第 2 章　运算方法和运算器

计算机的基本功能是进行算术运算和逻辑运算，运算器就是进行这些运算的功能部件。本章将主要讲述数值数据和非数值数据的表示方法、定点加减乘除运算的运算方法和运算器结构、逻辑运算和移位运算、定点运算器的组成、浮点加减乘除运算的运算方法和运算器结构，以及数据校验码等。

2.1　数值数据的表示方法

2.1.1　数据格式

计算机中数据的小数点并不是用某个二进制数字来表示的，而是用隐含的小数点的位置来表示。根据小数点的位置是否固定，将计算机中的数据表示格式分为两种，即定点格式和浮点格式。一般来说，定点格式所表示的数的范围有限，但运算复杂度和相应的处理硬件都比较简单，而浮点格式所表示的数的范围很大，但运算复杂度和相应的处理硬件都比较复杂。

1. 定点数的表示方法

所谓定点格式，是指在数据表示时，约定机器中所有数据的小数点的位置是固定不变的。由于小数点的位置是固定的，因此在数据存储和运算时，就不必专门用某个二进制数字来表示小数点。我们把用定点格式表示的数称为定点数。在计算机中，通常将定点数表示成纯小数或纯整数。

假设用一个 n+1 位字来表示一个定点数 x，其中数的符号称为数符，占 1 位，放在数据的最高位，并用数值 0 或 1 分别表示正号或负号；数的量值称为尾数，占 n 位。对于任意一个 n+1 位的定点数 x，在定点机中可表示成如图 2.1 所示的格式。

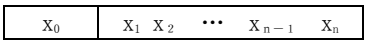


图 2.1　定点数的表示格式

如果数 x 表示的是纯小数，那么小数点在 x_0 和 x_1 之间，即数符和尾数之间。如果数 x 表示的是纯整数，那么小数点在 x_n 后面，即数据的最后。定点纯小数和定点纯整数的表示范围与数的机器码表示有关，在后面介绍各种数的机器码表示时，再详细讨论。本章后面所提到的定点小数均是指定点纯小数，定点整数均是指定点纯整数。

2. 浮点数的表示方法

在科学计算中，常常会遇到非常大或非常小的数值，如果用定点数来表示的话，很难同时满足数据的表示范围和运算精度的要求。为了解决这一问题，计算机中采用了浮点格式。所谓浮点格式，是指在数据表示时，将浮点数的范围和精度分别表示，相当于小数点的位置随比例因子的不同而在一定的范围内可自由浮动。我们把用浮点格式表示的数称为浮点数。

对于一个任意进制数 N，均可表示成 $N=M\times R^E$，比如十制数表示中的 23.67×10^{-2}、0.68×10^{3}。在式中 M 称为浮点数的尾数，用定点小数表示，值可正可负，尾数的符号就是浮点数的符号，尾数的位数决定了浮点数的表示精度；E 称为浮点数的阶码，即通常所说的指数，用定点整数表示，值可正可负，其位数决定了浮点数的表示范围；R 称为浮点数阶码的基数，在二进制浮点数据表示中，R 的取值通常为 2，由于 R 的取值是默认的，因此，在浮点数的表示格式中省去了对 R 的表示。

（1）浮点数的表示格式

在早期的计算机中，一个浮点数在机器中的表示格式，通常由阶码和尾数两部分组成。其中阶码又包括阶符和阶码值两部分，尾数又包括数符和尾数值两部分，如图 2.2 所示。

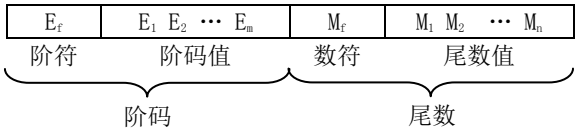


图 2.2　浮点数的表示格式

在上述浮点数的表示格式中，阶符占 1 位，阶码值占 m 位，数符占 1 位，尾数值占 n

位。由于尾数用定点小数表示，尾数的小数点位于数符与尾数值之间。由于阶码用定点整数表示，阶码的小数点位于阶码值的最后。浮点数的表示范围与尾数和阶码采用的机器码表示有关，一般来说，浮点数的尾数常用原码或补码表示，而阶码常用移码或补码表示。

后来为便于软件移植，IEEE754 规定了浮点数表示标准，这包括定义了单精度（32 位）和双精度（64 位）两种常规格式，如图 2.3 所示，以及两种扩展格式。限于篇幅，这里只介绍两种常规格式。

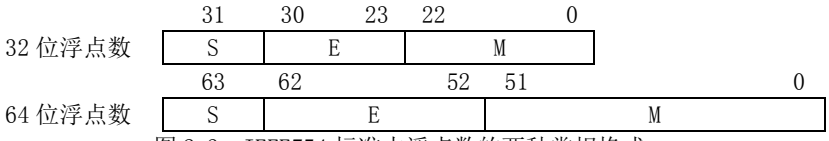


图 2.3　IEEE754 标准中浮点数的两种常规格式

32 位浮点数和 64 位浮点数中阶码的基数都是 2。32 位浮点数格式中，S 是浮点数的符号位，占 1 位，S=0 表示正数，S=1 表示负数；M 是浮点数的尾数，放在低位部分，占 23 位，小数点放在浮点数格式中的 E 和 M 之间，即 M 的最前面，实际尾数的取值为 1.M；E 是浮点数的阶码，占 8 位，阶符采用隐含方式。64 位浮点数格式中 S、E 和 M 的含义与 32 位浮点数格式相同，不同的是 64 位浮点数格式中的 M 占 52 位，E 占 11 位。

（2）浮点数的规格化

在浮点数的表示中，若不对浮点数的表示作出明确规定，同一个浮点数则可表示成多种不同的形式。例如，$(9.25)_{10}$可以表示成 1001.01×2^0，也可以表示成 100.101×2^1，还可以表示成 10.0101×2^2、1.00101×2^3、0.100101×2^4、0.0100101×2^5 等多种形式。为了使浮点数的表示方法有尽可能高的精度，充分利用尾数的有效数位，同时也是为了使浮点数的表示具有惟一性，通常采用浮点数规格化形式。即当尾数的值不为全 0 时，规定尾数的最高位必须是一个有效值。规格化浮点数定义如下：

若尾数用双符号位原码表示时，则规格化正数的尾数形式为 00.1××…××，规格化负数的尾数形式为 11.1××…××；

若尾数用双符号位补码表示时，则规格化正数的尾数形式为 00.1××…××，规格化负数的尾数形式为 11.0××…××。

对于非规格化的浮点数，要进行尾数的规格化处理，尾数每向左移动 1 位，阶码减 1；当尾数溢出时，要进行尾数右移的规格化处理，尾数向右移动 1 位，阶码加 1。

在 IEEE754 标准中，尾数用原码表示，尾数的符号即浮点数的符号，由 S 来表示。因为规格化浮点数尾数域最左位（最高有效位）总是 1，故这一位经常不予存储，而认为隐藏在小数点的左边。

在 IEEE754 标准中，一个规格化的 32 位浮点数 x 的真值可表示为：

$$x=(-1)^S\times(1.M)\times2^{E-127} \qquad (2.1)$$

其中 S、M、E 分别为 32 位浮点数表示格式和存储格式中的数符、尾数和阶码。公式中的 E-127 表示浮点数 x 的指数 e，即 e=E-127 或 E=e+127。

在 IEEE754 标准中，一个规格化的 64 位浮点数 x 的真值可表示为：

$$x=(-1)^S\times(1.M)\times2^{E-1023} \qquad (2.2)$$

其中 S、M、E 分别为 64 位浮点数表示格式和存储格式中的数符、尾数和阶码。公式中的 E-1023 表示浮点数 x 的指数 e，即 e=E-1023 或 E=e+1023。

当阶码 E 为全 0 或全 1 时用于表示包括±0 和±∞等特殊值。对于 32 位的规格化浮点数，E 的范围是从 1 到 254，对应真值指数的范围是从-126 到+127。32 位浮点数表示的数的绝对值范围是 10^{-38}～10^{38}（以 10 的幂表示）。对于 64 位的规格化浮点数，E 的范围是从 1 到 2046，对应真值指数的范围是从-1022 到+1023。64 位浮点数表示的数的绝对值范围是 10^{-308}～10^{308}（以 10 的幂表示）。

当阶码 E 为全 0 且尾数 M 也为全 0 时，表示的真值 x 为 0。结合符号 S 为 0 或 1，有正零和负零之分。如果阶码 E 为全 0 而尾数 M 不为全 0，这是一种非规格化数，表示的值为 $\pm(0.M)\times2^{E-126}$（32 位浮点数）或 $\pm(0.M)\times2^{E-1022}$（64 位浮点数）。

当阶码 E 为全 1 且尾数 M 为全 0 时，表示的真值 x 为无穷大。结合符号 S 为 0 或 1，有

+∞和-∞之分。如果阶码 E 为全 1 而尾数 M 不为全 0，表示的是一个非数，即由零除以零、取负数平方根或无穷大减无穷大等无效操作的结果。

［**例 2.1**］ 若浮点数 x 的 IEEE754 标准的 32 位存储格式为$(C2540000)_{16}$，求其浮点数的十进制数值。

解：首先将十六进制数转换成二进制数，然后根据 IEEE754 标准中 32 位浮点数的表示格式，将二进制数分成 S、E 和 M 三部分。

1100 0010 0101 0100 0000 0000 0000 0000

S E（8 位） M（23 位）

即 S=1，E=10000100=$(132)_{10}$，M=10101000000000000000000

包括隐藏位的尾数 1.M=1.10101000000000000000000=1.10101

根据 IEEE754 标准中的 32 位浮点数真值与存储格式之间的转换公式

$$x=(-1)^S\times(1.M)\times 2^{E-127}$$

有：

$$\begin{aligned}x&=(-1)^1\times(1.10101)\times 2^{132-127}\\&=-(1.10101)\times 2^5\\&=-110101\\&=(-53)_{10}\end{aligned}$$

［**例 2.2**］ 将数$(35.875)_{10}$转换成 IEEE754 标准的 32 位浮点数的二进制存储格式。

解：首先将十进制数 35.875 转换成二进制数：

$(35.875)_{10}=(100011.111)_2$

然后将二进制数表示成浮点数形式，并使其尾数为 1.M 的形式。

$$100011.111=1.00011111\times 2^5$$

根据 IEEE754 标准中的 32 位浮点数真值与存储格式之间的转换公式

$$x=(-1)^S\times(1.M)\times 2^{E-127}$$

有：

S=0，E=$(5)_{10}$+$(127)_{10}$=$(132)_{10}$=$(10000100)_2$，M=00011111000000000000000

最后得到该 32 位浮点数的二进制存储格式为：

0100 0010 0000 1111 1000 0000 0000 0000=$(420F8000)_{16}$

3. 十进制数串的表示方法

大多数通用性较强的计算机都能直接处理十进制形式表示的数据。十进制数串在计算机内主要有两种表示形式：

（1）字符串形式

在字符串表示形式中，一个字节存放一个十进制的数位或符号。在主存中，这样的一个十进制数占用连续的多个字节，故为了指明一个十进制数，需要给出该数在主存中的起始地址和位数（串的长度）。这种方法表示的十进制字符串主要用在非数值计算的应用领域中。

（2）压缩的十进制数串形式

在压缩的十进制数串表示形式中，一个字节存放两个十进制的数位。它比前一种形式节省存储空间，又便于直接完成十进制数的算术运算，是广泛采用的较为理想的方法。

用压缩的十进制数串表示一个十进制数，也要占用主存连续的多个字节。每个数位占用半个字节，即用 4 位二进制表示一位十进制数，其值可用 BCD（Binary Code for Decimal）码或数字符的 ASCII 码的低 4 位表示。符号位也占半个字节并存放在最低数字位之后，其值选用 4 位二进制编码中的六种冗余状态中的有关值，如用 12（C）表示正号，用 13（D）表示负号。在这种表示中，规定数位加符号位之和必须为偶数，当和不为偶数时，应在最高数字位之前补一个 0。例如+239 和-56 分别被表示成：

2	3	9	C

(+239)

0	5	6	D

(-56)

在上述表示中，一个实线框表示一个字节，虚线把一个字节分为高低各半个字节，每半个字节给出一个数字位或符号位的编码值（用十六进制形式给出）。符号位在数字位之后。

与第一种表示形式类似，要指明一个压缩的十进制数串，也得给出它在主存中的首地址和数字位个数（不含符号位），又称位长，位长为 0 的数其值为 0。十进制数串表示法的优点是位长可变，许多机器中规定该长度从 0 到 31，有的甚至更长。

2.1.2 数的机器码表示

所谓无符号数，就是整个机器字长的全部二进制位均表示数值位（没有符号位）。若机器字长为 n+1 位，无符号数的表示范围为 $0\sim2^{n+1}-1$。如机器字长为 8 位，则无符号数的表示范围为 0～255。

所谓有符号数，就是用正、负符号加绝对值来表示数的大小，这种按一般书写形式表示的数值在计算机技术中称为真值。但计算机中所能表示的数或其它信息都是数码化的，对于一个有符号数，要将数的符号连同数一起编码，并作为数的一部分同数一起参与运算。这种在机器中使用的连同数符一起进行编码的数称为机器数或机器码。在计算机中根据运算方法的需要，数的机器码表示往往会不相同，常见的有原码、反码、补码和移码四种表示方法。

1. 原码的表示法

原码表示法是一种比较直观的机器码表示法。原码的最高位作为符号位，用“0”表示正号，用“1”表示负号，有效值部分用二进制数的绝对值表示。定点小数和定点整数的原码表示定义如下：

对于定点小数，设 $[x]_{原}=x_0.x_1x_2\cdots x_n$，共 n+1 位，其中 x_0 为符号位，则

$$[x]_{原}=\begin{cases}x & 0\leqslant x\leqslant 1-2^{-n}\\ 1-x=1+|x| & -(1-2^{-n})\leqslant x\leqslant 0\end{cases}\tag{2.3}$$

对于定点整数，设 $[x]_{原}=x_0x_1x_2\cdots x_n$，共 n+1 位，其中 x_0 为符号位，则

$$[x]_{原}=\begin{cases}x & 0\leqslant x\leqslant 2^{n}-1\\ 2^{n}-x=2^{n}+|x| & -(2^{n}-1)\leqslant x\leqslant 0\end{cases}\tag{2.4}$$

[**例 2.3**] 已知 $x_1=0.1101$，$x_2=-0.1010$，求 $[x_1]_{原}$、$[x_2]_{原}$。

解： $[x_1]_{原}=0.1101$

$[x_2]_{原}=1.1010$

[**例 2.4**] 已知 $x_1=1001$，$x_2=-1110$，求 $[x_1]_{原}$、$[x_2]_{原}$。

解： $[x_1]_{原}=01001$

$[x_2]_{原}=11110$

由例 2.3 和例 2.4 可以看出，不管是定点小数，还是定点整数，一个正数的原码等于符号位为 0，数值位为原来的；一个负数的原码等于符号位为 1，数值位为原来的。实际上就是将数的符号表示放到了机器码的符号位。

对于真值零，其原码有正零和负零之分，即零的原码表示不惟一。故对于定点小数和定点整数，零的表示各有两种形式。

对于定点小数，$[+0]_{原}=0.00\cdots00$，$[-0]_{原}=1.00\cdots00$

对于定点整数，$[+0]_{原}=000\cdots00$，$[-0]_{原}=100\cdots00$

如果已知一个数的原码，求它的真值的方法是：

对于定点小数，直接将符号位 0 还原成正号“+”或缺省，将符号位 1 还原成负号“-”，整数位为 0，数值位是原来的。

对于定点整数，直接将符号位 0 还原成正号“+”或缺省，将符号位 1 还原成负号“-”，数值位是原来的。

[**例 2.5**] 已知 $[x_1]_{原}=0.1011$，$[x_2]_{原}=1.0001$，$[y_1]_{原}=11001$，$[y_2]_{原}=01001$，求 x_1、x_2、y_1、y_2。

解： $x_1=0.1011$　　$x_2=-0.0001$

$y_1=-1001$　　$y_2=1001$

原码表示法的优点是直观易懂。机器码和真值之间的转换很容易，用原码实现乘、除法运算的规则很简单。缺点是实现加减运算的规则较复杂。在计算机中，加减运算中的数一般

采用补码来表示。

2. 补码表示法

在计算机中，机器字长是有限的，例如某机器字长为 32 位，两个 32 位的数据进行某种运算后，若运算的结果位数超过了 32 位，则由第 32 位向更高位产生的进位就被丢失，被丢失位的大小就是该计算机的“模”。 如果是 n+1 位定点小数（最高位为符号位），则其模为 2。如果是 n+1 位定点整数（最高位为符号位），则其模为 2^{n+1}。定点小数和定点整数的补码表示定义如下：

对于定点小数，设$[x]_{补}=x_0.x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则

$$[x]_{补}=\begin{cases}x & 0\leqslant x\leqslant 1-2^{-n}\\ & (\text{mod } 2)\\ 2+x=2-|x| & -1\leqslant x\leqslant 0\end{cases} \tag{2.5}$$

对于定点整数，设$[x]_{补}=x_0x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则

$$[x]_{补}=\begin{cases}x & 0\leqslant x\leqslant 2^{n}-1\\ & (\text{mod } 2^{n+1})\\ 2^{n+1}+x=2^{n+1}-|x| & -2^{n}\leqslant x\leqslant 0\end{cases} \tag{2.6}$$

[**例 2.6**] 已知 $x_1=0.0101$，$x_2=-0.1100$，求$[x_1]_{补}$、$[x_2]_{补}$。

解： $[x_1]_{补}=0.0101$

$[x_2]_{补}=1.0100$

[**例 2.7**] 已知 $x_1=1011$，$x_2=-0100$，求$[x_1]_{补}$、$[x_2]_{补}$。

解： $[x_1]_{补}=01011$

$[x_2]_{补}=11100$

由例 2.6 和例 2.7 可以看出，不管是定点小数，还是定点整数，一个正数的补码等于符号位为 0，数值位为原来的；一个负数的补码等于符号位为 1，数值位按位取反并在末位加 1。

对于真值零，其补码表示是惟一的。

对于定点小数，$[+0]_{补}=[-0]_{补}=0.00\cdots00$

对于定点整数，$[+0]_{补}=[-0]_{补}=000\cdots00$

如果已知一个数的补码，求它的真值的方法是：

对于定点小数，若符号位为 0，则该数为正数，补码表示的数即为真值；若符号位为 1，则该数为负数，将符号位 1 还原成负号“-”，整数位为 0，数值位按位取反并在末位加 1。

对于定点整数，若符号位为 0，则该数为正数，将符号位 0 还原成正号“+”或缺省，数值位是原来的；若符号位为 1，则该数为负数，将符号位 1 还原成负号“-”，数值位按位取反并在末位加 1。

[**例 2.8**] 已知$[x_1]_{补}=0.1011$，$[x_2]_{补}=1.0001$，$[y_1]_{补}=11001$，$[y_2]_{补}=01001$，求 x_1、x_2、y_1、y_2。

解： $x_1=0.1011$　　$x_2=-0.1111$

$y_1=-0111$　　$y_2=1001$

3. 反码表示法

定点小数和定点整数的反码表示定义如下：

对于定点小数，设$[x]_{反}=x_0.x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则

$$[x]_{反}=\begin{cases}x & 0\leqslant x\leqslant 1-2^{-n}\\ & \\ (2-2^{-n})+x & -(1-2^{-n})\leqslant x\leqslant 0\end{cases} \tag{2.7}$$

对于定点整数，设$[x]_{反}=x_0x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则

$$[x]_{反}=\begin{cases}x & 0\leqslant x\leqslant 2^{n}-1\\ & \\ (2^{n+1}-1)+x & -(2^{n}-1)\leqslant x\leqslant 0\end{cases} \tag{2.8}$$

[**例 2.9**] 已知 $x_1=0.1110$，$x_2=-0.0001$，求$[x_1]_{反}$、$[x_2]_{反}$。

解： $[x_1]_{反}=0.1110$

$[x_2]_{反}$=1.1110

[**例 2.10**] 已知 x_1=0100，x_2=-1101，求$[x_1]_{反}$、$[x_2]_{反}$。

解： $[x_1]_{反}$=00100

$[x_2]_{反}$=10010

由例 2.9 和例 2.10 可以看出，不管是定点小数，还是定点整数，一个正数的反码等于符号位为 0，数值位为原来的；一个负数的反码等于符号位为 1，数值位按位取反。

对于真值零，其反码有正零和负零之分，即零的反码表示不惟一。故对于定点小数和定点整数，零的表示各有两种形式。

对于定点小数，$[+0]_{反}$=0.00…00，$[-0]_{反}$=1.11…11

对于定点整数，$[+0]_{反}$=000…00，$[-0]_{反}$=111…11

如果已知一个数的反码，求它的真值的方法是：

对于定点小数，若符号位为 0，则该数为正数，反码表示的数即为真值；若符号位为 1，则该数为负数，将符号位 1 还原成负号“-”，整数位为 0，数值位按位取反。

对于定点整数，若符号位为 0，则该数为正数，将符号位 0 还原成正号“+”或缺省，数值位是原来的；若符号位为 1，则该数为负数，将符号位 1 还原成负号“-”，数值位按位取反。

[**例 2.11**] 已知$[x_1]_{反}$=1.1011，$[x_2]_{反}$=0.0001，$[y_1]_{反}$=01001，$[y_2]_{反}$=11001，求 x_1、x_2、y_1、y_2。

解： x_1=-0.0100　　x_2=0.0001

y_1=1001　　y_2=-0110

4. 移码表示法

移码通常用于表示浮点数的阶码，一定为整数。对于定点整数，设$[x]_{移}=x_0x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则移码表示定义为：

$$[x]_{移}=2^n+x \qquad -2^n\leqslant x\leqslant 2^n-1 \tag{2.9}$$

[**例 2.12**] 已知 x_1=-0001，x_2=1100，求$[x_1]_{补}$、$[x_2]_{补}$、$[x_1]_{移}$、$[x_2]_{移}$。

解： $[x_1]_{补}$=11111　　$[x_2]_{补}$=01100

$[x_1]_{移}$=01111　　$[x_2]_{移}$=11100

由例 2.12 可以看出，同一个数的补码与移码表示的区别仅在于符号位刚好相反。因此，若已知一个数的真值要求其移码，可以先求出这个数的补码，再将符号位取反，即可得该数的移码表示；反过来，若已知一个数的移码表示要求其真值，可以先将移码的符号位取反得到该数的补码表示，再将补码转换成该数的真值。

对于真值整数零，其移码表示是惟一的。即：$[+0]_{移}$=$[-0]_{移}$=100…00

前面我们介绍了数的四种机器码表示的定义、数的表示范围，以及真值与机器码之间相互转换的方法。若定点数的表示格式如图 2.1 所示，则各种机器码所表示的数的范围如表 2.1 所示。从表 2.1 中我们可以看出，原码和反码的表示范围相同，且所能表示的最大正数和最小负数互为相反数，所能表示的最小正数与最大负数也互为相反数；补码和移码的表示范围相同，移码只能表示定点整数。补码所能表示的最大正数与原码和反码相同，但定点小数所能表示的最小负数为-1，定点整数所能表示的最小负数为-2^n。

表 2.1　各种机器码所表示的数的范围

机器码 / 表示范围	定点小数			定点整数			
	原码	反码	补码	原码	反码	补码	移码
最大正数	$1-2^{-n}$	$1-2^{-n}$	$1-2^{-n}$	2^n-1	2^n-1	2^n-1	2^n-1
最小正数	2^{-n}	2^{-n}	2^{-n}	1	1	1	1
最大负数	-2^{-n}	-2^{-n}	-2^{-n}	-1	-1	-1	-1
最小负数	$-(1-2^{-n})$	$-(1-2^{-n})$	-1	$-(2^n-1)$	$-(2^n-1)$	-2^n	-2^n

[**例 2.13**] 已知 x=-23/64，用 8 位定点小数表示，其中最高位为符号位，求$[x]_{原}$、$[x]_{反}$、$[x]_{补}$、$[-x]_{补}$。

解：先将 x 转换成 8 位二进制数，得 x=-0.0101110，则

$[x]_{原}$=1.0101110　　$[x]_{反}$=1.1010001

$[x]_{补}$=1.1010010　　$[-x]_{补}$=0.0101110

在例 2.13 中，需要将一个分式转换成小数，其转换方法与十进制数的转换方法相同，如 $67/10^3$，分子为十进制且分母为 10 的幂次方，若分母为 10^3 则表示小数位有 3 位，转换为十进制小数为 0.067。同样的道理，$-23/64=-10111/2^6$，分子为二进制且分母为 2 的幂次方，由于分母为 2^6 则表示小数位有 6 位，转换为二进制小数为-0.010111。

［**例 2.14**］ 若机器字长为 32 位，浮点数的表示格式中阶符占 1 位，阶码值占 7 位，数符占 1 位，尾数值占 23 位，阶码和尾数均用补码表示。则该浮点数所能表示的最大正数为是多少？最小正数是多少？最大负数是多少？最小负数是多少？

解：该浮点数的表示格式如图 2.4 所示。

1 位	7 位	1 位	23 位
阶符	阶码值	数符	尾数值

图 2.4　浮点数的表示格式

按照题意，阶码和尾数均用补码表示，根据浮点数的表示格式与真值之间的关系 $N=M\times 2^E$，有：

①当 M 取最大正数 $1-2^{-23}$，且 E 也取最大正数 $2^7-1=127$ 时，所表示的浮点数为最大正数，即 $(1-2^{-23})\times 2^{127}$；

②当 M 取最小正数 2^{-23}，且 E 取最小负数 $-2^7=-128$ 时，所表示的浮点数为最小正数，即 $2^{-23}\times 2^{-128}=2^{-151}$；

③当 M 取最大负数 -2^{-23}，且 E 取最小负数 $-2^7=-128$ 时，所表示的浮点数为最大负数，即 $(-2^{-23})\times 2^{-128}=-2^{-151}$；

④当 M 取最小负数-1，且 E 取最大正数 $2^7-1=127$ 时，所表示的浮点数为最小负数，即 $(-1)\times 2^{127}=-2^{127}$。

［**例 2.15**］ 假设由 S、E、M 三个域组成一个 32 位二进制数所表示的非零规格化浮点数 x，其中 S 占 1 位，E 占 8 位，M 占 23 位，真值表示为（注意此例不是 IEEE754 标准定义的格式）：

$$x=(-1)^s\times(1.M)\times 2^{E-128}$$

问：它所表示的规格化的最大正数、最小正数、最大负数、最小负数分别是多少？

解：按照真值与浮点数表示格式之间的关系

①当 S=0，1.M 取最大值$[1+(1-2^{-23})]$，且 E-128 取最大正数$(2^8-1)-128=127$ 时，所表示的最大正数为$[1+(1-2^{-23})]\times 2^{127}$；

1 位	8 位	23 位
0	11111111	11111111111111111111111
S	E	M

②当 S=0，1.M 取最小值 1.0，且 E-128 取最小负数 0-128=-128 时，所表示的最小正数为 1.0×2^{-128}；

1 位	8 位	23 位
0	00000000	00000000000000000000000
S	E	M

③当 S=1，1.M 取最小值 1.0，且 E-128 取最小负数 0-128=-128 时，所表示的最大负数为 -1.0×2^{-128}；

1 位	8 位	23 位
1	00000000	00000000000000000000000
S	E	M

④当 S=1，1.M 取最大值$[1+(1-2^{-23})]$，且 E-128 也取最大正数$(2^8-1)-128=127$ 时，所表示的最小负数为$-[1+(1-2^{-23})]\times 2^{127}$；

1 位	8 位	23 位
1	11111111	11111111111111111111111
S	E	M

2.2　非数值数据的表示方法

计算机不但要处理数值领域的问题，而且还要处理大量非数值领域的问题，如文字、字符、字符串以及一些专用符号等。

2.2.1 字符数据的表示

1. 字符的表示

字符是计算机中使用最多的信息形式之一，是人与计算机通信、交互作用的重要媒介。在国际上普遍采用 ASCII 码（美国国家信息交换标准码）来表示字符。ASCII 码共有 128 个字符，其中 95 个编码（包括大小写各 26 个字母、10 个数字符“0”～“9”、标点符号等），对应着计算机终端能键入并可以显示的 95 个字符，打印机也可打印出这 95 个字符；另外的 33 个字符是被用来表示控制码，控制计算机某些外部设备的工作特性和某些计算机软件的运行情况。在计算机中，用一个字节表示一个 ASCII 码，低 7 位可以给出不同字符的二进制编码，最高位可以作奇偶校验位，用来检查错误，也可以用于西文字符和汉字的区分标识。

ASCII 码对英语特别适合，但对其它语言就不太合适了，如带重音符的法语、带变音符的德语，非英语字母表的俄语等。为解决这个问题，一些主要的计算机公司形成一个联盟，推出了 Unicode 字符编码系统。目前，Unicode 已被绝大多数程序设计语言和操作系统所支持。

除以上两种字符编码方式以外，使用比较广泛的编码方式还有 ANSI（美国国家标准协会）编码和 EBCDIC（扩展二、十进制交换码）。

2. 字符串的表示

随着计算机在文字处理与信息管理中的广泛应用，字符串已成为最常用的数据类型之一，许多计算机中都提供了字符串操作功能，一些计算机还设计出了能读写字符串的机器指令。

字符串是指连续的一串字符，通常方式下，它们占用主存中连续的多个字节，每个字节存放一个字符的 ASCII 码。当主存的每一个存储单元由 2 个、4 个或 8 个字节组成时，在同一个存储单元中，既有按从低位字节向高位字节的顺序存放字符串内容的，也有按从高位字节向低位字节的顺序存放字符串内容的。这两种存放方式都是常用方式，不同的计算机可以选用其中任何一种。

2.2.2 汉字的表示

汉字处理技术是我国计算机推广应用工作中必须要解决的问题。汉字数量大，字形复杂，读音多变。常用汉字有 7000 个左右。和西文相比，汉字处理的主要困难在于汉字的输入、输出和汉字在计算机内部的表示。

1. 汉字的输入

输入码是为使输入设备能将汉字输入到计算机而专门编制的一种代码。目前已出现了数百种汉字输入方案，常见的有国标码、区位码、拼音码和五笔码等。这数百种汉字输入方案按其输入码的编码方法不同可分为三类，即数字编码、拼音编码和字形编码。

（1）数字编码

常用的数字编码有国标码和区位码，它们是专业人员使用的一种汉字编码，是以数字代码来区分每个汉字的。我国在 1981 年颁布了《通用汉字字符集及其交换码标准》GB2312-1980 方案，简称国标码。它把 6763 个汉字归结在一起称为汉字基本字符集，再根据使用频度分为两级。第一级为 3755 个汉字，按拼音排序。第二级为 3008 个汉字，按部首排序。此外，还有各种图形符号、数字、字母等 682 个，总计 7445 个汉字、符号等。GB2312-1980 规定每个汉字、图形符号都用两个字节表示，每个字节只使用低 7 位编码，因此最多能表示出 128×128=16384 个汉字。

