第3章 运算方法与运算器

计算机的基本功能是对数据信息进行各种加工处理。计算机内部对数据信息的加工可归结为两种基本运算：算术运算和逻辑运算。本章将讨论计算机中各种数据信息的加工方法，重点是四则运算的算法及其硬件实现。

3.1 运算器的设计方法

计算机具有强大的数值运算和信息处理能力，能够帮助人们完成各种复杂的工作。但作为计算机的核心部件——运算器，所具有的只是简单的算术、逻辑运算以及移位、计数等功能。因此，计算机中对数据信息加工的基本思想是：将各种复杂的运算处理分解为最基本的算术运算和逻辑运算。例如，在算术运算中，可以通过补码运算将减法化为加法；利用加减运算与移位功能的配合实现乘除运算；通过阶码与尾数的运算组合实现浮点运算。

运算器的逻辑组织结构设计通常可以分为以下层次：

(1) 根据机器的字长，将 N 个一位全加器通过加法进位链连接构成 N 位并行加法器；

(2) 利用多路选择逻辑在加法器的输入端实现多种输入组合，将加法器扩展为多功能的算术/逻辑运算部件；

(3) 根据乘除运算的算法，将加法器与移位器组合，构成定点乘法器与除法器。将计算定点整数的阶码运算器和计算定点小数的尾数运算器组合构成浮点运算器；

(4) 在算术/逻辑运算部件的基础上，配合各类相关的寄存器，构成计算机中的运算器。

3.2 定点补码加减运算

加减运算是计算机最基本的运算。定点数的加减运算可以用原码、补码、BCD 码等各种码制进行。由于补码运算可以把减法转换为加法，规则简单，易于实现，简化了加减运算的算法，所以现代计算机均采用补码进行加减运算。本节讨论定点数的补码加减运算。

3.2.1 补码加减运算的基础

1. 补码加法

补码加法所依据的基本关系是：

$$[x]_{补}+[y]_{补}=[x+y]_{补} \quad (\mathrm{mod}\ M) \tag{3-1}$$

式(3-1)中，如果 x、y 是定点小数，则 $M=2$；如果 x、y 是定点整数，则 $M=2^{n+1}$，n 为定点整数数值部分的位数。

式(3-1)说明了补码加法的规则，即两数补码之和等于两数之和的补码。下面以定点小

数为例，证明式(3-1)的正确性。

证明：设 x、y 的取值范围分别为 $-1\leqslant x<1$，$-1\leqslant y<1$；两数之和 $x+y$ 的值在正常范围之内，即 $-1\leqslant x+y<1$。

(1) 设 $x\geqslant 0$，$y\geqslant 0$。

由补码定义得：$[x]_{补}=x$，$[y]_{补}=y$，则 $[x]_{补}+[y]_{补}=x+y$。

因为 $x+y\geqslant 0$　所以 $[x+y]_{补}=x+y=[x]_{补}+[y]_{补}$。

(2) 若 $x\geqslant 0$，$y<0$，且 $|x|\geqslant|y|$。

由补码定义得：$[x]_{补}=x$，$[y]_{补}=2+y\ (\mathrm{mod}\ 2)$，则 $[x]_{补}+[y]_{补}=2+x+y$。

因为 $x+y\geqslant 0$，所以 $2+x+y\geqslant 2$，在(mod 2)的条件下，舍去模 2，得 $[x]_{补}+[y]_{补}=x+y$。

又因为 $x+y\geqslant 0$，所以 $[x+y]_{补}=x+y=[x]_{补}+[y]_{补}$。

(3) 若 $x\geqslant 0$，$y<0$，且 $|x|<|y|$。

由补码定义得：$[x]_{补}=x$，$[y]_{补}=2+y\ (\mathrm{mod}\ 2)$。

则 $[x]_{补}+[y]_{补}=2+x+y$。

因为 $|x|<|y|$，所以 $x+y<0$，$[x+y]_{补}=2+x+y=[x]_{补}+[y]_{补}(\mathrm{mod}\ 2)$。

(4) 若 $x<0$，且 $y<0$。

由补码定义得：$[x]_{补}=2+x$，$[y]_{补}=2+y$，则 $[x]_{补}+[y]_{补}=2+2+x+y$。

根据定点小数数据表示范围的要求，舍去 $[x]_{补}+[y]_{补}$ 中的模 2，得：$[x]_{补}+[y]_{补}=2+x+y$。

因为 $x<0$，$y<0$，所以 $x+y<0$，$[x+y]_{补}=2+x+y=[x]_{补}+[y]_{补}(\mathrm{mod}\ 2)$。

对于 $x<0$，$y\geqslant 0$ 的情况，可以按第(2)和第(3)步骤同样加以证明。

到此，证明了式(3-1)的正确性。

2. 补码减法

补码减法所依据的基本关系是：

$$[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}=[x-y]_{补}\quad(\mathrm{mod}\ M)\tag{3-2}$$

式(3-2)中，如果 x、y 是定点小数，则 $M=2$；如果 x、y 是定点整数，则 $M=2^{n+1}$，n 为定点整数数值部分的位数。

根据式(3-1)，可知 $[x]_{补}+[-y]_{补}=[x+(-y)]_{补}=[x-y]_{补}$，因此要证明式(3-2)成立，只需证明 $[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}$，即证明 $-[y]_{补}=[-y]_{补}$ 成立即可。

证明：因为 $[x]_{补}+[y]_{补}=[x+y]_{补}$，所以 $[y]_{补}=[x+y]_{补}-[x]_{补}$，

又因为 $[x-y]_{补}=[x+(-y)]_{补}=[x]_{补}+[-y]_{补}$，所以 $[-y]_{补}=[x-y]_{补}-[x]_{补}$，

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

由此证明了 $-[y]_{补}=[-y]_{补}$，即证明了式(3-2)成立。

根据式(3-1)和式(3-2)，可以给出补码加减运算的基本规则：

(1) 参加运算的各个操作数均以补码表示，运算结果仍以补码表示；

(2) 按二进制数逢二进一的运算规则进行运算；

(3) 符号位与数值位按同样规则一起参与运算，结果的符号位由运算得出；

(4) 进行补码加法时，将两补码数直接相加，得到两数之和的补码；进行补码减法时，将

减数变补(即由$[y]_{补}$求$[-y]_{补}$),然后与被减数相加,得到两数之差的补码。

(5) 补码总是对确定的模而言,如果运算结果超过了模(即符号位运算产生了进位),则将模自动丢掉。

例 3.1 $x=+0.1001, y=+0.0101$,求 $x\pm y$。

解:$[x]_{补}=0.1001, [y]_{补}=0.0101, [-y]_{补}=1.1011$。

$$[x+y]_{补}=[x]_{补}+[y]_{补}=0.1001+0.0101=0.1110$$

$$[x-y]_{补}=[x]_{补}+[-y]_{补}=0.1001+1.1011=0.0100$$

$x+y=0.1110, x-y=0.0100$。

$$\begin{array}{rr} 0.1001 & 0.1001 \\ +\ 0.0101 & +\ 1.1011 \\ \hline 0.1110 & \boxed{1}\ 0.0100 \end{array}$$

丢模

例 3.2 $x=-0.0110, y=-0.0011$,求 $x\pm y$。

解:$[x]_{补}=1.1010, [y]_{补}=1.1101, [-y]_{补}=0.0011$。

$$[x+y]_{补}=[x]_{补}+[y]_{补}=1.1010+1.1101=1.0111$$

$$[x-y]_{补}=[x]_{补}+[-y]_{补}=1.1010+0.0011=1.1101$$

$x+y=-0.1001, x-y=-0.0011$。

$$\begin{array}{rr} 1.1010 & 1.1010 \\ +\ 1.1101 & +\ 0.0011 \\ \hline \boxed{1}\ 1.0111 & 1.1101 \end{array}$$

丢模

例 3.3 $x=-0.1000, y=+0.0110$,求 $x\pm y$。

解:$[x]_{补}=1.1000, [y]_{补}=0.0110, [-y]_{补}=1.1010$。

$$[x+y]_{补}=[x]_{补}+[y]_{补}=1.1000+0.0110=1.1110$$

$$[x-y]_{补}=[x]_{补}+[-y]_{补}=1.1000+1.1010=1.0010$$

$x+y=-0.0010, x-y=-0.1110$。

$$\begin{array}{rr} 1.1000 & 1.1000 \\ +\ 0.0110 & +\ 1.1010 \\ \hline 1.1110 & \boxed{1}\ 1.0010 \end{array}$$

丢模

例 3.4 $x=+0.1010, y=+0.1001$,求 $x+y$。

解:$[x]_{补}=0.1010, [y]_{补}=0.1001$,

$$[x+y]_{补}=[x]_{补}+[y]_{补}=0.1010+0.1001$$

根据加法算式可以看出,两个正数相加,得到的结果的符号却为负,显然结果出错。

$$\begin{array}{rr} & 0.1010 \\ +\ & 0.1001 \\ \hline & 1.0011 \end{array}$$

例 3.5　$x=-0.1101$，$y=-0.1011$，求 $x+y$。

解：$[x]_{补}=1.0011$，$[y]_{补}=1.0101$，

$$[x+y]_{补}=[x]_{补}+[y]_{补}=1.0011+1.0101$$

```
   1.0011
+  1.0101
----------
 [1]0.1000
   ↑
  丢模
```

根据加法算式可以看出，两个负数相加，得到的结果的符号却为正，显然结果出错。

从例 3.4 和例 3.5 的计算结果可以发现补码加减运算出现了错误。出错的原因是运算结果超出了机器所能表示的数据范围，数值位侵占了符号位，正确符号被挤走了。如字长 5 位的定点小数中，最大正数的补码表示为 0.1111，例 3.4 中的 0.1010＋0.1001＝1.0011 结果值超过了 0.1111，最高数值位产生的数值进位向前侵占了符号位，于是出现了错误。同样，例 3.5 中出现的问题是运算结果超出了机器所能表示的最小负数，这种情况称为溢出。如果两个正数相加的结果超出机器所能表示的最大正数，称为正溢出。如果两个负数相加的结果小于机器所能表示的最小负数，称为负溢出。出现溢出后，机器将无法正确表示运算结果，因此计算机在运算过程中必须正确判别溢出并及时加以处理。

3.2.2　溢出判断与变形补码

设参加运算的操作数为

$$[x]_{补}=x_f.x_1x_2\cdots x_n \quad [y]_{补}=y_f.y_1y_2\cdots y_n$$

$[x]_{补}+[y]_{补}$ 的和为：

$$[s]_{补}=s_f.s_1s_2\cdots s_n$$

发生溢出时判别信号为：

$$OVR=1$$

常用的溢出判别方法有以下 3 种。

1. 根据两个操作数的符号与结果的符号判别溢出

因为参加运算的数都是定点数，只有两数同号相加时才可能出现溢出。所以，可以利用参加运算的两个操作数的符号与结果的符号的异同来判断是否发生了溢出，判断的条件为：

$$OVR=\bar{x}_f\,\bar{y}_f s_f+x_f y_f\,\bar{s}_f=(x_f\oplus s_f)(y_f\oplus s_f) \tag{3-3}$$

即如果 x_f 和 y_f 均与 s_f 不同，则产生溢出，OVR＝1。

如在例 3.4 中，$x_f=0$，$y_f=0$，$s_f=1$，由于 OVR＝($x_f\oplus s_f$) ($y_f\oplus s_f$)＝(0⊕1) (0⊕1)＝1，因此可以判定运算结果产生了溢出。又因为操作数 x、y 均为正数，所以产生的是正溢出。相应地，例 3.5 中运算结果产生的是负溢出。

2. 根据两数相加时产生的进位判别溢出

从例 3.1～例 3.3 可以看到，当补码运算的结果正确且溢出时，两数相加在符号位上产生的进位和数值最高位产生的进位情况是一致的。而从例 3.4 和例 3.5 可以看到，当补码运算的结果出现溢出时，两数相加在符号位上产生的进位和数值最高位产生的进位情况是不相同的。这样可利用两数相加产生的最高位(符号位)和次高位(数值最高位)进位进行溢出判断。

```
   0.1010
+  0.1001
----------
   1.0011
   ∨∨
C_f=0  C_1=1
```

设 C_f 为符号位上产生的进位，C_1 为最高数值位上产生的进位，则溢出的条件为：

$$OVR = C_f \oplus C_1 \tag{3-4}$$

如在例 3.4 中，计算 $[x+y]_{补}=[x]_{补}+[y]_{补}=0.1010+0.1001$ 时，由于 $C_f=0$，$C_1=1$，使得 $OVR=C_f \oplus C_1=0 \oplus 1=1$，所以可以判断运算结果出现了溢出。

3. 采用变形补码进行运算

如上所述，补码加减运算中使用了一个符号位，溢出时正确的符号位将被数值侵占，即被溢出的数值挤掉了，符号位含义发生混乱。因此，如果将符号位扩展为两位，这样在进行运算时，即使因为出现溢出，数值侵占了一个符号位，仍能保持最左边的符号是正确的。这种采用两个符号位表示的补码称为变形补码或双符号位补码。

定点小数的变形补码定义为：

$$[x]_{变形补} = \begin{cases} x, & 0 \leqslant x < 1 \\ 4+x, & -1 \leqslant x < 0 \end{cases} \pmod 4 \tag{3-5}$$

根据式(3-5)可知，定点小数的变形补码是以 4 为模的，所以也称其为模 4 补码。

定点整数的变形补码定义为：

$$[x]_{变形补} = \begin{cases} x, & 0 \leqslant x < 2^n \\ 2^{n+2}+x, & -2^n \leqslant x < 0 \end{cases} \pmod{2^{n+2}} \tag{3-6}$$

例 3.6 已知 x 的真值，求 x 对应的变形补码。

① $x=+0.1101$ ② $x=-0.1011$ ③ $x=+1101$ ④ $x=-1011$

解：① 因为 $x=+0.1101 \geqslant 0$，所以 $[x]_{变形补}=00.1101$

② 因为 $x=-0.1011<0$，所以 $[x]_{变形补}=4+(-0.1011)=11.0101$

③ 因为 $x=+1101 \geqslant 0$，所以 $[x]_{变形补}=001101$

④ 因为 $x=-1011<0$，所以 $[x]_{变形补}=2^{4+2}+(-1011)=110101$

与普通补码加减运算相同，变形补码加减运算时，两个符号位与数值部分一起参加运算。

例 3.7 利用变形补码求 $x+y$。

① $x=+0.1001$，$y=+0.0101$

② $x=-0.0110$，$y=-0.0011$

③ $x=+0.1010$，$y=+0.1001$

④ $x=-0.1101$，$y=-0.1011$

解：① $[x+y]_{变形补}=[x]_{变形补}+[y]_{变形补}=00.1001+00.0101=00.1110$；

② $[x+y]_{变形补}=[x]_{变形补}+[y]_{变形补}=11.1010+11.1101=11.0111$；

③ $[x+y]_{变形补}=[x]_{变形补}+[y]_{变形补}=00.1010+00.1001$，根据加法算式，相加结果出现了正溢出，结果的变形补码中，两个符号位不相同；

④ $[x+y]_{变形补}=[x]_{变形补}+[y]_{变形补}=11.0011+11.0101$，根据加法算式，相加结果出现了负溢出，结果的变形补码中，两个符号位不相同。

00.1001	11.1010	00.1010	11.0011
+ 00.0101	+ 11.1101	+ 00.1001	+ 11.0101
00.1110	1 11.0111	01.0011	1 10.1000
	↑ 丢模		↑ 丢模

设 s_{f1}、s_{f2}分别为结果的符号位，s_{f1}定义为第一符号位，s_{f2}定义为第二符号位。根据例 3.7 的结果，可以得出采用变形补码进行运算时 s_{f1}、s_{f2}的含义为：

$s_{f1}s_{f2}=00$，表示结果为正数，无溢出；

$s_{f1}s_{f2}=11$，表示结果为负数，无溢出；

$s_{f1}s_{f2}=01$，表示结果为正溢出；

$s_{f1}s_{f2}=10$，表示结果为负溢出。

由此可见，如果 s_{f1}、s_{f2}不一致，就表示运算结果产生了溢出。因此，采用变形补码进行运算时，结果是否溢出的判断条件是：

$$\mathrm{OVR}=\bar{s}_{f1}s_{f2}+s_{f1}\bar{s}_{f2}=s_{f1}\oplus s_{f2} \tag{3-7}$$

如在例 3.7 的第③小题中，$\mathrm{OVR}=s_{f1}\oplus s_{f2}=0\oplus 1=1$，表示运算结果出现了溢出，且因为 $s_{f1}s_{f2}=01$，所以表示结果为正溢出。

分析 s_{f1}、s_{f2}的含义还可知，无论运算结果是否产生溢出，第一符号位 s_{f1}始终指示结果的正确的正负符号。

需要说明的是，采用变形补码时，因为任何正确的数的两个符号位总是相同的，所以数据在寄存器或主存中保存时，只需保存一位符号位即可。然而，由于将操作数送到加法器中进行运算时，需要采用双符号位，所以在实际运算电路中，必须将一位符号的值同时送到加法器的两个符号位的输入端。

3.2.3　算术逻辑运算部件

运算器的基本功能是进行算术逻辑运算，其最基本也是最核心的部件是加法器。在加法器的输入端加入多种输入控制功能，就能将加法器扩展为多功能的算术/逻辑运算部件。

1. 补码加减运算的逻辑实现

设参加运算的操作数为 A、B，根据补码加减运算的规则，可知：

$$[A]_{补}+[B]_{补}=[A+B]_{补}$$

$$[A]_{补}-[B]_{补}=[A]_{补}+[-B]_{补}=[A]_{补}+[\bar{B}]_{补}+1=[A-B]_{补} \tag{3-8}$$

根据式(3-8)，在加法器的输入端增加控制信号 M，控制实现加法和减法。图 3-1 显示了采用串行进位的补码加减运算逻辑电路。

在图 3-1 中，当 $M=0$ 时，操作数 B 与 M 异或后得到的仍是 B 的原变量，因此加法器的运算结果为 $F=[A]_{补}+[B]_{补}=[A+B]_{补}$；当 $M=1$ 时，操作数 B 与 M 异或后得到的是 B 的反变量，由于 $M=1$ 又使 $C_0=1$，实现了最低位加 1 的功能，所以加法器的运算结果为 $F=[A]_{补}+[\bar{B}]_{补}+1=[A-B]_{补}$。

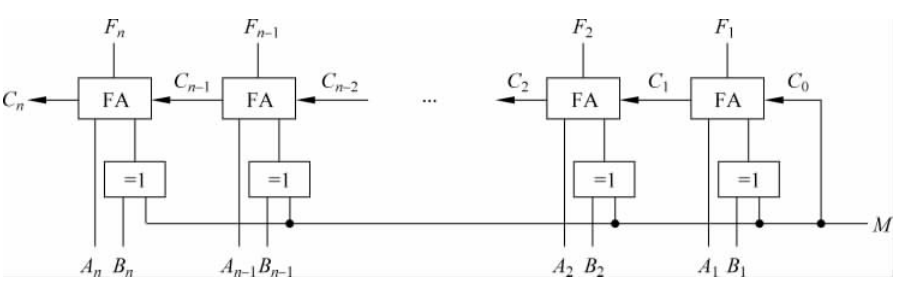


图 3-1　采用串行进位的补码加减运算逻辑电路

在实际的运算器中,参加运算的操作数和运算结果通常都存放在寄存器中,控制器通过对指令译码得到控制信号,控制将操作数输入加法器及将运算结果写回寄存器。图 3-2 显示了带有寄存器的实现 $A \leftarrow (A) \pm (B)$ 的补码加减运算逻辑电路。

图 3-2 中寄存器 A、B 分别存放参加运算的两个补码操作数,运算结束后,结果写回寄存器 A 保存。运算控制信号逻辑如下:

$F \leftarrow A$ = ADD + SUB, $F \leftarrow A$ 信号控制将寄存器 A 的正向信号输入加法器 F 的输入端;

$F \leftarrow B$ = ADD, $F \leftarrow B$ 信号控制将寄存器 B 的正向信号输入加法器 F 的输入端;

$F \leftarrow \overline{B}$ = SUB, $F \leftarrow \overline{B}$ 信号控制将寄存器 B 的反向信号输入加法器 F 的输入端;

$C_0 \leftarrow 1$ = SUB, $C_0 \leftarrow 1$ 信号控制使加法器 F 的最低位进位 $C_0 = 1$;

$A \leftarrow F$ = ADD + SUB, $A \leftarrow F$ 信号控制使加法器 F 的运算结果写入寄存器 A。

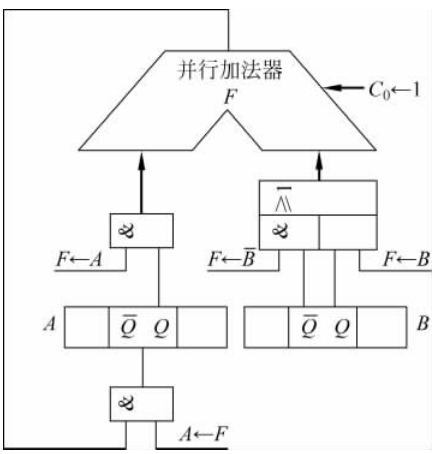


图 3-2 实现补码加减运算的逻辑电路

其中,ADD 和 SUB 分别为控制器根据加法指令和减法指令译码后得到的控制电位信号。

2. 算术逻辑运算部件举例

除了加减运算,运算器还需要完成其他算术逻辑运算,在加法器的输入端加以多种输入控制,就可以将加法器的功能进行扩展。算术逻辑运算单元(简称 ALU)就是一种以加法器为基础的多功能组合逻辑电路。其基本设计思想是:在加法器的输入端加入一个函数发生器,这个函数发生器可以在多个控制信号的控制下,为加法器提供不同的输入函数,从而构成一个具有较完善的算术逻辑运算功能的运算部件。下面以中规模集成电路芯片 SN74181 为例,说明 ALU 组件的工作原理。

SN74181 是一个 4 位 ALU 组件,它可以实现 16 种算术运算功能和 16 种逻辑运算功能,其具体功能由 $S_3S_2S_1S_0$ 和 M 信号控制实现。

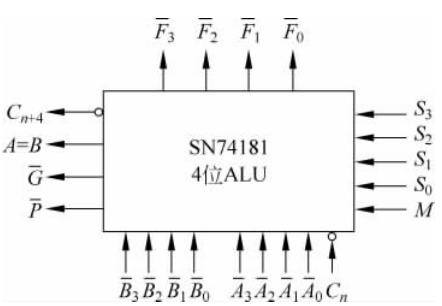


图 3-3 SN74181 的外部特性图

SN74181 有正逻辑和负逻辑两种芯片,图 3-3 给出了采用负逻辑方式工作的 SN74181 芯片的外部特性。其中,$A_{3\sim0}$、$B_{3\sim0}$ 为参加运算的两组 4 位操作数;C_n 为低位来的进位;$F_{3\sim0}$ 为输出的运算结果;C_{n+4} 为向高位的进位;G 为小组本地进位;P 为小组传递函数;$A=B$ 用于输出两个操作数的相等情况,如果将多片 SN74181 的"$A=B$"端按"与"逻辑连接,就可以检测两个字长超过 4 位的操作数的相等情况。在控制信号中,$S_3S_2S_1S_0$ 用于控制产生 16 种不同的逻辑函数;M 用于控制芯片执行算术运算还是逻辑运算,若 $M=0$,则允许位间进位,执行算术运算;若 $M=1$,则封锁位间进位,执行逻辑运算。

表 3-1 列出了采用负逻辑方式时 SN74181 完成的功能。表中,"加"是指算术加,而"+"是指逻辑加,即"或"运算。进行算术加运算时,最低位的进位为 0。如果要实现减法,可以用表中的"A 减 B 减 1"功能,并使最低位的进位 C_n 为 1,即通过最低进位实现的加 1 完成"A 减 B"。

表 3-1　SN74181 ALU 的功能表

工作方式选择 $S_3S_2S_1S_0$	F 的输出功能(负逻辑)	
	逻辑运算 $M=1$	算术运算 $M=0, C_n=0$
0 0 0 0	$\overline{A}$	A 减 1
0 0 0 1	$\overline{AB}$	AB 减 1
0 0 1 0	$\overline{A}+B$	$A\overline{B}$ 减 1
0 0 1 1	逻辑 1	全 1
0 1 0 0	$\overline{A+B}$	A 加$(A+\overline{B})$
0 1 0 1	$\overline{B}$	AB 加$(A+\overline{B})$
0 1 1 0	$\overline{A}\oplus\overline{B}$	A 减 B 减 1
0 1 1 1	$A+\overline{B}$	$A+\overline{B}$
1 0 0 0	$\overline{A}B$	A 加$(A+B)$
1 0 0 1	$A\oplus B$	A 加 B
1 0 1 0	B	$A\overline{B}$ 加$(A+B)$
1 0 1 1	$A+B$	$A+B$
1 1 0 0	逻辑 0	0
1 1 0 1	$A\overline{B}$	AB 加 A
1 1 1 0	AB	$A\overline{B}$ 加 A
1 1 1 1	A	A

