第 3 章 运算方法和运算器

3.1 信息表示

在计算机中广泛使用二进制编码，其主要原因如下。

(1) 物理上易于实现，即容易找到具有两个稳定状态且能方便地控制其状态转换的物理器件，可以用两个状态分别表示基本符号“0”和“1”。

(2) 编码、计数和算术运算规则简单，容易用开关电路实现，为提高计算机的运算速度和降低实现成本奠定了基础。

(3) 基本符号“0”“1”能方便地与逻辑命题的“否”“是”或“假”“真”相对应，为计算机中的逻辑运算和程序的逻辑判断提供了便利条件。

▶ 3.1.1 二进制数值型数据的表示方法

计算机中参与运算的两大类数，一类是无符号数，即没有符号的数，寄存器中的每一位均可用来存放数值。例如，8 位无符号数的取值范围为 0～255(2^8-1)；另一类是有符号数，它将符号数字化，并规定将其放在有效数字的前面，即组成有符号数。

机器字长相同时有符号数和无符号数的表示范围不同。

我们把正、负号加某进制数绝对值的形式称为真值，如二进制真值 $X=+1011$，$Y=-1011$；而将符号数码化的数称为机器数，并约定“0”表示“+”，“1”表示“−”，则前面的两个数对应的机器数分别是 $X=01011$，$Y=11011$。

▶ 3.1.2 十进制数编码

1. 十进制数的编码

计算机中使用 4 位二进制编码表示一个十进制数码，不同表示方式即代表不同码制。

1) BCD(以二进制编码的十进制)码

BCD 码是一种有权代码，其对应的数值 $N=8d_3+4d_2+2d_1+1d_0$。例如，十进制数 63.29 的 BCD 码为 0110 0011 . 0010 1001。

2) 2421 码、5211 码、4311 码等

这些编码均为有权代码，在这些编码中，任何两个相加之和等于 9 的二进制码之间互为反码。其中，2421 码表示的数值 $N=2d_3+4d_2+2d_1+1d_0$。十进制数 63.29 的 2421 码为 1100 0011.0010 1111。

需要注意的是，2421 以及 5211、4311 等有权码的 4 位二进制码之间不符合二进制规则。

3) 余 3 码、格雷码

余 3 码、格雷码均为无权代码。其中，余 3 码对应 8421 码加 0011，格雷码中任何两个相邻的代码有且只有一个二进制位的状态不同，其余三个二进制位必须相同。

表 3.1 给出了十进制数码在不同二进制编码表示下的编码。

表 3.1 十进制数码的各种编码

十进制数	有权代码				无权代码		
	BCD 码	2421 码	5211 码	4311 码	余 3 码	格雷码(1)	格雷码(2)
0	0000	0000	0000	0000	0011	0000	0000
1	0001	0001	0001	0001	0100	0001	0100
2	0010	0010	0011	0011	0101	0011	0110
3	0011	0011	0101	0100	0110	0010	0010
4	0100	0100	0111	1000	0111	0110	1010
5	0101	0101	1000	0111	1000	1110	1011
6	0110	1100	1010	1011	1001	1010	0011
7	0111	1101	1100	1100	1010	1000	0001
8	1000	1110	1110	1110	1011	1100	1001
9	1001	1111	1111	1111	1100	0100	1000

2. 数字串的表示与存储

十进制数字串在计算机内有两种表示与存储方式：字符形式和压缩的十进制数形式。

字符形式：一个字节存放一个十进制数位或符号位，存放的是 0～9 这 10 个数字和正负号的 ASCⅡ编码值。

例如，+123 的编码为 2B 31 32 33，占用 4 个连续的字节（这里 2B、31、32、33 是用十六进制形式给出的编码，其中，2B 表示正号，31、32、33 分别表示数字 1、2 和 3）；－123 在主存中为 2D 31 32 33，其中，2D 表示负号。

字符形式的主要问题是高 4 位不具有数值的意义，运算起来很不方便，主要用在非数值计算的应用领域。

压缩的十进制数形式：用一个字节存放两个十进制数位，既节省了存储空间，又便于完成十进制数的算术运算，其值用 BCD 码或 ASCII 码的低 4 位表示，符号位也占半个字节并放在最低数字位之后，其值可从 4 位二进制码中的 6 种冗余状态中选用。

例如，用 C(12)表示正号，用 D(13)表示负号。并且规定：数字和符号位数之和必须为偶数，否则在最高位数字之前补一个 0，则＋123 被表示成 12 3C(2 字节)，－12 被表示成 01 2D(2 字节)。

▶ 3.1.3 汉字的表示方法

计算机中除了数值型数据，还有非数值型数据，汉字是非数值型数据中的一种。

根据输入、内部处理、输出三种不同用途，有三种不同的汉字编码表示方法，即汉字的输入编码、汉字内码、字模码。

1. 汉字的输入编码

为了能直接使用西文标准键盘把汉字输入计算机，就必须为汉字设计相应的输入编码方法。

汉字输入编码主要指利用键盘输入汉字的方法，其设计的基本原则是：易学易用、操作方便；重码率低；规则简明、记忆量少。当前采用的方法主要有三类：数字编码、拼音码、字型编码。

数字编码常用的是国标区位码，用数字串代表一个汉字输入。

1981 年，国家技术监督局公布了国家标准 GB2312—1980《信息交换用汉字编码字符集-基本集》，它收集了常用汉字 6763 个，包括一级汉字 3755 个、二级汉字 3008 个。为了表示这些汉字，将这 6763 个字分成 94 个区，每个区分成 94 位，相当于把汉字表示成 94×94 的二维数组，用每个汉字在这个数组中的下标（即区、位）来指征这个汉字，因此这种数字编码的数字串汉字输入编码被称为区位码。因为区码（行号，高位）和位码（列号，低位）各两位十进制数字，因此输入一个汉字需按键 4 次。

在 GB2312—1980 中，01～09 区为特殊字符，共 682 个；16～87 区为汉字，其中 16～55 区的 3755 个一级汉字采用拼音字母序，56～87 区的二级汉字采用部首序。需要注意的是，区位码仅收录了常见的 6763 个汉字及 682 个特殊字符，对于这个范围之外的那些生僻字等使用区位码是无法输入的。

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。

事实上，区位码在日常使用中并不普遍，现在常用的一种输入编码是拼音码，它以汉字拼音为基础，使用简单方便，但汉字同音字太多，输入重码率很高，同音字选择影响了输入速度。

另一种常用的输入编码是字形编码，它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，曾经风靡一时的五笔字型输入法就是利用汉字的形状来进行编码的。好的字形编码方式能够极大地减少重码率，因此有利于汉字的快速输入，但是字形码的拆字规则、编码方案需要记忆，有些汉字很难正确拆成部件，一定程度上影响了它的使用范围。

为此，人们把拼音码和字形码结合起来，构成音形码或形音码，以争取取长补短，提高汉字输入的效率，降低记忆量。另外，词组输入、智能联想输入等技术的加持已经使得汉字的键盘输入越来越简便、快捷了。

当然，汉字的输入除了键盘方式，还有语音输入、手写输入、手势输入、文字扫描识别等。

2. 汉字内码

汉字内码是用于汉字信息的存储、交换、检索等操作的机内代码。

英文字符的机内代码是 7 位的 ASCII 码，用一个字节表示，其中最高位为“0”。包括汉字在内的非拉丁字母是对英文字符机内码的扩展，与英文字符一样，在计算机内使用特定格式的编码表示。

例如，GB2312—1980 中共收录常见汉字 6763 个，用 94×94 数组表示，则区号、位号各需用 7 位二进制表示。为了与英文字符能相互区别，汉字机内代码的区号、位号各用 1 字节表示，且两个字节的最高位均规定为“1”（注意：有些系统中字节的最高位用于奇偶校验位，这种情况下用三个字节表示汉字内码。）。

除了 GB2312 规定的编码方式，其他常用的编码方法还有 Unicode、UTF-8/16 等。

3. 汉字字模码

为了便于输出汉字，现代计算机都是使用点阵表示汉字的字形代码，这种字形码又称为汉字的字模码。根据汉字输出的要求不同，点阵的多少也不同。例如，如图 3.1 所示为某种字体下汉字“英”的 16×16 点阵及其编码。

一般地，简易型汉字为 16×16 点阵，提高型汉字为 24×24 点阵、32×32 点阵或更大，因此字模点阵的信息量很大，所占存储空间也很大，因此字模点阵只能用来构成汉字库，而不能

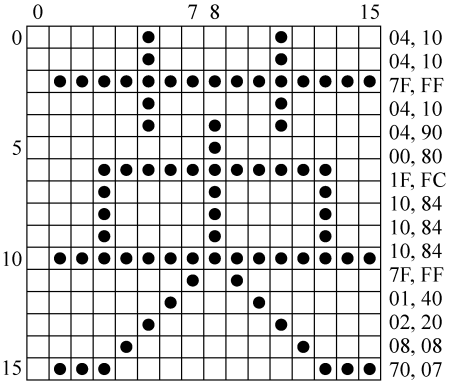


图 3.1　汉字点阵及其编码

用于机内存储。

当显示输出或打印输出时才检索字库，输出字模点阵，得到字形。

汉字的输出编码方式也在不断进步，为了提高字形质量、降低存储要求，后来出现了用矢量或者轮廓曲线表示汉字字形的方式。

3.2　带符号二进制数据表示及定点数加减法运算

▶ 3.2.1　带符号定点二进制数据表示

机器数将符号数值化，隐含规定小数点，有“定点”和“浮点”两种表示方式。

所谓“定点”，就是小数点位置固定，其中，定点小数的小数点固定在所有数值位的最前面，定点整数的小数点固定在所有数值位的最后面。

为了明确区别符号位，人们在讨论时用“,”将整数的符号位和数值部分隔开，而用“.”将小数的符号位和数值部分隔开。同时约定：如无特殊说明，n 表示机器字长，其中，符号位占 1 位，有效数值占 $n-1$ 位。

计算机中有三种常见的机器数表示方式：原码、补码和反码。

1. 原码表示法

原码表示法又称为带符号的绝对值表示。它用“0”表示正号，用“1”表示负号，有效值部分用二进制的绝对值表示，下面是小数原码的定义。

$$[x]_{原}=\begin{cases}x, & 0\leqslant x<1\\ 1-x=1+|x|, & -1<x\leqslant 0\end{cases} \tag{3-1}$$

即$[x]_{原}=$ 符号位$+|x|$。

原码小数的特点(设 $n=8$)如下。

- $[+0]_{原}=0.000\ 0000$；$[-0]_{原}=1.000\ 0000$。
- 最大值：$1-2^{-(n-1)}=1-2^{-7}=127/128$。
- 最小值：$-(1-2^{-(n-1)})=-(1-2^{-7})=-127/128$。
- 表示数的个数：$2^n-1=2^8-1=255$。

下面是整数原码的定义。

$$[x]_{原}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ 2^{n-1}-x=2^{n-1}+|x|, & -2^{n-1}<x\leqslant 0\end{cases} \tag{3-2}$$

即$[x]_{原}$＝符号位//$|x|$，其中，“//”表示拼接。

类似地，原码整数具有如下特点(设$n=8$)。

- $[+0]_{原}=0,000\ 0000$；$[-0]_{原}=1,000\ 0000$。
- 最大值：$2^{(n-1)}-1=2^7-1=127$。
- 最小值：$-(2^{(n-1)}-1)=-(2^7-1)=-127$。
- 表示数的个数：$2^n-1=2^8-1=255$。

原码表示简单，易于同真值之间进行转换，实现乘除运算规则简单；但是原码表示进行加减运算十分麻烦，相加时先判别两数符号，同号相加，异号相减。相减时先比较两数绝对值的大小，再用大绝对值减小绝对值，最后确定运算结果的正负号。

2. 补码表示法

1) 补码的定义

补码表示法规定：正数的补码是其本身，负数的补码是原负数加上模。

模是计量器具的容量，又称为模数。

设机器数字长为n，含1位符号位，则整数的模为2^n。例如，若$n=4$，则表示的二进制整数为0000～1111共16种状态，其模为$16=2^4$。

纯小数的模为2。

下面为小数补码定义：

$$[x]_{补}=\begin{cases}x, & 0\leqslant x<1\\ 2+x=2-|x|, & -1\leqslant x<0 \quad (\mathrm{mod}\ 2)\end{cases} \tag{3-3}$$

即$[x]_{补}=2\cdot$符号位$+x \quad (\mathrm{mod}\ 2)$。

下面为整数补码定义：

$$[x]_{补}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ 2^n+x=2^n-|x|, & -2^{n-1}\leqslant x<0 \quad (\mathrm{mod}\ 2^n)\end{cases} \tag{3-4}$$

根据补码的定义可以轻松实现真值到补码的转换。

2) 补码的表示范围

设机器字长为n，其中，包含1位符号位。

小数补码表示范围为$-1\sim1-2^{-(n-1)}$，共2^n个数。其中需要注意两点：一是补码表示时，$[+0]_{补}=0.0000=[-0]_{补}$；二是，虽然-1本不属于小数范围，却有$[-1]_{补}$的小数形式存在，$[-1]_{补}=10.0000+(-1.0000)=1.0000$，推而广之，补码可以表示定义区间的“下限”。

整数补码表示范围为$-2^{n-1}\sim2^{n-1}-1$，共2^n个数，同样有$[+0]_{补}=0,0000=[-0]_{补}$，且可表示定义区间的“下限”-2^{n-1}。

3) 模4补码

模4补码又称为变形补码、双符号位补码，它是通过将模2补码的模值增加1倍得到的，其定义如下。

$$[x]_{补}=\begin{cases}x, & 0\leqslant x<1\\ 4+x=4-|x|, & -1\leqslant x<0 \quad (\mathrm{mod}\ 4)\end{cases} \tag{3-5}$$

$$[x]_{补}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ 2^{n+1}+x=2^{n+1}-|x|, & -2^{n-1}\leqslant x<0 \quad (\mathrm{mod}\ 2^{n+1})\end{cases} \tag{3-6}$$

例如，00.1010110，11.0101001 等即是采用双符号位补码(模 4 补码)形式表示的数。

模 4 补码具有如下性质。

- 当$-1 \leqslant x < 1$时，$[x]_{补}$的两个符号位相同，00 表示正号，11 表示负号，其数值位与其模 2 补码相同。
- 用模 4 补码能表示-1的值，即$[-1]_{补}=11.00\cdots0$。
- 零有唯一的编码：$[+0]_{补}=[-0]_{补}=00.00\cdots0$。

从模 4 补码的性质发现，与模 2 补码相比，其数据多了 1 位，但是它不仅具有模 2 补码的全部优点，而且更容易判断溢出，因此在溢出判断等场合中有特殊的作用。另外，模 4 补码在移码运算中也有特殊作用。

对比模 4 补码和模 2 补码，可以得出结论：对补码符号位扩展，其值不变。根据这一结论，可以将补码的符号位扩展成更多位，它们可以用在不同位数补码相加以及多位乘法运算等场合。

4) 由原码求补码的方法

对于正数，$[x]_{补}=[x]_{原}$。对于负数，有以下两种方法。

方法 1：符号除外，各位取反，末位加 1。

例 3.1 已知$[x]_{原}=1.0101010$，求$[x]_{补}$。

解：

$$[x]_{补}=1.1010101+0.0000001=1.1010110$$

“求反加 1”规则同样适用于由负数补码求原码。

方法 2：从最低位开始，对遇到的 0 和第一个 1 取其原码，从第一个 1 以后开始直到数值最高位均按位取反。

这种方法其实对应计算机中将原码变为补码形式时采用的串行电路。

5) 补码的性质

相反数的补码等于补码的相反数，也等于数的补码连同符号按位取反后末位再加 1。

例 3.2 证明：相反数的补码等于补码的相反数，即$[-x]_{补}=-[x]_{补}$。

证明：(以定点小数为例)

(1) 若 x 为正数，即$[x]_{补}=0.x_1x_2\cdots x_n$，则$[-x]_{补}=1./x_1/x_2\cdots/x_n+2^{-n}\ (\mathrm{mod}\ 2)$

又$-[x]_{补}=-0.x_1x_2\cdots x_n=2-0.x_1x_2\cdots x_n \quad (\mathrm{mod}\ 2)$

$=1.11\cdots1+2^{-n}-0.x_1x_2\cdots x_n \quad (\mathrm{mod}\ 2)$

$=1./x_1/x_2\cdots/x_n+2^{-n} \quad (\mathrm{mod}\ 2)$

$\therefore [-x]_{补}=-[x]_{补}$

(2) 若 x 为负数，即$[x]_{补}=1.x_1x_2\cdots x_n$，则$[-x]_{补}=0./x_1/x_2\cdots/x_n+2^{-n}\ (\mathrm{mod}\ 2)$

又$-[x]_{补}=-1.x_1x_2\cdots x_n=2-1.x_1x_2\cdots x_n \quad (\mathrm{mod}\ 2)$

$=0./x_1/x_2\cdots/x_n+2^{-n} \quad (\mathrm{mod}\ 2)$

$\therefore [-x]_{补}=-[x]_{补}$

例 3.3 **证明**：相反数的补码等于数的补码连同符号按位取反后末位再加 1，即若$[x]_{补}=x_0.x_1x_2\cdots x_n$，则$[-x]_{补}=/x_0./x_1/x_2\cdots/x_n+2^{-n}\ (\mathrm{mod}\ 2)$。

证明(以定点小数为例)：

(1) 若 x 为正数，即$[x]_{补}=0.x_1x_2\cdots x_n$，

则 $x=0.x_1x_2\cdots x_n$，$-x=-0.x_1x_2\cdots x_n$，

$\therefore [-x]_{补}=1./x_1/x_2\cdots/x_n+2^{-n}(\text{mod } 2)$

可以验证，$x=0$ 时成立。

(2) 若 x 为负数，即 $[x]_{补}=1.x_1x_2\cdots x_n$，

则 $[x]_{原}=1./x_1/x_2\cdots/x_n-2^{-n}$

$\therefore x=-(0./x_1/x_2\cdots/x_n+2^{-n})$

$\therefore -x=0./x_1/x_2\cdots/x_n+2^{-n}$，为正数

$\therefore [-x]_{补}=0./x_1/x_2\cdots/x_n+2^{-n}(\text{mod } 2)$

可以验证，$x=-1$ 时成立。

例 3.4　证明：设 $[x]_{补}=x_0.x_1x_2\cdots x_n$，则 $x=-x_0+0.x_1x_2\cdots x_n$。

证明：(以定点小数为例)

若 $x\geqslant 0$，则 $x_0=0$，$[x]_{补}=0.x_1x_2\cdots x_n=x$

$\therefore x=-x_0+0.x_1x_2\cdots x_n$

若 $x<0$，则 $x_0=1$，$[x]_{补}=1.x_1x_2\cdots x_n=2+x$

$$\therefore x=[x]_{补}-2=1.x_1x_2\cdots x_n-2=-1+0.x_1x_2\cdots x_n$$
$$=-x_0+0.x_1x_2\cdots x_n$$

根据例 3.4 得到的这一性质可以由补码求得真值。

例 3.5　已知 $[x]_{补}=x_0.x_1x_2\cdots x_n$，求 $[x/2]_{补}$。

解：

$\because x=-x_0+0.x_1x_2\cdots x_n$

$$\therefore \frac{1}{2}x=-\frac{1}{2}x_0+\frac{1}{2}\cdot(0.x_1x_2\cdots x_n)$$
$$=-x_0+\frac{1}{2}x_0+\frac{1}{2}\cdot(0.x_1x_2\cdots x_n)$$
$$=-x_0+\frac{1}{2}(x_0+0.x_1x_2\cdots x_n)$$
$$=-x_0+0.x_0x_1x_2\cdots x_n$$
$$\therefore \left[\frac{1}{2}x\right]_{补}=x_0.x_0x_1x_2\cdots x_n$$

由例 3.5 可以得到补码算术右移方法：原符号位不变，符号与数值位均右移一位。

3. 反码表示法

一般来说，反码只用于由原码求补码或由补码求原码的中间过渡：正数的反码表示与原、补码相同；负数的反码符号位为 1，数值位是将原码的数值按位取反。

很容易得到：对于负数而言，一个数的补码等于反码末位+1，反码等于补码末位−1。这个结论在补码除法运算中将用到。

小数的反码定义如下。

$$[x]_{反}=\begin{cases}x, & 0\leqslant x<1\\ (2-2^{-(n-1)})+x, & -1<x\leqslant 0\end{cases}\tag{3-7}$$

整数的反码定义如下。

$$[x]_{反}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ (2^n-1)+x, & -2^{n-1}<x\leqslant 0\end{cases}\tag{3-8}$$

从定义不难看出，反码也可以看作 mod$(2-2^{-n})$的补码，它与补码相比仅在末位差 1。其实，上述反码定义中的 $2-2^{-(n-1)}$ 和 2^n-1 就是全“1”形式。有资料上称小数的补码为 2 的补码，小数的反码为全“1”的补码。

4. 几种编码方法的比较

(1) 对正数，机器数就是真值本身。各类不同的编码方法，都只是对负数有着不同的表示。编码方法主要用来解决负数在机器中的表示。

(2) 编码中，最高位都表示符号位。

(3) 机器数所能表示的数值范围，补码稍大一点。

(4) 一般在加减运算中，采用补码最好；在乘除运算中，采用原码却很方便。

▶ 3.2.2　定点数加减法运算

计算机中常用补码进行加减运算。

虽然原码表示方法表示简单，易于同真值之间进行转换，实现乘除运算规则简单，但原码在算术运算时符号位需单独处理，且进行加减运算步骤复杂且费时，有时加法需用减法来实现。相反地，虽然补码在乘除法运算时相对于原码而言复杂，但是补码在运算时符号位参与运算且结果符号自动产生，其减法运算通过加法运算实现，且补码的加法运算步骤简单。由于计算机中加减运算远多于乘除运算，因此一般地，计算机中的加减法运算采用补码。

1. 定点补码加减法运算规则

两个补码的和(差)等于和(差)的补码，即

$$[X]_{补} \pm [Y]_{补} = [X]_{补} + [\pm Y]_{补} = [X \pm Y]_{补} \quad \text{mod } 2 \text{ 或 } 2^n \tag{3-9}$$