区位码是将 GB2312-1980 方案中的字符，按其位置划分为 94 个区，每个区 94 个汉字（位），区和位组成一个二维数组，每个汉字在数组中对应一个惟一的区位码。区码和位码各用两位十进制数字表示，因此输入一个汉字需按四个数字键。区位码是国标码的变形，两者之间的关系为：“国标码=区位码+2020H”。

数字编码输入的优点是无重码，且输入码与内部编码的转换比较方便，缺点是代码难以

记忆。

（2）拼音编码

常用的拼音编码有全拼、双拼、微软拼音、智能 ABC 等，拼音编码是以汉语拼音为基础的输入方法。凡掌握汉语拼音的人，不需训练和记忆，即可使用。但汉字同音字太多，输入重码率很高，因此按拼音输入后还必须进行同音字选择，影响了输入速度。

（3）字形编码

常用的字形编码有五笔字型、五笔画等，字形编码是用汉字的形状来进行编码的。汉字总数虽多，但都是由一笔一画组成，全部汉字的部件和笔画是有限的。因此，把汉字的笔画部件用字母或数字进行编码，按笔画的顺序依次输入，就能表示一个汉字。例如，五笔字型就是以字形来区分每个汉字的，它的重码少，是目前最具影响力的一种字形编码方法。

2. 汉字在机内的表示

汉字在机内的表示由汉字内码来实现，它是用于汉字信息的存储、交换、检索等操作的机内代码，一般采用两个字节表示。英文字符在机内的表示是用七位的 ASCII 码来实现的，当用一个字节表示时，最高位为“0”。为了与英文字符能相互区别，汉字内码中两个字节的最高位均规定为“1”。它是在国标码的基础上，在每个字节的最高位置“1”作为汉字标记而组成的。汉字内码与国标码两者之间的关系为：“汉字内码=国标码+8080H”。

3. 汉字的输出与汉字字库

显示器是采用图形方式来显示汉字的，汉字的输出通过采用点阵表示的汉字字模码来完成。每个汉字至少需要 16×16 的点阵才能显示，若要获得更美观的字形，则需采用 24×24、32×32、48×48 等点阵来表示。因此，一个实用汉字系统的字模码要占用很大的存储容量。以 16×16 点阵的字模码为例，每个汉字要占用 32 个字节，国标两级汉字要占用 256K 字节。因此字模点阵只能用来构成汉字库，而不能用于机内存储，汉字库中存储了每个汉字的点阵代码。

在机器中建立汉字库有两种方法。一种是将汉字库存放在硬盘中，每次需要时自动装载到计算机的内存中。用这种方法建立的汉字库称为软字库。另一种是将汉字库固化在 ROM（称汉卡）中，再插在计算机的扩展槽中，这样不占内存，只需要安排一个存储器空间给字库即可。用这种方法建立的汉字库称为硬字库。

一般常用的汉字输出只有显示输出和打印输出两种形式。输出汉字的过程为：先将输入码转换为汉字内码，然后用汉字内码检索汉字库找到其字形点阵码，再输出汉字。

2.3 定点加法、减法运算

定点加法、减法运算既可采用补码也可采用原码和移码进行，不同机器码表示其运算方法也不相同。

2.3.1 补码加法、减法

1. 补码加法

计算机中采用补码进行加法运算，并约定存储单元和运算寄存器中的数都采用补码表示，数据运算结果也用补码表示。

定点小数补码加法的运算公式为

$$[x]_{补}+[y]_{补}=[x+y]_{补} \quad (\text{mod } 2) \qquad (2.10)$$

式(2.10)的含义是：两个定点小数的补码之和等于两个数和的补码。反过来，两个数相加所得到的和的补码等于这两个数补码的和。下面分几种情况来证明这个公式。

假设 x 和 y 均采用定点小数表示，运算结果仍在定点小数的表示范围之内。即$|x|<1$，$|y|<1$，$|x+y|<1$。

(1)$x>0$，$y>0$，则 $x+y>0$。

由于参加运算的两个数都为正数，故运算结果也一定为正数。正数的补码等于其真值，根据数据补码的定义可得：

$$[x]_{补}=x \qquad [y]_{补}=y$$

所以　　$[x]_{补}+[y]_{补}=x+y=[x+y]_{补}$　　(mod 2)

(2)x＞0，y＜0，则 x+y＞0 或 x+y＜0。

当参加运算的两个数中一个为正数，另一个为负数时，运算的结果有可能为正数，也有可能为负数。根据补码的定义可得：

$[x]_{补}=x$　　$[y]_{补}=2+y$

所以　　$[x]_{补}+[y]_{补}=x+2+y=2+(x+y)$

此时可能出现两种情况：

当 x+y＞0 时，2+(x+y)＞2，2 为符号位相加产生的进位，又因为 x+y＞0，进行 mod 2 运算后，得：

$[x]_{补}+[y]_{补}=x+y=[x+y]_{补}$　　(mod 2)

当 x+y＜0 时，2+(x+y)＜2，又因为 x+y＜0，所以

$[x]_{补}+[y]_{补}=2+(x+y)=[x+y]_{补}$　　(mod 2)

(3)x＜0，y＞0，则 x+y＞0 或 x+y＜0。

这种情况与第 2 种情况类似，把 x 和 y 的位置对调即可得证。

(4)x＜0，y＜0，则 x+y＜0。

由于参加运算的两个数都为负数，故运算结果也一定为负数。根据补码的定义可得：

$[x]_{补}=2+x$　　$[y]_{补}=2+y$

所以　　$[x]_{补}+[y]_{补}=2+x+2+y=2+(2+x+y)$

由于 x+y 为负数，其绝对值又小于 1，那么(2+x+y)就一定是小于 2 而大于 1 的数，所以进行 mod 2 运算后，得：

$[x]_{补}+[y]_{补}=2+x+y=2+(x+y)=[x+y]_{补}$　　(mod 2)

因此，在模 2 定义下，任意两个定点小数的补码之和等于这两个数和的补码。这是补码加法的理论基础，其结论推广到定点整数后得出定点整数补码加法的运算公式为：

$$[x]_{补}+[y]_{补}=[x+y]_{补}\quad(\text{mod } 2^{n+1}) \tag{2.11}$$

在式(2.11)中，假设定点整数补码表示的格式中符号位占 1 位，尾数占 n 位。式(2.11)的证明与式(2.10)的证明相似，这里不再重复。

[**例 2.16**]　已知 x=0.0011，y=0.0111，用单符号位补码计算 x+y。

解：$[x]_{补}=0.0011$　$[y]_{补}=0.0111$

$$\begin{array}{rl} & [x]_{补}\quad 0.0011 \\ + & [y]_{补}\quad 0.0111 \\ \hline & [x+y]_{补}\quad 0.1010 \end{array}$$

所以　　x+y=0.1010

[**例 2.17**]　已知 x=0.1101，y=-0.0011，用单符号位补码计算 x+y。

解：$[x]_{补}=0.1101$　$[y]_{补}=1.1101$

$$\begin{array}{rl} & [x]_{补}\quad 0.1101 \\ + & [y]_{补}\quad 1.1101 \\ \hline & [x+y]_{补}\quad 0.1010 \end{array}$$

在模 2 定义下，符号位相加向前产生的进位要丢掉

所以　　x+y=0.1010

[**例 2.18**]　已知 x=+1001，y=-0101，用单符号位补码计算 x+y。

解：$[x]_{补}=01001$　$[y]_{补}=11011$

$$\begin{array}{rl} & [x]_{补}\quad 01001 \\ + & [y]_{补}\quad 11011 \\ \hline & [x+y]_{补}\quad 00100 \end{array}$$

在模 2^{n+1} 定义下，符号位相加向前产生的进位要丢掉

所以　　x+y=+0100

由例 2.16、例 2.17 和例 2.18 可以看出，定点小数加法运算与定点整数加法运算的区别仅在于小数点的位置不同而已，即定点小数的小数点在符号位之后，而定点整数的小数点在机器码的最后。实际上，对于定点数的其它运算，定点小数与定点整数的区别也仅在于小数点的位置不同而已，运算规则完全相同。

2. 补码减法

计算机中补码减法运算是转换成补码加法运算来实现的，它们使用同一加法器电路，从而简化了运算器的设计。

定点小数补码减法的运算公式为

$$[x]_{补}-[y]_{补}=[x-y]_{补}=[x]_{补}+[-y]_{补} \quad (\text{mod } 2) \qquad (2.12)$$

将式(2.10)中 y 换成-y，得$[x+(-y)]_{补}=[x]_{补}+[-y]_{补}$，即$[x-y]_{补}=[x]_{补}+[-y]_{补}$

所以这里只要证明$[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}$ ，即$[-y]_{补}=-[y]_{补}$，式(2.12)即可得证。现证明如下：

因为 $[x+y]_{补}=[x]_{补}+[y]_{补} \quad (\text{mod } 2)$

所以 $[y]_{补}=[x+y]_{补}-[x]_{补}$ (2.13)

又因为 $[x-y]_{补}=[x]_{补}+[-y]_{补} \quad (\text{mod } 2)$

所以 $[-y]_{补}=[x-y]_{补}-[x]_{补}$ (2.14)

将式(2.13)和式(2.14)相加得

$$\begin{aligned}[y]_{补}+[-y]_{补}&=[x+y]_{补}-[x]_{补}+[x-y]_{补}-[x]_{补}\\&=([x+y]_{补}+[x-y]_{补})-[x]_{补}-[x]_{补}\\&=[x+x]_{补}-[x]_{补}-[x]_{补}\\&=[x]_{补}+[x]_{补}-[x]_{补}-[x]_{补}\\&=0\end{aligned}$$

因此 $[-y]_{补}=-[y]_{补} \quad (\text{mod } 2)$ (2.15)

由式(2.15)可以看出，已知$[y]_{补}$求$[-y]_{补}$，实际上就是求$[y]_{补}$的相反数，这类似于 x86 汇编语言中的 NEG 指令，计算方法是：将$[y]_{补}$包括符号位一起取反并在末位加 1，即可得到$[-y]_{补}$。

因此，在模 2 定义下，任意两个定点小数的补码之差等于这两个数差的补码。其结论推广到定点整数后得出定点整数补码减法的运算公式为：

$$[x]_{补}-[y]_{补}=[x-y]_{补}=[x]_{补}+[-y]_{补} \quad (\text{mod } 2^{n+1}) \qquad (2.16)$$

在式(2.16)中，假设定点整数补码表示的格式中符号位占 1 位，尾数占 n 位。式(2.16)的证明与式(2.12)的证明相似，这里不再重复。

[**例 2.19**] 已知$[x]_{补}=0.1011$，$[-y]_{补}=1.0011$，求$[-x]_{补}$、$[y]_{补}$。

解： $[-x]_{补}=1.0100+0.0001=1.0101$

$[y]_{补}=[-(-y)]_{补}=0.1100+0.0001=0.1101$

[**例 2.20**] 已知 x=-0.0001，y=0.0101，用单符号位补码计算 x-y。

解： $[x]_{补}=1.1111$ $[-y]_{补}=1.1011$

$$\begin{array}{rr}[x]_{补} & 1.1111\\ +\ [-y]_{补} & 1.1011\\ \hline [x-y]_{补} & 1.1010\end{array}$$

在模 2 定义下，符号位相加向前产生的进位要丢掉

所以 x-y=-0.0110

[**例 2.21**] 已知 x=0.1001，y=-0.0011，用单符号位补码计算$[x]_{补}-[y]_{补}$。

解： $[x]_{补}=0.1001$ $[-y]_{补}=0.0011$

$$\begin{array}{rr}[x]_{补} & 0.1001\\ +\quad [-y]_{补} & 0.0011\\ \hline [x]_{补}+[-y]_{补} & 0.1100\end{array}$$

所以 $[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}=0.1100$

[**例 2.22**] 已知 x=-1101，y=-1010，用单符号位补码计算 x-y。

解： $[x]_{补}=10011$ $[-y]_{补}=01010$

$$\begin{array}{rr}[x]_{补} & 10011\\ +\ [-y]_{补} & 01010\\ \hline [x-y]_{补} & 11101\end{array}$$

所以 x-y=-0011

3. 溢出的概念与判断方法

在定点小数机器中，数据的表示范围为$|x|<1$（小数补码表示的最小负数除外）；在定

点整数机器中，假设定点整数补码表示的格式中符号位占 1 位，尾数占 n 位，则数据的表示范围为$|x|<2^n$（整数补码表示的最小负数除外）。当定点数的运算结果超出了定点数所能表示的范围时，称运算结果发生溢出。浮点数的溢出及判断将在 2.8 节讨论。由于补码减法也是转换成补码加法进行运算，因此，下面我们以 5 位定点整数补码（其中含 1 位符号位）的加法为例来说明在定点加减运算中出现溢出时的特点。

［**例 2.23**］ 两个 5 位二进制整数补码相加，可能会出现的六种不同情况举例。

①9+3=12	②9+8=−15	③(−5)+(−7)=−12
01001	01001	11011
+ 00011	+ 01000	+ 11001
01100	10001	10100
④(−9)+(−12)=11	**⑤9+(−5)=4**	**⑥(−12)+4=−8**
10111	01001	10100
+ 10100	+ 11011	+ 00100
01011	00100	11000

在②运算中，两个正数相加的结果成为负数，在④运算中，两个负数相加的结果成为正数，这两种运算的结果都是错误的。之所以发生错误，是因为运算结果产生了溢出。溢出分为正溢和负溢两种。两个正数相加，若运算结果大于机器所能表示的最大正数，称为正溢。两个负数相加，若运算结果小于机器所能表示的最小负数，称为负溢。

5 位定点整数补码（其中含 1 位符号位）的表示范围为-16～+15，很明显，②④运算的正确结果应分别为 17 和-21，超出了 5 位定点整数补码的表示范围，因此定点整数补码加法时会产生溢出。我们将会产生溢出的②④运算与其它几种不会产生溢出的运算进行对比，得出在定点整数补码加法运算时若发生溢出，将同时存在以下两个相同的特点：

(1)同符号数相加，结果的符号位与被加数和加数的符号位相异；

(2)符号位向前产生的进位值与尾数最高位向前产生的进位值相异。

在定点小数补码加法运算时若发生溢出，存在的特点与定点整数补码加法运算完全相同。

对于定点数补码的加减运算，判断溢出的方法有下面三种：

(1)采用单符号位法。根据前面介绍的溢出的特点(1)可知，只有在同符号数相加，结果的符号与被加数和加数的符号位相异时才会产生溢出，其它情况下的补码加法均不会产生溢出。例如，两个正数相加的结果仍然为正数，两个负数相加的结果仍然为负数，一个正数与一个负数相加结果可能为正数也可能为负数，这些都不会超出定点数的表示范围。

设$[x]_{补}=x_0x_1x_2\cdots x_n$，$[y]_{补}=y_0y_1y_2\cdots y_n$，$[z]_{补}=[x]_{补}+[y]_{补}$，$[z]_{补}=z_0z_1z_2\cdots z_n$，则判断溢出的逻辑表达式为：

$$V = x_0y_0\overline{z_0} + \overline{x_0}\,\overline{y_0}z_0 \qquad (2.17)$$

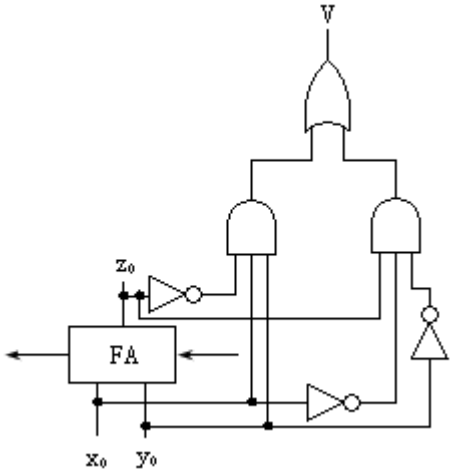


图 2.5　溢出判断方法之一

若 V=0，表示结果无溢出；若 V=1，表示结果有溢出。实现这种判断的逻辑电路如图 2.5 所示。

(2)采用进位判断法。根据前面介绍的溢出的特点(2)可知，当符号位向前产生的进位值与尾数最高位向前产生的进位值相异时才会产生溢出。设两个单符号位补码进行加减运算时，符号位向前产生的进位为 C，尾数的最高位向前产生的进位为 S，则判断溢出的逻辑表达式为：

$$V=C\oplus S \tag{2.18}$$

若 V=0，表示结果无溢出；若 V=1，表示结果有溢出。实现这种判断的逻辑电路如图 2.6 所示。

(3)采用双符号位法，这种方法又称为“变形补码”或“模 4 补码”。

对定点小数，设$[x]_{补}=x_0.x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则变形补码表示的定义为：

$$[x]_{补}=\begin{cases}x & 0\leqslant x\leqslant 1-2^{-n}\\ & (\text{mod } 4)\\ 4+x & -1\leqslant x\leqslant 0\end{cases} \tag{2.19}$$

对定点整数，设$[x]_{补}=x_0x_1x_2\cdots x_n$，共 n+1 位，其中 x_0为符号位，则变形补码表示的定义为：

$$[x]_{补}=\begin{cases}x & 0\leqslant x\leqslant 2^n-1\\ & (\text{mod } 2^{n+2})\\ 2^{n+2}+x=2^{n+2}-|x| & -2^n\leqslant x\leqslant 0\end{cases} \tag{2.20}$$

变形补码实际上是在一个数的补码表示时，用两个相同的符号位表示一个数的符号。用变形补码进行定点数补码加减运算的方法与用单符号位进行定点数补码加减运算的方法相同，即：

定点小数补码加法的运算公式为：

$$[x]_{补}+[y]_{补}=[x+y]_{补} \quad (\text{mod } 4) \tag{2.21}$$

定点整数补码加法的运算公式为：

$$[x]_{补}+[y]_{补}=[x+y]_{补} \quad (\text{mod } 2^{n+2}) \tag{2.22}$$

定点小数补码减法的运算公式为：

$$[x]_{补}-[y]_{补}=[x-y]_{补}=[x]_{补}+[-y]_{补} \quad (\text{mod } 4) \tag{2.23}$$

定点整数补码减法的运算公式为：

$$[x]_{补}-[y]_{补}=[x-y]_{补}=[x]_{补}+[-y]_{补} \quad (\text{mod } 2^{n+2}) \tag{2.24}$$

在实际运算时，须将数的两个符号位都看作是数的一部分参与运算，运算结果的最高符号位相加向前产生的进位要丢掉。若运算结果的双符号位相同，即为 00 或 11 时，表示运算结果未发生溢出；若运算结果的双符号位不相同，即为 01 或 10 时，表示运算结果发生溢出，第一符号位为结果的真正符号位。当运算结果的双符号位为 01 时，表示运算结果发生正溢；当运算结果的双符号位为 10 时，表示运算结果发生负溢。

设采用变形补码进行加减运算时，结果的双符号位分别为 z_0' 和 z_0，则判断溢出的逻辑表达式为：

$$V=z_0'\oplus z_0 \tag{2.25}$$

若 V=0，表示运算结果未发生溢出；若 V=1，表示运算结果发生溢出。实现这种判断的逻辑电路如图 2.7 所示。

[**例 2.24**] 已知 x=0.1100，y=0.1001，用变形补码计算 x+y，同时指出运算结果是否发生溢出。

解：$[x]_{补}$=00.1100 $[y]_{补}$=00.1001

$$\begin{array}{rll}[x]_{补} & 00.1100 & \\ +\ [y]_{补} & 00.1001 & \\ \hline [x+y]_{补} & 01.0101 & \text{运算结果发生正溢}\end{array}$$

[**例 2.25**] 已知 x=-1100，y=+1000，用变形补码计算 x-y，同时指出运算结果是否发

生溢出。

解： $[x]_{补}=110100$　$[-y]_{补}=111000$

$$\begin{array}{rl} & [x]_{补} \quad 110100 \\ + & [-y]_{补} \quad 111000 \\ \hline & [x-y]_{补} \quad 101100 \end{array}$$

运算结果发生负溢

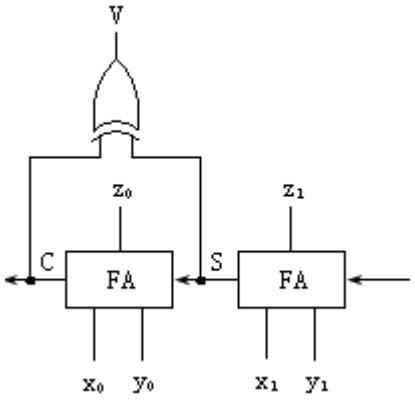


图 2.6　溢出判断方法之二

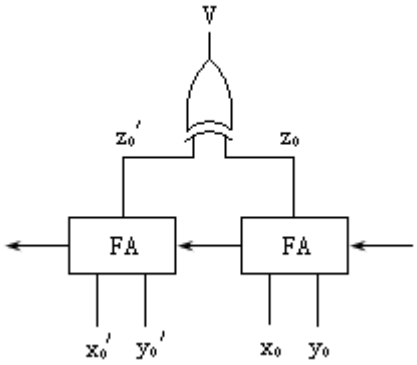


图 2.7　溢出判断方法之三

［**例 2.26**］ 已知 x=-1101，y=+0110，用变形补码计算 x+y，同时指出运算结果是否发生溢出。

解： $[x]_{补}=110011$　$[y]_{补}=000110$

$$\begin{array}{rl} & [x]_{补} \quad 110011 \\ + & [y]_{补} \quad 000110 \\ \hline & [x+y]_{补} \quad 111001 \end{array}$$

运算结果未发生溢出

所以　　x+y=-0111

4. 基本的二进制加法/减法器

表 2.2 给出了一位全加器的真值表，表中 A_i、B_i 和 C_i 为输入，S_i 和 C_{i+1} 为输出。其中，A_i 和 B_i 为两个相加的一位数，C_i 为低位向本位的进位，S_i 为相加产生的和，C_{i+1} 为本位相加时向前产生的进位。由真值表可以写出输出端 S_i 和 C_{i+1} 的逻辑表达式：

$$S_i=A_i \oplus B_i \oplus C_i$$

$$C_{i+1}=A_iB_i+B_iC_i+A_iC_i=(A_i \oplus B_i)C_i+A_iB_i \qquad (2.26)$$

表 2.2　一位全加器的真值表

输入			输出	
A_i	B_i	C_i	S_i	C_{i+1}
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

实现式(2.26)的一位全加器的逻辑电路图如图 2.8(a)所示，图 2.8(b)是一位全加器的符号表示。

一位全加器输出端 S_i 和 C_{i+1} 的逻辑表达式还可以写成与或非的形式：

$$S_i=\overline{\overline{A_i}\overline{B_i}\overline{C_i}+A_iB_i\overline{C_i}+A_i\overline{B_i}C_i+\overline{A_i}B_iC_i}$$

$$C_{i+1}=\overline{\overline{A_i}\overline{B_i}+\overline{A_i}\overline{C_i}+\overline{B_i}\overline{C_i}} \qquad (2.27)$$

实现式(2.27)的一位全加器的逻辑电路图如图 2.8(c)所示，由于两个表达式都可用与或非逻辑电路实现，因此在这种全加器中 S_i 和 C_{i+1} 的延迟时间相等，即 S_i 和 C_{i+1} 同时产生。

进行补码二进制加法/减法运算的逻辑结构图如图 2.9 所示，它是由 n 个一位的全加器

（FA）级联而成的一个 n 位的串行进位的补码加法/减法器。输入端$[A]_{补}=A_{n-1}A_{n-2}\cdots A_1A_0$，$[B]_{补}=B_{n-1}B_{n-2}\cdots B_1B_0$，其中 A_{n-1}为$[A]_{补}$的符号位，B_{n-1}为$[B]_{补}$的符号位，若为定点小数进行补码加/减运算，则定点小数的小数点在符号位之后。M 为方式控制输入端，若 M=0，$[B]_{补}$通过异或门时，输出保持不变，仍然是$[B]_{补}$，通过串行进位的 n 位全加器完成$[A]_{补}+[B]_{补}$运算，输出$[A+B]_{补}$；若 M=1，$[B]_{补}$通过异或门时，完成对$[B]_{补}$按位取反，并在加法时通过 C_0在末位加 1，得到$[-B]_{补}$，通过串行进位的 n 位全加器完成$[A]_{补}+[-B]_{补}$运算，输出$[A-B]_{补}$。

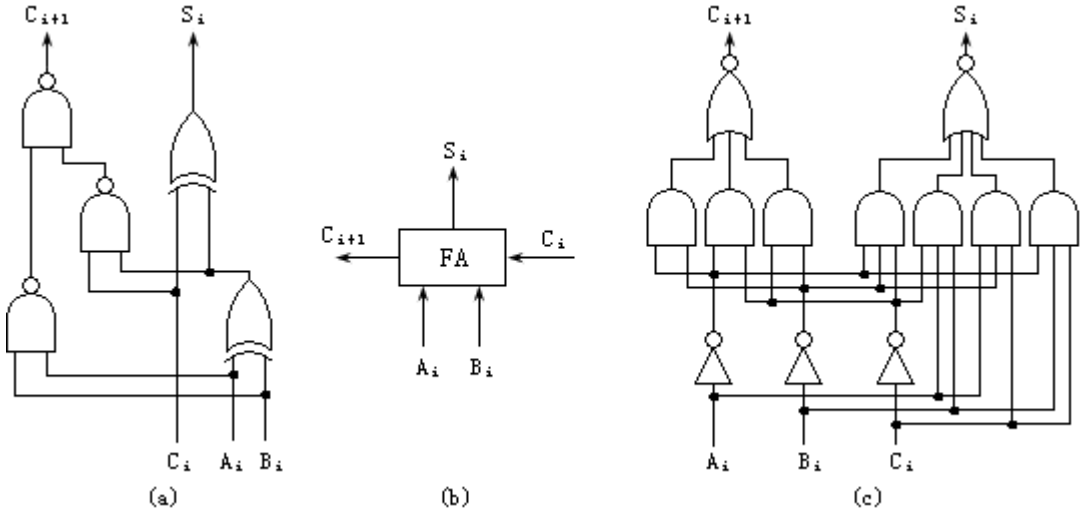


图 2.8　一位全加器的逻辑电路图及符号表示

图 2.9 采用了我们前面介绍的进位判断法来进行溢出检测，如图 2.9 左上角的异或门。当 $C_n\neq C_{n-1}$，即符号位运算产生的进位和数值的最高位运算产生的进位不相同时，异或门的输出为 1，表示运算结果发生溢出；当 $C_n=C_{n-1}$，即符号位运算产生的进位和数值的最高位运算产生的进位相同时，异或门的输出为 0，表示运算结果未发生溢出。

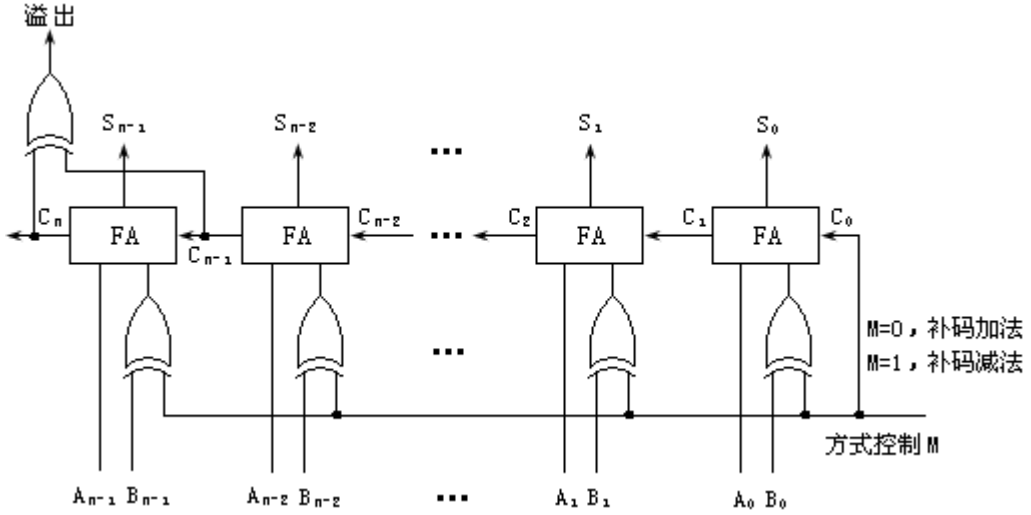


图 2.9　串行进位的补码加法/减法器

2.3.2　原码加法、减法

在进行原码加法、减法运算时，操作数与运算结果均用原码表示，运算时尾数进行加法或减法运算，符号位则单独处理，不能参加运算。原码加法、减法运算比较复杂，因为当两个操作数异号时，加法实际上变成了减法，减法实际上又变成了加法。在进行减法运算时，需先将减数求相反数，然后再进行加法运算。这样在原码运算时还需要使用补码，导致计算机系统将同时使用两种机器码进行运算。在计算机中，无论是进行加法运算，还是进行减法运算，最后都是转换成加法进行操作的。

设参加运算的两个定点小数的表示格式如图 2.10 所示，同时假设：

被加（减）数 x 的原码为：$[x]_{原}=x_0.x_1x_2\cdots x_{n-1}x_n$；

加（减）数 y 的原码为：$[y]_{原}=y_0.y_1y_2\cdots y_{n-1}y_n$；

和（差）z 的原码为：$[z]_{原}=z_0.z_1z_2\cdots z_{n-1}z_n$，$[z]_{原}$的尾数部分记为 $z_{尾}$；

机器操作后的结果为：$z'=z_0'.z_1'z_2'\cdots z_{n-1}'z_n'$，$z'$的尾数部分记为$z_{尾}'$。

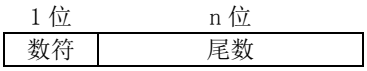


图 2.10　定点小数的表示格式

原码操作数进行加法或减法运算时，必须考虑符号位。原码加法、减法的运算规则如表 2.3 所示。

表 2.3　原码加法、减法运算规则

运算	x_0　y_0	机器操作	结果的尾数 $z_{尾}$	结果的符号位 z_0	溢出
加法	0　0	$\|x\|+\|y\|$	$z_{尾}=z_{尾}'$	$z_0=x_0=0$	$z_0'=1$，正溢出
	1　1			$z_0=x_0=1$	$z_0'=1$，负溢出
	0　1	$\|x\|+[-\|y\|]_{补}$	若 $z_0'=0$，则 $z_{尾}=z_{尾}'$； 若 $z_0'=1$，则 $z_{尾}$等于对 $z_{尾}'$进行求补运算的结果	若 $z_0'=0$，则 $z_0=x_0$	无溢出
	1　0			若 $z_0'=1$，则 $z_0=y_0$	
减法	0　0			若 $z_0'=0$，则 $z_0=x_0$	
	1　1			若 $z_0'=1$，则 $z_0=\overline{y_0}$	
	0　1	$\|x\|+\|y\|$	$z_{尾}=z_{尾}'$	$z_0=x_0=0$	$z_0'=1$，正溢出
	1　0			$z_0=x_0=1$	$z_0'=1$，负溢出

