第 3 章　运算方法和运算器

运算器是计算机进行算术运算和逻辑运算的主要部件。运算器的逻辑结构取决于机器指令系统的运算功能及数据的表示形式和运算方法。本章主要讨论数值数据在计算机中进行算术运算和逻辑运算的实现方法以及运算部件的基本结构和工作原理，使读者对 CPU 中的运算器的功能及实现方法有一个全面的认识。重点是定点算术运算、浮点算术运算和算术逻辑运算单元(ALU)。

3.1　定点加减运算

数值型数据在计算机中是以一定的编码方式表示的。即使是同一种算术运算，若采用的编码不同，其运算法则、实现方法也不同。

采用原码表示的数据，在进行加、减运算时，要先看是加还是减，还要看两数的符号。同号加、异号减要进行加运算；同号减、异号加要用减法运算，且还要比较两数的大小，运算结果的符号取决于绝对值大的数。在电路实现时，既有加法器还要有减法器。

采用补码表示数据时，可以将加减运算中的异号加和同号减，转换为加法运算。而且在进行加的过程中，符号位也像数值位一样参与运算。这大大简化了加减运算电路的实现。因此，计算机中普遍采用补码进行定点加减运算。本节也仅介绍补码的加减运算及其实现方法。

3.1.1　补码加法运算

可以证明，两个有符号数相加，和的补码等于两个数的补码相加的和。即

$$[x+y]_{补}=[x]_{补}+[y]_{补}$$

说明：补码相加时符号位和数值位一起参与运算。

根据补码的定义，下面分 4 种情况证明上式的正确性。

(1) $x>0, y>0$。因为 x、y 均为正数，则 $x+y$ 也为正数。

根据补码的定义，得

$$[x]_{补}=x,\quad [y]_{补}=y,\quad [x+y]_{补}=x+y$$

因此

$$[x+y]_{补}=[x]_{补}+[y]_{补}$$

(2) $x<0, y<0$。因为 x、y 均为负数，则 $x+y$ 也为负数。

根据补码的定义，得

$$[x]_{补}=M+x, [y]_{补}=M+y, [x+y]_{补}=M+x+y \pmod M$$

其中，定点 n 位整数机模的 $M=2^n$，对于定点小数机来说，$M=2$，则

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

即

$$[x+y]_{补}=[x]_{补}+[y]_{补}$$

(3) $x>0, y<0$。根据补码的定义，得

$$[x]_{补}=x, [y]_{补}=M+y$$

因此

$$[x]_{补}+[y]_{补}=M+x+y$$

其中，$x+y$ 有正、负两种可能：

$x+y>0$ 时，

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

$x+y<0$ 时，

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

(4) $x<0, y>0$。这种情况与(3)类似。

根据$[x+y]_{补}=[x]_{补}+[y]_{补}$，可以得到补码加法的运算规则如下：

将参加运算的两个数用补码表示，然后对两个补码进行加法运算，符号位和数值位一起参与运算，得到的就是两个数的和的补码。

【例 3-1】 设 $x=-1101, y=+0110$，利用补码加法运算求 $x+y$。

分析：补码加法运算，即运算电路的输入端是两数的补码。因此，在做题时要先根据真值写出两数的补码，然后再由加法运算电路进行运算。最后，要将运算结果转换为真值给出。

解：$[x]_{补}=1,0011$，　$[y]_{补}=0,0110$

$$\begin{array}{rll} & 1,0011 & [x]_{补} \\ + & 0,0110 & [y]_{补} \\ \hline & 1,1001 & [x+y]_{补} \end{array}$$

即$[x+y]_{补}=1,1001$，所以 $x+y=-0111\mathrm{B}=-7$。

验证：$x=-1101\mathrm{B}=-13, y=+0110\mathrm{B}=+6, x+y=-7$，结果正确。

说明：将符号位与数值部分之间增加逗号，达到区分二者的目的，同时突出符号位在运算时也像数值位一样参与了运算。

【例 3-2】 设 $x=+1011, y=-0101$，利用补码加法运算求 $x+y$。

解：$[x]_{补}=0,1011$，　$[y]_{补}=1,1011$

$$\begin{array}{rll} & 0,1011 & [x]_{补} \\ + & 1,1011 & [y]_{补} \\ \hline & \boxed{1}\,0,0110 & [x+y]_{补} \end{array}$$

即$[x+y]_{补}=0,0110$，所以 $x+y=+0110\mathrm{B}=+6$。

验证：$x=+1011\mathrm{B}=+11, y=-0101\mathrm{B}=-5, x+y=11-5=+6$，结果正确。

补码相加最高位的进位丢掉，不影响结果的正确性。

3.1.2 补码减法运算

根据补码加法公式可得

$$[x-y]_{补}=[x]_{补}+[-y]_{补}$$

那么，如果能根据$[y]_{补}$很方便地得到$[-y]_{补}$，就可以利用加法器来实现减法运算了。

可以证明，$[-y]_{补}$等于$[y]_{补}$连同符号位在内的变反后末位再加1，这个过程也称为对$[y]_{补}$变补。即

$$[-y]_{补}=[[y]_{补}]_{变补}$$

设n位机中$[y]_{补}=y_S,y_{n-2}\cdots y_1y_0$，下面分两种情况证明上式的正确性。

(1) $y>0$。

因为y是正数，则

$$[y]_{补}=[y]_{原}=0,y_{n-2}y_{n-3}\cdots y_1\ y_0 \tag{3-1}$$

因为$[-y]_{原}=1,y_{n-2}\cdots y_1y_0$，根据负数的补码和原码之间的关系：

$$[-y]_{补}=1,\overline{y_{n-2}y_{n-3}\cdots y_1y_0}+1 \tag{3-2}$$

对比式(3-1)和式(3-2)可知，$[-y]_{补}$等于$[y]_{补}$连同符号位在内的变反后末位再加1。即

$$[-y]_{补}=[[y]_{补}]_{变补}$$

(2) $y<0$。

因为y是负数，则

$$[y]_{补}=1,y_{n-2}y_{n-3}\cdots y_1\ y_0 \tag{3-3}$$

根据负数的补码和原码之间的关系：$[y]_{原}=1,\overline{y_{n-2}y_{n-3}\cdots y_1y_0}+1$

而$[-y]_{原}$只是将$[y]_{原}$的符号位变为0，即$[-y]_{原}=0,\overline{y_{n-2}y_{n-3}\cdots y_1y_0}+1$。

因为y是负数，$-y$为正数，所以

$$[-y]_{补}=[-y]_{原}=0,\overline{y_{n-2}y_{n-3}\cdots y_1y_0}+1 \tag{3-4}$$

对比式(3-3)和式(3-4)可知，$[-y]_{补}$等于$[y]_{补}$连同符号位在内的变反后末位再加1，即

$$[-y]_{补}=[[y]_{补}]_{变补}$$

注意：某数的“补码”和“变补”是不同的。一个数是正数时其补码和原码相同；对于负数，其补码也是在符号位不变的前提下，对数值位按位变反末位加1。而“变补”则不论是正还是负，一律包括符号位在内按位变反末位加1。

根据变补的概念，则

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

即$[x-y]_{补}$的运算可以变为求$[x]_{补}$和$[[y]_{补}]_{变补}$加法运算。这样，求两个有符号数差的过程中没有了减法运算，只需要将补码表示的减数进行变补后与被减数相加即可。

【例3-3】 设$x=-1001,y=+0101$，利用补码减法运算求$x-y$。

解：$[x]_{补}=1,0111,[y]_{补}=0,0101$

$$[-y]_{补}=[[y]_{补}]_{变补}=1,1011$$

$$\begin{array}{rll} & 1,0111 & [x]_{补} \\ + & 1,1011 & [-y]_{补} \\ \hline & \boxed{1}\ 1,0010 & [x-y]_{补} \end{array}$$

即$[x-y]_{补}=1,0010$，所以$x-y=-1110\mathrm{B}=-14$。

验证：$x=-1001\mathrm{B}=-9,y=+0101\mathrm{B}=+5,x-y=-14$，结果正确。

与例2-2类似，这里最高位的进位丢掉，结果正确性。

说明：补码运算电路的输入端是两数的补码，输出端也是补码。与补码加法运算一样，

补码减法运算时，参加运算的已知数据是 x、y 的补码。在进行求$[-y]_{补}$时，应根据$[y]_{补}$来求$[[y]_{补}]_{变补}$，不能直接根据 y 的真值写出$[-y]_{补}$。做题步骤是计算机工作过程的体现，步骤中第一步写出$[x]_{补}$、$[y]_{补}$，表示 x、y 的真值已转化为机器码存入内存或寄存器中，后续步骤是在这两个原始数据的基础上开始运算的。

【例 3-4】 设 $x=+0110$，$y=-0101$，利用补码减法运算求 $x-y$。

解：$[x]_{补}=0,0110$，　$[y]_{补}=1,1011$

$$[-y]_{补}=[[y]_{补}]_{变补}=0,0101$$

$$\begin{array}{rll} & 0,0110 & [x]_{补} \\ + & 0,0101 & [-y]_{补} \\ \hline & 0,1011 & [x-y]_{补} \end{array}$$

即$[x-y]_{补}=0,1011$，所以 $x-y=+1011\mathrm{B}=+11$

验证：$x=+0110\mathrm{B}=+6$，$y=-0101\mathrm{B}=-5$，$x-y=6-(-5)=+11$，结果正确。

综上所述，补码加减运算的规则如下：

(1) 参与运算的两个操作数用补码表示；

(2) 符号位与数值位一样参与运算；

(3) 若相加，两数的补码直接相加；若相减，将减数的补码连同符号位一起按位取反，末位再加 1，然后与被减数的补码相加；

(4) 运算的结果为和(或差)的补码。

3.1.3　溢出概念及检测方法

1. 溢出的概念

通过前面关于定点数据和浮点数据格式的介绍已经知道，不同数据格式的机器码所能表示的数据范围是不同的。例如 n 位机定点整数格式，补码能表示的数据范围是$[-2^{n-1}, 2^{n-1}-1]$。在运算过程中，若数据超出机器所能表示的数据范围就称为有溢出。如果超出机器所能表示的最大正数，称为正溢出；超出机器所能表示的最小负数，称为负溢出。

【例 3-5】 设 $x=+1011$，$y=-0110$，求利用补码减法运算试 $x-y$。

解：　$[x]_{补}=0,1011$，　$[y]_{补}=1,1010$，　$[-y]_{补}=[[y]_{补}]_{变补}=0,0110$

$$\begin{array}{rll} & 0,1011 & [x]_{补} \\ + & 0,0110 & [-y]_{补} \\ \hline & 1,0001 & [x-y]_{补} \end{array}$$

即$[x-y]_{补}=1,0001$，所以 $x-y=-1111\mathrm{B}=-15$。

验证：$x=+1011\mathrm{B}=+11$，$y=-0110\mathrm{B}=-6$，$x-y=11-(-6)=-17$，显然补码运算的结果不正确。

这是因为 5 位二进制补码所能表示的数据范围是$-16\sim+15$，而正确的 $x-y$ 运算结果 17 超出了这个范围。补码运算过程中，机器无法正确表示它。由于超出了所能表示的正数最大值，是正溢出。