其中，n 为机器字长。

进行补码加减法运算的原理框图如图 3.2 所示。

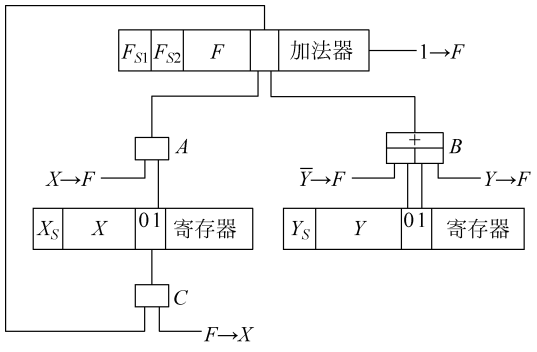


图 3.2　补码加减法运算的原理框图

例 3.6　已知机器字长 $n=8$，$X=44$，$Y=53$，求 $X+Y$ 的值。

解：

$[X]_{补}=0,0101100$，$[Y]_{补}=0,0110101$，

$$\begin{array}{r} 0,0\ 1\ 0\ 1\ 1\ 0\ 0 \\ +\ 0,0_1\ 1_1\ 1_1\ 0_1\ 1\ 0\ 1 \\ \hline 0,1\ 1\ 0\ 0\ 0\ 0\ 1 \end{array}$$

$\therefore [X+Y]_{补}=0,1100001$

$\therefore X+Y=+97$

例 3.7 已知机器字长 $n=8$，$X=-44$，$Y=-53$，求 $X-Y$ 的值。

解：

$[X]_{补}=1,1010100$，$[Y]_{补}=1,1001011$，$[-Y]_{补}=0,0110101$，

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

$\therefore [X-Y]_{补}=0,0001001$

$\therefore X-Y=+9$

例 3.8 已知机器字长 $n=8$，$X=120$，$Y=10$，求 $X+Y$ 的值。

解：

$[X]_{补}=0,1111000$，$[Y]_{补}=0,0001010$，

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

算出来显示 $[X+Y]_{补}=1,0000010$，即 $X+Y=-126$，结果显然不对。

例 3.8 结果之所以错了，是因为运算结果超出机器数值范围(8 位补码机器数表示范围为 $-128\sim+127$)，即发生溢出错误。

2. 溢出与溢出判断

溢出是一种**错误**，必须将溢出错误检出。

一旦检出溢出，系统要产生出错信号，由 CPU 执行纠检错程序进行处理，情况严重时将停机。

显然，溢出只可能发生在两个同号数相加或两个异号数相减时，两个异号数相加或两个同号数相减不可能发生溢出。

计算机中溢出判断有两类三种方法。两类分别是使用单符号位判断和使用双符号位判断。

不同的溢出判断规则与实际的判溢出硬件电路相对应，其性能、成本不尽相同。

1) 单符号位判断法

一种是通过结果符号进行判断：两同号数相加时，其运算结果符号应与被加数相同，若结果符号与被加数相异则溢出；两异号数相减时，其运算结果符号应与被减数相同，若结果符号与被减数相异则溢出。

这种单符号溢出判断电路如图 3.3 所示，其中，S_A、S_B 表示参与运算的两个数的符号，S_C 表示运算结果的符号，“+”“−”则表示所进行的运算；Over 是溢出标志，当它为“1”时表示运算结果溢出。

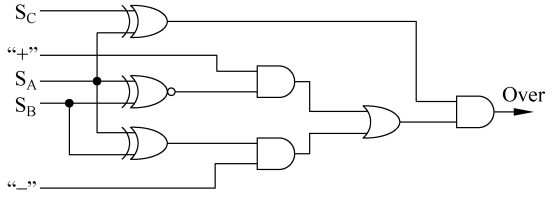


图 3.3 结果符号位溢出判断电路

显然，这种判断方法电路比较复杂。

另一种是通过进位符号进行判断：不论加或减，如果数值最高位向符号位的进位 C_s 与符号位向更高位的进位 C_f 相异，则发生了溢出。

显然，这种单符号位判断法比第一种方法实现起来更容易，电路简单（仅需要一个异或门），但是它需要把数值最高位进位与符号位进位单独引出来。

2）双符号位判断法

双符号位判断法通过变形补码实现，加法器采用双符号位：当结果的双符号相同时，表明不溢出，其中，如果都为“00”，那么结果为正数，如果都为“11”，那么结果就是负数；当结果的双符号位相异时，表明结果溢出，其中，如果为“01”，结果正溢出，如果为“10”，结果负溢出。换句话说，不论溢出或不溢出，最高符号位都是运算结果的真正符号位。

这种方法电路简单，也是只需要一个异或门就可以了，而且它不需要单独把进位信号引出来。当然，这里的加法器需要多加一位。

下面看几个计算机中加减法计算的例子，其中前 4 个例子采用变形补码计算并判溢出，后两个例子加法器采用 1 位符号位并采用进位符号法判溢出。

例 3.9　$X=0.1001, Y=0.0101$，求 $X+Y$ 的值。

解：

$[X]_{补}=00.1001, [Y]_{补}=00.0101$，

$$\begin{array}{r} 00.1001 \\ +\ 00.0101 \\ \hline 00.1110 \end{array}$$

两个符号位相同，无溢出。

$\therefore [X+Y]_{补}=00.1110$

$\therefore X+Y=0.1110$

例 3.10　$X=-0.1001, Y=-0.0101$，求 $X+Y$ 的值。

解：

$[X]_{补}=11.0111, [Y]_{补}=11.1011$，

$$\begin{array}{r} 11.0111 \\ +\ 11.1011 \\ \hline 111.0010 \end{array}$$

两个符号位相同，无溢出。

$\therefore [X+Y]_{补}=11.0010$

$\therefore X+Y=-0.1110$

例 3.11　$X=0.1011, Y=0.0111$，求 $X+Y$ 的值。

解：

$[X]_{补}=00.1011, [Y]_{补}=00.0111$，

$$\begin{array}{r} 00.1011 \\ +\ 00.0111 \\ \hline 01.0010 \end{array}$$

∵两个符号位为 01，相异

∴运算结果正向溢出

例 3.12 $X=-0.1011, Y=0.0111$，求 $X-Y$ 的值。

解：

$[X]_{补}=11.0101, [Y]_{补}=00.0111, [-Y]_{补}=11.1001$，

$$\begin{array}{r} 11.0101 \\ +\ 11.1001 \\ \hline 110.1110 \end{array}$$

∵两个符号位为 10，相异

∴运算结果负向溢出

例 3.13 $X=0.1001, Y=0.0101$，求 $X+Y$ 的值。

解：

$[X]_{补}=0.1001, [Y]_{补}=0.0101$，

$$\begin{array}{r} 0.1001 \\ +_0\ 0._0 0101 \\ \hline 0.1110 \end{array}$$

∵$C_s=C_f=0$，不溢出

∴$[X+Y]_{补}=0.1110$

∴$X+Y=+0.1110$

例 3.14 $X=0.1100, Y=0.0111$，求 $X+Y$ 的值。

解：

$[X]_{补}=0.1100, [Y]_{补}=0.0111$，

$$\begin{array}{r} 0.1100 \\ +_0\ 0._1 0111 \\ \hline 1.0011 \end{array}$$

∵$C_s=1, C_f=0, C_f\neq C_s$

∴运算结果溢出

3. 定点补码加减法运算电路原理图

定点补码加减法运算电路由加法器（或算术逻辑单元 ALU）和配套寄存器、门电路等构成，如图 3.4 所示。

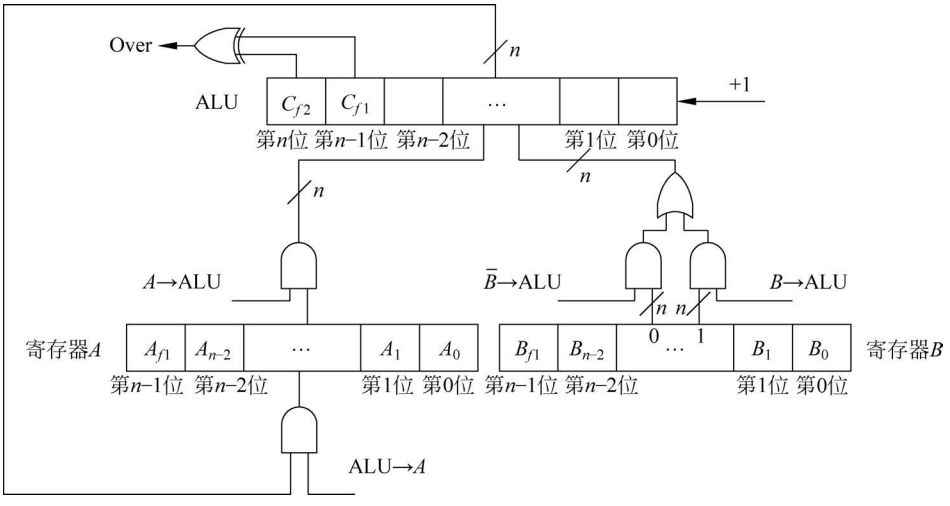


图 3.4　定点补码加减法运算电路原理图

图 3.4 中采用的是双总线结构，A 寄存器存放的是被加数/被减数以及最终的和/差，B 寄存器存放的是加数/减数。A、B 两个寄存器都是一位符号位，加法器则使用双符号位，溢出判断也采用双符号位判断法。

在进行补码加减法运算时，将被加数或者被减数由 A 寄存器送入加法器、将计算结果送入 A 寄存器都通过一个由相应控制信号控制的与门控制，将 B 寄存器中存放的加数或者减数送往加法器则由一个 2 选 1 多路选择器控制：当做加法运算时，将来自 B 寄存器的数据直接送往加法器；当做减法运算时，将来自 B 寄存器的数据按位取反送往加法器，并给出“末位加 1”信号，对于正逻辑 74181 来说，它其实就是将 C_n 置为“0”。

4. 十进制加减运算与十进制加法器

十进制运算以二进制运算为基础。

对十进制数码进行运算，根据编码方式和运算结果的不同，往往需要对结果进行修正。例如，8421 码十进制加法运算的规则是：若二进制相加的和数小于 10，不需修正；若二进制相加的和数大于或等于 10，则需加 0110 修正。如表 3.2 所示为 8421 码十进制加法运算结果修正表，容易得出修正条件为 $C_4+S_4S_3+S_4S_2$。

表 3.2 8421 码十进制加法运算结果修正表

修正前运算结果					修正后运算结果				
C_4	S_4	S_3	S_2	S_1	C_4'	F_4	F_3	F_2	F_1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	0	0
0	1	0	1	1	1	0	0	0	1
0	1	1	0	0	1	0	0	1	0
0	1	1	0	1	1	0	0	1	1
0	1	1	1	0	1	0	1	0	0
0	1	1	1	1	1	0	1	0	1
1	0	0	0	0	1	0	1	1	0
1	0	0	0	1	1	0	1	1	1
1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0	0	1

十进制加法器是十进制运算器的核心部件，它由二进制加法器加上一定的修正逻辑构成。例如，可以利用 74181 构成一位 8421 十进制加法器（如图 3.5 所示）。

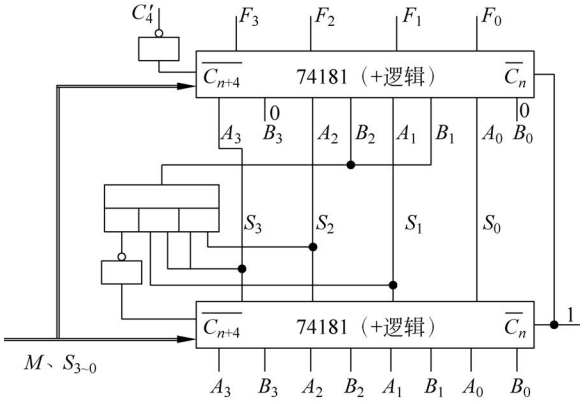


图 3.5　用 74181 构成一位 8421 十进制加法器

▶ 3.2.3　浮点数据表示

1. 浮点数据表示格式

计算机中的数据有定点数和浮点数两种表示方式。

所谓定点数，就是小数点位置固定，其中，定点小数的小数点固定在所有数值位的最前面，定点整数的小数点固定在所有数值位的最后面。前面讨论的都是定点数。

所谓浮点数是指小数点位置可浮动的数据。

不论是定点数还是浮点数，小数点都是隐含表示的。

同样机器字长的情况下，定点表示与浮点表示的表示范围和数据精度有所不同。

所谓数值范围是指一种数据类型所能表示的最大值和最小值；所谓数据精度是指实数所能表示的有效数字位数。数值范围与数据精度均与机器字长以及编码方式有关。

一般地，浮点数通常以下列形式表示。

$$N = M \times R^E$$

其中，N 是所表示的浮点数，M 是尾数，有 m 位，是一个纯小数（在 IEEE 754 标准中，带一位省略的整数位“1”）；E 称为阶码，是一个定点整数，$n+1$ 位，其中包含一位阶符 E_S，表示正阶或负阶；R 是阶的基数。对于一个确定的计算机而言，计算机中所有数据的 R 都是相同的，因此，不需要在每个数据中表示出来，即表示浮点数的时候只需要给出尾数和阶码的值就可以了。

显然，当基值相同时，阶码越长，表示范围越大；尾数越长，数据精度越高。

一般地，阶码部分使用补码或移码表示，尾数使用补码或原码表示，也有使用阶原尾原的；基值一般取 2，也有取 8 或 16 的。

浮点数的机内表示有两种格式，一种是数符与尾数数值部分放在一起：

E_s	$E_{n-1}\cdots E_0$	M_s	$M_{-1}\cdots M_{-m}$
阶符	阶码值	数符	尾数值

另一种是数符放在最前面：

M_s	E_s	$E_{n-1}\cdots E_0$	$M_{-1}\cdots M_{-m}$
数符	阶符	阶码值	尾数值

2. 移码

所谓移码，是为了便于进行定点整数比较大小而专门设计的一种机器数表示方式，它等于在真值上加一个常数，其定义为

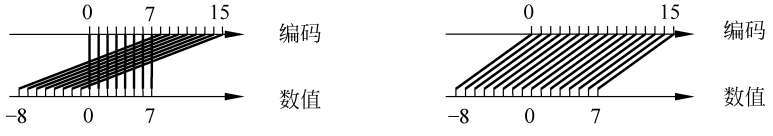
$$[X]_{移} = 2^{n-1} + X, -2^{n-1} \leqslant X < 2^{n-1}, \text{其中}, X \text{ 为整数} \tag{3-10}$$

移码定义中最小真值为-2^{n-1}，故最小真值的移码为0。

如图3.6(a)所示为补码表示时其数值与编码在数轴上的表示。

显然，对于同为正数或同为负数的情况可以方便地进行大小比较，但是一个正数与一个负数时则不能用编码直接进行比较。

如图3.6(b)所示为移码表示时其数值与编码在数轴上的表示。可见，移码在数轴上的表示范围恰好对应于真值在数轴上的范围向轴的正向移动若干单元，便于比较两个移码的大小。



(a) 补码表示时　　(b) 移码表示时

图3.6　数值与对应编码在数轴上的表示

根据定义：

若$0 \leqslant X < 2^{n-1}$，则$[X]_{移} = 2^{n-1} + X = 2^{n-1} + [X]_{补}$，

若$-2^{n-1} \leqslant X < 0$，则$[X]_{移} = 2^{n-1} + X = (2^n + X) - 2^{n-1} = [X]_{补} - 2^{n-1}$，

即$[X]_{补}$与$[X]_{移}$仅符号位相反，其余部分相同。

根据定义，$[+0]_{移} = 2^{n-1} + 0 = 2^{n-1} - 0 = [-0]_{移}$，即0的移码表现形式也是唯一的。

3. 浮点数的规格化

对于一个浮点数而言，字长固定的情况下提高表示精度有两种方法：增加尾数位数(但数值范围减小)或采用浮点规格化形式。

所谓浮点数规格化就是移动尾数，使尾数的有效数码尽可能地占满尾数的有位格。

采用浮点规格化形式可以提高数据精度，而且使同一个浮点数的表示唯一，并便于浮点数之间的运算与比较。

基数不同的浮点数，其规格化数的形式和过程也不同。本书只讨论基数为2的浮点数的规格化问题。

对规格化数的定义是这样的：设M为尾数，则规格化数应满足$1/2 \leqslant |M| < 1$。

将非规格化数变成规格化数的方法如下。调整阶码使尾数满足下列关系：尾数为原码表示时，无论正负应满足$1/2 \leqslant |M| < 1$，即小数点后的第一位数一定要为1，也就是说，正数的尾数应为$0.1x\cdots x$，负数的尾数应为$1.1x\cdots x$。尾数用补码表示时，小数最高位应与数符符号位相反，也就是说，正数应满足$0.1x\cdots x$，即$1/2 \leqslant d < 1$；负数应满足$1.0x\cdots x$，则$-1/2 > d \geqslant -1$。

因此，对原码表示的尾数，规格化就是移动尾数，使尾数的整数位为0，且小数位第一位为1；对于补码表示的尾数，规格化就是使尾数的符号位与数值位不同。

为便于操作，补码表示时尾数规格化采用双符号位。

对于补码表示的尾数，若尾数两位符号位相等，且与尾数第一位相等，则需左规。

左规是指尾数左移，每左移一位，阶码减1，直至尾数第一位与尾符不等为止。

例 3.15 $M=00.01xx\cdots x$，需要进行左规：尾数左移一位，阶码减 1，得 $M=00.1xx\cdots x$。

例 3.16 $M=11.110xx\cdots x$，需要进行左规：尾数左移两位，阶码减 2，得 $M=11.0xx\cdots x$。

若尾数两位符号位不等，则需右规。

右规是指尾数右移，每右移一位，阶码加 1，直至尾数两位符号位相等为止。

例 3.17 $M=01.xx\cdots x$，需要进行右规：尾数右移一位，阶码加 1，得 $M=00.1xx\cdots x$。

例 3.18 $M=10.xx\cdots x$，需要进行右规：尾数右移一位，阶码加 1，得 $M=11.0xx\cdots x$。

计算机中的浮点数一般都是规格化的，只有在进行加减运算时才会出现运算结果双符号位不一致的情况，这时候才需要右规，显然右规时位数只需要移动一位。

4. 浮点数溢出

定点数的溢出根据数值本身判断，浮点数的溢出则根据规格化后的阶码判断。

浮点数的上溢是指浮点数阶码大于机器最大阶码，此时发生错误，需要进行中断处理。

浮点数的下溢是指浮点数阶码小于机器最小阶码，这意味着所表示的数很小，几乎可以忽略不计。

当一个**规格化**浮点数的尾数为 0（不论阶码为何值），或阶码的值比能在机器中表示的最小值还小（**特例，阶码用移码表示时，移码**$\leqslant -2^n$）而不论尾数为何值时，计算机都把该浮点数看成零值，称为机器零。

设阶码为 $n+1$ 位（含一位符号位），则浮点数中的“零”值有以下几种。

（1）$M=0$，不论阶码为何值，$N=0$。

（2）$M\neq 0$，$E<-2^n$ 时，下溢，**认为 N=0（机器零）**。

（3）$M=0$，$E=-2^n$ 时，$N=0$，这是零的标准形式。

① 若浮点数采用阶补尾补格式，则为 1,00…00；0.00…00。

② 若浮点数采用阶移尾补格式，则为 0,00…00；0.00…00，即为全零形式。

5. 浮点数表示范围

设阶码的数值位为 n 位，尾数的数值位为 m 位，当浮点数用非规格化表示，且阶码、尾数均用原码表示时，其表示范围如下。

最大正数：$(1-2^{-m})\times 2^{(2^n-1)}$

最小负数：$-(1-2^{-m})\times 2^{(2^n-1)}$

最小正数：$2^{-m}\times 2^{-(2^n-1)}$

最大负数：$-2^{-m}\times 2^{-(2^n-1)}$

设阶码的数值位为 n 位，尾数的数值位为 m 位，当浮点数用规格化表示，且阶码、尾数均用原码表示时，其表示范围如下。

最大正数：$(1-2^{-m})\times 2^{(2^n-1)}$

最小负数：$-(1-2^{-m})\times 2^{(2^n-1)}$

最小正数：$2^{-1}\times 2^{-(2^n-1)}$

最大负数：$-2^{-1}\times 2^{-(2^n-1)}$

设阶码的数值位为 n 位，尾数的数值位为 m 位，当浮点数用规格化表示，且阶码、尾数均用补码表示时，其表示范围如下。

最大正数：$(1-2^{-m})\times 2^{(2^n-1)}$

最小负数：$-1\times2^{(2^n-1)}$

最小正数：$2^{-1}\times2^{-2^n}$

最大负数：$-(2^{-1}+2^{-m})\times2^{-2^n}$

字长相同的情况下，浮点表示数值的表示范围比定点表示范围大，但未增加所表示数值的个数。同时，计算机中的浮点数是离散空间，绝对值越大，浮点数分布越稀疏（如图 3.7 所示）。

需要注意的是，浮点数据在表示和运算时可能会出现舍入等，这是因为有些十进制数不能用浮点数精确表示。也正是因为存在舍入等现象，浮点数运算是不满足结合律的，即$(d+f)-d\neq f$。

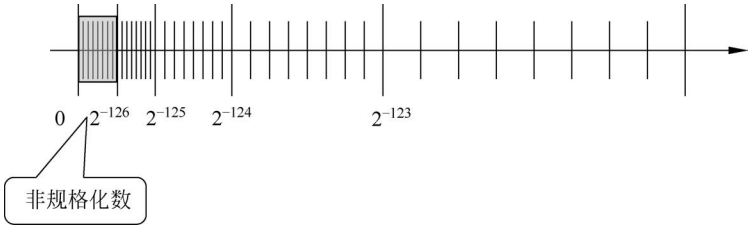


图 3.7　计算机中浮点数在数轴上的分布

6. IEEE 754 标准

微机中的数值有三类：无符号整数、带符号整数和浮点数。

微机中的无符号整数是二进制定点整数，分为字节、字、双字等不同的长度，其表示的数据范围也不同。

微机中的带符号整数有两类表示方式，一种是二进制定点整数表示，分为 16、32、64 位字长三种，均使用补码表示；一种是 80 位字长的 18 位 BCD 整数表示，如表 3.3 所示。

表 3.3　微机中带符号整数表示方法

整数类型	数值范围	精　度	格　式
16 位整数	$-32\,768\sim32\,767$	二进制 16 位	补码
短整数	$-2^{31}\sim2^{31}-1$	二进制 32 位	补码
长整数	$-2^{63}\sim2^{63}-1$	二进制 64 位	补码
BCD 整数	$-10^{18}+1\sim10^{18}-1$	十进制 18 位	原码，80b，最左字节最高位符号位（余 7 位无效）；余 72b 为 18 位 BCD 码

微机中的浮点数采用 IEEE 754 国际标准。IEEE 754 标准浮点数包括数符 S、阶码 E 和尾数 M 三个字段，格式如下。

$$(-1)^S\ 2^E\quad(M_0.M_{-1}\cdots M_{-(P-1)})$$

其中，数符 S 占 1 位，规定放在最高位，0 表示正、1 表示负；指数项 E 的基数为 2，是一个带有一定偏移量的无符号整数；尾数部分 M 是带一位整数的二进制小数真值（即原码）。