注：1＝高电平，0＝低电平。

将多片 SN74181 组合，可以构成更多位数的 ALU。例如，从低位到高位依次将 4 片 SN74181 的 C_{n+4} 与高位芯片的 C_{-1} 相连，就可以构成 16 位的 ALU。如果需要进一步提高进位速度，可以采用与 SN74181 配套的并行进位链芯片 SN74182 组成快速的并行加法器。图 3-4 给出了利用 4 片 SN74181 和 1 片 SN74182 构成的 16 位快速并行加法器的例子。

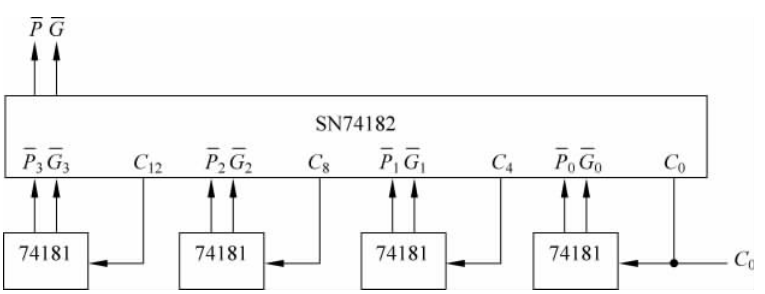


图 3-4　16 位快速并行加法器

3.3　定点乘法运算

乘除运算是经常遇到的基本算术运算。计算机中实现乘除运算通常采用以下 3 种方式：

(1) 利用乘除运算子程序。

基本思想是：采用软件实现乘除运算，即利用计算机的加减运算指令、移位指令及控制类

指令组成循环程序,通过运算器中的加法器、移位器等基本部件的反复操作,得到运算结果。这种方式所需硬件简单,但实现速度较慢,主要用于早期的小、微型机上。

(2) 在加法器的基础上增加左、右移位及计数器等逻辑线路构成乘除运算部件。

基本思想是:采用硬件实现乘除运算。在采用乘除运算部件实现乘除运算的计算机中,设有乘除运算指令,用户执行乘除指令即可进行乘除运算。这种方式实现乘除运算的速度比第一种方式快,但需要根据一定的乘除算法构建乘除运算部件,所需的硬件线路较复杂。

(3) 设置专用的阵列乘除运算器。

由于方式(2)在实现乘除运算时,通常是在一个加法器的基础上,通过对操作数多次串行地进行运算、移位得到运算结果的,所以依然需要较多的运算时间。随着大规模集成电路技术的发展带来的硬件成本的降低,出现了专用的阵列乘除运算器。阵列乘除运算器将多个加减运算部件排成乘除运算阵列,依靠硬件资源的重复设置,同时进行多位乘除运算,赢得了乘除运算的高速度。

本书主要介绍乘除运算后两种方法的算法及硬件实现。

3.3.1 原码乘法运算

原码乘法的算法基本是从二进制乘法的手算方法演化而来的。在定点机中,两个数的原码乘法运算的实现包括两个部分:乘积的符号处理和两数绝对值相乘。

设:被乘数$[x]_{原}=x_f.x_1x_2\cdots x_n$

乘数$[y]_{原}=y_f.y_1y_2\cdots y_n$

乘积$[z]_{原}=[x]_{原}\times[y]_{原}=[x\times y]_{原}=z_f.z_1z_2\cdots z_n$

根据同号相乘,乘积为正;异号相乘,乘积为负的原则,可得符号运算的真值表如表 3-2 所示。

表 3-2 符号运算真值表

x_f	y_f	z_f
0	0	0
0	1	1
1	0	1
1	1	0

根据真值表,可得乘积符号运算的逻辑表达式为 $z_f=x_f\oplus y_f$。由于乘积的符号单独进行处理,所以乘法运算中实际需要解决的问题是两个数的绝对值相乘或者说是两个正数相乘的算法与实现。我们先分析一下乘法的手算过程。

例 3.8 设 $x=0.x_1x_2x_3x_4=0.1101$,$y=0.y_1y_2y_3y_4=0.1011$,求 $x\times y$。

解:根据二进制乘法规律,可得 $x\times y$ 的手算过程如下:

```
      0.1101
×     0.1011
-------------------------------------------
      1101   因为 y4=1 所以得部分积为x
     1101    因为 y3=1 所以得部分积为x
    0000     因为 y2=0 所以得部分积为0
   1101      因为 y1=1 所以得部分积为x
-------------------------------------------
0.10001111   将所有部分积相加,得到最后的乘积
```

得:$x\times y=0.10001111$。

分析例 3.8 可以发现,在乘法的手算过程中,是将乘数一位一位地与被乘数相乘,当乘数位 $y_i=1$ 时,与被乘数 x 相乘所得的部分乘积就是 x;当 $y_i=0$ 时,与 x 相乘所得的部分乘积

就是 0；由于相乘的乘数的位权是逐次递增的，所以每次得到的部分积都需要在上次部分积的基础上左移一位。将各次相乘得到的部分积相加，即可得到最后的乘积。可以在计算机中用硬件模仿手算运算过程实现原码乘法。但是仔细分析后可知，在例 3.8 中两个 4 位数相乘，共得到 4 个部分积，相加后得到的最后的乘积为 8 位，因此具体实现时，需使用 8 位加法器对 4 个部分积进行相加。推而广之可知，两个 n 位数相乘共得到 n 个部分积，需要 n 个寄存器保存 n 个部分积；同时由于乘积为 $2n$ 位，所以需用 $2n$ 位加法器进行相加运算。显然，模仿手算运算所需硬件太多。在计算机中实现乘法时，必须对算法加以改进。

1. 原码一位乘法

在原码一位乘法中，参加运算的被乘数和乘数均用原码表示；运算时符号位单独处理，被乘数与乘数的绝对值相乘；所得的积也采用原码表示。

设参加运算的被乘数为 $x=0.x_1x_2\cdots x_n$，乘数为 $y=0.y_1y_2\cdots y_n$，则有：

$$\begin{aligned} x\times y &= x\times 0.y_1y_2\cdots y_n \\ &= x\times(2^{-1}y_1+2^{-2}y_2+\cdots+2^{-(n-1)}y_{n-1}+2^{-n}y_n) \\ &= 2^{-1}xy_1+2^{-2}xy_2+\cdots+2^{-(n-1)}xy_{n-1}+2^{-n}xy_n \\ &= 2^{-1}\{2^{-1}[2^{-1}\cdots(2^{-1}<0+xy_n>+xy_{n-1})+\cdots+xy_2]+xy_1\} \end{aligned} \tag{3-9}$$

式(3-9)的运算过程可以用式(3-10)的递推公式表示：

$$\begin{aligned} z_0 &= 0 \quad (\text{初始部分积 } z_0 \text{ 为 } 0) \\ z_1 &= 2^{-1}(z_0+xy_n) \\ z_2 &= 2^{-1}(z_1+xy_{n-1}) \\ &\cdots\cdots \\ z_i &= 2^{-1}(z_{i-1}+xy_{n-i+1}) \\ &\cdots\cdots \\ z_n &= 2^{-1}(z_{n-1}+xy_1)=x\times y \end{aligned} \tag{3-10}$$

其中，$z_0,z_1,\cdots,z_n$称为部分积。从式(3-10)可以看出，可以把乘法转换为一系列加法与移位操作。考虑了符号的处理后，可得出原码一位乘法的算法如下：

(1) 积的符号单独按两个操作数的符号模 2 加(异或)得到，即 $z_f=x_f\oplus y_f$。用被乘数和乘数的数值部分进行运算。

(2) 以乘数的最低位作为乘法判别位，若判别位为 1，则在前次部分积(初始部分积为 0)上加上被乘数，然后连同乘数一起右移一位；若判别位为 0，则在前次部分积上加 0(或不加)，然后连同乘数一起右移一位。

(3) 重复第(2)步直到运算 n 次为止(n 为乘数数值部分的长度)。

(4) 将乘积的符号与数值部分结合，即可得到最终结果。

例 3.9　根据原码一位乘法的算法计算例 3.8。

解：$[x]_{原}=0.1101$，$[y]_{原}=1.1011$，乘积$[z]_{原}=[x\times y]_{原}$。

① 符号位单独处理得$[z]_{原}$的符号 $z_f=0\oplus 1=1$。

② 将被乘数和乘数的绝对值的数值部分相乘。

$$|x|=0.1101,\quad |y|=0.1011$$

数值部分为 4 位，共需运算 4 次。运算过程如下：

	部分积	乘数 y_n	说明
	0.0000	1011	初始部分积 $z_0=0$
+	0.1101		因为乘数$y_n=1$，所以加x
	0.1101		
→	0.0110	1101	部分积与乘数同时右移一位
+	0.1101		因为乘数$y_n=1$，所以加x
	1.0011		
→	0.1001	1110	部分积与乘数同时右移一位
+	0.0000		因为乘数$y_n=0$，所以加0
	0.1001		
→	0.0100	1111	部分积与乘数同时右移一位
+	0.1101		因为乘数$y_n=1$，所以加x
	1.0001		
→	0.1000	1111	部分积与乘数同时右移一位
			运算了4次，计算结束

得$|x\times y|=0.10001111$，加上符号部分得$[x\times y]_{原}=1.10001111$，即 $x\times y=-0.10001111$。

比较例 3.8 和例 3.9 可见，采用原码一位乘法的算法所得的结果与手算的结果是一致的。

分析例 3.9 的运算过程，可知在用硬件实现原码一位乘法算法时，只需用一个寄存器保存部分积，并且只需用一个 n 位加法器即可完成运算，因此该算法适合于乘法的硬件实现。实现原码一位乘法算法的硬件逻辑电路如图 3-5 所示。图中 A、B、C 为 3 个寄存器，在运算开始时，A 用于存放部分积、B 用于存放被乘数、C 用于存放乘数；乘法运算结束后，A 用于存放乘积高位部分，C 用于存放乘积低位部分。CR 为计数器，用于记录乘法运算的次数。C_j 为进位位。C_T 为乘法控制触发器，用于控制乘法运算的开始与结束。$C_T=1$，允许发出移位脉冲，控制进行乘法运算；$C_T=0$，不允许发出移位脉冲，停止进行乘法运算。

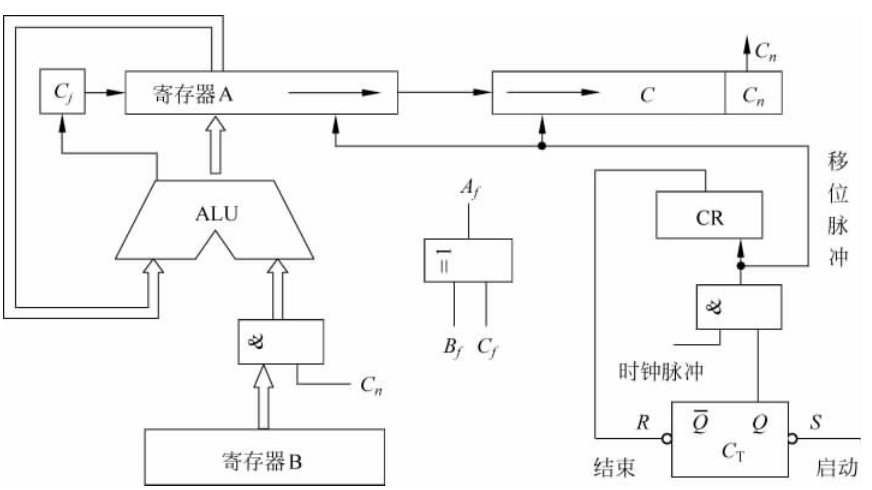


图 3-5 原码一位乘法硬件逻辑电路图

按图 3-5 的硬件线路实现原码一位乘法的流程如图 3-6 所示。执行乘法运算前，把被乘数的绝对值$|x|$送入寄存器 B，乘数的绝对值$|y|$送入寄存器 C，把存放部分积的寄存器 A、进位标志 C_j 及计数器 CR 都清 0。乘法运算开始时，将触发器 C_T 置 1，使乘法线路可以在时钟

脉冲的作用下进行右移操作。寄存器 C 的最低位 C_n 用于控制被乘数是否与上次的部分积相加。相加后，在时钟脉冲的作用下将 C_j 位与寄存器 A、C 一起右移一位，即 CF 移入 A 的最高位，A 的最低位移入 C 的最高位，作为本次运算控制用的 C_n 被移出；同时计数器 CR 加 1。循环 n 次相加、移位后，寄存器 A 中存放的是 $|x\times y|$ 的高 n 位乘积，C 中存放的是 $|x\times y|$ 的低 n 位乘积。此时，计数器 CR 计满 n 次，向触发器 C_T 发出置 0 信号，结束乘法运算。将乘积符号 $z_f=B_f\oplus C_f$ 与 $|x\times y|$ 结合，即得 $[x\times y]_{原}$。

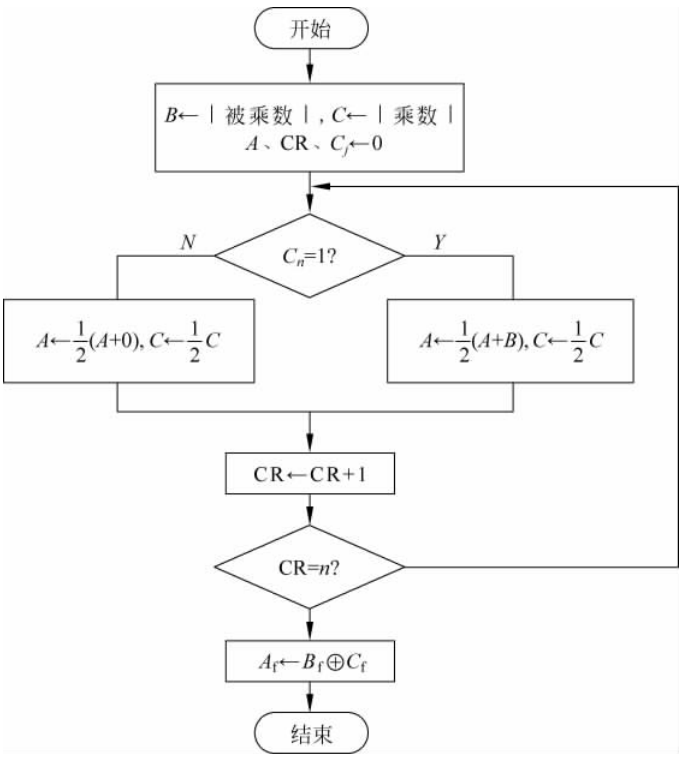


图 3-6　原码一位乘法算法流程

在实际机器中，寄存器 C 通常为具有左移和右移功能的移位寄存器，但寄存器 A 一般不具有移位功能，因此由 ALU 计算出的部分积是采用斜送到寄存器 A 的方法实现移位的。图 3-7 是具有左、右斜送和直接传送的移位器的示意图。图中 F_{i-1}、F_i、F_{i+1} 分别是加法器的第 $i-1$、i、$i+1$ 位输出，$A\leftarrow\frac{1}{2}F$、$A\leftarrow 2F$、$A\leftarrow F$ 分别为将加法器的运算结果右移、左移和直接传送到 A 的控制信号。

在利用原码一位乘法进行乘法时，因为每次判别乘数的一位，因此 n 位乘数需作 n 次加法与移位，使乘法计算的速度较慢。如果能够一次判别多位乘数，就可以提高乘法速度，这就是多位乘法的思想。有两位、四位甚至更多位的乘法，其中原码两位乘法与原码一位乘法相比，增加的逻辑电路不多，但可使乘法速度提高将近一倍。下面讨论原码两位乘法的算法。

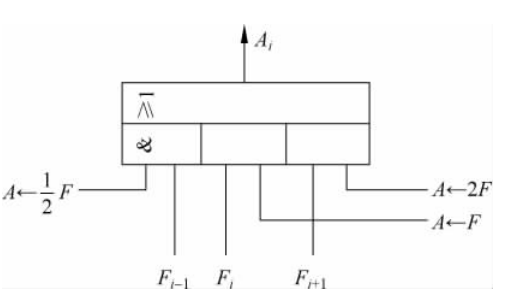


图 3-7　实现移位功能的逻辑电路

2. 原码两位乘法

原码两位乘法算法的思想是每次判别乘数的两位,将一位乘法中的两步用一步替代。设 $y_{n-1}y_n$ 为判别位,z_{i-1} 为前次部分积,z_i 为两位乘法的第 i 位部分积。观察原码一位乘法的运算,我们可以发现部分积 z_i 与 $y_{n-1}y_n$ 和 z_{i-1} 的关系如下:

$y_{n-1}y_n=00$,$z_i=\frac{1}{2}\left[\frac{1}{2}(z_{i-1}+0)+0\right]=\frac{1}{4}(z_{i-1}+0)$,即部分积 z_{i-1} 加 0,右移两位;

$y_{n-1}y_n=01$,$z_i=\frac{1}{2}\left[\frac{1}{2}(z_{i-1}+x)+0\right]=\frac{1}{4}(z_{i-1}+x)$,即部分积 z_{i-1} 加被乘数 x,右移两位;

$y_{n-1}y_n=10$,$z_i=\frac{1}{2}\left[\frac{1}{2}(z_{i-1}+0)+x\right]=\frac{1}{4}(z_{i-1}+2x)$,即部分积 z_{i-1} 加两倍被乘数 x,右移两位;

$y_{n-1}y_n=11$,$z_i=\frac{1}{2}\left[\frac{1}{2}(z_{i-1}+x)+x\right]=\frac{1}{4}(z_{i-1}+3x)$,即部分积 z_{i-1} 加三倍被乘数 x,右移两位;

在上述操作中,$z_{i-1}+2x$ 可以通过将被乘数 x 左移一位后与部分积 z_{i-1} 相加来实现,但 $z_{i-1}+3x$ 却难以简单地用移位后相加来实现。注意到:

$$\frac{1}{4}(z_{i-1}+3x)=\frac{1}{4}(z_{i-1}+4x-x)=\frac{1}{4}(z_{i-1}-x)+x$$

即在做 $\frac{1}{4}(z_{i-1}+3x)$ 时,可以本次先做 $\frac{1}{4}(z_{i-1}-x)$,加 x 到下次再做,本次先欠着。为此设置了一个欠账触发器 C_J,记录本次欠账的情况。若 $C_J=1$,表示本次欠账,下次需多加一个 x;$C_J=0$,表示本次无欠账,下次就不用多加 x。可见原码两位乘法的运算规则是由两个乘数判别位 $y_{n-1}y_n$ 和欠账触发器 C_J 的状态共同确定的。原码两位乘法的运算规则如表 3-3 所示。

表 3-3 原码两位乘法的运算规则

$y_{n-1}y_nC_J$	操 作	说 明
0 0 0	部分积 z_{i-1} 右移两位,$C_J\leftarrow0$;	$z_{i-1}+0$ 的加 0 操作可以不做,直接将 z_{i-1} 右移即可
0 0 1	部分积 z_{i-1} 加 x,右移两位,$C_J\leftarrow0$;	为还上次欠账,做 $z_{i-1}+x$
0 1 0	部分积 z_{i-1} 加 x,右移两位,$C_J\leftarrow0$	
0 1 1	部分积 z_{i-1} 加 $2x$,右移两位,$C_J\leftarrow0$;	为还上次欠账,做 $z_{i-1}+2x$
1 0 0	部分积 z_{i-1} 加 $2x$,右移两位,$C_J\leftarrow0$	
1 0 1	部分积 z_{i-1} 减 x,右移两位,$C_J\leftarrow1$;	为了还上次欠账,需加 $3x$,所以本次减 x,再欠账
1 1 0	部分积 z_{i-1} 减 x,右移两位,$C_J\leftarrow1$;	为了加 $3x$,本次减 x,欠账
1 1 1	部分积 z_{i-1} 右移两位,$C_J\leftarrow1$;	为了还上次欠账,需做 $z_{i-1}+4x$,因为 $\frac{1}{4}(z_{i-1}+4x)=\frac{1}{4}z_{i-1}+x$,所以本次将 z_{i-1} 右移,不加减 x,再欠账

原码两位乘法运算次数的控制方法为:

(1) 若操作数字长为奇数,去掉一位符号位后,数值部分长度 n 为偶数,共需做 $n/2$ 次运算。

(2) 若操作数字长为偶数,去掉一位符号位后,数值部分长度 n 为奇数,因此需将乘数再加上一个符号位并使之为 0,以便形成偶数位,此时共需做 $(n+1)/2$ 次运算,但最后一次移位

仅移一位。

(3) 若最后一次运算后 C_J 仍为 1，则需再做一次加 x 操作，最高符号位作为真正的符号位，才能保证运算过程正确无误。以便还清欠账。

因为在原码两位乘法中需要 $2x$，即需要将 x 左移 1 位，这时被乘数的绝对值可能会大于 2，数值会侵占符号位；又因在运算过程中，做加法所得到的正常进位不得丢失，所以在进行原码两位乘法运算时，部分积(初始时为被乘数)需要使用 3 个符号位，以便记录左移和进位的数值。

原码两位乘法的算法规定：使用 3 个符号位时，用 000 表示“+”，用 111 表示“−”；在运算过程中，最高符号位为真正的符号，低两位符号位可以用于记录左移和进位的数值。由于原码乘法是被乘数和乘数的绝对值参加运算，所以在运算初始时，被乘数的符号位一定为 000。另外，在原码两位乘法的运算过程中，需要做 $-x$ 操作，因此采用了补码减法的方法，即采用加 $[-|x|]_{补}$ 的方法实现减法操作。由于在原码两位乘法的运算过程中使用了补码加减运算，所以右移两位的操作也必须按补码右移的规则进行。

例 3.10　按原码两位乘法的算法，计算 $[x\times y]_{原}$。

① $[x]_{原}=0.1101$，$[y]_{原}=1.1011$　② $[x]_{原}=1.01101$，$[y]_{原}=1.10111$

解：① $|x|=000.1101$，$|y|=0.1011$，$[-|x|]_{补}=1.0011=111.0011$，积 $[z]_{原}$ 的符号 $z_f=0\oplus1=1$。

部分积	乘数 C_J	说明
000.0000	1011　0	初始部分积 $z_0=0$
+ 111.0011		因为 $y_{n-1}y_nC_J=110$，所以减 x；$C_J=1$
111.0011		
→ 111.1100	11 10　1	部分积与乘数同时右移两位
+ 111.0011		因为 $y_{n-1}y_nC_J=101$，所以减 x；$C_J=1$
110.1111		
→ 111.1011	1111　1	部分积与乘数同时右移两位
+ 000.1101		因为 $C_J=1$，所以需要再加一次 x
000.1000	1111	

得 $|x\times y|=0.10001111$，加上符号部分得 $[x\times y]_{原}=1.10001111$，即 $x\times y=-0.10001111$。

② $|x|=000.01101$，$|y|=0.10111$，$[-|x|]_{补}=111.10011$，积 $[z]_{原}$ 的符号 $z_f=1\oplus1=0$。因为乘数数值部分为 5 位，所以运算时要在乘数上加一位符号且为 0。

部分积	乘数　C_J	说明
000.00000	01011　0	初始部分积 $z_0=0$
+ 111.10011		因为 $y_{n-1}y_nC_J=110$，所以减 x；$C_J=1$
111.10011		
→ 111.11100	11 01 01　1	部分积与乘数同时右移两位
+ 000.11010		因为 $y_{n-1}y_nC_J=011$，所以加 $2x$；$C_J=0$
000.10110		
→ 000.00101	1011 01　0	部分积与乘数同时右移两位
+ 000.01101		因为 $y_{n-1}y_nC_J=010$，所以加 x；$C_J=0$
000.10010		
→ 000.01001	01011	最后一次右移一位，运算结束

得 $|x\times y|=0.0100101011$，加上符号部分得 $[x\times y]_{原}=0.0100101011$，即 $x\times y=+0.0100101011$。

原码乘法实现比较简单，但由于实际机器中都采用补码作加减运算，数据的存放也采用补码形式，因此如果在作乘法前要将补码转换成原码，相乘之后又要将原码转换为补码，会增添许

多操作步骤,使运算复杂。为了减少原码与补码之间的转换,有不少机器直接采用了补码乘法。

3.3.2 补码乘法运算

补码乘法有多种算法,计算机中常用的有校正法和布斯乘法,其中布斯乘法是由布斯(A. D. Booth)夫妇提出的,算法实现比较方便。下面我们主要讨论的补码一位乘法就是布斯乘法。

以定点小数为例,设参加运算的被乘数 x 的补码为$[x]_{补}=x_0.x_1x_2\cdots x_n$,乘数 y 的补码为$[y]_{补}=y_0.y_1y_2\cdots y_n$,乘积为$[z]_{补}=[x\times y]_{补}$。

(1) 设被乘数 x 的符号任意,乘数 y 为正数,即:

$$[x]_{补}=x_0.x_1x_2\cdots x_n$$
$$[y]_{补}=0.y_1y_2\cdots y_n$$

根据补码的定义(2-5)及模 2 运算的性质,有:

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

则:$[x]_{补}\times[y]_{补}=2^{n+1}y+x\times y=2\times(y_1y_2\cdots y_n)+x\times y \pmod 2 \tag{3-11}$