2. 溢出的检测方法

发生溢出时，补码运算得到的结果不是正确的实际运算结果。因此要采取一定的办法检测出是否有溢出，以便及时进行相应的处理。常用的溢出的检测方法有以下两种。

(1) 采用双进位位判别。设 C_n 为两补码相加时符号位的进位，C_{n-1} 为最高数值位向符号位的进位，溢出条件为

$$\mathrm{OVR}=C_n \oplus C_{n-1}$$

即，若两进位位相同，则没有溢出，若不同，则表示有溢出。

【例 3-6】 设 $x=-1011$，$y=-0110$，利用补码加法运算求 $x+y$。

解：$[x]_{补}=1,0101$，$[y]_{补}=1,1010$

$$\begin{array}{rll} & 1,0101 & [x]_{补} \\ + & 1,1010 & [y]_{补} \\ \hline & 10,1111 & [x+y]_{补} \end{array}$$

此时，$C_n=1$，$C_{n-1}=0$，OVR=1，表示运算过程中发生了溢出。因此，运算结果是错误的。大家可以自己验证。

这种检测方法的电路实现相当简单，因此广泛采用。但是，它只能检测出有溢出，说明结果是错误的，却不能得到正确的结果。

(2) 采用双符号位的方法。一个符号位只能表示正、负两种情况，当发生溢出时，符号位的含义就产生了混乱。如果将符号位扩充为两位(S_{f1}、S_{f2})，可以证明，其 4 种组合分别表示以下 4 种情况。

$S_{f1}\ S_{f2}=00$，无溢出，结果为正；

$S_{f1}\ S_{f2}=11$，无溢出，结果为负；

$S_{f1}\ S_{f2}=01$，正溢出；

$S_{f1}\ S_{f2}=10$，负溢出。

即，若两符号位相同，则无溢出，否则有溢出。因此，溢出条件为

$$\mathrm{OVR}=S_{f_1} \oplus S_{f_2}$$

当有溢出时，最高位仍然表示正确结果的符号。

双符号位的补码也称为“变形补码”。可以证明，变形补码的加减运算规则与上述普通补码的运算规则一样成立，双符号位同时像数值位一样参与运算。

【例 3-7】 利用变形补码重做例 3-4。

解：$[x]_{补}=\mathbf{00},0110$，$[y]_{补}=\mathbf{11},1011$，$[-y]_{补}=[[y]_{补}]_{变补}=\mathbf{00},0101$

$$\begin{array}{rll} & \mathbf{00},0110 & [x]_{补} \\ + & \mathbf{00},0101 & [-y]_{补} \\ \hline & \mathbf{00},1011 & [x-y]_{补} \end{array}$$

即$[x-y]_{补}=\mathbf{00},1011$。由于 $S_{f1}\ S_{f2}=\mathbf{00}$，无溢出，结果为正，所以 $x-y=+1011\mathrm{B}=+11$

【例 3-8】 利用变形补码重做例 3-6。

解：$[x]_{补}=\mathbf{11},0101$，$[y]_{补}=\mathbf{11},1010$

$$\begin{array}{rll} & 11,0101 & [x]_{补} \\ + & 11,1010 & [y]_{补} \\ \hline & \boxed{1}\,10,1111 & [x+y]_{补} \end{array}$$

最高符号位的进位自然丢掉。由于 $S_{f1}\ S_{f1}=\mathbf{10}$，所以有负溢出。

双符号位检测溢出的方法不仅可以判断溢出，并能指出是正溢出还是负溢出。更重要

的是，如果有溢出，还可以对结果做简单的处理后得到正确的结果。只需要将低位符号位作为最高数值位，即将区分符号位和数值位的标记左移一位，即可以得到正确的运算结果。

如例 3-8 中，补码加运算有溢出，$[x+y]_{补}$ 的正确结果 1,01111，其中分隔符右边为数值部分，左边为数据的符号位。即 $x+y=-10001\mathrm{B}=-17$，结果正确。

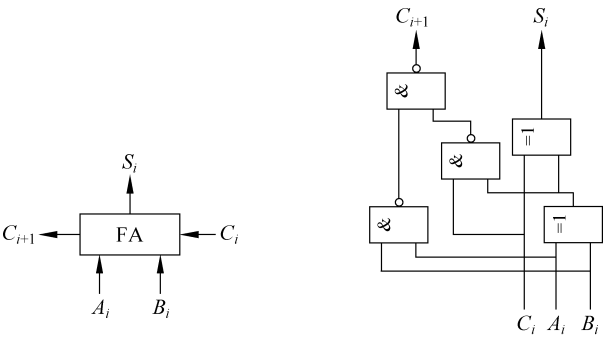
第 1 章中已经知道，参加运算的操作数和运算结果存放在主存单元或寄存器中。为了减少电路的开支，采用双符号位的方法时，操作数和结果在寄存器和主存单元存储时，仍保持单符号位，只是在运算时才扩充为双符号位。

3.1.4 基本二进制加法/减法器

补码加减运算的核心是多位二进制加法器，后面介绍的定点乘、除法运算和浮点加、减、乘、除运算，其运算的核心部件仍是加法器，加法器的性能直接影响着运算器的性能。这里讨论基本二进制加/减法器的实现方法和提高运算速度的相关措施。

1. 串行进位的加/减法器

全加器的逻辑符号和内部原理如图 3-1 所示。



(a) 逻辑符号　　(b) 原理

图 3-1　全加器的逻辑符号及原理

图 3-1 中，A_i、B_i 为参加运算的两个二进制数，C_i 是低位传来的进位，S_i 为本位和，C_{i+1} 为向高位的进位。其输入输出间的逻辑关系表达式为

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

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$$

根据多位二进制运算规则，多个一位二进制全加器级联可以构成多位二进制加法器。若再将每个全加器的 B 输入端附加一定的控制电路，即可构成串行进位的基本二进制加/减法器，如图 3-2 所示。

(1) 电路的基本功能。

该电路可以实现有符号数的加法、减法，也可以实现无符号数的加法和减法。

① 当电路输入信号为两个有符号数 A 和 B 的补码时的结果。

$M=0$，B 的补码经过异或门不受影响。多位全加器的加法运算实现的是 A 的补码和 B 的补码的求和，即两个有符号数的加法运算。输出结果 S 为两个有符号数补码的和，或者说和的补码。

$M=1$，B 的补码经过异或门后取非，同时最低位全加器的 C_0 为 1，即多位全加器的输

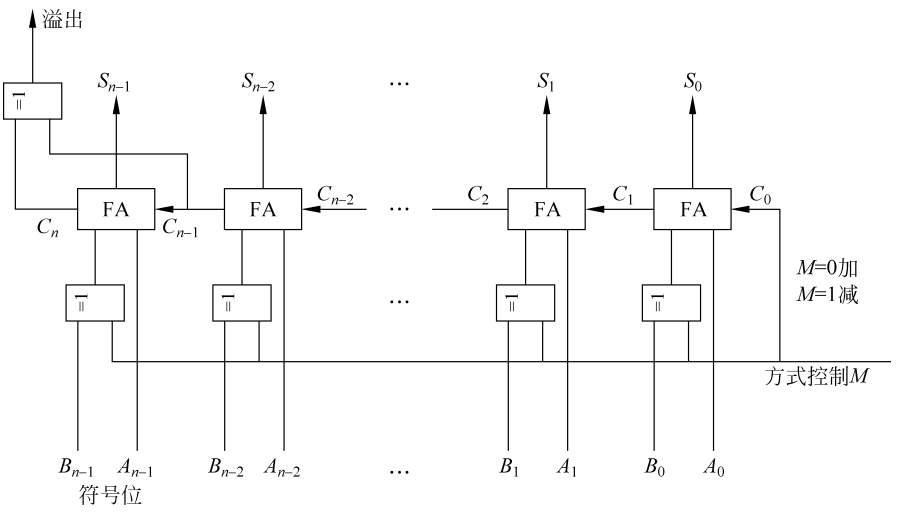


图 3-2　串行进位的基本二进制加/减法器

入端是$[[B]_{补}]_{变补}$。因此，多位全加器的加法运算完成的是$[A]_{补}$和$[[B]_{补}]_{变补}$相加，即$[A]_{补}+[-B]_{补}$，实现了两个有符号数的减法运算。输出结果 S 为两个有符号数补码的差，或者说差的补码。

② 当电路输入信号为两个无符号数 A 和 B 时的结果。

$M=0$，无符号数 B 经过异或门不受影响，多位全加器的加法运算实现的是无符号数 A 和 B 的求和，即两个无符号数的加法运算，结果为两无符号数的和。

$M=1$，无符号数 B 经过异或门后取非，同时最低位全加器的 C_i 为 1。经过多位全加器的加法运算实现的是 A 和 B 变补之后的求和，相当于两个无符号数的减法运算。结果为两无符号数的差，并以补码形式表示。

(2) 电路的时延。

令 $G_i=A_i B_i$，$P_i=A_i \oplus B_i$，上述 C_{i+1} 的函数表达式变为

$$C_{i+1}=G_i+P_i C_i$$

则

$$C_1=G_0+P_0 C_0$$
$$C_2=G_1+P_1 C_1$$
$$\cdots$$
$$C_n=G_{n-1}+P_{n-1} C_{n-1}$$

若根据上述表达式实现运算过程，由于每一级的进位都直接依赖于低一级的进位，即进位信号是逐级形成的。因此，这样的加法器也称为串行进位加法器，或行波进位加法器。

假设每个“与”门、“或”门的延迟时间为 T，则每一级全加器单纯由进位引起的延迟时间为 $2T$。设一个异或门的延迟时间近似为 $3T$，则图 3-2 所示的 n 位串行二进制加法器的总延迟时间为 $9T+2nT$。

当二进制运算的位数较多时，进位信号传递带来的延迟时间将成为影响加法器运算速度的主要因素，且与位数成正比例关系。如何改进以提高加法器的运算速度？

2. 并行加法器的快速进位

将上述串行进位关系表达式进行如下变换：

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

…

若以每个表达式中最后的等号为依据实现运算过程，每一个进位表达式都是多个与门之后再相或的逻辑，即各级进位均可以由两级门电路组成。由于每一个 G_i、P_i 都只与该位的输入数据 A_i、B_i 有关，因此各进位信号可以同时产生，即进位不需要传递。这样构成的加法器叫做并行进位加法器。

并行进位也叫先行进位(Carry Look Ahead，CLA)。对于 n 位并行加法器，其进位的延迟时间只有 $2T$，与 n 无关。这种方式大大提高了加法器的运算速度。

但是，随着加法位数的增多，高位进位表达式中的乘积项及每个乘积项的因子数会很多。实现高位进位的电路中与门、或门的输入端会很多，因此采用单级并行进位是不现实的。一般单级并行进位加法器 4 位的较多，3.6.2 节介绍的 Intel 74181 就是 4 位并行进位加法器实现的 ALU。

3. 分组并行进位方式

实际上，多位并行进位通常采用分组方式。把 n 位字长分成多个小组，组内各位之间是并行进位，组与组之间有串行进位或并行进位两种。

(1) 组内并行，组间串行方式。以 16 位二进制加法为例。16 位可以分成 4 组，每 4 位一组采用一个 4 位并行进位加法器。4 个 4 位并行进位加法器级联可以构成 16 位组内并行进位、组间串行进位加法器，其逻辑框图如图 3-3 所示。