IEEE 754 标准浮点数的规格化是调整阶码，使尾数整数位 M_0 为 1 且与小数点一起隐含掉（扩展精度除外）。

IEEE 754 标准浮点数有三种表示格式，其中，单精度浮点数共 32 位，含阶码 8 位、尾数 24 位；双精度浮点数共 64 位，含阶码 11 位、尾数 53 位；扩展精度浮点数共 80 位，含阶码 15 位，尾数 64 位。其详细格式如表 3.4 所示。

表 3.4　IEEE 754 标准浮点数格式

参　数	单 精 度	双 精 度	扩展精度
浮点数长度(位)	32	64	80
符号位数	1	1	1
尾数长度 P(位)	23+1(隐)	52+1(隐)	64
阶码 E 长度(位)	8	11	15
最大阶码	+127	+1023	+16 383
最小阶码	−126	−1022	−16 382
阶码偏移量	$+127(2^7-1)$	$+1023(2^{10}-1)$	$+16\,383(2^{14}-1)$
表示数范围	$10^{-38}\sim10^{+38}$	$10^{-308}\sim10^{+308}$	

另外，利用 IEEE 754 标准格式可以表示一些特殊数据。

- 若阶码值全为 0，当尾数全为 0 时表示 0(0 有正、负之分)；当尾数非全 0 时为**可表示的非规格化数**，其指数为−126，尾数不隐含整数位“1”。
- 若阶码值全为 1，当尾数全为 0 时表示(正/负)无穷，这个结果可能来自被 0 除或上溢；当尾数非全 0 时表示**非数**(Not a Number，NaN)，可以用于表示$\sqrt{-1}$等操作的结果，或者在尾数部分存放系统状态、中断原因等特殊内容，供中断处理程序使用。

下面给出几个微机中浮点数据转换的例子。

例 3.19　将十进制数 178.125 表示成微机单精度浮点数(用十六进制表示)。

解：

$178.125=10110010.001\text{B}=1.0110010001\times2^7$

∵单精度浮点数应加的指数偏移量为 127

∴$E=7+127=134=(1000\ 0110)_2$

∴单精度浮点数形式为 0 10000110 011 0010 0010 0000 0000 0000

即十进制数 178.125 表示成微机单精度浮点数为 4332 2000H。

例 3.20　Pentium 机中的单精度浮点数 3F580000H 表示成十进制真值是多少？

解：

3F580000H ＝0011 1111 0101 1000 0000 0000 0000 0000B

＝0 011 1111 0 101 1000 0000 0000 0000 0000B

数符：$S=0$，正数

阶码：$E=(01111110)_2-127=126-127=-1$

尾数：$D=(1.1011)_2$

∴$X=(1.1011)_2\times2^{-1}=(0.11011)_2=(0.84375)_{10}$

需要注意的是，在将十进制数据转换成计算机内部数据过程中可能出现偏差。

例 3.21　将十进制数 3.3 表示成微机单精度浮点数。

解：

$$3.3_{(10)}=11.01\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\cdots_{(2)}$$

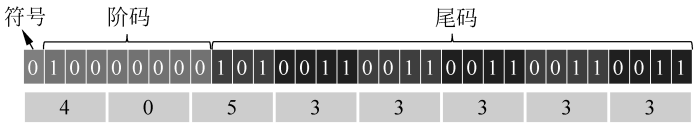
使整数位为 1，取小数 23 位，舍入采用末位置 1 法，得

$$3.3_{(10)}=1.101\ 0011\ 0011\ 0011\ 0011\ 0011\ (0011\cdots)\times2^1$$

则阶码

$$E=1+127=10000000_{(2)}$$

即计算机内表示为



值得注意的是，数据转换过程中有“舍”，因此实际保存的数据比原来的值要小。

下面再看一个例子。

例 3.22　将十进制数 1.1 表示成微机单精度浮点数。

解：

$$1.1_{(10)} = 1.0\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\cdots_{(2)}$$

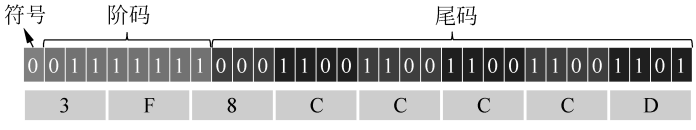
使整数位为 1，取小数 23 位，舍入采用末位置 1 法，得

$$1.1_{(10)} = 1.0\ 0011\ 0011\ 0011\ 0011\ 0011\ 01(11\cdots)\times 2^{0}$$

则阶码

$$E = 0+127 = 01111111_{(2)}$$

即计算机内表示为



在这个例子中，数据转换过程中有“入”，因此实际保存的数据比原来的值要大。

7. 定点数与浮点数比较

定点数小数点位置固定，其表示方法比浮点数简单，运算规则、运算速度以及进行运算的硬件成本方面都优于浮点数；浮点数在数的表示范围、数值精度、溢出处理方面优于定点数。

采用定点数还是浮点数，应根据具体应用选用。一般地，通用大型计算机多数采用浮点数，或同时采用定、浮点数；小型、微型及某些专用计算机、控制计算机，则多数采用定点数，当需要做浮点运算时，可通过软件实现，也可外加浮点扩展硬件（如协处理器）来实现。

3.3　二进制乘法运算

▶ 3.3.1　定点原码乘法

原码又称为带符号的绝对值表示方法，其乘法规则是：先取绝对值相乘，再根据同号相乘为正、异号相乘为负，单独决定符号位。

1. 定点原码一位乘

计算机中原码乘法的具体实现步骤是由笔算竖式乘法推导而来的。

设 $X=0.1101$，$Y=0.1011$，首先看笔算计算乘积 $X\times Y$ 的乘法过程（如图 3.8 所示）。

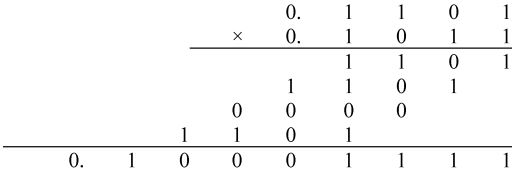


图 3.8　笔算竖式乘法过程

定点原码一位乘由笔算方法推导而来，先取绝对值相乘，再根据同号相乘结果为正、异号相乘结果为负，单独决定符号位。

把这个过程直接用计算机实现存在的第一个困难就是这里的多个数同时相加的问题。计算机中所有的运算本质上都是通过加法器实

现的，但是加法器每次只能实现两个数相加的操作。要解决这个问题，可以将多个数同时相加的操作改成多次两两相加的过程来实现。做乘法运算的时候，设置初始的部分积为 0，每次根据乘数位是“1”还是“0”来决定是加上被乘数的绝对值还是加上 0 形成新的部分积，乘数数值位有多少位就进行多少次加操作。

但是这里还有一个问题，就是不对齐 $2n$ 位相加的问题。事实上，每一次加操作只需要对相应的那 n 位做加法操作即可，在这以后，当前部分积最后一位以及往后的各位是不再参与运算的，所以，可以将笔算竖式乘法每次左移一位，不对齐加法操作改成 n 位加，每次加操作结束后将部分积右移一位，移出部分依次保存起来，部分积左边补“0”。这样，乘数数值位有多少位，就需要进行多少次加和移位操作。

还有一个需要解决的问题是控制各次加数的乘数数值位的位置不固定。事实上，进行乘法运算时，从乘数的低位开始，依次控制每次加操作的加数，一旦相应的加法操作完成以后，这一位就不再需要了，只需要在加法操作完成后将乘数与部分积同时右移，乘数右移时移出部分丢弃，而部分积移出部分保存在乘数寄存器的高位部分。当乘法运算结束时，乘数恰好都被移出，此时加法器中的结果是 n 位的乘积绝对值高位部分，乘数寄存器中存放的是 n 位的乘积绝对值低位部分。

在这里需要注意的是，这里的加操作用的就是以前学习过的加法器。加法操作是可能出现溢出的，但是对于乘法运算而言，每次加操作后紧接着一个右移操作，因此这种溢出是假“溢出”，因此在乘法运算过程中可以不理会这个“溢出”警告，并且每次右移的时候左边直接补“0”就可以了。

归纳上面的分析，可以得到定点小数原码一位乘的规则：符号位单独处理，绝对值相乘，通过加和移位实现，乘数数值位决定加和移位的次数；加法器采用单符号位或者双符号位，由乘数数值当前最低位决定本次加的内容，乘法过程中即使出现溢出也不算溢出；每次加操作完成后部分积与乘数数值位同时右移，乘数与部分积低位共用一个寄存器，右移时部分积左边补“0”，最低位移入乘数寄存器，乘数原来的最低位（也就是乘数寄存器原来的最低位）丢弃。

需要解释一点的是，这里的乘数寄存器在做乘法运算的时候里面存放的是乘数，但是在做除法运算的时候存放的是商，因此往往被称为“乘商寄存器”。

因为乘法过程中只是取的数值位相乘，所以上述规则对于定点整数乘法依然适用。

例 3.23　设 $X=0.1101$，$Y=0.1011$，求 $X \cdot Y$。

解：

	部分积 A						乘数 C				
	0.	0	0	0	0	0	1	0	1	1	
$+\|x\|$	0.	0	1	1	0	1					
	0.	0	1	1	0	1					
右移一位→	0.	0	0	1	1	0	1	1	0	1	$1_{(丢失)}$
$+\|x\|$	0.	0	1	1	0	1					
	0.	1	0	0	1	1					
右移一位→	0.	0	1	0	0	1	1	1	1	0	$1_{(丢失)}$
$+0$	0.	0	0	0	0	0					
	0.	0	1	0	0	1					
右移一位→	0.	0	0	1	0	0	1	1	1	1	$0_{(丢失)}$
$+\|x\|$	0.	0	1	1	0	1					
	0.	1	0	0	0	1					
右移一位→	0.	0	1	0	0	0	1	1	1	1	$1_{(丢失)}$
	乘积高位						乘积低位				

$\because X_0 \oplus Y_0 = 0 \oplus 0 = 0$

$\therefore [X \cdot Y]_{原} = 0.1000\ 1111$

$\therefore X \cdot Y = 0.1000\ 1111$

实现原码一位乘法的逻辑电路框图如图 3.9 所示。这里对两个 n 位数(含 1 位符号位)相乘,加法器、部分积寄存器采用双符号位,被乘数寄存器、乘商寄存器采用单符号位。需要提醒的是,这里的“移位器”并不一定需要一个真正的部件,其实可以通过右斜连线的方式实现移位功能。

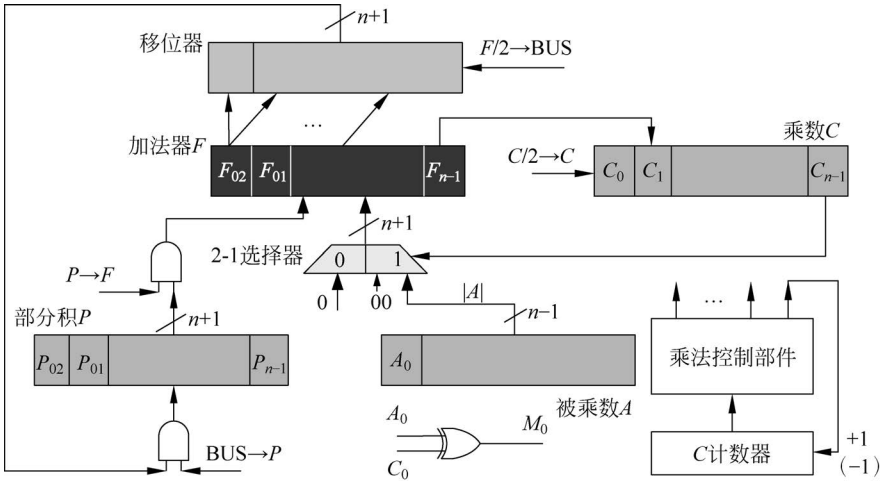


图 3.9　原码一位乘法电路逻辑框图

2. 定点原码两位乘

采用两位乘的目的是提高乘法的运算速度,其基本思想就是将两步并作一步走,它从乘数的低位开始,数值位每两位一组,由这个组合来决定每次加的对象,而且每次加以后不再是右移一位,而是每次右移两位,这样就能够将乘法的运算效率提升到原来的二倍。

两位乘数数值位一共有 4 种组合,对应 4 种操作:当两位乘数数值位为“00”时,相当于本次要做 $0 \cdot |X|$,此时只需要将部分积 P_i 右移两位即可,不需要进行其他操作;当两位乘数数值位为“01”时,相当于本次要做 $1 \cdot |X|$,然后进行部分积 $P_i + |X|$ 操作,并将部分积右移两位即可;当两位乘数数值位为“10”时,相当于本次要做 $2 \cdot |X|$,然后进行部分积 $P_i + 2|X|$ 操作,并将部分积右移两位即可;当两位乘数数值位为“11”时,相当于本次要做 $3 \cdot |X|$,然后进行部分积 $P_i + 3|X|$ 操作,并需要将部分积右移两位。

显然,$+2|X|$ 可等价于把 $|X|$ 左移一位即得;同理,$+3|X|$ 可等价于 $4|X| - |X|$;即以 $(4|X| - |X|)$ 代替 $+3|X|$,在本次运算中只执行 $-|X|$,而将 $+4|X|$ 归并到下一步执行(欠账)。到下一步时,由于部分积已经右移了两位,上一步欠下的 $+4|X|$ 已经变成了 $+|X|$。

具体的实现方法是,在机器中设置一个欠账触发器 C 来记录是否欠下 $+4|X|$,若本次欠账,则 $1 \rightarrow C$,即每一步乘法的实际操作需由 Y_{i-1}、Y_i、C 三位来控制。

特别地,如果完成最后一次加和移位操作后,欠账触发器 C 仍然为“1”,则最后还要加 X 还上欠账(但不移位)。

归纳起来,定点数原码两位乘的方法是:加法器采用三符号位,乘积符号由 $X_0 \oplus Y_0$ 得到。进行乘法运算之前(即初始时),将欠账触发器 C 置“0”。此后,根据当前乘数数值最低两位以及欠账触发器 C 的值进行操作的规则如表 3.5 所示。

表 3.5　原码两位乘操作规则

Y_{i-1}	Y_i	C	操　作	
0	0	0	$(P_i+0)2^{-2}$	$0\rightarrow C$
0	0	1	$(P_i+\|X\|)2^{-2}$	$0\rightarrow C$
0	1	0	$(P_i+\|X\|)2^{-2}$	$0\rightarrow C$
0	1	1	$(P_i+2\|X\|)2^{-2}$	$0\rightarrow C$
1	0	0	$(P_i+2\|X\|)2^{-2}$	$0\rightarrow C$
1	0	1	$(P_i-\|X\|)2^{-2}$	$1\rightarrow C$
1	1	0	$(P_i-\|X\|)2^{-2}$	$1\rightarrow C$
1	1	1	$(P_i+0)2^{-2}$	$1\rightarrow C$

乘数数值位数 n 应为偶数，共做 $n/2$ 步乘法和移位；当乘数数值位数 n 为奇数时，对于定点小数，应该在乘数末位补一个“0”，对于定点整数，应该在数值位最前面添一个“0”，凑成偶数位，一共做$(n+1)/2$ 步乘法和移位。当最后一步完成后，若 $C=1$，则还要做一次加 $|X|$ 的操作(但不移位)，即还上欠账。

例 3.24　设$[X]_{原}=0.100111$，$[Y]_{原}=0.100111$，求$[XY]_{原}$的值。

解：$[-|X|]_{补}=111.011001$，$2|X|=001.001110$，$X_0\oplus Y_0=0\oplus 0=0$。

	部分积	乘　数						欠账位 C	备注
	000.000000	.1	0	0	1	$\underline{1}$	$\underline{1}$	$\underline{0}$	
$+[-\|X\|]_{补}$	111.011001							$\underline{1}$	欠账
	111.011001								
$\rightarrow 2+2\|X\|$	111.110110	0	1	1	0	$\underline{0}$	$\underline{1}$	$\underline{1}$	
	001.001110								
	001.000100								
$\rightarrow 2+2\|X\|$	000.010001	0	0	0	1	$\underline{1}$	$\underline{0}$	$\underline{0}$	
	001.001110								
	001.011111								
$\rightarrow 2$	000.010111	1	1	0	0	0	1		
	乘积高位	乘积低位							

$\therefore[XY]_{原}=(0\oplus 0).010111\ 110001=0.010111\ 110001$

例 3.25　设$[X]_{原}=0.111111$，$[Y]_{原}=1.111001$，求$[XY]_{原}$的值。

解：$[-|X|]_{补}=111.000001$，$2|X|=001.111110$

	部分积	乘　数						欠账位 C	备注
	000.000000	.1	1	1	0	$\underline{0}$	$\underline{1}$	$\underline{0}$	
$+\|X\|$	000.111111	Y_1	Y_2	Y_3	Y_4	$\underline{Y_5}$	$\underline{Y_6}$		
	000.111111								
$\rightarrow 2+2\|X\|$	000.001111	1	1.	1	1	$\underline{1}$	$\underline{0}$	$\underline{0}$	
	001.111110					$\underline{Y_3}$	$\underline{Y_4}$		
	010.001101								
$\rightarrow 2+[-\|X\|]_{补}$	000.100011	0	1	1	1	$\underline{1}$	$\underline{1.}$	$\underline{0}$	
	111.000001					$\underline{Y_1}$	$\underline{Y_2}$	$\underline{1}$	欠账
	111.100100								
$\rightarrow 2+\|X\|$	111.111001	0	0	0	1	1	1.	$\underline{1}$	
	000.111111								还账
	000.111000								不移位
	乘积高位	乘积低位							

$\therefore [XY]_{原}=(0\oplus 1).111000\ 000111=1.111000\ 000111$

3. 阵列乘法器

两位乘法能够一定程度上提高乘法运算效率，但是对于需要大量乘法运算的场合，需要使用速度更快的乘法器，例如，采用阵列乘法器。

阵列乘法器仍然从无符号乘法手工计算过程（见图 3.10）分析而来。

阵列乘法器将上述笔算竖式中按位操作单元电路按阵列排列，直接实现乘法算式，其核心是全加器，通过这种资源重复的方式，避免重复的加和移位操作，从而换取运算速度的提升。

阵列乘法器单元电路如图 3.11 所示。

$$
\begin{array}{rcccc}
 & a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 \quad P_2 \quad P_1 \quad P_0
\end{array}
$$

图 3.10　无符号乘法手工计算过程

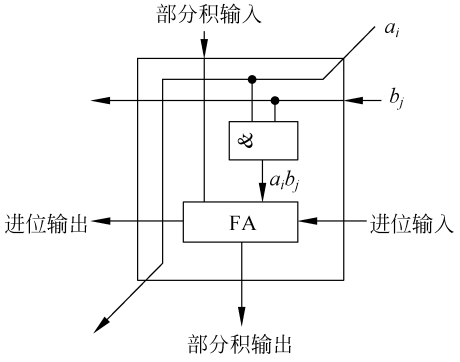


图 3.11　阵列乘法器单元电路

在图 3.11 中，由于本位参与乘法运算的非“0”即“1”，因此可以通过与门实现两个二进制位的乘法操作；全加器将本位乘积与本列已有部分积相加，进位则向水平传递。

4×4 无符号数阵列乘法器原理图如图 3.12 所示。

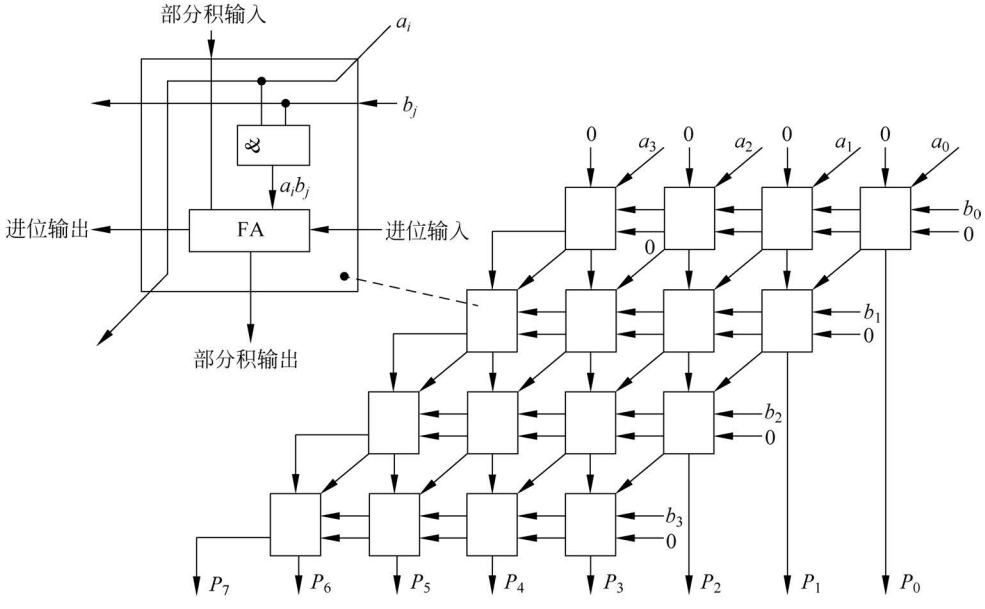


图 3.12　4×4 无符号数阵列乘法器原理图

$n\times n$ 位阵列乘法器需 n^2 个全加器和 n^2 个与门，但是，由于最上面一行的部分积输入为全“0”，因此这一行的全加器可以省略，即 $n\times n$ 位阵列乘法器共需 $n\times(n-1)$ 个全加器和 n^2 个与门。

▶ 3.3.2 定点补码乘法

计算机中的数据都是以补码形式存储的，如果采用原码方法计算，每次计算之前需要先将它变成原码，计算结束后还要将结果变成补码。

事实上，也可以直接用补码进行乘法运算，而且跟原码乘法不同的是，补码乘法运算结果的符号位是在运算中同时产生的。

1. 定点补码一位乘校正法

定点补码一位乘校正法的基本思想是：设被乘数$[X]_{补}=X_0.X_1X_2\cdots X_n$，乘数$[Y]_{补}=Y_0.Y_1Y_2\cdots Y_n$，则$[X\cdot Y]_{补}=[X]_{补}\cdot(-Y_0+0.Y_1Y_2\cdots Y_n)$。

用文字描述就是：把$[X]_{补}$与$[Y]_{补}$的数值位按原码乘法规则运算，如果乘数为正数，则得到的结果就是最终的结果；如果乘数为负数，则需将结果$+[-X]_{补}$进行校正才得到最终的结果。