由表 2.3 可得出如下结论：

（1）两个数绝对值相加的条件

两个数同号且进行加法运算，或两个数异号且进行减法运算时，机器完成操作 $z'=|x|+|y|$；

和（差）的结果为：$[z]_{原}=x_0+z_{尾}'=x_0.z_1'z_2'\cdots z_{n-1}'z_n'$。

（2）两个数绝对值相减的条件

两个数异号且进行加法运算，或两个数同号且进行减法运算时，机器完成操作 $z'=|x|+[-|y|]_{补}$；

当 $z_0'=0$ 时，和（差）的结果为：$[z]_{原}=x_0+z_{尾}'=x_0.z_1'z_2'\cdots z_{n-1}'z_n'$

当 $z_0'=1$ 时，和的结果为：$[z]_{原}=y_0.\overline{z_1'}\,\overline{z_2'}\cdots\overline{z_{n-1}'}\,\overline{z_n'}+2^{-n}$

当 $z_0'=1$ 时，差的结果为：$[z]_{原}=\overline{y_0}.\overline{z_1'}\,\overline{z_2'}\cdots\overline{z_{n-1}'}\,\overline{z_n'}+2^{-n}$

（3）判断溢出

无论是进行加法运算还是进行减法运算，只有当两个数的绝对值相加（$|x|+|y|$）时才有可能产生溢出。而当两个数的绝对值相减（$|x|+[-|y|]_{补}$）时，相当于两个异符号数相加，永远不会产生溢出。因此，产生溢出的条件是：

正溢=($|x|+|y|$)・($z_0'=1$)・($x_0=0$)；

负溢=($|x|+|y|$)・($z_0'=1$)・($x_0=1$)。

（4）符号位

原码加法、减法运算，因符号位无需参加运算和参与溢出判断，故只需一位符号位。

从原码加法、减法运算规则可以看出，原码运算方法比补码运算方法要复杂得多，其对应的运算器也要复杂得多。目前的计算机系统中，应用最普遍的是补码表示，即以补码形式存储、传送和加工数据。

[**例 2.27**]　已知 x=-0.1101，y=-0.1011，用原码运算规则计算 x-y，同时指出运算结果是否发生溢出。

解：$[x]_{原}=1.1101$，$[y]_{原}=1.1011$

因为两个数同号且进行减法运算，所以机器完成$|x|+[-|y|]_{补}$操作。

$|x|=0.1101$，$|y|=0.1011$

$$\begin{array}{rll} & |x| & 0.1101 \\ + & [-|y|]_{补} & 1.0101 \end{array}$$

$|x|+[-|y|]_{补}$　0.0010

因为完成$|x|+[-|y|]_{补}$操作，所以运算结果不会发生溢出。

因为$|x|+[-|y|]_{补}$操作结果的符号位为0，所以差的符号与被减数同号。

所以　$[x+y]_{原}=1.0010$

$x+y=-0.0010$

[**例 2.28**]　已知 x=0.1001，y=0.1101，用原码运算规则计算 x+y，同时指出运算结果是否发生溢出。

解：$[x]_{原}=0.1001$，$[y]_{原}=0.1101$

因为两个数同号且进行加法运算，所以机器完成$|x|+|y|$操作。

$|x|=0.1001$，$|y|=0.1101$

$$\begin{array}{rl} & |x| \quad 0.1001 \\ + & |y| \quad 0.1101 \\ \hline & |x|+|y| \quad 1.0110 \end{array}$$

因为完成$|x|+|y|$操作且操作结果的符号位为 1，被加数为正数，所以运算结果发生正溢。

定点整数原码加法运算与定点小数原码加法运算的区别仅在于小数点的位置不同而已，运算规则完全相同。

[**例 2.29**]　已知 x=1011，y=-1101，用原码运算规则计算 x+y，同时指出运算结果是否发生溢出。

解：$[x]_{原}=01011$，$[y]_{原}=11101$

因为两个数异号且进行加法运算，所以机器完成$|x|+[-|y|]_{补}$操作。

$|x|=01011$，$|y|=01101$

$$\begin{array}{rl} & |x| \quad 01011 \\ + & [-|y|]_{补} \quad 10011 \\ \hline & |x|+[-|y|]_{补} \quad 11110 \end{array}$$

因为完成$|x|+[-|y|]_{补}$操作，所以运算结果不会发生溢出。

因为$|x|+[-|y|]_{补}$操作结果的符号位为 1，所以和的符号与加数同号，和的尾数为操作结果的尾数按位取反并在末位加1。

所以　$[x+y]_{原}=10010$

$x+y=-0010$

2.3.3　移码加法、减法

两个 n+1 位定点整数（包含 1 位符号位）的移码进行加法或减法运算有以下规则：操作数用移码表示，结果也用移码表示，两个数的符号位一起参与运算。由于移码是在补码的基础上将符号位取反得到的，因此 n+1 位定点整数 x（包含 1 位符号位）的移码为：

$$[x]_{移}=[x]_{补}+2^{n} \quad (\mathrm{mod}\ 2^{n+1}) \tag{2.28}$$

因此，移码加（减）运算可以依据如下公式：

$$\begin{aligned} [x+y]_{移}&=[x+y]_{补}+2^{n} \quad (\mathrm{mod}\ 2^{n+1}) \\ &=[x]_{补}+[y]_{补}+2^{n} \quad (\mathrm{mod}\ 2^{n+1}) \\ &=[x]_{移}+[y]_{补} \quad (\mathrm{mod}\ 2^{n+1}) \end{aligned} \tag{2.29}$$

$$\begin{aligned} [x-y]_{移}&=[x-y]_{补}+2^{n} \quad (\mathrm{mod}\ 2^{n+1}) \\ &=[x]_{补}+[-y]_{补}+2^{n} \quad (\mathrm{mod}\ 2^{n+1}) \\ &=[x]_{移}+[-y]_{补} \quad (\mathrm{mod}\ 2^{n+1}) \end{aligned} \tag{2.30}$$

为便于判断溢出，移码采用两位符号位，即采用变形移码，数据的第 1 位符号位（最高位）恒为 0，第 2 位符号位代表数据的正负。即当 x 为正数时，$[x]_{移}$的两个符号位为 01，而当 x 为负数时，$[x]_{移}$的两个符号位为 00。这样，移码加法、减法的运算公式可写作：

$$[x+y]_{移}=[x]_{移}+[y]_{补} \quad (\mathrm{mod}\ 2^{n+2}) \tag{2.31}$$

$$[x-y]_{移}=[x]_{移}+[-y]_{补} \quad (\mathrm{mod}\ 2^{n+2}) \tag{2.32}$$

式(2.31)和式(2.32)中使用的变形移码只是在运算过程中采用，在正常情况下，只有第

2 位符号位才代表数据的正负，故在传送和存储时仍只保留一位符号位。

采用变形移码进行运算，判断溢出的条件是：①当两位符号位的第 1 位符号位为 0 时，表示运算结果未发生溢出。此时，若两位符号位为 00 时，表示结果为负；若两位符号位为 01 时，表示结果为正。②当两位符号位的第 1 位符号位为 1 时，表示运算结果发生溢出。此时，若两位符号位为 10 时，表示正溢，若两位符号位为 11 时，表示负溢。

［**例 2.30**］ 已知 x=1011，y=-1110，用移码运算方法计算 x+y，同时指出运算结果是否发生溢出。

解：$[x]_{移}$=011011，$[y]_{补}$=110010

$$\begin{array}{rll} & [x]_{移} & 011011 \\ + & [y]_{补} & 110010 \\ \hline & [x+y]_{移} & 001101 \end{array}$$

运算结果未发生溢出

所以　　x+y=-0011

［**例 2.31**］ 已知 x=1001，y=-1100，用移码运算方法计算 x-y，同时指出运算结果是否发生溢出。

解：$[x]_{移}$=011001，$[-y]_{补}$=001100

$$\begin{array}{rll} & [x]_{移} & 011001 \\ + & [-y]_{补} & 001100 \\ \hline & [x-y]_{移} & 100101 \end{array}$$

运算结果发生正溢

2.3.4　十进制加法器

利用基本的二进制加法/减法器可完成二进制的加法或减法运算，如果计算机采用十进制数据表示，数据又如何进行运算呢？

处理十进制数通常有两种方法，一种方法是先将十进制数转换为二进制数，然后按二进制运算，最后再将结果转换为十进制数；另一种方法是采用 BCD 码，直接进行十进制运算，即将每组 4 位先当成二进制数运算，再按十进制运算的进位规律进行修正。第一种方法适用于数据量不多而计算量较大的情况，第二种方法适用于数据量较多但计算较简单的场合。

4 位 8421 码和余 3 码是最常用的 BCD 码，下面以 8421 BCD 码为例介绍其加法器的构成及工作原理。

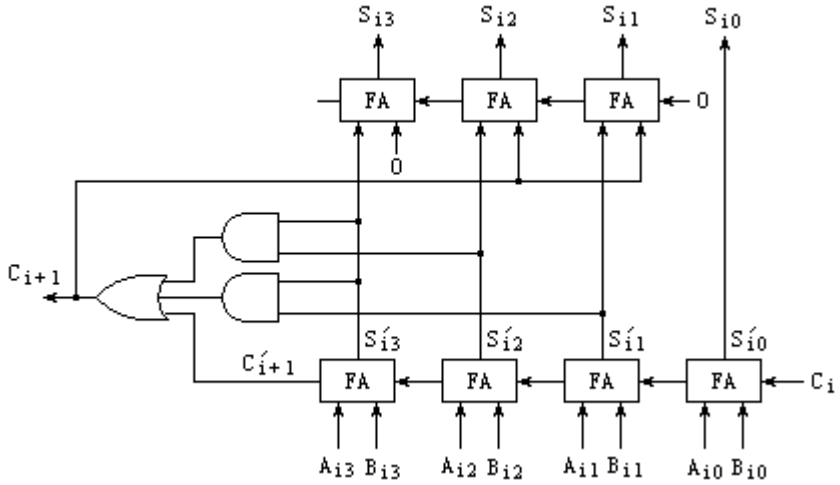


图 2.11　一位 8421 码十进制加法器

两个一位十进制数相加，其和大于 9 时，便产生进位。因此，用 8421 码表示的两个一位十进制数相加，当和大于 9 时，必须对和数加上 6 进行修正，才能产生进位。这是因为，采用 8421 码后，在两个一位十进制数相加的和数小于等于 9 时，十进制运算的结果是正确的，而当相加的和数大于 9 时，结果不正确，必须加上 6 修正后才能得出正确的结果。因此，在一位 8421 码十进制加法器设计时，可先将两个 4 位二进制数 $A_{i3}A_{i2}A_{i1}A_{i0}$ 、 $B_{i3}B_{i2}B_{i1}B_{i0}$ 和低位产生的进位 C_i 采用串行进位的二进制加法器来求和，再对求得的和 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 进行修正。设两个一位 8421 码采用串行进位的二进制加法器运算产生的进位为 C'_{i+1}，而 $S_{i3}S_{i2}S_{i1}S_{i0}$ 代表

正确的 BCD 和数，C_{i+1} 代表正确的进位。

当和数 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 的取值出现 1010、1011、1100、1101、1110、1111 或 C'_{i+1}=1 等情况之一时，对和数 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 加上 6 进行修正才能得到正确的 BCD 和数，当和数 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 的取值小于等于 9 时，不需要对和数 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 进行修正。显然，当 C'_{i+1}=1 或 $S'_{i3}S'_{i2}S'_{i1}S'_{i0}$ 大于 9 时，C_{i+1}=1，即 $C_{i+1}=S'_{i3}S'_{i1}+S'_{i3}S'_{i2}+C'_{i+1}$。因此，可利用 C_{i+1} 的状态来产生所要求的校正因子，当 C_{i+1}=1 时校正因子为 6，当 C_{i+1}=0 时，校正因子为 0。由此，可以设计出一位 8421 码十进制加法器如图 2.11 所示。

n 位 8421 码十进制串行进位加法器的一般结构如图 2.12 所示，它由 n 级组成，每一级是一个一位 8421 码十进制加法器，每级之间通过一根进位线与其相邻级连接。

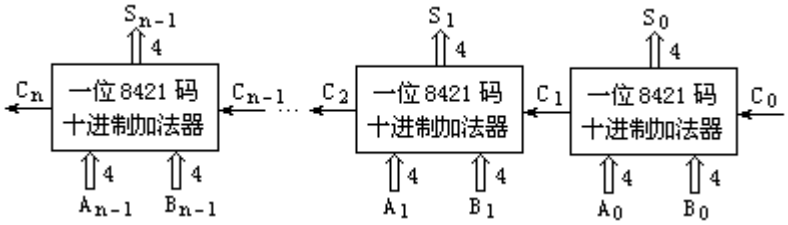


图 2.12　n 位 8421 码十进制串行进位加法器

2.4　定点乘法运算

乘、除法运算是计算机的基本运算之一。实现乘、除法运算的方法较多，归纳起来为两种。第一种是采用软件的方法，在早期的低档计算机中采用软件的方法，即利用计算机中的基本指令编写子程序，当需要做乘、除法运算时，通过调用子程序来实现；第二种是采用硬件的方法，在功能较强的计算机中，用加法器和移位器来实现乘、除法运算，在速度高、功能强的计算机中，则通过设置专门的阵列乘除部件来实现乘、除法运算。

2.4.1　原码一位乘法

由于原码的数值部分与真值相同，所以原码的乘法与采用真值运算时的手工算法非常相似。首先我们来看二进制乘法的人工运算过程。

设 x=0.1101，y=−0.1001，那么 x×y 的运算如下：

$$
\begin{array}{r}
0.1101 \\
\times\ \ 0.1001 \\
\hline
1101 \\
0000 \\
0000 \\
+\quad 1101 \\
\hline
0.01110101
\end{array}
$$

由于 x 为正数，y 为负数，所以

$$x \times y = -0.01110101$$

二进制乘法运算过程与十进制乘法相似，先用 x 和 y 的数值部分相乘，然后用 x 和 y 小数位的位数来确定乘积中小数位的位数，最后根据 x 和 y 的符号来确定乘积的符号。在数值部分相乘时，都是从 y 的最低位开始，逐位与被乘数相乘。若这一位为 1，则乘得的结果为被乘数，若这一位为 0，则乘得的结果为全 0。然后再对 y 的高一位进行乘法运算，其规则同上，由于这一位乘数的权值与低位乘数的权值不一样，因此这一位乘得的结果与低位乘得的结果相比，要左移一位。依此类推，直到乘数各位乘完为止，最后相加便得到乘积。若为整数相乘，则只需考虑数值部分相乘和结果的符号即可，方法同上。

在计算机进行乘法运算时，由于只有一次实现两个数相加的加法器，无法同时实现 n 位积的并行相加运算。为此，必须修改上述乘法运算规则，使之能在计算机上实现。

设被乘数$[x]_{原}=x_f.x_1x_2\cdots x_{n-1}x_n$

乘数$[y]_{原}=y_f.y_1y_2\cdots y_{n-1}y_n$

则乘积 $[z]_{原}=(x_f\oplus y_f)+(0.x_1x_2\cdots x_{n-1}x_n)\times(0.y_1y_2\cdots y_{n-1}y_n)$ (2.33)

或 $[z]_{原}=(x_f\oplus y_f).((x_1x_2\cdots x_{n-1}x_n)\times(y_1y_2\cdots y_{n-1}y_n))$ (2.34)

式(2.33)表明，乘积的符号位为被乘数和乘数的符号位相异或，乘积的绝对值为被乘数的绝对值与乘数的绝对值相乘，它是原码一位乘法算法的设计基础；式(2.34)表明，乘积的符号位为被乘数和乘数的符号位相异或，乘积的数值部分为被乘数的数值部分与乘数的数值部分相乘，它是原码阵列乘法算法的设计基础。

下面我们来看原码一位乘法中，被乘数的绝对值与乘数的绝对值相乘的算法推导过程。为简单起见，令 $x'=|x|=0.x_1x_2\cdots x_{n-1}x_n$，$y'=|y|=0.y_1y_2\cdots y_{n-1}y_n$

同时令乘积 $z'=|z|=x'\times y'$，有：

$$\begin{aligned}x'\times y'&=x'\times(0.y_1y_2\cdots y_{n-1}y_n)\\&=x'\times(y_12^{-1}+y_22^{-2}+\cdots+y_{n-1}2^{-(n-1)}+y_n2^{-n})\\&=2^{-1}(y_1x'+2^{-1}(y_2x'+\cdots2^{-1}(y_{n-1}x'+2^{-1}(y_nx'+0))\cdots))\end{aligned}\quad(2.35)$$

令 z_i 表示 z' 第 i 次的部分积，则式(2.35)可写成如下递推公式：

$$\begin{aligned}z_0&=0\\z_1&=2^{-1}(y_nx'+z_0)\\z_2&=2^{-1}(y_{n-1}x'+z_1)\\&\cdots\\z_i&=2^{-1}(y_{n-i+1}x'+z_{i-1})\\&\cdots\\z_{n-1}&=2^{-1}(y_2x'+z_{n-2})\\z_n&=2^{-1}(y_1x'+z_{n-1})\end{aligned}\quad(2.36)$$

显然，欲求 $x'\times y'$，则需设置一个保存部分积的累加器。乘法开始时，令部分积的初值为 $z_0=0$，然后先加上 y_nx'，右移 1 位得第 1 个部分积 z_1，又加上 $y_{n-1}x'$，再右移 1 位得第 2 个部分积 z_2。依此类推，直到求得 y_1x'加上 z_{n-1}并右移 1 位得最后部分积 z_n，即得 $x'\times y'$。显然，两个 n 位小数相乘需要重复进行 n 次“加”和“右移”操作，才能得到最后乘积。这就是实现原码一位乘法的算法。

[**例 2.32**] 已知 x=-0.1101，y=0.0101，用原码一位乘法计算 x×y。

解： $[x]_{原}=1.1101$ $[y]_{原}=0.0101$

乘积的符号位为：$x_f\oplus y_f=1\oplus0=1$

令 $x'=|x|=0.1101$，$y'=|y|=0.0101$

$[x']_{补}=0.1101$ $[y']_{补}=0.0101$

	部分积	乘数	说明
	00.0000	**0**.0101	部分积 $z_0=0$
$+[x']_{补}$	00.1101		$y_4=1$，$+[x']_{补}$
	00.1101		
→	00.0110	1**0**.010	右移 1 位，得 z_1
$+[0]_{补}$	00.0000		$y_3=0$，$+[0]_{补}$
	00.0110		
→	00.0011	01**0**.01	右移 1 位，得 z_2
$+[x']_{补}$	00.1101		$y_2=1$，$+[x']_{补}$
	01.0000		
→	00.1000	001**0**.0	右移 1 位，得 z_3
$+[0]_{补}$	00.0000		$y_1=0$，$+[0]_{补}$
	00.1000		
→	00.0100	0001**0**.	右移 1 位，得 $z_4=\|x\|\times\|y\|$

$[x\times y]_{原}=1.01000001$

所以 $x\times y=-0.01000001$

由例 2.32 可以看出，乘数的最高位不参与运算。乘积的原码依次由乘积的符号位、部分积（寄存器）中的低 4 位、乘数（寄存器）中的高 4 位组成。定点整数原码一位乘法与定点小数原码一位乘法的区别仅在于被乘数和乘数中没有小数点，乘积输出时乘积原码的符号位之后没有小数点。

2.4.2 补码一位乘法

原码一位乘法的主要问题在于符号位不能直接参与运算，而是单独用一个异或门产生乘积的符号位。为了能像补码加减运算一样，将数的符号位连同数一起参与运算，可采用补码一位乘法。

为了分析补码一位乘法的机器算法和逻辑实现，首先讨论补码与真值的转换公式。

1. 补码与真值的转换公式

设$[x]_{补}=x_0.x_1x_2\cdots x_{n-1}x_n$

当 $x\geqslant 0$ 时，$x_0=0$，根据定义$[x]_{补}=x$，得

$$0.x_1x_2\cdots x_{n-1}x_n=x$$

即 $$x=-0+0.x_1x_2\cdots x_{n-1}x_n$$

当 $x<0$ 时，$x_0=1$，根据定义$[x]_{补}=2+x$，得

$$1.x_1x_2\cdots x_{n-1}x_n=2+x$$

即 $$x=-1+0.x_1x_2\cdots x_{n-1}x_n$$

由此可以得出，补码与真值之间的转换公式为：

$$x=-x_0+0.x_1x_2\cdots x_{n-1}x_n$$

即
$$x=-x_0+\sum_{i=1}^{n}(x_i\times 2^{-i}) \tag{2.37}$$

同样的道理，对定点整数，若$[x]_{补}=x_nx_{n-1}x_{n-2}\cdots x_1x_0$，则

$$x=-x_n\times 2^n+x_{n-1}x_{n-2}\cdots x_1x_0$$

即
$$x=-x_n\times 2^n+\sum_{i=0}^{n-1}(x_i\times 2^i) \tag{2.38}$$

由式(2.37)和式(2.38)可以得出这样的结论，若将一个数的补码表示中符号位变为负权，便是该数的真值。例如，若$[x]_{补}=10101$，则真值 $x=(1)0101=1\times(-2^4)+0101=-1011$，其中(1)表示符号位 1 为负权。

补码与真值之间的转换，除了采用式(2.37)和式(2.38)这两个公式之外，还有另外一种方法，我们已在 2.1.2 节介绍过，这里不再重复。

[**例 2.33**] 已知$[x_1]_{补}=1.1101$，$[x_2]_{补}=0.0101$，$[y_1]_{补}=10111$，$[y_2]_{补}=01011$，求 x_1、x_2、y_1、y_2。

解： $x_1=-1+0.1101=-0.0011$

$x_2=-0+0.0101=0.0101$

$y_1=1\times(-2^4)+0111=-10000+0111=-1001$

$y_2=0\times(-2^4)+1011=1011$

2. 补码的移位

对于一个数的补码，无论其是正数还是负数，每左移一位，低位补 0，相当于这个数乘以 2，若在左移的过程中符号发生改变，则表示运算结果发生溢出。

对于一个数的补码，无论其是正数还是负数，每右移一位，符号位保持不变，相当于这个数除以 2（即乘以 1/2）。

[**例 2.34**] 已知$[x]_{补}=1.1101001$，$[y]_{补}=00101011$，若字长固定为 8 位，采用 0 舍 1 入法，求$[2x]_{补}$、$\left[\frac{1}{2}x\right]_{补}$、$[2y]_{补}$、$\left[\frac{1}{2}y\right]_{补}$。

解： $[2x]_{补}=1.1010010$

$\left[\frac{1}{2}x\right]_{补}=1.1110101$

$[2y]_{补}$=01010110

$\left[\frac{1}{2}y\right]_{补}$=00010110

3. 补码一位乘法的机器算法

设被乘数$[x]_{补}=x_0.x_1x_2\cdots x_{n-1}x_n$，乘数$[y]_{补}=y_0.y_1y_2\cdots y_{n-1}y_n$，根据式(2.37)，有

$$[x\times y]_{补}=[x]_{补}\times y=[x]_{补}\times\left(-y_0+\sum_{i=1}^{n}(y_i\times 2^{-i})\right)\quad(\mathrm{mod}\ 2)\qquad(2.39)$$

式(2.39)中有关$[x\times y]_{补}=[x]_{补}\times y$的证明可分$y\geqslant 0$和$y<0$两种情况并结合补码的定义来证明，这里不再展开论述。将式(2.39)展开，有：

$$\begin{aligned}[x\times y]_{补}&=[x]_{补}\times(-y_0+y_12^{-1}+y_22^{-2}+\cdots+y_{n-1}2^{-(n-1)}+y_n2^{-n})\\&=[x]_{补}\times[-y_0+(y_1-y_12^{-1})+(y_2-y_22^{-2})+\cdots+(y_{n-1}-y_{n-1}2^{-(n-1)})+(y_n-y_n2^{-n})]\\&=[x]_{补}\times[(y_1-y_0)+(y_2-y_1)2^{-1}+(y_3-y_2)2^{-2}+\cdots+(y_n-y_{n-1})2^{-(n-1)}+(0-y_n)2^{-n}]\\&=\sum_{i=0}^{n}((y_{i+1}-y_i)\times[x]_{补}\times 2^{-i})\qquad(\text{其中 }y_{n+1}=0)\qquad(2.40)\end{aligned}$$

设$[z]_{补}=[x\times y]_{补}$，$[z_i]_{补}$表示$[z]_{补}$第 i 次的部分积，则式(2.40)可写成如下递推公式：

$$\begin{aligned}&[z_0]_{补}=0\\&[z_1]_{补}=2^{-1}([z_0]_{补}+(y_{n+1}-y_n)[x]_{补})\qquad(\text{其中 }y_{n+1}=0)\\&[z_2]_{补}=2^{-1}([z_1]_{补}+(y_n-y_{n-1})[x]_{补})\\&\qquad\cdots\\&[z_i]_{补}=2^{-1}([z_{i-1}]_{补}+(y_{n-i+2}-y_{n-i+1})[x]_{补})\\&\qquad\cdots\\&[z_n]_{补}=2^{-1}([z_{n-1}]_{补}+(y_2-y_1)[x]_{补})\\&[z_{n+1}]_{补}=[z_n]_{补}+(y_1-y_0)[x]_{补}\qquad(2.41)\end{aligned}$$

开始时，部分积为 0，即$[z_0]_{补}=0$，然后每一步都在前一次部分积的基础上，由$(y_{i+1}-y_i)$(i=0、1、2、…、n)决定对$[x]_{补}$的操作，再右移 1 位，得到新的部分积。如此重复 n+1 步，最后一步不移位，便得到$[x\times y]_{补}$。由于这种算法最早是由 Booth 夫妇提出的，所以又称为 Booth 算法。

实现这种补码一位乘法的机器算法时，在乘数的最末位要增加 1 位附加位y_{n+1}，y_{n+1}的初值为 0。开始时，由y_ny_{n+1}的取值判断第一步该完成什么运算，执行完运算后，部分积要右移 1 位。此时y_n正好移到原来y_{n+1}的位置上，y_{n-1}正好移到原来y_n的位置上。然后再由当前y_ny_{n+1}的取值判断第二步该完成什么运算，依此类推。

若$y_ny_{n+1}=01$，即$y_{n+1}-y_n=1$，则用部分积加上被乘数$[x]_{补}$，部分积右移 1 位；若$y_ny_{n+1}=10$，即$y_{n+1}-y_n=-1$，则用部分积减去被乘数$[x]_{补}$，即加上$[-x]_{补}$，部分积右移 1 位；若$y_ny_{n+1}=00$或 11，即$y_{n+1}-y_n=0$，则用部分积加上$[0]_{补}$，部分积右移 1 位。

[**例 2.35**] 已知 x=-0.1101，y=0.0101，用补码一位乘法计算 x×y。

解：$[x]_{补}=1.0011$　$[y]_{补}=0.0101$

$[-x]_{补}=0.1101$

	部分积	乘数	说明
	00.0000	**0**.01010	部分积$[z_0]_{补}=0$，附加位$y_{n+1}=0$
$+[-x]_{补}$	00.1101	y_{n+1}	$y_ny_{n+1}=10$，$+[-x]_{补}$
	00.1101		
→	00.0110	1**0**.0101	右移 1 位，得$[z_1]_{补}$
$+[x]_{补}$	11.0011		$y_ny_{n+1}=01$，$+[x]_{补}$
	11.1001		
→	11.1100	11**0**.010	右移 1 位，得$[z_2]_{补}$
$+[-x]_{补}$	00.1101		$y_ny_{n+1}=10$，$+[-x]_{补}$
	00.1001		
→	00.0100	111**0**.01	右移 1 位，得$[z_3]_{补}$

+ $[x]_{补}$	11.0011		$y_n y_{n+1}$=01，+$[x]_{补}$
	11.0111		
⟶	11.1011	1111**0**.0	右移 1 位，得$[z_4]_{补}$
+ $[0]_{补}$	00.0000		$y_n y_{n+1}$=00，+$[0]_{补}$
	11.1011	1111**0**.0	最后一步不移位，得$[z_5]_{补}=[x\times y]_{补}$

$[x\times y]_{补}$=1.10111111

所以　x×y=-0.01000001

由例 2.35 可以看出，被乘数和乘数的符号位一起参与运算。n 位小数与 n 位小数相乘，共要进行 n+1 次加法，右移 n 次，最后一次加法完成后，部分积不右移。定点整数补码一位乘法与定点小数补码一位乘法的区别仅在于被乘数和乘数中没有小数点，乘积输出时乘积补码的符号位之后没有小数点。

[**例 2.36**]　已知 x=-1101，y=-1111，分别用原码一位乘法和补码一位乘法计算 x×y。

解：①原码一位乘法

$[x]_{原}$=11101　$[y]_{原}$=11111

乘积的符号位为：$x_f \oplus y_f=1\oplus 1=0$

令 x'=|x|=1101，y'=|y|=1111

$[x']_{补}$=01101　$[y']_{补}$=01111

	部分积	乘数	说明
	000000	**0**1111	部分积 z_0=0
+$[x']_{补}$	001101		y_4=1，+$[x']_{补}$
	001101		
⟶	000110	1**0**111	右移 1 位，得 z_1
+$[x']_{补}$	001101		y_3=1，+$[x']_{补}$
	010011		
⟶	001001	11**0**11	右移 1 位，得 z_2
+$[x']_{补}$	001101		y_2=1，+$[x']_{补}$
	010110		
⟶	001011	011**0**1	右移 1 位，得 z_3
+$[x']_{补}$	001101		y_1=1，+$[x']_{补}$
	011000		
⟶	001100	0011**0**	右移 1 位，得 $z_4=\|x\|\times\|y\|$