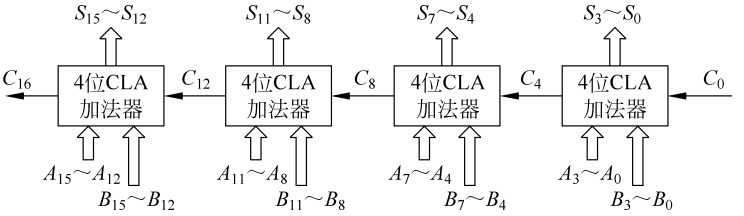


图 3-3　16 位组内并行、组间串行进位加法器

这种方式的进位延迟时间为 $4 \times 2T$，与组数成正比。当组数较多时，这种方式的时延是不容忽视的。

(2) 组内并行，组间并行方式。当加法器的位数大于等于 16 时，为了加快运算速度，一般采用多级先行进位方式。下面以 16 位加法器为例，说明两级先行进位的方法。

对 4 位先行进位的最高进位信号 C_4 的表达式进行变换：

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 = G_1^* + P_1^* C_0$$

其中，

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

$$P_1^* = P_3 P_2 P_1 P_0$$

同理可推出16位加法器的高位的进位如下：

$$C_8 = G_2^* + P_2^* C_4 = G_2^* + P_2^* G_1^* + P_2^* P_1^* C_0$$

$$C_{12} = G_3^* + P_3^* C_8 = G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* C_0$$

$$C_{16} = G_4^* + P_4^* C_{12} = G_4^* + P_4^* G_3^* + P_4^* P_3^* G_2^* + P_4^* P_3^* P_2^* G_1^* + P_4^* P_3^* P_2^* P_1^* C_0$$

由于 P_i^*、G_i^*（$i=1\sim4$）都只与自己组内的4位加法器的 P_i 和 G_i 有关，而 P 和 G 又只与自己的4位加法器的输入数据 A_i、B_i 有关，因此，如果上述4个函数表达式采用同样的与门、或门实现，则 C_4、C_8、C_{12}、C_{16} 可以同时输出，即组间是并行的，无须等待低位组的进位。16位组间并行进位加法器的逻辑如图3-4所示。其中BCLA是先行进位加法器(CLA)的改进电路：最高位向前的进位(C_4)不引出，而是将 P_i^*、G_i^* 引出。

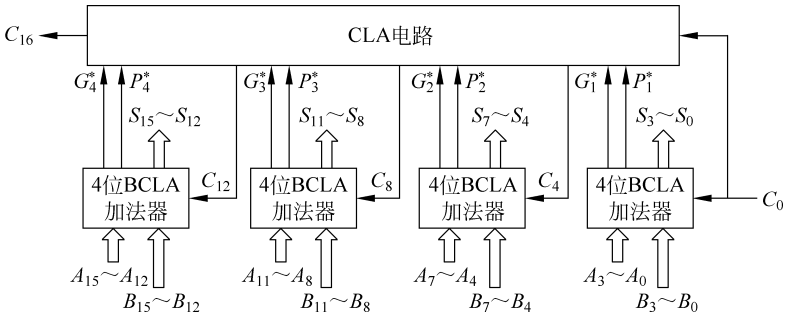


图3-4　16位组间并行进位加法器的逻辑

估算一下，组间并行进位方式下，16位二进制加法器的延迟时间。若不考虑每个4位二进制先行进位加法器的所有 P 和 G 的延迟时间，C_0 经过 $2T$ 产生第1小组的 C_1、C_2、C_3 及 G_i^* 和 P_i^*；再经过 $2T$，由CLA电路产生 C_4、C_8、C_{12}、C_{16}；再经过 $2T$，才能产生第2、3、4小组内的 $C_5\sim C_7$、$C_9\sim C_{11}$、$C_{13}\sim C_{15}$。即组间先行进位加法器的输出需要延迟时间是 $6T$，与 n 无关。

采用三级先行进位结构可以扩展到64的位加法器。这样，加法器的字长对延迟时间的影响很小。

3.1.5　十进制数的加法运算

前述定点加减运算的数据是二进制数，有些通用计算机中还设置了十进制整数的加减运算功能。十进制数的加减运算可以直接设计相应的逻辑电路实现，也可以在二进制加法器的基础上加上适当的调整电路来实现，在计算机中多数采用后者。

1. 十进制BCD码相加的运算规则

一位十进制数相加，向前的进位代表10。例如7＋8＝15，得到两位数，高位的1代表10。在计算机中，十进制数采用BCD码表示，即一位十进制数采用4位二进制编码表示，7和8的8421BCD码为0111和1000。若采用4位二进制数加法电路直接实现十进制数的BCD码加法运算，则0111＋1000＝1111，显然不是15的BCD码(0001 0101)，结果不正确。这是由于4位二进制相加，最高位向前的进位表示16，与十进制相加的进位相差6。若将4位二进制相加的结果再加上6，即1111＋0110＝0001 0101，正好是15的BCD码。因此，利用二进制加减运算电路实现十进制数的BCD码加法运算需要进行校正。由于计算机中的

BCD 码可以有 8421BCD 码、2421BCD 码、余 3 码等，不同的 BCD 码的校正规则不同，相应的实现电路也不同。这里仅以 8421 码为例说明其校正方法。

两个一位十进制数(8421 码表示)相加的运算规则如下。

(1) 将两个 BCD 码数按“逢二进一”的原则相加，即按二进制数的加法规则相加。

(2) 当相加的和小于等于 9 时，无须校正，二进制加的结果正确。

(3) 当相加的和大于等于 10 时，+6 校正。加 6 时产生的进位为本位十进制数相加产生的进位。

例如，实现 9+8=17 的计算过程如下：

① 十进制的 9 和 8 以 8421 码 1001 和 1000 两个编码输入到二进制运算电路。

② 二进制加法器对 1001 和 1000 按照二进制加法规则进行运算，本位结果为 1000，最高位先前有进位。

③ 因运算结果大于等于 10，必须进行修正。由二进制电路完成本位 1000 加 0110(十进制 6)的运算，结果为 0111，是十进制 9+8 的本位和 7 的 BCD 码。

同理，实现 3+5 的计算过程是 0011+0101=1000，因为和小于等于 9，无须修正，1000 为 3+5 的运算结果 8 的 BCD 码。

2. 一位十进制加法电路

根据上述 BCD 运算规则，在基本二进制加法运算电路的基础上，增加校正实现一位十进制 BCD 加法运算的逻辑电路设计过程如下。

设 $S_3S_2S_1S_0$ 和 C_4 分别为两个一位十进制数 BCD 码加法运算的本位和及进位，$S'_3S'_2S'_1S'_0$ 和 C'_4 为普通的 4 位二进制加法的本位和及进位。4 位 8421 码结合一位进位最多可以表示 20 个数，其对应的二进制与一位十进制的 BCD 码之间的关系如表 3-1 所示。

表 3-1　8421 码的校正关系

十进制数	8421 码					校正前的二进制数					校正关系
	C_4	S_3	S_2	S_1	S_0	C'_4	S'_3	S'_2	S'_1	S'_0	
1～9	0	0	0	0	0	0	0	0	0	0	不校正
	…					…					
	0	1	0	0	1	0	1	0	0	1	
10	1	0	0	0	0	0	1	0	1	0	+6 校正
11	1	0	0	0	1	0	1	0	1	1	
12	1	0	0	1	0	0	1	1	0	0	
13	1	0	0	1	1	0	1	1	0	1	
14	1	0	1	0	0	0	1	1	1	0	
15	1	0	1	0	1	0	1	1	1	1	
16	1	0	1	1	0	1	0	0	0	0	
17	1	0	1	1	1	1	0	0	0	1	
18	1	1	0	0	0	1	0	0	1	0	
19	1	1	0	0	1	1	0	0	1	1	

根据表 3-1 的校正关系，容易得到 $C_4=C'_4+S'_3S'_2+S'_3S'_1$

$C_4=1$ 代表两个十进制数相加之和大于 10，需要进行校正。因此，可以利用 C_4 作为校

正因子，在第一级普通 4 位二进制加法器输出后，设置二级加法器，根据 C_4 的不同将第一级输出的和再加 0 或 6。具体电路如图 3-5 所示。

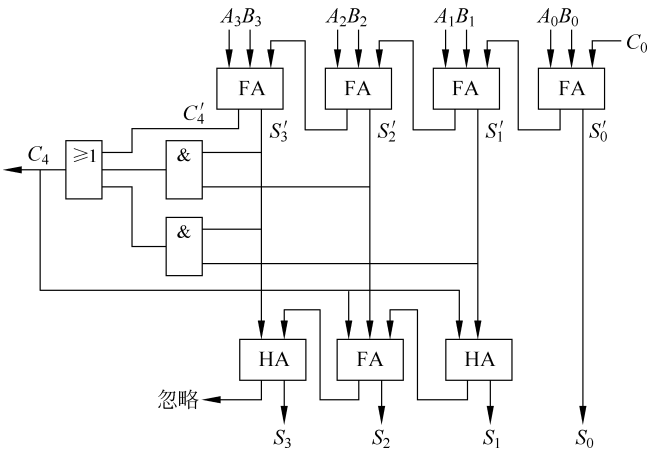


图 3-5　一位 8421 码十进制加法器

显然，十进制加法运算比二进制运算复杂，速度也要慢很多。

3.2　定点乘法运算

与加法和减法相比，定点乘法、除法运算要复杂得多。一些功能简单的单片机和早期的计算机中，通常不设置乘/除法指令，也就是说运算器中没有乘法运算的功能电路，而是采用软件的方式实现乘法运算。这样，运算器的硬件结构简单，但是软件实现乘法运算的速度太慢。随着微电子技术的发展，硬件实现乘法运算的成本降低，目前大多数计算机采用硬件实现乘法运算。

硬件实现乘法运算的方法大体上有两种：

(1) 在加法器的基础上，增加右移及其他一些控制电路实现；

(2) 采用高速阵列乘法器。

前者只需在加减运算器的基础上增加少量硬件，但是运算速度受限，多在微小型计算机中采用。大中型计算机中，由于其对乘法运算的性能要求较高，普遍采用阵列乘法器。

在乘法运算和后续的除法和浮点运算中，都会涉及移位操作，在移位时又会产生舍入处理问题。因此，本节先介绍移位和舍入操作，然后介绍前述两种硬件实现乘法运算的方法。

3.2.1　移位和舍入操作

1. 移位操作

移位可以分为左移和右移。对于带符号数和不带符号数，其移位操作的具体过程不同。

对于不带符号数来说，很简单，数据直接逐位左移或右移一位，结果等于原来值乘 2(左移)或除 2(右移)。例如二进制数 10010(十进制为 18)，左移一位，低位补 0，结果为 100100(十进制为 36)，是移位前的 2 倍。如果右移一位，最高位补 0 的结果为 01001(十进制为 9)，

是移位前的 1/2。这样的移位称为逻辑移位。

对于有符号数来说，通过移位也能实现乘 2 或除 2 的功能。由于带符号数的最高位是符号位，因此无论是左移还是右移，符号位都要保持不变。左移时最高数值位移出去丢掉，右移时最低数值位移出去丢掉。对于不同机器码，移位时空出位的处理方法不同。

(1) 原码的移位规则。无论正、负，无论左移、右移，符号位不变，空出的位补 0。

例如，在 6 位机中，二进制数－01010(十进制数是－10)移位前原码为 101010，左移后原码为 110100(十进制数－20 的原码)，右移后原码为 100101(十进制数－5 的原码)。