下面对这一结论进行简单的证明。

证明：

(1) 若 X 正负任意，Y 为正数

$\because[X]_{补}=2+X=2^{n+1}+X\pmod 2$

$[Y]_{补}=Y=0.Y_1Y_2\cdots Y_n$

$\therefore[X]_{补}\cdot[Y]_{补}=2^{n+1}\cdot Y+X\cdot Y\pmod 2$

$\because 2^{n+1}\cdot Y=2^{n+1}\cdot(0.Y_1Y_2\cdots Y_n)=2^{n+1}\cdot\sum_{i=1}^{n}Y_i2^{-i}=2\cdot\sum_{i=1}^{n}Y_i2^{n-i}$

且$\sum_{i=1}^{n}Y_i2^{n-i}$是大于或等于1的正整数(或0)

$\therefore 2\cdot\sum_{i=1}^{n}Y_i2^{n-i}=2\pmod 2$

即$[X]_{补}\cdot[Y]_{补}=2+X\cdot Y=[X\cdot Y]_{补}\pmod 2$

$\therefore[X\cdot Y]_{补}=[X]_{补}\cdot[Y]_{补}=[X]_{补}\cdot(-Y_0+0.Y_1Y_2\cdots Y_n)=[X]_{补}\cdot(0.Y_1Y_2\cdots Y_n)$

(2) 若 X 正负任意，Y 为负数

$\because[Y]_{补}=1.Y_1Y_2\cdots Y_n=2+Y$

$\therefore Y=[Y]_{补}-2=0.Y_1Y_2\cdots Y_n-1$

$\therefore X\cdot Y=X\cdot(0.Y_1Y_2\cdots Y_n)-X$

$\therefore[X\cdot Y]_{补}=[X\cdot(0.Y_1Y_2\cdots Y_n)]_{补}+[-X]_{补}$

$\because 0.Y_1Y_2\cdots Y_n>0$

$\therefore[X\cdot(0.Y_1Y_2\cdots Y_n)]_{补}=[X]_{补}\cdot(0.Y_1Y_2\cdots Y_n)$

$\therefore[X\cdot Y]_{补}=[X]_{补}(0.Y_1Y_2\cdots Y_n)+[-X]_{补}$

$=[X]_{补}(0.Y_1Y_2\cdots Y_n)-[X]_{补}$

$=[X]_{补}(0.Y_1Y_2\cdots Y_n-Y_0)$

结合(1)和(2)可得补码乘法的统一算法：

$[X\cdot Y]_{补}=[X]_{补}(0.Y_1Y_2\cdots Y_n)-[X]_{补}\cdot Y_0=[X]_{补}\cdot(-Y_0+0.Y_1Y_2\cdots Y_n)$

具体地讲，定点补码一位乘校正法运算规则如下。

- 乘法运算时，把乘数的补码$[Y]_{补}$去掉符号位，当成正数与$[X]_{补}$按照同原码一样的方法相乘。
- 被乘数和部分积的符号位一同参与运算。
- 部分积是否$+[X]_{补}$由Y_i决定：$Y_i=1$时加$[X]_{补}$，$Y_i=0$时不加，然后右移一位，两个n位数相乘，共做n次加法和n次移位。
- 相加时采用两位符号位，以便存放乘法过程中绝对值大于或等于1的中间值。
- 右移必须按补码规则进行，即左边补符号位。
- 乘数为负时，最后求出的部分积需$+[-X]_{补}$进行校正，但不移位。

例 3.26　设$X=-0.1101$，$Y=0.1011$，即$[X]_{补}=11.0011$，$[Y]_{补}=Y=0.1011$，求$[X\cdot Y]_{补}$的值。

解：

	部分积 A						乘数 C				说　明
	0	0	0	0	0	0	1	0	1	1	
$+[X]_{补}$	1	1	0	0	1	1					
	1	1	0	0	1	1					
右移一位→	1	1	1	0	0	1	1	1	0	1	$1_{(丢失)}$
$+[X]_{补}$	1	1	0	0	1	1					
	1	0	1	1	0	0					
右移一位→	1	1	0	1	1	0	0	1	1	0	$1_{(丢失)}$
右移一位→	1	1	1	0	1	1	0	0	1	1	$0_{(丢失)}$
$+[X]_{补}$	1	1	0	0	1	1					
	1	0	1	1	1	0					
右移一位→	1	1	0	1	1	1	0	0	0	1	$1_{(丢失)}$
	乘积高位						乘积低位				

$\therefore [XY]_{补}=1.0111\ 0001$

例 3.27　设$X=-0.1101$，$Y=-0.1011$，即$[X]_{补}=11.0011$，$[Y]_{补}=11.0101$，求$[X\cdot Y]_{补}$的值。

解：$[-X]_{补}=00.1101$

	部分积 A						乘数 C				说　明
	0	0	0	0	0	0	0	1	0	1	
$+[X]_{补}$	1	1	0	0	1	1					
	1	1	0	0	1	1					
右移一位→	1	1	1	0	0	1	1	0	1	0	$1_{(丢失)}$
右移一位→	1	1	1	1	0	0	1	1	0	1	$0_{(丢失)}$
$+[X]_{补}$	1	1	0	0	1	1					
	1	0	1	1	1	1					
右移一位→	1	1	0	1	1	1	1	1	1	0	$1_{(丢失)}$
右移一位→	1	1	0	1	1	1	1	1	1	1	$0_{(丢失)}$
$+[-X]_{补}$	0	0	1	1	0	1					校正
	0	0	1	0	0	0	1	1	1	1	不移位
	乘积高位						乘积低位				

$\therefore [XY]_{补}=0.1000\ 1111$

2. 定点补码一位乘比较法

使用校正法当乘数为正和为负时其运算规律不统一，控制较复杂，而且计算机无法准确预计执行乘法指令所需要的时间。布斯(Booth)对前述的校正法补码乘法公式进行变换，得出了另一公式，又称为布斯公式，它避免区分乘数的正负，而且让乘数的符号也参加运算，使运算规律统一。

$$
\begin{aligned}
[X\cdot Y]_{补} &= [X]_{补}\cdot(-Y_0+0.Y_1Y_2\cdots Y_n)\\
&= [X]_{补}\cdot(-Y_0+2^{-1}Y_1+2^{-2}Y_2+\cdots+2^{-n}Y_n)\\
&= [X]_{补}\cdot[-Y_0+(Y_1-2^{-1}Y_1)+(2^{-1}Y_2-2^{-2}Y_2)+\cdots+(2^{-(n-1)}Y_n-2^{-n}Y_n)]\\
&= [X]_{补}\cdot[(Y_1-Y_0)+(Y_2-Y_1)\,2^{-1}+(Y_3-Y_2)\,2^{-2}+\cdots+\\
&\quad (Y_n-Y_{n-1})\,2^{-(n-1)}+(Y_{n+1}-Y_n)\,2^{-n}]\\
&= [X]_{补}\cdot(Y_1-Y_0)+2^{-1}([X]_{补}(Y_2-Y_1)+2^{-1}([X]_{补}(Y_3-Y_2)+\cdots+\\
&\quad 2^{-1}([X]_{补}(Y_n-Y_{n-1})+2^{-1}([X]_{补}(Y_{n+1}-Y_n)))\cdots))
\end{aligned}
$$

其中，乘数的最低一位为 Y_n，为了得到统一的形式，在其后再添加一位 Y_{n+1}，值为 0(对于补码小数而言，无论正负，在后面添“0”不影响其值)。

将上式改写，得到递推公式：

$$
\begin{aligned}
&[P_0]_{补}=0\\
&[P_1]_{补}=2^{-1}([P_0]_{补}+(Y_{n+1}-Y_n)[X]_{补}),Y_{n+1}=0\\
&[P_2]_{补}=2^{-1}([P_1]_{补}+(Y_n-Y_{n-1})[X]_{补})\\
&\qquad\cdots\\
&[P_i]_{补}=2^{-1}([P_{i-1}]_{补}+(Y_{n-i+2}-Y_{n-i+1})[X]_{补})\\
&\qquad\cdots\\
&[P_n]_{补}=2^{-1}([P_{n-1}]_{补}+(Y_2-Y_1)[X]_{补})\\
&[P_{n+1}]_{补}=[P_n]_{补}+(Y_1-Y_0)[X]_{补}=[XY]_{补}
\end{aligned}
$$

由此得补码一位乘比较法运算规则：部分积和被乘数采用两位符号位，乘数采用一位符号位，并设附加位 $Y_{n+1}=0$，通过乘数寄存器的末两位决定下一步的操作(如表 3.6 所示)。

表 3.6　补码一位乘比较法操作规则

判断位 Y_iY_{i+1}	操 作 规 则
0　0	$[P_i]_{补}\rightarrow$
0　1	$[P_i]_{补}+[X]_{补}$后$\rightarrow$
1　0	$[P_i]_{补}+[-X]_{补}$后$\rightarrow$
1　1	$[P_i]_{补}\rightarrow$

但是当进行到最后一步($i=n+1$)时，不移位。

例 3.28　设 $X=-0.1101$，$Y=0.1011$，即$[X]_{补}=11.0011$，$[Y]_{补}=0.1011$，$[-X]_{补}=00.1101$，求$[X\cdot Y]_{补}$的值。

解：

	部分积 A						乘数 C					说　明
	0	0	0	0	0	0	0.	1	0	1	1	0 $Y_4Y_5=10$，$+[-X]_{补}$
$+[-X]_{补}$	0	0	1	1	0	1					Y_i	Y_{i+1}
	0	0	1	1	0	1						
→	0	0	0	1	1	0	1	0	1	0	1	1 $Y_3Y_4=11$，直接→
→	0	0	0	0	1	1	0	1	0	1	0	1 $Y_2Y_3=01$，$+[X]_{补}$
$+[X]_{补}$	1	1	0	0	1	1						
	1	1	0	1	1	0						
→	1	1	1	0	1	1	0	0	1	0	1	0 $Y_1Y_2=10$，$+[-X]_{补}$
$+[-X]_{补}$	0	0	1	1	0	1						
	0	0	1	0	0	0						
→	0	0	0	1	0	0	0	0	0	1	0	1 $Y_0Y_1=01$，$+[X]_{补}$
$+[X]_{补}$	1	1	0	0	1	1						
	1	1	0	1	1	1	0	0	0	1		最后一步不移位

$\therefore[X\cdot Y]_{补}=1.0111\ 0001$

3. 定点补码两位乘

跟原码一样，同样定点数补码两位乘通过两步并做一步走提高乘法运算效率。

根据补码一位乘的布斯算法，将两步合并成一步：

$$[P_{i+1}]_{补}=2^{-1}([P_i]_{补}+(Y_{n-i+1}-Y_{n-i})[X]_{补})$$

$$[P_{i+2}]_{补}=2^{-1}([P_{i+1}]_{补}+(Y_{n-i}-Y_{n-i-1})[X]_{补})$$

将上面两步合成一步得：

$$\begin{aligned}[P_{i+2}]_{补}&=2^{-1}(2^{-1}([P_i]_{补}+(Y_{n-i+1}-Y_{n-i})[X]_{补})+(Y_{n-i}-Y_{n-i-1})[X]_{补})\\&=2^{-2}([P_i]_{补}+(Y_{n-i+1}-Y_{n-i}+2Y_{n-i}-2Y_{n-i-1})[X]_{补})\\&=2^{-2}([P_i]_{补}+(Y_{n-i+1}+Y_{n-i}-2Y_{n-i-1})[X]_{补})\end{aligned}$$

即部分积$[P_i]_{补}$加上乘数寄存器低两位和附加位组合值与$[X]_{补}$的积后右移两位，即为$[P_{i+2}]_{补}$。

由此得到补码两位乘比较法规则：加法器设三符号位，做乘法运算之前添加一位附加位$Y_{n+1}=0$。此后，根据当前乘数数值最低两位以及附加位的值进行操作的规则如表 3.7 所示。

表 3.7　补码两位乘操作规则

Y_{n-i-1}	Y_{n-i}	Y_{n-i+1}	组合值	$[P_{i+2}]_{补}$	说　明
0	0	0	0	$([P_i]_{补}+0)\times2^{-2}$	直接右移 2 位
0	0	1	1	$([P_i]_{补}+[X]_{补})\times2^{-2}$	$+[X]_{补}$，右移 2 位
0	1	0	1	$([P_i]_{补}+[X]_{补})\times2^{-2}$	$+[X]_{补}$，右移 2 位
0	1	1	2	$([P_i]_{补}+2[X]_{补})\times2^{-2}$	$+2[X]_{补}$，右移 2 位
1	0	0	−2	$([P_i]_{补}+2[-X]_{补})\times2^{-2}$	$+2[-X]_{补}$，右移 2 位
1	0	1	−1	$([P_i]_{补}+[-X]_{补})\times2^{-2}$	$+[-X]_{补}$，右移 2 位
1	1	0	−1	$([P_i]_{补}+[-X]_{补})\times2^{-2}$	$+[-X]_{补}$，右移 2 位
1	1	1	0	$([P_i]_{补}+0)\times2^{-2}$	直接右移 2 位

若乘数由一位符号位和 n（奇数）位数值组成，则只需做$(n+1)/2$步乘法，且最后一步只右移一位；若乘数数值位数 n 为偶数，可以有两种方法：对于小数，在乘数末位补“0”，对于整数，在数值最高位补“0”，使乘数总位数仍为偶数，还按照原方法进行；或者，乘数设两位符号

位，使总位数仍为偶数，共做 $n/2+1$ 步乘法，且最后一步不右移。

例 3.29 设 $X=-0.1101$，$Y=-0.1011$，即$[X]_{补}=1.0011$，$[Y]_{补}=1.0101$，求$[XY]_{补}$的值。

解：

$[X]_{补}=111.0011$，$2[X]_{补}=110.0110$，$[-X]_{补}=000.1101$，$2[-X]_{补}=001.1010$

方法一：$[Y]_{补}=1.01010$

部分积		乘 数						附加位	备注
	000.0000	1.	0	1	0	1	0	0	组合值=−2
$+2[-X]_{补}$	001.1010								
	001.1010								
→2	000.0110	1	0	1	0	1	0	1	
$+[-X]_{补}$	000.1101								组合值=−1
	001.0011								
→2	000.0100	1	1	1	0	1	0	1	
$+[-X]_{补}$	000.1101								组合值=−1
	001.0001								
→1	000.1000	1	1	1	1	0	1	0	
	乘积高位			乘积低位					

$\therefore[X\cdot Y]_{补}=0.1000\ 1111$

方法二：$[Y]_{补}=11.0101$

部分积		乘 数						附加位	备注
	000.0000	1	1.	0	1	0	1	0	组合值=1
$+[X]_{补}$	111.0011								
	111.0011								
→2	111.1110	1	1	1	1	0	1	0	组合值=1
$+[X]_{补}$	111.0011								
	110.1111								
→2	111.1011	1	1	1	1	1	1	0	组合值=−1
$+[-X]_{补}$	000.1101								
	000.1000								最后一步不移位
	乘积高位			乘积低位					

$\therefore[X\cdot Y]_{补}=0.1000\ 1111$

3.4 二进制除法运算

计算机定点小数除法要求被除数、除数均为定点小数（纯小数），且位数相同或者被除数是除数位数的 2 倍。除法运算的商为定点小数（纯小数），位数同除数。为避免出现非纯小数商的情况，要求必须$|X|<|Y|$（当被除数是除数位数的 2 倍时，要求被除数的高位部分组成的数（X）的绝对值小于除数（Y）），否则$|X|\geqslant|Y|$会导致结果溢出。一旦溢出，停止运算，报溢出中断。

▶ 3.4.1 定点原码一位除

原码除法运算跟原码乘法运算类似，也是取绝对值相除，单独决定商的符号位。计算机中

的原码除法规则同原码乘法规则一样，也是通过分析笔算竖式除法，并将其改为便于计算机实现的方式而得到的。

1. 恢复余数法

设 $X=0.1011$，$Y=0.1101$，二进制小数笔算除法求商 C 和余数 R 的过程如下。

		0.1101	
1101		0.10110	$\|X\|<\|Y\|$，商 0，末位添 0
	−	0.01101	$R_0>2^{-1}\|Y\|$，商 1，做减法并末位添 0
		0.010010	
	−	0.001101	$R_1>2^{-2}\|Y\|$，商 1，做减法并末位添 0
		0.0001010	
		0.00010100	$R_2<2^{-3}\|Y\|$，商 0，末位添 0
	−	0.00001101	$R_3>2^{-4}\|Y\|$，商 1，做减法得余数
		0.00000111	R_4

$\therefore C=0.1101$，$R=R_4=0.0111\times 2^{-4}$

将上述笔算竖式除法用计算机实现，首先第一个问题，就是如何比较大小。生活中比较两个数的大小是通过对应位从高到低依次比较完成的，但是计算机中比较大小要利用减法运算，通过判断符号位来判断够减还是不够减。也就是说，计算机确定每一位商都需要做一次减法操作，即通过 $+[-|Y|]_{补}$ 实现。如果加法器采用双符号位，当结果符号位为“00”时，表示够减；当结果符号位为“11”时，表示不够减。因为原码除法过程中是取绝对值进行相减，所以一定是不会溢出的。

如果够减，按照笔算竖式做法，本来就应该商“1”，然后做减法，得到余数以后后边补“0”接着除，只不过这里为了比大小先做了减法，所以这里直接商“1”，余数后边补“0”接着除没有问题。

但是如果不够减呢？在笔算竖式除法中，如果不够减，商“0”，此时是不做减法的，直接在原来的余数（第一步为被除数）后边补“0”接着除，但是这里为了比较大小，已经先做了减法，那么此时要先做一个加法，即做一次 $+[|Y|]_{补}$ 操作恢复余数，然后补“0”接着除。

需要注意的是，在除法不溢出时，进行第一步减法操作一定是不够减的，这个时候上了一位商“0”，这个“0”位于商的符号位的位置，但是它并不是商的符号。

除了需要恢复余数，与原码乘法类似，这里也有总共 $2n$ 位、不对齐减的问题。在笔算竖式除法过程中，每次比大小实际做的是 $n+1$ 位减法，而且本位商确定并得出当前余数后，最高位就不再参与运算了。所以，使用与原码除法相似的思路，可以将笔算竖式总共 $2n$ 位的不对齐减、每次上完商后除数右移的操作过程，改成每次数值位 n 位的对齐减、每次上完商后被除数或余数左移的操作过程。左移时右边补 0，无用的高位移出丢弃，最终的余数是左移 n 次以后的结果。

除了不对齐减法，这里还有一个上商的位置不固定的问题。在笔算竖式除法过程中，每次上商的位置是与当前余数的最低位一致的。联想到刚才每上完一位商以后余数是要左移一位并且在右边补“0”的，可以设置一个商寄存器，它的初始值就是全“0”，每次将它与余数一起左移，这样只需要每次将商放在商寄存器的最低位就可以了。

当然，这里初始的时候也可以不是全“0”，假设被除数的数值位数是除数数值位数的二倍，这里可以存放被除数数值位的低位部分。

由此，可以归纳出定点数原码一位除恢复余数法的运算规则，这里假设被除数 $[X]_{原}=X_0.X_1X_2\cdots X_n$，除数 $[Y]_{原}=Y_0.Y_1Y_2\cdots Y_n$。

(1) 原码相除,符号位单独处理,结果符号位 $C_0=X_0\oplus Y_0$。

(2) 数值部分取绝对值相除,其中要求 $|X|<|Y|$,否则除法溢出。

(3) 第一步先做 $|X|-|Y|=R_0'$,若 $R_0'<0$,商"0",做 $R_0'+|Y|=R_0$ 恢复余数。

(4) 以后每步除法都通过余数左移一位,然后 $-|Y|$ 实现,也就是 $2R_i-|Y|$。

① 如果 $2R_i-|Y|=R_{i+1}\geqslant 0$,也就是说,余数为正,那么商"1"。

② 如果 $2R_i-|Y|=R_{i+1}<0$,也就是说,余数为负,就商"0",并且做 $+|Y|$ 恢复余数。

(5) 用这种方法一共求出小数点后 n 位商,拼接上符号位得到商 $C=C_0.C_1C_2\cdots C_n$。

(6) 余数符号同被除数符号,结果余数为 $2^{-n}R_n$,如果最后的余数为负,要做 $+|Y|$ 操作恢复余数。

需要注意的是,在进行除法运算的过程中,即使已经除尽,也就是某一步余数已经为 0,计算机仍然会继续做除法,直到所求商满足位数要求为止。

例 3.30 设 $X=0.1011$,$Y=0.1101$,求 X/Y。

解:$|X|=0.1011$,$|Y|=0.1101$,$[-|Y|]_{补}=11.0011$

	被除数/余数		商 R 的状态	说 明
	00.1011		0.0 0 0 0	开始情形
$+[-\lvert Y\rvert]_{补}$	11.0011			
	11.1110	$R_0'<0$	0.0 0 0 $\underline{0}$	不够减,商 0
$+\lvert Y\rvert$	00.1101		C_0	**恢复余数**
	00.1011	R_0		
←	01.0110		0.0 0 $\underline{0}$ 0	
$+[-\lvert Y\rvert]_{补}$	11.0011		C_0	
	00.1001	$R_1>0$	0.0 0 $\underline{0}$ $\underline{1}$	够减,商 1
←	01.0010		0.0 $\underline{0}$ $\underline{1}$ 0	
$+[-\lvert Y\rvert]_{补}$	11.0011		C_0C_1	
	00.0101	$R_1>0$	0.0 $\underline{0}$ $\underline{1}$ $\underline{1}$	够减,商 1
←	00.1010		0.$\underline{0}$ $\underline{1}$ $\underline{1}$ 0	
$+[-\lvert Y\rvert]_{补}$	11.0011		$C_0C_1C_2$	
	11.1101	$R_3'<0$	0.$\underline{0}$ $\underline{1}$ $\underline{1}$ $\underline{0}$	不够减,商 0
$+\lvert Y\rvert$	00.1101		$C_0C_1C_2C_3$	**恢复余数**
	00.1010	R_3		
←	01.0100		$\underline{0}$.$\underline{1}$ $\underline{1}$ $\underline{0}$ 0	
$+[-\lvert Y\rvert]_{补}$	11.0011		$C_0C_1C_2C_3$	
	00.0111	$R_4>0$	$\underline{0}$.$\underline{1}$ $\underline{1}$ $\underline{0}$ $\underline{1}$	够减,商 1
			$C_0C_1C_2C_3C_4$	