因为式(3-11)中 $y_1y_2\cdots y_n$ 为大于 0 的正整数,根据模 2 性质有:

$$2\times(y_1y_2\cdots y_n)=2 \pmod 2$$

所以得$[x]_{补}\times[y]_{补}=2+x\times y=[x\times y]_{补} \pmod 2$

因为 $y>0$,$[y]_{补}=y$,$y_0=0$,所以

$$[x\times y]_{补}=[x]_{补}\times[y]_{补}=[x]_{补}\times y=[x]_{补}\times(0.y_1y_2\cdots y_n)$$
$$=[x]_{补}\times\left(\sum_{i=1}^{n}y_i2^{-i}\right) \tag{3-12}$$

(2) 设被乘数 x 的符号任意,乘数 y 为负数,即:

$$[x]_{补}=x_0.x_1x_2\cdots x_n$$
$$[y]_{补}=1.y_1y_2\cdots y_n=2+y \pmod 2$$

因为 $y=[y]_{补}-2=0.y_1y_2\cdots y_n-1$,所以 $x\times y=x\times(0.y_1y_2\cdots y_n)-x$,得:

$$[x\times y]_{补}=[x\times(0.y_1y_2\cdots y_n)]_{补}-[x]_{补} \tag{3-13}$$

因为 $0.y_1y_2\cdots y_n>0$,所以$[x\times(0.y_1y_2\cdots y_n)]_{补}=[x]_{补}\times(0.y_1y_2\cdots y_n)$。

可得

$$[x\times y]_{补}=[x]_{补}\times(0.y_1y_2\cdots y_n)-[x]_{补} \tag{3-14}$$

(3) 设被乘数 x 和乘数 y 均为任意符号数,将情况(1)、(2)综合,可得:

$$[x\times y]_{补}=[x]_{补}\times(0.y_1y_2\cdots y_n)-[x]_{补}\times y_0=[x]_{补}\times(0.y_1y_2\cdots y_n-y_0)$$
$$=[x]_{补}\times\left(-y_0+\sum_{i=1}^{n}y_i2^{-i}\right)$$
$$=-y_0[x]_{补}+2^{-1}y_1[x]_{补}+2^{-2}y_2[x]_{补}+\cdots+2^{-n}y_n[x]_{补}$$
$$=(y_1-y_0)[x]_{补}+2^{-1}(y_2-y_1)[x]_{补}+2^{-2}(y_3-y_2)[x]_{补}$$
$$+\cdots+2^{-(n-1)}(y_n-y_{n-1})[x]_{补}+2^{-n}(y_{n+1}-y_n)[x]_{补} \tag{3-15}$$

仿照原码一位乘法的推导方法,令部分积的初始值$[z_0]_{补}=0$,可将式(3-15)写成部分积的递推形式:

$$
\begin{aligned}
&[z_0]_{补}=0 \quad (\text{初始部分积为 } 0)\\
&[z_1]_{补}=2^{-1}\{[z_0]_{补}+(y_{n+1}-y_n)\,[x]_{补}\}\\
&[z_2]_{补}=2^{-1}\{[z_1]_{补}+(y_n-y_{n-1})\,[x]_{补}\}\\
&\vdots\\
&[z_i]_{补}=2^{-1}\{[z_{i-1}]_{补}+(y_{n-i+2}-y_{n-i+1})\,[x]_{补}\}\\
&\vdots\\
&[z_n]_{补}=2^{-1}\{[z_{n-1}]_{补}+(y_2-y_1)\,[x]_{补}\}\\
&[z_{n+1}]_{补}=\{[z_n]_{补}+(y_1-y_0)\,[x]_{补}\}=[x\times y]_{补}
\end{aligned}
\tag{3-16}
$$

根据式(3-16),可以归纳出补码一位乘法的运算规则:

(1) 参加运算的数均以补码表示,符号位 x_0、y_0均参加运算。考虑到运算时可能出现部分积的绝对值大于 1 而占用的情况(此时并不属于溢出),部分积与被乘数采用双符号位。

(2) 在乘数最低位增设附加位 y_{n+1},且初始 $y_{n+1}=0$。

(3) 以乘数最低位的 y_ny_{n+1}作为乘法判别位,依次比较相邻两位乘数的状态,以决定相应的操作。具体操作如表 3-4 所示。

表 3-4　补码一位乘法的操作

y_ny_{n+1}	操　作	说　明
0　0	$[z_{i+1}]_{补}=2^{-1}[z_i]_{补}$	本次部分积等于前次部分积加 0(或不加)后连同乘数右移一位
1　1	$[z_{i+1}]_{补}=2^{-1}[z_i]_{补}$	本次部分积等于前次部分积加 0(或不加)后连同乘数右移一位
0　1	$[z_{i+1}]_{补}=2^{-1}\{[z_i]_{补}+[x]_{补}\}$	本次部分积等于前次部分积加$[x]_{补}$后连同乘数右移一位
1　0	$[z_{i+1}]_{补}=2^{-1}\{[z_i]_{补}-[x]_{补}\}$	本次部分积等于前次部分积减$[x]_{补}$后连同乘数右移一位

(4) 重复第(3)步,共做 $n+1$ 次,但最后一次(第 $n+1$ 次)只运算、不移位。

在补码一位乘法的运算过程中应注意的是:部分积的初始值 $z_0=0$;减$[x]_{补}$的操作用加$[-x]_{补}$实现;部分积右移时必须按补码右移的规则进行。

例 3.11　设 $x=-0.1101$,$y=-0.1011$,用补码一位乘法计算 $x\times y$。

解:$[x]_{补}=11.0011$,$[y]_{补}=1.0101$,$[-x]_{补}=00.1101$

部分积	乘数　y_ny_{n+1}	说明
00.0000	1.0 1 0 1 0	初始部分积$z_0=0$，附加位$y_{n+1}=0$
+ 00.1101		因为$y_ny_{n+1}=10$，所以$+[-x]_{补}$
00.1101		
→ 00.0110	1 1 0 1 0 1	部分积与乘数同时右移一位
+ 11.0011		因为$y_ny_{n+1}=01$，所以$+[x]_{补}$
11.1001		
→ 11.1100	1 1 1 0 1 0	部分积与乘数同时右移一位
+ 00.1101		因为$y_ny_{n+1}=10$，所以$+[-x]_{补}$
00.1001		
→ 00.0100	1 1 1 1 0 1	部分积与乘数同时右移一位
+ 11.0011		因为$y_ny_{n+1}=01$，所以$+[x]_{补}$
11.0111		
→ 11.1011	1 1 1 1 1 0	部分积与乘数同时右移一位
+ 00.1101		因为$y_ny_{n+1}=10$，所以$+[-x]_{补}$
00.1000	1 1 1 1	最后一次只运算、不移位

得 $[x\times y]_{补}=0.10001111$,所以 $x\times y=0.10001111$。

从例 3.11 中可以看出,采用补码一位乘法的算法,乘积的符号是在运算过程中自然形成的,不需要加以特别处理,这是补码乘法与原码乘法的重要区别。实现补码一位乘法的硬件逻辑结构如图 3-8 所示。

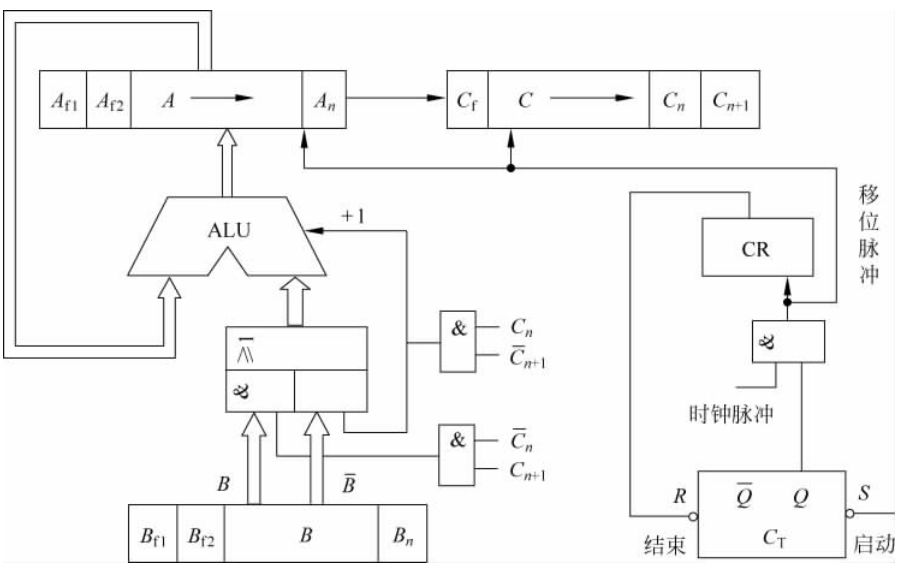


图 3-8 补码一位乘法的硬件逻辑结构图

实现补码一位乘法的硬件逻辑结构与实现原码一位乘法的硬件逻辑结构很相似,只是部分控制线路不同。图 3-8 中寄存器 A 用于存放乘积和部分积高位部分,初始时其内容为 0;A_{f1}、A_{f2}是部分积的两个符号位,补码乘法中符号位和数值位同时参加运算。寄存器 C 用于存放乘数和部分积低位部分,初始时其内容为乘数;C_n和C_{n+1}用于控制电路中是进行$+[x]_{补}$操作还是$+[-x]_{补}$操作。寄存器 B 用于存放被乘数,可以在 C_n和C_{n+1}的控制下输出正向信号 B 和反向信号 $\overline{B}$;当执行$+[x]_{补}$时,输出正向信号 B,进行 $A+B$ 操作;当执行$+[-x]_{补}$时,输出反向信号 $\overline{B}$,进行 $A+\overline{B}+1$ 操作。C_T是乘法控制触发器,$C_T=1$,允许发出移位脉冲,控制进行乘法运算;$C_T=0$,不允许发出移位脉冲,停止进行乘法运算。CR 是计数器,用于记录乘法次数。在运算初始时,CR 清 0,每进行一次运算,CR+1;当计数到 CR=$n+1$ 时,结束运算。另外,由于线路中控制在 CR=n 时,就将 C_T清 0,所以在第$n+1$ 次运算时,不再进行移位。补码一位乘法的算法流程如图 3-9 所示。

注意,在补码一位乘法的流程图中,寄存器 A 和 C 的移位是在对 CR 进行判断之后进行的,说明在第 $n+1$ 次运算后不进行移位。

为了提高运算速度,可以采用补码两位乘法。与原码两位乘法的算法类似,将补码一位乘法中的两步用一步代替,就可以得到补码两位乘法的运算规则。限于篇幅,本书中不再讲解,感兴趣的读者可以进一步查询资料。

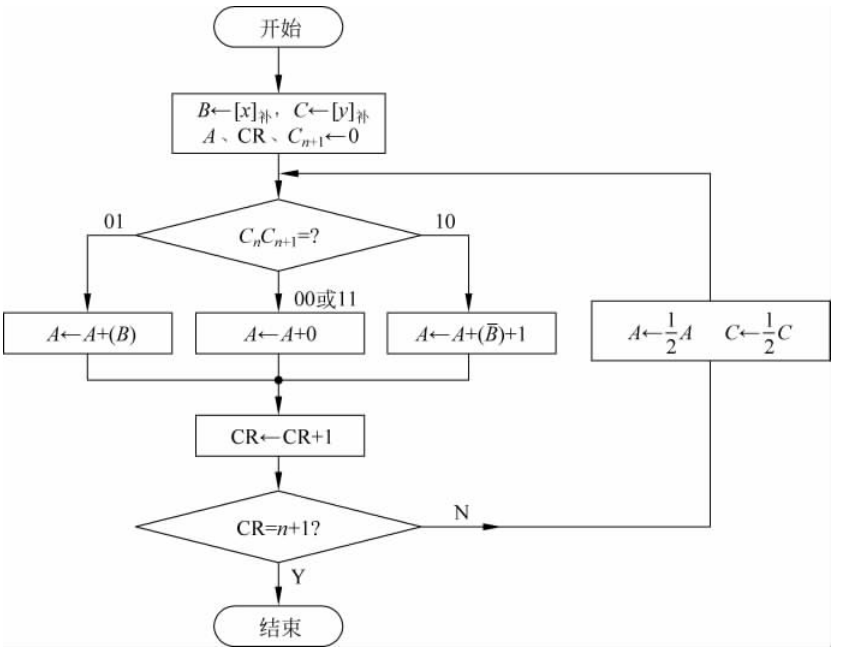


图 3-9　补码一位乘法的算法流程

3.3.3　快速乘法运算

在科学计算中，乘法运算约占全部算术运算的 1/3。因此，无论从提高计算机的运算速度还是从提高计算效率来说，都有必要研究高速乘法部件以进一步提高乘法的运算速度。根据 3.3.1 节的内容，可知在常规乘法器中，两位乘法比一位的运算速度快，显然可采用一次判断更多位乘数（如一次判断四位）的方法进一步提高乘法运算的速度。但多位乘法运算的控制复杂性将呈几何级数性的增加，实现的难度很大。随着大规模集成电路的迅速发展和硬件价格的降低，出现了多种阵列乘法组件，目的就是利用硬件的叠加方法或流水处理的方法来提高乘法运算速度。在本小节中，我们将简单讨论阵列乘法器的基本原理。

1. 无符号数阵列乘法器

设有两个 4 位无符号二进制整数：$A=a_3a_2a_1a_0$，$B=b_3b_2b_1b_0$，求 $P=A\times B$。按手算方法的运算过程为：

在手算算式中，每个 a_ib_j（$i=0\sim3$，$j=0\sim3$）都是由两个 1 位的二进制数相乘得到的，称为位积，每个位积都可以用一个二输入端的与门予以实现。两个 4 位的二进制整数相乘所得的乘积的有效位数最多可达到 8 位，即 $P=P_7P_6P_5P_4P_3P_2P_1P_0$。利用二进制加法器将位权相等的位积相加，即可得到相应位的乘积。

					a_3	a_2	a_1	a_0
				$\times$	b_3	b_2	b_1	b_0
				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				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

例如,位积 a_3b_0、a_2b_1、a_1b_2、a_0b_3的位权都是 2^3,可以利用 3 个加法器逐次对它们求和。其过程是先对 a_3b_0、a_2b_1求和,产生的和数与 a_1b_2及相邻低位(a_2b_0加 a_1b_1)来的进位相加,然后再将所得的和数与 a_0b_3及相邻低位来的进位相加,最后形成相应位的乘积 P_3以及向高位 P_4的进位。分析了手算过程后,可以想到如果把大量的加法器单元电路按一定的阵列形式排列起来,直接实现手算算式的运算过程,就可以避免在一位和两位乘法中所需的大量重复的相加和移位操作,从而提高乘法运算的速度,这就是阵列乘法器的基本思想。

图 3-10 给出了一个 4×4 位无符号数阵列乘法器的逻辑原理图。图中方框内的电路由一个与门和一个一位全加器 FA 组成,内部结构如图 3-10 中左上角的电路所示。其中,与门用于产生位积,全加器用于位积的相加。图中方框的排列阵列与笔算乘法的位积排列相似,阵列的每一行送入乘数的一位数位 b_i,而各行错开形成的每一斜列则送入被乘数的一位数位 a_i。

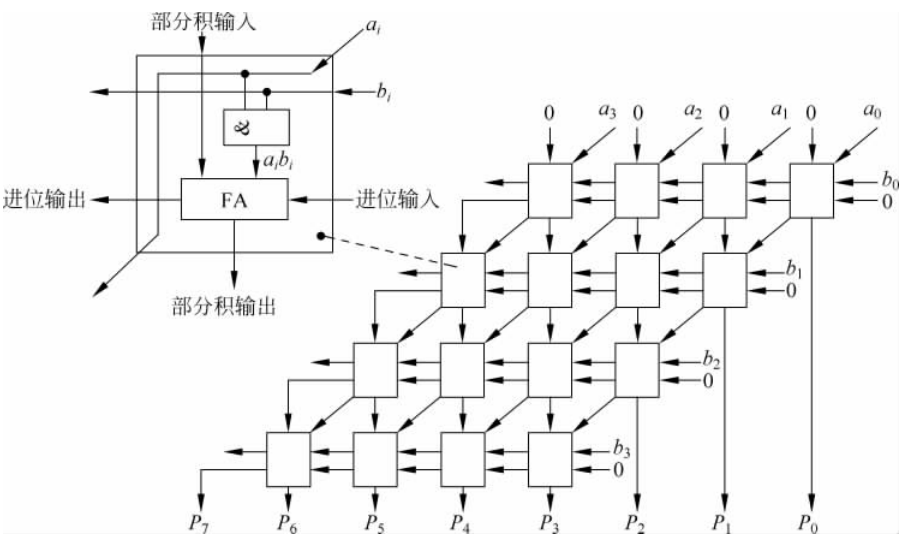


图 3-10 4×4 位无符号数阵列乘法器的逻辑原理图

2. 带符号数阵列乘法器

在无符号数阵列乘法器的基础上增加符号处理电路和求补电路,即可实现带符号数乘法器。带符号阵列乘法器既可以实现原码乘法也可以实现补码乘法。

设被乘数和乘数分别为 $n+1$ 位带符号数,$A=a_f.\,a_{n-1}a_{n-2}\cdots a_0$,$B=b_f.\,b_{n-1}b_{n-2}\cdots b_0$。图 3-11 给出了一个$(n+1)\times(n+1)$位带符号数乘法的阵列乘法器的逻辑原理图。在图 3-11 中,如果需要进行原码乘法运算,则不用算前求补器与算后求补器,直接把被乘数和乘数的绝对值送入乘法阵列中进行计算,得到 $2n$ 位乘积的绝对值 $p_{2n-1}p_{2n-2}\cdots p_1p_0$;将被乘数 a_f和乘数的符号 b_f通过异或门的处理得到积的符号 P_f。将积的符号加入到乘积的绝对值中,即得到 $2n+1$ 位原码形式的乘积 $P_fP_{2n-1}P_{2n-2}\cdots P_1P_0$。

如果需要进行补码乘法,则需要由两个算前求补器先将两个补码操作数转换为两数的绝对值,然后再送入无符号乘法阵列中计算,即可得到 $2n$ 位的乘积绝对值 $p_{2n-1}p_{2n-2}\cdots p_1p_0$。然后根据异或门输出的积的符号 P_f控制算后求补器对 $p_{2n-1}p_{2n-2}\cdots p_1p_0$求补,将求补结果与符号 P_f结合,就得到 $2n+1$ 位的补码形式的乘积 $P_fP_{2n-1}P_{2n-2}\cdots P_1P_0$。

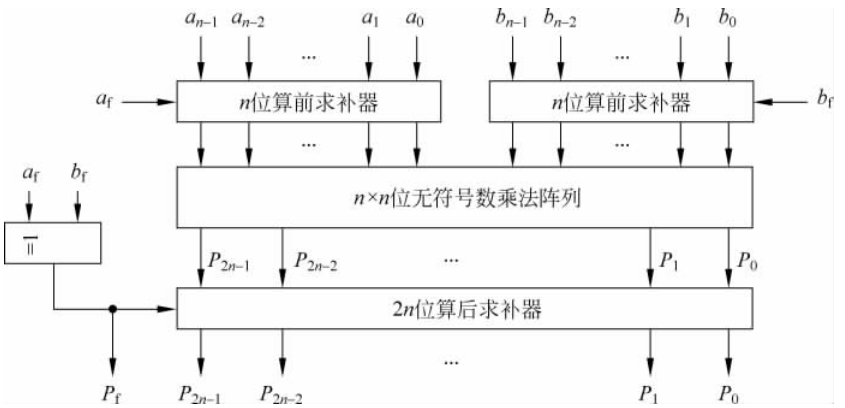


图 3-11　$(n+1)\times(n+1)$位带符号数乘法的阵列乘法器的逻辑原理图

在图 3-11 中，算前和算后求补器可以在符号位的控制下对数值部分进行求补，以便满足补码乘法运算的需要。图 3-12 给出了一个 4 位二进制对 2 求补器的逻辑电路。其中每一位二进制对 2 求补电路的逻辑表达式为：

$$a_i^* = a_i \oplus EC_{i-1}, \quad 0 \leqslant i \leqslant n$$
$$C_i = a_i + C_{i-1}, \quad \text{其中 } C_{-1} = 0 \tag{3-17}$$

式(3-17)中，E 为控制信号，用于控制是否进行求补。

$E=0$，不进行求补，$a_i^* = a_i$；

$E=1$，进行求补，$a_i^* = a_i \oplus EC_{i-1}$。

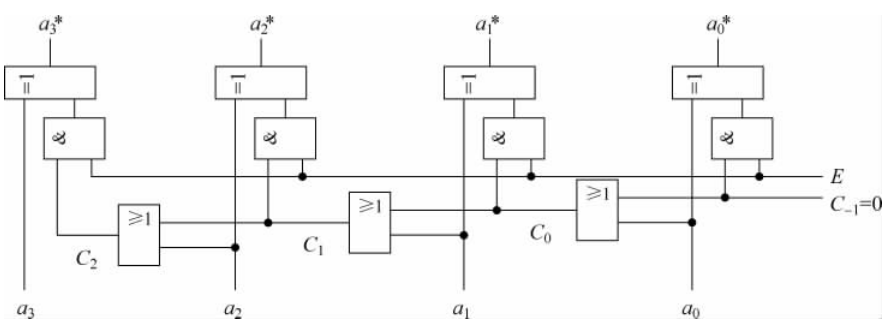


图 3-12　一个 4 位二进制对 2 求补器的逻辑电路

由于阵列乘法器采用重复设置大量器件的方法构成乘法阵列，避免了乘法运算中的重复相加和移位操作，换取了高速的乘法运算速度。而且乘法阵列内部结构规整，便于用超大规模集成电路实现，使得阵列乘法器得到了广泛的应用。

3.4　定点除法运算

除法运算的处理思想与乘法运算的处理思想相似，其常规算法也是将除法的计算过程转换成若干次“加减—移位”循环来实现。

定点除法运算可以分为原码除法和补码除法。由于定点运算的结果不应超过机器所能表示的数据范围，所以为了不使商产生溢出，在进行定点除法时应满足下列条件：

(1) 对定点小数除法要求|被除数|<|除数|,且除数不为0;

(2) 对定点整数除法要求|被除数|≥|除数|,且除数不为0。

3.4.1 原码除法运算

首先分析一下定点小数除法运算的手算过程。

例 3.12 设 $x=-0.1011, y=0.1101$ 求 x/y。

解:在手算 x/y 时,商的符号根据除法对符号的处理规则"正正得正,正负得负"心算得到;商的数值部分采用被除数和除数的绝对值进行计算,手算过程如图 3-13 所示。运算结果得:商 $q=-0.1101$,余数 $r=-0.00000111$。

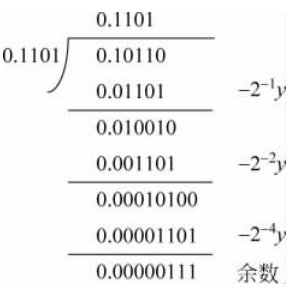


图 3-13 小数除法的手算过程

分析例 3.12 的运算过程,可得手算除法的规则:

(1) 商的各位是通过比较余数(初始时为被除数)与除数的大小得到的。若余数大于除数,则相应位上商为1,将余数减去除数,再把除数向右移一位与余数相比较;若余数小于除数,则相应位上商为0,把除数向右移一位再与余数相比较。

(2) 每次做减法时,总是余数不动,低位补0,再与右移一位后的除数相减。

(3) 商的符号单独处理。

由于上述算法是通过不断比较余数和除数来决定上商的,所以也将其称为比较法。

在计算机中可以参照比较法的算法实现除法,但实现过程中需解决的问题有:

(1) 在手算过程中,余数和除数的大小比较是通过心算得到的,而计算机中进行比较就需要在加减电路之外设置比较电路,这将增加硬件成本。

(2) 如果通过做减法来进行余数和除数的比较,则可以节省电路,即用余数(初始时为被除数)减去除数,若减得结果为正,表示够减,上商为1;若减得结果为负,表示不够减,上商为0。但如果不够减,进行减法后,余数已经被减去了,下一步应该如何处理?