$[x\times y]_{原}$=011000011

所以　x×y=+11000011

②补码一位乘法

$[x]_{补}$=10011　$[y]_{补}$=10001

$[-x]_{补}$=01101

	部分积	乘数	说明
	000000	**1**00010	部分积$[z_0]_{补}$=0，附加位 y_{n+1}=0
+$[-x]_{补}$	001101	↑ y_{n+1}	$y_n y_{n+1}$=10，+$[-x]_{补}$
	001101		
⟶	000110	1**1**0001	右移 1 位，得$[z_1]_{补}$
+ $[x]_{补}$	110011		$y_n y_{n+1}$=01，+$[x]_{补}$
	111001		
⟶	111100	11**1**000	右移 1 位，得$[z_2]_{补}$
+ $[0]_{补}$	000000		$y_n y_{n+1}$=00，+$[0]_{补}$
	111100		
⟶	111110	011**1**00	右移 1 位，得$[z_3]_{补}$
+ $[0]_{补}$	000000		$y_n y_{n+1}$=00，+$[0]_{补}$
	111110		

→	111111	001110	右移 1 位，得$[z_4]_{补}$
$+[-x]_{补}$	001101		y_ny_{n+1}=10，$+[-x]_{补}$
	001100	001110	最后一步不移位，得$[z_5]_{补}=[x\times y]_{补}$

$[x\times y]_{补}$=011000011

所以　x×y=+11000011

2.4.3　阵列乘法器

1. 不带符号的阵列乘法器

前面介绍的原码一位乘法和补码一位乘法都是通过串行移位和并行加法相结合的方法实现的，这种方法不需要很多器件，但串行操作的速度太慢。自从大规模集成电路问世以来，高速的阵列乘法器应运而生，它由大量的与门和全加器构成，其中与门构成与阵列，全加器构成乘法器阵列。乘法器阵列中的全加器采用并行流水的方式进行运算，大大提高了乘法运算的速度。

设有两个不带符号的 n 位二进制整数：

$A=a_{n-1}a_{n-2}\cdots a_1a_0$　　$B=b_{n-1}b_{n-2}\cdots b_1b_0$

A×B 的人工计算方法如下：

				a_{n-1}	a_{n-2}	…		a_1	a_0		
			×	b_{n-1}	b_{n-2}	…		b_1	b_0		
			$a_{n-1}b_0$	$a_{n-2}b_0$	…		a_1b_0	a_0b_0		第 1 行被加数
		$a_{n-1}b_1$	$a_{n-2}b_1$	…		a_1b_1	a_0b_1			第 2 行被加数
	$a_{n-1}b_2$	$a_{n-2}b_2$	…		a_1b_2	a_0b_2				第 3 行被加数
	…									…
+	$a_{n-1}b_{n-1}$	$a_{n-2}b_{n-1}$	…	a_1b_{n-1}	a_0b_{n-1}						第 n 行被加数
p_{2n-1}	p_{2n-2}	p_{2n-3}	…	p_{n-1}	p_{n-2}	…	p_1	p_0			

上述过程中的每一个乘积项（位积）a_ib_j叫做一个被加数，这 n^2个被加数（a_ib_j，0≤i，j≤n-1）可以用个 n^2与门并行产生，如图 2.13 的上半部所示，所有被加数形成所花的时间实际上就是一个与门所花的时间。为了提高乘法运算的速度，可将上述过程中第 1 行被加数与第 2 行被加数的对应位通过第 1 行 n-1 个一位的全加器并行相加（每个全加器的进位输入端为 0），每个全加器产生的进位并不直接传递给此次运算的高位全加器，而是将每个全加器的和、低位产生的进位与第 3 行被加数的对应位再通过第 2 行 n-1 个一位的全加器并行相加。同样，此次运算中每个全加器产生的进位也不直接传递给此次运算的高位全加器，而是将每个全加器的和、低位产生的进位与第 4 行被加数的对应位再通过第 3 行 n-1 个一位的全加器并行相加。依次类推，直到最后一行，即上述运算过程中的第 n 行被加数参与运算后为止。但由于第 n 行被加数运算后并行产生的进位并没有参与运算，因此，必须再加入第 n 行 n-1 个一位的全加器将第 n-1 行全加器产生的和、第 n-1 行全加器低位产生的进位、以及本行全加器低位产生的进位串行相加来产生最后的乘积。这一乘法运算过程由如图 2.13 下半部的乘法阵列来实现，n 位二进制整数与 n 位二进制整数相乘其结果为 2n 位，即 $p_{2n-1}p_{2n-2}p_{2n-3}\cdots p_1p_0$。

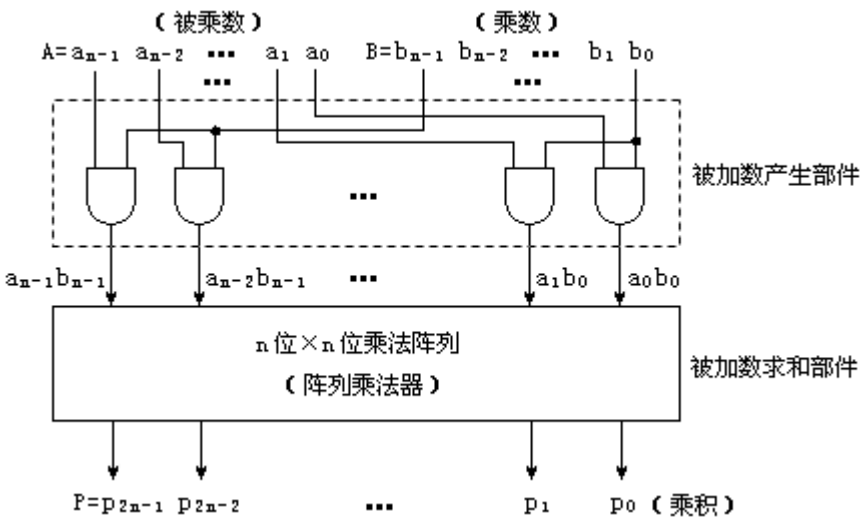


图 2.13　n 位×n 位不带符号的阵列乘法器逻辑框图

图 2.14 给出了按上述思想设计的 5 位×5 位不带符号的阵列乘法器逻辑原理图，图中 FA 的内部结构如图 2.8(c)所示，通过 FA 进行运算时，和与进位同时产生。图 2.14 中 FA 的上边和右边输入的是两个被加数，斜上方输入的是上一行低位运算产生的进位，FA 的下边和斜下方输出的分别是运算产生的和、以及此位运算产生的进位。每行为 4 个全加器，共 5 行，上面 4 行中每行的全加器并行相加，将和与进位并行传递给下一行被加数的相应位。图 2.14 中最后一行的全加器采用的是串行进位以形成最后的乘积。当然，为了缩短加法时间，最后一行的串行进位也可以采用先行进位加法器来代替。有关先行进位加法器的概念和原理，我们将在 2.7 节中介绍。

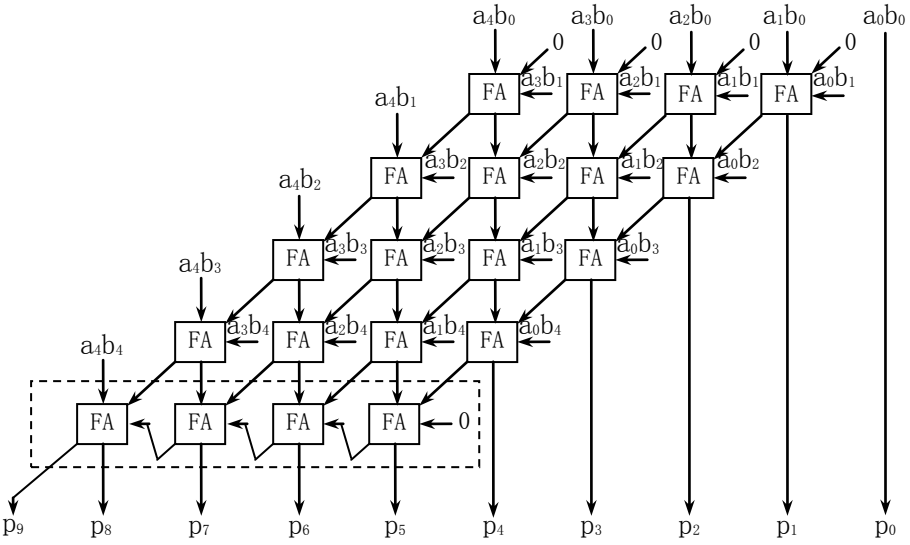


图 2.14　5 位×5 位不带符号的阵列乘法器逻辑原理图

2. 带求补器的阵列乘法器

在介绍带求补器的阵列乘法器基本原理之前，我们先来介绍在此阵列乘法器中用到的对 2 求补电路。4 位对 2 求补电路的电路图如图 2.15 所示，其逻辑表达式如下：

$C_0=0$，$C_{i+1}=a_i+C_i$

$a_i^* = a_i \oplus (E \cdot C_i)$　　$0 \leqslant i \leqslant 3$

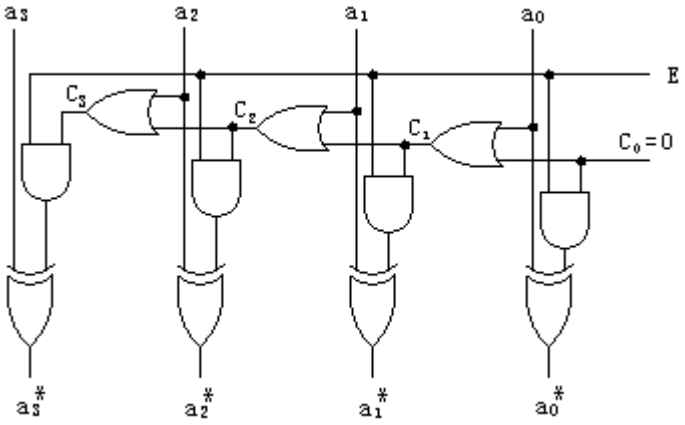


图 2.15　4 位对 2 求补电路图

图 2.15 中 E 为使能控制端，当 E=0 时，对 2 求补电路的输出 $a_3^*a_2^*a_1^*a_0^*$就等于其输入

$a_3a_2a_1a_0$；当 E=1 时，从输入数据 $a_3a_2a_1a_0$ 的最右端开始，由右向左，直到找到某一位 a_i=1（0≤i≤3），a_i 和 a_i 右边的输入对应的输出都保持输入值不变，而 a_i 左边的输入对应的输出则按对应位取反。也就是说，当 E=1 时，对输入端的数据按位取反并在末位加 1，即可得到输出数据。

对于一个定点整数的补码，设$[a]_{补}=a_4a_3a_2a_1a_0$，其中 a_4 为符号位，若 E=a_4，即用一个数补码的符号位作为对 2 求补电路的使能控制信号，则其输出为该补码表示数的真值的绝对值。例如，在图 2.15 中，若输入 $a_3a_2a_1a_0$=0110，E=0 则输出 0110，E=1 则输出 1010。

若在图 2.15 中 $a_3a_2a_1a_0$ 输入的是一个数的绝对值，E 为该数的符号位，我们可以发现，当 E=0 时，对 2 求补电路的输出 $a_3^*a_2^*a_1^*a_0^*$就等于其输入 $a_3a_2a_1a_0$，E$a_3^*a_2^*a_1^*a_0^*$为该数的补码；当 E=1 时，对 2 求补电路的输出 $a_3^*a_2^*a_1^*a_0^*$等于其输入 $a_3a_2a_1a_0$ 按位取反并在末位加 1，E$a_3^*a_2^*a_1^*a_0^*$也正好为该数的补码。这类似于我们在 2.1 节中介绍的，如何已知一个数的真值来求一个数的补码的方法，只是将正、负号换成了这里的符号位。

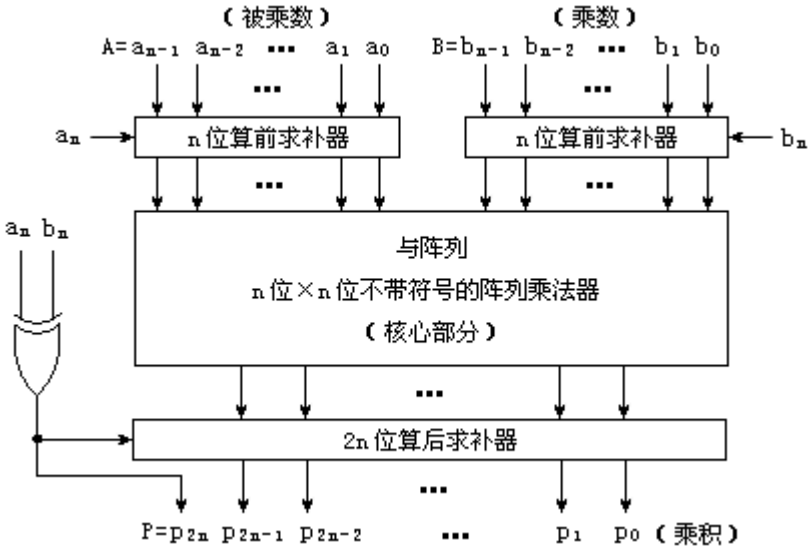


图 2.16　(n+1)位×(n+1)位带求补器的阵列乘法器逻辑框图

利用对 2 求补电路和不带符号的阵列乘法器构成了如图 2.16 所示的(n+1)位×(n+1)位带求补器的阵列乘法器。在定点整数补码相乘时，首先通过 n 位算前求补器求出被乘数和乘数的绝对值（即将补码的数值位转换成原码的数值位），再将两数的绝对值（原码的数值位）送入与阵列和 n 位×n 位不带符号的阵列乘法器，以求出乘积的绝对值（两数原码数值位的乘积），同时被乘数和乘数的符号位相异或产生乘积的符号位。在乘积符号位的控制下，对乘积的绝对值（两数原码数值位的乘积）通过 2n 位算后求补器进行算后求补，即根据一个数的符号位和绝对值（原码的数值位）来求该数的补码。

定点小数补码相乘与定点整数补码相乘的区别主要有两点：①小数点位于被乘数、乘数和乘积的符号位之后；②算前求补后的输出不是定点小数的绝对值，但它与定点整数补码相乘一样，都是被乘数和乘数原码的数值位。

不管是定点小数补码相乘，还是定点整数补码相乘，都应先通过 n 位的算前求补器将补码的数值位转换成原码的数值位，然后再送入与阵列和 n 位×n 位不带符号的阵列乘法器进行运算，最后通过 2n 位的算后求补器根据乘积的符号位和两数原码数值位的乘积来求乘积的补码。由此可以看出，n 位×n 位不带符号的阵列乘法器实际上是对被乘数和乘数原码的数值位进行乘法运算，尽管被乘数和乘数均用补码表示，但参与运算的数的范围必须与原码相同。由于输入输出数据均用补码表示，我们称之为带求补器的补码阵列乘法器。

对图 2.16，若 n 位算前求补器和 2n 位算后求补器的使能控制信号恒置为 0，即不受符号位控制，便可用带求补器的阵列乘法器完成原码并行乘法。此时，输入输出数据均用原码表示，我们称之为带求补器的原码阵列乘法器。

[**例 2.37**]　设 x=−1101，y=1111，分别用带求补器的原码阵列乘法器和带求补器的补码阵列乘法器计算 x×y。

解：①带求补器的原码阵列乘法器

$[x]_{原}$=11101　　$[y]_{原}$=01111

乘积的符号位为：$x_f \oplus y_f=1 \oplus 0=1$

因符号位单独考虑，算前求补器的使能控制信号为 0，经算前求补后输出

|x|=1101，|y|=1111

```
          1101
        × 1111
        ------
          1101
         1101
        1101
   +   1101
   ----------
   11000011
```

因算后求补器的使能控制信号为 0，经算后求补后输出为 11000011，加上乘积符号位 1，得

$[x \times y]_{原}$=111000011

所以　$x \times y=-11000011$

②带求补器的补码阵列乘法器

$[x]_{补}$=10011　　$[y]_{补}$=01111

乘积的符号位为：$x_f \oplus y_f=1 \oplus 0=1$

因算前求补器的使能控制信号分别为被乘数和乘数的符号位，经算前求补后输出

|x|=1101，|y|=1111

```
          1101
        × 1111
        ------
          1101
         1101
        1101
   +   1101
   ----------
   11000011
```

因算后求补器的使能控制信号为乘积的符号位，经算后求补后输出为 00111101，加上乘积符号位 1，得

$[x \times y]_{补}$=100111101

所以　$x \times y=-11000011$

3. 直接补码阵列乘法器

1）一般化的全加器形式

常规的全加器都是假定它所有的输入和输出都是正权。对于全加器的一般化形式有四种，如表 2.4 所示。表 2.4 中列出了这四类一般化全加器的名称、逻辑符号和完成的操作。每一类全加器都是用它输入端所包含的负权的个数来命名的。如 0 类全加器表示输入端均为正权，没有负权输入；1 类全加器表示有 1 个负权输入和 2 个正权输入；依此类推。

根据表 2.4，可以推导出这四类全加器 S 和 C 的逻辑表达式。对 0 类、3 类全加器而言有：

$$S=\overline{X}\overline{Y}Z+\overline{X}Y\overline{Z}+X\overline{Y}\overline{Z}+XYZ$$

$$C=XY+YZ+XZ \tag{2.42}$$

对 1 类、2 类全加器，则有：

$$S=\overline{X}\overline{Y}Z+\overline{X}Y\overline{Z}+X\overline{Y}\overline{Z}+XYZ$$

$$C=XY+X\overline{Z}+Y\overline{Z} \tag{2.43}$$

0 类和 3 类全加器中 S 和 C 的逻辑表达式相同，它和常规的全加器是一致的。这是因为 3 类全加器可以简单地把 0 类全加器的所有输入和输出值全部反相来得到，反之亦然。1 类和 2 类全加器中 S 和 C 的逻辑表达式也是相同的。表 2.2 和表 2.5 分别给出了 0 类全加器和

1 类全加器的真值表，读者可自行推导 0 类和 1 类全加器中 S 和 C 的逻辑表达式，注意表 2.5 中 Z 和 S 为负权，即它们取值为 0 时表示-0、取值为 1 时表示-1。

式(2.42)和式(2.43)都可改写成类似于式(2.27)的逻辑表达式，由类似于图 2.8(c)所示的与或非逻辑电路实现。因此各类全加器的和 S 与进位 C 的延迟时间相同，即通过这四类全加器中的任何一类全加器进行运算时，和与进位都是同时产生的。

表 2.4　四类一般化全加器的名称、逻辑符号和完成的操作

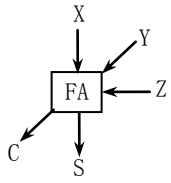
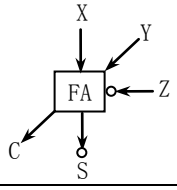
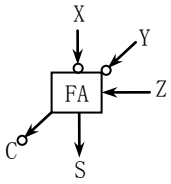
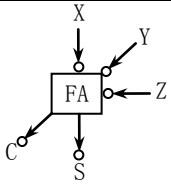
类型	逻辑符号	完成的操作
0 类 全加器		X Y + Z C S
1 类 全加器		X Y + (-Z) C (-S)
2 类 全加器		(-X) (-Y) + Z (-C) S
3 类 全加器		(-X) (-Y) + (-Z) (-C) (-S)

表 2.5　1 类全加器的真值表

输入			输出	
X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

2）直接补码阵列乘法器

利用混合型的全加器就可以构成直接补码阵列乘法器。设被乘数和乘数均为 n 位带符号的二进制整数补码，其中最高位为符号位，如：

$[A]_{补}=a_{n-1}a_{n-2}\cdots a_1a_0$　　$[B]_{补}=b_{n-1}b_{n-2}\cdots b_1b_0$

根据补码与真值之间的转换公式(2.38)，可得被乘数和乘数的真值分别为：

$A=(a_{n-1})a_{n-2}\cdots a_1a_0$　　$B=(b_{n-1})b_{n-2}\cdots b_1b_0$

这样$[A\times B]_{补}$就可以转换成两个真值 A、B 直接相乘，乘法过程如下所示：

$$
\begin{array}{rccccc}
 & (a_{n-1}) & a_{n-2} & \cdots & a_1 & a_0 \\
\times & (b_{n-1}) & b_{n-2} & \cdots & b_1 & b_0 \\
\hline
 & (a_{n-1}b_0) & a_{n-2}b_0 & \cdots & a_1b_0 & a_0b_0 \\
 & (a_{n-1}b_1) & a_{n-2}b_1 & \cdots & a_1b_1 & a_0b_1 \\
 & (a_{n-1}b_2) & a_{n-2}b_2 & \cdots & a_1b_2 & a_0b_2
\end{array}
$$

…

$+\quad a_{n-1}b_{n-1}\ (a_{n-2}b_{n-1})\ \cdots\ (a_1b_{n-1})\ (a_0b_{n-1})$

$(p_{2n-1})\ p_{2n-2}\ p_{2n-3}\ \cdots\ p_{n-1}\ p_{n-2}\ \cdots\ p_1\ p_0$

竖式中 p_{2n-2} 为符号位，p_{2n-1} 为扩充符号位，由于定点乘法运算的结果不会发生溢出，因此 p_{2n-1} 和 p_{2n-2} 的值一定相同。结果取单符号位，由此可得，$[A\times B]_{补}=p_{2n-2}p_{2n-3}\cdots p_1p_0$。

5 位×5 位的直接补码阵列乘法器逻辑原理图如图 2.17 所示，其中使用了不同的全加器符号来代表 0 类、1 类和 2 类全加器。直接补码阵列乘法器的工作原理与不带符号的阵列乘法器相同，不同之处仅在于直接补码阵列乘法器在运算过程中将符号位当作为负数参与运算。图 2.17 所示的方案称为三段阵列乘法器，其中右上角的三角形只用 0 类全加器，左上角的三角形只用 1 类全加器，阵列的最后两行只用 2 类全加器。图 2.17 下方输出端乘积中 p_8 为符号位，最高位 p_9 为扩充符号位，由于定点乘法运算的结果不会发生溢出，因此 p_9 和 p_8 的值一定相同。结果取单符号位，由此可得，$[A\times B]_{补}=p_8p_7p_6p_5p_4p_3p_2p_1p_0$。

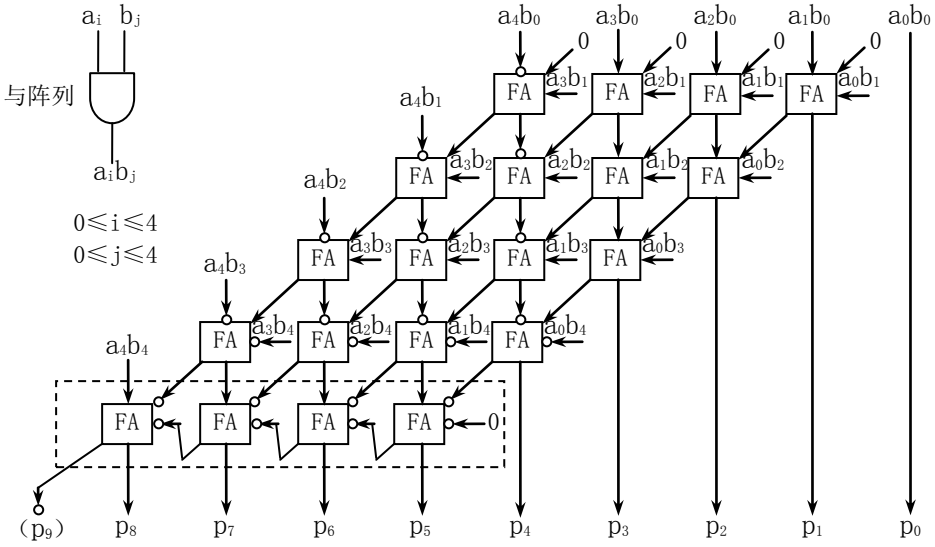


图 2.17　5 位×5 位直接补码阵列乘法器逻辑原理图

一般地，对于 n 位×n 位的直接补码阵列乘法器，需要(n-1)(n-2)/2 个 0 类全加器，(n-1)(n-2)/2 个 1 类全加器，(2n-2) 个 2 类全加器，总共是 n(n-1)个全加器。

[**例 2.38**]　设 x=-1101，y=1111，用直接补码阵列乘法器计算 x×y。

解：$[x]_{补}=10011$　　$[y]_{补}=01111$

$$
\begin{array}{rccccccccc}
 & & & (1) & 0 & 0 & 1 & 1 & & \\
 & & \times & (0) & 1 & 1 & 1 & 1 & & \\
\hline
 & & & (1) & 0 & 0 & 1 & 1 & & \\
 & & (1) & 0 & 0 & 1 & 1 & & & \\
 & & (1) & 0 & 0 & 1 & 1 & & & \\
 & (1) & 0 & 0 & 1 & 1 & & & & \\
+ & 0 & (0) & (0) & (0) & (0) & & & & \\
\hline
(1) & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1
\end{array}
$$

$[x\times y]_{补}=100111101$

所以　$x\times y=-11000011$

由于只有符号位才能带负权，若除符号位以外的其它乘积位中包含有负权，则需对乘积位进行调整。调整方法为：从右至左，若乘积位为负权，则向高一位借位（相当于进位为负权的 1），借 1 当 2，与本位的值相加，消除负权，直到除扩充符号位以外的其它乘积位全部变为正权为止。

[**例 2.39**]　设 x=0.1101，y=-0.1011，分别用带求补器的原码阵列乘法器、带求补器的补码阵列乘法器和直接补码阵列乘法器计算 x×y。

解：①带求补器的原码阵列乘法器

$[x]_{原}$=0.1101　　$[y]_{原}$=1.1011

乘积的符号位为：$x_f \oplus y_f=0 \oplus 1=1$

因符号位单独考虑，算前求补器的使能控制信号为 0，经算前求补后输出 x'=1101，y'=1011，其中 x'和 y'分别是 x 和 y 原码的数值位。

```
        1101
      × 1011
    ------
        1101
       1101
      0000
  +  1101
  ---------
    10001111
```

因算后求补器的使能控制信号为 0，经算后求补后输出为 10001111，加上乘积符号位 1，得

$[x \times y]_{原}$=1.10001111

所以　$x \times y$=-0.10001111

②带求补器的补码阵列乘法器

$[x]_{补}$=0.1101　　$[y]_{补}$=1.0101

乘积的符号位为：$x_f \oplus y_f=0 \oplus 1=1$

因算前求补器的使能控制信号分别为被乘数和乘数的符号位，经算前求补后输出 x'=1101，y'=1011，其中 x'和 y'分别是 x 和 y 原码的数值位。

```
        1101
      × 1011
    ------
        1101
       1101
      0000
  +  1101
  ---------
    10001111
```

因算后求补器的使能控制信号为乘积的符号位，经算后求补后输出为 01110001，加上乘积符号位 1，得

$[x \times y]_{补}$=1.01110001

所以　$x \times y$=-0.10001111

③直接补码阵列乘法器

$[x]_{补}$=0.1101　　$[y]_{补}$=1.0101

```
                    (0) 1  1  0  1
                ×   (1) 0  1  0  1
              ------------------------
                    (0) 1  1  0  1
                (0) 0  0  0  0
            (0) 1  1  0  1
        (0) 0  0  0  0
  +  0 (1)(1)(0)(1)
  ------------------------------------
  (1) 1  0  1  1  1  0  0  0  1
```

$[x \times y]_{补}$=1.01110001

所以　$x \times y$=-0.10001111

2.5　定点除法运算

2.5.1　原码一位除法

两个原码表示的数相除时，商的符号位由被除数和除数的符号位相异或求得，而商的数值部分由两个数的绝对值相除求得。

设被除数$[x]_{原}=x_f.x_1x_2\cdots x_n$，除数$[y]_{原}=y_f.y_1y_2\cdots y_n$

则有 $[x\div y]_{原}=(x_f\oplus y_f)+(0.x_1x_2\cdots x_n/0.y_1y_2\cdots y_n)$

对于定点小数，为使商不发生溢出，必须保证$|x|<|y|$；对于定点整数，为使商不发生溢出，必须保证双字$|x|$的高位字部分$<|y|$。

计算机实现原码除法，有恢复余数法和不恢复余数法两种方法。

1. 恢复余数法

设被除数 x=0.1001，除数 y=0.1011，x÷y 的人工计算过程如下：

```
                0.1101
          ------------
0.1011 ) 0.10010
         - 1011
         ------
           1110
         - 1011
         ------
           1100
         - 1011
         ------
              1
```

所以 x÷y=0.1101，余数=0.00000001

人工进行二进制除法的运算规则是：判断被除数与除数的大小，若被除数小于除数，则商 0，并在余数的最低位补 0，再用余数和右移一位后的除数相比，若余数大于除数，则商 1，否则商 0。然后重复上述步骤，直到除尽（即余数为 0）或已得到的商的位数满足精度要求为止。

上述计算方法要求加法器的位数为除数位数的两倍，通过分析可以发现，右移除数可以通过左移余数来替代，左移丢掉的余数的高位都是无用的零，对运算结果不会产生任何影响。由于计算机不会像人一样直接比较被除数（余数）与除数的大小，每次只能通过试商 1 来判断，即每次先用被除数（余数）减去除数。若运算所得的新的余数大于 0，表示试商 1 正确，余数和商左移 1 位，并开始下一次试商；若运算所得的新的余数小于 0，表示试商 1 错误，由于试商时已减去除数，所以必须先恢复余数，即加上除数，余数和商左移 1 位，再开始下一次试商。如此反复，直到除尽（即余数为 0）或已得到的商的位数满足精度要求为止。由于每次商 0 之前都要先恢复余数，因此这种方法称之为恢复余数法。

[**例 2.40**] x=0.1001，y=-0.1011，用原码恢复余数法计算 x÷y。

解：$[x]_{原}=0.1001$，$[y]_{原}=1.1011$

商的符号位为：$x_f\oplus y_f=0\oplus 1=1$

令 x’=0.1001，y’=0.1011，其中 x’和 y’分别为 x 和 y 的绝对值

$[x']_{补}=0.1001$　　$[y']_{补}=0.1011$　　$[-y']_{补}=1.0101$

被除数/余数	商	说明
00.1001		被除数$[x']_{补}$
$+[-y']_{补}$ 11.0101		试商，减去除数，即$+[-y']_{补}$
11.1110		余数＜0，商 0
$+[y']_{补}$ 00.1011		恢复余数，即$+[y']_{补}$
00.1001		
← 01.0010	0	余数和商左移 1 位
$+[-y']_{补}$ 11.0101		试商，减去除数，即$+[-y']_{补}$
00.0111		余数＞0，商 1
← 00.1110	0.1	余数和商左移 1 位
$+[-y']_{补}$ 11.0101		试商，减去除数，即$+[-y']_{补}$
00.0011		余数＞0，商 1
← 00.0110	0.11	余数和商左移 1 位
$+[-y']_{补}$ 11.0101		试商，减去除数，即$+[-y']_{补}$

	11.1011		余数＜0，商 0
+ [y']补	00.1011		恢复余数，即+[y']补
	00.0110		
←	00.1100	0.110	余数和商左移 1 位
+[-y']补	11.0101		试商，减去除数，即+[-y']补
	00.0001		余数＞0，商 1
		0.1101	商左移 1 位，最后一步余数不左移