商的符号位 $X0\oplus Y0=0\oplus 0=0$。

$\therefore[C]_{原}=0.1101$ $[R]_{原}=0.0111\times 2^{-4}$

$\therefore X/Y=0.1101$ 余数 $R=0.0000\ 0111$

显然,定点原码一位除恢复余数法的上商和左移必须分为两步完成。

定点原码一位除恢复余数法除法时间较长,速度低;除法步数不固定,无法准确预测一次除法运算所需要的时间,而且因为有时需要恢复余数、有时不需要,故控制较复杂。

2. 加减交替法

加减交替法是对恢复余数法的一种修正,其优点是操作步骤固定且易于编程。

在恢复余数法中，当第 i 次余数 $R_i<0$ 时，第 i 位上商“0”，且恢复余数，即 $R_i+|Y|$，然后左移一位，$-|Y|$，这个过程合起来即 $R_{i+1}=2(R_i+|Y|)-|Y|=2R_i+|Y|$。因此，无论是否够减，求 R_{i+1} 不必恢复余数。

当 $R_i\geqslant 0$ 时，上商“1”，$R_{i+1}=2R_i-|Y|$。

当 $R_i<0$ 时，上商“0”，$R_{i+1}=2R_i+|Y|$。

即加减交替法的处理思想是：先减后判，如发现不够减，则在下一步将减去除数的操作改做加除数操作，但是，这里表面上是做的加法，其实是将上一步的恢复余数和这一步的减去除数操作合并在一起了。

需要注意的是，如果求得最后一步商的时候余数为负（不够减），需要恢复余数。

例 3.31 设 $X=0.1011$，$Y=0.1101$，$[-|Y|]_{补}=11.0011$，用加减交替法求 X/Y。

解：

	被除数/余数	除数/商	操作说明
	00.1011	0.0 0 0 0	开始情形
$+[-\|Y\|]_{补}$	11.0011		开始先做减法
	11.1110	0.0 0 0 **0**	不够减，商 0
←	11.1100	0.0 0 **0** **0**	左移
$+\|Y\|$	00.1101		$+\|Y\|$
	00.1001	0.0 0 **0** **1**	够减，商 1
←	01.0010	0.0 **0** **1** **0**	左移
$+[-\|Y\|]_{补}$	11.0011		$-\|Y\|$
	00.0101	0.0 **0** **1** **1**	够减，商 1
←	00.1010	0.**0** **1** **1** **0**	左移
$+[-\|Y\|]_{补}$	11.0011		$-\|Y\|$
	11.1101	0.**0** **1** **1** **0**	不够减，商 0
←	11.1010	**0**.**1** **1** **0** **0**	左移
$+\|Y\|$	00.1101		$+\|Y\|$
	00.0111	**0**.**1** **1** **0** **1**	够减，商 1

符号位 $=0\oplus 0=0$，$\therefore X/Y=0.1101$，余数 $=0.0000\ 0111$。

下面归纳一下定点数原码一位除加减交替法的规则。

(1) 加法器取双符号位，商寄存器取单符号位；商的符号位单独处理，数值部分按绝对值相除。

(2) 如果被除数与除数位数相同，要求 $|X|<|Y|$，且商寄存器初始为全“0”；如果被除数的位数是除数的两倍，被除数数值低位部分开始时放在商寄存器中，并且要求被除数数值高位部分的绝对值小于除数的绝对值。

(3) 第一步操作：$-|Y|$，得余数 R_0，此后根据余数 R_i 上商并确定下一步操作。

① 如果 R_i 为正，表明够减，余数左移一位，上商“1”，下一步做减法。

② 如果 R_i 为负，表明不够减，余数左移一位，上商“0”，下一步做加法。

(4) 移位操作时当前余数和商寄存器的内容一起左移，商上在左移后商寄存器的最低位。

(5) 求不含符号位的 n 位商，除了第一步的减法操作，还需要做 n 次“左移-加/减”循环。

(6) 在求出第 n 位商时，如果余数为负，需要恢复余数。

同样，这里第一步操作其实是用于判溢出，若第一步够减，表明除法溢出；不溢出的时候商“0”，这第一个“0”上在商的符号位上，但它不是商的符号；另外，除法得到的余数经过 n 次

左移后的结果，最终的真余数的符号应该与被除数的符号一致。

如图 3.13 所示是原码一位除加减交替法原理框图。

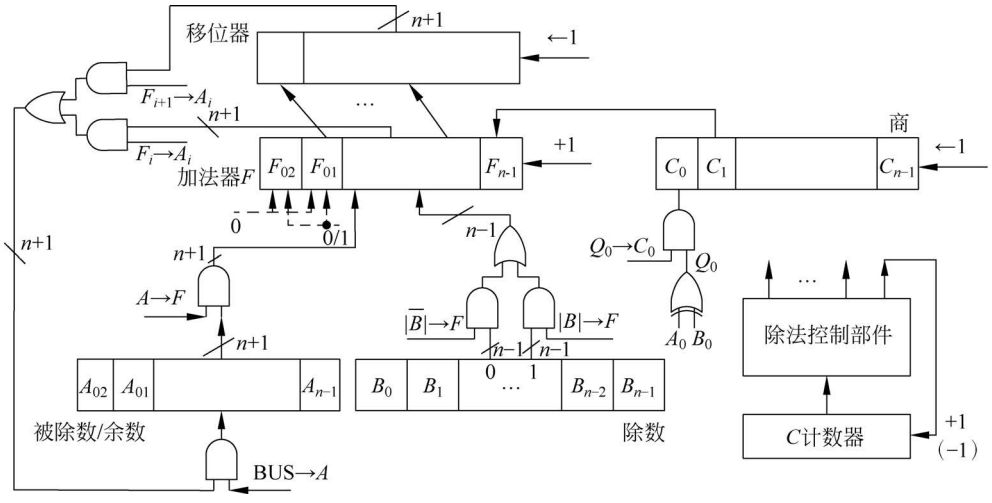


图 3.13　原码一位除加减交替法原理框图

可以发现这个图与前面学习过的定点数原码一位乘原理框图非常相似。事实上，这里的商寄存器与那里的乘数寄存器使用的是同一个寄存器，因此它也叫乘商寄存器。

另外，从图 3.13 可以看出，无论是乘法还是除法，其核心运算部件还是加法器，乘除法过程往往需要多次加减操作，这也是乘除法运算速度慢的根本原因，同时也可以深刻体会到加法器的性能很大程度上影响计算机的运算速度这一结论了。

另外，由于除法运算过程漫长，在做除法运算之前往往先对被除数和除数做“0 判断”：如果被除数为 0，结果直接为 0；如果除数为 0，结果溢出。

定点整数除法方法与定点小数相似，只是要注意：

- 要求|除数|＜|被除数|（否则，不能得到整数商），且二者均不可为 0（在除法前进行判断）。
- 被除数的位数必须是除数位数的两倍，且被除数的高 n 位比除数（n 位）小（否则溢出），若被除数为 n 位，需在前面加上 n 位 0 扩展成除数位数的两倍再进行计算，此时只需初始化时将寄存器 A 清零，而将被除数放在乘商寄存器 C 中即可。

3. 提高除法运算速度的方法

1）跳 0 跳 1 除法

跳 0 跳 1 除法是提高规格化小数绝对值相除速度的方法。

设被除数$[X]_{原}=X_0.X_1\cdots X_n$，除数$[Y]_{原}=Y_0.Y_1\cdots Y_n$，所求得的商$[C]_{原}=C_0.C_1\cdots C_n$，余数$[R]_{原}=R_0.R_1\cdots R_n$，其中，$X\cdot Y\neq 0$，$|X|<|Y|$，$Y$ 是规格化数，跳 0 跳 1 除法规则如下。

（1）符号位单独处理，$C_0=X_0\oplus Y_0$。

（2）数值部分按绝对值相除，即$|X|/|Y|$，其中设某步除法的余数为 R_i。

① 若 $R_i\geqslant 0$，且 R_i 符号位后 K 个数位均为 0，即 $R_i=0.00\cdots 01\ XX\cdots X$，小数点后连续 K 个 0，本次上商 1，再连商 $K-1$ 个 0，将余数左移 K 位，$-|Y|$，得新余数。

② 若 $R_i<0$，且 R_i 符号位后 K 个数位均为 1，即 $R_i=1.11\cdots 10XX\cdots X$，小数点后连续 K 个 1，则本次上商 0，再连商 $K-1$ 个 1，将余数左移 K 位，$+|Y|$，得新余数。

（3）两个 $n+1$ 位数（含一位符号位）相除，得 $n+1$ 位商，余数共左移 n 次。若是最后一次

移位操作，余数左移位数为 n 与累积已移位数的差。

(4) 最后余数为负时，要+|Y|恢复余数，余数符号同被除数符号，结果余数为 $2^{-n}R_n$。

例 3.32 $[X]_{原}=1.1001$，$[Y]_{原}=0.1011$，$[-|Y|]_{补}=1.0101$，求 X/Y 的值。

解：

	被除数/余数	除数/商	操作说明
	00.1001		初始状态
$+[-\|Y\|]_{补}$	11.0101		第一步做$-\|Y\|$
	11.1110	011	$R<0$，三个 1，上商 011
$\times 2^3$	11.0000		左移三次
$+\|Y\|$	00.1011		上次不够减，$+\|Y\|$
	11.1011	0110	$R<0$，一个 1，上商 0
$\times 2^1$	11.0110		左移一次
$+\|Y\|$	00.1011		上次不够减，$+\|Y\|$
	00.0001	0110100	$R>0$，三个 0，上商 100
			除法共需要左移 4 位，已移 3+1=4 位，不需要再移

$R>0$ 无须恢复，符号位=1⊕0=1，故 $X/Y=-0.1101$，$R_n=-0.0001\times 2^{-4}$。

例 3.33 $[X]_{原}=0.1010000$，$[Y]_{原}=0.1100011$，$[-|Y|]_{补}=1.0011101$，求 X/Y 的值。

解：

	被除数/余数	除数/商	操作说明
	00.1010000		初始状态
$+[-\|Y\|]_{补}$	11.0011101		第一步做$-\|Y\|$
	11.1101101	0.1	$R<0$，两个 1，上商 01
$\times 2^2$	11.0110100		左移两次
$+\|Y\|$	00.1100011		上次不够减，$+\|Y\|$
	00.0010111	0.110	$R>0$，两个 0，上商 10
$\times 2^2$	00.1011100		左移两次
$+[-\|Y\|]_{补}$	11.0011101		上次够减，$-\|Y\|$
	11.1111001	0.1100111	$R<0$，4 个 1，上商 0111
$\times 2^3$	11.1001000		除法共需要左移 7 位，已移 2+2=4 位，还需要移 3 位
$+\|Y\|$	00.1100011		恢复余数，$+\|Y\|$
	00.0101011		

符号位=0⊕0=0

$\therefore X/Y=0.1100111$，$R_n=0.0101011\times 2^{-7}$。

2) 迭代除法

事实上，计算机中使用乘法的概率远大于除法，当机器中没有除法器但是设置有快速乘法器时，可以通过乘法操作实现除法运算。

设 X 为被除数，Y 为除数，原码表示，且 X 和 Y 最高数值位均为 1(如果不是，可以通过调整比例因子使之满足要求)，则

$$\frac{X}{Y}=\frac{X\cdot F_0\cdot F_1\cdot\cdots\cdot F_i\cdot\cdots\cdot F_r}{Y\cdot F_0\cdot F_1\cdot\cdots\cdot F_i\cdot\cdots\cdot F}$$

若有一组合适的迭代系数 $F_i(0\leqslant i\leqslant r)$，迭代后使 $Y\times F_0\times F_1\times F_2\times\cdots\times F_r\to 1$，则分子

即为商：$X\times F_0\times F_1\times F_2\times\cdots\times F_r$。这样，求 X 除 Y 的商的过程即转换为先获得迭代系数 $F_i(0\leqslant i\leqslant r)$然后做乘法的过程。

令 $Y_1=Y_0\times F_0=Y\times F_0=1-\delta$，其中，$0\leqslant\delta\leqslant 2^{-m}$，$m$ 为正整数，这里 $\delta=1-Y\times F_0$ 即表示第一次迭代以后的精度，则：

$F_0=(1-\delta)/Y$，故 $X_1=X_0\times F_0=X\times F_0=X\times((1-\delta)/Y)=(X/Y)\times(1-\delta)$

令 $Y_2=Y_1\times F_1=1-\delta^2=(1+\delta)(1-\delta)$，则

$$F_1=1+\delta，故\ X_2=X_1\times F_1=(X/Y)\times(1-\delta^2)$$

令 $Y_3=Y_2\times F_2=1-\delta^4$，则

$$F_2=1+\delta^2，X_3=X_2\times F_2=(X/Y)\times(1-\delta^4)$$

以此类推，有

$$F_i=1+\delta^{2^{i-1}}\quad，i\geqslant 1$$

则

$$Y_i=Y_{i-1}\times F_{i-1}=(1-\delta^{2^{i-2}})(1+\delta^{2^{i-2}})=1-\delta^{2^{i-1}}，\quad i\geqslant 2$$

即经过 $r(r\geqslant 1)$次迭代后，此时的分母 Y_i 与 1 之间的误差仅为 $\delta^{2^{r-1}}$。

例如，取 $m=6$，即初始误差精度为 2^{-6}；第一次迭代后，误差为 2^{-6}；第二次迭代后，误差为 2^{-12}；第三次迭代后，误差为 2^{-24}；第四次迭代后，误差为 2^{-48}……

显然，经过少数几次迭代就能满足精度要求，此时，通过几次迭代乘法可得到近似的商为

$$X_r=((((X\times F_0)\times F_1)\times F_2)\times\cdots\times F_r)$$

迭代过程中的乘数即迭代系数，当 $1\leqslant i\leqslant r$ 时，$F_i=1+\delta^{2^{i-1}}=2-1+\delta^{2^{i-1}}=2-(1-\delta^{2^{i-1}})=2-Y_i$，即它等于上一轮分母迭代中间结果 Y_i 对 2 的补码(注意：在迭代系数 F_i 中小数点前面的“1”仅表示数值，不表示数的正负)，显然它是一个不小于 1 的正数，而且由于上一轮分母迭代的结果 Y_i 已经获得，因此可以容易得到本轮的迭代系数 F_1、F_2、F_3……

至于初始迭代系数 F_0，计算机通过查表方式获得。对于给定初始误差精度 δ，计算机提前算出每一个数据 Y 对应的 F_0 并将它以表格的方式存入存储器，这个表称为倒数表，每次运算时通过查表方式获得相应的初始迭代系数 F_0。

显然，初始精度越高，使商达到指定精度所需迭代次数越少；反之，所需迭代次数越多。其中，假设倒数表容量足够大，能记录所有数据的倒数 $1/Y$(即 $\delta=0$ 时的初始迭代系数 F_0)，只需做一次乘法，称一次迭代成功。

除数 Y 的位数越多，$1/Y$ 误差越小，例如，假设 Y 共 10 位，则误差为 2^{-10}；反过来，除数位数越多，倒数表容量越大、成本越高，例如，假设 Y 共 10 位，则倒数表共需 2^{10} 项。

实际计算机中，对于较长位数，往往仅存放较短位数的倒数表。例如，除数 Y 共 64 位，但计算机中仅存储 10 位的倒数表，经过第一次迭代，误差为 $|\delta|\leqslant 2^{-10}$，尚不满足要求($|\delta|\leqslant 2^{-64}$)；经过第二次迭代，收敛至 2^{-20}，还不满足；经过第三次迭代，收敛至 2^{-40}，仍不满足；但是当经过第四次迭代，则收敛至 2^{-80}，此时误差已经满足要求。

例 3.34 已知 $[X]_{原}=0.101\ 0000$，$[Y]_{原}=0.11\ 00011$，查表得到 $F_0=1.010\ 0100$(第一次迭代以后的精度 $\delta=2^{-7}$)，要求用乘法求 X/Y。其中，乘法运算每一次仅保存小数点后 7 位数据，多余部分采用“0 舍 1 入”法。

解：

$F_0=1.010\ 0100$，

$\therefore Y_1=Y_0\times F_0=Y\times F_0=0.110\ 0011\times 1.010\ 0100=0.111\ 1110\ (110\ 1100)$

即 $Y_1=0.111\ 1111$，它已经无限接近 1。

$\therefore X/Y=X\times F_0=0.101\ 0000\times 1.010\ 0100=0.110\ 0110\ (100\ 0000)$

即 $X/Y=0.110\ 0111$。

例 3.35　已知$[X]_{原}=0.101\ 0000$，$[Y]_{原}=0.110\ 0011$，查表得到 $F_0=1.00111$（第一次迭代以后的精度 $\delta=2^{-5}$，计算机中仅存储 5 位的倒数表），要求用乘法求 X/Y。其中，乘法运算每一次仅保存小数点后 7 位数据，多余部分采用“0 舍 1 入”法。

解：

$F_0=1.00111$

$\therefore Y_1=Y_0\times F_0=Y\times F_0=0.110\ 0011\times 1.00111=0.111\ 1000\ (10101)$

即 $Y_1=0.111\ 1001$

$\therefore F_1=2-Y_1=1.000\ 0111$

$\therefore Y_2=Y_1\times F_1=0.111\ 1001\times 1.000\ 0111=0.111\ 1111\ (100\ 1111)$，它已经无限接近 1。

$$\begin{aligned}\therefore X/Y &= X\times F_0\times F_1=0.101\ 0000\times 1.00111\times 1.000\ 0111\\ &=0.110\ 0010(11)\times 1.000\ 0111\\ &=0.110\ 0011\times 1.000\ 0111\\ &=0.110\ 1000\ (011\ 0101)\\ &=0.110\ 1000\end{aligned}$$

将上面两个例子与前面的例子对比，可以验证前面的结论，即初始迭代精度越高，所需要的迭代次数越少，而且计算结果与精确结果之间的误差越小（例如前一个例子初始迭代精度为 2^{-7}，完成计算仅需一次迭代，且结果误差为 0；后一个例子初始迭代精度为 2^{-5}，完成计算则需两次迭代，且结果误差为 2^{-7}）。

另外，若需计算更精确，可采用双倍字长乘运算。

3）阵列除法器

阵列除法器借鉴阵列乘法器的结构，通过分析原码加减交替法过程，以器件为代价，用资源重复换取速度提升。

阵列除法器以全加器为核心构建，其中的关键是如何实现受控加减法运算。

事实上，将全加器逻辑中的 Y_n 取反，则得到求差操作逻辑（如图 3.14 所示）。

考察求差操作时的进位，$C_n=X_n\overline{Y}_n+X_nC_{n-1}+\overline{Y}_nC_{n-1}$，如果 $X_n>Y_n$，够减，则一定有进位，此时 $C=$“1”；如果 $X_n<Y_n$，不够减，则一定无进位，此时 $C=$“0”；如果 $X_n=Y_n$，则 $C_n=C_{n-1}$。

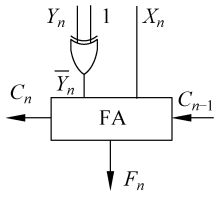


图 3.14　求差操作逻辑图

对照原码加减交替法：当上一步够减时，上商“1”，下一步做减法；当上一步不够减时，上商“0”，下一步做加法。故本行最高进位即为本位商，它可以用来决定下一行是做加法操作还是减法操作。

初始时将最低位进位 C_{n-1} 设置为“1”，第一步固定做减法。

由此得到阵列除法器原理图（见图 3.15），在阵列除法器中最高位商被上在符号位上，同样也用来进行溢出判断。

这种阵列除法器结构规整，适合大规模集成电路制造，而且速度快，一个 $2n$ 位数除以 n 位数只需一拍即可完成；但是它一共需要$(n+1)^2$ 个全加器和$(n+1)^2$ 个异或门，因此成本较高。

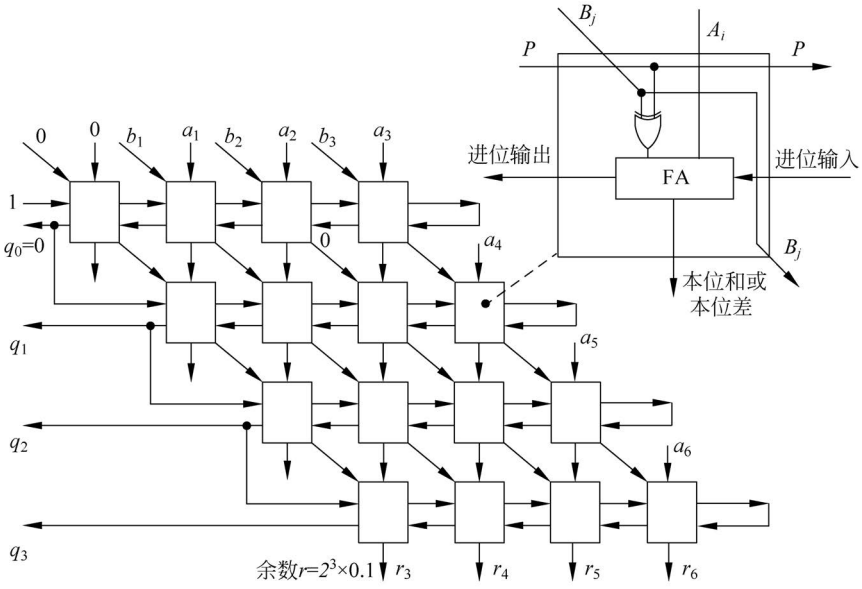


图 3.15　阵列除法器原理图

▶ 3.4.2　定点数补码除法

当除数和被除数用补码表示时，判别是否够减，要通过比较它们绝对值的大小。若两数同号，用减法；若异号，用加法。

对于判断是否够减，及确定本次上商 1 还是 0 的规则，还与结果的符号有关。当商为正时，商的每一位上的值与原码表示一致；而当商为负时，商的各位应为补码形式的值。

但是负数补码形式数值位各位的码字与真值对应位数值之间是没有直接的对应关系的，即负数直接用补码上商是比较困难的。另外，负数补码与反码之间是存在简单对应关系的，一个负数的补码等于它的反码末位加 1；而反码与原码的数值位之间恰好是按位取反的关系，因此，在进行补码除法时一般先按各位的反码值上商，最后利用反码与补码之间的关系对结果进行修正，从而得到补码商。

定点补码一位除有三种方法：恢复余数法、加减交替法、Booth 法。其中，恢复余数法与加减交替法的关系与原码除法类似。下面仍以定点小数为例，简要介绍定点补码一位除加减交替法和 Booth 法。

1. 定点补码一位除加减交替法

若 $X_{补}$、$Y_{补}$、$R_{补}$ 分别为被除数、除数和余数，定点补码一位除加减交替法规则如表 3.8 所示。

表 3.8　定点补码一位除加减交替规则

$X_{补}$ $Y_{补}$ 数符	商符	第一步操作	$R_{补}$ $Y_{补}$ 数符	上商	下一步操作
同号	0	减法	同号（够减）	1	$2[R_i]_{补}-Y_{补}$
			异号（不够减）	0	$2[R_i]_{补}+Y_{补}$
异号	1	加法	同号（不够减）	1	$2[R_i]_{补}-Y_{补}$
			异号（够减）	0	$2[R_i]_{补}+Y_{补}$