例如，在 6 位机中，二进制数＋00110(十进制数是＋6)移位前原码为 000110，左移后原码为 001100(十进制数＋12 的原码)，右移后原码 000011(十进制数＋3 的原码)。

(2) 补码的移位规则。无论正、负，无论左移、右移，符号位不变。正数空出的位均补 0；负数左移时空出的低位补 0，右移时空出的高位补 1。

例如，在 6 位机中，二进制数－01010(十进制数是－10)移位前补码为 110110，左移后补码为 101100(十进制数＋20 的补码)，右移后补码为 111011(十进制数＋12 的补码)。

例如，在 6 位机中，二进制数＋00110(十进制数是＋6)移位前补码为 000110，左移后补码为 001100(十进制数＋12 的补码)，右移后补码为 000011(十进制数＋3 的补码)。

上述例子通过左移、右移均实现了有符号数的乘 2(左移)或除 2(右移)运算。但是，若原始数据的最低位是 1 时，右移时会出现误差。若原始数据乘 2 的结果超出机器字长所能表示的数据范围时，左移时最高位的有效值被丢掉，保留的移位结果就是错误的。例如在 6 位机中，若二进制数为－10010，则十进制数是－18，移位前原码为 110010，左移后原码为 100100，即十进制数－4 的原码。左移时最高数值位 1 移丢，结果错误。移位前补码为 101110，左移后补码为 111100，即十进制数－4 的补码。左移时最高数值位 1 移丢，结果错误。

其实，乘 2 结果超出机器所能表示的数据范围，也是前面所说的溢出。因此，运算结果不发生溢出的情况下，按照上述规则左移或右移，可以实现无符号或有符号数乘 2 或除 2 的运算功能。

计算机中，移位功能通常由移位寄存器来实现。也有很多计算机不专门设置移位寄存器，而是在加法器的输出端加一个移位器，通过简单的与、或门，实现直传(不移位)、左斜移位(左移一位)和右斜移位(右移一位)的功能。其逻辑电路如图 3-6 所示。

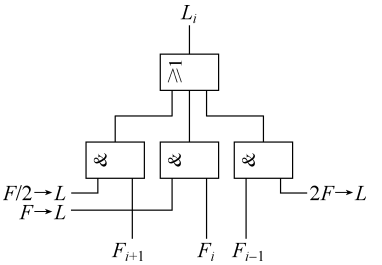


图 3-6　移位器逻辑电路

图 3-6 中，$2F\rightarrow L$、$F\rightarrow L$ 和 $F/2\rightarrow L$ 分别表示左移一位、直传和右移一位的 3 个控制信号。当 $2F\rightarrow L$ 有效时，F_{i-1} 位变为移位后的第 i 位，即 L_i，相当于左移一位，实现乘 2 运算；当 $F/2\rightarrow L$ 有效时，F_{i+1} 位变为移位后的第 i 位，即 L_i，相当于右移一位，实现除 2 运算；$F\rightarrow L$ 有效时，移位后的等于 F_i，即直传。

注意：移位器和移位寄存器不同，它本身只有移位功能，没有寄存功能，所以移位后的结果一定要保存到有关的寄存器中。3.6.1 节中的 AM2901 中就采用移位逻辑的方法。

2. 舍入操作

在移位操作中，由于受到硬件的限制，右移时低位部分移出去，需要进行一些舍入处理，尽可能减少移出带来的误差。舍入的方法有很多种，常用的有下面 4 种。

(1) 就近舍入。其实质类似于十进制的四舍五入，超出规定位数的多余位的值大于最低有效位值的一半，最低有效位加 1；小于一半，舍掉；正好是一半时，最低有效位为 0，多余位舍掉，最低有效位为 1，最低有效位加 1，向上进 1 使其变为 0。

例如，字长为 8 位的运算器，由于移位操作运算结果为 1000 1101 101，共 11 位。多余 3 位的值为 101，大于一半 100 的值，带划线的 1 是最低有效位，要加 1，舍入操作后的结果为 1000 1110。若移位操作的运算结果为 1000 1101 011，由于 011 小于 100，被舍掉，结果为 1000 1101。若移位操作的运算结果为 1000 1101 100，由于多余位等于 100，且最低有效位为 1，最低有效位加 1 后向上进 1，舍入操作后的结果为 1000 1110。若移位操作的运算结果为 1000 1100 100，多余位直接舍掉，舍入操作的结果为 1000 1100。

(2) 朝 0 舍入。即朝数轴原点方向舍入，就是简单截尾。这种方法容易导致误差积累。

(3) 朝$+\infty$舍入。对正数来说，只要多余位不为 0，最低有效位就加 1；对于负数来说，截尾，绝对值变小，数值变大。

(4) 朝$-\infty$舍入。这种处理方法正好与朝$+\infty$舍入相反。若为正数，只需简单截尾；若为负数，只要多余位不为 0，最低有效位就要加 1。

各种舍入方法特点不同，适应的场合不同，人们可以根据所处理数据的特点选择相应的舍入处理方法。

3.2.2 原码一位乘法

由于原码表示时，一个数的数值部分是该数的绝对值，因此对于有符号数的乘法运算，原码表示比补码表示更容易实现。

原码乘法只需要将原码的数值部分直接相乘，就可以得到乘积的数值部分，两数的符号位进行异或得到乘积的符号位。其本质是无符号数的乘法运算。下面从手工乘法运算入手，说明原码乘法运算的实现方法。

手工计算两个无符号二进制数的乘法过程与十进制乘法运算过程类似。例如：$x=1101$，$y=1011$，其求乘积的过程为

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

为了后续叙述的方便，乘数的每一位与被乘数的乘积被称为“位积”，位积累加过程中得到的结果，被称为“部分积”。

直接采用基本的加法器实现上述过程存在一些问题。一方面，计算机只能实现两个操作数相加，不能进行 n 个位积相加的运算；另一方面，n 位机中，加法器一般只能实现 n 位

加法，不能直接进行 $2n$ 位的加运算。

在早期的计算机中采用串行的 1 位乘法方案，把 n 个位积求和转化为 n 次“累加—移位”操作。具体过程是，初始部分积为 0；从乘数的最低位开始，每次求得的 1 位乘数的位积，与部分积进行一次累加，得到的部分积右移一次，再累加；这样进行 n 次“求位积—累加—移位”，最后得到的结果就是两数的乘积。

如果已知两数的原码，只需在上述无符号乘法运算结果的最高位前增加乘积的符号位，就可以得到原码的乘积。乘积的符号位通过两数的符号位异或运算得到。这种原码乘法运算的方案称为原码一位乘法，其逻辑实现如图 3-7 所示。

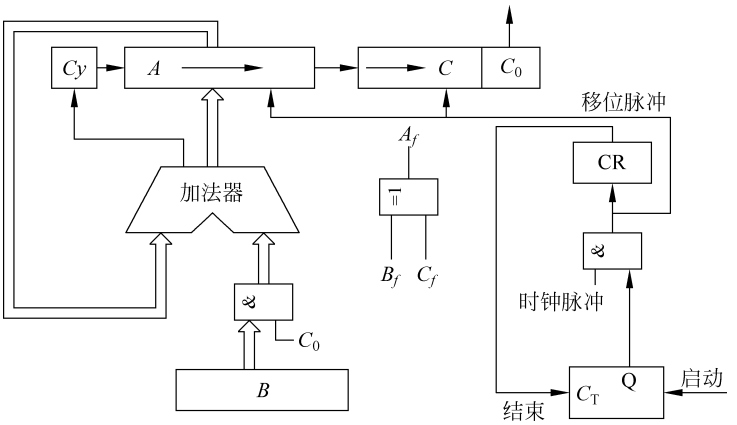


图 3-7　原码一位乘法运算原理图

说明：

(1) 运算前，被乘数和乘数的数值部分分别放在寄存器 B 和 C 中，符号位 B_f 和 C_f 单独处理。A 为部分积的高位，初始值为 0；CR 为减一计数器，初值为乘数的位数，即运算器的字长。运算后，乘数不再保留，A 存放乘积数值部分的高位，C 存放低位部分。

(2) 操作过程为，若乘数寄存器 C 的最低位 C_0 为 1，被乘数 B 进入加法器的输入端，与部分积 A 相加；如果 C_0 为 0，加法器输入端的值为 0，相当于不加。加法运算后，进位位 C_y、寄存器 A 和 C 的内容作为一个整体右移一位，即进位位成为 A 的最高位，原来的 A 的最低位进入 C 寄存器成为 C 的最高位，C 的最低位移出去自然丢掉。上述过程重复执行多次，执行次数由计数器的值(乘数的位数)决定。

这种方法不需要很多器件，只是在原有基本加法器的电路基础上，增加简单的移位及其控制电路，硬件简单。早期的微小型计算机中广泛采用。

【例 3-9】 设 $x=+1101$，$y=-1011$，试采用原码一位乘法计算 $x\times y$。

解：

作为原码乘法运算，其输入的原始数据是 x 和 y 的原码。

$$[x]_{原}=01101,\quad [y]_{原}=11011$$

运算前各寄存器的状态为

$$|x|=1101\rightarrow B,\quad |y|=1011\rightarrow C,\quad 0\rightarrow A\quad ,B_f=0,\quad C_f=1$$

开始运算：

	A寄存器	C寄存器	说明
	0 0 0 0	1 0 1 **1**	
+|x|	1 1 0 1		
	1 1 0 1		C_0=1，部分积加被乘数
	0 1 1 0	1 1 0 **1**	部分积右移一位
+|x|	1 1 0 1		
	□1 0 0 1 1		C_0=1，部分积加被乘数
	1 0 0 1	1 1 1 **0**	部分积右移一位，进位位移进来
+0	0 0 0 0		C_0=0，部分积加0
	1 0 0 1		
	0 1 0 0	1 1 1 **1**	部分积右移一位
+|x|	1 1 0 1		
	□1 0 0 0 1		C_0=1，部分积加被乘数
	1 0 0 0	1 1 1 1	部分积右移一位

$|x|\times|y|$=1 0 0 0 1 1 1 1

因为 $A_f=B_f\oplus C_f=0\oplus1=1$，所以$[x\times y]_{原}=110001111$，即 $x\times y=-10001111$。

验证：$x=+1101\text{B}=+13$，$y=-1011\text{B}=-11$，$(+13)\times(-11)=-143$，而 $x\times y=-10001111\text{B}=-143$，运算结果正确。

3.2.3 补码一位乘法运算

虽然原码乘法运算比补码乘法简单，但由于计算机中加减运算的有符号数多采用补码表示，也就是说，计算机存储的有符号数通常情况下是补码形式。因此运算器中的乘法运算也要针对补码表示的数据进行。

在基本加减法器基础上的补码乘法运算，大体有以下两种方法。

(1) 在原码一位乘法的基础上，增加算前求补和算后求补电路。算前求补电路根据乘数和被乘数的补码，得到相应的绝对值；然后由无符号一位乘法电路进行乘法运算；算后求补是将无符号数乘法运算的结果(即乘积的绝对值)，结合符号位得到乘积的补码。该方法是在无符号数的基础上增加算前求补和算后求补实现的补码乘法，也称为间接补码乘法。由于前述介绍了无符号数一位乘法的实现，后续仅介绍求补电路的实现。