所以　　$[x\div y]_{原}$=1.1101

$[余数]_{原}$=0.00000001　　其中，余数的符号位与被除数相同

即　　x÷y=-0.1101　　余数=0.00000001

用恢复余数法进行原码除法运算时，由于要进行恢复余数的操作，这不仅会降低运算的速度，而且控制线路复杂，因此在计算机中很少使用。计算机中普遍采用的是不恢复余数的除法方法，即加减交替法。

2. 不恢复余数法

不恢复余数法又称加减交替法，它是恢复余数法的一种变形。设 r_i 表示第 i 次运算后所得的余数，按照恢复余数法，有：

若 r_i＞0，则商 1，余数和商左移 1 位，再减去除数，即

$r_{i+1}=2r_i-y$

若 r_i＜0，则先恢复余数，再商 0，余数和商左移 1 位，再减去除数，即

$r_{i+1}=2(r_i+y)-y=2r_i+y$

由以上两点可以得出原码加减交替法的运算规则：

若 r_i＞0，则商 1，余数和商左移 1 位，再减去除数，即 $r_{i+1}=2r_i-y$；

若 r_i＜0，则商 0，余数和商左移 1 位，再加上除数，即 $r_{i+1}=2r_i+y$。

由于此种方法在运算时不需要恢复余数，因此称之为不恢复余数法。原码加减交替法是在恢复余数的基础上推导而来的，当末位商 1 时，所得到的余数与恢复余数法相同，是正确的余数。但当末位商 0 时，为得到正确的余数，需增加一步恢复余数，在恢复余数后，商左移一位，最后一步余数不左移。

[**例 2.41**]　x=0.1001，y=-0.1011，用原码加减交替法计算 x÷y。

解：$[x]_{原}$=0.1001，$[y]_{原}$=1.1011

商的符号位为：$x_f\oplus y_f=0\oplus 1=1$

令 x'=0.1001，y'=0.1011，其中 x' 和 y' 分别为 x 和 y 的绝对值

$[x']_{补}$=0.1001　　$[y']_{补}$=0.1011　　$[-y']_{补}$=1.0101

被除数/余数		商	说明
	00.1001		被除数$[x']_{补}$
+[-y']补	11.0101		试商，减去除数，即+[-y']补
	11.1110		余数＜0，商 0
←	11.1100	0	余数和商左移 1 位
+ [y']补	00.1011		加上除数，即+[y']补
	00.0111		余数＞0，商 1
←	00.1110	0.1	余数和商左移 1 位
+[-y']补	11.0101		减去除数，即+[-y']补
	00.0011		余数＞0，商 1
←	00.0110	0.11	余数和商左移 1 位
+[-y']补	11.0101		减去除数，即+[-y']补
	11.1011		余数＜0，商 0
←	11.0110	0.110	余数和商左移 1 位
+ [y']补	00.1011		加上除数，即+[y']补
	00.0001		余数＞0，商 1
		0.1101	商左移 1 位，最后一步余数不左移

所以　[x÷y]原=1.1101

[余数]原=0.00000001　　其中，余数的符号位与被除数相同

即　x÷y=-0.1101　　余数=0.00000001

由例 2.41 可以看出，运算过程中每一步所上的商正好与当前运算结果的符号位相反，在原码加减交替除法硬件设计时每一步所上的商便是由运算结果的符号位取反得到的。由例 2.41 还可以看出，当被除数（余数）和除数为单符号时，运算过程中每一步所上的商正好与符号位运算向前产生的进位相同，在原码阵列除法器硬件设计时每一步所上的商便是由单符号位运算向前产生的进位得到的。

[**例 2.42**]　x=-10110000，y=1101，用原码加减交替法计算 x÷y。

解：[x]原=110110000，[y]原=01101

商的符号位为：$x_f \oplus y_f = 1 \oplus 0 = 1$

令 x'=10110000，y'=1101，其中 x'和 y'分别为 x 和 y 的绝对值，即数值部分

[x']补=010110000　　[y']补=01101　　[-y']补=10011

被除数/余数		商 q	说明
	0010110000		被除数[x']补
+[-y']补	110011		试商，减去除数，即+[-y']补
	1111100000		余数＜0，商 0
←	111100000	0	余数和商左移 1 位
+ [y']补	001101		加上除数，即+[y']补
	001001000		余数＞0，商 1
←	01001000	01	余数和商左移 1 位
+[-y']补	110011		减去除数，即+[-y']补
	00010100		余数＞0，商 1
←	0010100	011	余数和商左移 1 位
+[-y']补	110011		减去除数，即+[-y']补
	1111010		余数＜0，商 0
←	111010	0110	余数和商左移 1 位
+ [y']补	001101		加上除数，即+[y']补
	000111		余数＞0，商 1
		01101	商左移 1 位，最后一步余数不左移

所以　[x÷y]原=11101

[余数]原=10111　　其中，余数的符号位与被除数相同

即　x÷y=-1101　　余数=-0111

2.5.2　补码一位除法

在补码一位除法中仅讨论加减交替法，补码一位除法与补码加、减、乘法运算一样，符号位和数一起参与运算。在补码加减交替法中，是通过比较被除数（余数）和除数的符号位来上商的，其运算规则如下：

(1)若被除数与除数同号，则用被除数减去除数；若被除数与除数异号，则用被除数加上除数。被除数经过一次运算后，我们称之为余数。

(2)若余数与除数同号，则商 1，余数和商左移 1 位，再减去除数；若余数与除数异号，则商 0，余数和商左移 1 位，再加上除数。

(3)重复(2)，若商的校正采用“末位恒置 1 法”，则包括符号位在内重复(2)的操作共 n 次（设数值位为 n 位），此时最大误差为 $\pm 2^{-n}$；若商的校正采用“校正法”，则包括符号位在内重复(2)的操作共 n+1 次（设数值位为 n 位）。商的校正一般采用“末位恒置 1 法”，如要提高精度，则按上述规则多求一位商，再对商进行校正。

[**例 2.43**]　x=0.1001，y=-0.1011，用补码加减交替法计算 x÷y。

解：[x]补=0.1001，[y]补=1.0101，[-y]补=0.1011

被除数/余数	商	说明

	00.1001		被除数与除数异号
+ $[y]_{补}$	11.0101		加上除数，即+$[y]_{补}$
	11.1110		余数与除数同号，商 1
←	11.1100	1	余数和商左移 1 位
+ $[-y]_{补}$	00.1011		减去除数，即+$[-y]_{补}$
	00.0111		余数与除数异号，商 0
←	00.1110	1.0	余数和商左移 1 位
+ $[y]_{补}$	11.0101		加上除数，即+$[y]_{补}$
	00.0011		余数与除数异号，商 0
←	00.0110	1.00	余数和商左移 1 位
+ $[y]_{补}$	11.0101		加上除数，即+$[y]_{补}$
	11.1011		余数与除数同号，商 1
←	11.0110	1.001	余数和商左移 1 位
+ $[-y]_{补}$	00.1011		减去除数，即+$[-y]_{补}$
	00.0001		商的末位恒置 1
		1.0011	商左移 1 位，最后一步余数不左移

所以　$[x \div y]_{补}=1.0011$

$[余数]_{补}=0.00000001$

即　$x \div y=-0.1101$　　余数=0.00000001

[**例 2.44**]　x=-10110000，y=-1101，用补码加减交替法计算 x÷y。

解：$[x]_{补}=101010000$，$[y]_{补}=10011$，$[-y]_{补}=01101$

	被除数/余数	商	说明
	1101010000		被除数与除数同号
+ $[-y]_{补}$	001101		减去除数，即+$[-y]_{补}$
	0000100000		余数与除数异号，商 0
←	000100000	0	余数和商左移 1 位
+ $[y]_{补}$	110011		加上除数，即+$[y]_{补}$
	110111000		余数与除数同号，商 1
←	10111000	01	余数和商左移 1 位
+ $[-y]_{补}$	001101		减去除数，即+$[-y]_{补}$
	11101100		余数与除数同号，商 1
←	1101100	011	余数和商左移 1 位
+ $[-y]_{补}$	001101		减去除数，即+$[-y]_{补}$
	0000110		余数与除数异号，商 0
←	000110	0110	余数和商左移 1 位
+ $[y]_{补}$	110011		加上除数，即+$[y]_{补}$
	111001		商的末位恒置 1
		01101	商左移 1 位，最后一步余数不左移

所以　$[x \div y]_{补}=01101$

$[余数]_{补}=11001$

即　$x \div y=1101$　　余数=-0111

2.5.3　阵列除法器

为了提高除法运算的速度，可采用与阵列乘法器相似的思想来设计阵列除法器。阵列除法器有多种形式，这里仅以不恢复余数的原码阵列除法器为例，来介绍这类除法器的设计原理。在介绍不恢复余数的原码阵列除法器之前，首先介绍可控加法/减法（CAS）单元，它是原码阵列除法器的基本构件。

1. 可控加法/减法（CAS）单元

可控加法/减法（CAS）单元的内部电路图如图 2.18 所示，它由一个异或门和一个全加

器（FA）组成，$B_i \oplus P$ 的结果、A_i 和 C_i 被送入全加器进行加法运算，产生的和为 S_i，进位为 C_{i+1}。当控制信号 P=0 时，$B_i \oplus P=B_i$，CAS 完成 A_i 加 B_i 加 C_i 运算；当控制信号 P=1 时，$B_i \oplus P=\overline{B_i}$，CAS 完成 A_i 加 $\overline{B_i}$ 加 C_i 运算。若 $[B]_{补}$ 的每一位 B_i 都受同一个控制信号 P 控制，并且采用串行进位，串行进位的最低进位端为 P 信号，则当 P=0 时，CAS 完成加法运算，当 P=1 时，CAS 完成减法运算。

图 2.18 中，CAS 单元的逻辑表达式为：

$$S_i=A_i \oplus (B_i \oplus P) \oplus C_i$$

$$C_{i+1}=(A_i+C_i) \cdot (B_i \oplus P)+A_iC_i \qquad (2.44)$$

式(2.44)中的 S_i、C_{i+1} 都可改写成类似于式(2.27)的逻辑表达式，由类似于图 2.8(c)所示的与或非逻辑电路实现。因此 CAS 单元产生和 S_i 与进位 C_{i+1} 的延迟时间相同，即和与进位都是同时产生的。

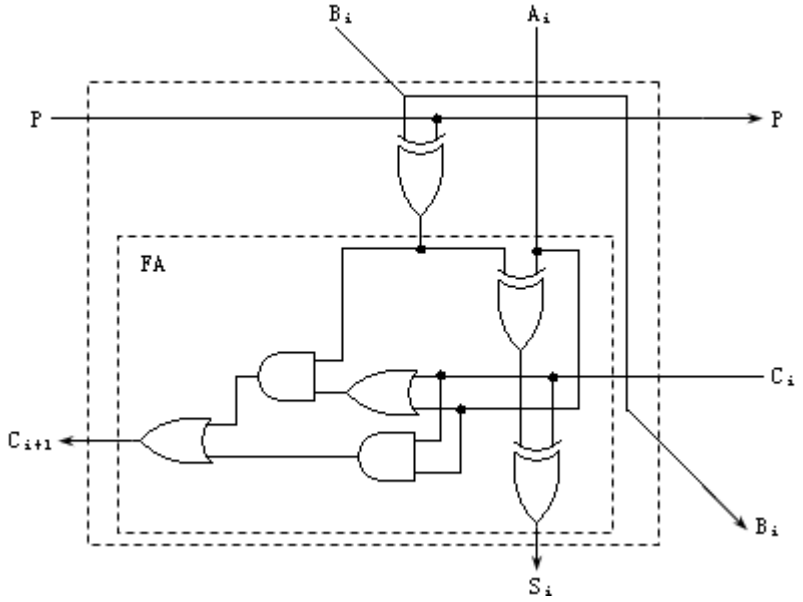


图 2.18　可控加法/减法（CAS）单元的内部电路图

2. 不恢复余数的原码阵列除法器

不恢复余数的原码阵列除法器利用了不恢复余数原码一位除法中的设计思想，它利用两个数原码的数值部分直接相除，其中被除数为双字长数，除数为单字长数，为避免运算结果发生溢出，要求被除数高位字部分的绝对值必须要小于除数的绝对值。加减运算时采用单符号位补码完成，在 2.5.1 节中曾介绍过，当被除数（余数）和除数为单符号时，运算过程中每一步所上的商正好与符号位运算向前产生的进位相同，这里的原码阵列除法器正是根据此特点来上商的，如图 2.19 所示。

由于第一步是试商，需用被除数减去除数，因此图 2.19 中第一行的 P 控制端恒为 1。在原码加减交替法中，若商为 1，则余数和商一起左移 1 位再减去除数；若商 0，则余数和商一起左移 1 位再加上除数。在原码阵列除法器中，被除数（余数）的位置不变，与人工算法相同，若商为 1，则向右错开 1 位，减去除数；若商 0，则向右错开 1 位，加上除数。

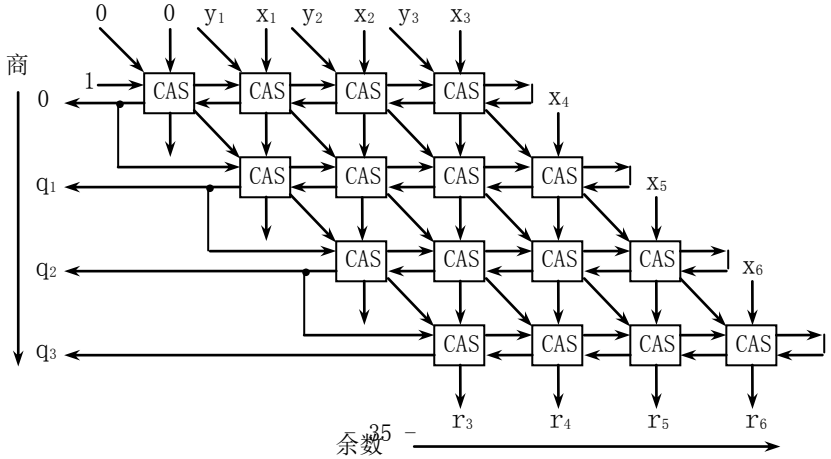


图 2.19　不恢复余数的原码阵列除法器

若被除数$[x]_{原}$的数值部分为 $x'=x_1x_2x_3x_4x_5x_6$，除数$[y]_{原}$的数值部分为 $y'=y_1y_2y_3$，则原码阵列除法器实际上是完成$[x']_{补}\div[y']_{补}$，即完成 $0x_1x_2x_3x_4x_5x_6\div0y_1y_2y_3$，如图 2.19 所示，运算结果为：

$[x']_{补}\div[y']_{补}$的商 $q=q_0q_1q_2q_3$，其中 q_0一定为 0，余数 $r=r_3r_4r_5r_6$

设商的符号 $q_f=x_f\oplus y_f$，其中 x_f和 y_f分别为$[x]_{原}$和$[y]_{原}$中的符号位。

①对定点小数，则$[x\div y]_{原}=q_f.q_1q_2q_3$，$[余数]_{原}=x_f.00r_3r_4r_5r_6$，余数与被除数同号；

②对定点整数，则$[x\div y]_{原}=q_fq_1q_2q_3$，$[余数]_{原}=x_fr_4r_5r_6$，余数与被除数同号。

当最后一位商 0 时，由于采用的是不恢复余数法，此时的余数会有误差。

[**例 2.45**]　设 x=101001，y=−111，用原码阵列除法器计算 x÷y。

解：$[x]_{原}=0101001$　　$[y]_{原}=1111$

商的符号位为：$x_f\oplus y_f=0\oplus1=1$

令 x'=101001，y'=111，其中 x'和 y'分别为$[x]_{原}$和$[y]_{原}$的数值部分

$[x']_{补}=0101001$，$[y']_{补}=0111$，$[-y']_{补}=1001$

被除数/余数	商	说明
0101001		被除数$[x']_{补}$
$+[-y']_{补}$　1001		第一步减去除数，即$+[-y']_{补}$
1110001	$q_0=0$	最高位向前产生的进位为 0，即商 0
$+[y']_{补}\longrightarrow$　0111		向右错开 1 位，加上除数，即$+[y']_{补}$
001101	$q_1=1$	最高位向前产生的进位为 1，即商 1
$+[-y']_{补}\longrightarrow$　1001		向右错开 1 位，减去除数，即$+[-y']_{补}$
11111	$q_2=0$	最高位向前产生的进位为 0，即商 0
$+[y']_{补}\longrightarrow$　0111		向右错开 1 位，加上除数，即$+[y']_{补}$
0110	$q_3=1$	最高位向前产生的进位为 1，即商 1

故得　商 $q=q_0q_1q_2q_3=0101$

余数 $r=r_3r_4r_5r_6=0110$

所以　$[x\div y]_{原}=1101$

$[余数]_{原}=0110$　　其中，余数的符号位与被除数相同

即　x÷y=−101，余数=110

[**例 2.46**]　设 x=−0.10110，y=0.11011，用原码阵列除法器计算 x÷y。

$[x]_{原}=1.10110$　　$[y]_{原}=0.11011$

解：商的符号位为：$x_f\oplus y_f=1\oplus0=1$

令 x'=1011000000，y'=11011，其中 x'和 y'分别为$[x]_{原}$和$[y]_{原}$的数值部分，且 x'为双字长

$[x']_{补}=01011000000$，$[y']_{补}=011011$，$[-y']_{补}=100101$

被除数/余数	商	说明
01011000000		被除数$[x']_{补}$
$+[-y']_{补}$　100101		第一步减去除数，即$+[-y']_{补}$
11101100000	$q_0=0$	最高位向前产生的进位为 0，即商 0
$+[y']_{补}\longrightarrow$　011011		向右错开 1 位，加上除数，即$+[y']_{补}$
0100010000	$q_1=1$	最高位向前产生的进位为 1，即商 1
$+[-y']_{补}\longrightarrow$　100101		向右错开 1 位，减去除数，即$+[-y']_{补}$
000111000	$q_2=1$	最高位向前产生的进位为 1，即商 1
$+[-y']_{补}\longrightarrow$　100101		向右错开 1 位，减去除数，即$+[-y']_{补}$
11001100	$q_3=0$	最高位向前产生的进位为 0，即商 0

$+[y']_{补}$ ⟶	011011		向右错开 1 位，加上除数，即$+[y']_{补}$
	0000010	$q_4=1$	最高位向前产生的进位为 1，即商 1
$+[-y']_{补}$ ⟶	100101		向右错开 1 位，减去除数，即$+[-y']_{补}$
	100111	$q_5=0$	最高位向前产生的进位为 0，即商 0

故得　商 $q=q_0q_1q_2q_3q_4q_5=011010$

余数 $r=r_5r_6r_7r_8r_9r_{10}=100111$

所以　$[x\div y]_{原}=1.11010$

$[余数]_{原}=1.0000100111$　　其中，余数的符号位与被除数相同

即　$x\div y=-0.11010$　　余数$=-0.0000100111$

2.6　逻辑运算和移位运算

计算机中除了进行加、减、乘、除等基本算术运算外，还可以进行逻辑运算和移位运算。

2.6.1　逻辑运算

计算机中常用的逻辑运算主要有逻辑非、逻辑加、逻辑乘、逻辑异或等四种。利用逻辑运算可以对某个寄存器或存储单元内容的某一位或某几位进行取反、测试、复位或置位等操作。

1. 逻辑非

逻辑非也称求反。对某个数进行逻辑非运算，就是对它进行按位求反运算。常用变量上方加一横线来表示。一位二进制数的逻辑非运算规则如表 2.6 所示。

表 2.6　逻辑非运算规则

x_i	$\overline{x_i}$
0	1
1	0

设一个数 x 表示成：

$$X=X_nX_{n-1}\cdots X_2X_1X_0$$

若 $\overline{x}=Z=Z_nZ_{n-1}\cdots Z_2Z_1Z_0$，则有：

$$Z_i=\overline{x_i}\ ,\quad i=0,1,2,\cdots,n$$

[**例 2.47**]　x=10100011，y=00001111，求 $\overline{x}$，$\overline{y}$。

解： $\overline{x}=01011100$

$\overline{y}=11110000$

2. 逻辑加

对两个数进行逻辑加，就是对这两个数进行按位求“或”运算，所以逻辑加又称逻辑或。常用记号“∨”或“+”表示。一位二进制数的逻辑加运算规则如表 2.7 所示，当两个变量的取值中有一个为 1 时，逻辑加的运算结果为 1，只有当两个变量的取值同时为 0 时，逻辑加的运算结果才为 0。

表 2.7　逻辑加运算规则

x_i	y_i	$x_i \vee y_i$
0	0	0
0	1	1
1	0	1
1	1	1

设有两个数 x 和 y，表示成：

$$x=x_nx_{n-1}\cdots x_2x_1x_0,\quad y=y_ny_{n-1}\cdots y_2y_1y_0$$

若 $x\vee y=z=z_nz_{n-1}\cdots z_2z_1z_0$，则有：

$$z_i=x_i\vee y_i\ ,\quad i=0,1,2,\cdots,n$$

[**例 2.48**]　x=10000101，y=11110000，求 $x\vee y$。

解：　10000101

$$\begin{array}{rr} \vee & 11110000 \\ \hline & 11110101 \end{array}$$

即　　$x \vee y=11110101$

3. 逻辑乘

对两个数进行逻辑乘，就是对这两个数进行按位求“与”运算，所以逻辑乘又称逻辑与。常用记号“∧”或“ • ”表示。一位二进制数的逻辑乘运算规则如表 2.8 所示，当两个变量的取值中有一个为 0 时，逻辑乘的运算结果为 0，只有当两个变量的取值同时为 1 时，逻辑乘的运算结果才为 1。

设有两个数 x 和 y，表示成：

$$x=x_n x_{n-1}\cdots x_2 x_1 x_0, \quad y=y_n y_{n-1}\cdots y_2 y_1 y_0$$

若 $x \wedge y=z=z_n z_{n-1}\cdots z_2 z_1 z_0$，则有：

$$z_i=x_i \wedge y_i, \quad i=0, 1, 2, \cdots, n$$

表 2.8　逻辑乘运算规则

x_i	y_i	$x_i \wedge y_i$
0	0	0
0	1	0
1	0	0
1	1	1

[**例 2.49**]　x=01100101，y=11001100，求 $x \wedge y$。

解：

$$\begin{array}{rr} & 01100101 \\ \wedge & 11001100 \\ \hline & 01000100 \end{array}$$

即　　$x \wedge y=01000100$

4. 逻辑异或

对两个数进行逻辑异或，就是对这两个数进行按位求“模 2 和”运算，即按位相加不考虑进位，所以逻辑异或又称按位加。常用记号“⊕”表示。一位二进制数的逻辑异或运算规则如表 2.9 所示，当两个变量的取值相异时，逻辑异或的运算结果为 1；相反，当两个变量的取值相同时，逻辑异或的运算结果为 0。

表 2.9　逻辑异或运算规则

x_i	y_i	$x_i \oplus y_i$
0	0	0
0	1	1
1	0	1
1	1	0

设有两个数 x 和 y，表示成：

$$x=x_n x_{n-1}\cdots x_2 x_1 x_0, \quad y=y_n y_{n-1}\cdots y_2 y_1 y_0$$

若 $x \oplus y=z=z_n z_{n-1}\cdots z_2 z_1 z_0$，则有：

$$z_i=x_i \oplus y_i, \quad i=0, 1, 2, \cdots, n$$

[**例 2.50**]　x=01101111，y=11110000，求 $x \oplus y$。

解：

$$\begin{array}{rr} & 01101111 \\ \oplus & 11110000 \\ \hline & 10011111 \end{array}$$

即　　$x \oplus y=10011111$

2.6.2　移位运算

计算机中机器数的字长往往是固定的，当机器数左移 n 位或右移 n 位时，必然会使其低 n 位或高 n 位出现空位。那么，对空出的空位应该添补 0 还是添补 1 呢？这与机器数采用的是有符号数还是无符号数有关。对有符号数的移位称为算术移位，对无符号数的移位称为逻辑移位。

1. 算术移位

对于正数，由于$[x]_{原}=[x]_{补}=[x]_{反}=x$ 的真值，故移位后出现的空位均添补 0。对于负数，由于原码、补码和反码的表示形式不同，故当机器数移位时，对其空位的添补规则也不同。表 2.10 列出了三种不同机器码表示（整数或小数均可），分别对应正数或负数，移位后的添补规则。必须注意的是，对于原码和反码表示的数，不论是正数还是负数，移位操作只针对尾数部分，其符号位均保持不变；对于补码表示的数，不论是正数还是负数，移位操作针对整个机器数，包括符号位。

由表 2.10 可得出如下结论：

（1）机器数为正时，不论左移还是右移，添补代码均为 0。

（2）由于负数原码的尾数部分与真值相同，故在移位时符号位不变，其空位均添补 0。

（3）由于负数反码的尾数部分与其原码的尾数部分正好相反，故在移位时符号位不变，其空位均添补 1。

（4）负数的补码在左移时，低位出现的空位均补 0；负数的补码在右移时，高位出现的空位均添补 1。

表 2.10　不同机器码表示的机器数算术移位后的空位添补规则

	机器码	添补代码
正数	原码、补码、反码	0
负数	原码	0
	补码	左移添补 0
		右移添补 1
	反码	1

机器中实现算术左移和算术右移的操作示意图如图 2.20 所示。其中(a)表示真值为正的原码和反码表示的机器数的移位操作；(b)表示补码表示的机器数的移位操作；(c)表示真值为负的原码表示的机器数的移位操作；(d)表示真值为负的反码表示的机器数的移位操作。

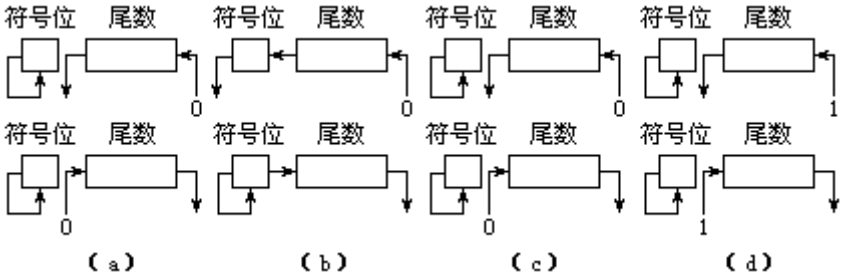


图 2.20　实现算术左移和算术右移的操作示意图

[**例 2.51**]　设机器字长为 8 位（含一位符号位），若 x=38，y=-38，分别写出 x、y 的原码、补码和反码表示的机器数在左移一位、左移两位、右移一位和右移两位后的机器数及对应的真值。

解：(1)$x=38=(100110)_2$

x 的三种机器码表示及移位结果如表 2.11 所示。

表 2.11　对 x=38 算术移位后的结果

移位操作	机器数		对应的真值
移位前	原码、反码	00100110	+38
左移一位		01001100	+76
左移两位		00011000	+24
右移一位		00010011	+19
右移两位		00001001	+9
移位前	补码	00100110	+38
左移一位		01001100	+76
左移两位		10011000	-104
右移一位		00010011	+19
右移两位		00001001	+9

可见，对于正数，原码和反码表示的机器数移位后符号位均保持不变，而补码表示的机器数移位后若出现溢出（正溢），符号位会发生改变。算术左移时，若尾数的最高位丢掉的是 1，结果出错；算术右移时，若尾数的最低位丢掉的是 1，影响精度。

(2) $y=-38=(-100110)_2$

y 的三种机器码表示及移位结果如表 2.12 所示。

表 2.12　对 y=-38 算术移位后的结果

移位操作	机器数		对应的真值
移位前	原码	10100110	-38
左移一位		11001100	-76
左移两位		10011000	-24
右移一位		10010011	-19
右移两位		10001001	-9
移位前	补码	11011010	-38
左移一位		10110100	-76
左移两位		01101000	+104
右移一位		11101101	-19
右移两位		11110110	-10
移位前	反码	11011001	-38
左移一位		10110011	-76
左移两位		11100111	-24
右移一位		11101100	-19
右移两位		11110110	-9

可见，对于负数，原码和反码表示的机器数移位后符号位均保持不变，而补码表示的机器数移位后若出现溢出（负溢），符号位会发生改变。负数的原码算术左移时，若尾数的最高位丢掉的是 1，结果出错，算术右移时，若尾数的最低位丢掉的是 1，影响精度；负数的反码算术左移时，若尾数的最高位丢掉的是 0，结果出错，算术右移时，若尾数的最低位丢掉的是 0，影响精度；负数的补码算术左移时，若符号位发生改变，结果出错，算术右移时，若尾数的最低位丢掉的是 1，影响精度。

2. 逻辑移位

由于逻辑移位针对的是无符号数，所以移位规则很简单。逻辑左移时，机器数的最高位移出，最低位添补 0；逻辑右移时，机器数的最低位移出，最高位添补 0。机器中实现逻辑左移和逻辑右移的操作示意图如图 2.21 所示。

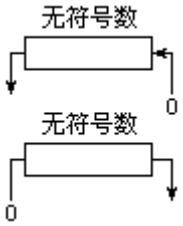
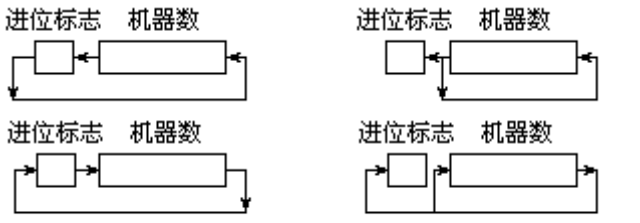


图 2.21　实现逻辑左移和逻辑右移的操作示意图

3. 循环移位

循环移位包括带进位标志的循环移位和不带进位标志的循环移位，每一种又包括左移和右移两种。在带进位标志的循环移位中，进位标志和机器数的所有位形成闭合的移位环路。在不带进位标志的循环移位中，机器数的最高位和最低位形成闭合的移位环路。在进行循环左移操作时，整个环路一起向左移动；在进行循环右移操作时，整个环路一起向右移动。进位标志的值为最后一次从机器数中移出的代码。机器中实现循环移位的操作示意图如图 2.22 所示。



(a) 带进位标志的循环移位　　(b) 不带进位标志的循环移位

图 2.22　实现循环移位的操作示意图

2.7　定点运算器的组成与结构

运算器是 CPU 的重要组成部分，主要用来进行数据的加工处理，完成各种算术运算和逻辑运算。尽管各种计算机的运算器在设计上有较大的区别，但它们最基本的结构中必须有算术逻辑运算单元(ALU)、通用寄存器、多路开关/锁存器、三态缓冲器和数据总线等逻辑构件，运算器的核心是算术逻辑运算单元。在计算机中，所有的算术运算和逻辑运算，一般都可以通过加法器来实现，因此，加法器是运算器中的一个最基本、最重要的部件。