(3) 如果每次减法均采用余数不动,低位补0,再与右移一位后的除数相减,则所需的加法器的位数必须是除数的两倍,这将使加法器的规模增大。

(4) 在手算过程中,上商是从高位向低位逐位求的,而在计算机中要求机器把每位商直接写到寄存器的不同位是较难控制的。

在实际的计算机中实现除法时,一般采用以下方法解决上述问题:

(1) 通过做减法来进行比较余数和除数的比较,即用余数(初始时为被除数)减去除数,若减得结果为正,表示够减,上商为1;若减得结果为负,表示不够减,上商为0。

(2) 采用恢复余数法或不恢复余数法解决余数减去除数后不够减的处理问题。

(3) 在余数不动,低位补0,再与右移一位后的除数相减的操作中,用左移余数的方法代替右移除数的操作。这样操作,实际运算结果是一样的,但对线路结构更有利。不过这样操作所得到的余数不是真正的余数,必须将它乘上 2^{-n} 才是真正的余数。

(4) 为了便于控制,可以在运算过程中通过将每次得到的商直接写到寄存器的最低位并与前面运算所得到部分商一起左移一位的方法实现商的定位。

1. 原码恢复余数法

在原码除法中,参加运算的被除数和除数均采用原码表示,所得的商和余数也采用原码表

示。运算时，符号位单独处理，被除数和除数的绝对值相除。为了保证定点除法的运算结果不超过机器所能表示的定点数据范围，在进行除法之前必须判定被除数和除数是否满足定点小数除法或定点整数除法的要求，如果不满足要求，由于运算结果会将产生溢出，因此不能继续进行除法运算。下面以定点小数除法为例，讨论原码恢复余数法的算法。

定点小数的原码恢复余数法由以下步骤实现：

(1) 判溢出，要求|被除数|<|除数|。若|被除数|>|除数|，则除法将发生溢出。

(2) 符号位单独处理，商的符号由被除数和除数符号的异或运算求得。

(3) 用被除数和除数的数值部分进行运算，被除数减去除数。

(4) 若所得余数为正，表示够减，相应位上商为 1，余数左移一位(相当于除数右移)减去除数；若所得余数为负，表示不够减，相应位上商为 0，余数加上除数(即恢复余数)，再左移一位后减去除数。

(5) 重复第(4)步，直到求得所要求的商的各个位为止。

需注意的是：在原码除法的运算过程中，数值部分的计算是对被除数和除数的绝对值进行的；因为需要进行减法，所以采用补码加减法来实现运算；为了不使余数左移时产生数值侵犯符号位的情况，加减运算时采用双符号位。

例 3.13　已知 $x=-0.1011$，$y=+0.1101$，用原码恢复余数法求 x/y。

解：$[x]_{原}=1.1011$，$[y]_{原}=0.1101$，$|x|=00.1011$，$|y|=00.1101$，$[-|y|]_{补}=11.0011$，商符 $q_f=x_f\oplus y_f=1\oplus 0=1$。

	余数	上商	说明
	00.1011	0.0000	初始余数为被除数
+	11.0011		减 y，即加$[-\|y\|]_{补}$
	11.1110	0.0000	余数为负，上商为 0
+	00.1101		加 y 恢复余数
	00.1011		
←	01.0110	0.0000	左移一位
+	11.0011		减 y
	00.1001	0.0001	余数为正，上商为 1
←	01.0010	0.0010	左移一位
+	11.0011		减 y
	00.0101	0.0011	余数为正，上商为 1
←	00.1010	0.0110	左移一位
+	11.0011		
	11.1101	0.0110	余数为负，上商为 0
+	00.1101		加 y 恢复余数
	00.1010		
←	01.0100	0.1100	左移一位
+	11.0011		减 y
	00.0111	0.1101	余数为正，上商为 1，结束运算

得商的绝对值为$|q|=|x/y|=0.1101$，余数的绝对值为$|r|=0.0111$。因为 $q_f=1$，所以商$[q]_{原}=[x/y]_{原}=1.1101$，$x/y=-0.1101$。因为本题中 $n=4$，所以所得的余数需乘以 2^{-4}才是真正的余数，即$|r|=0.0111\times 2^{-4}$。因为余数的符号与被除数一致，所以余数$[r]_{原}=1.0111\times 2^{-4}$，$r=-0.00000111$。

从例 3.13 中可以看出，在数值部分的长度 $n=4$ 的除法运算中，共上商 5 次，其中第一次商位于商的整数部分，对于定点小数除法而言，如果该位商为 1，则表示|被除数|>|除数|，除

法溢出,不能继续进行运算;如果该位商为0,则表示|被除数|<|除数|,除法合法,可以继续进行运算。

分析恢复余数法的运算过程可知,当余数为正时,需作余数左移、相减,共两步操作;当余数为负时,需作相加、左移、相减,共三步操作。由于操作步骤的不一致,使得控制复杂,而且恢复余数的过程也降低了除法速度。因此,在实际应用中,很少采用恢复余数法。

2. 原码不恢复余数法

在恢复余数法的运算过程中:

当余数 $r_i>0$ 时,执行的操作是左移一位→减除数,结果是 $2r_i-y$;当余数 $r_i<0$ 时,执行的操作是加除数(恢复余数)→左移→减除数,结果是 $2(r_i+y)-y$。变换后得 $2(r_i+y)-y=2r_i+2y-y=2r_i+y$。

根据上述分析,可以发现将"加除数(恢复余数)→左移→减除数"的操作用"余数左移→加除数"的操作来替代,所得结果是一样的。而且这样做,既节省了恢复余数的时间,又简化了除法控制逻辑(无论余数为正还是为负,余数的操作均为左移、加减运算两步操作)。由此导出了原码不恢复余数法,其算法为:

(1) 判溢出,比较被除数和除数。若在定点小数运算时,|被除数|>|除数|,则除法将发生溢出,不能进行除法运算。

(2) 符号位单独处理,商的符号由被除数和除数符号的异或运算求得。

(3) 用被除数和除数的数值部分进行运算,被除数减去除数。

(4) 若所得余数为正,表示够减,相应位上商为1,将余数左移一位后,减去除数;若所得余数为负,表示不够减,相应位上商为0,将余数左移一位后,加上除数。

(5) 重复第(4)步,直到求得所要求的商的各位为止。如果最后一次所得余数仍为负,则需再做一次加除数的操作,以得到正确的余数。

运算时对除数的加减是交替进行的,所以原码不恢复余数法也称为原码加减交替除法。

例 3.14 已知 $x=-0.1011$,$y=+0.1101$,用原码不恢复余数法求 x/y。

解:$[x]_{原}=1.1011$,$[y]_{原}=0.1101$,$|x|=00.1011$,$|y|=00.1101$,$[-|y|]_{补}=11.0011$,商符 $q_f=x_f\oplus y_f=1\oplus 0=1$。

	余数	上商	说明
	00.1011	0.0000	初始余数为被除数,初始商为0
+	11.0011		减y
	11.1110	0.0000	余数为负,上商为0
←	11.1100	0.0000	左移一位
+	00.1101		加y
	00.1001	0.0001	余数为正,上商为1
←	01.0010	0.0010	左移一位
+	11.0011		减y
	00.0101	0.0011	余数为正,上商为1
←	00.1010	0.0110	左移一位
+	11.0011		减y
	11.1101	0.0110	余数为负,上商为0
←	11.1010	0.1100	左移一位
+	00.1101		加y
	00.0111	0.1101	余数为正,上商为1

因为 $q_f=1$，所以商$[q]_{原}=[x/y]_{原}=1.1101$，$x/y=-0.1101$，余数$[r]_{原}=1.0111\times2^{-4}$，$r=-0.00000111$。

例 3.15　已知$[x]_{原}=0.10101$，$[y]_{原}=0.11110$，用原码不恢复余数法求 x/y。

解：$|x|=00.10101$，$|y|=00.11110$，$[-|y|]_{补}=11.00010$，商符 $q_f=x_f\oplus y_f=0\oplus0=0$。

	余数	上商	说明
	00.10101	0.00000	初始余数为被除数，初始商为0
+	11.00010		减 y
	11.10111	0.00000	余数为负，上商为 0
←	11.01110	0.00000	左移一位
+	00.11110		加 y
	00.01100	0.00001	余数为正，上商为 1
←	00.11000	0.00010	左移一位
+	11.00010		减 y
	11.11010	0.00010	余数为负，上商为 0
←	11.10100	0.00100	左移一位
+	00.11110		加 y
	00.10010	0.00101	余数为正，上商为 1
←	01.00100	0.01010	左移一位
+	11.00010		减 y
	00.00110	0.01011	余数为正，上商为 1
←	00.01100	0.10110	左移一位
+	11.00010		减 y
	11.01110	0.10110	余数为负，上商为 0
+	00.11110		最后一步，因余数为负，加 y 恢复余数
	00.01100		

因为 $q_f=0$，所以商$[q]_{原}=[x/y]_{原}=0.10110$，$x/y=0.10110$，余数$[r]_{原}=0.01100\times2^{-5}$，$r=0.0000001100$。

以上讨论的定点小数的除法算法也适用于定点整数的除法运算。如前所述，为了不使商超出定点整数所能表示的数据范围，要求满足条件|除数|≤|被除数|。因为只有这样才能得到整数商，满足定点整数的要求。因此，在做整数除法前，通常先要对被除数和除数进行判断，如果不满足上述条件，则机器将发出出错信号。

另外，因为在乘法运算时，两个 n 位数相乘可得到 $2n$ 位的积，由于除法是乘法运算的逆运算，所以 $2n$ 位被除数除以 n 位除数，可以得到 n 位的商。在整数除法中，为了得到 n 位整数商，被除数位数的长度应该是除数位数长度的两倍，并且为了使商不超过 n 位，要求被除数的高 n 位比除数（n 位）小，否则商将超过 n 位，即运算结果溢出。如果被除数和除数的位数都为 n 位，则应在被除数前面加上 n 个 0，使被除数的长度扩展为 $2n$ 后再进行运算。在小数除法中，也可以使被除数位数的长度为除数位数长度的两倍。在字长为 n 的计算机中，称被除数采用双字长、除数采用单字长的除法为双精度除法。相应地称被除数和除数均采用单字长的除法为单精度除法。

例 3.16　已知$[x]_{原}=111011$，$[y]_{原}=000010$，用原码不恢复余数法求 x/y。

解：因为被除数 x 和除数 y 的数值位数都为 5 位，所以在 x 前面加上 5 个 0，使其长度扩展为 $2\times5=10$，得：$|x|=\underline{00}\ 0000011011$，$|y|=\underline{00}\ 00010$，$[-|y|]_{补}=\underline{11}\ 11110$，商符 $q_f=x_f\oplus y_f=1\oplus0=1$。

被除数高位	被除数低位 上商	说明
0000000	1 1 0 1 1 0	初始余数为被除数
+ 1111110		减y
1111110	1 1 0 1 1 0	余数为负，上商为0
← 1111101	1 0 1 1 0 0	左移一位
+ 0000010		加y
1111111	1 0 1 1 0 0	余数为负，上商为0
← 1111111	0 1 1 0 0 0	左移一位
+ 0000010		加y
0000001	0 1 1 0 0 1	余数为正，上商为1
← 0000010	1 1 0 0 1 0	左移一位
+ 1111110		减y
0000000	1 1 0 0 1 1	余数为正，上商为1
← 0000001	1 0 0 1 1 0	左移一位
+ 1111110		减y
1111111	1 0 0 1 1 0	余数为负，上商为0
← 1111111	0 0 1 1 0 0	左移一位
+ 0000010		加y
0000001	0 0 1 1 0 1	余数为正，上商为1

因为 $q_f=1$，所以商$[q]_{原}=[x/y]_{原}=101101$。得：$x/y=-1101$，余数$[r]_{原}=100001$，$r=-1$。

例 3.17 设 $n=5$，$x=+567$，$y=+27$，用原码不恢复余数法求 x/y。

解：$x=(+567)_{10}=(+1000110111)_2$，$y=(+27)_{10}=(+11011)_2$，可见被除数 x 的长度为 $10=2\times5$，除数 y 的长度为 5，应采用双精度除法。

有$|x|=\underline{00}\ 1000110111$，$|y|=\underline{00}\ 11011$，$[-|y|]_{补}=\underline{11}\ 00101$。

被除数高位	被除数低位 上商	说明
0010001	1 0 1 1 1 0	初始余数为被除数
+ 1100101		减y
1110110	1 0 1 1 1 0	余数为负，上商为0
← 1101101	0 1 1 1 0 0	左移一位
+ 0011011		加y
0001000	0 1 1 1 0 1	余数为正，上商为1
← 0010000	1 1 1 0 1 0	左移一位
+ 1100101		减y
1110101	1 1 1 0 1 0	余数为负，上商为0
← 1101011	1 1 0 1 0 0	左移一位
+ 0011011		加y
0000110	1 1 0 1 0 1	余数为正，上商为1
← 0001101	1 0 1 0 1 0	左移一位
+ 1100101		减y
1110010	1 0 1 0 1 0	余数为负，上商为0
← 1100101	0 1 0 1 0 0	左移一位
+ 0011011		加y
0000000	0 1 0 1 0 1	余数为正，上商为1

因为 $q_f=x_f\oplus y_f=0\oplus0=0$，所以商$[q]_{原}=[x/y]_{原}=010101$，$x/y=(+10101)_2=(+21)_{10}$，余数$[r]_{原}=000000$，$r=0$。

由例 3.16、例 3.17 可见，在进行双精度除法时，双字长的被除数的低字节部分在开始运算之前需要占用商的位置，在运算过程中随着商的左移，不断地与除数进行计算。

实现原码不恢复余数法的硬件逻辑结构如图 3-14 所示。图中 3 个寄存器 A、B、C 分别用于存放被除数、除数和商。对于单精度除法，在除法运算前，A 中存放的是被除数、B 中存放的

是除数，而 C 的初始值为 0；除法计算结束后，A 中存放的是余数、B 中存放的仍是除数，而 C 中存放的是商。对于双精度除法，在除法运算前，A 中存放的是被除数的高位、B 中存放的是除数，C 中存放的是被除数的低位；除法计算结束后，A 中存放的是余数、B 中的内容不变，C 中存放的是商。表 3-5 列出了在各种除法情况下寄存器的分配情况。

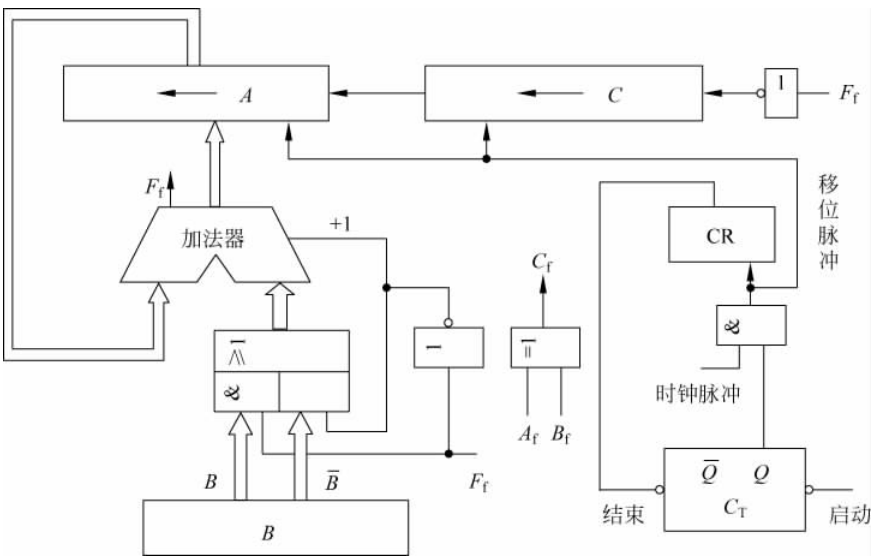


图 3-14　原码不恢复余数法的逻辑结构图

表 3-5　原码不恢复余数除法寄存器的分配

操作数类型		A 寄存器 初态　　终态	B 寄存器	C 寄存器 初态　　终态
定点小数	单字长	被除数→(部分余数)→余数	除数	0→商
	双字长	被除数高位→(部分余数)→余数	除数	被除数低位→商
定点整数	单字长	0→(部分余数)→余数	除数	被除数→商
	双字长	被除数高位→(部分余数)→余数	除数	被除数低位→商

为了便于控制上商，图 3-14 中将上商的位置固定在 C 的最低位，并要求在余数左移的同时，商数也随之向左移位，因此要求 C 寄存器具有左移功能。上商是由加法器的符号位 F_f 控制的。$F_f=0$，表示余数为正，经非门将 F_f 取反后，在 C 的最低位上商为 1，并控制下次做减法，即控制进行 $A+\overline{B}+1$；$F_f=1$，表示余数为负，取反后上商为 0，并控制下次做加法，即控制进行 $A+B$ 操作。C_T 为除法控制触发器，用于控制除法运算的开始与结束。$C_T=1$，允许发出左移移位脉冲，控制进行除法运算；$C_T=0$，不允许发出移位脉冲，停止进行除法运算。一般寄存器 A 不具有移位功能，加法器计算出的余数可以通过图 3-7 的电路斜送到寄存器 A 中来实现余数的左移。

3.4.2　补码除法运算

在补码除法运算中，参加运算的数均为补码数，与补码加、减、乘法一样，符号位参加运算，所得商也是补码形式。并且与原码除法类似，补码除法同样要求除数 $y\neq0$。如果进行定点小数除法，要求|被除数|<|除数|；如果进行定点整数除法，要求|被除数|>|除数|。

补码除法也可分为恢复余数法和不恢复余数法。因为后者用得较多,所以本书中只讨论补码不恢复余数除法的算法。下面以定点小数的补码除法为例进行讨论。

1. 补码不恢复余数除法

在进行补码除法时,需考虑以下问题。

1) 比较规则

比较规则用于判别被除数(或余数)减除数时是否够减。由于上商是根据比较被除数(或余数)与除数的绝对值的大小确定的,因此被除数$[x]_{补}$和除数$[y]_{补}$的大小比较就不能简单地用$[x]_{补}$减去$[y]_{补}$,它与操作数的符号有关。

例 3.18 被除数$[x]_{补}$和除数$[y]_{补}$的大小比较。

(1) 当 x 与 y 同号时,应作减法$[x]_{补}-[y]_{补}$进行比较。

① $x>0, y>0$ 且$|x|>|y|$;

设 $x=0.1011, y=0.1001$,则余数 $r=[x]_{补}-[y]_{补}=0.1011+1.0111=0.0010$。

这时表示$|x|-|y|$够减,余数与除数同号。

② $x>0, y>0$ 且$|x|<|y|$;

设 $x=0.1001, y=0.1101$,余数 $r=[x]_{补}-[y]_{补}=0.1001+1.0011=1.1100$。

这时$|x|-|y|$不够减,余数与除数异号。

③ $x<0, y<0$ 且$|x|>|y|$;

设 $x=-0.1011, y=-0.0011$,余数 $r=[x]_{补}-[y]_{补}=1.0101+0.0011=1.1000$。

这时表示$|x|-|y|$够减,余数与除数同号。

④ $x<0, y<0$ 且$|x|<|y|$;

设 $x=-0.0110, y=-0.1010$,余数 $r=[x]_{补}-[y]_{补}=1.1010+0.1010=0.0100$。

这时表示$|x|-|y|$不够减,余数与除数异号。

(2) 当 x 与 y 异号时,应作加法$[x]_{补}+[y]_{补}$进行比较。

① $x>0, y<0$ 且$|x|>|y|$;

设 $x=0.1011, y=-0.1101$,余数 $r=[x]_{补}+[y]_{补}=0.1011+1.1101=0.1000$。

这时实际表示$|x|-|y|$够减,且余数与除数异号。

② $x>0, y<0$ 且$|x|<|y|$;

设 $x=0.0110, y=-0.1101$,余数 $r=[x]_{补}+[y]_{补}=0.0110+1.0011=1.1001$。

这时实际表示$|x|-|y|$不够减,余数与除数同号。

③ $x<0, y>0$ 且$|x|>|y|$;

设 $x=-0.1011, y=0.0011$,余数 $r=[x]_{补}+[y]_{补}=1.0101+0.0011=1.1000$。

这时实际表示$|x|-|y|$够减,且余数与除数异号。

④ $x<0, y>0$ 且$|x|<|y|$。

设 $x=-0.0001, y=0.1001$,余数 $r=[x]_{补}+[y]_{补}=1.1111+0.1001=0.1000$。

这时实际表示$|x|-|y|$不够减,余数与除数同号。

根据例 3.18 的分析,可以归纳出被除数与除数绝对值大小的比较规则。若被除数与除数同号,则应通过减法比较它们绝对值的大小;若所得余数与除数同号,则表示够减;若所得余数与除数异号,则表示不够减。如果被除数与除数异号,则通过做加法比较其绝对值的大小;若所得余数与除数同号,表示不够减;若所得余数与除数异号,表示够减。表 3-6 列出了被除数与除数的比较规则。

表 3-6　比较与上商规则

$[x]_{补}$ 与 $[y]_{补}$	比较操作	余数 $[r]_{补}$ 与除数 $[y]_{补}$	上商
同号	$[x]_{补}-[y]_{补}$	同号，表示够减	1
		异号，表示不够减	0
异号	$[x]_{补}+[y]_{补}$	同号，表示不够减	1
		异号，表示够减	0

2）上商规则

参照原码除法的上商方法可推出补码除法的上商规则。如果被除数与除数同号，则商为正，上商方法与原码相同，余数与除数够减，上商为 1；不够减，上商为 0。如果被除数与除数异号，则商为负。由于负数的补码与原码存在“取反加 1”的关系，如不考虑末位加 1，则补码与原码的数值部分各位刚好相反，这时如余数与除数够减，对原码应上商 1，而补码则应上商 0；若不够减，则补码应上商 1。把这一规则与表 3-6 综合起来，得到补码除法的上商规则为每次加减所得余数与除数同号，则上商为 1；余数与除数异号，则上商为 0。

3）商符的确定

因为补码除法中，被除数与除数的符号参加运算，所得的商也是补码，因此商的符号是在求商的过程中自动形成的。在运算过程中，第一次比较上商的结果，实际就是商的正确符号。这是因为在除法过程中已判别溢出，因此第一次被除数加减除数肯定是不够减的，这样若被除数与除数同号，商应为正，被除数减除数因不够减，所得余数必与除数异号，按前面上商规则上商为 0，刚好为正商的符号；若被除数与除数异号，商为负，被除数加除数因不够减，所得余数必与除数同号，则上商为 1，刚好为负商的符号。因此，商符的确定与其他数值位上商规则完全相同。

根据商符的确定方法可知，商的符号也可以用于判断商是否溢出。例如，当被除数 $[x]_{补}$ 与除数 $[y]_{补}$ 同号时，如果余数 $[r]_{补}$ 与 $[y]_{补}$ 同号，且上商为 1，则表示商溢出；当被除数 $[x]_{补}$ 与除数 $[y]_{补}$ 异号时，如果余数 $[r]_{补}$ 与 $[y]_{补}$ 异号，且上商为 0，则表示商溢出。

4）求新余数

在补码不恢复余数法除法中，求新余数的方法与原码不恢复余数法相类似。

若被除数与除数同号，则做减法，所得余数与除数同号，表示够减，因此，将余数左移一位，减去除数，求得新余数；如果所得余数与除数异号，表示不够减，因此将余数左移一位，加上除数，求得新余数。

若被除数与除数异号，则做加法，所得余数与除数异号，表示够减，余数左移一位后仍加上除数，求得新余数；若所得余数与除数同号，表示不够减，按不恢复余数法，左移一位后应减去除数，求得新余数。

综上分析，得到求新余数的规则，如表 3-7 所示。

表 3-7　新余数规则

$[r_i]_{补}$ 与 $[y]_{补}$	商	新余数 $[r_{i+1}]_{补}$
同号	1	做减法 $[r_{i+1}]_{补}=2[r_i]_{补}+[-y]_{补}$
异号	0	做加法 $[r_{i+1}]_{补}=2[r_i]_{补}+[y]_{补}$

5) 商的校正

从前面上商规则可以看出,补码除法实质是按反码上商。如果商为正,则原码、补码、反码均相同,所得商是正确的;如果商为负,因负数反码与补码相差末位的1,因此按反码上商得到的补码商,就存在一定的误差。常用的处理方法有两种:

(1) 末位恒置1法

即最末位商不是通过比较上商,而是固定置为1。这种方法简单、容易,其最大误差为2^{-n}(对定点小数而言),所以在精度要求不高的情况下,通常都采用此方法。

(2) 校正法

在精度要求较高的情况下通常采用校正法。校正法的方法是:

① 若在所要求的位数内能够除尽,则除数为正时,商不必校正;除数为负时,商加2^{-n}校正;

② 若在所要求的位数内不能除尽,则商为正时,不必校正;商为负时,商加2^{-n}校正。

有关校正法的说明如下:

(1) 当在所要求的位数内不能除尽时,即$[r_n]_{补}\neq 0$且任一步$[r_i]_{补}\neq 0$时,若商为正,商的反码与补码相同,不必修正;若商为负,形成反码商后,应在末位(2^{-n}位)加1,即加2^{-n},才是商的补码。

(2) 当在所要求的位数内能够除尽时,即$[r_n]_{补}=0$或任一步$[r_i]_{补}=0$,除尽那一步的上商,将根据除数的正、负不同而不同。设除数为B,根据$B>0$还是$B<0$,分别加以说明。

① $B>0$时,若除尽那一步除法所得的余数$[r_i]_{补}=0$,由于r_i的符号位为正,所以上商为1,按补码除法规则,下一步除法的余数为:

$$[r_{i+1}]_{补}=2[r_i]_{补}+[-B]_{补}=[-B]_{补}$$

由于$[r_{i+1}]_{补}$与除数异号,所以上商为0,并且再下一步除法的余数为:

$$[r_{i+2}]_{补}=2[r_{i+1}]_{补}+[B]_{补}=2[-B]_{补}+[B]_{补}=[-B]_{补}$$

由于$[r_{i+2}]_{补}$仍与除数异号,所以仍然上商为0。

以此类推,直到$[r_n]_{补}$为止,以后各位商均为0。可见,在所要求的位数内能够除尽时,若除数为正,则除尽那位上商为1,以后各位商上0,商是正确的,不必修正。