(2) 像补码加减运算一样，将补码中的符号位与数值位一起进行乘法运算。这种补码乘法运算称为直接补码乘法。这里重点介绍直接的补码一位乘法。

补码一位乘法常用的方法是比较法，是由英国的 Booth 夫妇提出的，所以又称为 Booth 算法。Booth 算法实现补码一位乘法的过程与原码一位乘法的过程类似。不同之处如下：

(1) 运算前，寄存器 B 中存放被乘数的补码，C 中存放乘数的补码。

(2) 在乘数寄存器的最低位后增加了一个附加位，用 C_{-1} 表示。C_{-1} 的初值为 0。

(3) 考虑到符号数补码加减运算的溢出，加法器采用双符号位变形补码运算，即部分积和被乘数采用双符号位。

(4) 补码一位乘法运算时，控制逻辑要根据 C_0 与 C_{-1} 两位的组合情况，来决定部分积所加的内容。若 C_0 与 C_{-1} 的值相同，部分积加 0；若 C_0C_{-1} 两位的值为(0,1)，部分积加被乘数；若 C_0C_{-1} 两位的值为(1,0)，则部分积减被乘数。部分积累加运算后，寄存器 A、C 及 C_{-1} 整体右移。

(5) 由于加减法器进行的是补码运算，也就是说寄存器 A 中最高位存放的是符号位。因此移位时寄存器 A 的移位按照补码移位的规则进行。

(6) n 位补码数相乘时，共累加 n 次，移位 $n-1$ 次，最后一次累加后不移位。

【例 3-10】 设 $x=-1101$(十进制 -13)，$y=-1011$(十进制 -11)，试采用 Booth 算法计算 $x\times y$。

解：运算前各寄存器存储的原始数据：

$$[x]_{补}=11,0011\rightarrow B,\quad [y]_{补}=1,0101\rightarrow C,\quad 0\rightarrow A,\quad 0\rightarrow C_{-1}$$

由于检查相邻两位的值为(1,0)时，部分积要减被乘数，因此先求出$[-x]_{补}$

$$[-x]_{补}=[[-x]_{补}]_{变补}=00,1101$$

开始运算：

	A	C C_{-1}	说明
	00,0000	1,0 1 0 **1 0**	C_0为C的末位，C_{-1}为附加位
$+[-x]_{补}$	00,1101		C_0C_{-1}=10，部分积减被乘数
	00,1101		
	00,0110	1 1,0 1 **0 1**	部分积右移一位
$+[x]_{补}$	11,0011		C_0C_{-1}=01，部分积加被乘数
	11,1001		
	11,1100	1 1 1,0 **1 0**	部分积右移一位
$+[-x]_{补}$	00,1101		C_0C_{-1}=10，部分积减被乘数
	00,1001		
	00,0100	1 1 1 1,**0 1**	部分积右移一位
$+[x]_{补}$	11,0011		C_0C_{-1}=01，部分积加被乘数
	11.0111		
	11,1011	1 1 1 1**1, 0**	部分积右移一位
$+[-x]_{补}$	00,1101		C_0C_{-1}=10，部分积减被乘数
	00,1000		

由于 A、C 中的原始数据的最高位是符号位，最后运算结果中 C 的最低位是原来乘数的符号位，因此取 C 中的高位部分(丢掉最低位 1)1111 与 A 中的部分 00,1000 组成最终乘积结果。即$[x\times y]_{补}=00,10001111$，$x\times y=+10001111$(十进制 $+143$)

验证：$(-13)\times(-11)=+143$，运算正确。

【例 3-11】 设 $x=-1101$，$y=+1011$，试采用 Booth 算法计算 $x\times y$。

解：运算前各寄存器存储的原始数据：

$$[x]_{补}=11,0011\rightarrow B$$

$$[y]_{补}=0,1011\rightarrow C,0\rightarrow A,0\rightarrow C_{-1}$$

$$[-x]_{补}=00,1101$$

开始运算：

	A	C	C_{-1}	说明
	00,0000	0,1011	**10**	
$+[-x]_{补}$	00,1101			C_0C_{-1}=10，部分积减被乘数
	00,1101			
	00,0110	10,11	**11**	部分积右移一位
+0	00,0000			C_0C_{-1}=11，部分积加0
	00,0110			
	00,0011	01,10	**01**	部分积右移一位
$+[x]_{补}$	11,0011			C_0C_{-1}=01，部分积加被乘数
	11,0110			
	11,1011	001	0,**10**	部分积右移一位
$+[-x]_{补}$	00,1101			C_0C_{-1}=10，部分积减被乘数
	00.1000			
	00,0100	0001	**0,1**	部分积右移一位
$+[x]_{补}$	11,0011			C_0C_{-1}=01，部分积加被乘数
	11,0111			

$[x\times y]_{补}$=1 1,0 1 1 1 0 0 0 1

验证：$x\times y=-10001111\mathrm{B}=-143=(-13)\times(+11)$，运算正确。

【知识拓展】

Booth 算法原理的数学推理

设被乘数$[x]_{补}=x_s.x_1\cdots x_{n-1}x_n$，乘数$[y]_{补}=y_s.y_1\cdots y_{n-1}y_n$，最高位为符号位。

由于补码乘法时，补码的符号位要参与运算。若将$[x]_{补}$和$[y]_{补}$按原码规则运算，对所得结果通过校正可以得到正确的结果$[x\times y]_{补}$。校正关系如下式：

$$[x\times y]_{补}=[x]_{补}\times 0.y_1\cdots y_{n-1}y_n+[-x]_{补}\times y_s$$

对上式进行如下变换：

$$\begin{aligned}[x\times y]_{补}&=[x]_{补}\times(0.y_1y_2\cdots y_n)+[-x]_{补}\times y_s\\&=[x]_{补}\times(y_12^{-1}+y_22^{-2}+\cdots+y_n2^{-n})+[-x]_{补}\times y_s\\&=[x]_{补}\times\{-y_s+(y_1-y_12^{-1})+(y_22^{-1}-y_22^{-2})+\cdots\\&\quad+(y_n2^{-(n-1)}-y_n2^{-n})+0\}\\&=[x]_{补}\times\{(y_1-y_s)+(y_2-y_1)2^{-1}+\cdots+(0-y_n)2^{-n}\}\\&=[x]_{补}\times\{(y_1-y_s)+(y_2-y_1)2^{-1}+\cdots+(y_{n+1}-y_n)2^{-n}\}\end{aligned}$$

其中y_{n+1}为附加位，初值为0，$[x\times y]_{补}$，可以通过如下递推得到$[x\times y]_{补}$：

$$\begin{aligned}[z_0]_{补}&=0\\[z_1]_{补}&=2^{-1}\{[z_0]_{补}+(y_{n+1}-y_n)[x]_{补}\}\\[z_2]_{补}&=2^{-1}\{[z_1]_{补}+(y_n-y_{n-1})[x]_{补}\}\\&\vdots\\[z_n]_{补}&=2^{-1}\{[z_{n-1}]_{补}+(y_2-y_1)[x]_{补}\}\\[x\times y_n]_{补}&=2^{-1}\{[z_n]_{补}+(y_1-y_s)[x]_{补}\}\end{aligned}$$

其中，$[z_0]_{补}$为初始部分积，$[z_1]_{补}\sim[z_n]_{补}$分别为各次累加并移位后的部分积。

可以发现，每次累加时的增量取决于乘数相邻两位差值。若前一位大于后一位(1,0)，差值为负，部分积减去$[x]_{补}$；若前一位小于后一位(0,1)，差值为正，部分积加$[x]_{补}$；两位相同时，部分积加0。正是由于这种运算是根据乘数相邻两位的比较结果(y_i-y_{i-1})来确

定运算操作，因此称为比较法。

说明：这里是以定点小数为例进行的推导，事实上，计算机中的数据没有小数点，上述推导所得的运算方法，对整数乘法运算同样适用。

3.2.4 阵列乘法器

无论是原码一位乘法，还是 Booth 算法，实现 n 位数的相乘，都需要累加（或减）、移位 n 次（或 $n-1$ 次），随着 n 值的增大，即使采用并行进位的加法器实现，时延也是较大的。为了提高乘法的运算速度，可以将一位乘改为两位乘法，每次部分积右移两次，运算速度可以提高一倍，但这样仍不能满足某些高速运算器的要求。自从大规模集成电路问世以来，高速的单元阵列乘法器应运而生。为了达到高性能的乘法运算，目前普遍采用阵列乘法器。

1. 阵列乘法器的基本原理

设两个不带符号的二进制整数

$$A = a_{m-1}\cdots a_1 a_0$$

$$B = b_{n-1}\cdots b_1 b_0$$

则二进制乘积

$$P = A \times B = \left(\sum a_i 2^i\right)\left(\sum b_j 2^j\right) = \sum\sum (a_i b_j) 2^{i+j}$$

下面以 5 位×5 位为例说明并行阵列乘法器的基本原理。

设 $A = a_4 a_3 a_2 a_1 a_0$，$B = b_4 b_3 b_2 b_1 b_0$，则 $A \times B$ 的算式如下：

					a_4	a_3	a_2	a_1	a_0
				$\times$	b_4	b_3	b_2	b_1	b_0
					a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
		a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
+	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

若所有的 $a_i b_j$ 同时用与门逻辑产生，剩下的就是多项带进位的加法求和运算。显然，设计高速并行乘法器的基本问题，就是如何缩短被加数矩阵中执行加法的时间。

将上述求和过程做如下分解实现：

前两行纵向对齐的两项（a_1b_0，a_0b_1）、（a_2b_0，a_1b_1）、（a_3b_0，a_2b_1）、（a_4b_0，a_3b_1）各采用一个全加器分别相加（全加器的 C_{-1} 为 0）；得到的本位和向下分别与下一行对应的乘积项再次相加，此时全加器的 C_{-1} 为前一行全加器斜向的进位；得到的本位和再向下分别与再下一行的乘积项相加，以此类推，即图 3-8 所示实现过程。

图 3-8 中，FA 是一位全加器，FA 的斜线方向为进位输出，竖线方向为本位和输出。由于同一级的全加器运算之间没有任何关系，因此不存在进位时延。根据全加器内部结构，当本位和输出时，其对应的进位位已经输出，也就是说下一级的全加器运算时不需要额外等待上一级全加器的进位输出。这样，只有最后一次部分积相加时，要考虑全加器之间的进位时

