\
0\ 0\ 0\ 0 \\
1\ 0\ 1\ 0 \\
1\ 0\ 1\ 0 \\
\hline
1\ 0\ 0\ 0\ 0\ 0\ 1\ 0
\end{array}
$$

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二进制乘法的步骤如下：



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 R0、R1、R2 均为 n 位寄存器，其中，R1 和 R2 分别存放乘数和被乘数的 n 位数字位(符号位另行处理)，R0 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。

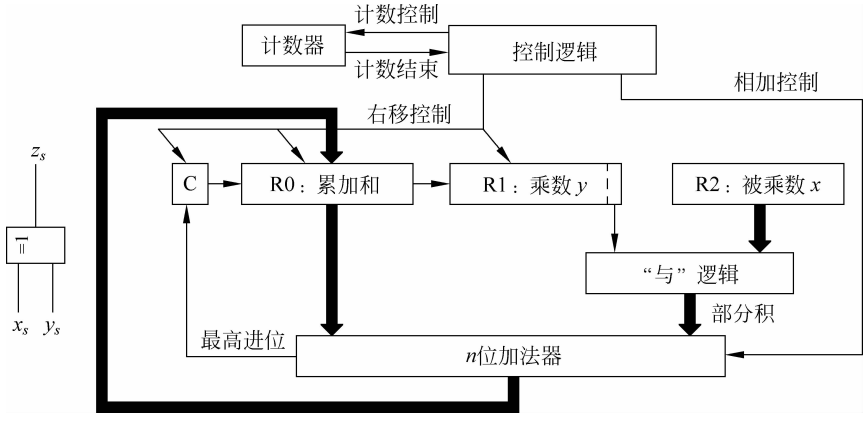


图 3.4 原码一位乘法逻辑电路框图

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 R1 和 R2 寄存器中，同时将 R0、C 和计数器清零；

(2) 由 R1 的最低位(也就是乘数的当前位)与 R2(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 R0 中的数相加，得到部分积的累加和，并送回 R0 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、R0 和 R1 串联起来向右移动 1 位，即 C 的值移到 R0 的最高位，R0 的最低位移到 R1 的最高位，R1 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 R1 中，高位数字部分在 R0 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。A 寄存器的初值为 0，最终存放的是用补码表示的乘积；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。

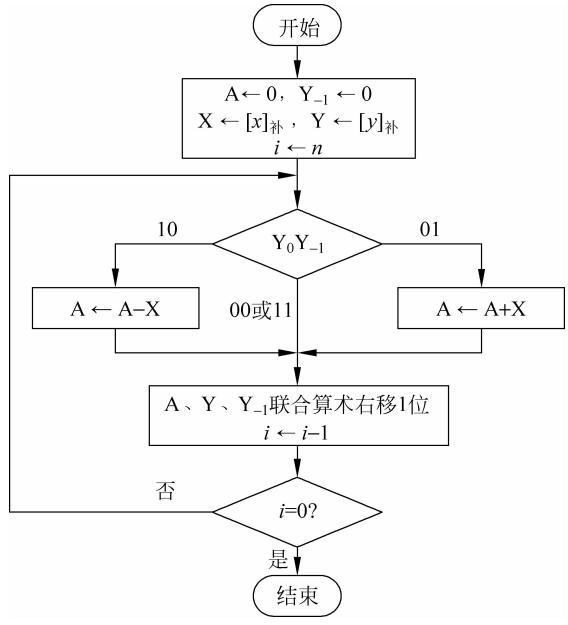


图 3.5　补码乘法的布斯算法流程图

【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_0	Y_{-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

$$[x \cdot y]_{补} = \mathrm{A} = 11011100110110$$

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，符号位保持不变。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，其平均运算速度比原码一位乘法快。

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。

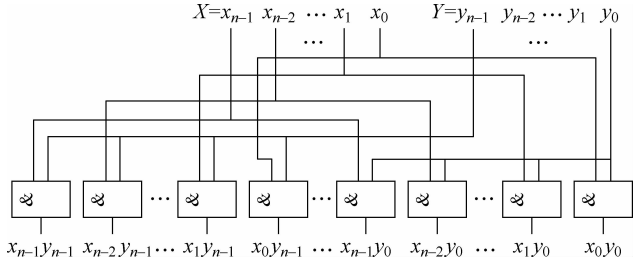


图 3.6 部分积产生电路

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 3.7 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作处理。乘法阵列的输出，即为所求的乘积。

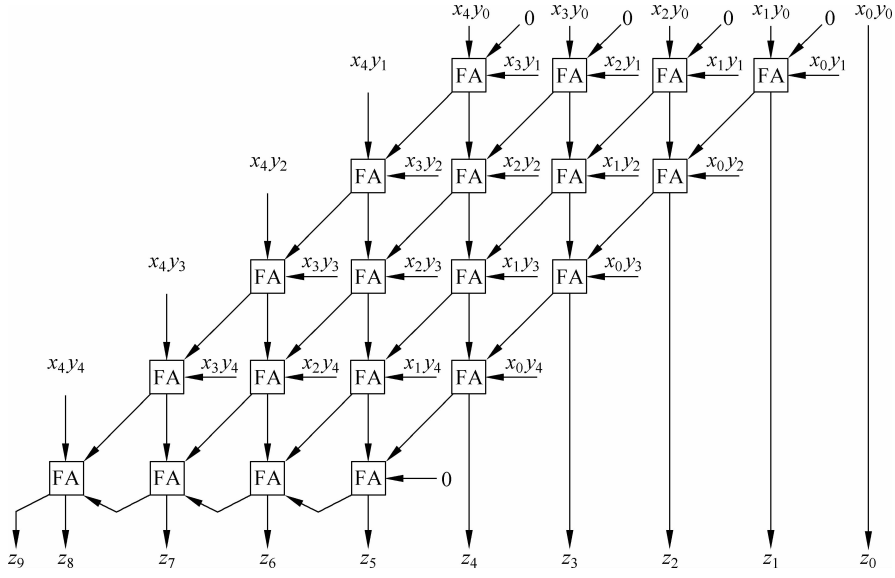


图 3.7　用于两个 5 位无符号数相乘的乘法阵列逻辑电路

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大约是一位乘法器的 $n/4$ 倍。

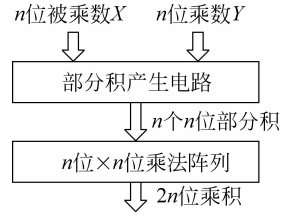


图 3.8　无符号数阵列乘法器组成框图

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。