② $B<0$时,若除尽那一步$[r_i]_{补}=0$,由于r_i的符号位为正并与除数异号,因此除尽那一步上商为0,按补码除法规则,下一步除法的余数为:

$$[r_{i+1}]_{补}=2[r_i]_{补}+[B]_{补}=[B]_{补}$$

由于$[r_{i+1}]_{补}$与除数同号,所以上商为1,并且再下一步除法的余数为:

$$[r_{i+2}]_{补}=2[r_{i+1}]_{补}+[-B]_{补}=[B]_{补}$$

由于$[r_{i+2}]_{补}$与除数同号,所以仍然上商为1。

以此类推,直到$[R_n]_{补}$为止,以后各位均商1。

可见,在所要求的位数内能够除尽且除数为负时,不论商为正或负,除尽那位上商为0,以后各位上商为1,将商加上2^{-n},正好修正为正确的商。

综合上面的讨论,可得补码不恢复余数除法的运算规则:

(1) 被除数与除数同号,则被除数减去除数;被除数与除数异号,则被除数加上除数。

(2) 若所得余数与除数同号,则上商为1,余数左移一位减去除数;若所得余数与除数异号,则上商为0,余数左移一位加上除数。

(3) 重复第(2)步,若采用末位恒置1法,则共做n次;若采用校正法,共做$n+1$次。

由于运算过程中，对除数的加减运算是交替进行，所以补码不恢复余数除法也称补码加减交替法。

例 3.19　已知 $x=-0.1011$，$y=-0.1101$，用补码不恢复余数法求 x/y。

解：$[x]_{补}=11.0101$，$[y]_{补}=11.0011$，$[-y]_{补}=00.1101$。

被除数(余数)	上商	
11.0101	0.0000	
+ 00.1101		x,y 同号，$[x]_{补}-[y]_{补}$
00.0010	0.0000	余数 r 与 y 异号，上商为 0
← 00.0100	0.0000	左移一位，加 y
+ 11.0011		
11.0111	0.0001	余数 r 与 y 同号，上商为 1
← 10.1110	0.0011	左移一位，减 y
+ 00.1101		
11.1011	0.0011	余数 r 与 y 同号，上商为 1
← 11.0110	0.0111	左移一位，减 y
+ 00.1101		
00.0011	0.0111	余数 r 与 y 异号，上商为 0
← 00.0110	0.1110	左移一位。若采用末位恒置 1 法，到此结束
+ 11.0011		若采用校正法，继续运算，加 y
11.1001	0.1110	余数 r 与 y 同号，上商为 1。商为正，不必校正

得 $[x/y]_{补}=0.1101$，$x/y=+0.1101$，$[r]_{补}=1.1001\times2^{-4}$，$r=-0.0111\times2^{-4}$。

例 3.20　已知 $x=0.10101$，$y=-0.11110$，用补码不恢复余数法求 $[x/y]_{补}$。

解：$[x]_{补}=00.10101$，$[y]_{补}=11.00010$，$[-y]_{补}=00.11110$。

经运算得 $[x/y]_{补}=1.01001$（末位恒置 1 法）或 $[x/y]_{补}=1.01010$（校正法），余数 $[r]_{补}=0.01100\times2^{-5}$。

被除数(余数)	上商	
00.10101	0.0000	
+ 11.00010		x,y 异号，$[x]_{补}+[y]_{补}$
11.10111	0.0000	余数 r 与 y 同号，上商为 1
← 11.01110	0.0001	左移一位，减 y
+ 00.11110		
00.01100	0.0001	余数 r 与 y 异号，上商为 0
← 00.11000	0.0010	左移一位，加 y
+ 11.00010		
11.11010	0.0010	余数 r 与 y 同号，上商为 1
← 11.10100	0.0101	左移一位，减 y
+ 00.11110		
00.10010	0.0101	余数 r 与 y 异号，上商为 0
← 01.00100	0.1010	左移一位，加 y
+ 11.00010		
00.00110	0.1010	余数 r 与 y 异号，上商为 0
← 00.01100	1.0100	左移一位，若采用末位恒置 1 法，到此结束
+ 11.00010		若采用校正法，继续运算，加 y
11.01110	1.0100	余数 r 与 y 同号，上商为 1
+ 00.11110	+ 0.0000	商为负，加 2^{-n} 校正；同时要恢复余数，减 y
00.01100	1.0101	

在除法运算中，一般情况下是不需要保留余数的。在采用末位恒置 1 法时，不需要求得余数。在采用校正法时，如需保留余数，则当最后一次运算所得余数仍为不够减时，就进行恢复余数操作，以恢复正确余数，如例 3.20 所示。

2. 布斯除法

在补码不恢复余数法的算法中,被除数与除数的计算和余数与除数的计算规则不同,不便于控制。如果把被除数当作初始余数看待,采用余数与除数的计算方法和上商规则,就可把补码不恢复余数除法规则的第一步与第二步统一起来,更加便于控制。采用这种思想的补码除法就是布斯除法。布斯除法的规则如下:

(1) 余数(初始为被除数)与除数同号,上商为 1,余数左移一位,减去除数;

余数(初始为被除数)与除数异号,上商为 0,余数左移一位,加上除数;

(2) 重复上述步骤,直到求得所需位数为止;

(3) 将商符变反,若采用校正法,则对商校正。

例 3.21 已知 $x=-0.1011$,$y=+0.1101$,用布斯除法求$[x/y]_{补}$。

解:$[x]_{补}=11.0101$,$[y]_{补}=00.1101$,$[-y]_{补}=11.0011$。

被除数(余数)	上商	
11.0101	0.000 0	x,y 异号,上商为 0
10.1010	0.000 0	左移一位,加 y
+ 00.1101		
11.0111	0.000 0	余数 r 与 y 异号,上商为 0
10.1110	0.000 0	左移一位,加 y
+ 00.1101		
11.1011	0.000 0	余数 r 与 y 异号,上商为 0
11.0110	0.000 0	左移一位,加 y
+ 00.1101		
00.0011	0.000 1	余数 r 与 y 同号,上商为 1
00.0110	0.001 [1]	左移一位。若采用末位恒置 1 法,到此结束
+ 11.0011		若采用校正法,继续运算,减 y
11.1001	0.001 0	余数 r 与 y 异号,上商为 0
	1.001 0	将商符变反
	+ 0.000 1	商为负,需加 2^{-n} 进行校正
	1.001 1	

得$[x/y]_{补}=1.0011$。

图 3-15 给出了实现布斯除法的硬件逻辑结构。图中 3 个寄存器 A、B、C 分别用于存放被除数、除数和商。在各种除法情况下,寄存器的分配情况与原码不恢复余数法的情况类似。对于单精度除法,在除法运算前,A 中存放的是被除数、B 中存放的是除数,C 的初始值为 0;除法计算结束后,A 中存放的是余数、B 中存放的仍是除数,C 中存放的是商。对于双精度除法,在除法运算前,A 中存放的是被除数的高位、B 中存放的是除数,C 中存放的是被除数的低位;除法计算结束后,A 中存的是余数、B 中的内容不变,C 中存放的是商。图 3-15 中上商由被除数符号 A_f和除数符号 B_f控制。$A_f \oplus B_f=0$ 时,表示余数与除数同号,经非门取反后,在 C 的最低位上商为 1,并控制下次做减法,即控制进行 $A+\overline{B}+1$;$A_f \oplus B_f=1$ 时,表示余数为负,取反后上商为 0,并控制下次做加法,即控制进行 $A+B$ 操作。因为运算结束后需要将商符取反,所以最终的商符应由 C 的最高位取反得到。一般寄存器 A 不具有移位功能,加法器计算出的余数可以通过图 3-7 的电路斜送到寄存器 A 中来实现余数的左移。

补码除法也可用于整数运算,但在进行整数除法运算时,需要将单字长的被除数扩展为双字长,因此要按照补码规则对被除数按符号进行扩展。即被除数为正,高位应补充为全 0;被除数为负,高位应补充为全 1。例如,设机器字长为 8 位,$[x]_{补}=10100000$,如果需要将$[x]_{补}$

扩展为双字长则按照补码的符号扩展规则，应将高位补充为全 1，即所得的双字长$[x]_{补}$=1111111110100000。在寄存器初值的安排时，必须注意符号扩展问题，若单字长的被除数为正，则寄存器 A 的初值为全 0；若单字长的被除数为负，则寄存器 A 的初值为全 1。

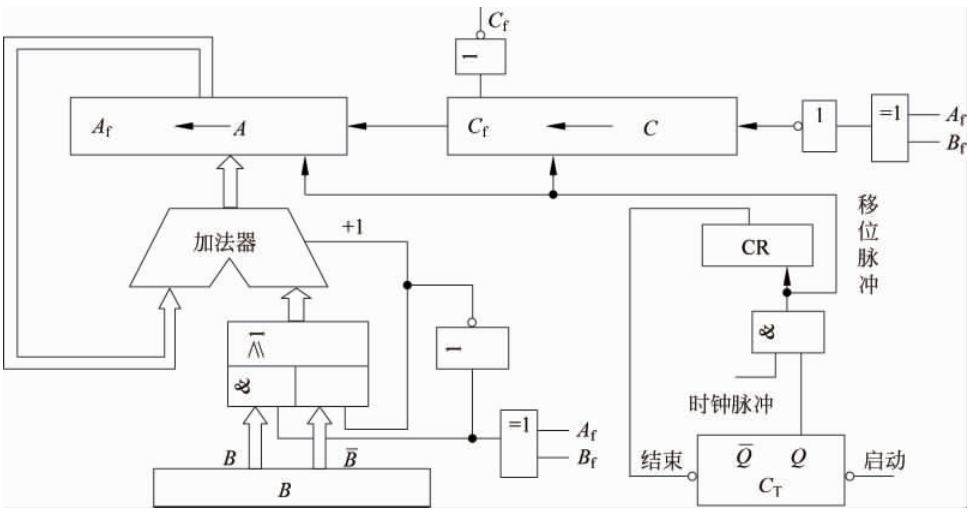


图 3-15　布斯除法的硬件逻辑结构

3.4.3　阵列除法器

与阵列乘法器类似，阵列除法器的思想是利用多个加减单元组成除法阵列，将除法各步的加减、移位操作在一个节拍内完成，从而提高除法运算速度。

1. 可控加减单元(CAS)

构成阵列除法器的基本电路是可控加减单元(CAS)，其逻辑结构如图 3-16 所示。CAS 单元有 4 个输入端和 4 个输出端。其中 P 用于控制加减运算；A_i、B_i为操作数；C_i为相邻低位的进位或借位；S_i 为运算结果；C_{i-1}为向相邻高位的进位或借位。当控制信号 $P=0$ 时，CAS 作为全加器单元；当 $P=1$ 时，输入 B_i被变反，CAS 作为减法单元。CAS 单元电路的输入输出逻辑关系为：

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

$$C_{i-1} = (A_i + C_i)(B_i \oplus P) + A_i C_i \qquad (3\text{-}18)$$

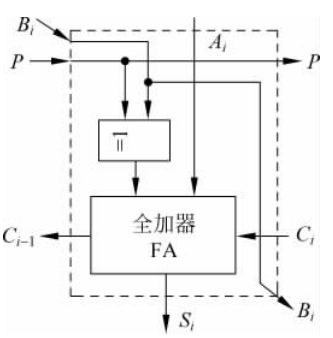


图 3-16　可控加减单元逻辑结构

在式(3-18)中，当 $P=0$ 时，S_i和C_{i-1}变为：

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

$$C_{i-1} = A_i B_i + B_i C_i + A_i C_i \qquad (3\text{-}19)$$

式(3-19)恰好是全加器的逻辑表达式，所以 $P=0$ 时，CAS 单元实现的是加法运算。

当 $P=1$ 时，S_i和C_{i-1}变为：

$$S_i = A_i \oplus \overline{B}_i \oplus C_i$$

$$C_{i-1} = A_i \overline{B}_i + \overline{B}_i C_i + A_i C_i \qquad (3\text{-}20)$$

根据 3.2.3 节介绍的补码加减运算电路(图 3-1)，可知 $P=1$ 时，CAS 单元实现的是减法运算。此时 C_i为相邻低位的借位输入，C_{i-1}为向相邻高位的借位输出。运算时，将控制信号

$P=1$ 接到最低位 CAS 单元的 C_i,即可实现补码减法要求的末位加 1 的功能。

2. 不恢复余数除法阵列除法器

设被除数 $x=0.x_1x_2x_3x_4x_5x_6$(双字长),除数 $y=0.y_1y_2y_3$,且 $x<y$,$y>0$,x/y 的商为 $q=0.q_1q_2q_3$,余数 $r=0.000r_3r_4r_5r_6$,则利用 CAS 单元组成的实现上述运算的不恢复余数除法阵列除法器逻辑结构如图 3-17 所示。

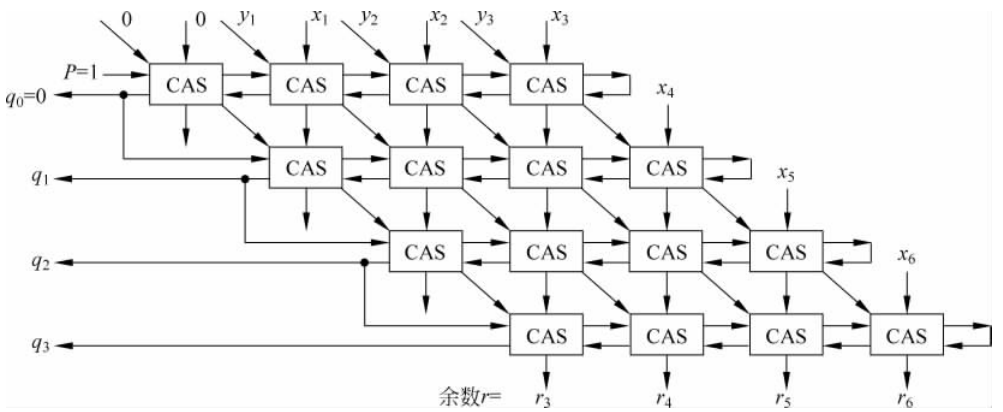


图 3-17 不恢复余数除法阵列除法器逻辑结构

因为 $x<y$,所以图 3-17 所示的阵列除法器第一行(最上面一行)的 CAS 单元所执行的初始操作是减法,因此将第一行的控制线 P 固定置为 1,这时 P 直通最右端 CAS 单元上的反馈线用作初始的进位输入,即实现了最低位上加 1。因为第一次的余数 $r=x-y<0$,所以商 $q_0=0$。在阵列除法器中,减法是补码运算实现的。考虑在做 $x-y$ 的过程中,当 $x<y$ 时,$x-y$ 将发生借位。由于是用 $+[-y]_{补}$ 实现减法,而 $[x]_{补}+[-y]_{补}=x+2+y<2$,不会发生进位,实际是产生了借位。所以在阵列除法器中,每一行最左边的 CAS 单元的进位输出可以决定商的数值,同时将当前的商反馈到下一行,就可确定下一行的操作是加法还是减法。

在图 3-17 中只需另外增加一级异或门来求商的符号,即可用于原码的阵列除法。类似于带符号数阵列乘法器,在图 3-17 电路的操作数输入端增加算前求补电路、在商的输出端增加算后求补电路,即可实现补码阵列除法。

3.5 浮点四则运算

由于浮点数比定点数表示范围大、有效精度高,更适合于科学与工程计算的要求,因此计算机中除了能够实现定点加减乘除四则运算外,通常还要求能够实现浮点四则运算。在第 2 章中我们了解到,浮点数据包括尾数和阶码两部分,尾数代表数的有效数字,一般用定点小数表示;阶码代表数的小数点实际位置,一般用定点整数表示,因此在浮点运算中,阶码与尾数需分别进行运算。这样,浮点运算实质上可以归结为定点运算。为了能保留更多的有效数字和使浮点数的表示唯一,计算机中一般都采用规格化的浮点运算,即要求参加运算的数都是规格化的浮点数,运算结果也应进行规格化处理。

3.5.1 浮点加减运算

设有两个规格化浮点数 x 与 y,分别为 $x=S_x\times 2^{e_x}$,$y=S_y\times 2^{e_y}$。其中,S_x、S_y 分别为数 x、

y 的尾数，e_x、e_y 分别为数 x、y 的阶码。实现两个浮点数的加减运算，一般需要对阶、尾数加减、结果规格化、尾数舍入 4 个步骤。

1. 对阶

浮点数的小数点实际位置是由阶码表示的，若两个规格化的浮点数的阶码不相等，则两数小数点的实际位置就不同，因而也就不能对它们的尾数直接进行加减运算。要进行两个浮点数的加减运算，首先必须把两数的小数点对齐。在浮点运算中，使两个浮点数的阶码取得一致的过程称为对阶。

对阶的标志是使两个浮点数阶码相等。对阶的方法首先是求出两数阶码之差，即：

$$\Delta e = e_x - e_y \tag{3-21}$$

若 $\Delta e=0$，表示两数阶码相等，小数点已经对齐；若 $\Delta e>0$，则表示 $e_x>e_y$；若 $\Delta e<0$，则表示 $e_x<e_y$。

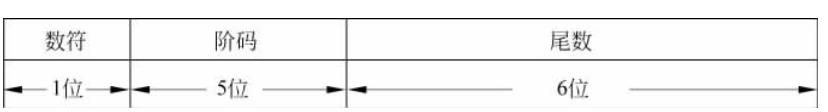
当阶差 $\Delta e\neq 0$ 时，需进行对阶移位，即通过尾数移位，改变阶码，使两数阶码相等。

根据浮点表示的规则，在保证数值不变的条件下，浮点数的尾数每向右移一位，阶码加 1，尾数每向左移一位，阶码减 1。因此对阶既可以通过将阶码小的数的尾数向右移位，阶码增量，直到等于阶码大的数的阶码为止；也可以通过将阶码大的数的尾数向左移位，阶码减量，直到等于阶码小的数的阶码为止。但是，由于在规格化的浮点运算中，参加运算的浮点数均为规格化尾数，尾数左移将引起数值最高有效位的丢失，从而造成很大的误差；而尾数右移丢失的是数值的最低有效位，造成的误差小。所以对阶的基本方法是：小阶向大阶看齐，即将阶码小的数的尾数向右移位，每右移一位，阶码加 1，直到两数的阶码相等为止。右移位数等于两数阶码之差 $|\Delta e|$。

2. 尾数求和(差)

对阶完毕，两数阶码相等，即可进行尾数的加减运算。若求和，则将两数尾数直接相加；若求差，则将对阶后的减数的尾数变补与被减数的尾数相加。因为浮点数的尾数为定点小数，所以尾数的加减运算规则与定点加减运算规则相同。根据加减运算阶码不变的原则，和差的阶码与对阶后的阶码即两数中的大阶相等。

例 3.22　设某机浮点数格式为：



阶码和尾数均采用补码表示。

已知　$x=+0.110101\times 2^{+0011}$，$y=-0.111010\times 2^{+0010}$，求 $x\pm y$。

解：把 x、y 转换成机器数形式，得：

$$x = 0\ 00011\ 110101,\quad y = 1\ 00010\ 000110$$

首先进行对阶，求阶差：$[\Delta e]_{补}=00011+11110=00001$，因为 Δe 为正，所以 $e_x>e_y$。

由于 $|\Delta e|=1$，根据小阶对大阶的原则，把 y 的尾数右移一位(按补码移位规则)，阶码加 1，得到 y 的阶码为 00011，与 x 的阶码相等。对阶后，$y=1\ 00011\ 100011$。

① 作 $x+y$ 时，将 x、y 的尾数相加。

得 $[S_{x+y}]_{补}=0.011000$。

取 x，y 的大阶作为和的阶码，得 $[x+y]_{补}$ 的机器数形式为 $[x+y]_{补}=0\ 00011\ 011000$。

② 做 $x-y$ 时，将尾数相减，即将 $[-y]_{补}$ 的尾数与 x 的尾数相加。

在计算$[S_{x-y}]_{补}$过程中,进入符号位和符号位输出的进位不一致,计算结果发生了溢出。

$$\begin{array}{rr} [S_x]_{补}= & 0\ 110101 \\ +\ [S_y]_{补}= & 1\ 100011 \\ \hline [S_{x+y}]_{补}= & 0\ 011000 \end{array} \qquad \begin{array}{rr} [S_x]_{补}= & 0\ 110101 \\ +\ [-S_y]_{补}= & 0\ 011101 \\ \hline [S_{x-y}]_{补}= & 1\ 010010 \end{array}$$

3. 结果规格化

在规格化浮点运算中,若运算结果不是规格化数,则必须进行规格化处理。根据浮点规格化数的定义可知,对于基值为 2 的浮点数,若尾数 s 采用原码表示,则满足 $1/2\leqslant|s|<1$ 的数为规格化数;在补码表示中,满足 $-1\leqslant s<-1/2$ 和 $1/2\leqslant s<1$ 的数为规格化数。如果运算结果不满足上述条件,则称为破坏规格化。

当尾数运算结束后,需要进行尾数的规格化判断,通常运算结果破坏规格化有两种情况。第一种是尾数的运算结果发生溢出,称为向左破坏规格化;另一种情况是尾数的运算结果未发生溢出,但不满足规格化条件,称为向右破坏规格化。

设浮点数的尾数采用原码表示,$[s]_{原}=s_f.s_1s_2\cdots s_n$。如果尾数发生溢出,则为向左破坏规格化;如果尾数未发生溢出,但 $s_1=0$,则为向右破坏规格化。

设尾数用补码表示,$[s]_{补}=s_f.s_1s_2\cdots s_n$。如果尾数发生溢出,则称为向左破坏规格化;如尾数未溢出,但 $s_f\oplus s_1=0$,即 s_f 与 s_1 相同,则为向右破坏规格化。

为了便于判断溢出,尾数可采用变形补码表示。设尾数为$[s]_{补}=s_{f1}s_{f2}.s_1s_2\cdots s_n$。如果 $s_{f1}\oplus s_{f2}=1$,则表示尾数发生溢出,即结果向左破坏规格化;如果$\bar{s}_{f1}\bar{s}_{f2}\bar{s}_1\oplus s_{f1}s_{f2}s_1=1$,即 s_{f1}、s_{f2} 与 s_1 相同,则表示尾数未溢出,但符号位与最高数值位相同,结果向右破坏规格化。

当运算结果出现向左破坏规格化时,必须进行向右规格化(也称右规)。右规时,需将尾数向右移位(要按照原码和补码的右移规则进行移位),每移一位,阶码加 1,一直移位到满足规格化要求为止。当运算结果出现向右破坏规格化时,必须进行向左规格化(也称左规)。左规时,将尾数向左移位,每移一位,阶码减 1,一直移位到满足规格化要求为止。

在例 3.22 中,计算 $x+y$ 时,尾数的运算结果$[s_{x+y}]_{补}=0.011000$,由于参加运算的 x、y 异号,根据运算结果可知尾数没有发生溢出。但因为 $s_f\oplus s_1=0$,所以运算结果出现了向右破坏规格化,需要进行向左规格化。将尾数向左移一位,阶码减 1,得尾数$[s_{x+y}]_{补}=0.110000$,阶码 $e_{x+y}=00011-1=00010$,因此规格化后,$[x+y]_{补}=0\ 00010\ 110000$,得:$x+y=+0.110000\times2^{+0010}=+0.110000\times2^{+2}$。

在 $x-y$ 的运算过程中,尾数的运算结果发生了溢出。如果在定点运算中,结果溢出将不能继续运算。但在浮点运算中,运算结果出现了溢出,即表示发生了运算结果向左破坏规格化,可以通过对运算结果进行向右规格化而获得正确结果。右规格化时,将尾数向右移一位,阶码加 1,得尾数$[s_{x-y}]_{补}=0.101001$,阶码 $e_{x-y}=00011+1=00100$,因此规格化后,$[x-y]_{补}=0\ 00100\ 101001$,得:$x-y=+0.101001\times2^{+0100}=+0.101001\times2^{+4}$。

4. 舍入

浮点运算过程中,当对阶操作时小阶对应的尾数需右移和运算结果需右规时,都将尾数的低位移出。为减少因尾数右移而造成的误差,提高运算精度,需要进行舍入处理。

计算机中对采用的舍入方法有两个要求,第一是单次舍入引起的误差不超过所允许的范围,一般要求不大于保留的尾数最低位的位权,即 2^{-n};第二是误差应有正有负,使得多次舍入不会产生积累误差。常用的舍入方法有以下几种。

1）截断法(恒舍法)

截断法是：将右移移出的值直接舍去。该方法简单，精度较低。

2）0 舍 1 入法

0 舍 1 入法是：若右移时被丢掉数位的最高位为 0，则舍去；若右移时被丢掉数位的最高位为 1，则将 1 加到保留的尾数的最低位。

0 舍 1 入法类似于十进制数的四舍五入。主要优点是单次舍入引起的误差小，精度较高；缺点是加 1 时需多做一次运算，而且可能造成尾数溢出，需要再次右规。

3）末位恒置 1 法

末位恒置 1 法也称冯·诺依曼舍入法。其方法是：尾数右移时，无论被丢掉的数位的最高位为 0 还是为 1，都将保留的尾数的最低位恒置为 1。

末位恒置 1 法的主要优点是舍入处理不用做加法运算，方法简单、速度快且不会有再次右规的可能，并且没有积累误差，是常用的舍入方法。其缺点是单次舍入引起的误差较大。

4）查表舍入法(ROM 舍入法)