2.7.1　多功能算术逻辑运算单元

集成电路的发展使人们可利用现成的集成电路芯片像搭积木一样构成 ALU。常见的产品有 SN74181，一片能完成 4 位数的算术运算和逻辑运算。当然，也有其它能完成 8 位、16 位数算术运算和逻辑运算的芯片。下面先介绍 SN74181 芯片，然后再介绍 ALU 芯片的扩展。

1. 一位 ALU 单元

图 2.23(a)所示为 SN74181 中的一位 ALU 单元，图中 $\overline{A_i}$、$\overline{B_i}$ 分别表示参加运算的数的某一位（反变量表示），C_{n+i} 表示低位的进位信号，$\overline{F_i}$ 表示运算的和（反变量表示），M 用来控制 ALU 进行算术运算还是逻辑运算。由于 SN74181 采用先行进位，所以在此图中未画出全加器向高位产生的进位 C_{n+i+1}。一位 ALU 单元可分作函数发生器和一位全加器，如图 2.23(b)所示。为了将全加器的功能进行扩展以完成多种算术或逻辑运算，这里没有将 $\overline{A_i}$、$\overline{B_i}$ 和低位的进位信号 C_{n+i} 直接进行全加，而是将 $\overline{A_i}$ 和 $\overline{B_i}$ 先经过由控制参数 S_0、S_1、S_2、S_3 控制的函数发生器形成组合函数 X_i 和 Y_i，然后再将 X_i、Y_i 和低位产生的进位 C_{n+i} 通过全加器进行全加，产生和数 F_i 和进位 C_{n+i+1}。

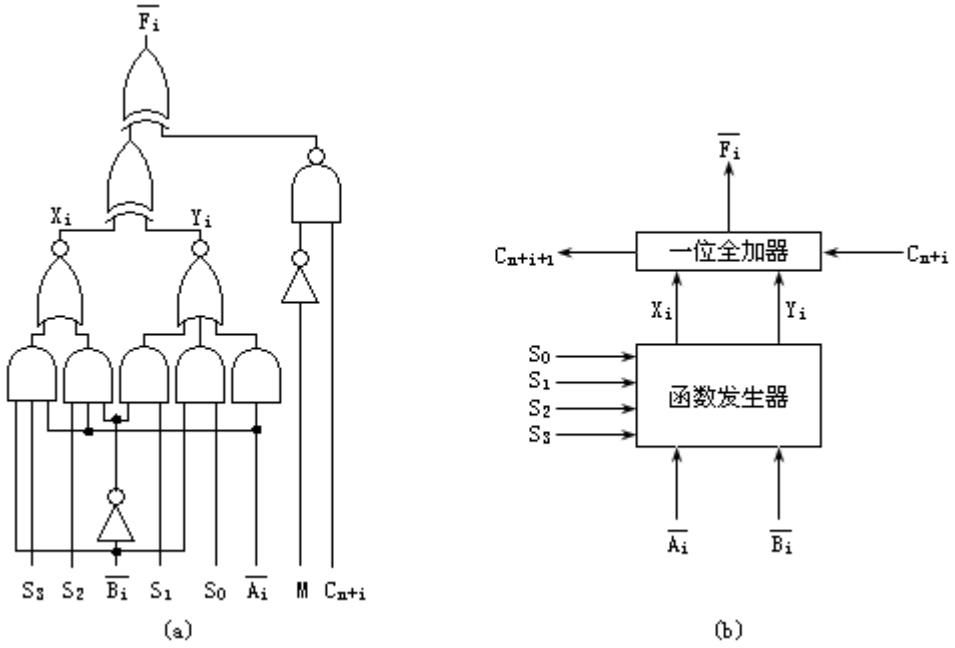


图 2.23　一位 ALU 单元

图 2.23 中，一位全加器的逻辑表达式为：

$$\overline{F_i} = X_i \oplus Y_i \oplus \overline{C_{n+i}}$$

$$C_{n+i+1}=X_iY_i+Y_iC_{n+i}+X_iC_{n+i} \tag{2.45}$$

式(2.45)中由于运算的和$\overline{F_i}$为反变量，所以在$\overline{F_i}$的表达式中，C_{n+i}取了反变量$\overline{C_{n+i}}$，如图 2.23(a)所示。式中的下标 i 表示 ALU 输入输出数据的第 i 位，下标 n 则表示由一个或多个 ALU 芯片组成的某个完整的 ALU。

控制参数 S_0、S_1、S_2、S_3分别控制输入$\overline{A_i}$和$\overline{B_i}$，产生 X_i和 Y_i函数。其中 Y_i是受 S_0、S_1控制的 A_i和 B_i的组合函数，而 X_i是受 S_2、S_3控制的 A_i和 B_i的组合函数，其函数关系如表 2.13 所示。

表 2.13　X_i、Y_i与控制参数和输入量的关系

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

根据表 2.13 所给出的函数关系，即可列出 X_i和 Y_i的逻辑表达式：

$$Y_i=\overline{S_0}\,\overline{S_1}A_i+\overline{S_0}S_1A_i\overline{B_i}+S_0\overline{S_1}A_iB_i$$

$$X_i=\overline{S_2}\,\overline{S_3}+\overline{S_2}S_3(A_i+B_i)+S_2\overline{S_3}(A_i+\overline{B_i})+S_2S_3A_i \tag{2.46}$$

对式(2.46)进一步化简后，代入上面的式(2.45)，可得到一位 ALU 的输出与输入之间的逻辑表达式：

$$Y_i=\overline{\overline{A_i}+S_0\overline{B_i}+S_1B_i}$$

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

$$\overline{F_i}=X_i\oplus Y_i\oplus\overline{C_{n+i}}$$

$$C_{n+i+1}=Y_i+X_iC_{n+i} \tag{2.47}$$

在一位 ALU 单元中，实现 X_i、Y_i和$\overline{F_i}$表达式功能的逻辑电路如图 2.23(a)所示。

2. SN74181 芯片

SN74181 芯片包含 4 个一位的 ALU 单元，并有 4 位并行的进位链，即根据最低位的进位 C_n和全加器的输入 X_i、Y_i直接产生其它位的进位输入信号 C_{n+1}、C_{n+2}、C_{n+3}，以及最高位的进位输出信号 C_{n+4}。除此之外，还提供了$\overline{P}$和$\overline{G}$以实现组间并行进位。下面，我们来根据式(2.47)中进位 C_{n+i+1}的表达式进行推导。

当 i=0 时，$C_{n+1}=Y_0+X_0C_n$

当 i=1 时，$C_{n+2}=Y_1+X_1C_{n+1}=Y_1+Y_0X_1+X_0X_1C_n$

当 i=2 时，$C_{n+3}=Y_2+X_2C_{n+2}=Y_2+Y_1X_2+Y_0X_1X_2+X_0X_1X_2C_n$

当 i=3 时，$C_{n+4}=Y_3+X_3C_{n+3}=Y_3+Y_2X_3+Y_1X_2X_3+Y_0X_1X_2X_3+X_0X_1X_2X_3C_n$

设　　$G=Y_3+Y_2X_3+Y_1X_2X_3+Y_0X_1X_2X_3$

$$P=X_0X_1X_2X_3$$

则　　$$C_{n+4}=G+PC_n=\overline{\overline{G}\,\overline{P}+\overline{G}\,\overline{C_n}}$$

$$\overline{G}=\overline{Y_3+Y_2X_3+Y_1X_2X_3+Y_0X_1X_2X_3}$$

$$\overline{P}=\overline{X_0X_1X_2X_3} \tag{2.48}$$

这样，对一片 4 位的 ALU 来说，可有三个进位输出。其中 G 称为进位发生输出，P 称为进位传送输出。由 G、P 的表达式可知，它们仅与函数发生器的输出有关，与进位无关，因此，在电路中多加这两个进位来实现高速运算。G、P 进位输出常与先行进位发生器 SN74182CLA 配合使用，来实现组间并行进位。

根据式(2.47)和式(2.48)设计出来的负逻辑表示的 SN74181 逻辑电路如图 2.24 所示。图 2.24 下半部分为函数发生器，上半部分为全加器和并行进位链。由于输入变量在控制参数 S_0、S_1、S_2、S_3的控制下形成组合函数 X_i和 Y_i后再送给全加器进行运算，并且 M 的控制又区分了算术运算和逻辑运算，因此 SN74181ALU 具有较强的算术运算和逻辑运算功能。

图 2.24 右边的控制信号 M 用来控制 ALU 是进行算术运算还是逻辑运算。当 M=0 时，M

经非门后输出为 1，M 对进位信号没有任何影响。此时 $\overline{F_i}$ 不仅与本位的 X_i 和 Y_i 有关，而且与向本位的进位值 C_{n+i} 有关，执行算术运算；当 M=1 时，M 经非门后输出为 0，封锁了各位的进位输出，即 C_{n+i}=0。此时 $\overline{F_i}$ 仅与本位的 X_i 和 Y_i 有关，执行与进位无关的逻辑运算。

SN74181ALU 有两种工作方式，即负逻辑输入与输出和正逻辑输入与输出，它们的区别如表 2.14 所示。表 2.15 分别列出了 SN74181ALU 分别在两种工作方式下的运算功能。由于 S_0、S_1、S_2、S_3有 16 种状态组合，因此对于负逻辑输入与输出而言，有 16 种算术运算功能和 16 种逻辑运算功能。同样，对于正逻辑输入与输出而言，也有 16 种算术运算功能和 16 种逻辑运算功能。

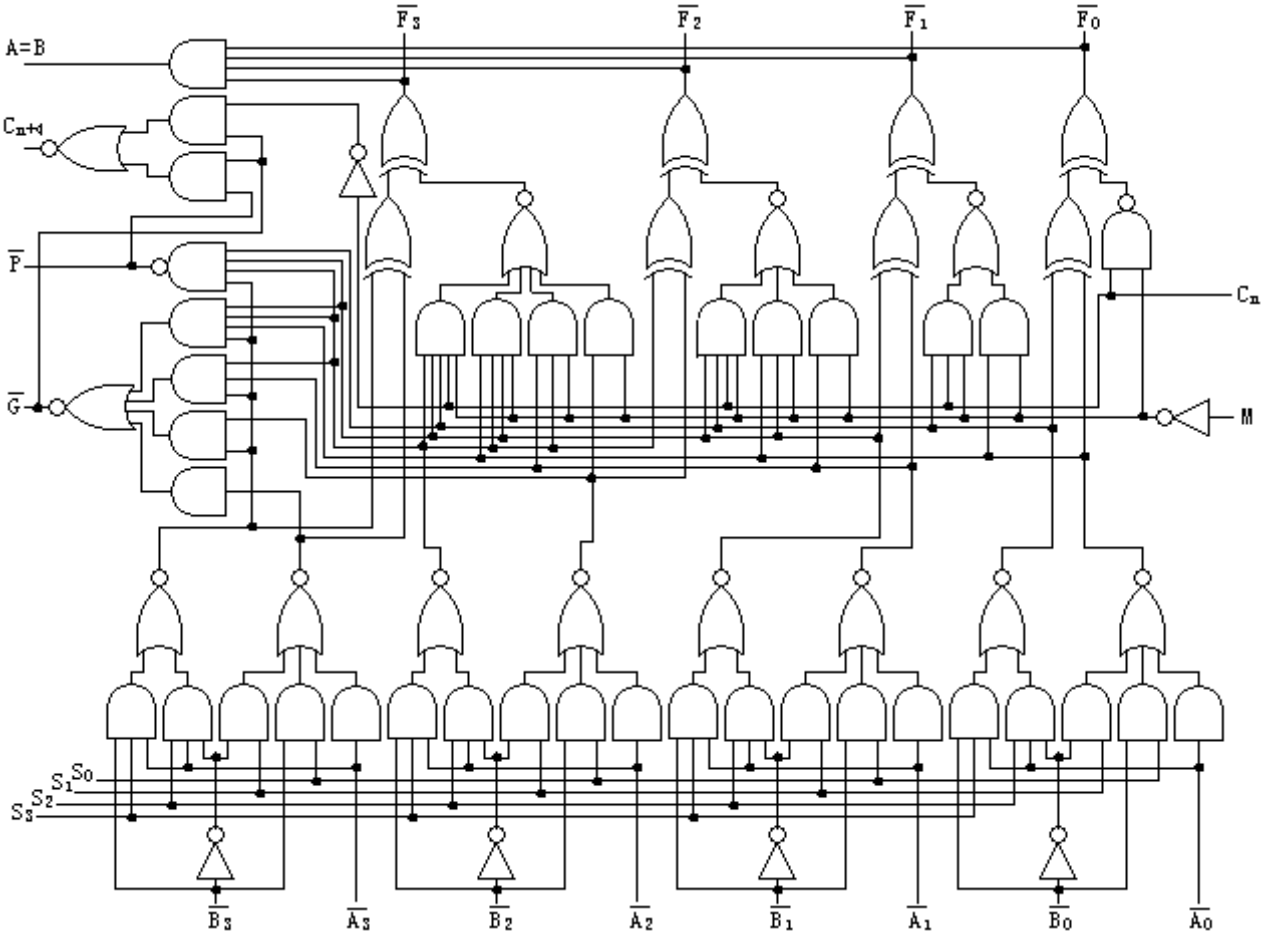


图 2.24　负逻辑操作数表示的 74181ALU 逻辑电路图

表 2.14　SN74181ALU 的两种工作方式

工作方式	负逻辑输入与输出	正逻辑输入与输出
操作数	输入反变量、输出反变量	输入原变量、输出原变量
输入	$\overline{A_0}\sim\overline{A_3}$ 、 $\overline{B_0}\sim\overline{B_3}$ 、 C_n	$A_0\sim A_3$ 、 $B_0\sim B_3$ 、 $\overline{C_n}$
输出	$\overline{F_0}\sim\overline{F_3}$ 、 C_{n+4} $\overline{G}$ 、 $\overline{P}$ 、A=B	$F_0\sim F_3$ 、 $\overline{C_{n+4}}$ G、P、A=B
控制信号	M、S_0、S_1、S_2、S_3	M、S_0、S_1、S_2、S_3

表 2.15　SN74181ALU 算术/逻辑运算功能表

控制参数 \ 工作方式				负逻辑输入与输出		正逻辑输入与输出	
S_3	S_2	S_1	S_0	逻辑运算 M=1	算术运算 M=0　C_n=0	逻辑运算 M=1	算术运算 M=0　C_n=1
0	0	0	0	F= $\overline{A}$	F=A 减 1	F= $\overline{A}$	F=A
0	0	0	1	F= $\overline{AB}$	F=AB 减 1	F= $\overline{A+B}$	F= $A+B$

0	0	1	0	F=$\overline{A}+B$	F=$A\overline{B}$ 减 1	F=$\overline{A}B$	F=$A+\overline{B}$
0	0	1	1	F=逻辑 1	F=减 1	F=逻辑 0	F=减 1
0	1	0	0	F=$\overline{A+B}$	F=A 加($A+\overline{B}$)	F=$\overline{AB}$	F=A 加 $A\overline{B}$
0	1	0	1	F=$\overline{B}$	F=AB 加($A+\overline{B}$)	F=$\overline{B}$	F=(A+B)加 $A\overline{B}$
0	1	1	0	F=$\overline{A\oplus B}$	F=A 减 B 减 1	F=$A\oplus B$	F=A 减 B 减 1
0	1	1	1	F=$A+\overline{B}$	F=$A+\overline{B}$	F=$A\overline{B}$	F=$A\overline{B}$ 减 1
1	0	0	0	F=$\overline{A}B$	F=A 加(A+B)	F=$\overline{A}+B$	F=A 加 AB
1	0	0	1	F=$A\oplus B$	F=A 加 B	F=$\overline{A\oplus B}$	F=A 加 B
1	0	1	0	F=B	F=$A\overline{B}$ 加(A+B)	F=B	F=($A+\overline{B}$)加 AB
1	0	1	1	F=$A+B$	F=$A+B$	F=AB	F=AB 减 1
1	1	0	0	F=逻辑 0	F=A 加 A	F=逻辑 1	F=A 加 A
1	1	0	1	F=$A\overline{B}$	F=AB 加 A	F=$A+\overline{B}$	F=(A+B)加 A
1	1	1	0	F=AB	F=$A\overline{B}$ 加 A	F=$A+B$	F=($A+\overline{B}$)加 A
1	1	1	1	F=A	F=A	F=A	F=A 减 1

在表 2.15 中，算术运算时操作数是用补码表示的。其中“加”是指算术加，运算时要考虑进位，而符号“+”是指逻辑加。在进行算术运算时，对于负逻辑输入与输出的 ALU，C_n=0 表示两个数进行运算时最低位没有进位输入，C_n=1 则表示两个数进行运算时最低位有进位输入，即两个数进行某种算术运算，并在末位加 1；对于正逻辑的 ALU 则刚好相反，C_n=1 表示两个数进行运算时最低位没有进位输入，C_n=0 则表示两个数进行运算时最低位有进位输入，即两个数进行某种算术运算，并在末位加 1。对于正逻辑输入与输出的 ALU，若要实现 A 减 B 功能，除了 M=0 和 $S_3S_2S_1S_0$=0110 外，C_n 的值必须为 0。

3. SN74182 芯片

前面说过，SN74181 设置了 P 和 G 两个本组先行进位输出，如果将 4 片 SN74181 的输出 P 和 G 送入先行进位部件（CLA）SN74182，可实现组与组之间的先行进位。

假设 4 片（组）SN74181 的先行进位输出依次为 P_0、G_0、P_1、G_1、P_2、G_2、P_3、G_3，每片（组）的进位输出依次为 C_{n+4}、C_{n+8}、C_{n+12}、C_{n+16}，由式(2.48)C_{n+4}的表达式

$$C_{n+4}=G_0+P_0C_n$$
$$=\overline{\overline{P_0}\,\overline{G_0}+\overline{G_0}\,\overline{C_n}}$$

可推导出

$$C_{n+8}=G_1+P_1C_{n+4}=G_1+G_0P_1+P_0P_1C_n$$
$$=\overline{\overline{P_1}\,\overline{G_1}+\overline{P_0}\,\overline{G_0}\,\overline{G_1}+\overline{G_0}\,\overline{G_1}\,\overline{C_n}}$$
$$C_{n+12}=G_2+P_2C_{n+8}=G_2+G_1P_2+G_0P_1P_2+P_0P_1P_2C_n$$
$$=\overline{\overline{P_2}\,\overline{G_2}+\overline{P_1}\,\overline{G_1}\,\overline{G_2}+\overline{P_0}\,\overline{G_0}\,\overline{G_1}\,\overline{G_2}+\overline{G_0}\,\overline{G_1}\,\overline{G_2}\,\overline{C_n}} \qquad (2.49)$$
$$C_{n+16}=G_3+P_3C_{n+12}=G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3+P_0P_1P_2P_3C_n$$

设 $G^*=G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3$

$P^*=P_0P_1P_2P_3$

则 $C_{n+16}=G^*+P^*C_n$

$$\overline{G^*}=\overline{G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3}=\overline{\overline{P_3}\,\overline{G_3}+\overline{P_2}\,\overline{G_2}\,\overline{G_3}+\overline{P_1}\,\overline{G_1}\,\overline{G_2}\,\overline{G_3}+\overline{G_0}\,\overline{G_1}\,\overline{G_2}\,\overline{G_3}}$$
$$\overline{P^*}=\overline{P_0P_1P_2P_3}=\overline{P_0}+\overline{P_1}+\overline{P_2}+\overline{P_3} \qquad (2.50)$$

根据式(2.49)和式(2.50)设计出来的 SN74182 逻辑电路如图 2.25 所示。其中 G^*称为成组进位发生输出，P^*称为成组进位传送输出。

4. 利用 SN74181 芯片构成 32 位 ALU

SN74181 的结构很适合将它们连接成不同位数的 ALU，每片 SN74181 芯片作为一个 4 位的小组，由于芯片提供了进位信号 C_{n+4}、先行进位信号 P 和 G，所以用该芯片既可构成组间串行进位的 ALU，也可以构成组间并行进位的 ALU。

1）组间串行进位的 32 位 ALU 的构成

若组间采用串行进位方式，则只需将 8 片 SN74181 芯片作简单的连接，就可以获得一个 32 位组内并行、组间串行进位的 ALU，其逻辑方框图如图 2.26 所示。

2）组间并行进位的 32 位 ALU 的构成

若组间采用并行进位方式，除了需要 8 片 SN74181 芯片外，还需增加 2 片 SN74182 芯片。前面已介绍过，SN74182 是一个产生并行进位信号的部件，它是与 SN74181 配套的产品，每片 SN74182 与 4 片 SN74181 芯片连接，图 2.27 示出了组间并行进位的 32 位 ALU 逻辑方框图。很显然，图 2.27 利用 SN74182 和 SN74181 部件构成了两级先行进位逻辑，实现组（芯片）与组（芯片）之间的先行进位，从而使运算时间比起组间串行进位的 32 位 ALU 来说大大缩短。

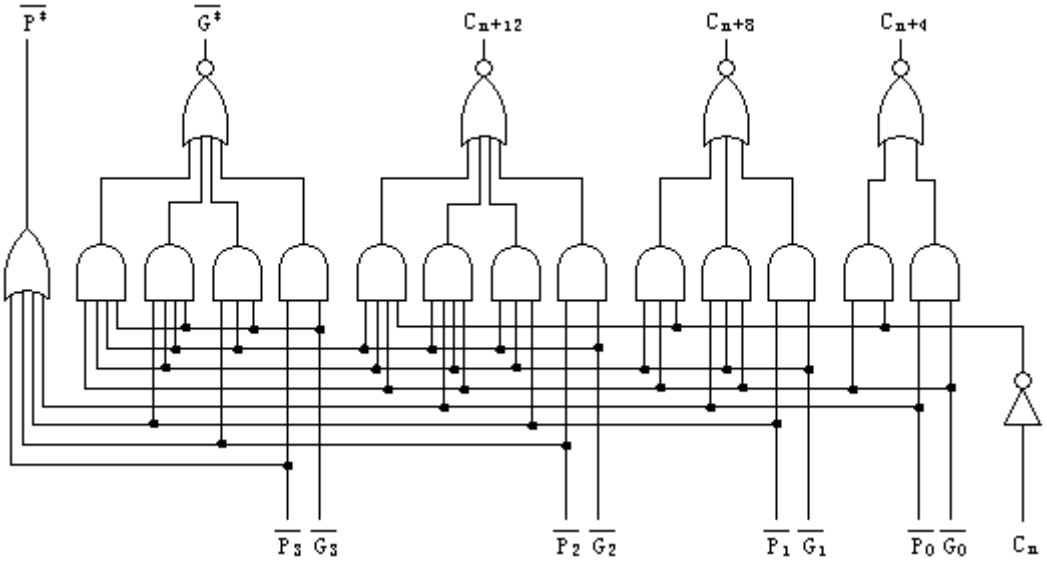


图 2.25　成组先行进位部件 SN74182 的逻辑电路图

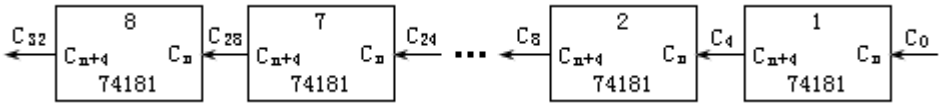


图 2.26　组间串行进位的 32 位 ALU 逻辑方框图

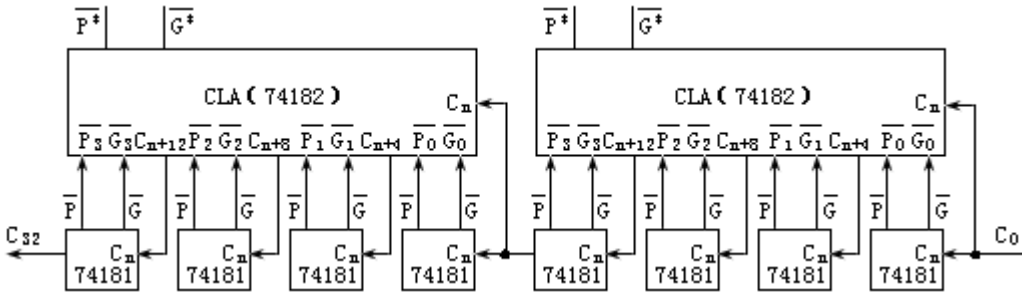


图 2.27　组间并行进位的 32 位 ALU 逻辑方框图

2.7.2　定点运算器

1. 内部总线

数字计算机的硬件是由若干个系统功能部件构成的，这些系统功能部件在 CPU 内控制器的控制下统一协调地工作，保证信息在各功能部件之间正确地传递，从而完成程序实现的功能。总线是指一个或多个信息源传递信息到一个或多个目的地的数据通路，它是多个部件之间传送信息的一组传输线。

在单处理机系统中，按总线相对于 CPU 的位置可分为内部总线和外部总线两种。内部总线是指在CPU内部，各寄存器之间和算术逻辑部件ALU与控制部件之间传输数据所用的总线，

它是 CPU 的内部数据通路。外部总线是指 CPU 与内存、输入和输出设备接口之间进行通讯的通路。本节仅讨论内部总线。

按总线逻辑结构的不同，总线可分为单向传送总线和双向传送总线。单向传送总线是指总线上的信息只能向一个方向传送，而双向传送总线是指总线上的信息可以向两个方向传送。

图 2.28(a)是由三态门构成的带有缓冲器的 8 位双向数据总线。当接收端为高电平且发送端为低电平时，数据从右往左发送。当发送端为高电平且接收端为低电平时，数据从左往右发送。由于使用三态门可以控制数据的传送方向，并且具有信号的驱动放大作用，因此这种类型的缓冲器常作为数据缓冲器或总线驱动器使用。

图 2.28(b)是由触发器和三态门构成的带有锁存器的 8 位双向数据总线。当发送端为低电平、接收端为高电平且时钟控制端为上边沿，即触发器的使能端 E 为高电平且触发器的时钟 CLK 为上边沿时，接收数据总线上的数据且将其保存在由 8 个触发器构成的锁存器中。当接收端为低电平，发送端为高电平时，锁存器中的数据发送至数据总线上。

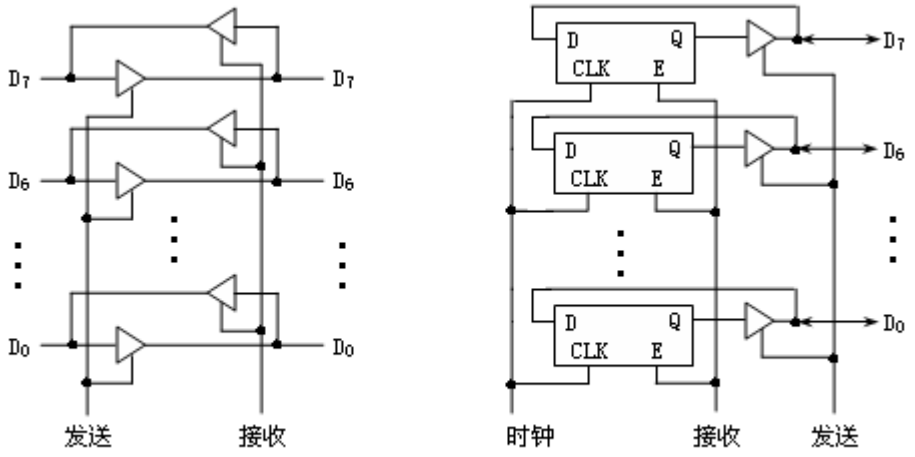


图 2.28　由三态门组成的 8 位双向数据总线

2. 定点运算器的基本结构

计算机中定点运算器的基本结构包括算术逻辑运算单元 ALU、通用寄存器、多路开关/锁存器、三态缓冲器和数据总线等逻辑部件。功能比较强大的定点运算器还设计有阵列乘除部件和其它专用部件等。

计算机的运算器根据内部数据总线条数的不同，可分为如下三种结构形式：

(1)单总线结构的运算器

单总线结构的运算器如图 2.29(a)所示，所有部件均连接到同一条数据总线上，数据可以在任意两个寄存器之间、寄存器与 ALU 之间传送。这种单总线结构的运算器在同一时间内，只能有一个操作数放在总线上。若要将两个操作数输入到 ALU 进行运算，则需分两次先将两个操作数分别暂存于缓冲器 A 和缓冲器 B 中，只有当两个操作数同时出现在 ALU 的两个输入端时，ALU 才能执行相应的运算。运算结束后，其结果通过单总线传送至某个目的寄存器，此时，总线上不能传送其它数据。由于只需对单总线上传送的数据进行控制，并且每一时刻只允许一个数据出现在总线上，因此这种结构的优点是控制简单，缺点是操作速度较慢。

(2)双总线结构的运算器

双总线结构的运算器如图 2.29(b)所示，参与 ALU 运算的两个操作数分别由总线 1 和总线 2 提供，故这两个操作数可以同时送到 ALU 进行运算，只需一次操作控制，立即可得到运算结果。但 ALU 并不能马上将运算结果送到总线上，这是因为 ALU 是组合逻辑部件，形成运算结果输出时，两条总线均被输入数据占据。为此，在 ALU 的输出端设置有缓冲寄存器，在运算产生结果时，先将运算结果暂存于数据缓冲器中。当参与运算的输入数据从总线 1 和总线 2 上消失后，再将缓冲器中暂存的运算结果通过总线 1 或总线 2 传送至某个目的寄存器，此时，总线上不能传送其它数据。由于参与运算的数据通过总线 1 和总线 2 同时提供，因此

这种结构的优点是具有数据并行传送的能力，加快了数据的传输速度，提高了机器性能。但这种结构需对两条总线进行控制，并需保证各数据传输之间的同步，因此这种结构的缺点是操作控制复杂。

(3)三总线结构的运算器

三总线结构的运算器如图 2.29(c)所示，参与 ALU 运算的两个操作数分别由总线 1 和总线 2 提供，而 ALU 的运算结果直接通过总线 3 送至目的寄存器，这样，运算操作可在一步控制之内完成。由于 ALU 本身有时间延迟，因此将运算结果打入目的寄存器的选通脉冲必须考虑这个延迟。如果一个操作数不需经过任何运算直接从总线 2 传送至总线 3，则可直接通过总线旁路器完成，而不必经过 ALU。由于参与运算的数据和运算的结果都分别通过不同的总线进行传送，并且数据传送和运算操作可在一步控制之内完成（要考虑 ALU 本身的时间延迟），因此这种结构的优点是操作速度快，缺点是操作控制更加复杂。