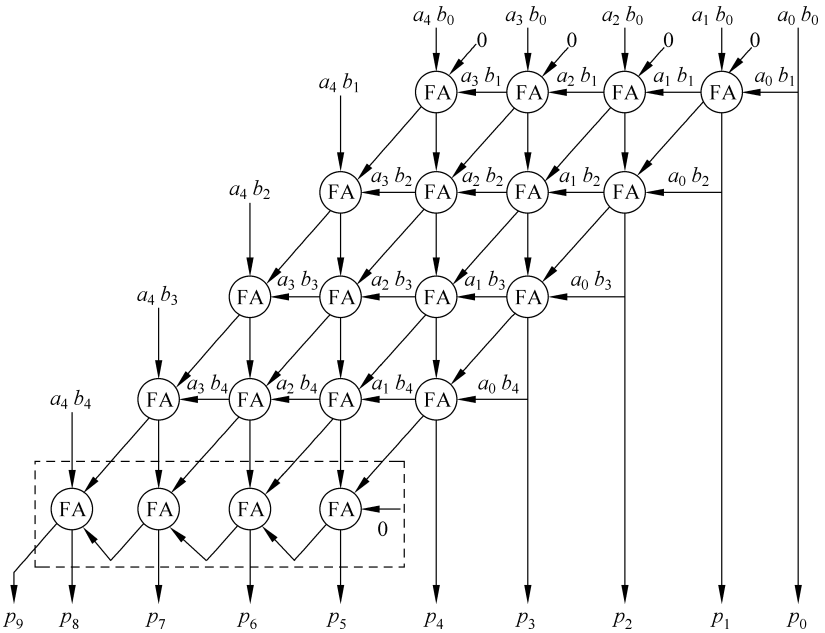


图 3-8　5 位×5 位不带符号数的并行乘法原理

延问题，也就是说只有一级的加法进位链时延要考虑。而前述相加——移位方法中每次的部分积相加都有进位链时延。因此，阵列乘法的运算速度大大提高。这种乘法器要实现 $n\times n$ 位时，需要 $n\times(n-1)$ 个全加器和 n^2 个“与”门，电路相对移位相加的方法复杂。

2. 带符号阵列乘法器

对于有符号数补码阵列乘法器，与前述一位乘法运算的原理类似，可以在无符号阵列乘法器的基础上增加算前求补和算后求补电路来实现，称为间接补码阵列乘法器。也可以将补码的符号位作为数值位直接参与阵列乘法运算，称为直接补码阵列乘法器。由于后者推导较为复杂，这里只介绍间接补码阵列乘法器的实现。间接补码阵列乘法器的原理如图 3-9 所示。

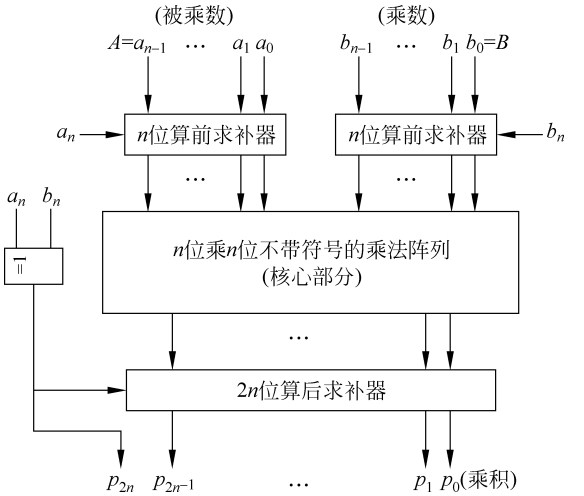


图 3-9　间接补码阵列乘法器原理

3. 求补电路

第 2 章中已经介绍，正数的补码，符号位为 0，数值部分与真值一样。负数补码的符号位为 1，数值部分的求法有两种：一种是真值的数值部分按位变反末位加 1；另一种是将真值数值部分从低位向高位依次检查，遇到第一个 1 之前的各位（包括第一个 1）保持不变，之后按位变反。第一种负数求补的运算可以在加法器的基础上增加一点电路来实现，这已在本章基本加减运算器中介绍。第二种求补运算的逻辑电路，如图 3-10 所示。

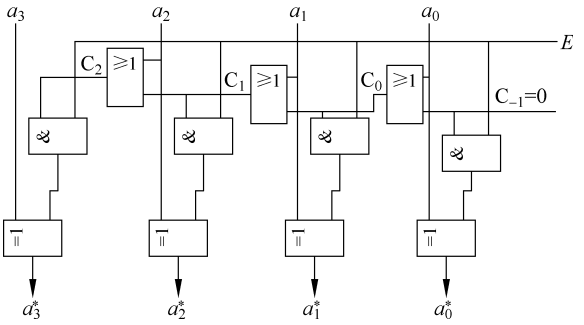


图 3-10　求补运算逻辑电路

图 3-10 中控制信号 E 是数的符号，$E=1$，表示负；$E=0$，表示正。对图中逻辑进行分析，如果输入信号是真值的数值部分，输出就是对应补码的数值部分；如果输入信号是补码的数值部分，输出的是对应真值的数值部分，也即绝对值。因此，该电路可以作为算前求补器，也可以作为算后求补器使用。

下面通过一个示例说明间接补码阵列乘法器的运算过程。

【例 3-12】 设 $x=+15$，$y=-13$，用带求补器的补码阵列乘法器实现 $x\times y$ 运算。

解：补码阵列乘法器要求输入信号是乘数和被乘数的补码，输出是两有符号数乘积的补码。

运算电路的输入数据：

$$[x]_{补}=0,1111,\quad [y]_{补}=1,0011$$

乘积的符号位单独处理，$[x]_{补}$ 和 $[y]_{补}$ 经算前求补后输出 x、y 的绝对值：

$$|x|=1111,\quad |y|=1101$$

$|x|$ 和 $|y|$ 经过无符号阵列乘法运算：

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

输出为两数绝对值的乘积，即 $|x|\times|y|=11000011$。

两数符号位异或得到乘积的符号：$1\oplus 0=1$。

在乘积的符号位的控制下，$|x|\times|y|$ 经过算后求补，得到整个电路的输出：$[x\times y]_{补}=100111101$，$x\times y=-11000011\mathrm{B}=-195$。

验证：$(+15)\times(-13)=-195$，运算结果正确。

3.3 定点除法运算

除法运算与乘法运算类似，对于有符号数的补码除法，也有间接补码除法器和直接补码除法器两种。前者是在无符号数除法运算电路的基础上增加算前求补器和算后求补器实现；后者是直接将补码的符号位和数值位一起参与运算。由于直接补码除法的原理更加复杂，而间接补码的算前、算后求补的原理又与乘法器的类似，因此本节只介绍无符号数的除法运算和实现方法。对于有符号数的符号处理问题参照乘法运算的相关内容。

计算机实现各种运算，都是在模拟人工运算的基础上，改进存在的问题，得到便于计算机实现的运算方法。对于定点除法运算，人们也从无符号数的手工除法运算入手。

3.3.1 手工运算

手工进行二进制除法运算与十进制运算完全类似。由于整数的商有可能是小数或包含小数，对于定点整数机来说，商的存储要进行一定的处理。因此，为了便于计算机实现，除法运算器较多地采用定点小数除法。要求被除数的绝对值小于除数的绝对值。若不满足，进行除法之前要先进行一些处理。

设 $x=0.1011$，$y=0.1101$，手工进行计算 x/y。

完全手工运算的过程如下：

```
                0.1 1 0 1
          ┌──────────
1 1 0 1 ) 1 0 1 1 0
          1 1 0 1
          ─────────
            1 0 0 1 0
            1 1 0 1
            ─────────
              1 0 1 0 0
              1 1 0 1
              ─────────
                0 1 1 1
```

运算结果：商等于 0.1101，余数为 0.0111×2^{-4}。

计算机实现时，不能完全模拟人的思维，可以考虑像乘法运算那样通过减—移位—再减的方法实现除法运算。将上述手工运算变换为如下过程。

```
                  0.1 1 0 1
              ┌──────────
0.1 1 0 1 ) 0.1 0 1 1 0          被除数小于除数，商上0
           −0.0 1 1 0 1          除数右移1位，相减，商上1
            ─────────
            0.0 1 0 0 1 0        得余数
           −0.0 0 1 1 0 1        除数右移1位，相减，商上1
            ─────────
            0.0 0 0 1 0 1 0 0    得余数。补一位后不够减，商上0，除数右移1位
           −0.0 0 0 0 1 1 0 1    除数右移1位，相减，商上1
            ─────────
            0.0 0 0 0 0 1 1 1    得余数。得到4位商，除运算结束
```

在上述运算过程中，去掉减法中高位的无效 0，与十进制除法的算式是完全类似的。这里这样写，主要是为了对步骤运算进行总结，以便计算机实现。实际上，上述运算过程中的“余数不动，除数右移”，也可以按照“除数不动，余数左移”运算。因为余数左移出去的高位是无用的 0，这样每次加法运算的位数是固定的，等于除数的位数。因此，计算机中通常都采用余数左移减除数的方法实现。

3.3.2 恢复余数除法

手工运算时，若不够减，则商上 0，补一位后再减。而计算机不能像人那样能够心算，如果不够减，即余数为负时，正常考虑是将减掉的除数再加上，恢复减之前的余数，然后再继续向下运算。这种处理方法叫做恢复余数法。

恢复余数法的具体运算过程为被除数减除数，若余数为正，商上 1，余数左移后减除数；若余数为负，商上 0，恢复余数，即把减掉的除数再加回去，然后余数左移后再减除数；得到新的余数后，再“判断正负→上商→余数左移→减除数”，连续进行多次，得到 n 位商，终止运算。

由于恢复余数法中，不够减时要增加一次加法，因此运算步骤不固定，控制比较复杂，而且恢复余数的加法运算也额外增加了运算时间。在实际计算机设计时多采用不恢复余数法。

3.3.3 不恢复余数除法

不恢复余数法与恢复余数法的不同在于，当余数为负，即不够减时，不再恢复余数，而是将下一步的减除数变为加除数运算。其数学依据如下：

设 R_i 为第 i 次的余数，B 表示除数。当余数为负时，(R_i+B)表示恢复余数，再进行第 $i+1$ 次处理，即余数左移再减除数，则第 $i+1$ 次的余数 $R_{i+1}=2\times(R_i+B)-B=2\times R_i+B$。等式右边的表达式相当于直接将 R_i 左移一位，再加除数。即当第 i 次余数为负时，可以不恢复余数，在下一步直接将 R_i 左移后加除数，得到第 $i+1$ 次的余数，即不恢复余数。

不恢复余数的运算步骤如下：

① 被除数减除数，余数为负，商上 0，余数左移后减除数。

② 若新的余数为正，商上 1，余数左移后继续减除数；若余数为负，商上 0，余数左移后下次加除数。

③ 重复步骤②，直到得到所需要位数的商，终止运算。

不恢复余数法运算步骤数固定，每次得到的位商可以控制下一步进行加法还是减法，因此控制简单、方便。在基本加减运算器的基础上实现的原理框图如图 3-11 所示。

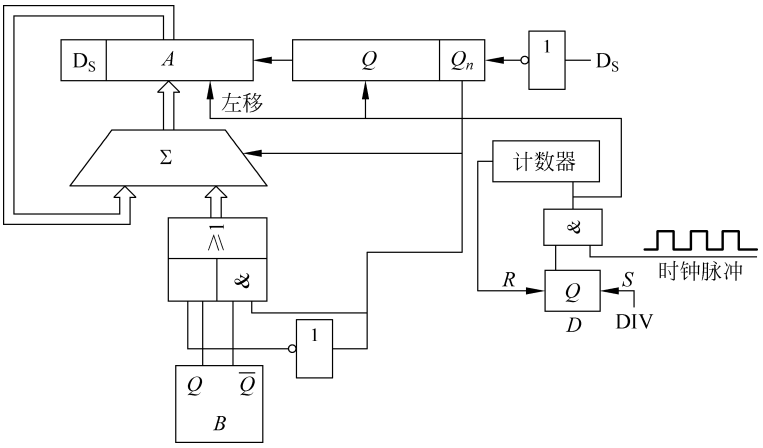


图 3-11　使用基于不恢复余数法的除法器

(1) 运算前，寄存器 B(n 位)放除数，寄存器 A($n+1$ 位)和 Q(n 位)存放被除数，其中 A 存放被除数的高位部分，Q 存放低位部分，不够部分低位补 0。运算过程中，每求出一位商放在寄存器 Q 的最低位，寄存器 A 存放余数。运算后，寄存器 Q 为最终的商，A 的内容右移多次后得到真正的余数。CR 计数器为减一计数，初值为除数的位数。