查表舍入法根据尾数的低 k 位的代码值及被丢掉数位的最高位值，按一定舍入规则编制成舍入表，并将该表存到只读存储器中。当需要舍入操作时，以尾数低 k 位及被丢掉数位的最高位作为 ROM 地址，查找舍入表，得到舍入后尾数的低 k 位值，如图 3-18 所示。

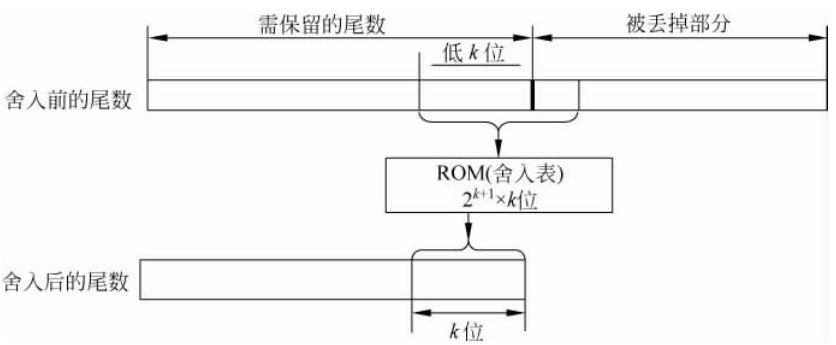


图 3-18　查表舍入法

舍入表编制原则是：若尾数低 k 位值不为全 1，则按 0 舍 1 入法编制；若尾数低 k 位值为全 1，则按截断法编制。

查表舍入法既具有 0 舍 1 入法的优点，又可以避免 0 舍 1 入法中的进位传送。

5）设保护位(guard bit)法

设保护位(guard bit)法是：在尾数后面设若干位保护位，运算时保护位与尾数一起参加运算和移位，运算结果根据保护位的值决定舍入。

尾　数	保护位

例如，DG-MV 系列机中，双精度浮点数的尾数为 56 位，设置了 8 位保护位，运算时，尾数加保护位共 64 位数据参加运算。运算结果的舍入规则为：

若保护位的值为 00H～7FH(0×××××××)，则采用截断法；

若保护位值为 80H(10000000)，则尾数最低位加到尾数最低位上；

若保护位值为 81H～FFH(1×××××××)，则尾数最低位上加 1。

这种方法可使误差小于尾数最低位的1/2。

例3.23 设$[x]_{原}=1.101010011$,$[y]_{补}=1.101010010$,要求保留8位数据(包括1位符号位)。请用恒舍法、0舍1入法、末位恒置1法进行舍入。

解:由于原码是符号代码化,所以对数值部分的舍入,只要根据舍去部分的最高位是否为1来决定舍入。

① 对于$[x]_{原}=1.101010011$

按恒舍法舍入后得 $[x]_{原}=1.1010100$

按0舍1入法舍入后得 $[x]_{原}=1.1010101$

按末位恒置1法舍入后得 $[x]_{原}=1.1010101$

② 对于$[y]_{补}=1.101010010$

按恒舍法舍入后得 $[y]_{补}=1.1010100$

按0舍1入法舍入后得 $[y]_{补}=1.1010100$

按末位恒置1法舍入后得 $[y]_{补}=1.1010100$

按0舍1入法对补码进行舍入时,需注意的是:当$y<0$时,若舍去部分的最高位为1,其余位为全0,则只将舍去部分全部舍去即可。因为对于$[y]_{补}=1.101010010$,其对应的真值$y=-0.010101110$,按0舍1入法对y舍入后得$y=-0.0101100$。将舍入后的y取补,得$[y]_{补}=1.1010100$。可见,只要将要舍去的10全部舍去即可,不需要进行0舍1入。

5. 浮点运算的溢出处理

与定点运算相同,浮点运算结束时,也需要进行溢出判断。

如第2章所述,如果浮点运算结果的阶码大于所能表示的最大正阶,则表示运算结果超出了浮点数所能表示的绝对值最大的数,进入了上溢区;如果浮点运算结果的阶码小于所能表示的最小负阶,则表示运算结果小于浮点数所能表示的绝对值最小的数,进入了下溢区。由于下溢时,浮点数的数值趋近于0,所以通常不作溢出处理,而是将其作为机器零处理。而当运算结果出现上溢时,表示浮点数真正溢出,通常需要将计算机中的溢出标志置1,转入溢出中断处理。可见浮点数的溢出通常是指浮点数上溢,并且浮点数是否溢出是由阶码是否大于浮点数所能表示的最大正阶来判断的。

例如,设浮点数的阶码采用补码表示,双符号位,这时浮点数的溢出与否可由阶码的符号进行判断:

若阶码$[j]_{补}=\underbrace{01}_{\text{符号}}\times\times\cdots\times$,则表示出现上溢,需作溢出处理;

若阶码$[j]_{补}=10\times\times\cdots\times$,则表示出现下溢,按机器零处理。

浮点加减运算的流程图,如图3-19所示。

3.5.2 浮点乘除运算

浮点乘除运算实质上是尾数和阶码分别按定点运算规则运算。

设有浮点数$x=s_x\times 2^{e_x}$,$y=s_y\times 2^{e_y}$,则:

浮点乘法为

$$x\times y=(s_x\times s_y)\times 2^{e_x+e_y} \tag{3-22}$$

浮点除法为

$$x/y=(s_x/s_y)\times 2^{e_x-e_y} \tag{3-23}$$

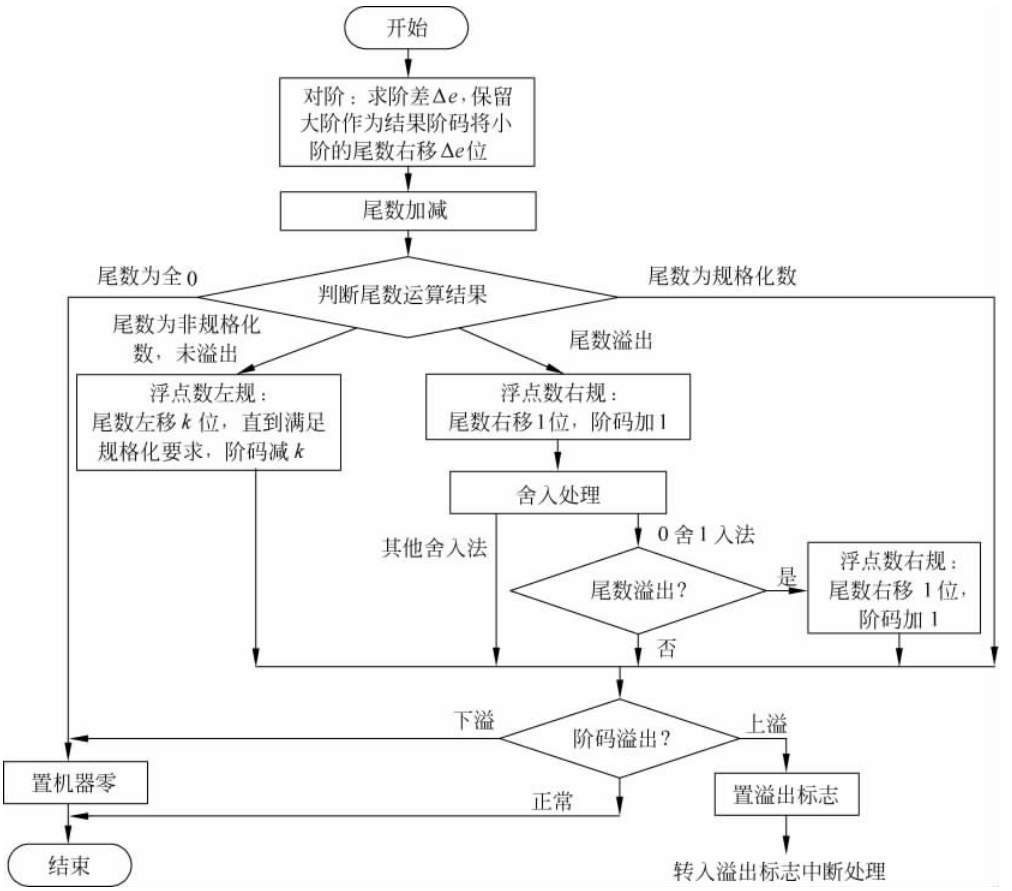


图 3-19　浮点加减运算流程图

1. 阶码运算及溢出判断

如果参加运算的浮点数的阶码采用补码表示，则乘积的阶码为被乘数与乘数阶码之和 $[e_x+e_y]_{补}=[e_x]_{补}+[e_y]_{补}$，商的阶码为被除数与除数阶码之差 $[e_x-e_y]_{补}=[e_x]_{补}-[e_y]_{补}$。我们可以根据补码运算的规则进行阶码运算，并按照补码的溢出条件判断阶码是否产生了溢出。

如果阶码采用移码表示，则阶码的加减运算必须按照移码的加减运算规则进行。下面讨论一下移码的加减运算规则。

根据移码的定义 $[x]_{移}=2^n+x(-2^n\leqslant x<2^n)$，有：

$[e_x]_{移}=2^m+e_x$，$[e_y]_{移}=2^m+e_y$（m 为不含阶符的阶码位数）

$$[e_x]_{移}+[e_y]_{移}=2^m+e_x+2^m+e_y=2^m+[2^m+(e_x+e_y)]=2^m+[e_x+e_y]_{移}$$

可见直接用移码求阶码之和，结果比两数之和的移码多了 2^m，即最高位上多加了一个 1，所以要求得两数和的移码，必须将两数移码之和的最高位（符号位）取反。

例 3.24　求 $[e_x+e_y]_{移}$。

① $[e_x]_{移}=10011$，$[e_y]_{移}=01001$；

② $[e_x]_{移}=01100$，$[e_y]_{移}=10101$。

解：① 因为 $[e_x]_{移}+[e_y]_{移}=10011+01001=11100$，

所以将符号位取反得$[e_x+e_y]_{移}=01100$。

② 因为 $[e_x]_{移}+[e_y]_{移}=01100+10101=00001$,

所以将符号位取反得$[e_x+e_y]_{移}=10001$。

由于补码与移码的数值位相同、符号位相反,因此可以将移码与补码混合使用,即利用 x 的移码与 y 的补码之和来表示 $x+y$ 的移码,如式(3-24)所示。

$$\begin{aligned}[e_x]_{移}+[e_y]_{补}&=2^m+e_x+2^{m+1}+e_y\\&=2^{m+1}+[2^m+(e_x+e_y)]=2^{m+1}+[e_x+e_y]_{移}\\&=[e_x+e_y]_{移}\quad(\mathrm{mod}\ 2^{m+1})\end{aligned}\tag{3-24}$$

同理可推出:

$$\begin{aligned}[e_x]_{移}+[-e_y]_{补}&=2^m+e_x+2^{m+1}+(-e_y)\\&=2^{m+1}+[2^m+(e_x-e_y)]=2^{m+1}+[e_x+e_y]_{移}\\&=[e_x-e_y]_{移}\quad(\mathrm{mod}\ 2^{m+1})\end{aligned}\tag{3-25}$$

根据式(3-24)、式(3-25),在进行移码加减运算时,应将加减数的移码符号位取反后进行加减。

为便于判断移码加减运算的溢出情况,采用双符号位进行运算。设移码的双符号位为 $s_{f1}s_{f2}$,并规定运算初始时,移码的第一符号位 s_{f1} 恒用 0 参加运算(注意:与双符补码不同)。移码加减运算的溢出判断方法是:若运算结果的第一符号位 s_{f1} 为 1,则表示溢出;若 s_{f1} 为 0,则表示未溢出。s_{f1} 与 s_{f2} 配合,可以表示运算结果的具体溢出情况,如式(3-26)所示。

$s_{f1}s_{f2}=00$,结果为负;$s_{f1}s_{f2}=01$,结果为正;

$s_{f1}s_{f2}=10$,结果上溢;$s_{f1}s_{f2}=11$,结果下溢。 (3-26)

图 3-20 显示了一个具有行波进位的实现移码加减的阶码加法器逻辑电路。图中两个操作数及所得结果均为移码表示的数:

$$[e_x]_{移}=e_{xn}e_{xn-1}\cdots e_{x1};\ [e_y]_{移}=e_{yn}e_{yn-1}\cdots e_{y1}$$

$$[e_x\pm e_y]_{移}=s_ns_{n-1}\cdots s_1$$

其中,e_{xn}、e_{yn}、s_n为移码的符号位。

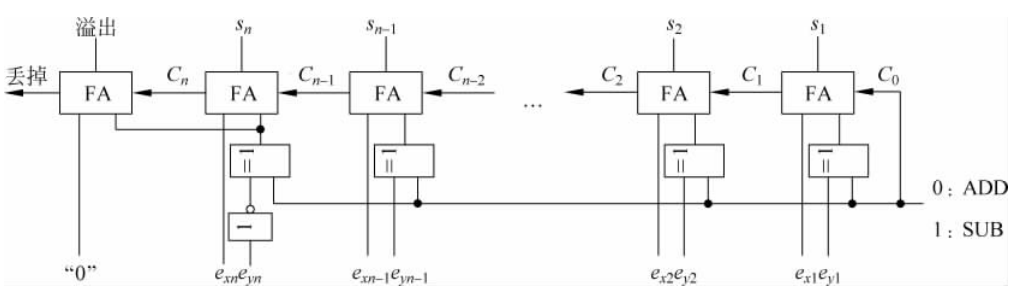


图 3-20 实现移码加减运算的阶码加法器逻辑电路

例 3.25 设不含阶符的阶码位数 $m=4$,求$[e_x\pm e_y]_{移}$。

① $e_x=+1001$,$e_y=+0101$;

② $e_x=+1010$,$e_y=-1001$;

③ $e_x=-1010$,$e_y=-1101$。

解:① 按照移码双符号位的规定,得:

$[e_x]_{移}=011001$,$[e_y]_{补}=000101$,$[-e_y]_{补}=111011$。

则:$[e_x+e_y]_{移}=[e_x]_{移}+[e_y]_{补}=011001+000101=011110$,$e_x+e_y=+1110$;

$[e_x-e_y]_{移}=[e_x]_{移}+[-e_y]_{补}=011001+111011=010100$，$e_x-e_y=+0100$。

② $[e_x]_{移}=011010$，$[e_y]_{补}=110111$，$[-e_y]_{补}=001001$。

则：$[e_x+e_y]_{移}=[e_x]_{移}+[e_y]_{补}=011010+110111=010001$，$e_x+e_y=+0001$；

$[e_x-e_y]_{移}=[e_x]_{移}+[-e_y]_{补}=011010+001001=100011$，运算结果上溢。

$e_x=-1010$，$e_y=-1101$。

③ $[e_x]_{移}=000111$，$[e_y]_{补}=110011$，$[-e_y]_{补}=001101$。

则：$[e_x+e_y]_{移}=[e_x]_{移}+[e_y]_{补}=000111+110011=111010$，运算结果下溢；

$[e_x-e_y]_{移}=[e_x]_{移}+[-e_y]_{补}=000111+001101=010100$，$e_x-e_y=+0100$。

2. 尾数运算

1) 浮点乘法尾数运算

由式(3-22)可知，在浮点乘法运算中，乘积的尾数是相乘的两个浮点数的尾数之积，并按定点小数的乘法规则进行运算。浮点乘法尾数运算的运算步骤一般为：

(1) 检测被乘数和乘数的尾数是否为0，若有一个为0，则乘积必然为0，不需再进行计算。只有当两数皆不为0时，方可进行运算。

(2) 被乘数和乘数的尾数相乘。根据尾数采用的是原码表示还是补码表示，可采用任意一种相应的定点小数乘法完成运算。

(3) 运算结果规格化。如果尾数乘积的绝对值小于1/2，则需对运算结果进行左规。如果在左规调整阶码时，出现阶码下溢，则应将运算结果作机器零处理。在规格化浮点乘法中，参加运算的尾数均为规格化尾数，因此尾数乘积的绝对值必然大于等于1/4，所以乘法运算结果最多只会做一次左规。如果尾数采用补码表示，由于-1是规格化数，而当两尾数均为-1时，由于$(-1)\times(-1)=1$，因此需要对运算结果进行一次右规，如果在右规调整阶码时，出现阶码上溢，则表示浮点数上溢，应转入溢出中断处理。注意，如果尾数采用原码表示，则乘法运算结果不会出现右规。

(4) 舍入处理。两个n位(除符号位外)尾数相乘，乘积为$2n$位。如果只需要取乘积的高n位，则需要对乘法运算结果进行舍入处理。

2) 浮点除法尾数运算

由式(3-23)可知，在浮点除法运算中，商的尾数是被除数尾数除以除数尾数之商，尾数按定点小数除法规则运算。如果运算结果不满足规格化要求或出现溢出情况，则需进行相应的处理。浮点除法尾数运算的运算步骤一般为：

(1) 检测被除数和除数的尾数是否为0，若被除数为0，商必然为0，不需再进行计算；若除数为0，则商为无穷大，转入除数0中断处理。只有当两数皆不为0时，方可进行运算。

(2) 被除数和除数的尾数相除。根据尾数采用的是原码表示还是补码表示，可采用任意一种相应的定点小数除法完成运算。由于在定点小数除法中，要求|被除数|<|除数|，因此当被除数和除数的尾数$|s_x|\geqslant|s_y|$时，需对被除数进行调整。由于在规格化浮点运算中，被除数和除数的尾数均为规格化数，所以只需将s_x右移一位，阶码加1，即可满足$|s_x|<|s_y|$，正常进行定点小数除法运算，且此时获得的商必为规格化定点小数。另一种方法是，先进行尾数的除法运算，此时运算结果必然溢出，然后按向左破坏规格化对结果进行右规处理。

(3) 运算结果规格化。如果商的绝对值小于1/2，则需对运算结果进行左规。如果在左规调整阶码时，出现阶码下溢，则应将运算结果作机器零处理。如果尾数采用补码表示，当被除数的尾数$s_x=1/2$，$[s_x]_{补}=(0.100\cdots0)_2$；除数的尾数$s_y=-1$，$[s_y]_{补}=(1.00\cdots0)_2$时，由于

$s_x/s_y=-1/2$,$[s_x/s_y]_{补}=(1.100\cdots0)_2$不是规格化数,所以必须对运算结果进行一次左规。而若尾数采用的是原码表示,则当$[s_x/s_y]_{原}=(1.100\cdots0)_2$时不需要进行左规。

例 3.26 设浮点数的阶码用移码表示,尾数用补码表示。已知两个浮点数:

$x=-0.1001\times2^5$,$y=+0.1011\times2^{-3}$,求 $x\times y$。

解:设阶码包括符号位为 4 位;尾数包括符号位为 5 位,因此可得:

x 的尾数$[s_x]_{补}=1.0111$,x 的阶码$[e_x]_{移}=1\ 101$;

y 的尾数$[s_y]_{补}=0.1011$,y 的阶码$[e_y]_{移}=0\ 101$。

① 阶码求和

因为 $[e_y]_{补}=1\ 101$,采用双符号位进行移码加法运算,得:

所以 $[e_x+e_y]_{移}=[e_x]_{移}+[e_y]_{补}=01101+11101=01010$。

结果的阶码的第一符号位 s_{f1} 为 0,则表示阶码无溢出。

② 尾数相乘

尾数可采用定点补码乘法(双符号位),得:

$$[s_x\times s_y]_{补}=[s_x]_{补}\times[s_y]_{补}=11.0111\times00.1011=11.10011101$$

因为尾数的符号与最高数值位相同,所以尾数需要进行向左规格化。

尾数左移 1 位,阶码减 1,得:

$[s_x\times s_y]_{补}=11.00111010$,$[e_x+e_y]_{移}+[-1]_{补}=01010+11111=01001$,若尾数取 8 位数值位,则:

$x\times y$ 的尾数的补码为:$[s_x\times s_y]_{补}=1.00111010$,$x\times y$ 的阶码为:$[e_x+e_y]_{移}=1001$。

所以 $x\times y=-0.11000110\times2^{+001}=-0.11000110\times2^{+1}$。

例 3.27 设浮点数的阶码用移码表示,尾数用补码表示。已知两个浮点数:

$x=-0.1011\times2^4$,$y=+0.1101\times2^{-3}$,求 x/y。

解:设阶码包括符号位为 4 位;尾数包括符号位为 5 位,因此可得:

x 的尾数$[s_x]_{补}=1.0101$,x 的阶码$[e_x]_{移}=1\ 100$;

y 的尾数$[s_y]_{补}=0.1101$,y 的阶码$[e_y]_{移}=0\ 101$。

① 阶码求差

因为 $[-e_y]_{补}=0\ 011$,采用双符号位进行移码加法运算,得:

所以 $[e_x-e_y]_{移}=[e_x]_{移}+[-e_y]_{补}=01100+00011=01111$。

结果的阶码的第一符号位 s_{f1} 为 0,则表示阶码无溢出。

② 尾数相除

尾数可采用定点补码除法,得:$[x/y]_{补}=[s_x]_{补}\div[s_y]_{补}=1.0101\div0.1101=1.0011$。

因为尾数的符号与最高数值位相异,所以尾数不需要进行规格化。

若尾数取 4 位数值位,x/y 的尾数的补码为$[s_x/s_y]_{补}=1.0011$,x/y 的阶码为$[e_x-e_y]_{移}=1111$。

所以 $x\times y=-0.1101\times2^{+111}=-0.1101\times2^{+7}$。

3.6 运算器的组成

3.6.1 定点运算器

1. 定点运算器的基本结构

如前所述,运算器的核心是算术/逻辑运算单元(ALU),但是作为一个完整的数据加工处

理部件，运算器中还需要有各类通用寄存器、累加器、多路选择器、状态/标志触发器、移位器和数据总线等逻辑部件，辅助 ALU 完成规定的工作。设计运算器的逻辑结构时，为了使各部件能够协调工作，主要需要考虑的是 ALU 和寄存器与数据总线之间传递操作数和运算结果的方式以及数据传递的方便性与操作速度。

根据运算器中各部件之间如何传递操作数和运算结果的方式以及总线数目的不同，可将运算器分为单总线结构、双总线结构和三总线结构，如图 3-21 所示。

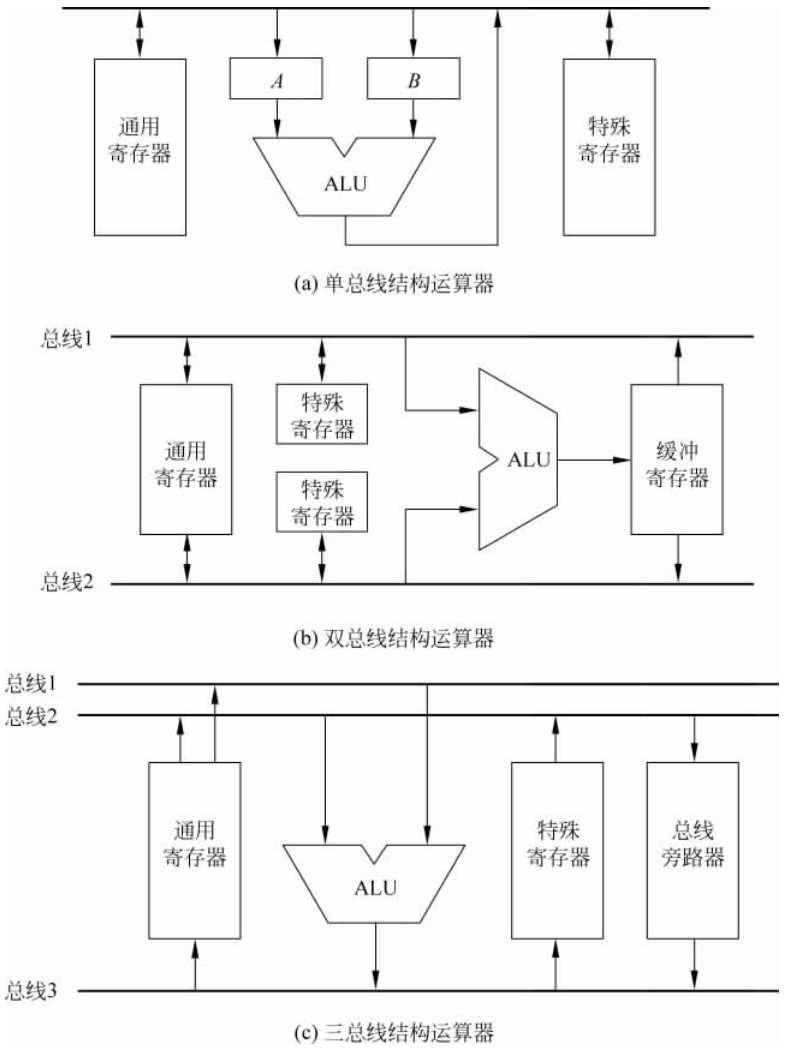


图 3-21　运算器的 3 种基本结构形式

1) 单总线结构运算器

图 3-21(a)为单总线结构运算器。单总线结构运算器的特点是所有部件都接在同一总线上。由于所有部件都通过同一总线传送数据，因此在同一时间内，只能有一个操作数放在单总线上，所以需 A、B 两个缓冲器。当执行双操作数运算时，首先把一个操作数送入缓冲器 A，然后把另一操作数送入缓冲器 B，只有两个操作数同时出现在 ALU 的输入端时，ALU 才能正确执行相应运算。运算结束后，再通过单总线将运算结果存入目的寄存器。单总线结构的主要缺点就是操作速度慢。