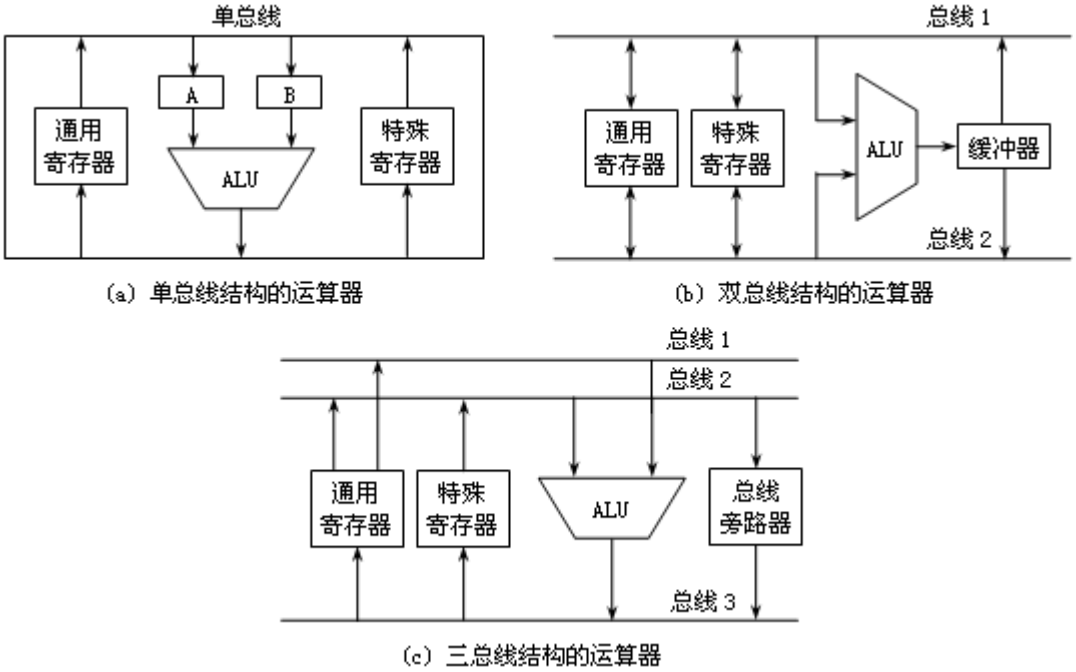


图 2.29　运算器的三种基本结构形式

3. 定点运算器举例

Intel 8086 运算器结构框图如图 2.30 所示。从图 2.30 可以看出，这是一种典型的采用单总线结构的运算器，Intel 8086 字长为 16 位，运算器内部包含一个 16 位的 ALU，其输入端通过暂存器与内部总线相连。参加运算的数据可来自于运算器内部的通用寄存器，运算结果直接通过内部总线被送往某个通用寄存器，运算结果的状态被保存到程序状态字（PSW）寄存器。通用寄存器组中包含 4 个 16 位的通用寄存器（AX、BX、CX、DX），它们也可当作 8 个 8 位的通用寄存器（AH、AL、BH、BL、CH、CL、DH、DL）使用，用来存放参与运算的数据或保存运算的结果。4 个 16 位的专用寄存器分别为堆栈指针（SP）、基址指针（BP）、源变址（SI）寄存器和目标变址（DI）寄存器，它们均直接与内部总线相连，运算器内部各部件之间通过内部总线相互传送信息。

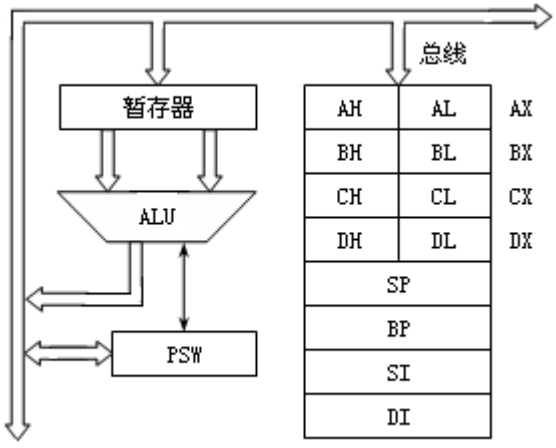


图 2.30　Intel 8086 运算器结构框图

2.8　浮点运算方法和浮点运算器

浮点数比定点数的表示范围大，运算精度高，更适合于科学与工程计算的需要。当要求计算精度较高时，往往采用浮点运算。但浮点数的格式较定点数格式复杂，硬件实现的成本相应高一些，完成一次浮点运算所需的时间也比定点运算要长。在早期一些微机的 CPU 中没有浮点运算功能，但另有配套的浮点协处理器，以提高浮点运算的速度。现代计算机的 CPU 中都设有浮点寄存器和浮点运算功能部件，相应地，指令系统中包含有浮点运算指令。

在 2.1 节中我们介绍了浮点数的表示格式，在浮点数的表示格式中阶码用定点整数表示，尾数用定点小数表示。因此浮点数的运算实质上包含两组定点运算，即阶码运算与尾数运算，但这两部分有各自的作用与相互间的关联。

2.8.1　浮点加法、减法运算

设有两个浮点数 x 和 y，它们分别为

$$x=2^{E_x}\times M_x \qquad y=2^{E_y}\times M_y$$

其中，E_x 和 E_y 分别表示 x 和 y 的阶码，M_x 和 M_y 分别表示 x 和 y 的尾数。则两个浮点数进行加减运算的规则是

$$x\pm y=2^{E_y}(M_x\times 2^{E_x-E_y}\pm M_y) \qquad \text{其中 } E_x\leqslant E_y \tag{2.51}$$

由式(2.51)可以看出，两个浮点数进行加减法运算时，先必须让两个数的阶码相同，然后再对尾数进行加减法运算。完成浮点加减法运算的操作过程大体上可分为五步：第一步，0 操作数检查；第二步，比较阶码大小并完成对阶；第三步，尾数进行加法或减法运算；第四步，结果规格化并进行舍入处理；第五步，判断溢出。

假设浮点数的格式如图 2.2 所示，阶码和尾数均用补码表示，在浮点加减运算时，为便于浮点数尾数的规格化处理和浮点数的溢出判断，阶码和尾数均采用双符号位表示。

①0 操作数检查

用来判断两个操作数 x 和 y 中是否有一个数为 0，若有，马上就可得出运算结果而没有必要再进行后续的一系列操作，以节省运算时间。

②比较阶码大小并完成对阶

两个浮点数进行加减运算时，首先要使两个数的阶码相同，即小数点的位置对齐。若两个数的阶码相同，表示小数点的位置是对齐的，就可以对尾数进行加减运算。反之，若两个数的阶码不相同，表示小数点的位置没有对齐，此时必须使两个数的阶码相同，这个过程称为对阶。

要对阶，首先应求出两个浮点数的阶码之差，即

$$\Delta E=[E_x]_{补}-[E_y]_{补}=[E_x]_{补}+[-E_y]_{补}$$

若 $\Delta E=0$，表示两个浮点数的阶码相等，即 $[E_x]_{补}=[E_y]_{补}$；若 $\Delta E>0$，表示 $[E_x]_{补}>[E_y]_{补}$；若 $\Delta E<0$，表示 $[E_x]_{补}<[E_y]_{补}$。

当 $\Delta E\neq 0$ 时，要通过浮点数尾数的算术左移或算术右移来改变阶码，使两个浮点数的阶码相等。理论上讲，既可以通过移位 $[M_x]_{补}$ 以改变 $[E_x]_{补}$ 来达到 $[E_x]_{补}=[E_y]_{补}$，也可以通过移位 $[M_y]_{补}$ 以改变 $[E_y]_{补}$ 来达到 $[E_x]_{补}=[E_y]_{补}$。但是，由于浮点数的尾数在算术左移的过程会改变尾数的符号位，同时，尾数在算术左移的过程中还会使尾数的高位数据丢失，造成运算结果错误。因此，在对阶时规定使小阶向大阶看齐，通过小阶的尾数算术右移以改变阶码来达到 $[E_x]_{补}=[E_y]_{补}$，尾数每右移一位，阶码加 1，其数值保持不变，直到两个浮点数的阶码相等，右移的次数等于 ΔE 的绝对值。

③尾数进行加法或减法运算

对阶结束后，即可对浮点数的尾数进行加法或减法运算。不论是加法运算还是减法运算，都按加法进行操作，其方法与定点加减运算完全一样。

④结果规格化并进行舍入处理

在 2.1 节曾介绍了规格化浮点数的定义，当尾数用二进制补码表示时，规格化浮点数的尾数形式为 00.1××…××或 11.0××…××。若浮点数的尾数不是这两种形式，则称之为非规格化浮点数，需进行浮点数的规格化。

若浮点数的尾数形式为 00.0××…××或 11.1××…××，应利用向左规格化使其变为规格化浮点数，尾数每算术左移 1 位，阶码减 1，直到浮点数的尾数变成规格化形式。

若浮点数的尾数形式为 01.××…××或 10.××…××，表示尾数求和的结果发生溢出，应利用向右规格化使其变为规格化浮点数，尾数算术右移 1 位，阶码加 1，此时浮点数的尾数就变成了规格化形式。

在对阶或向右规格化时，尾数都要进行算术右移操作，为了保证运算结果的精度，运算过程中需保留右移中移出的若干位数据，称为保护位。在运算结果进行规格化后再按照某种规则进行舍入处理以去除这些数据。舍入处理就是消除保护位数据并按照某种规则调整剩下的部分，舍入处理总要影响到数据的精度。舍入规则应当有舍有入，选择舍入方法时要考虑方法的简单性，使得舍入处理的速度比较快。舍入处理的方法一般有以下三种：

第一种是截去法，无条件地将正常尾数最低位之后的全部数据截去。其最大误差接近于正常尾数最低位上的 1，其好处是处理简单，缺点是有舍无入，具有误差积累，影响运算结果的精度。

第二种是“末位恒置 1”法，在截去正常尾数最低位之后的全部数据时，将正常尾数的最低位置 1。其最大误差是正常尾数最低位上的-1 到 1 之间。尽管误差范围扩大了，但正误差可以和负误差抵消，从统计角度，平均误差为 0。因此最后运算结果的准确性提高了。

第三种是“0 舍 1 入”法，它是常用的舍入方式，若被截去数据的最高位的值为 0 时，则直接截去正常尾数最低位之后的全部数据；若被截去的最高位的值为 1 时，则在正常尾数最低数值位上加 1 进行修正。其最大误差是正常尾数最低位上的-1/2 到 1/2 之间，正误差可以和负误差抵消，是一种比较理想的方法，但实现起来比较复杂，因为它需要做加法运算，速度比较慢。

⑤判断溢出

浮点数尾数的溢出可通过规格化进行处理，而浮点数运算结果的溢出则根据运算结果中浮点数的阶码来确定。若阶码未发生溢出，则表示运算结果未发生溢出；若阶码溢出，则需进行溢出处理。图 2.31 给出了浮点数运算结果出现溢出的四种情况，若浮点数为正数，当阶码发生正溢时称为正上溢，当阶码为负溢时，称为正下溢；若浮点数为负数，当阶码发生正溢时称为负上溢，当阶码为负溢时，称为负下溢。溢出处理方法为：当浮点数发生下溢时，置运算结果为 0；当浮点数发生上溢时，置溢出标志。

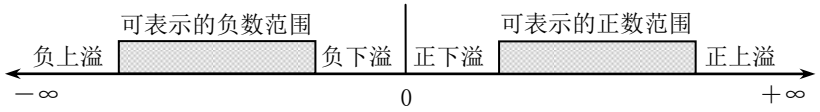


图 2.31　浮点数运算结果出现溢出的四种情况

若阶码用双符号位补码表示，判断溢出的方法为：若阶码的双符号位相同，表示结果未发生溢出；若阶码的双符号位不相同，表示结果发生溢出。

[**例 2.52**]　设两浮点数 $x=2^{001}\times(0.1101)$，$y=2^{011}\times(-0.1010)$，在浮点数的表示格式中阶码占 3 位，尾数占 4 位（都不包括符号位）。阶码和尾数均采用含双符号位的补码表示，运算结果的尾数取单字长（含符号位共 5 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x+y、x-y。

解：$[x]_{浮}$=00001，00.1101　　$[y]_{浮}$=00011，11.0110

①对阶，小阶向大阶对齐

$\Delta E=[E_x]_{补}-[E_y]_{补}=[E_x]_{补}+[-E_y]_{补}$=00001+11101=11110

x 的尾数$[M_x]_{补}$右移 2 位，阶码$[E_x]_{补}$加 2

$[x]_{浮}$=00011，00.0011(01)

其中(01)表示$[M_x]_{补}$右移 2 位后移出的最低两位数。

②尾数进行加法、减法运算

$$\begin{array}{rr} & 00.0011(01) \\ + & 11.0110 \\ \hline & 11.1001(01) \end{array} \qquad \begin{array}{rr} & 00.0011(01) \\ + & 00.1010 \\ \hline & 00.1101(01) \end{array}$$

即 $[x+y]_{浮}$=00011，11.1001(01)

$[x-y]_{浮}$=00011，00.1101(01)

③结果规格化并进行舍入处理

和的尾数左移 1 位，阶码减 1，采用“0 舍 1 入”法进行舍入处理后，得

$[x+y]_{浮}$=00010，11.0011

差已为规格化浮点数，采用“0 舍 1 入”法进行舍入处理后，得

$[x-y]_{浮}$=00011，00.1101

④判断溢出

和、差的阶码的双符号位均相同，故和、差均无溢出。

所以 $x+y=2^{010}\times(-0.1101)$

$x-y=2^{011}\times(0.1101)$

2.8.2 浮点乘法、除法运算

设有两个浮点数 x 和 y，它们分别为

$x=2^{E_x}\times M_x$ $y=2^{E_y}\times M_y$

其中，E_x和E_y分别表示 x 和 y 的阶码，M_x和M_y分别表示 x 和 y 的尾数。则两个浮点数进行乘法运算的规则是

$$x\times y=2^{(E_x+E_y)}(M_x\times M_y) \tag{2.52}$$

由式(2.52)可以看出，两个浮点数进行乘法运算产生的乘积的尾数等于两个数尾数之积，乘积的阶码等于两个数阶码之和。完成浮点乘法运算的操作过程大体上可分为五步：第一步，0 操作数检查；第二步，阶码相加；第三步，尾数相乘；第四步，结果规格化并进行舍入处理；第五步，判断溢出。

两个浮点数进行除法运算的规则是

$$x\div y=2^{(E_x-E_y)}(M_x\div M_y) \tag{2.53}$$

由式(2.53)可以看出，两个浮点数进行除法运算产生的商的尾数等于两个数尾数相除的商，商的阶码等于两个数阶码之差。完成浮点除法运算的操作过程大体上可分为五步：第一步，0 操作数检查；第二步，阶码相减；第三步，尾数相除；第四步，结果规格化并进行舍入处理；第五步，判断溢出。

①0 操作数检查

在浮点数的乘法运算时，若被乘数或乘数中有一个数为 0，则乘积为 0。在浮点数除法运算时，若被除数为 0，则商为 0；若除数为 0，则产生溢出中断。

②阶码相加减

按照定点整数的加减法运算方法对两个浮点数的阶码进行加减运算，若阶码用补码表示，则使用式(2.22)和式(2.24)，即：

$[E_x+E_y]_{补}=[E_x]_{补}+[E_y]_{补}$ $(\mathrm{mod}\ 2^{n+2})$

$[E_x-E_y]_{补}=[E_x]_{补}+[-E_y]_{补}$ $(\mathrm{mod}\ 2^{n+2})$

若阶码用移码表示，则使用式(2.31)和式(2.32)，即：

$[E_x+E_y]_{移}=[E_x]_{移}+[E_y]_{补}$ $(\mathrm{mod}\ 2^{n+2})$

$[E_x-E_y]_{移}=[E_x]_{移}+[-E_y]_{补}$ $(\mathrm{mod}\ 2^{n+2})$

③尾数相乘或相除

按照定点小数的乘除法运算方法对两个浮点数的尾数进行乘除运算。为了保证尾数相除时商的正确性，必须保证被除数尾数的绝对值一定小于除数尾数的绝对值。若被除数尾数的绝对值大于除数尾数的绝对值，需对被除数进行调整，即被除数的尾数每右移 1 位，阶码加 1，直到被除数尾数的绝对值小于除数尾数的绝对值。

④结果规格化并进行舍入处理

浮点数乘除运算结果的规格化和舍入处理与浮点数加减运算结果的规格化和舍入处理方法相同。并且在浮点数乘除运算的结果中，由于乘积和商的绝对值一定小于 1，因此在浮点乘除运算结果进行规格化处理时只存在向左规格化，不可能出现向右规格化。

⑤判断溢出

浮点数乘除运算结果的尾数不可能发生溢出，而浮点数运算结果的溢出则根据运算结果中浮点数的阶码来确定，溢出的判定和处理方法与浮点加减运算完全相同。

[**例 2.53**] 设两浮点数 $x=2^{-001}\times(-0.100010)$，$y=2^{-100}\times(0.010110)$，在浮点数的表示格式中阶码占 3 位，尾数占 6 位（都不包括符号位），阶码采用双符号位的补码表示，尾数用单符号位的补码表示。要求用直接补码阵列乘法完成尾数乘法运算，运算结果的尾数取单字长（含符号位共 7 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x×y。

解： $[x]_{浮}=11111,\ 1.011110$　　$[y]_{浮}=11100,\ 0.010110$

①阶码相加

$$[E_x+E_y]_{补}=[E_x]_{补}+[E_y]_{补}=11111+11100=11011$$

②尾数作直接补码阵列乘法运算

```
                                  (1) 0  1  1  1  1  0
                            ×     (0) 0  1  0  1  1  0
                           ------------------------------
                                  (0) 0  0  0  0  0  0  0
                               (1) 0  1  1  1  1  0
                            (1) 0  1  1  1  1  0
                         (0) 0  0  0  0  0  0
                      (1) 0  1  1  1  1  0
                   (0) 0  0  0  0  0  0
             +     0 (0)(0)(0)(0)(0)(0)
          ---------------------------------------------------
            (1) 1  1  1  0  1  0  0  0  1  0  1  0  0
```

$$[M_x]_{补}\times[M_y]_{补}=1.110100010100$$

③结果规格化并进行舍入处理

积的尾数左移 2 位，阶码减 2，采用“0 舍 1 入”法进行舍入处理后，得

$$[x\times y]_{浮}=11001,\ 1.010001$$

④判断溢出

乘积的阶码的双符号位相同，故乘积无溢出。

所以　$x\times y=2^{-111}\times(-0.101111)$

[**例 2.54**] 设两浮点数 $x=2^{-010}\times(0.011010)$，$y=2^{011}\times(-0.111100)$，在浮点数的表示格式中阶码占 3 位，尾数占 6 位（都不包括符号位），阶码采用双符号位的补码表示，尾数用单符号位的原码表示。要求用原码阵列除法完成尾数除法运算，运算结果的尾数取单字长（含符号位共 7 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x÷y。

解： $[x]_{浮}=11110,\ 0.011010$　　$[y]_{浮}=00011,\ 1.111100$

①阶码相减

$$[E_x-E_y]_{补}=[E_x]_{补}+[-E_y]_{补}=11110+11101=11011$$

②尾数作原码阵列除法运算

$[M_x]_{原}=0.011010$　　$[M_y]_{原}=1.111100$

商的符号位为：$M_{xf}\oplus M_{yf}=0\oplus 1=1$

令 $M_x'=011010000000$，$M_y'=111100$，其中 M_x'和 M_y'分别为$[M_x]_{原}$和$[M_y]_{原}$的数值部分，且 M_x'为双字长

$[M_x']_{补}=0011010000000$，$[M_y']_{补}=0111100$，$[-M_y']_{补}=1000100$

```
                        被除数/余数                    商
                          0011010000000
  +[-My']补               1000100
  ------------------------------------------------------
                          1011110000000                0
```

$+[M_y']_{补}\longrightarrow$	0111100	
	111100000000	0
$+[M_y']_{补}\longrightarrow$	0111100	
	01011000000	1
$+[-M_y']_{补}\longrightarrow$	1000100	
	0011100000	1
$+[-M_y']_{补}\longrightarrow$	1000100	
	111110000	0
$+[M_y']_{补}\longrightarrow$	0111100	
	01101000	1
$+[-M_y']_{补}\longrightarrow$	1000100	
	0101100	1

故得　商 q=0011011

所以　$[M_x \div M_y]_{原}$=1.011011

因此　$[x \div y]_{浮}$=11011，1.011011

③尾数规格化

商的尾数左移 1 位，阶码减 1。

$[x \div y]_{浮}$=11010，1.110110

④判断溢出

商的阶码的双符号位相同，故商无溢出。

所以　$x \div y=2^{-110} \times (-0.110110)$

2.8.3　浮点运算器举例

目前在微机系统中往往配置有专门的浮点运算部件，可直接用浮点运算指令对浮点数进行算术运算，其运算速度比采用软件子程序实现时要快得多。例如，x86 系列机中的 80×87 就是浮点运算器，对于早期的 386SX 及以下 CPU，80×87 是任选芯片，而对于 486DX 及以上 CPU，已将浮点运算器设计到了 CPU 芯片内部，并逐步采用多级流水线技术来完成浮点运算，使得浮点运算速度得到了很大的提高。80×87 之所以被称为协处理器，是因为它只能协助主 CPU 工作，不能单独工作。这里以 80387 浮点运算器为例，主要介绍其指令执行过程、数据类型和内部结构，简单了解浮点运算器的组成和工作原理。

1. 80387 的指令执行过程

80387 浮点运算器相当于 80386 CPU 的一个 I/O 部件，它有 80 多条指令，按功能可分为浮点加、减、乘、除、对数和指数运算、三角函数以及传送、中断、处理控制等。80387 的指令系统称为 ESC 指令。

80387 的指令是 80386 指令的扩充，在编程时可直接使用这些指令。编制好的程序被放在主存中，当程序执行时，全部指令都由 80386 逐条读取。如果取回的是 80386 指令，则在 80386 内部处理；如果取回的是 ESC 指令，则通过 I/O 口地址传送给 80387。这时 80386 和 80387 可以独立地并行对指令进行加工。但是，80387 在执行指令时，80386 不能再向 80387 传送新的 80387 指令，这就要求 80386 要与 80387 之间进行同步。为此，80386 使用了 BUSY 引脚，80387 在执行 ESC 指令期间向 80386 的 BUSY 引脚发出低电平信号。当 80386 取到 ESC 指令时，首先要对 BUSY 引脚上的信号进行检查。如果该信号为低电平，则 80386 暂停向 80387 传送指令，等待它变为高电平后，才开始发送操作。

80387 访问存储器中的数据，也是由 80386 生成地址并进行读/写，再通过 I/O 口对 80387 进行数据的输入或输出操作。为了在 80386 和 80387 之间进行数据的输入或输出操作，80386 将 800000F8H～800000FFH 的 I/O 口地址分配给 80387。

2. 80387 的数据类型

80387 浮点运算器可处理包括二进制整数、二进制浮点数和压缩十进制数串三大类共 7 种不同的数据类型，这些数据类型的表示格式如图 2.32 所示。对整数来说，最高位为符号位，用补码表示，有 16、32 和 64 位三种格式。压缩十进制数串是用特殊形式表示的整数，

其最高位为符号位，最低 72 位可表示 18 位十进制数。对浮点数来说，最高位为符号位，符号位为 0 表示正数，符号位为 1 表示负数。浮点数有 32、64 和 80 位三种格式，三种浮点数阶码的基数均为 2，阶码用移码表示，尾数用原码表示。

3. 80387 的内部结构

80387 的内部结构如图 2.33 所示，它由总线控制逻辑部件、数据接口与控制部件、浮点运算部件三个主要功能模块组成。在 80387 的浮点运算部件中，分别设置有阶码（指数）运算部件和尾数运算部件，并设有加速移位操作的桶形移位器。它们通过指数总线和尾数总线与 8 个 80 位的堆栈寄存器相连。这些寄存器按后进先出的方式工作，此时栈顶寄存器被用作累加器，也可以按寄存器的编号直接访问任何一个寄存器。

80387 从主存取数和向主存写数时，均用 80 位的临时浮点数和其它 6 种数据类型执行自动转换。全部数据在 80387 中均以 80 位临时浮点数的形式表示。

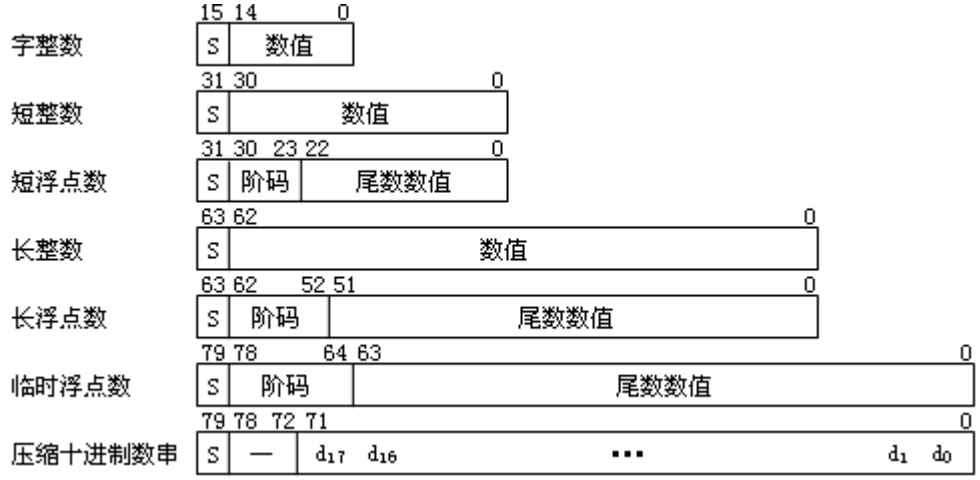


图 2.32　80387 各种数据类型的表示格式

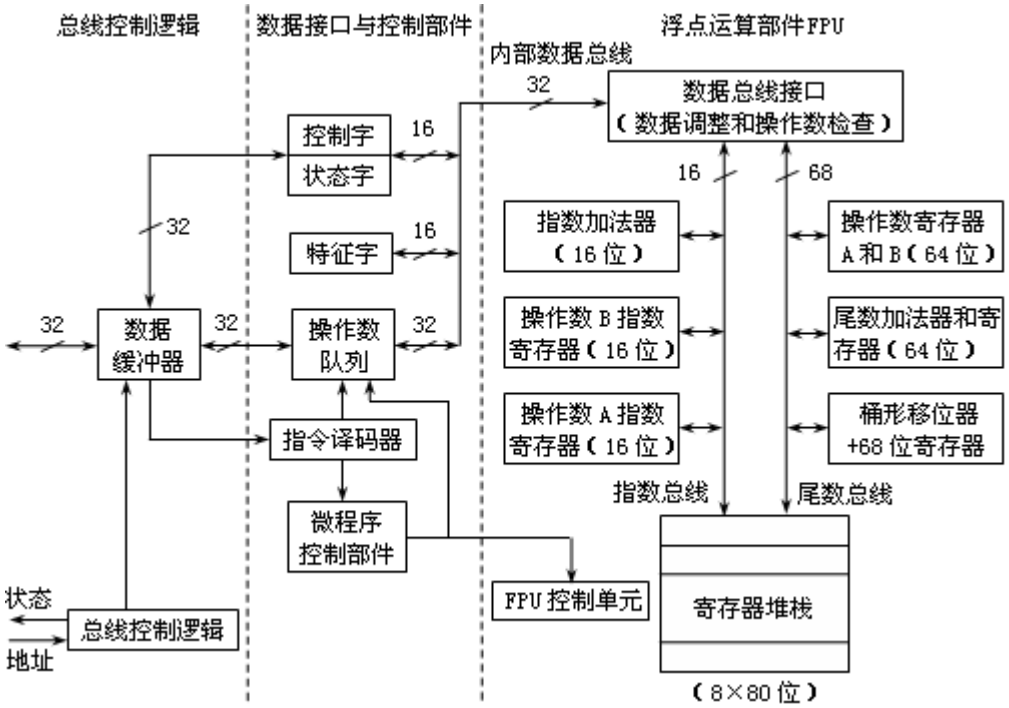


图 2.33　80387 的内部结构框图

2.9　数据校验码

数据校验码是一种常用的带有发现某些错误或自动纠错能力的数据编码方法。它的实现原理是加进一些冗余码，使合法数据编码在出现某些错误时就变为非法编码，这样就可以通过检测编码的合法性来达到发现错误的目的。合理地安排非法编码数量和编码规则，就可以提高发现错误的能力，或达到自动纠正错误的目的。这里用到一个码距的概念，码距是根据任意两个合法码之间至少有几个二进制位不相同而确定的，若任意两个合法码之间仅有一位不同，称其码距为1。例如，用4个二进制位表示16种状态，则16种编码都用到了，此时码距为1，也就是说任何一个状态的4位码中的1位或几位出错，就变成另一个合法码，此时无查错能力。若用4位二进制表示8个状态，就可以只用其中的8种编码，而把另外8种编码作为非法编码，合理地安排合法数据编码，可使得任意两个合法码之间有2个二进制位不相同，此时码距为2。

当码距d为奇数时，如用来检错，可发现d-1位错，如用来纠错，可纠正(d-1)/2位错；当码距d为偶数时，如用来检错，可发现d/2位错，如用来纠错，可纠正d/2-1位错。一般来说，合理地增大码距，就能提高发现错误的能力，但编码所使用的二进制位数变多，会增加数据存储的容量或数据传送的数量。在确定与使用数据校验码的时候，通常要考虑在不过多增加硬件开销的情况下，尽可能发现或纠正更多的错误。常用的数据校验码有奇偶校验码、海明校验码和循环冗余码。

2.9.1 奇偶校验码

奇偶校验码是一种开销最小，能发现数据代码中一位出错情况的编码，常用于存储器读写检查，或ASCII字符传送过程中的检查。它的实现原理是，在每组代码中增加一个冗余位，使码距由1增加到2。如果合法编码中有奇数个位发生了错误，这个编码就将成为非法编码。增加的冗余位称为奇偶校验位。实现的具体方法是，使每个编码（包括校验位）中1的个数为奇数或偶数，前者称为奇校验，后者称为偶校验。

设有效数据代码为$D_7D_6D_5D_4D_3D_2D_1D_0$，校验位为P，则

奇校验定义为：

$$P=\overline{D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0} \tag{2.54}$$

偶校验定义为：

$$P=D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0 \tag{2.55}$$

一个编码（包括校验位）从发送端到达接收端后，对接收到的实际编码按如下方法计算：

奇校验：

$$P'=\overline{D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0\oplus P} \tag{2.56}$$

偶校验：

$$P'=D_7\oplus D_6\oplus D_5\oplus D_4\oplus D_3\oplus D_2\oplus D_1\oplus D_0\oplus P \tag{2.57}$$