由此有定点补码一位除加减交替法规则如下。

设 X 为被除数，Y 为除数，C 为商，R 为余数。其中，X、Y 均为 n 位（含 1 位符号位）。加

法器采用双符号位，商寄存器采用单符号位。

第一步：求余数$[R_0]_{补}$。如果被除数 X、除数 Y 同号，做减法；如果被除数 X、除数 Y 异号，做加法。

第二步：根据余数上第 i 位($i\geqslant 0$)的商，并根据需要求新余数$[R_{i+1}]_{补}$。如果余数跟除数符号相同，上商“1”；如果余数跟除数符号相异，上商“0”。如果这个时候商的位数不够，需要继续上商，那么就继续求新余数$[R_{i+1}]_{补}$：如果这一步的余数跟除数符号相同，把余数左移 1 位，做减法操作得到新余数；如果这一步的余数跟除数符号相异，把余数左移 1 位，做加法操作得到新余数。

这样，连同符号位在内，一共做 n 次加或减，$n-1$ 次移位，就得到 n 位反码形式的商 $C_0.C_1C_2\cdots C_{n-1}$，下面按照补码与反码的关系对结果进行修正就可以得到补码的商了。

商的符号位就是第一次上的商，它是在运算中产生的。其中，如果正商第一次上商“1”或者负商第一次上商“0”，表明除法溢出。

商的最后一位一般采用“恒置 1”的方法，并省略最低位+1 操作，最大误差为 2^{-n}，操作简便。另外一种简便方法是“0 舍 1 入”法：多求一位商($n+2$ 位，含符号位)，然后执行类似十进制“四舍五入”的操作。

如果对商的精度要求较高，则可按规则再进行一次操作以求得商的下一数值位，然后采用以下方法对商进行处理：除不尽时，正商不必修正，负商要$+2^{-n}$ 修正；除尽时，除数为正不必修正，除数为负，商要$+2^{-n}$ 修正。

最后一步余数为 0，表明能除尽；若除法过程中任意一步余数为 0，也表明能除尽。除法除得尽时应使最后的余数为 0。

若除法过程中任意一步余数为 0，因除法运算过程需继续左移(即继续求商)，当最后一位商求出以后，若余数不为 0，应加减$[Y]_{补}$使余数为 0，或者将余数直接清为 0。

当除不尽时，正商且余除同号或负商且余除异号(简单地讲，余数与被除数同符号)，所得余数为真余数，否则为假余数，应恢复余数：如果商为正且余除异号，做$[R_{i+1}]_{补}+[Y]_{补}$恢复余数；如果商为负且余除同号，做$[R_{i+1}]_{补}-[Y]_{补}$恢复余数。

例 3.36 设$[X]_{补}=1.0111$，$[Y]_{补}=0.1101$，求$[X/Y]_{补}$。

解：

$[-Y]_{补}=11.0011$，计算过程如下。

	被除数/余数	除数/商	操作说明
	11.0111	00000	开始情形
$+[Y]_{补}$	00.1101		两数异号做加法
	00.0100	0000**1**	余除同号，商 1
←	00.1000	000**1**0	左移
$+[-Y]_{补}$	11.0011		上次商 1，做减法
	11.1011	000**10**	余除异号，商 0
	11.0110	00**10**0	左移
$+[Y]_{补}$	00.1101		上次商 0，做加法
	00.0011	00**101**	余除同号，商 1
	00.0110	0**101**0	左移
$+[-Y]_{补}$	11.0011		上次商 1，做减法
	11.1001	0**1010**	余除异号，商 0
	11.0010	**10101**	左移，**末位恒置 1**

∴$[X/Y]_{补}=1.0101$，余数不正确（还缺一位没有做）

采用末位恒置 1 法所得结果并非精确值，因此不修正商。

例 3.37 设 $X=0.0100$，$Y=-0.1000$，求$[X/Y]_{补}$。

解：

$[X]_{补}=00.0100$，$[Y]_{补}=11.1000$，$[-Y]_{补}=00.1000$

	被除数/余数	除数/商	操作说明
	00.0100	00000	开始情形
$+[Y]_{补}$	11.1000		两数异号做加法
	11.1100	0000**1**	余除同号，商 1
←	11.1000	000**1**0	左移
$+[-Y]_{补}$	00.1000		上次商 1，做减法
	00.0000	000**10**	余除异号，商 0
←	00.0000	00**10**0	左移
$+[Y]_{补}$	11.1000		上次商 0，做加法
	11.1000	00**101**	余除同号，商 1
←	11.0000	0**101**0	左移
$+[-Y]_{补}$	00.1000		上次商 1，做减法
	11.1000	0**1011**	余除同号，商 1
←	11.0000	**1011**0	左移
$+[-Y]_{补}$	00.1000		上次商 1，做减法
	11.1000	**10111**	余除同号，商 1
+	00.1000		恢复余数，也可直接清零
	00.0000		

$[X/Y]_{反}=1.0111$，∴$[X/Y]_{补}=[X/Y]_{反}+0.0001=1.1000$。余数为 0，已除尽。

例 3.38 $[X]_{补}=1.0111$，$[Y]_{补}=1.0011$，求$[X/Y]_{补}$。

解：

$[-Y]_{补}=0.1101$

	被除数/余数	除数/商	操作说明
	11.0111		开始情形
$+[-Y]_{补}$	00.1101		两数同号做减法
	00.0100	**0**	余除**异**号，商 **0**
←	00.1000		**左**移
$+[Y]_{补}$	11.0011		**加**
	11.1011	**0 1**	余除**同**号，商 **1**
←	11.0110		**左**移
$+[-Y]_{补}$	00.1101		**减**
	00.0011	**0 1 0**	余除**异**号，商 **0**
←	00.0110		**左**移
$+[Y]_{补}$	11.0011		**加**
	11.1001	**0 1 0 1**	余除**同**号，商 **1**
←	11.0010		**左**移
$+[-Y]_{补}$	00.1101		**减**
	11.1111	**0 1 0 1 1**	余除**同**号，商 **1**

正商且余除同号，末位无须修正；真余数，余数无须恢复。

∴$[X/Y]_{补}=0.1011$，[余数]$_{补}=1.1111\times2^{-4}$。

2. 定点补码一位除 Booth 法

定点补码一位除加减交替法第 1 位商(符号位)规则与其余位不同，不便于控制。定点补码一位除 Booth 法将被除数看作余数，采用统一的计算与上商方法。

定点补码一位除 Booth 法规则：若余数(初始为被除数)除数同号，上商 1，余数左移一位，减去除数；否则，上商为 0，余数左移一位，加上除数。重复上述步骤，直到求得所需位数为止。最后将商符变反，根据情况修正商的最后一位。

例 3.39 设$[X]_{补}=1.0111$，$[Y]_{补}=0.1101$，求$[X/Y]_{补}$。

解：

$[-Y]_{补}=11.0011$，计算过程如下。

	被除数/余数	除数/商	操作说明
	11.0111	**0**	余除**异**号，商 **0**
←	10.1110		**左**移
$+[Y]_{补}$	00.1101		**加**
	11.1011	**0 0**	余除**异**号，商 **0**
←	11.0110		**左**移
$+[Y]_{补}$	00.1101		**加**
	00.0011	**0 0 1**	余除**同**号，商 **1**
←	00.0110		**左**移
$+[-Y]_{补}$	11.0011		**减**
	11.1001	**0 0 1 0**	余除**异**号，商 **0**
←	11.0010		**左**移
$+[Y]_{补}$	00.1101		**加**
	11.1111	**0 0 1 0 0**	余除**异**号，商 **0**

将符号位取反，则为负商，但除不尽，末位需+1 修正，

∴$[X/Y]_{补}=1.0101$，[余数]$_{补}=1.1111\times2^{-4}$。

3.5 浮点数运算方法

浮点数分为阶码和尾数两个部分，这两个部分其实都是定点数，所以计算机中浮点数运算实际上就是通过定点数运算来完成的，其中，参与运算的两个操作数都是规格化数，最终结果也应是规格化数。

▶ 3.5.1 浮点数加减法运算

规格化浮点数的加减运算需经过 5 步完成：对阶操作、尾数运算、结果规格化、舍入操作、判断溢出。

1. 对阶操作

对阶操作就是使两个浮点数的阶码相等。

对阶的原则是：小阶向大阶对齐。所以为了完成对阶操作，先要对阶码进行减操作，通过结果的符号位判断出哪个阶码较大、哪个阶码较小。

对阶操作实际上要做的是调整阶码较小的那个浮点数的尾数，它需要将尾数右移，阶码值差多少，就需要移多少位。如果尾数是原码表示的，尾数右移符号位不动，尾数数值位最高位补 0；如果尾数是补码形式的，则尾数连同符号位右移，最高位补符号位。

需要注意的是，在实际计算过程中，因为浮点数字长有限，在尾数右移过程中，可能会出现因为低位舍入而导致精度损失。

2. 尾数运算

阶码对齐后直接对尾数运算。需要注意的是，这里尾数运算采用的是定点运算部件，但是在这个环节出现的溢出并不一定是真正的溢出，暂时不用管它。

3. 结果规格化

由于计算机中存放的都是规格化浮点数，所以在完成尾数运算以后需要对结果进行规格化处理，如尾数不溢出且非规格化则需要左规，此时将尾数左移、阶码减小；如尾数溢出则应右规，因为参与运算的数都是规格化数，所以尾数运算结果即使发生溢出，最多也只需要右规 1 位，也即尾数右移 1 位、阶码＋1。

当然，右规时也可能因为尾数低位舍入而导致精度损失。

4. 舍入操作

在对阶操作和规格化过程中可能出现数据位数超出给定位数的情况，这个时候往往需要对多余的尾数位数进行处理。处理多余尾数位数一种是截断处理，即无条件地丢掉正常尾数最低位之后的全部数值；另一种是进行舍入处理，即保留运算过程中右移出去的若干位的值，待运算完成后按某种规则用这些位的值对尾数进行修正。需要注意的是，舍入操作有时候会导致需要重新进行规格化处理，这个时候要再次规格化以后重新进行舍入操作，直到最终结果满足位数要求，并且是一个规格化数。

原码数据舍入处理方法主要有“末位置 1”法、“末位恒置 1”法和“0 舍 1 入”法。

所谓“末位置 1”法是当运算结果尾数最低位为 1，或者移出的 n 位中有为 1 的数值位时，使尾数最低位为 1，否则保持最低位不变；而“末位恒置 1”法则不同，它无论移除的数值如何，恒使最低位为 1；至于“0 舍 1 入”法则类似于“四舍五入”法，当丢失的最高位的值是 1 时，使尾数最低位加 1，否则直接舍去。

显然“0 舍 1 入”法要多进行一次加法运算。

例 3.40 求 $0.1001\times2^3+0.1001\times2$。

解：

$0.1001\times2^3+0.1001\times2 = 0.1001\times2^3+0.001001\times2^3 = 0.101101\times2^3$

如采用“0 舍 1 入”法，结果为 0.1011×2^3。

如采用“末位恒置 1”法，结果为 0.1011×2^3。

显然，本例中结果是一样的。

例 3.41 求 $0.1001\times2^3+0.1010\times2$。

解：

$0.1001\times2^3+0.1010\times2 = 0.1001\times2^3+0.001010\times2^3 = 0.101110\times2^3$

如采用“0 舍 1 入”法，结果为 0.1100×2^3。

如采用“末位恒置 1”法，结果为 0.1011×2^3。

本例中结果相差尾数末尾“1”代表的数值(本例中为 0.0001×2^3)。

补码正数跟原码是一样的，补码负数的舍入规则稍显复杂。一般地，负数补码的舍入原则

是这样的：若丢失的各位均为“0”，不舍不入；若丢失的最高位为“0”或除最高位之外均为“0”，全舍；若丢失的最高位为“1”，其余各位非全“0”，则末位+1。

5. 判断溢出

完成规格化、舍入操作后，最后需要判断结果是否溢出。对于一个规格化浮点数，只有阶码发生上溢，才算溢出，这个时候要产生溢出中断；如果阶码发生下溢，表明这个数已经很小，此时需要将运算结果置“0”。

例 3.42 设 $X=2^{010}\times 0.11011011$，$Y=2^{100}\times(-0.10101100)$，求$[X+Y]$的值。

解：

$[X]_{补}\Rightarrow 00,010;00.1101\ 1011$，$[Y]_{补}\Rightarrow 00,100;11.0101\ 0100$

(1) 对阶操作。

求阶差：$[\Delta E]_{补}=[Ex]_{补}+[-Ey]_{补}=00,010+11,100=11,110$。

$[\Delta E]_{补}$为负，阶差−2，说明 $Ex<Ey$，以 Ey 作为结果阶。

将 X 的尾数右移$|\Delta E|(=2)$位，得$[X]_{补}\Rightarrow 00,100;00.0011\ 0110\ \underline{11}$。

(2) 尾数相加。

$[M_x+M_y]_{补}=00.0011\ 0110\ \underline{11}+11.0101\ 0100=11.1000\ 1010\ \underline{11}$

$\therefore[X+Y]_{补}\Rightarrow 00,100;11.1000\ 1010\ \underline{11}$

(3) 规格化。

需要左规 1 位，$[X+Y]_{补}\Rightarrow 00,011;11.0001\ 0101\ \underline{10}$。

(4) 舍入处理：按照负数补码舍入规则，应舍去。

$[X+Y]_{补}\Rightarrow 00,011;11.0001\ 0101$

(5) 判溢出：阶码符号位为 00，不溢出。

$\therefore [X+Y]_{补}\Rightarrow 00,011;11.0001\ 0101$

即 $X+Y=2^{011}\times(-0.1110\ 1011)$。

例 3.43 假设 $X=0.1101\times 2^{10}$，$Y=-0.1111\times 2^{11}$，其中，指数和小数均为二进制真值，求 $X+Y$ 的值。(阶码 4 位(含阶符)，补码表示；尾数 6 位，补码表示，尾数符号在最高位，尾数数值 5 位。)

解：

			尾符	阶码	尾数(5 位)
	$[X]_{浮}$	⇒	0	0010	11010
	$[Y]_{浮}$	⇒	1	0011	00010
对阶得：	$[X]_{浮}$	⇒	0	0011	01101

尾数求和：$[M_x+M_y]_{补}=00.01101+11.00010=11.01111$

故	$[X]_{浮}+[Y]_{浮}$	⇒	1	0011	01111

进行规格化、舍入操作、阶码溢出判断后：$X+Y=-0.10001\times 2^{+11}$。

例 3.44 假设 $X=0.1101\times 2^{10}$，$Y=-0.1111\times 2^{11}$，其中，指数和小数均为二进制真值，求 $X-Y$ 的值。(阶码 4 位(含阶符)，补码表示；尾数 6 位，补码表示；尾数符号在最高位，尾数数值 5 位。)

解：

		尾符	阶码	尾数(5 位)
$[X]_{浮}$	⇒	0	0010	11010
$[Y]_{浮}$	⇒	1	0011	00010

对阶得：　　$[X]_{浮}$　　⇒　　0　　0011　　01101

尾数求差：$[M_x-M_y]_{补}$＝＝00.01101＋00.11110＝01.01011

规格化处理：需右规，尾数连同符号右移一位得 00.101011，阶码加 1 得 0100。

舍入采用“末位恒置 1 法”得

$[X]_{浮}-[Y]_{浮}$　⇒　　0　　0100　　10101

$\therefore X-Y=0.10101\times 2^{100}$

▶ 3.5.2　浮点数乘除法运算

两个浮点数相乘，乘积的阶码是两个乘数的阶码之和，乘积的尾数是两个乘数的尾数之积；两个浮点数相除，商的阶码是被除数阶码减去除数阶码所得的差，商的尾数是被除数尾数除以除数尾数所得的商。即浮点数乘除法运算其实是由阶码的定点加减法运算和尾数的定点乘除法运算两部分组成的。

下面先看阶码加减法运算。

当阶码 E（这里的 E 表示对应的真值）采用移码表示时，根据移码的定义：

$[E]_{移}=E+2^n \quad (-2^n\leqslant 2<2^n)$

$\therefore [Ex]_{移}+[Ey]_{移}=2^n+E_x+2^n+E_y=2^n+(2^n+(E_x+E_y))=2^n+[E_x+E_y]_{移}$

$[Ex]_{移}-[Ey]_{移}=2^n+E_x-2^n-E_y=-2^n+(2^n+(E_x-E_y))=-2^n+[E_x-E_y]_{移}$

即如果直接用移码加减运算，需要对结果的符号位进行修正。

又

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

$\therefore [E_x+E_y]_{移}=[Ex]_{移}+[Ey]_{补}$

同理，$[E_x-E_y]_{移}=[E_x]_{移}+[-E_y]_{补}$。

即和的移码等于移码与补码之和，差的移码等于移码与补码之差。

但是如果移码运算的结果溢出，上述结论不成立。

为了判断阶码加减运算是否溢出，阶码使用双符号位进行加减运算，并规定初始时最高符号位恒用 0：当结果最高符号位为“1”时表示溢出，其中，双符号位为“10”表示上溢，双符号位为“11”表示下溢；当结果最高符号位为“0”时表示无溢出，其中，双符号位为“01”表示结果为正，双符号位为“00”表示结果为负。

一般地，浮点乘法运算步骤如下。

（1）判“0”。提前判断操作数中的“0”有利于提高运算速度。

（2）阶码相加。阶码相加采用移码。

（3）尾数相乘。尾数相乘按定点小数乘法运算规则进行。

（4）规格化处理。

在进行规格化处理过程中，因为参与运算的数为规格化数，故 $|M_{积}|\geqslant 1/4$，即最多只需左规一位；只有负数补码表示且针对 $(-1.0)\times(-1.0)=+1.0$ 时才有右规问题，此时 $M_{积}=01.00\cdots00$，即右规也是最多一位。

（5）舍入处理。

(6) 判溢出。

浮点除法运算步骤如下。

(1) 预置(即判"0")。

如果除数为0,则商为∞,上溢,进行中断处理;如果除数非0但被除数为0,则商和余数均为0。

(2) 尾数调整。

浮点除法运算中,是允许$|M_x|\geqslant|M_y|$的,但是浮点数运算实际上使用的是定点运算部件,如果直接用原来的尾数进行相除,将导致除法溢出。为了保证尾数相除时不会产生除法溢出,要调整被除数,使被除数尾数高位部分的绝对值小于除数尾数的绝对值(即使$|M_x'|<|M_y|$),并同时修改阶码的值(将被除数尾数M_x右移一位得到M_x',并使被除数阶码E_x增1得到E_x')。

(3) 阶码相减。阶码相减使用移码计算。

(4) 求阶差后判溢出。

(5) 尾数相除。

进行浮点数除法运算时商的尾数一定是一个规格化数。这是因为,当尾数不需要调整时,因为$|M_x|<|M_y|$,且二者均为规格化数,即$1/2\leqslant|M_x|<1$,$1/2\leqslant|M_y|<1$,所以$1/2\leqslant|M_x|<|M_x|/|M_y|<1$,也就是说,商的尾数是规格化数;如果尾数需要调整,因为$|M_x|\geqslant|M_y|$,所以$|M_x|/|M_y|\geqslant1$,调整尾数得到M_x',其中,$|M_x'|<|M_y|$,且$|M_x'|=1/2|M_x|$;又因为调整前M_x、M_y都是规格化数,也就是$1/2\leqslant|M_x|<1$,$1/2\leqslant|M_y|<1$,所以$1/2\leqslant1/2\times|M_x|/|M_y|=|M_x'|/|M_y|<1$,即商的尾数仍然是规格化数。

例3.45 设$X=0.101101\times2^{-100}$,$Y=-0.110101\times2^{-011}$,要求阶码用移码运算,尾数用补码运算,计算$X\cdot Y$(结果保留1倍字长,舍入按补码规则进行)。

解:

$$X=0,100;0.101101,\quad Y=0,101;1.001011$$

(1) 阶码相加。

$$[E_{x\cdot y}]_{移}=[E_x]_{移}+[E_y]_{补}=00,100+11,101=00,001$$

不溢出,结果正确,$[E_{x\cdot y}]_{移}=0,001$。

(2) 尾数相乘:补码2位乘,乘数通过添加符号位凑成偶数位。

$$[2M_x]_{补}=01.011010,\quad [-M_x]_{补}=11.010011,\quad [-2M_x]_{补}=10.100110$$

操作	部分积高位	部分积低位/乘数	附加位
	000.000000	11.001011	0
$+[-M_x]_{补}$	111.010011		
	111.010011		
→2	111.110100	1111.0010	1
$+[-M_x]_{补}$	111.010011		
	111.000111		
→2	111.110001	111111.00	1
$+[M_x]_{补}$	000.101101		
	000.011110		
→2	000.000111	10111111.	0
$+[-M_x]_{补}$	111.010011		
	111.011010		

$\therefore M_{x\cdot y}=1.011010\ 101111$

(3) 规格化处理：结果是规格化数。

(4) 舍入：负数补码，舍去部分最高位为“1”、其余非全“0”，低位加 1。

$\therefore M_{x\cdot y}=1.011011$

(5) 判溢出：不溢出，故 $X\cdot Y=0,0015;1.011011$。

$\therefore X\cdot Y=-0.100101\times 2^{-111}$

例 3.46 设 $X=0.100111\times 2^{101}$，$Y=-0.101011\times 2^{011}$，要求阶码用移码运算，尾数用补码运算，计算 X/Y。

解：

$X=1,101;0.100111$，$Y=1,011;1.010101$，两数均非“0”且 $|M_x|<|M_y|$，无须调整尾数。

(1) 阶码相减。

$$[E_{x/y}]_{移}=[E_x]_{移}-[E_y]_{补}=[E_x]_{移}+[-E_y]_{补}=01,101+11,101=01,010$$

移码运算不溢出，结果正确，$[E_{x/y}]_{移}=1,010$。

(2) 尾数相除：补码加减交替法(末位恒置“1”)。

$$[-M_y]_{补}=0.101011$$

操作	被除数	商	备注
	00.100111		
$+[M_y]_{补}$	11.010101		两数异号，第一步做加
	11.111100	1.	
←1	11.111000	1.x	
$+[-M_y]_{补}$	00.101011		
	00.100011	1.0	
←1	01.000110	1.0x	
$+[M_y]_{补}$	11.010101		
	00.011011	1.00	
←1	00.110110	1.00x	
$+[M_y]_{补}$	11.010101		
	00.001011	1.000	
←1	00.010110	1.000x	
$+[M_y]_{补}$	11.010101		
	11.110011	1.0001	
←1	11.100110	1.0001x	
$+[-M_y]_{补}$	00.101011		
	00.010001	1.00010	
←1	00.100010	1.000101	末位恒置“1”