2) 双总线结构运算器

图 3-21(b)为双总线结构运算器。双总线结构运算器的特点是操作部件连接在两组总线上,可以同时通过两组总线传输数据。在执行双操作数运算时,可以将两个操作数同时加到 ALU 的输入端进行运算,一步完成操作并得到结果。但由于在输出 ALU 的运算结果时,两条总线都被输入的操作数占用着,运算结果不能直接加到数据总线上,所以需要利用输出缓冲器来暂存运算结果,等到下一个步骤,再将缓冲器中的运算结果通过总线送入目的寄存器。显然双总线结构运算器的执行速度比单总线结构运算器的执行速度快。

3) 三总线结构运算器

图 3-21(c)为三总线结构运算器。三总线结构运算器的特点是操作部件连接在三组总线上,可以同时通过三组总线传输数据。在执行双操作数运算时,由于能够利用三组总线分别接收两个操作数和 ALU 的运算结果,因此只需一步就可完成一次运算。与前两种结构相比较,三总线结构运算器的操作速度最快,不过其控制也更复杂。

在三总线结构运算器中,还可以设置一个总线旁路器。如果一个操作数不需要运算操作或修改,可通过总线旁路器直接从总线 2 传送到总线 3,而不必经过 ALU。

2. 定点运算器举例

由算术逻辑部件(ALU)、累加器(AC)、数据缓冲寄存器(MDR)可以组成最基本、最简单的运算器,如图 3-22 所示。图中运算器与存储器之间通过一条双向数据总线进行联系。从存储器中读取的数据,可经过数据寄存器(MDR)、ALU 存放到 AC 中;AC 中的信息也可经过 MDR 存入主存中指定的单元。运算器可以将 AC 中数据与主存某一单元的数据经 ALU 进行运算,并将结果暂存于 AC 中。

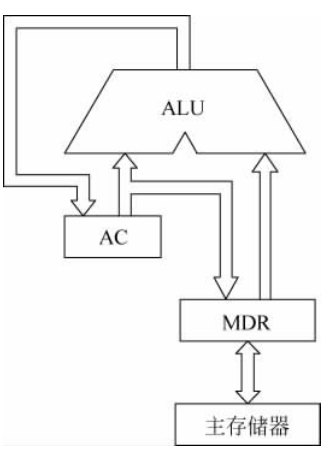


图 3-22 最简单的运算器

利用大规模集成电路技术(LSI)可以将 ALU 与寄存器集成为位片式结构的运算器芯片,如 Am2901A 就是一种 4 位的位片式结构运算器组件。图 3-23 为 Am2901A 的逻辑示意图。

Am2901A 运算器组件的特点:

(1) Am2901A 采用位片式结构,内部有 4 位线路,把多块 Am2901A 芯片级联起来,可实现不同位数的运算器;

(2) Am2901A 中的 ALU 可实现 8 种运算功能,其中包括 3 种算术运算功能和 5 种逻辑运算功能。通过外部送入的 3 位控制信号 $I_5I_4I_3$的编码,可以实现 8 种功能的选择控制。$I_5I_4I_3$与 ALU 具体功能的选择关系如表 3-8 所示。表中 R 和 S 分别为 ALU 的两个输入端。

(3) ALU 的 R 输入端可以接收外部送入运算器的数据 D(如从主存读入数据)、寄存器组的 A 输出及逻辑 0 值;S 输入端可接收寄存器组的 A 输出、B 输出、寄存器 Q 输出及逻辑 0 值。通过外部送入的控制信号 $I_2I_1I_0$的编码,可以控制 R、S 端多路选择器的输入选择。$I_2I_1I_0$编码与 R、S 端的输入选择关系如表 3-9 所示。

(4) 运算器中有一个 16×4 位的通用寄存器组和一个 4 位的 Q 寄存器。通用寄存器组为双端口输出部件,可将各寄存器的值分别送到输出端口 A 或 B,每一个寄存器可以用 A 地址或 B 地址选择。当 A 和 B 地址不同时,在输出端口 A 和 B 将得到两个不同寄存器中的内容。不过寄存器组的写入控制,只取决于 B 地址。写入端口 B 的数据来自可移位的多路选择器,即移位器。移位器可执行直送、左移一位或右移一位的操作,使加减运算和移位操作可在同一

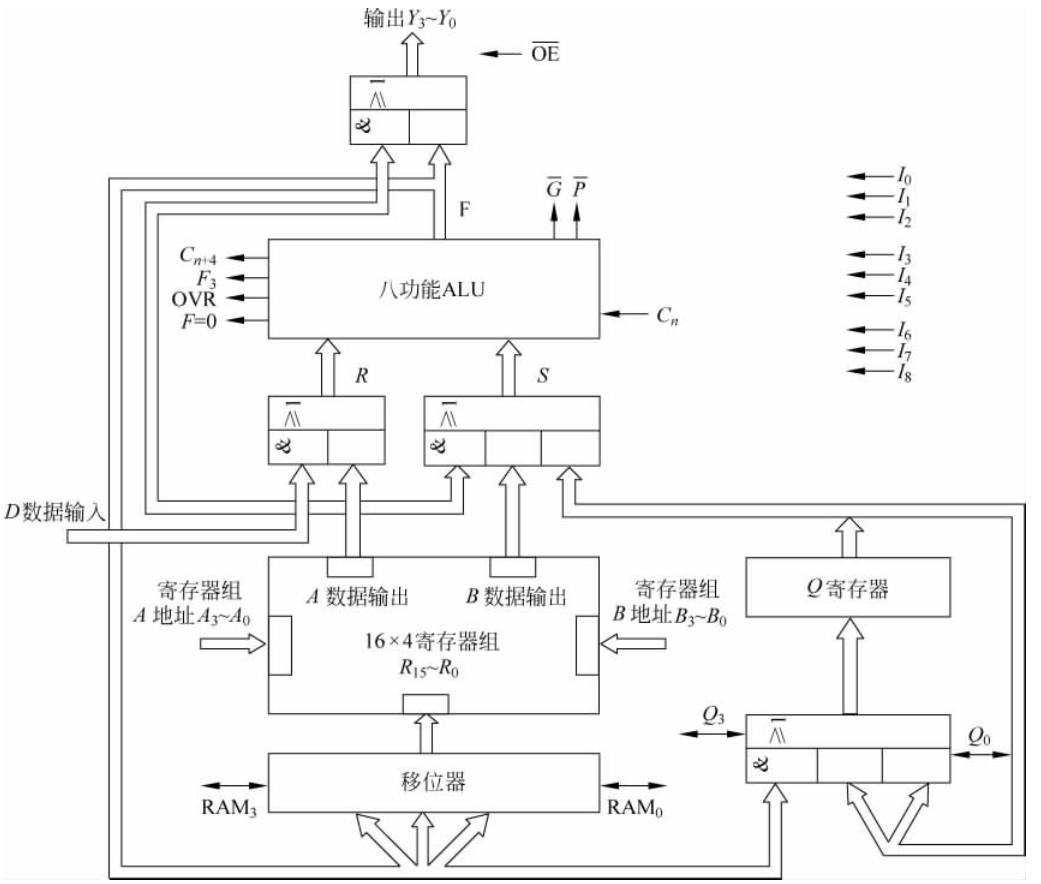


图 3-23　Am2901A 的逻辑示意图

个操作步骤中完成。寄存器 Q 本身具有移位功能，可以实现左移一位或右移一位的功能。寄存器 Q 还可以接收 ALU 的输出 F 的值。Q 的输出可以经 ALU 的 S 输入端送入 ALU。寄存器 Q 可作为乘积和商寄存器。

表 3-8　ALU 功能选择

编　码 $I_5\ I_4\ I_3$	功　能
0 0 0	$R+S$
0 0 1	$S-R$
0 1 0	$R-S$
0 1 1	$R \vee S$
1 0 0	$R \wedge S$
1 0 1	$\overline{R} \wedge S$
1 1 0	$R \oplus S$
1 1 1	$R \odot S$

表 3-9　ALU 操作数选择

编　码 $I_2\ I_1\ I_0$	ALU 操作数输入 R　S
0 0 0	A　Q
0 0 1	A　B
0 1 0	0　Q
0 1 1	0　B
1 0 0	0　A
1 0 1	D　A
1 1 0	D　Q
1 1 1	D　0

（5）根据运算结果，ALU 向外输出 4 个状态信息，它们是：

C_{n+4}：本片 4 位运算器产生的向更高位的进位；

F_3：本片运算结果最高位的取值(可用作符号位)；

OVR：运算结果溢出的判断信号；

$F=0$：结果为 0 信号。

另外，ALU 还需要接收从更低位片送入的进位信号 C_n；向外提供提前进位信号，即小组本地进位 $\overline{G}$ 和小组传递函数 $\overline{P}$。

(6) RAM_3、RAM_0、Q_3、Q_0是移位寄存器接收与送出移位数值的引线。可利用由三态门组成的具有双向传送功能的线路实现。

(7) 运算器的 4 位输出为 $Y_3 \sim Y_0$，它可以是 ALU 的运算结果，也可以是寄存器组 A 输出端口上的内容。输出端采用三态门电路，用$\overline{OE}$信号控制。$\overline{OE}=0$，Y 的值有效，可以输出；$\overline{OE}=1$，Y 输出处于高阻状态。

(8) 用 $I_8I_7I_6$编码决定移位寄存器的输出和结果的输出，可以控制数据传送的方式(移不移位)和数据发送的方向。具体规定如表 3-10 所示。

表 3-10 ALU 的功能选择

编码	功能		
$I_5\ I_4\ I_3$	寄存器组	Q 寄存器	Y 输出
0 0 0		$F\rightarrow Q$	F
0 0 1			F
0 1 0	$F\rightarrow B$		A
0 1 1	$F\rightarrow B$		F
1 0 0	$F/2\rightarrow B$	$Q/2\rightarrow Q$	F
1 0 1	$F/2\rightarrow B$		F
1 1 0	$2F\rightarrow B$	$2Q\rightarrow Q$	F
1 1 1	$2F\rightarrow B$		F

例 3.28 给出实现指令 $R_0+R_1\rightarrow M$，的控制信号。

解：实现指令 $R_0+R_1\rightarrow M$ 的控制信号如下：

控制选择 R_0：A 地址=0000；

控制选择 R_1：B 地址=0001；

控制 ALU 的输入 $R=A$，$S=B$：$I_2I_1I_0=001$；

控制执行运算 $R+S$：$I_5I_4I_3=000$；

控制输出运算结果 $F(Y=F)$：$I_8I_7I_6=001$；

控制允许运算结果 F 输出：$\overline{OE}=0$。

例 3.29 给出实现指令 $2(D-R_9)\rightarrow R_{10}$的控制信号。

解：实现指令 $2(D-R_9)\rightarrow R_{10}$的控制信号如下：

控制选择 R_9：A 地址=1001；

控制选择 R_{10}：B 地址=1010；

控制 ALU 的输入 $R=D$，$S=A$：$I_2I_1I_0=101$；

控制执行运算 $R-S$：$I_5I_4I_3=010$；

控制输出运算结果 $2F$ 到 R_{10}：$I_8I_7I_6=111$；

控制封锁 $Y=F$ 的输出：$\overline{OE}=1$。

3.6.2　浮点运算器

定点数中小数点的位置固定，所以可以直接运算，所需运算设备比较简单。而浮点数由于小数点位置是浮动的，在进行加减运算时，需要将参加运算的数据的小数点位置对齐，即需要进行对阶后，才能正确执行运算。为了尽可能多地保留运算结果中的有效数字，还需要进行规格化。可见浮点运算既需要进行尾数运算又需要进行阶码运算，算法复杂，因此所需设备量大，线路复杂，运算速度也比定点数运算慢。

图 3-24 显示了一个简单的浮点运算器的逻辑图。浮点运算中阶码运算与尾数运算需要分别进行，图 3-24 所示的浮点运算部件中包括了尾数部件和阶码部件两个部分。

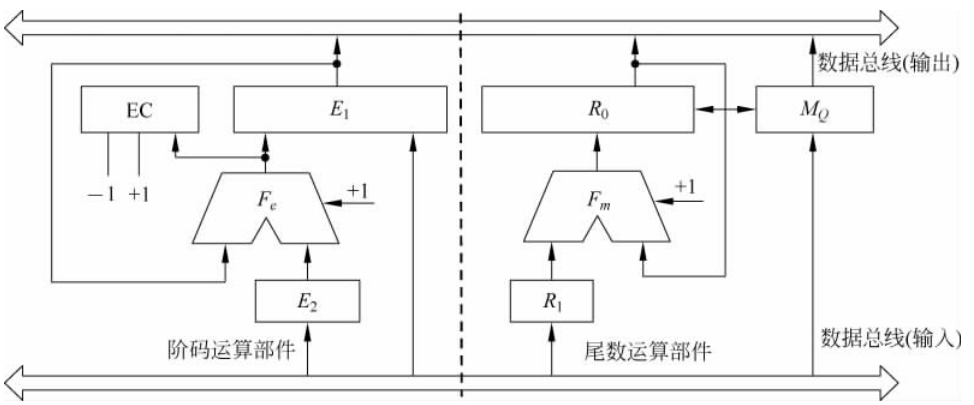


图 3-24　浮点运算器的简单逻辑示意图

1. 尾数运算部件

尾数运算部件用于进行尾数的加减乘除运算，由寄存器 R_0、R_1、M_Q及并行加法器 F_m组成。其中 R_0、R_1用于暂存操作数，R_0还用于存放运算结果；M_Q是乘商寄存器，用于进行乘除运算。R_0、M_Q具有联合左移、右移的功能，移位的实现方法与定点乘、除法器中同类寄存器的移位实现方法相类似。R_1具有右移功能，可以用于实现对阶移位。

表 3-11 给出了不同运算的尾数部件中各寄存器的分配情况。借助于时序部件(图 3-24 中未画出)的控制，采用移位－加减的算法，尾数部件可以实现加、减、乘、除四则运算。

表 3-11　浮点运算器尾数部件的寄存器分配

运算种类	寄存器分配			实现的操作
	R_0	R_1	M_Q	
加	被加数	加数	不用	$(R_0)+(R_1)\rightarrow R_0$
减	被减数	减数	不用	$(R_0)-(R_1)\rightarrow R_0$
乘	乘积(高位)	被乘数	乘数/乘积(低位)	$(R_1)\times(M_Q)\rightarrow R_0$，$M_Q$
除	被除数/余数	除数	商	$(R_0)\div(R_1)\rightarrow M_Q$(商)，$R_0$(余数)

2. 阶码运算部件

阶码运算部件用于进行阶码的加减运算。由寄存器 E_1、E_2、阶差计数器 EC 以及并行加法器 F_e组成。其中 E_1、E_2用于存放与 R_0、R_1中尾数相对应的阶码。

浮点运算部件的工作原理如下：

1) 做加减运算时

(1) 由阶码运算部件求出阶差 $\Delta E=E_1-E_2$,并存入阶差计数器 EC 中,EC 可根据符号判断哪个阶码小,控制将对应的尾数(R_0或 R_1)进行右移。

若 ΔE 为+,则判断 E_2小,控制 R_1右移,且每右移一位,EC-1;

若 ΔE 为-,则判断 E_1小,控制 R_0右移,且每右移一位,EC+1。

一直控制移位到 EC=0,完成对阶工作。

(2) 尾数部件做加减运算,结果存入 R_0。

(3) 判别运算结果,进行规格化。

在规格化处理过程中,每将 R_0左移(右移)一位,应将 E_1与 E_2中的较大者减 1(加 1),规格化结束后,将其作为结果的阶码。

2) 做乘除运算时

尾数运算部件和阶码运算部件独立工作,阶码仅做加减运算。运算结束后,对结果进行规格化处理。

3.7 十进制数的加减运算方法

在第 2 章中我们了解到,为适应商用的需要,可以对十进制数的二进制编码进行一些特殊的规定,使计算机内部具有直接进行十进制运算的能力。因此,有些计算机中设置了十进制指令,可以直接对十进制数进行运算,从而减少了十进制数和二进制数之间的转换,方便了商用指令的处理。计算机中实现十进制数加减运算通常采用的方法有:

(1) 利用原有的二进制加法器对采用 BCD 编码表示的十进制数进行运算,然后再用十进制修正指令对运算结果进行修正,以获得正确的十进制运算结果。

(2) 直接利用十进制加法器实现十进制运算。

第(1)种方法是目前微型计算机中常用的方法。这种方法几乎不需要对原有的二进制加法器做任何修改,就可以实现十进制加减运算,但由于需要利用指令进行“二进制加减运算——结果修正”两步操作,所以实现十进制运算的速度较慢。第(2)种方法需要专用的十进制运算器,增加了硬件系统的复杂性,但实现十进制运算的速度较快。

3.7.1 一位十进制加法器的设计

运算器的核心部件是十进制加法器,它实际是在二进制加法器的基础上加上一定的修正逻辑构成的。十进制加法器的基本思路是将两个十进制数的 BCD 码按二进制加法运算,再根据运算结果与十进制和数的正确 BCD 码的差别求得修正逻辑,将二进制结果修正为十进制和数。由于 BCD 码中 8421 码用得较多,因此下面我们主要讨论 8421 码十进制加法器。

利用 8421 码进行十进制运算,实际就是将每个十进制数对应的 4 位二进制编码先当作二进制数进行计算,然后再按十进制运算的进位规律对运算结果进行必要的修正。我们知道两个一位十进制数相加,其和小于等于 18,若考虑低位来的进位,其和的最大值不会超过 19。表 3-12 列出了两个 8421 码按二进制加法规则相加的结果和正确的 8421 码之间的关系。

表 3-12　8421 码十进制加法器运算结果的修正关系

十进制数	用 8421 码表示的十进制和数 $C'_4\ F_4F_3F_2F_1$	两个 8421 码按二进制规则相加得到的和数 $C_4\ S_4S_3S_2S_1$	修正逻辑
0	0 0000	0 0000	不修正
1	0 0001	0 0001	
2	0 0010	0 0010	
3	0 0011	0 0011	
4	0 0100	0 0100	
5	0 0101	0 0101	
6	0 0110	0 0110	
7	0 0111	0 0111	
8	0 1000	0 1000	
9	0 1001	0 1001	
10	1 0000	0 1010	加“0110”修正
11	1 0001	0 1011	
12	1 0010	0 1100	
13	1 0011	0 1101	
14	1 0100	0 1110	
15	1 0101	0 1111	
16	1 0110	1 0000	
17	1 0111	1 0001	
18	1 1000	1 0010	
19	1 1001	1 0011	

根据表 3-12 中两个 8421 码按二进制加法规则相加后的结果与正确的 8421 码和数之间的关系，可以看到，两个 8421 码相加后，若相加的和数＜10，则不需修正，按二进制规则相加的结果就是正确的 8421 码的和数；若相加的和数≥10，则需在二进制相加的结果上加“0110”进行修正。由此可以得到两个 8421 码相加后，结果需要修正的条件为：

$$C_4 + S_4S_3 + S_4S_2 \tag{3-27}$$

其中，C_4 为两个 8421 码相加后向高位的进位。分析表 3-12 还可发现，在正确的 8421 码和数中，本位十进制数向高位十进制数产生的进位 C'_4 的条件与修正条件一致，即：

$$C'_4 = C_4 + S_4S_3 + S_4S_2$$

例 3.30　利用 8421 码加法规则计算 3＋4。

解：十进制数 3 和 4 的 8421 码为 0011 和 0100，

3＋4 的 8421 码和数＝0011＋0100＝0111。

$$\begin{array}{r} 0011 \\ +\ 0100 \\ \hline 0111 \end{array}$$

因为和数＜10，所以结果不需要修正，3＋4＝7。

例 3.31　利用 8421 码加法规则计算 3＋9。

解：十进制数 3 和 9 的 8421 码为 0011 和 1001，

3＋9 的 8421 码和数＝0011＋1001＝1100。

因为结果≥10,所以结果需要进行加 0110 修正。

$$\begin{array}{r} 0011 \\ +\ 1001 \\ \hline 1100 \\ +\ 0110 \\ \hline \underline{1}\ 0010 \\ \text{进位} \end{array}$$

修正后 3+9 的正确的 8421 码和数为 0001 0010,即 3+9=12。

根据上述修正条件和修正方法设计的一位 8421 码十进制加法器的逻辑电路如图 3-25 所示。从图中可见两个 8421 码 $A_4A_3A_2A_1$ 与 $B_4B_3B_2B_1$ 相加后的和经过修正后,才能得到正确的 8421 码 $F_4F_3F_2F_1$。按 8421 码的设计思路,读者可以设计其他 BCD 码十进制加法器。

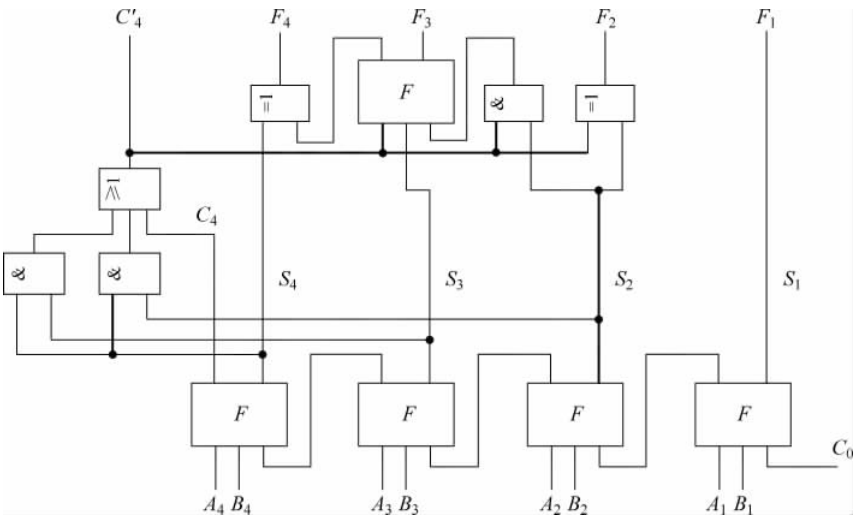


图 3-25 一位 8421 码十进制加法器的逻辑电路

3.7.2 多位十进制整数的加减运算

1. 多位十进制加法

多位十进制加法可以按照一位十进制加法规则进行运算和结果修正。需注意的是,在进行结果修正时,每一位十进制数运算结果的修正,必须是在低位十进制数运算结果修正的基础上进行的,即高位结果的修正必须考虑低位结果修正后的进位。

例 3.32 利用 8421 码加法规则计算 137+376。

解:十进制数 137 的 8421 码为 0001 0011 0111,376 的 8421 码为 0011 0111 0110,137+376 的 8421 码和数=0001 0011 0111+0011 0111 0110。

运算和结果修正过程为:

$$\begin{array}{r} 0001\quad 0011\quad 0111 \\ +\ 0011\quad 0111\quad 0110 \\ \hline 0100\quad 1010\quad 1101 \\ +\quad\ \ {}_{1}\quad 0110\ {}_{1}0110 \\ \hline 0101\quad 0001\quad 0011 \\ \uparrow\qquad\ \ \uparrow \\ \text{进位}\quad \text{进位} \end{array}$$

所以 137＋376 的 8421 码和数为 0101 0001 0011，即 137＋376＝513。

多位十进制加法运算器可以用多个一位十进制加法运算器级联起来实现。

2. 多位十进制减法

十进制减法与二进制减法类似，用加上减数的补码代替。设 x、y 均为 n 位十进制整数，则利用十进制减法进行运算时，它们的差 F 为：

$$F = x - y = x + [-y]_{10补} \tag{3-28}$$

十进制补的定义为：

$$[-y]_{10补} = 10^n - y \tag{3-29}$$

因此：

$$x - y = x + (10^n - y) = 10^n + (x - y) \tag{3-30}$$

根据式(3-30)，可以得出：

若 $x-y \geqslant 0$，最高位产生的进位丢掉，结果正确，为正数；

若 $x-y<0$，最高位不产生进位，结果为负，需再变一次补，才能得到正确结果。

例 3.33　利用 8421 码减法规则计算 319－146。

解：$[-146]_{10补}=10^3-146=854$，$319+[-146]_{10补}$ 的和数 $319+[-146]_{10补}$＝0011 0001 1001＋1000 0101 0100，运算和结果修正过程为：

	0011	0001	1001
＋	1000	0101	0100
	1011	0110	1101
＋	0110	0000	0110
	1 0001	0111	0011

进位丢掉

因为最高位产生的进位丢掉，结果正确，为正数。所以 319－146＝173。

例 3.34　利用 8421 码减法规则计算 257－582。

解：$[-582]_{10补}=10^3-582=418$，运算和结果修正过程为：

	0010	0101	0111	
＋	0100	0001	1000	
	0110	0110	1111	
＋	0000	0000	1 0110	
	0110	0111	0101	(675)