(2) 操作步骤如下：

① 被除数减除数，差为正，即被除数大于除数，进行溢出处理。差为负($D_S=1$)，商上 0(Q_n 置“0”)；A、Q 整体左移一位，$Q_n=0$ 控制加法器下一步做加法运算。在 $Q_n=0$ 的控制下，加法器的输入端为除数 B 的原变量，即余数(A)加除数(B)。

② 若 A(新余数)为负，即($D_S=1$)，商上 0；A、Q 整体左移一位，余数(A)加除数(B)。若 A(新余数)为正，即($D_S=0$)，商上 1(Q_n 置“1”)；A、Q 整体左移一位，$Q_n=1$ 控制加法器下一步做减法运算。在 $Q_n=0$ 的控制下，加法器的输入端为除数 B 的反变量，即余数(A)减除数(B)。

③ CR 计数器减一，若不等于 0，继续执行步骤②，直到 CR 减一为 0 结束运算。

注意：采用余数左移减除数的方法，最后运算结果的余数要进行右移。运算过程中左移了多少次，最后要右移多少次，才能得到正确的余数。

【例 3-13】 设 $x=0.101001$，$y=0.111$，试采用不恢复余数法计算 x/y。

解：运算前，$x=0.101001\rightarrow A$、Q，其中 $A=00.101$，$Q=0010$，$y=0.111\rightarrow B$，CR 计数器初值为 3。

由于运算过程中有减法，减法变为补码加，$[-y]_{补}=11.001$。

开始运算：

A	Q	说明
00.101	0010	
$+[-y]_{补}$ 11.001		
11.110	001**0**.	部分余数为负，商上0
11.100	01**0**.0	A、Q整体左移一位
$+[y]_{补}$ 00.111		部分余数加y
00.011	01**0.1**	部分余数为正，商上1
00.110	1**0.1**0	A、Q整体左移一位
$+[-y]_{补}$ 11.001		部分余数减y
11.111	1**0.1 0**	部分余数为负，商上0
11.111	**0.1 0** 0	A、Q整体左移一位
$+[y]_{补}$ 00.111		部分余数加y
00.110	**0.1 0 1**	部分余数为正，商上1

x/y的商为 0.1 0 1

余数为 0.110×2^{-3}

说明：最后得到的余数有可能是负数，结果以补码形式在寄存器 A 中。要得到正确的余数需要按补码右移的方式移位。

3.3.4 阵列除法器

在乘法运算的实现中，采用了全加器阵列，实现了并行操作，大大提高了乘法运算的速度。对于除法运算，也有阵列除法器提高除法运算的速度。而且对于不同的运算方法都有相应的阵列除法器。这里仅以不恢复余数串行阵列除法器为例，说明阵列除法器的思想。图 3-12 是实现不恢复余数法的串行阵列除法器的原理图。

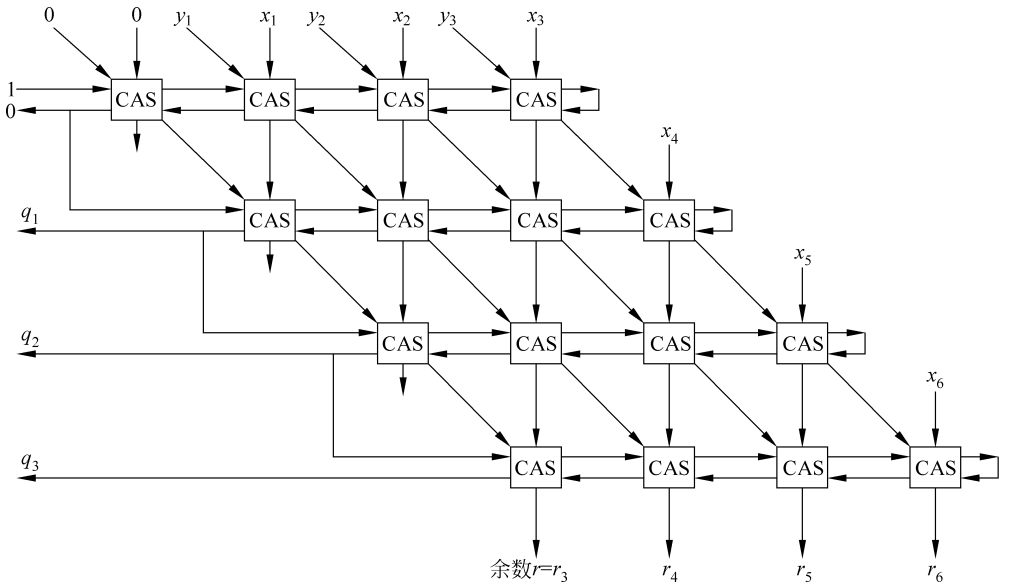


图 3-12　使用不恢复余数法的串行阵列除法器

图 3-12 中，被除数 $x=0.x_1x_2x_3x_4x_5x_6$，除数 $y=0.y_1y_2y_3$，商 $q=0.q_1q_2q_3$，余数 $r=0.00r_3r_4r_5r_6$。其中每一个方框为一个可控加法/减法（CAS）单元，其内部逻辑如图 3-13 所示。

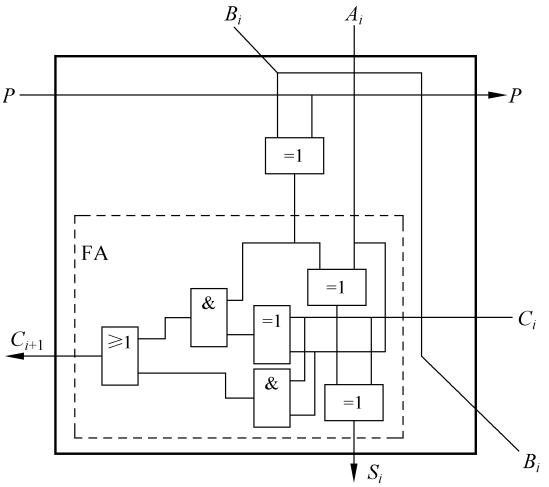


图 3-13　可控加法/减法(CAS)单元逻辑电路

可控加法/减法(CAS)单元的原理：当 $P=1$ 时，CAS 做减法运算，实现 $A_i-B_i-C_i$；当 $P=0$ 时，CAS 做加法运算，实现 $A_i+B_i+C_i$。C_{i+1} 表示产生的借位或进位，S_i 为本位和或差。

图 3-12 所示不恢复余数法阵列除法器的运算过程：

(1) 第一行 $P=1$，执行减法操作，即被除数减除数(n 位)，且余数为负，得商的最高位 0。

(2) 前一行的商作为下一行的 P 控制下一行进行加法或减法操作。即上一行的商为 0，下一行做加法运算；上一行的商为 1，下一行做减法运算；

(3) 余数不动，除数右移。

(4) 步骤(2)和(3)重复进行多次，直到得到 n 位商为止。

图 3-12 所示阵列除法器是采用多个 CAS 单元实现的不恢复余数除法。运算时，沿着每一行都有进位(或借位)传播，同时所有行在它们的进位链上都是串行连接。如果将 CAS 的 S_i 和 C_{i+1} 的函数表达式做如下变换：

$$\begin{aligned} S_i &= A_i \oplus (B_i \oplus P) \oplus C_i \\ &= A_iB_i\overline{C}_iP + A_i\overline{B}_i\overline{C}_iP + \overline{A}_iB_iC_iP + A_iB_iC_i\overline{P} + A_i\overline{B}_iC_iP + \\ &\quad \overline{A}_i\overline{B}_i\overline{C}_iP + \overline{A}_iB_i\overline{C}_iP + \overline{A}_i\overline{B}_iC_iP \\ C_{i+1} &= (A_i \oplus C_i) \cdot (B_i \oplus P) + A_iC_i \\ &= A_iB_i\overline{P} + A_i\overline{B}_iP + B_iC_i\overline{P} + \overline{B}_iC_iP + A_iC_i \end{aligned}$$

这两个表达式都可以采用一个三级组合逻辑电路(包括反向器)来实现，即每一个基本的 CAS 单元的延迟时间为 $3T$ 单元。

因此，对一个 $2n$ 位除以 n 位的不恢复余数阵列除法器来说，CAS 单元的数量为 $(n+1)^2$，考虑最大情况下的信号延迟，其除法执行时间为

$$t_d = 3(n+1)^2T$$

其中，n 为除数的位数。

显然，除法运算的时间比阵列乘法器的运算时间要长得多。因此，在设计程序时，能用乘法运算实现的功能，尽量不使用除法。例如，乘(除)2^n 的运算通常采用左(右)移的方法去实现，而不采用乘(除)法进行运算。

3.4 浮点运算

浮点运算的实现，可根据需要采用软件或硬件两种方式实现。这里仅讨论硬件实现的有关问题。

第 2 章中已介绍，数的浮点格式包括阶码和尾数两大部分，前者是定点整数，后者是定点小数，所以浮点运算可以在定点运算的基础上实现。为了确保浮点数的唯一性和最长的有效位数，参加浮点运算的数据和运算结果都必须是规格化的。

3.4.1 浮点加减运算

浮点数的加减运算比定点数的加减运算要复杂得多。手工进行浮点数运算时，首先要

通过小数点左移或右移将两数的小数点对齐，然后进行加或减。因此，计算机进行浮点加减运算，基本步骤大致包括对阶、尾数加减、规格化处理等。

设两个非 0 的规格化浮点数 x、y 分别为

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

其中，M_x、M_y 为浮点数 x 和 y 的尾数，E_x、E_y 为浮点数 x 和 y 的阶码。

1. 对阶

两个浮点数相加或相减，首先要把小数点对齐。浮点数中的小数点的实际位置取决于阶码的大小，对齐小数点就是使两数的阶码相等，这个过程叫做对阶。

首先求两阶码的差：

$$\Delta E=E_x-E_y$$

当 $\Delta E=0$ 时，两阶码相等，无须对阶，直接进入下一步，尾数进行加或减法操作。

当 $\Delta E\neq 0$ 时，需要进行对阶。对阶时，可以右移较小的数（增加它的指数），或左移较大的数（减小它的指数）。无论哪种操作都可能导致数字的丢失。但右移丢失的是低位的数字，比左移丢失的高位数字所造成的误差相对要小。因此，对阶操作总是阶码小的向阶码大的看齐，阶码小的尾数右移。尾数每右移一位，相应的阶码加 1，直到两数的阶码相等。右移的次数等于两数的阶差。当 $\Delta E>0$ 时，M_y 右移，每右移一位，E_y 加 1，直到 $E_x=E_y$ 为止。若 $\Delta E<0$ 时，M_x 右移，每右移一位，E_x 加 1，直到 $E_x=E_y$ 为止。