$\therefore M_{x/y}=1.000101$

即 $X/Y=1,010;1.000101$

$\therefore X/Y=-0.111011\times 2^{010}$

▶ 3.5.3 关于阶码的底为 8 或 16 的浮点数运算

阶码的底为 8 或 16 的浮点数的阶码 E 和尾数 M 仍用二进制表示，它们可以用相同位数的阶码表示更大范围的浮点数。

例如，假设浮点数阶码4位（含1位阶符）、尾数数值位7位，现有阶补尾补浮点数 $N=0\ 0010\ 1010000$。

若基为2，则 $N=0.1010\ 000\times2^{2}$

$=10.10_{(2)}$

$=2.5_{(10)}$

若基为8，则 $N=0.1010\ 000\times8^{2}$

$=0.1010\ 000\times2^{6}$

$=1010\ 00_{(2)}$

$=40_{(10)}$

若基为16，则 $N=0.1010\ 000\times16^{2}$

$=0.1010\ 000\times2^{8}$

$=160_{(10)}$

阶码的底为8或16的浮点数运算规则与阶码以2为底的基本相同，但对阶和规格化操作不同。

尾数原码表示时，阶码以8为底的规格化浮点数的尾数 M 应满足 $1/8\leqslant|M|<1$（即数值位最高3位不全为“0”），阶码以16为底的规格化浮点数的尾数 M 应满足 $1/16\leqslant|M|<1$（即数值位最高4位不全为“0”）；尾数补码表示时，阶码以8（或16）为底的规格化浮点数的尾数 M 应满足数值的最高3位（以8为底）或4位（以16为底）中至少有1位与符号位不同。

执行对阶和规格化操作时，阶码以8（或16）为底的浮点数每当阶码的值增1或减1时，尾数相应右移或左移3位（或4位）。

3.6　运算部件

定点运算部件由算术逻辑运算（ALU）部件、若干寄存器、移位电路、计数器、门电路等组成。ALU部件主要完成加减法算术运算及逻辑运算，其中还应包含快速进位电路。

ALU和寄存器同数据总线之间如何传送操作数和运算结果映射着运算部件内部的数据传送通路结构，也就是运算器的组织方式，常见的有三种内部总线结构：单总线、双总线、三总线。

在单总线结构中，ALU以及寄存器等各种部件均连接到同一组数据总线上（如图3.16所示），并且ALU配有两个临时寄存器，当需要进行ALU操作，例如，完成 $R_3=R_1+R_2$（其中 R_1、R_2、R_3 均为寄存器）操作时，需要先把寄存器 R_1 的内容经过总线送到寄存器 A，再把寄存器 R_2 的内容经过总线送到寄存器 B，然后才能够通过ALU对 A、B 中的数据进行运算，并将结果经过数据总线送回到寄存器 R_3。

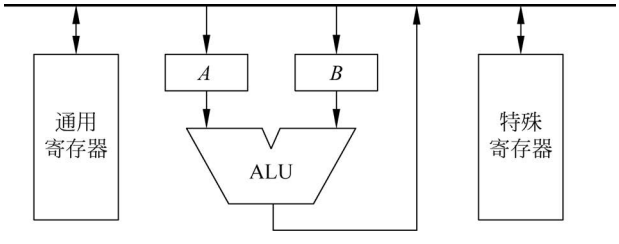


图3.16　单总线结构

双总线结构比单总线结构多了一条数据总线（如图3.17所示），并且ALU可以分别通过两条数据总线直接对通用寄存器中的内容进行操作。例如，同样是完成 $R_3=R_1+R_2$ 操作，它可以直接对寄存器 R_1 和 R_2 的内容进行运算，并将结果暂时存放在缓冲寄存器中，在下一拍

再将缓冲寄存器中的结果经过(某一条)数据总线送回到寄存器 R_3 即可。

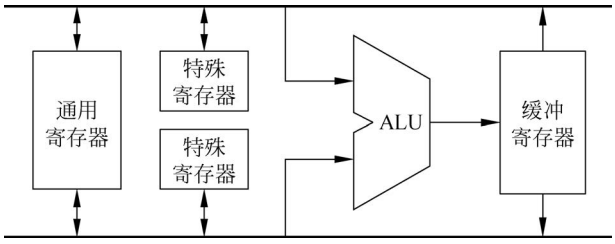


图 3.17　双总线结构

显然,双总线结构比单总线结构速度要快,成本也高一些。

三总线结构内部有三条数据总线(见图 3.18),其运算速度更快,当然成本也更高。

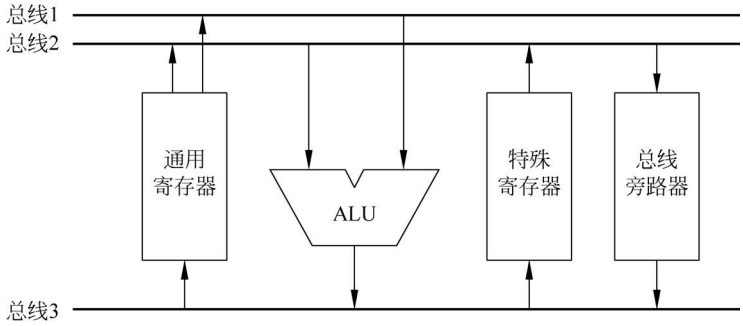


图 3.18　三总线结构

同样是完成 $R_3=R_1+R_2$ 操作,它可以直接对寄存器 R_1 和 R_2 的内容进行运算,并将结果直接送回寄存器 R_1。

具体运算器内部采用何种结构,往往需要根据性能需要及成本等综合考虑。

浮点运算部件通常由阶码运算部件和尾数运算部件组成(见图 3.19),其各自的结构与定点运算部件相似,其中,阶码部分仅执行加减法运算(包括对阶操作、阶码加减操作),尾数部分则可执行加减乘除运算。

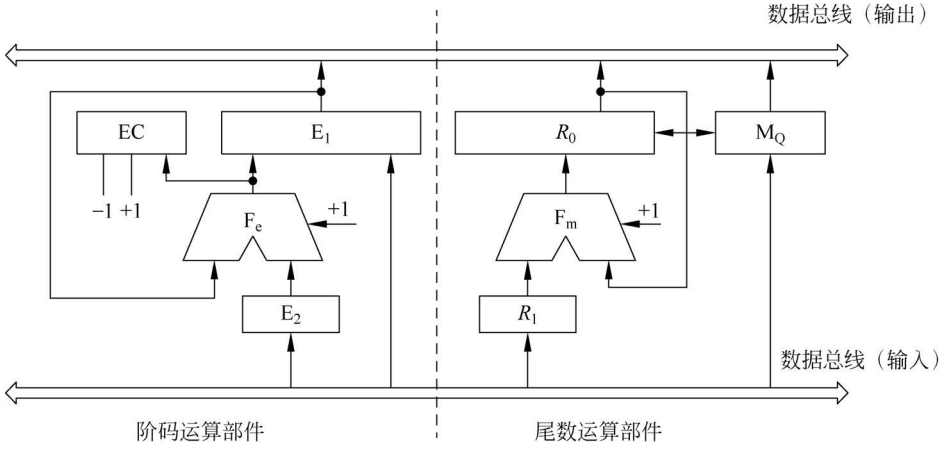


图 3.19　浮点运算部件

另外,浮点数规格化中的左规操作有时需要左移多位,为加速移位过程,有的机器还设置了可移动多位的电路。

3.7 数据校验码

计算机系统中的数据，在读/写、存储和传送的过程中可能发生错误。提高计算机的可靠性一方面可以从提高硬件可靠性入手，例如，选用更高可靠性器件、更好生产工艺，精心设计高可靠性硬件电路等；另一方面可以从编码上想办法，即采用纠错检错编码技术。

纠错检错编码技术通过增加冗余线路，在原有数据位之外再增加一到几位校验位，使新得到的码字带上某种特性；接收方通过检查码字是否仍保持有这一特性来发现是否出现错误，甚至定位并自动改正错误(即纠错)。

校验码中将由若干代码组成的一个字称为码字，例如，8421 码中 6 对应的码字为 0110、7 对应的码字为 0111；将两个码字之间对应位代码不同的个数称为距离，而将一种码制中任意两个合法码字间距离的最小值称为码距。

例如，8421 码最小距离为 1，如 0000 和 0001 之间、0110 和 0111 之间等，即 8421 码码距为 1；8421 码最大距离为 4，如 0111 和 1000 之间、1001 和 0110 之间等。

设可检测错误数为 e、可纠错误数为 $t(e>t)$，码距 D_{min} 与查错纠错的关系如下。

- 码距 $D_{min}\geqslant e+1$ 时可检测 e 个错误，其中，码距 D_{min} 为 1 时，既不能查错也不能纠错。
- 码距 $D_{min}\geqslant 2t+1$ 时可纠 t 个错误。
- 码距 $D_{min}\geqslant e+t+1$ 时可纠 t 个错误，同时检测 e 个错误。

码距选择取决于特定系统的参数、信息发生差错的概率、系统能容许的最小差错率等。显然，码距越大，查错、纠错能力越强。如表 3.9 所示为码距与纠检错能力对应关系。

表 3.9 码距与纠检错能力对应关系

码距	1	2	3	4	5	6	7
纠检错能力	无	检 1	检 2 或纠 1	检 2 纠 1	检 2 纠 2	检 3 纠 2	检 3 纠 3

当然，码距越大，数据冗余越大、编码效率越低、成本越高。

常用数据校验码包括奇偶校验码、海明校验码和循环冗余校验码。

▶ 3.7.1 奇偶校验码

奇偶校验法比较简单，计算机中应用比较广泛。它在每组数据信息上附加一位校验位，校验位的取值(0 或 1)取决于这组信息中“1”的个数和校验方式(奇或偶校验)，其中对于奇校验，这组数据加上校验码位后数据中“1”的位数应为奇数；对于偶校验，这组数据加上校验码位后数据中“1”的位数应为偶数。

一般地，校验位被固定放在一个地方。例如，所有码字的最后边，或者所有码字的最前边。如表 3.10 所示为几个奇偶校验码的例子，其中最高位为校验位。

表 3.10 奇校验与偶校验

数 据	校验码	
	奇校验	偶校验
1010 1011	**0** 1010 1011	**1** 1010 1011
0101 1111	**1** 0101 1111	**0** 0101 1111

奇偶校验法可通过一系列异或门实现，硬件电路比较简单。

为了得到校验位的值，对于偶校验方式，校验位的值由数据位按位异或得到；对于奇校验，只需把这个异或结果取反即可（如图 3.20(a)所示）。

接收方收到奇偶校验码后要通过译码对结果进行校验。不论是奇校验还是偶校验，译码的时候只需要把收到的所有码字按位异或即可（如图 3.20(b)所示为偶校验译码电路）。

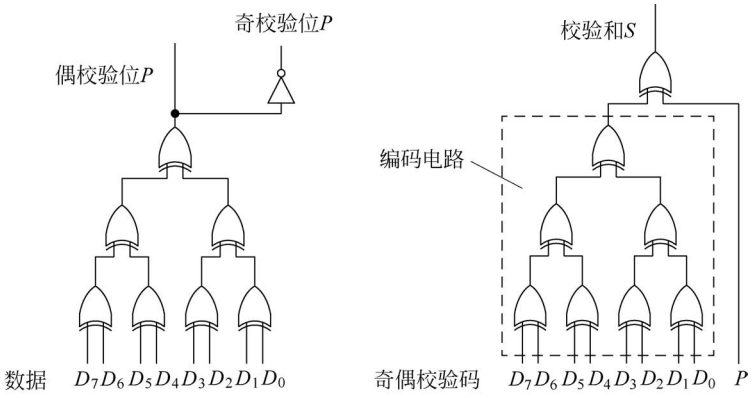


图 3.20　奇偶校验电路

我们把校验位与它参与校验的各位异或的结果称为这个校验位的校验和 S，所以，奇偶校验码的译码电路其实就是校验和形成电路。

显然，对于偶校验方式，如果收到的奇偶校验码中同时出现偶数个错，则校验和 S 必为“0”；反之，如果收到的奇偶校验码中同时出现奇数个错，则校验和 S 必为“1”。

对于奇校验方式则恰好相反，校验和 S 为“1”表示同时出现偶数个错，S 为“0”表示同时出现奇数个错。

但是在使用奇偶校验法时有一个默认的工作前提，即计算机数据传输基本上是不出错的，即使出错，顶多出现 1 位错，因此，当偶校验方式下校验和 $S=0$ 时或者奇校验方式下 $S=1$ 时，就认为出现了 0 个错（即没有出错），此时，因为校验位的位置是固定的，只需将接收到的码字去掉校验位就得到正确的数据了；而一旦发现有错，接收方将直接丢掉数据，并让发送方再发送一遍。

奇偶校验法使数据的码距为 2，可检出数据在传送过程中奇数个数位出错的情况；但是它只能发现错误，并不知错在何处，因此不能自动纠正。

由于奇偶校验码实现简便、可靠易行，所以奇偶校验码应用比较广泛，它既可以对串行数据进行校验，也可以对并行数据进行校验。

为了提升奇偶校验码纠检错能力，一种改进的方法是采用交叉奇偶校验，它对多个数据行、列同时校验，当仅某一位数据发生错误时，那么只有该数据所在的那一行和那一列发生奇偶校验错，通过行号、列号就能够定位它。由于数据是采用二进制编码的，如果能够定位具体是哪个数据发生错误，只需要将收到的数据取反就可以把它纠正过来，从而避免了数据重传。

显然，交叉奇偶校验的本质是让每个数据位参加多个校验组，当某个数据位出现错误时，可在多个校验和中有反应。

▶ 3.7.2 海明校验码

海明校验码以奇偶校验法为基础，通过采用多个校验组来提高检错能力，由美国数学家理查德·海明(Richard Hamming)于1950年提出。

海明校验码在数据中加入 r 个校验位，并把数据的每一个二进制位分配在 r 个奇偶校验组中。当某一位数值位出错后，就会引起有关的 r 个校验组的值发生变化，这样不但可发现错误，还能自动定位错误，从而为自动纠错提供依据。

被传送的 k 位数据信息加入 r 位海明校验位以后所形成的 $k+r$ 位码字称为海明码 H，海明码的位序号自右至左从1开始编号[①]，即 $H_{k+r}H_{k+r-1}\cdots H_2H_1$。

跟奇偶校验一样，海明校验码也是先对数据编码形成海明码，然后将海明码传输到对方；接收方收到海明码后进行校验，并且根据校验结果做相应的处理。

r 个校验位一共可以组成 2^r 个码字，可表示 2^r 种信息。在这些信息中，需要用1个码字表示无误信息，即谁都没出错，那么还有 2^r-1 个码字，可以具体标明 2^r-1 个错误的位置。

由于接收方收到的海明码是数据位和校验位的总和，所以在这些错误的位置中还有 r 个是属于校验位本身的，所以 r 位校验码只可以标明 2^r-1-r 个数据错误信息。

也就是说，假设被传送数据的位数为 k，它需要的校验位的个数为 r，要满足 $2^r-1-r\geqslant k$。根据这个关系，可以算出来用4个校验位能可靠传输 $2^4-1-4=11$ 位信息，而要校验32位数据则需至少6个校验位。

海明码的排列规则是：每个校验位 P_i($i\geqslant 1$)被放在海明码位序号为 2^{i-1} 的位置，从低(D_0)向高(D_{k-1})将所传输的数据依次排列海明码从右向左的其余空位。

例如，4位欲传送的数据 $D_3D_2D_1D_0$，需三个校验码 $P_3P_2P_1$，那么海明码排列次序即为 $D_3D_2D_1P_3D_0P_2P_1$；11位欲传送数据 $D_{10}D_9D_8D_7D_6D_5D_4\ D_3D_2D_1D_0$，需4个校验码 $P_4P_3P_2P_1$，相应的海明码排列次序即为 $D_{10}D_9D_8D_7D_6D_5D_4\ P_4D_3D_2D_1P_3D_0P_2P_1$。

海明码的每一位由多个校验位一起进行校验，被校验的位号等于校验它的各校验位位号和；某校验位 P_i 参与校验的所有数据位构成校验组 g_i，偶校验时将校验组 g_i 中各数据位进行异或即得到校验位 P_i 的值(奇校验时将上述值取反)。

若海明码总位数为7，包括数据4位、校验位3位(即 $D_3D_2D_1P_3D_0P_2P_1$)，它又被称为(7,4)分组码。如表3.11所示为(7,4)分组码的海明校验表。

表3.11 (7,4)分组码海明校验表

海明码位号	对应信息	参与海明码该位校验的校验位位号	参与的校验位
H_1	P_1	1	P_1
H_2	P_2	2	P_2
H_3	D_0	2,1 (3=2+1)	P_2,P_1
H_4	P_3	4	P_3
H_5	D_1	4,1 (5=4+1)	P_3,P_1
H_6	D_2	4,2 (6=4+2)	P_3,P_2
H_7	D_3	4,2,1 (7=4+2+1)	P_3,P_2,P_1

很容易有偶校验方式下校验位形成表达式：

$$P_1=D_0\oplus D_1\oplus D_3$$

① 注意，海明码码字位序号也可以从左到右从“1”开始依次编号。

$$P_2 = D_0 \oplus D_2 \oplus D_3$$
$$P_3 = D_1 \oplus D_2 \oplus D_3$$

例 3.47 写出 1010 偶校验方式下的海明码。

解：

$\because 2^r \geqslant k+r+1, k=4, \therefore r=3$

$\therefore$ 海明码排列顺序为 $D_3D_2D_1P_3D_0P_2P_1$

校验位的取值为

$$P_1 = D_0 \oplus D_1 \oplus D_3 = 0 \oplus 1 \oplus 1 = \underline{0}$$
$$P_2 = D_0 \oplus D_2 \oplus D_3 = 0 \oplus 0 \oplus 1 = \underline{1}$$
$$P_3 = D_1 \oplus D_2 \oplus D_3 = 1 \oplus 0 \oplus 1 = \underline{0}$$

$\therefore$ 所求海明码为 $D_3D_2D_1P_3D_0P_2P_1 = 101\underline{0}0\underline{1}\underline{0}$

海明码数据传送到接收方后，利用奇偶校验电路求出各校验位的校验和。当采用偶校验方式且传送数据正确时，校验和的值 S_i 分别都为“0”（奇校验则为“1”）。当校验和 S_i 不为上述值时表示发生了错误。

若仅 1 位错，对于偶校验，出错位置由各校验和 S_i 从高到低依序排列后所形成的二进制数直接指明。例如，若三个校验和 $S_3S_2S_1$ 依序排列后 $S_3S_2S_1 = (101)_2 = (5)_{10}$，则表明海明码的第 5 位（$H_5$，即数据 D_1）发生了错误，此时只需将收到的 H_5 取反，即纠正了错误。

例 3.48 已知接收到的海明码为 100 0010，当采用偶校验时，试问要求传送的信息是什么？

解：

$$S_1 = H_1 \oplus H_3 \oplus H_5 \oplus H_7 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$
$$S_2 = H_2 \oplus H_3 \oplus H_6 \oplus H_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$
$$S_3 = H_4 \oplus H_5 \oplus H_6 \oplus H_7 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$

$\because S_3S_2S_1$ 为 101，故第 5 位发生错误

$\therefore$ 正确的海明码字应该为 1010010

$\therefore$ 即所要传送的信息为 1010

偶校验方式下海明码校验电路原理图如图 3.21 所示，可以看到，当出现一位错时，相应的纠错电路其实就是一个异或门。

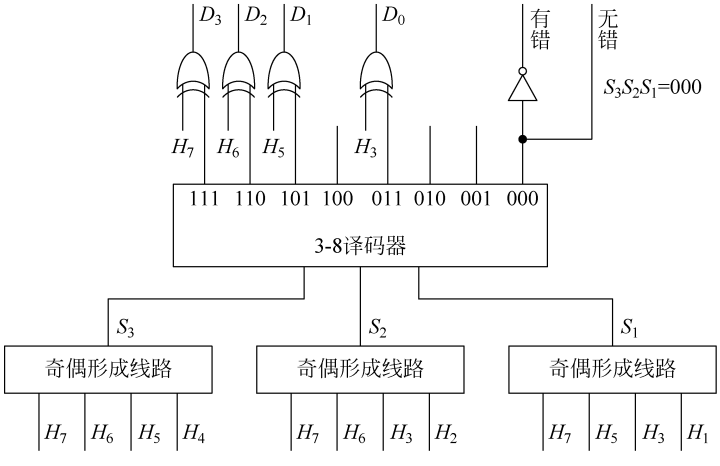


图 3.21 （偶校验）海明码校验电路原理图

上述形成的海明校验码只能纠1位错，当2位或以上同时发生错误时，是无法通过校验和获得出错位置的。若想自动纠正1位错并发现2位错，需增加1位全局校验位形成扩展海明码以增大码距，此时 r 和 k 应满足关系：$2^{r-1} \geqslant k+r$。

全局校验位总是放在最后(即海明码字的最前端)，它等于除自己外海明码字所有位的异或结果，偶校验方式下，当海明码中出现任何偶数个(含0个)错时，全局校验位一定为0；相应地，全局校验和则等于海明码字所有位的异或结果。

例如，传输8位数据时对应的扩展海明校验码与数据位的关系如表3.12所示。

表3.12　传输8位数据时的扩展海明码校验表

海明码位号	对应信息	参与校验的校验位位号	参与的校验位
H_1	P_1	1	P_1
H_2	P_2	2	P_2
H_3	D_1	2,1　(3=2+1)	P_2,P_1
H_4	P_3	4	P_3
H_5	D_2	4,1　(5=4+1)	P_3,P_1
H_6	D_3	4,2　(6=4+2)	P_3,P_2
H_7	D_4	4,2,1　(7=4+2+1)	P_3,P_2,P_1
H_8	P_4	8	P_4
H_9	D_5	8,1　(9=8+1)	P_4,P_1
H_{10}	D_6	8,2　(10=8+2)	P_4,P_2
H_{11}	D_7	8,2,1　(11=8+2+1)	P_4,P_2,P_1
H_{12}	D_8	8,4　(12=8+4)	P_4,P_3
H_{13}	P_5	(全局校验位)	(H_1～H_{12})

偶校验方式下，若全局校验和为“0”，则表示海明码出现了偶数个(包括0个)错，此时若其他校验和全为“0”，则表示海明码无错误，若其他校验和非全“0”，则表示海明码出现了两位错，但不知道具体出错位置，不可纠正；若全局校验和为“1”，则表示海明码出现了单个错，其中，若其他校验和为全“0”，表明全局校验位本身出错，若其他校验和非全“0”，则表明除全局校验位之外的某一位错，具体出错位置可通过校验和的组合得到。