若$P'=0$，则表示没有错误，若$P'=1$，则表示有错误。

奇偶校验码只能发现一位错或奇数个位错，但不能确定是哪一位错，因此无纠错能力，也不能发现偶数个位错。考虑到一位出错的概率比多位同时出错的概率要大得多，因此奇偶校验码还是有很高的实用价值。

[**例2.55**] 求下列数据代码的奇校验编码和偶校验编码（设校验位放在最低位）。

(1)11001110　(2)10101010　(3)00000000　(4)11111111

解：(1)奇校验编码为：110011100；偶校验编码为：110011101。

(2)奇校验编码为：101010101；偶校验编码为：101010100。

(3)奇校验编码为：000000001；偶校验编码为：000000000。

(4)奇校验编码为：111111111；偶校验编码为：111111110。

2.9.2 海明校验码

奇偶校验码不能发现两位错，它只能发现一位错，并且不知道是哪位出错，也就是说，它无法纠正错误。对一组数据使用多重奇偶校验，便有可能指出是哪一位出错。以下描述的海明校验码实际上就是使用了多重奇偶校验的检错纠错方法，海明校验码是由贝尔实验室的Richard Hamming于1950年提出的，目前还被广泛采用，主存的检错与纠错采用的就是这

类校验码。其实现原理是，在数据中加入几个校验位，并把数据的每一个二进制位分配在几个奇偶校验组中。当某一位出错后，就会引起有关的几个校验组的值发生变化，这不但可以发现出错，还能指出是哪一位出错，为自动纠错提供了依据。

假设校验位的个数为 r，则它能表示 2^r 个信息，用其中一个信息指出“没有错误”，其余的 2^r-1 个信息指出错误发生在哪一位。然而错误也可能发生在校验位，因此只有 $k=2^r-1-r$ 个信息能用于纠正被传送数据的位数，也就是说要满足如下关系：

$$2^r \geqslant k+r+1 \qquad (2.58)$$

如要能检测与自动校正一位错，并发现两位错，此时校验位的位数 r 和数据位的位数 k 应满足如下关系：

$$2^{r-1} \geqslant k+r \qquad (2.59)$$

按式(2.59)，可计算出数据位 k 与校验位 r 的对应关系，如表 2.16 所示。

表 2.16　数据位 k 与校验位 r 的对应关系

k 值	最小的 r 值
1～4	4
5～11	5
12～26	6
27～57	7
58～120	8

若海明码的最高位号为 m，最低位号为 1，即 $H_mH_{m-1}\cdots H_2H_1$，则此海明码的编码规律通常是：

（1）校验位与数据位的位数之和为 m，每个校验位 P_i 在海明码中被分在位号为 2^{i-1} 的位置，其余各位为数据位，并按从低向高逐位依次排列的关系分配各数据位。

（2）海明码的每一位码 H_i（包括数据位和校验位本身）由多个校验位校验，其关系是被校验的每一位的位号要等于校验它的各校验位的位号之和。这样安排的目的，是希望校验的结果能正确反映出出错位的位号。

下面，我们来按上述规律讨论一个字节的海明码。

每个字节由 8 个二进制位组成，此处的 k 为 8，按式(2.59)求出校验位的位数 r 应为 5，故海明码的总位数为 13，可表示为：

$H_{13}H_{12}H_{11}H_{10}H_9H_8H_7H_6H_5H_4H_3H_2H_1$

5 个校验位 P_5～P_1 对应的海明码位号应分别为 H_{13}、H_8、H_4、H_2 和 H_1。P_5 只能放在 H_{13} 这一位上，因为它已经是海明码的最高位了，其它 4 位满足 P_i 的位号等于 2^{i-1} 的关系。其余为数据位 D_i，则有如下排列关系：

$P_5D_8D_7D_6D_5P_4D_4D_3D_2P_3D_1P_2P_1$

按前面讲的，每个海明码的位号，要等于参与校验它的几个校验位的位号之和的关系，可以给出如表 2.17 所示的结果。

表 2.17　出错的海明码位号和校验位位号的关系

海明码位号	数据位/校验位	参与校验的校验位位号	被校验位的海明码位号=校验位位号之和
H_1	P_1	1	1=1
H_2	P_2	2	2=2
H_3	D_1	1，2	3=1+2
H_4	P_3	4	4=4
H_5	D_2	1，4	5=1+4
H_6	D_3	2，4	6=2+4
H_7	D_4	1，2，4	7=1+2+4
H_8	P_4	8	8=8
H_9	D_5	1，8	9=1+8
H_{10}	D_6	2，8	10=2+8
H_{11}	D_7	1，2，8	11=1+2+8
H_{12}	D_8	4，8	12=4+8
H_{13}	P_5	13	13=13

表 2.17 给出了每一位海明码和参与对其进行校验的有关校验位的对应关系，即 5 个校验位各自与本身有关，数据位 D_1 由校验位 P_1 和 P_2 校验，查表 2.17，D_1 的海明码位号为 3，而 P_1 和 P_2 的海明码位号分别为 1 和 2，满足 3=1+2 的关系。又如数据位 D_2（H_5）由校验位 P_1（H_1）和 P_3（H_4）校验，数据位 D_7（H_{11}）由校验位 P_1（H_1）、P_2（H_2）和 P_4（H_8）三个校验位校验等。

从表 2.17 中，可以进一步找出 4 个校验位各自与哪些数据位有关。如 P_1 参与数据位 D_1、D_2、D_4、D_5 和 D_7 的校验，P_2 参与 D_1、D_3、D_4、D_6 和 D_7 的校验等。由此关系，就可以进一步求出由各有关数据位形成 P_i 值的偶校验的结果。

$$P_1=D_1\oplus D_2\oplus D_4\oplus D_5\oplus D_7 \tag{2.60}$$

$$P_2=D_1\oplus D_3\oplus D_4\oplus D_6\oplus D_7 \tag{2.61}$$

$$P_3=D_2\oplus D_3\oplus D_4\oplus D_8 \tag{2.62}$$

$$P_4=D_5\oplus D_6\oplus D_7\oplus D_8 \tag{2.63}$$

如果要分清是两位出错还是一位出错，还要补充一个总校验位 P_5，使

$$P_5=D_1\oplus D_2\oplus D_3\oplus D_4\oplus D_5\oplus D_6\oplus D_7\oplus D_8\oplus P_4\oplus P_3\oplus P_2\oplus P_1 \tag{2.64}$$

在这种安排中，每一位数据位，都至少地出现在 3 个 P_i 值的形成关系中。当任一位数据码发生变化时，必将引起 3 个或 4 个 P_i 值随之变化，该海明码的码距为 4。

按如下关系对所得到的海明码实现偶校验：

$$S_1=P_1\oplus D_1\oplus D_2\oplus D_4\oplus D_5\oplus D_7 \tag{2.65}$$

$$S_2=P_2\oplus D_1\oplus D_3\oplus D_4\oplus D_6\oplus D_7 \tag{2.66}$$

$$S_3=P_3\oplus D_2\oplus D_3\oplus D_4\oplus D_8 \tag{2.67}$$

$$S_4=P_4\oplus D_5\oplus D_6\oplus D_7\oplus D_8 \tag{2.68}$$

$$S_5=P_5\oplus P_4\oplus P_3\oplus P_2\oplus P_1\oplus D_1\oplus D_2\oplus D_3\oplus D_4\oplus D_5\oplus D_6\oplus D_7\oplus D_8 \tag{2.69}$$

则校验得到的结果值 $S_5S_4S_3S_2S_1$ 能反映 13 位海明码的出错情况。

海明码有如下关系成立：

（1）当所有位均没有错时，S_5 为 0 且 $S_4S_3S_2S_1$ 为全 0。反过来不成立，因为任何偶数个海明码位出错时，S_5 一定为 0，此时 $S_4S_3S_2S_1$ 为全 0 也有可能。如 H_3（D_1）、H_6（D_3）、H_9（D_5）、H_{12}（D_8）这 4 位同时出错时，S_5 为 0 且 $S_4S_3S_2S_1$ 为全 0。但是，若校验结果是 S_5 为 0 且 $S_4S_3S_2S_1$ 为全 0，我们可以认为所有位均是正确的，即数据位 $D_8D_7D_6D_5D_4D_3D_2D_1=H_{12}H_{11}H_{10}H_9H_7H_6H_5H_3$。因为满足 S_5 为 0 且 $S_4S_3S_2S_1$ 为全 0 对应所有位均正确的概率比起其它情况的概率要大得多。若 S_5 为 0 且 $S_4S_3S_2S_1$ 不为全 0，则可以肯定有偶数个海明码位出错，一般认为是有两位错，因为两位错比起其它情况的概率要大得多。

（2）当出现一位错时，S_5 为 1，$S_4S_3S_2S_1$ 不全为 0，$S_4S_3S_2S_1$ 的二进制取值对应的十进制数值就是海明码出错的位号，直接将出错位的数值取反即可纠正错误。这是因为，若 H_{12}（D_8）位出错，由式(2.65)～式(2.69)可知，必然使 $S_4S_3S_2S_1$=1100。同理若 H_{11}（D_7）位出错，则必然使 $S_4S_3S_2S_1$=1011，依此类推。反过来不成立，这是因为 S_5 为 1 并不意味着只有一位错，实际上，当有奇数位出错时，S_5 都为 1。但是，若 S_5 为 1，我们可以认为只有一位出错，因为一位出错的概率比起其它情况的概率要大得多。

综上所述，海明码具有检测两位错和纠正一位错的能力。

例如，有效数据为 12 的海明码在传送时，第 11 位发生了错误。

将数据 12 转换成 8 位二进制得 $D_8D_7D_6D_5D_4D_3D_2D_1$=00001100，由式(2.60)～式(2.64)，可以计算出 $P_5P_4P_3P_2P_1$=00001，因此，

$P_5D_8D_7D_6D_5P_4D_4D_3D_2P_3D_1P_2P_1$

发送的海明码为：0 0 0 0 0 0 0 1 1 0 0 0 0 1

接收的海明码为：0 0 1 0 0 0 0 1 1 0 0 0 0 1

由式(2.65)～式(2.69)，可以计算出 $S_5S_4S_3S_2S_1$=11011。S_5=1，表示一位出错，$S_4S_3S_2S_1$=1011，表示海明码的第 11 位（D_7）发生了错误。只要将出错位的数值取反即可纠正错误。

[**例 2.56**] 采用海明校验码，一个 8 位的数据字为 00111001，与它一起存储的校验位应该是 0111。假定由存储器读出时计算出的校验位是 1101，那么由存储器读出的数据字是

什么？

解：由于由存储器读出时计算出的校验位与存储的校验位不相同，则可以肯定出现了传送或存储错误。根据海明码校验得到的结果值 $S_4S_3S_2S_1$ 的公式，得：

$$\begin{array}{rl} 0111 & \text{存储的校验位 } P_4P_3P_2P_1 \\ \oplus\ 1101 & \text{读出时计算出的校验位 } P_4P_3P_2P_1 \\ \hline 1010 & \text{海明码校验得到的结果值 } S_4S_3S_2S_1 \end{array}$$

由 $S_4S_3S_2S_1$=1010 可知，海明码的 H_{10} 位有错，按海明码的如下排列关系：

$H_{13}H_{12}H_{11}H_{10}H_9H_8H_7H_6H_5H_4H_3H_2H_1$

$P_5\ D_8\ D_7\ D_6\ D_5P_4D_4D_3D_2P_3D_1P_2P_1$

可知 D_6 有错，因此，由存储器读出的数据字是 00011001。

2.9.3 循环冗余校验码

在串行数据传送中，经常会遇到瞬间干扰而导致同时多位数据出错。循环冗余校验（CRC，cycle redundancy check）码可以发现并纠正信息存储或传送过程中连续出现的多位错误，因此在计算机网络、同步通信以及磁表面存储器中得到了广泛应用。

1. 代码格式

CRC 码由两部分组成，如图 2.34 所示。左边为信息码，右边为校验码。若 CRC 码的字长为 n 位，信息码占 k 位，则校验码占 n-k 位。故该校验码又称作（n，k）码。

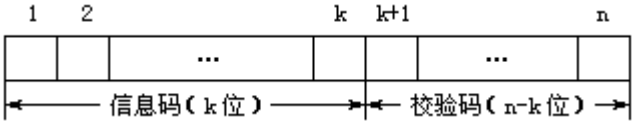


图 2.34　CRC 码的格式

校验码是由信息码产生的，校验码位数越长，该代码的校验能力就越强。

2. 校验码的生成方法

先将 k 位信息码表达成多项式 M(x)：

$$M(x)=C_{k-1}x^{k-1}+C_{k-2}x^{k-2}+\cdots+C_ix^i+\cdots+C_1x^1+C_0$$

式中 C_i 为 0 或 1，　$1\leqslant i\leqslant k-1$。

校验码的生成方法可按如下步骤进行：

①将信息码多项式 M(x)乘以 x^{n-k}，得到 $M(x)\cdot x^{n-k}$。其中，x 为基数，用二进制表示时，x=2。该步也就是信息码左移 n-k 位，以便拼接 n-k 位校验码。

②给出生成多项式 G(x)。注意，生成多项式的最高次幂就是校验码的位数，最低次幂必须为 0。

③用第①步所得的多项式 $M(x)\cdot x^{n-k}$ 除以生成多项式 G(x)。注意，该除法运算是模 2 运算，即进行除法时，每一步的减运算是按位减，不发生借位。所得的余数表达为 R(x)，商为 Q(x)。余数 R(x)所对应的二进制编码就是校验码。将 n-k 位的校验码拼接在 k 位的信息码之后，就构成了这个有效信息的 CRC 码。

[**例 2.57**]　有一个（7，4）码，设生成多项式为 x^3+x+1，令信息码为 1100，请求出其校验码和 CRC 码。

解：①$M(x)\cdot x^{n-k}=(x^3+x^2)\cdot x^{7-4}=x^6+x^5$。

②$G(x)=x^3+x+1$。

③用 $M(x)\cdot x^{n-k}$ 除以 G(x)，运算过程如下：

$$\begin{array}{r} x^3+x^2+x \\ x^3+x+1\ \overline{)\ x^6+x^5\ \ \ \ \ \ } \\ x^6\ \ \ +x^4+x^3 \\ \hline x^5+x^4+x^3 \\ x^5\ \ \ +x^3+x^2 \\ \hline x^4\ \ \ +x^2 \\ x^4\ \ \ +x^2+x \\ \hline x \end{array}$$

余数 x 就是校验码的多项式，对应的校验码为 010，CRC 码为 1100010。

例 2.57 也可以直接用二进制求解，步骤如下：

①把信息码 1100 左移 7-4=3 位得 1100000。

②写出生成多项式所对应的二进制数，即 1011。

③用 1100000 除以 1011，运算过程如下：

$$
\begin{array}{r}
1110 \\
1011\ \sqrt{1100000} \\
\underline{1011} \\
1110 \\
\underline{1011} \\
1010 \\
\underline{1011} \\
10
\end{array}
$$

故该代码的校验码为 010，CRC 码为 1100010。

以上算法既可用软件方法实现，也可以通过硬件实现。在使用硬件实现时，信息码多项式 M(x)乘以 x^{n-k} 是通过移位寄存器的移位来实现的，而除以生成多项式 G(x)是靠除法电路实现的。目前 CRC 校验码大都采用硬件实现。

3. CRC 码的校验方法

CRC 码传送到接收方后，接收方就用 CRC 码除以生成多项式 G(x)来校验。如果余数为 0，就说明传送正确。这是因为 CRC 码可用多项式表示为：

$$
\begin{aligned}
M(x)\cdot x^{n-k}+R(x) &= [Q(x)\cdot G(x)+R(x)]+R(x) \\
&= Q(x)\cdot G(x)+[R(x)+R(x)] \\
&= Q(x)\cdot G(x)
\end{aligned}
$$

在 CRC 码传送正确时，CRC 码正好是生成多项式 G(x)的整数倍。在例 2.57 中，信息码 1100 的 CRC 码为 1100010，如果该码传送正确，我们可以用 CRC 码 1100010 除以生成多项式所对应的二进制数 1011，余数确实为 0。

值得注意的是，并非所有 n-k 位的多项式都可以作为生成多项式。为了能够校验出究竟是哪一位出错，生成多项式应具备如下条件：

①当任何一位传送出错时，都能使余数不为 0；

②不同的位传送出错时，应当使余数互不相同；

③对余数继续做模 2 除法，余数是循环的。

只有具备上述三个条件，才能使 CRC 码不仅能检测传送错误，而且还能判定是哪位发生错误。通过例 2.57 求出其出错模式如表 2.18 所示。更换不同的信息码可以证明，余数与出错位的对应关系是不变的，只与码制和生成多项式有关。因此，表 2.18 给出的关系可作为（7，4）码的判别依据。对于其它码制或其它生成多项式，出错模式将发生变化。

表 2.18　（7，4）循环码的出错模式（生成多项式 $G(x)=x^3+x+1$）

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	余数			出错位
正确	1	1	0	0	0	1	0	0	0	0	无
错误	1	1	0	0	0	1	1	0	0	1	7
	1	1	0	0	0	0	0	0	1	0	6
	1	1	0	0	1	1	0	1	0	0	5
	1	1	0	1	0	1	0	0	1	1	4
	1	1	1	0	0	1	0	1	1	0	3
	1	0	0	0	0	1	0	1	1	1	2
	0	1	0	0	0	1	0	1	0	1	1

4. CRC 码的特点

在例 2.57 中，当生成多项式 G(x)为 x^3+x+1 的情况下，信息码 0000～1111 对应的 CRC 码如表 2.19 所示。

表 2.19　信息码 0000～1111 对应的 CRC 码

信息码	校验码	CRC 码

0000	000	0000000
0001	011	0001011
0010	110	0010110
0011	101	0011101
0100	111	0100111
0101	100	0101100
0110	001	0110001
0111	010	0111010
1000	101	1000101
1001	110	1001110
1010	011	1010011
1011	000	1011000
1100	010	1100010
1101	001	1101001
1110	100	1110100
1111	111	1111111

由表 2.19 可以看出，对任何两个 CRC 码进行按位异或运算，所得结果仍是一个 CRC 码。

本章小结

运算器是 CPU 的重要组成部分，主要用来进行数据的加工处理，完成各种算术运算和逻辑运算。尽管各种计算机的运算器在设计上有较大的区别，但它们最基本的结构中必须有算术逻辑运算单元(ALU)、通用寄存器、多路开关/锁存器、三态缓冲器和数据总线等逻辑构件，运算器的核心是算术逻辑运算单元，其主要功能是进行算术运算和逻辑运算。

计算机中的数据表示格式分为两种，即定点格式和浮点格式。一般来说，定点格式所表示的数的范围有限，但运算复杂度和相应的处理硬件都比较简单，而浮点格式所表示的数的范围很大，但运算复杂度和相应的处理硬件都比较复杂。

用定点格式表示的数称为定点数，定点数由符号位和尾数两部分组成。按小数点位置的不同，定点数有定点纯小数和定点纯整数两种表示方法。用浮点格式表示的数称为浮点数，浮点数一般由阶码和尾数两部分组成，其中阶码又包括阶符和阶码值两部分，尾数又包括数符和尾数值两部分。在 IEEE754 标准中，一个浮点数由符号位 S、阶码 E 和尾数 M 三个域组成。浮点数的表示格式中，阶码的位数决定了浮点数的表示范围，尾数的位数决定了浮点数的表示精度。为了使浮点数的表示具有惟一性，通常采用浮点数规格化形式。

若整个机器字长的全部二进制位均表示数值位，即没有符号位的数称为无符号数。有符号数在机器中采用机器码表示。常见的机器码有原码、反码、补码和移码四种表示方法，其中移码只能表示定点整数。对于真值零，其补码表示是惟一的；对于整数零，其移码表示也是惟一的。

字符信息属于符号数据，是非数值数据，在国际上普遍采用 ASCII 码来表示字符。字符串是指连续的一串字符，通常方式下，它们占用主存中连续的多个字节，每个字节存放一个字符的 ASCII 码。

汉字处理过程包括汉字的输入、输出和汉字在计算机内部的表示。汉字的输入采用输入码的形式来完成，输入码按编码方法的不同可分为三类，即数字编码、拼音编码和字形编码。汉字在机内的表示由汉字内码来实现，一般采用两个字节表示。汉字的输出则是通过采用点阵表示的汉字字模码来完成。汉字的字模码存储在汉字库中，在机器中建立汉字库有软字库和硬字库两种方法。

对于一个简单的运算器，运算方法中算术运算通常采用补码加减法、原码一位乘除法或补码一位乘除法。为了提高运算器的运算速度，采用了先行进位、阵列乘法器、阵列除法器、流水线等并行处理技术。除此之外，运算器还具有逻辑运算和逻辑移位等功能。为了提高浮点运算的速度，一般在 CPU 内部或外部配有浮点运算部件，主要用来完成浮点数的加减乘除运算，但浮点数的运算过程比定点数的运算过程要复杂得多。数的机器码表示和运算方法是

本章的重点。

总线是指一个或多个信息源传递信息到一个或多个目的地的数据通路，它是多个部件之间传送信息的一组传输线。在单处理机系统中，按总线相对于 CPU 位置的不同，可分为内部总线和外部总线。内部总线指 CPU 内部各寄存器之间、寄存器与 ALU 之间进行连接的总线，它是 CPU 的内部数据通路。

按总线逻辑结构的不同，总线可分为单向传送总线和双向传送总线。按运算器内部数据总线条数的不同，可分为单总线结构的运算器、双总线结构的运算器和三总线结构的运算器等三种结构形式。

数据校验码是一种常用的带有发现某些错误或自动纠错能力的数据编码方法。常用的数据校验码有奇偶校验码、海明校验码和循环冗余码。奇偶校验码只能发现一位错，无纠错能力。海明校验码能发现两位错，并能纠正一位错。循环冗余码则可以发现多位错和纠正多位错。

习题 2

1. 写出下列各数的原码、反码、补码、移码（用 8 位二进制表示），其中 MSB 是最高位（符号位），LSB 是最低位。如果是小数，则小数点在 MSB 之后；如果是整数，则小数点在 LSB 之后。

(1)-59/64　　(2)27/128　　(3)-127/128　　(4)用小数表示-1

(5)用整数表示-1　(6)-127　　(7)35　　(8)-128

2. 设$[x]_{补}=x_0.x_1x_2x_3x_4$，其中 x_i 取 0 或 1，若要使 x＞－0.5，则 x_0、x_1、x_2、x_3、x_4 的取值应满足什么条件？

3. 若 32 位定点小数的最高位为符号位，用补码表示，则所能表示的最大正数为_____，最小正数为_____，最大负数为_____，最小负数为_____；若 32 位定点整数的最高位为符号位，用原码表示，则所能表示的最大正数为_____，最小正数为_____，最大负数为_____，最小负数为_____。

4. 若机器字长为 32 位，在浮点数据表示时阶符占 1 位，阶码值占 7 位，数符占 1 位，尾数值占 23 位，阶码用移码表示，尾数用原码表示，则该浮点数格式所能表示的最大正数为_____，最小正数为_____，最大负数为_____，最小负数为_____。

5. 某机浮点数字长为 18 位，格式如图 2.35 所示，已知阶码（含阶符）用补码表示，尾数（含数符）用原码表示。

(1)将$(-1027)_{10}$表示成规格化浮点数；

(2)浮点数$(0EF43)_{16}$是否是规格化浮点数？它所表示的真值是多少？

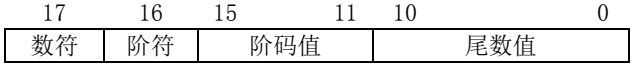


图 2.35　浮点数的表示格式

6. 有一个字长为 32 位的浮点数，格式如图 2.36 所示，已知数符占 1 位；阶码占 8 位，用移码表示；尾数值占 23 位，尾数用补码表示。

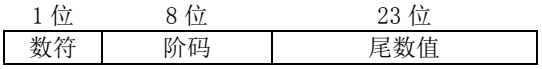


图 2.36　浮点数的表示格式

请写出：

(1)所能表示的最大正数；

(2)所能表示的最小负数；

(3)规格化数所能表示的数的范围。

7. 若浮点数 x 的 IEEE754 标准的 32 位存储格式为$(8FEFC000)_{16}$，求其浮点数的十进制数值。

8. 将数$(-7.28125)_{10}$转换成 IEEE754 标准的 32 位浮点数的二进制存储格式。

9. 已知 $x=-0.x_1x_2\cdots x_n$，求证：$[x]_{补}=1.\overline{x_1}\,\overline{x_2}\cdots\overline{x_n}+0.00\cdots01$。

10. 已知$[x]_{补}=1.x_1x_2x_3x_4x_5x_6$，求证：$[x]_{原}=1.\overline{x_1x_2x_3x_4x_5x_6}+0.000001$。

11. 已知 x 和 y，用变形补码计算 x+y，同时指出运算结果是否发生溢出。

(1)x=0.11011　　y=-0.10101

(2)x=-10110　　y=-00011

12. 已知 x 和 y，用变形补码计算 x-y，同时指出运算结果是否发生溢出。

(1)x=0.10111　　y=0.11011

(2)x=11011　　y=-10011

13. 已知$[x]_{补}=1.1011000$，$[y]_{补}=1.0100110$，用变形补码计算 $2[x]_{补}+1/2[y]_{补}=?$，同时指出结果是否发生溢出。

14. 已知 x 和 y，用原码运算规则计算 x+y，同时指出运算结果是否发生溢出。

(1)x=0.1011，y=-0.1110

(2)x=-1101，y=-1010

15. 已知 x 和 y，用原码运算规则计算 x-y，同时指出运算结果是否发生溢出。

(1)x=0.1101，y=0.0001

(2)x=0011，y=1110

16. 已知 x 和 y，用移码运算方法计算 x+y，同时指出运算结果是否发生溢出。

(1)x=-1001，y=1101

(2)x=1101，y=1011

17. 已知 x 和 y，用移码运算方法计算 x-y，同时指出运算结果是否发生溢出。

(1)x=1011，y=-0010

(2)x=-1101，y=-1010

18. 余 3 码编码的十进制加法规则如下：两个一位十进制数的余 3 码相加，如结果无进位，则从和数中减去 3（加上 1101）；如结果有进位，则和数中加上 3（加上 0011），即得和数的余 3 码。试设计余 3 码编码的十进制加法器单元电路。

19. 已知 x 和 y，分别用原码一位乘法和补码一位乘法计算 x×y。

(1)x=0.10111　　y=-0.10011

(2)x=-11011　　y=-11111

20. 已知 x 和 y，分别用带求补器的原码阵列乘法器、带求补器的补码阵列乘法器和直接补码阵列乘法器计算 x×y。

(1)x=0.10111　　y=-0.10011

(2)x=-11011　　y=-11111

21. 已知 x 和 y，分别用原码加减交替法和补码加减交替法计算 x÷y。

(1)x=0.10011　　y=-0.11011

(2)x=-1000100101　y=-11101

22. 已知 x 和 y，用原码阵列除法器计算 x÷y。

(1)x=0.10011　　y=-0.11011

(2)x=-1000100000　y=-11101

23. 设机器字长为 8 位（含一位符号位），若 x=46，y=-46，分别写出 x、y 的原码、补码和反码表示的机器数在左移一位、左移两位、右移一位和右移两位后的机器数及对应的真值。

24. 某加法器进位链小组信号为 $C_4C_3C_2C_1$，最低位来的进位信号为 C_0，请分别按下述两种方法写出 $C_4C_3C_2C_1$ 的逻辑表达式：

(1)串行进位方式；

(2)并行进位方式。

25. 用 74181 和 74182 设计如下三种方案的 64 位 ALU。

(1)组间串行进位方式；

(2)两级组间并行进位方式；

(3)三级组间并行进位方式。

26. 设浮点数的表示格式中阶码占 3 位，尾数占 6 位（都不包括符号位）。阶码和尾数均采用含双符号位的补码表示，运算结果的尾数取单字长（含符号位共 7 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x+y、x-y。

(1)$x=2^{-011}\times(0.100101)$　　$y=2^{-010}\times(-0.011110)$

(2)$x=2^{-101}\times(-0.010110)$　　$y=2^{-100}\times(0.010110)$

27. 设浮点数的表示格式中阶码占 3 位，尾数占 6 位（都不包括符号位），阶码采用双符号位的补码表示，尾数用单符号位的补码表示。要求用直接补码阵列乘法完成尾数乘法运算，运算结果的尾数取单字长（含符号位共 7 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x×y。

(1)$x=2^{011}\times(0.110100)$　　$y=2^{-100}\times(-0.100100)$

(2)$x=2^{-011}\times(-0.100111)$　　$y=2^{101}\times(-0.101011)$

28. 设浮点数的表示格式中阶码占 3 位，尾数占 6 位（都不包括符号位），阶码采用双符号位的补码表示，尾数用单符号位的原码表示。要求用原码阵列除法完成尾数除法运算，运算结果的尾数取单字长（含符号位共 7 位），舍入规则用“0 舍 1 入”法，用浮点运算方法计算 x÷y。

(1)$x=2^{-010}\times(0.011010)$　　$y=2^{-111}\times(-0.111001)$

(2)$x=2^{011}\times(-0.101110)$　　$y=2^{101}\times(-0.111011)$

29. 定点补码加减法运算中，产生溢出的条件是什么？溢出判断的方法有哪几种？如果是浮点加减运算，产生溢出的条件又是什么？

30. 设有 4 个数：00001111、11110000、00000000、11111111，请问答：

(1)其码距为多少？最多能纠正或发现多少位错？如果出现数据 00011111，应纠正成什么数？当已经知道出错位时如何纠正？

(2)如果再加上 2 个数 00110000，11001111（共 6 个数），其码距是多少？能纠正或发现多少位错？

31. 如果采用偶校验，下述两个数据的校验位的值是什么？

(1)0101010　　(2)0011011

32. 设有 16 个信息位，如果采用海明校验，至少需要设置多少个校验位？应放在哪些位置上？

33. 写出下列 4 位信息码的 CRC 编码，生成多项式为 $G(x)=x^3+x^2+1$。

(1)1000

(2)1111

(3)0001

(4)0000

34. 当从磁盘中读取数据时，已知生成多项式 $G(x)=x^3+x^2+1$，数据的 CRC 码为 1110110，试通过计算判断读出的数据是否正确？

35. 有一个 7 位代码的全部码字为：

a：0000000	b：0001011	c：0010110	d：0011101
e：0100111	f：0101100	g：0110001	h：0111010
i：1000101	j：1001110	k：1010011	l：1011000
m：1100010	n：1101001	o：1110100	p：1111111

(1)求这个代码的码距；

(2)这个代码是不是 CRC 码。