结果最高位没有进位，结果为负，需再变一次补，才能得到正确结果。

因为 $[-675]_{10补}=10^3-675=325$，所以 257－582＝－325。

3.8　逻辑运算和移位操作

3.8.1　逻辑运算

运算器除了要完成数值数据的算术运算外，还要完成逻辑运算和移位操作。计算机中的逻辑运算包括与、或、非、异或等运算。由于逻辑数是非数值数据，其每一位的 0 和 1 仅用于表示逻辑上的真与假，不存在符号位、数值位、阶码、尾数之分，因此逻辑运算的特点是：按位运算，运算简单，运算结果的各位之间互不影响，不存在进位、借位、溢出等问题。

例 3.35 设 $x=11010$，$y=00101$，求 $x \cdot y$，$x+y$、$x \oplus y$、$\bar{x}$。

解：$x \cdot y$ 是逻辑与运算，$x \cdot y=11010 \cdot 00101=00000$；

$x+y$ 是逻辑或运算，$x+y=11010+00101=11111$；

$x \oplus y$ 是逻辑异或运算，$x \oplus y=11010 \oplus 00101=11111$；

$\bar{x}$ 是逻辑非运算，$\bar{x}=\overline{11010}=00101$。

逻辑运算还可用于对数据字中某些位(一位或多位)进行操作，常见的应用有：

1. 按位测

利用“逻辑与”操作可以屏蔽掉数据字中的某些位。例如，让被检测的数作为目的操作数，屏蔽字作为源操作数，要检测被检数的某些位时，可使屏蔽字的相应位为 1，其余位为 0，将两者进行“逻辑与”操作，根据结果是否为全 0，检测出所要求的位是 0 还是 1。

2. 按位清

利用“逻辑与”可以将数据字的某些位清 0。例如，把待清除的数作为目的操作数，操作模式作为源操作数，要清除数据字中的哪些位，就使源操作数的相应位为 0，其余位为 1，然后将两者进行“逻辑与”操作，即可将目的操作数的相应位清 0。

3. 按位置

利用“逻辑或”可以使数据字的某些位置 1。例如把需设置的数作为目的操作数，操作模式作为源操作数，要设置数据字中的哪些位，就使操作模式的相应位为 1，其余位为 0，然后将两者进行“逻辑或”操作，就可使目的操作数的相应位置 1。

4. 判符合或修改

根据异或运算的特点可知，若两数相同，则两者的异或结果必为 0。而任何数与 1 相异或，所得结果必为该数的反。因此根据两数异或结果是否为 0，即可判断两数是否相符。如果需要修改数据的某些位(即将相应位取反)，可使操作模式的相应位设为 1，其余为 0，将操作模式与数据字异或之后，就可实现对数据字相应位的修改。

3.8.2 移位操作

如第 2 章所述，用二进制表示的机器数在相对小数点做 n 位左移或右移时，相当于使该数乘以或除以 $2n$。在定点乘法和除法运算中，也要用到移位操作。可见移位操作是运算器中的重要操作。

由于计算机中机器的字长是固定的，当机器数进行左移或右移时，必然会使机器数的低位或高位产生空位。对这些空位是填 0 还是填 1，需要根据机器数采用的是无符号数还是带符号数来确定。移位操作包括逻辑移位、算术移位和循环移位，如图 3-26 所示。

1. 逻辑移位

进行逻辑移位时，认为需要移位的机器数代码为无符号数或纯逻辑代码，所以移位时不考虑符号问题。移位时所有代码均参加移位。

(1) 逻辑左移：各位按位左移，最高位向左移出，最低位空位填 0。通常，向左移出的最高位可保存到运算器的进位状态寄存器 C 中。

(2) 逻辑右移：各位按位右移，最低位向右移出，最高位空位填 0。通常，向右移出的最低位可保存到运算器的进位状态寄存器 C 中。

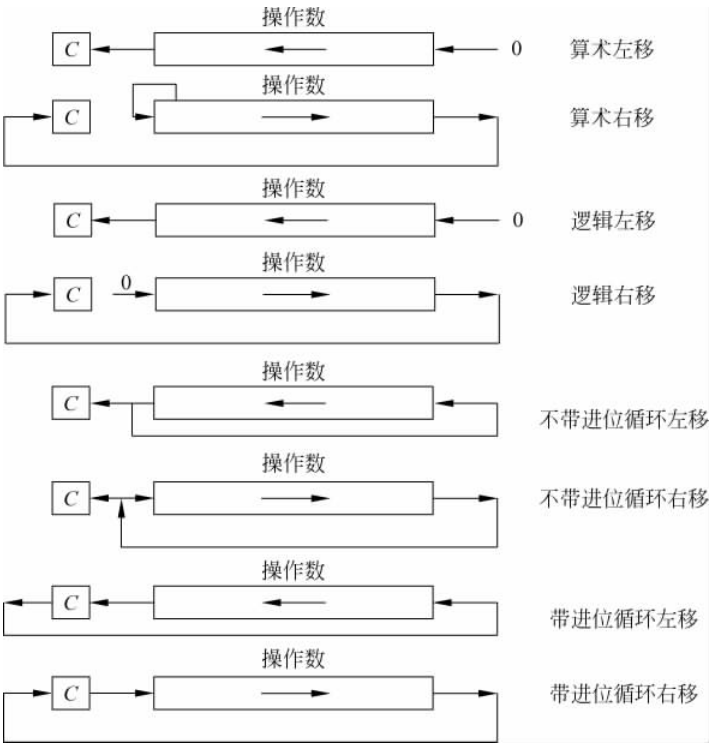
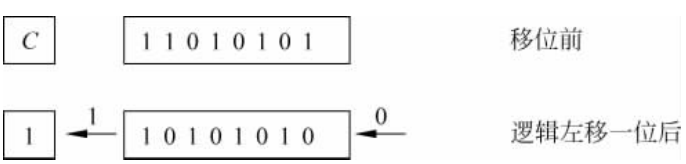


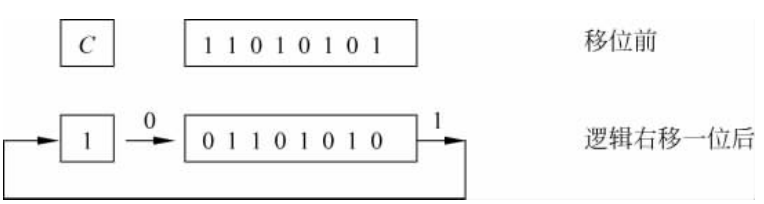
图 3-26　各种移位的操作过程

例 3.36　设 x 为无符号数，$x=11010101$，写出 x 逻辑左移一位和逻辑右移一位的结果。

解：(1) x 逻辑左移一位后，最低位空位填 0，得 $x=10101010$。最高位移入进位状态寄存器，$C=1$。



(2) x 逻辑右移一位后，最高位空位填 0，得 $x=01101010$。最低位移入进位状态寄存器，$C=1$。



2. 算术移位

进行算术移位时，需要移位的代码为带符号数，具有数值含义且带有符号位，因此在算术移位中，必须保持移位前后的符号位不变。

根据第 2 章中带符号数的原码、补码、反码的移位规则可知，对于带符号数 x，若 $x>0$，则

$[x]_{原}=[x]_{补}=[x]_{反}$=真值,故对 x 进行移位后产生的空位均填 0。若 $x<0$,由于$[x]_{原}$、$[x]_{原}$、$[x]_{反}$的表示形式不同,因而移位后产生空位的填补规则不同。

对于$[x]_{原}$,因为负数原码的数值部分与真值数值部分相同,所以在移位时只要保持符号位不变,移位后产生的空位均填 0。

对于$[x]_{反}$,因为负数反码除符号位外的各位与负数原码正好相反,所以移位后空位所填的代码应与原码相反,即移位时保持符号位不变,移位后产生的空位均填 1。

对于$[x]_{补}$,因为从负数补码的最低位向高位寻找,遇到第一个 1 的左边的各位均与所对应的反码相同,而在包括该 1 在内的右边的各位均与对应的原码相同,所以在对负数补码进行左移时,低位产生的空位中应填入与原码相同的值,即在移位后产生的空位中填 0;在对负数补码进行右移时,高位产生的空位中应填入与反码相同的值,即在移位后产生的空位中填 1。

根据上述分析,可以归纳出算术移位的规则:

(1) 算术左移:各位按位左移,最高位向左移出,最低位产生的空位填 0。向左移出的最高位可保存到进位状态寄存器 C 中。

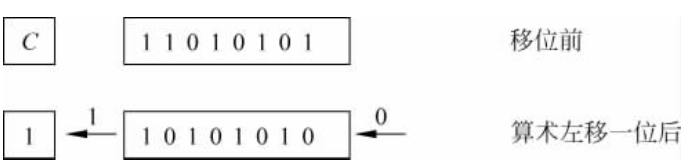
注意:算术左移后数据的符号不应改变,如果左移前后的符号位发生了变化,说明数据的符号被破坏,移位溢出。

(2) 算术右移:各位按位右移,最低位向右移出,最高位产生的空位填入与原最高位相同的值,即符号位保持不变。向右移出的最低位可保存到进位状态寄存器 C 中。

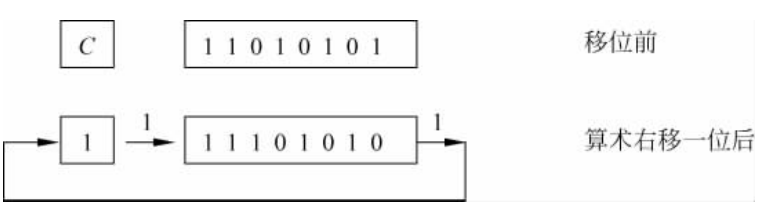
例 3.37 设$[x]_{补}$=11010101,写出$[x]_{补}$算术左移一位和算术右移一位的结果。

解:因为 $[x]_{补}$=11010101,所以 $x<0$。

① $[x]_{补}$算术左移一位后,最低位空位填 0,得 x=10101010,最高位移入 C 中,C=1。左移前后的符号位未发生变化,移位正确。



② $[x]_{补}$算术右移一位后,最高位产生的空位填入与原最高位相同的值,即填 1,得 x=11101010。最低位移入 C 中,C=1。



3. 循环移位

在计算机中通常还设有循环移位操作指令。所谓循环移位,就是指移位时数据的首尾相连进行移位,即最高(最低)位的移出位又移入数据的最低(最高)位。根据循环移位时的进位是否一起参加循环,可将循环移位分为不带进位循环和带进位循环两类。其中,不带进位循环是指进位状态寄存器 C 中的内容不与数据部分一起循环移位,也称小循环。带进位循环是指进位状态寄存器 C 中的内容与数据部分一起循环移位,也称大循环。具体可

分为：

(1) 不带进位循环左移：各位按位左移，最高位移入最低位，同时保存到 C 中；

(2) 不带进位循环右移：各位按位右移，最低位移入最高位，同时保存到 C 中；

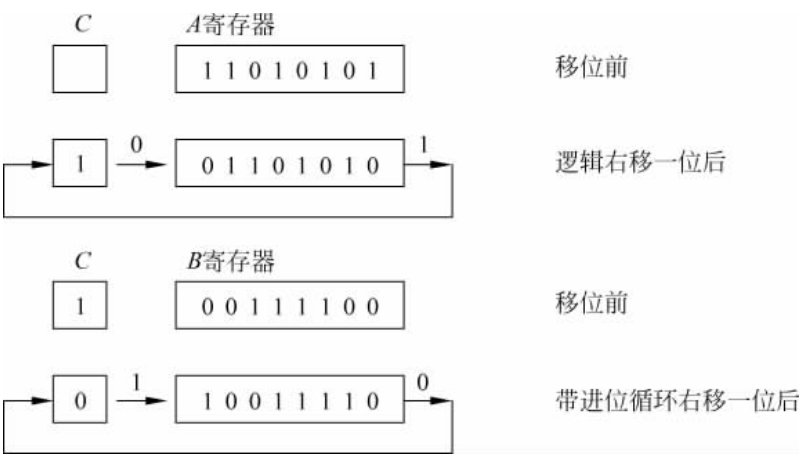
(3) 带进位循环左移：各位按位左移，最高位移入 C 中，C 中的内容移入最低位；

(4) 带进位循环右移：各位按位右移，最低位移入 C 中，C 中的内容移入最高位。

循环移位一般用于实现循环式控制、高低字节的互换，还可以用于实现多倍字长数据的算术移位或逻辑移位。

例 3.38　设有两个 8 位寄存器 A 和 B，A 中的内容为 11010101，B 中的内容为 00111100。试利用移位指令将两个寄存器的内容联合逻辑右移一位，其中寄存器 A 为高 8 位，B 为低 8 位。

解：先将寄存器 A 中的内容逻辑右移一位，其最低位移入进位状态寄存器 C 中，再将寄存器 B 中的内容带进位循环右移一位，即可完成两个寄存器的联合逻辑右移。



寄存器 A、B 联合逻辑右移后，A 中的内容为 01101010，B 中的内容为 10011110，进位状态寄存器 C 中的内容为 0。

习　　题

3.1　已知 $[x]_{补}$、$[y]_{补}$，计算 $[x+y]_{补}$ 和 $[x-y]_{补}$，并判断溢出情况。

(1) $[x]_{补}=0.11011$　$[y]_{补}=0.00011$

(2) $[x]_{补}=0.10111$　$[y]_{补}=1.00101$

(3) $[x]_{补}=1.01010$　$[y]_{补}=1.10001$

3.2　已知 $[x]_{补}$、$[y]_{补}$，计算 $[x+y]_{变形补}$ 和 $[x-y]_{变形补}$，并判断溢出情况。

(1) $[x]_{补}=100111$　$[y]_{补}=111100$

(2) $[x]_{补}=011011$　$[y]_{补}=110100$

(3) $[x]_{补}=101111$　$[y]_{补}=011000$

3.3　设某机字长为 8 位，给定十进制数 $x=+49$，$y=-74$。试按补码运算规则计算下列各题，并判断溢出情况。

(1) $[x]_{补}+[y]_{补}$

(2) $[x]_{补}-[y]_{补}$

(3) $[-x]_{补}+\left[\frac{1}{2}y\right]_{补}$

(4) $\left[2x-\frac{1}{2}y\right]_{补}$

(5) $\left[\frac{1}{2}x+\frac{1}{2}y\right]_{补}$

(6) $[-x]_{补}+[2y]_{补}$

3.4 分别用原码一位乘法和补码一位乘法计算$[x\times y]_{原}$和$[x\times y]_{补}$。

(1) $x=0.11001$ $y=0.10001$

(2) $x=0.01101$ $y=-0.10100$

(3) $x=-0.10111$ $y=0.11011$

(4) $x=-0.01011$ $y=-0.11010$

3.5 分别用原码不恢复余数法和补码不恢复余数法计算$[x/y]_{原}$和$[x/y]_{补}$。

(1) $x=0.01011$ $y=0.10110$

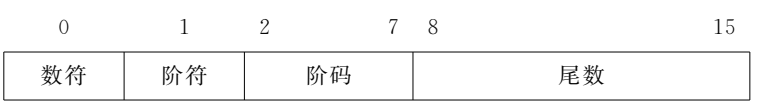
(2) $x=0.10011$ $y=-0.11101$

(3) $x=-0.10111$ $y=-0.11011$

(4) $x=+10110$ $y=-00110$

3.6 在进行浮点加减运算时,为什么要进行对阶?说明对阶的方法和理由。

3.7 已知某模型机的浮点数据表示格式如下:



其中,浮点数尾数和阶码的基值均为2,均采用补码表示。

(1) 求该机所能表示的规格化最小正数和非规格化最小负数的机器数表示及其所对应的十进制真值。

(2) 已知两个浮点数的机器数表示为EF80H和FFFFH,求它们所对应的十进制真值。

(3) 已知浮点数的机器数表示为:

$$[x]_{补}=1\ 1111001\ 00100101,\quad [y]_{补}=1\ 1110111\ 00110100$$

试按浮点加减运算算法计算$[x\pm y]_{补}$。

3.8 已知某机浮点数表示格式如下:

其中,浮点数尾数和阶码的基值均为2,阶码用移码表示,尾数用补码表示。设:

$$x=0.110101\times 2^{-001}\qquad y=-0.100101\times 2^{+001}$$

试用浮点运算规则计算$x+y$、$x-y$、$x\times y$、x/y(要求写出详细运算步骤,并进行规格化)。

3.9 图3-27给出了实现补码乘法的部分硬件框图。

(1) 请将图3-27中逻辑门AND_1和AND_2的输入信号填写正确。

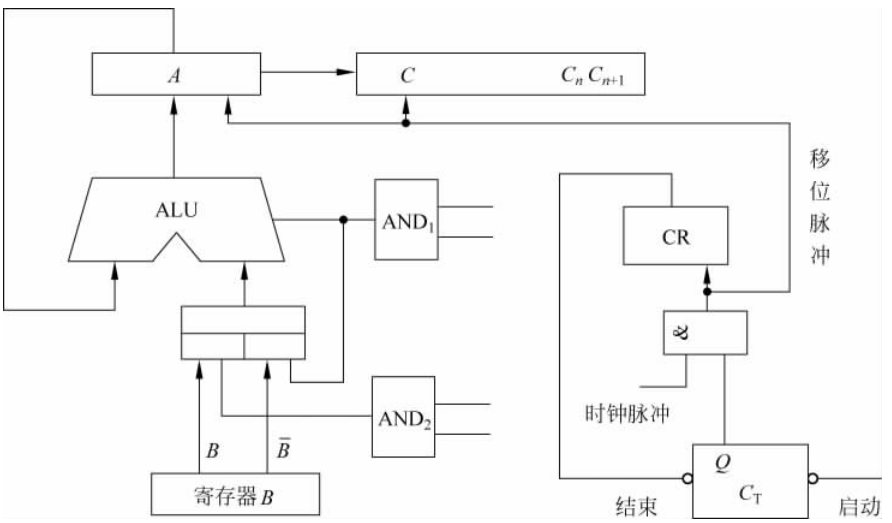


图 3-27　补码乘法的部分硬件框图

(2) 按补码乘法规则将下列乘法运算算式完成，写出 $x \times y$ 的真值。

	00.000000	1001100
→	00.000000	0100110
	00.110001	
	00.110001	
→	00.011000	1010011
→	00.001100	0101001

(3) 根据(2)的乘法算式，将乘法运算初始和结束时，3 个寄存器中的数据填入下列表格中。

寄存器	A	B	C
运算初始			
运算结束			

3.10　说明定点补码和浮点补码加减运算的溢出判断方法。

3.11　说明定点原码除法和定点补码除法运算的溢出判断方法。

3.12　比较舍入方法中截断法、恒置 1 法和 0 舍 1 入法的优缺点。

3.13　利用十进制加减运算计算下列各题：

(1) 125＋436

(2) 125－436

(3) 436－125

3.14　参照第 2 章表 2-13 的余 3 码的编码规则，设计利用余 3 码进行十进制加法的修正逻辑。

3.15　设有一个 16 位定点补码运算器，数据最低位的序号为 1。运算器可实现下述

功能:

(1) $A \pm B \rightarrow A$

(2) $B \times C \rightarrow A$、C(乘积高位在 A 中)

(3) $A \div B \rightarrow C$(商在 C 中)

请设计并画出运算器第 3 位及寄存器 A、C 第 3 位输入逻辑。加法器本身逻辑可以不画,原始操作数输入问题可以不考虑。

3.16 根据布斯除法的算法规则,参照补码一位乘法的逻辑电路,设计一个可以实现布斯除法的逻辑电路。

3.17 比较算术右移和逻辑右移,试说明其主要区别。

3.18 设一个 8 位寄存器中的内容为十六进制数 C5H,连续经过一次算术右移、一次逻辑左移、一次大循环右移、一次小循环左移。写出每次移位后寄存器的内容和进位标志 C 的状态。

3.19 已知寄存器 A 的内容为 01011010,寄存器 B 的内容为 11011011,分别写出经过下列移位操作后,寄存器 A、B 中的内容。

(1) 算术左移两位;

(2) 逻辑左移两位;

(3) 算术右移两位;

(4) 逻辑右移两位。

3.20 选择题。

(1) 运算器的核心部分是_____。

A. 数据总线　　B. 累加寄存器

C. 算术逻辑运算单元　　D. 多路开关

(2) 在浮点运算中下面的论述正确的是_____。

A. 对阶时应采用向左规格化

B. 对阶时可以使小阶向大阶对齐,也可以使大阶向小阶对齐

C. 尾数相加后可能会出现溢出,但可采用向右规格化的方法得出正确结论

D. 尾数相加后不可能得出规格化的数

(3) 当采用双符号位进行数据运算时,若运算结果的双符号位为 01,则表明运算_____。

A. 无溢出　　B. 正溢出　　C. 负溢出　　D. 不能判别是否溢出

(4) 补码加法运算的规则是_____。

A. 操作数用补码表示,符号位单独处理

B. 操作数用补码表示,连同符号位一起相加

C. 操作数用补码表示,将加数变补,然后相加

D. 操作数用补码表示,将被加数变补,然后相加

(5) 原码乘除法运算要求_____。

A. 操作数必须都是正数　　B. 操作数必须具有相同的符号位

C. 对操作数符号没有限制　　D. 以上都不对

(6) 进行补码一位乘法时,被乘数和乘数均用补码表示,运算时_____。

A. 首先在乘数最末位 y_n 后增设附加位 y_{n+1},且初始 $y_{n+1}=0$,再依照 $y_n y_{n+1}$ 的值确

定下面的运算

B. 首先在乘数最末位 y_n 后增设附加位 y_{n+1}，且初始 $y_{n+1}=1$，再依照 y_ny_{n+1} 的值确定下面的运算

C. 首先观察乘数符号位，然后决定乘数最末位 y_n 后附加位 y_{n+1} 的值，再依照 y_ny_{n+1} 的值确定下面的运算

D. 不应在乘数最末位 y_n 后增设附加位 y_{n+1}，而应直接观察乘数的末两位 $y_{n-1}y_n$ 确定下面的运算

(7) 下面对浮点运算器的描述中正确的是_____。

A. 浮点运算器由阶码部件和尾数部件实现

B. 阶码部件可实现加、减、乘、除四种运算

C. 阶码部件只能进行阶码的移位操作

D. 尾数部件只能进行乘法和加法运算

(8) 若浮点数的阶码和尾数都用补码表示，则判断运算结果是否为规格化数的方法是_____。

A. 阶符与数符相同为规格化数

B. 阶符与数符相异为规格化数

C. 数符与尾数小数点后第一位数字相异为规格化数

D. 数符与尾数小数点后第一位数字相同为规格化数

(9) 已知 $[x]_{补}=1.01010$，$[y]_{补}=1.10001$，下列答案正确的是_____。

A. $[x]_{补}+[y]_{补}=1.11011$　　B. $[x]_{补}+[y]_{补}=0.11011$

C. $[x]_{补}-[y]_{补}=0.11011$　　D. $[x]_{补}-[y]_{补}=1.11001$

(10) 下列叙述中概念正确的是_____。

A. 定点补码运算时，其符号位不参加运算

B. 浮点运算中，尾数部分只进行乘法和除法运算

C. 浮点数的正负由阶码的正负符号决定

D. 在定点小数一位除法中，为了避免溢出，被除数的绝对值一定要小于除数的绝对值

3.21　填空题。

(1) 在补码加减运算中，符号位与数据 ① 参加运算，符号位产生的进位 ② 。

(2) 在采用变形补码进行加减运算时，若运算结果中两个符号位 ① ，表示发生了溢出。若结果的两个符号位为 ② ，表示发生正溢出；为 ③ ，表示发生负溢出。

(3) 在原码一位乘法的运算过程中，符号位与数值位 ① 参加运算，运算结果的符号位等于 ② 。

(4) 浮点乘除法运算的运算步骤包括：① 、② 、③ 、④ 和 ⑤ 。

(5) 在浮点运算过程中，如果运算结果的尾数部分不是 ① 形式，则需要进行规格化处理。设尾数采用补码表示形式，当运算结果 ② 时，需要进行右规操作；当运算结果 ③ 时，需要进行左规操作。

(6) 将两个 8421BCD 码相加，为了得到正确的十进制运算结果，需要对结果进行修正，其修正方法是 ① 。

(7) 浮点运算器由 ① 和 ② 两部分组成，它们本身都是定点运算器，其中①要求能

够进行__③__运算;②要求能够进行__④__运算。

(8) 设有一个16位的数据存放在由两个8位寄存器AH和AL组成的寄存器AX中,其中数据的高8位存放在AH寄存器中,低8位存放在AL寄存器中。现需要将AX中的数据进行一次算术左移,其操作方法是:先对__①__进行一次__②__操作,再对__③__进行一次__④__操作。

3.22 是非题。

(1) 运算器的主要功能是进行加法运算。

(2) 加法器是构成运算器的主要部件,为了提高运算速度,运算器中通常都采用并行加法器。

(3) 只有定点运算才会发生溢出,浮点运算不会发生溢出。

(4) 在定点整数除法中,为了避免运算结果的溢出,要求|被除数|<|除数|。

(5) 浮点运算器中的阶码部件可实现加、减、乘、除运算。

(6) 在浮点加减运算中,对阶时既可以使小阶向大阶对齐,也可以使大阶向小阶对齐。

(7) 根据数据的传递过程和运算控制过程来看,阵列乘法器实现的是全并行运算。

(8) 逻辑右移执行的操作是进位标志位移入符号位,其余数据位依次右移1位,最低位移入进位标志位。