偶校验方式下扩展海明码((7,4)分组码)校验电路原理图如图3.22所示。

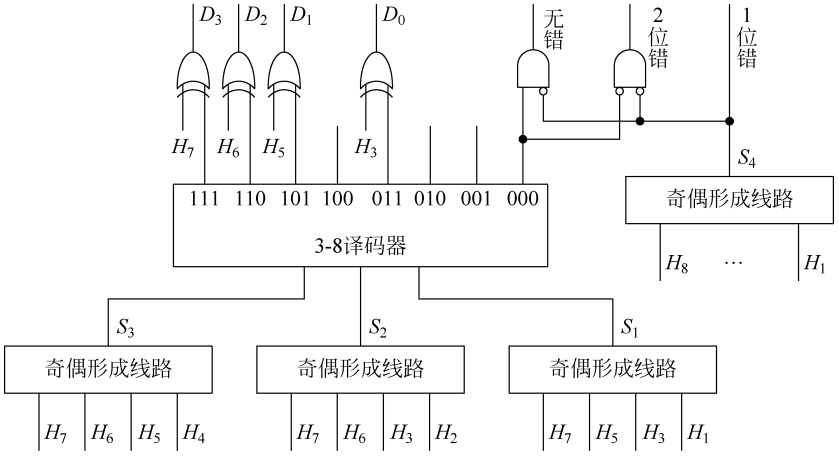


图3.22　(偶校验)扩展海明码((7,4)分组码)校验电路原理图

扩展海明码以奇偶校验法为基础，可以用在大多数不错、如果出错最多错 2 位的场景，且当仅 1 位错的时候能够自动纠错，实现简便，被广泛用在内存数据校验等并行数据校验场合。

海明码校验码需要为每一位单独设置纠错电路，且只适用出现 1 位错概率远大于 2 位同时出错概率的情形，而且认为不出现 2 位以上同时出错，在网络通信等串行数据传输中，由于瞬间干扰可能使得连续多位，或者在较短间隔内多位同时出错，这个时候就需要采用其他的数据校验码，如循环冗余校验码了。

▶ 3.7.3 循环冗余校验码

循环冗余校验码(Cyclic Redundancy Check，CRC)通过某种数学公式建立信息位和校验位之间的约定关系，该约定关系能校验传送信息的对错，并且能自动修正错误，广泛用于通信、数据压缩和磁介质存储器中。

用于构成循环冗余校验码的数学式称为生成多项式。

循环冗余校验码中的“循环”指的是这样一种循环码。循环码是指线性码中若一个 n 位编码 $V=\{v_0,v_1,v_2,\cdots,v_{n-1}\}$ 是码 C 的一个码字，那么 V 循环移动一位后的 n 位编码如 $V_1=\{v_1,v_2,\cdots,v_{n-1},v_0\}$ 也是码 C 的一个码字(**可能是原码字本身，或另外一个码字**)。

设 CRC 码是(n，k)码，即数据位 k 位，校验位(冗余位)r 位，要纠 1 位错，k 和 r 同样需满足关系：$k+r\leqslant 2^r-1$。

CRC 编码格式是在 k 位信息后加 r 位检验码，其实现方法如下。

(1) k 位数据串行移位传输，在输出过程中，用带有异或门控制的移位寄存器形成 r 个校验位的值，其中，带有异或门控制的移位寄存器实际构成模 2 除法电路，而异或门控制方法由生成多项式决定。

(2) 校验位同数据位一起传送走，接收端再使用相同的带异或门控制的移位寄存器对接收到的 $k+r$ 位的码字进行合法与出错检查，若可能则自动改错。

1. 循环冗余校验方式

要构成循环冗余校验码，并非任意数学式都能满足要求。一个生成多项式需具备以下特征：最高项、最低项的系数必须为 1；任意位发生错误都应使余数不为 0；余数补 0 左移一位继续模 2 除，应使余数循环。如要求纠 1 位错，还要求不同的某一位发生错误应使余数不同。

对于(n，k)码，为寻找生成多项式，可将 X^n+1 利用模 2 运算分解为若干质因式，然后根据码距要求选择其中的某个因式或多个因式的乘积为生成多项式。

例如，$n=7$，$X^7+1=(X+1)(X^3+X+1)(X^3+X^2+1)$，可以通过选取不同的因式或因式的乘积得到具有不同性质的 CRC 码。

(1) 取 $G(X)=X+1$，可生成(7，6)码，它可判 1 位错。

(2) 取 $G(X)=X^3+X+1$ 或 $G(X)=X^3+X^2+1$ 都生成(7，4)码，它们可检 2 位错或纠 1 位错，且 2 位错、1 位错余数均不为零，但余数有重叠。

(3) 取 $G(X)=(X+1)(X^3+X+1)=X^4+X^3+X^2+1$ 可生成一种(7，3)码(取 $G(X)=(X+1)(X^3+X^2+1)$则生成另一种(7，3)码)，它可检 2 位错并纠 1 位错，其中，2 位错、1 位错余数均不为零，且余数无重叠。

表 3.13 给出了一些常用的生成多项式。

表 3.13 一些常用的生成多项式

N	K	码距 d	$G(X)$	N	K	码距 d	$G(X)$
7	4	3	X^3+X+1	31	26	3	X^5+X^2+1
			X^3+X^2+1		21	5	$X^{10}+X^9+X^8+X^6+X^5+X^3+1$
	3	4	$X^4+X^3+X^2+1$	63	57	3	X^6+X+1
			X^4+X^2+X+1		51	5	$X^{12}+X^{10}+X^5+X^4+X^2+1$
15	11	3	X^4+X+1				
	7	5	$X^8+X^7+X^6+X^4+1$				

2. CRC 码原理

假设被传送的 k 位二进制信息位用 $M(X)$ 表示，系统选定的生成多项式用 $G(X)$ 表示，将信息位组左移 r 位，即 $M(X)\cdot 2^r$，可空出 r 位，以便拼接 r 位校验位。

将 $M(X)\cdot 2^r$ 使用模 2 除除以生成多项式 $G(X)$，所得商用 $Q(X)$ 表示，余数用 $R(X)$ 表示。即：

$$M(X)\cdot 2^r/G(X)=Q(X)+R(X)/G(X) \tag{3-11}$$

对式(3-11)两边同时乘以 $G(X)$，得：

$$M(X)\cdot 2^r=R(X)+Q(X)\cdot G(X) \tag{3-12}$$

对式(3-12)稍做变形，得：

$$M(X)\cdot 2^r+R(X)=Q(X)\cdot G(X) \quad (\mathrm{mod}\ 2) \tag{3-13}$$

$M(X)\cdot 2^r+R(X)$ 即为所求的 n 位 CRC 码，其中，余数表达式 $R(X)$ 就是校验位(r 位)。若用二进制码字表示，它相当于将余数直接“跟在”数据后即得到 CRC 码。

显然，为了得到 r 位余数，生成多项式 $G(X)$ 必须是 $r+1$ 位的。

需要注意的是，发送方利用生成多项式生成 CRC 校验码、接收方利用 CRC 对数据进行校验均使用模 2 除运算。

模 2 运算，包括模 2 加减运算、模 2 乘运算和模 2 除运算。

进行模 2 加减运算时，加减过程中不考虑进位与借位，即：

$$0\pm 0=0,\quad 0\pm 1=1,\quad 1\pm 0=1,\quad 1\pm 1=0$$

显然，模 2 加减运算本质上就是进行异或运算。

进行模 2 乘运算时，部分积之和按模 2 加法计算。

利用模 2 除运算求余数的操作是这样的：当部分余数首位为“1”时，商“1”，模 2 减除数；当部分余数首位为“0”时，商“0”，模 2 减全“0”。每求得 1 位商将使部分余数减少 1 位，当部分余数位数小于除数位数时，此时的余数即为最后的余数。

例 3.49 有一个(7,4)码，已确定生成多项式为 $G(X)=X^3+X+1=1011$，被传输的信息 $M(X)=1101$，求 $M(X)$ 的 CRC 码。

解：

将 $M(X)$ 左移 $r=n-k=3$ 位，即 $M(X)\cdot 2^r=1010\times 2^3=1101000$。

将上式采用模 2 除法，除以给定的 $G(X)=1011$：

$$1101000/1011=1111+001/1011$$

得到余数表达式 $R(X)=001$。

∴所求 CRC 码为 $M(X)\cdot 2^3+R(X)=1101000+001=1101001$。

接收方将收到的循环校验码后，再用约定的生成多项式 $G(X)$ 去除，如果检测码字无误则余数应为 0，如检测有错，则余数不为 0。

注意：余数($E(X)$)为 0 只是表明检测无错，它可能是因为确实无错，也可能是有错，但 $E(X)$恰好可被 $G(X)$整除。

一旦检测有错，不同位数出错余数不同，而且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。例如，表 3.14 即为生成多项式 $G(X)=1011$ 的(7,4)CRC 码的出错模式。

表 3.14　生成多项式 $G(X)=1011$ 的 CRC 码的出错模式

CRC	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	000	无
某一位出错	1	1	0	0	0	1	1	001	A_7
	1	1	0	0	0	0	0	010	A_6
	1	1	0	0	1	1	0	100	A_5
	1	1	0	1	0	1	0	011	A_4
	1	1	1	0	0	1	0	110	A_3
	1	0	0	0	0	1	0	111	A_2
	0	1	0	0	0	1	0	101	A_1

如果循环码有 1 位出错，用 $G(X)$作模 2 除将得到一个不为 0 的余数。例如，如果显示余数为 001，对应这个出错模式表，它指示当前码字的第 7 位 A_7，也就是最右边 1 位错了。

假定把收到的码字左移 1 位，同时继续对当前余数补 0 做模 2 除，余数变为 010，它表示是当前码字的第 6 位 A_6 错了。因为刚才把码字左移了一位，所以它其实就是刚才的第 7 位。

继续对码字左移，并且对余数补 0 做模 2 除，可以发现这个出错模式是循环且不重复的，当余数为 101 时，出错位也移到 A_1 位置，此时通过异或门将它纠正后可在下一次移位时送回 A_7。

也就是说，假定用于纠错的异或门在最高位，在求出余数不为 0 后，一边对余数补 0 继续做模 2 除，同时让被检错的校验码字循环左移；当出错位移到最高位时，通过异或门纠正，在下一次移位操作开始之前送回最高位；然后继续移位，直至移满一个循环，即得纠正后的码字。

可见，循环冗余校验码不必像海明校验那样用译码电路对每一位提供纠正条件，只需要设置一个即可，当位数增多时循环码校验能有效地降低硬件代价。

3. CRC 码编码电路

CRC 码编码可通过除法电路实现，该电路结构由生成多项式确定。例如，设 $G(X)=X^3+X+1=1011$，编码电路如图 3.23 所示。

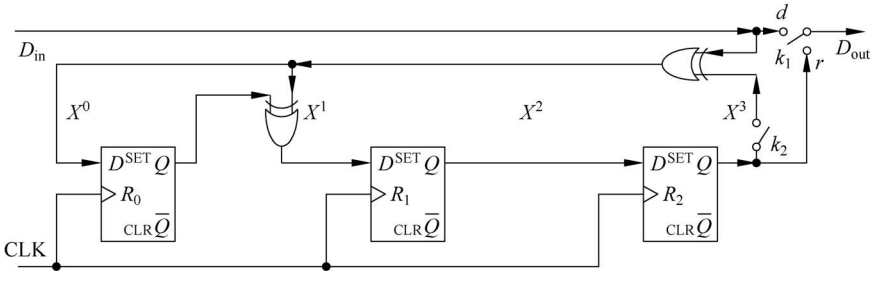


图 3.23　CRC 码编码电路

初始时单刀双掷开关 k_1 向上接通、开关 k_2 接通，同时数据由高位到低位的顺序从循环移位寄存器右侧依次输入，数据全部送入后触发器 $R_2 \sim R_0$ 的值依序排列就是校验位信息，此时将 k_1 向下接通，同时断开 k_2，从数据输入端继续输入三个“0”，使触发器 $R_2 \sim R_0$ 中的检验位信息拼接上原数据信息，最后输出端的输出结果就是 CRC 码。

这种串行编码电路结构简单，但是效率低下。事实上，模 2 除法其实就是按位异或运算，故它满足结合律，即两个数各自模 2 除生成多项式的余数异或的结果等于两数异或结果模 2 除生成多项式的余数，这样一个数对生成多项式的余数可拆解成多个数对生成多项式的余数再做异或。

例如，(7,4)码求 1010 的余数，可以将数据拆分成 1000、0010，让这两个码字分别进入除法电路(即并行求余数)，最后各结果异或即得到所求的余数。

例 3.50 有一个(7,4)码，已确定生成多项式为 $G(X)=X^3+X+1=1011$，被传输的信息 $M(X)=1101$，求 $M(X)$的 CRC 码。

解：

$$1101=1000 \oplus 0100 \oplus 0001$$

易得 1000、0100、0001 对生成多项式的余数分别是 101、111、011。

∴数据 1101 对生成多项式的余数是 $101 \oplus 111 \oplus 011=001$。

∴所求 CRC 码为 1101 001。

4. CRC 码校验能力

CRC 码校验能力特别强大，其中，具有 r 位校验位的 CRC 码检错可检测：所有奇数个错、所有 2 位错；所有突发长度小于或等于 r 的突发错误；概率为 $1-2^{-(r-1)}$ 的突发长度为 $r+1$ 的突发错误；概率为 $1-2^{-r}$ 的突发长度大于 $r+1$ 的突发错误。

所谓**突发长度是指从出错的第一位到最后一位的长度(中间不一定都出错)**。

例如，若 $r=16$，可检测所有 2 位错、所有奇数位错；所有突发长度小于或等于 16 的突发错误；约 99.997%的突发长度为 17 的突发错误；约 99.998%的突发长度大于 17 的突发错误。

事实上，在数据通信等场合，因数据量较大，往往不要求纠错；同时由于偶发因素影响可能出现突发错误，CRC 码超强检错能力恰好派上用场。

5. CRC 码的特点与应用

总结起来，CRC 码具有如下特点：检错能力强，几乎能发现所有错误；开销小，速度快，易于用硬件实现；不能定位 2 位及以上错误，用于原数据较长时只能查错，不能纠错。

上述特点使得 CRC 码被广泛应用在数据传输等各种场合，表 3.15 给出了典型的 CRC 码应用。

表 3.15 CRC 码的典型应用

名 称	生成多项式	应 用 举 例
CRC-1	$X+1$	奇偶校验
CRC-3	X^3+X+1	GSM 移动网络
CRC-4	X^4+X+1	ITU-T G.704
CRC-5-ITU	X^5+X^4+X+1	
CRC-5-EPC	X^5+X^3+1	二代 RFID
CRC-5-USB	X^5+X^2+1	USB 令牌包

续表

名　　称	生成多项式	应 用 举 例
CRC-6-GSM	$X^6+X^5+X^3+X^2+X+1$	GSM 移动网络
CRC-7	X^7+X^3+1	MMC/SD 卡
CRC-16-CCCIT	$X^{16}+X^{15}+X^2+1$	USB、bluetooth
CRC-32	$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X+1$	ZIP，RAR，IEEE1394，Ethernet，SATA，MPEG-2

表 3.16 对几种常见的数据校验码进行了简单对比。

表 3.16　几种常见数据校验码的主要特点与典型应用

校 验 方 式	主 要 特 点	典 型 应 用
奇偶校验码	可检出奇数个数位出错的情况，但不能自动纠正，也不能确定无错	
海明校验码	可检 2 纠 1，并行编码译码	内存 ECC 校验
循环冗余校验码	检错能力强，具有自动纠正单一错误能力；开销小，速度快，易于用硬件实现	数据通信、网络通信、压缩解压缩、辅存

习题

3.1　最少用几位二进制数即可表示任一 5 位长的十进制正整数？

3.2　已知二进制数 $X=0.a_1a_2a_3a_4a_5a_6$，讨论下列几种情况时 a_i 各取何值。

(1) $X>1/2$　　(2) $X\geqslant 1/8$　　(3) $1/16<X\leqslant 1/4$

3.3　写出表 3.17 中各十进制数的原码、反码、补码(用 8 位二进制表示，符号位 1 位)。

表 3.17　十进制数的表示

序　　号	真　　值	原　　码	反　　码	补　　码
(1)	−59/64			
(2)	27/128			
(3)	−127/128			
(4)	0.0			
(5)	−0.0			
(6)	−1.0			
(7)	0.25			
(8)	−0.25			
(9)	0.625			
(10)	−0.625			
(11)	1			
(12)	−1			
(13)	35			
(14)	127			
(15)	−127			
(16)	−128			

3.4　已知$[X]_{补}$，求$[X]_{原}$和 X(X 分别用二进制和十进制真值表示)。

$[X_1]_{补}=1.1100$　　$[X_2]_{补}=1.1001$　　$[X_3]_{补}=0.1110$　　$[X_4]_{补}=1.0000$

$[X_5]_{补}=1,0101$　　$[X_6]_{补}=1,1100$　　$[X_7]_{补}=0,0111$　　$[X_8]_{补}=1,0000$

3.5　已知二进制数 $X=0.1011$、$Y=-0.0101$，试求：$[X]_{补}$，$[-X]_{补}$，$[Y]_{补}$，$[-Y]_{补}$，$[X/2]_{补}$，$[X/4]_{补}$，$[2X]_{补}$，$[Y/2]_{补}$，$[Y/4]_{补}$，$[2Y]_{补}$，$[-2Y]_{补}$。

3.6　当十六进制数 9BH 和 0FFH 分别表示为原码、补码、反码、移码和无符号数时，所对应的十进制数各为多少(设机器数采用一位符号位。)

3.7　设机器数字长为 8 位(含 1 位符号位)，用补码运算规则计算下列各题。

(1) $A=9/64$，$B=-13/32$，求 $A+B$。

(2) $A=19/32$，$B=-17/128$，求 $A-B$。

(3) $A=-3/16$，$B=9/32$，求 $A+B$。

(4) $A=-87$，$B=53$，求 $A-B$。

(5) $A=115$，$B=-24$，求 $A+B$。

3.8　试通过 74181 构成一位 8421 码十进制加法器。

3.9　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。

3.10　设浮点数格式为：阶码 5 位 3.10(含 1 位阶符)，尾数 11 位(含 1 位数符)。写出 51/128、27/1024、7.375、−86.5、128.75×2^{-10} 所对应的机器数。要求如下。

(1) 阶码和尾数均为原码。

(2) 阶码和尾数均为补码。

(3) 阶码为移码，尾数为补码。

3.11　浮点数格式同上题，当阶码基值分别取 2 和 16 时：

(1) 说明 2 和 16 在浮点数中如何表示。

(2) 基值不同对浮点数什么有影响？

(3) 当阶码和尾数均用补码表示，且尾数采用规格化形式，给出两种情况下所能表示的最大正数和非零最小正数真值。

3.12　设机器字长 16 位。定点表示时，数值 15 位，符号位 1 位；浮点表示时，阶码 6 位，其中，阶符 1 位；尾数 10 位，其中，数符 1 位；阶码底为 2。试求：

(1) 定点原码整数表示时，最大正数、最小负数各是多少？

(2) 定点原码小数表示时，最大正数、最小负数各是多少？

(3) 浮点原码表示时，最大浮点数、最小浮点数各是多少？

(4) 绝对值最小的非 0 数是多少？试估算这个数对应的十进制有效数字位数。

3.13　设浮点数字长为 32 位，欲表示±60 000 间的十进制数，在保证数的最大精度条件下，除阶符、数符各取一位外，阶码和尾数各取几位？按这样分配，该浮点数溢出的条件是什么？

3.14　按下列要求设计一个尽可能短的浮点数格式(阶的底取 2)(已知：$\log_2 10=3.322$)。

(1) 数值范围为 $1.0\times10^{\pm38}$。

(2) 有效数字为十进制 7 位。

(3) 0 的机器数为全“0”。

3.15　假定一台 32 位字长的机器中带符号整数用补码表示，浮点数用 IEEE 754 标准表示，寄存器 R_1 和 R_2 的内容分别为 R_1：0000108BH，R_2：8080108BH。不同指令对寄存器进行不同的操作，因而，不同指令执行时寄存器内容对应的真值不同。假定执行下列运算指令

时，操作数为寄存器 R_1 和 R_2 的内容，则 R_1 和 R_2 中操作数的真值分别为多少？

(1) 无符号数加法指令。

(2) 带符号整数乘法指令。

(3) 单精度浮点数减法指令。

3.16 以下是一个 C 语言程序，用来计算一个数组 a 中每个元素的和。当参数 len 为 0 时，返回值应该是 0，但是在机器上执行时，却发生了存储器访问异常。请问这是什么原因造成的，并说明程序应该如何修改。

```
1  float sum_elements(float a[], unsigned len)
2  {
3      int  i;
4      float  result = 0;
5
6      for  (i = 0; i <= len - 1; i++)
7          result += a[i];
8      return result;
9  }
```

3.17 用原码一位乘、两位乘和补码一位乘(Booth 算法)、两位乘计算 $X \times Y$(前三小题为二进制数)。

(1) $X=0.110111, Y=-0.101110$。

(2) $X=-0.010111, Y=-0.010101$。

(3) $X=0.11011, Y=-0.11101$。

(4) $X=19, Y=35$。

3.18 用原码加减交替法和补码加减交替法计算 $X \div Y$(前三小题为二进制数)。

(1) $X=0.100111, Y=0.101011$。

(2) $X=-0.10101, Y=0.11011$。

(3) $X=0.10100, Y=-0.10001$。

(4) $X=13/32, Y=-27/32$。

3.19 设阶码采用补码形式、尾数采用原码形式的两个浮点数 X、Y，其中，数 X 的阶码为 0001、尾数为 0.1010；数 Y 的阶码为 1111、尾数为 0.1001。设基数为 2。

(1) 求 $X+Y$(阶码运算用补码，尾数运算用补码)。

(2) 求 $X \times Y$(阶码运算用移码，尾数运算用原码一位乘)。

(3) 求 X/Y(阶码运算用移码，尾数运算用原码加减交替法)。

3.20 设有 8 个信息位，如果采用海明校验，且要求具有检 2 纠 1 能力，至少需要设置多少个校验位？应放在哪些位置上？若 8 位信息为 01101101，采用偶校验，海明码为何值？

3.21 已知收到的海明码(采用偶校验，无全局校验位)为 1100100、1100111、1100000、1100001，请问所传输的有效信息是什么？

3.22 设有效信息为 110，试用生成多项式 $G(x)=11011$ 将其编成循环冗余校验码。

3.23 有一个(7,4)码，生成多项式 $G(x)=x^3+x+1$，写出代码 1001 的循环冗余校验码。