2. 尾数加减运算

对阶之后，尾数的加减运算实际上还是定点加减运算。其运算方法与前面介绍的定点加减运算方法相同，采用变形补码加减。这里不再赘述。

3. 尾数结果的规格化

尾数加减运算得到的数不是规格化数时，必须进行规格化。

尾数加减运算时，多采用双符号位补码运算。运算后的结果可能是以下 6 种情况之一。

(1) 00.1…；

(2) 11.0…；

(3) 00.0…；

(4) 11.1…；

(5) 01.…；

(6) 10.…。

第(1)和(2)两种情况，符合规格化的要求，已是规格化数，不需要再进行规格化处理。

第(4)种情况中，若 11.100…0，也是规格化数。

第(3)种情况和第(4)种中“××…×”为非全 0 的情况，不符合规格化的要求，需要尾数左移进行规格化处理。尾数每左移一位，相应的阶码减 1，直至成为规格化为止。这种规格化的过程称为左规。

第(5)和(6)两种情况，在定点运算中是溢出；但在浮点运算中，只表明尾数的运算结果大于 1，不能认为是浮点数的溢出，此时将尾数右移实现规格化。这种规格化的过程称为右规。右规最多只有一次。

4. 舍入处理

由于硬件的限制，在对阶和右规时有可能将尾数的低位丢失，对结果带来一定的误差。

关于舍入操作方法参见 3.2.1 小节。

5. 溢出判断和处理

与定点加减运算一样，浮点加减运算也可能出现溢出。浮点数的溢出是由阶码决定的。当阶码部分超出了它能表示的数的范围，就认为浮点数有溢出。若阶码正溢出(上溢)，浮点数真正溢出，机器停止运算，做溢出中断处理；若阶码负溢出(下溢)，浮点数趋于 0，机器不做溢出处理，而是当做机器零处理。

若两数中有一个为零，无须进行上述运算过程：

(1) 加法运算中的任何一个数为 0，结果直接就是另一个数。

(2) 减法运算时，若减数 y 为 0，结果等于被减数 x。若被减数 x 为 0，结果是将减数 y 改变符号。如果尾数采用补码表示，结果的尾数就是$[M_y]_{变补}$。

综上所述，浮点加减运算的过程可以用图 3-14 所示流程表示。

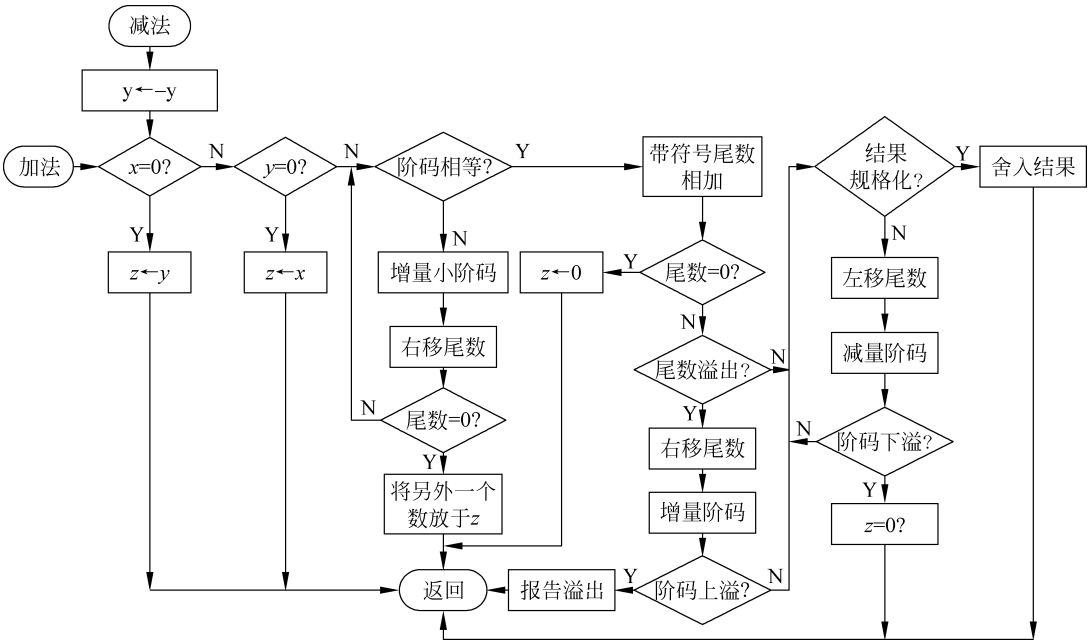


图 3-14　浮点加减运算的流程

【例 3-14】 若浮点数 $x=2^{010}\times 0.11011011$，$y=2^{100}\times(-0.10101100)$，求 $x+y$。假设浮点格式为：阶码占 4 位(包括阶符)，尾数占 9 位(包括数符)，阶码、尾数均用补码表示。

解：x、y 的浮点表示为

阶码　　尾数

$[x]_{浮}=\overbrace{0010}^{}\,,\overbrace{0.11011011}^{}$

$[y]_{浮}=0100,1.01010100$

注意：上式中中间的逗号只是为了阅读方便而添加的。

(1) 对阶。对阶过程和后续的规格化处理中，阶码的加、减运算采用双符号位的变形补码运算。

$$[\Delta E]_{补}=[E_x-E_y]_{补}=00\ 010+11\ 100=11\ 110$$

$$\Delta E = -010 < 0$$

x 的阶码小，其尾数右移 2 位，对阶后的 x 浮点格式为

$$[x]'_{浮} = 0100, 0.00110110(11)$$

(2) 尾数求和。尾数求和运算时采用双符号位的变形补码运算：

$$\begin{array}{rr} & 00.00110110(11) \\ + & 11.01010100 \\ \hline & 11.10001010(11) \end{array}$$

(3) 尾数规格化处理。由于尾数运算结果的符号位与最高数值位相同，是非规格化数据，所以应进行左规。尾数左移一次，变为 11.00010101(1)；阶码减 1，变为 00 011。

(4) 舍入处理。采用 0 舍 1 入的处理方法，尾数末位加 1，结果为 11.00010110。

(5) 判断溢出。阶码的符号位为 00，没有溢出。

因此，最终结果为

$$[x+y]_{浮} = 00\ 011, 1.00010110$$

$$x + y = 2^{011} \times (-0.11101010)$$

说明：运算前和运算后，寄存器中存放的浮点数的阶码和尾数全部是单符号位的补码形式。在运算过程中，阶码和尾数的加减运算器中采用的是双符号位补码。

注意：对于 IEEE 754 浮点格式，其尾数采用原码表示，整数部分的“1”是隐含的。运算是要按照其具体格式进行运算和规格化。

3.4.2 浮点乘除运算

浮点乘除运算相对加减运算来说简单得多。计算机实现两浮点数乘(除)的运算方法，与手工进行浮点乘除运算类似，两浮点数乘(除)，就是尾数乘(除)，指数加(减)。

设两个非 0 规格化浮点数：

$$x = M_x \times 2^{Ex}, \quad y = M_y \times 2^{Ey}$$

则

$$x \times y = (M_x \times M_y) \times 2^{Ex+Ey}$$

$$x / y = (M_x / M_y) \times 2^{Ex-Ey}$$

其中，尾数乘(除)的运算方法与前述定点数乘(除)的运算方法一样，这里就不再赘述。指数的加(减)运算则根据阶码的编码方式不同，运算方法不同。如果阶码采用补码表示，阶码的加(减)运算与定点数的补码加(减)运算方法一样。这里仅讨论阶码采用移码表示时的加减运算。

因为

$$[E_x]_{移} = 2^n + E_x, \quad [E_y]_{移} = 2^n + E_y$$

则

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

即

$$[E_x + E_y]_{移} = [E_x]_{移} + [E_y]_{移} - 2^n$$

因此，移码表示时，乘积的阶码是两浮点数的阶码相加(符号位于数值位一起参与运算)后再减 2^n。

同样

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

即

$$[E_x-E_y]_{移}=[E_x]_{移}-[E_y]_{移}+2^n$$

浮点数除法运算时，商的阶码是两浮点数的阶码相减（符号位与数值位一起参与运算）后再加 2^n。

说明：前面浮点数加减运算的对阶中，若阶码采用移码表示，求阶差的移码减法运算也要在阶码相减后加偏移量 2^n。

尾数乘（除）运算的结果也有可能不是规格化数据，因此也要进行规格化和舍入处理。阶码的加（减）也有可能产生上溢或下溢，也要进行溢出的判断和处理。

综上所述，浮点乘除运算的具体过程可以用图 3-15 所示流程表示。

【例 3-15】 设浮点数 $x=0.5$，$y=-0.4375$，求 $x\times y$。假设浮点数格式：阶码 4 位（包括阶码符号 1 位），移码表示；尾数 5 位（包括数的符号 1 位），补码表示。

解：先将十进制数变成二进制，然后写出相应的浮点形式：

$$x=0.5=0.1\mathrm{B}=(0.1)_2\times 2^0$$
$$y=-0.4375=-0.0111\mathrm{B}=(-0.111)_2\times 2^{-1}$$

则两操作数的浮点形式为

阶码　尾数

$$[x]_{浮}=1\ 000,0.1000$$
$$[y]_{浮}=0\ 111,1.0010$$

（1）阶码相加。

$$[E_x+E_y]_{移}=[E_x]_{移}+[E_y]_{移}-2^3=1\ 000+0\ 111-1000=0111$$

（2）尾数相乘。采用间接补码阵列除法器实现：

$$[M_x]_{补}=0.1000,\quad |M_x|=0.1000$$
$$[M_y]_{补}=1.0010,\quad |M_y|=0.1110$$
$$|M_x|\times|M_y|=0.1000\times 0.1110=0.01110000$$

因两操作数符号不同，乘积的符号为 1。

$$[M_x\times M_y]_{补}=1.10010000$$

（3）规格化及舍入处理。因为$[M_x\times M_y]_{补}=1.10010000$ 为非规格化数，所以要进行左规一次：

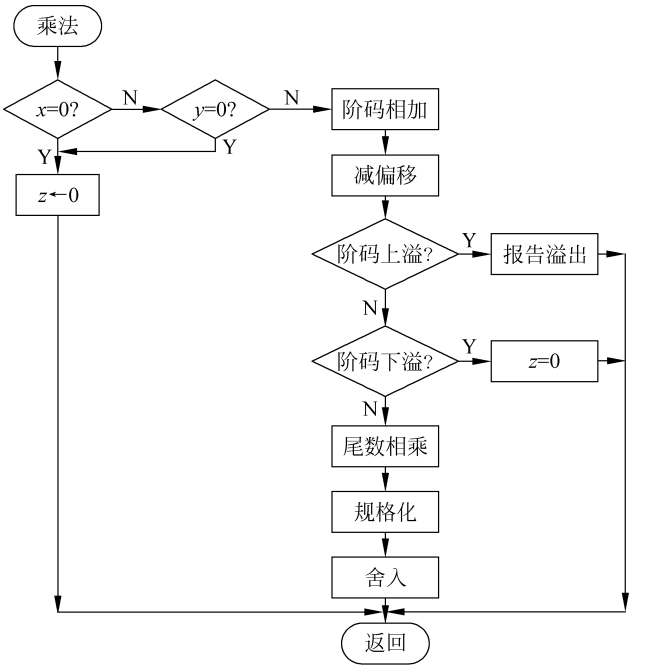
尾数左移一位，阶码减 1。故$[M_x\times M_y]_{补}=1.00100000$ 的尾数保留 5 位，舍入后，

$$[M_x\times M_y]_{补}=1.0010$$
$$[E_x+E_y]_{移}=0110$$

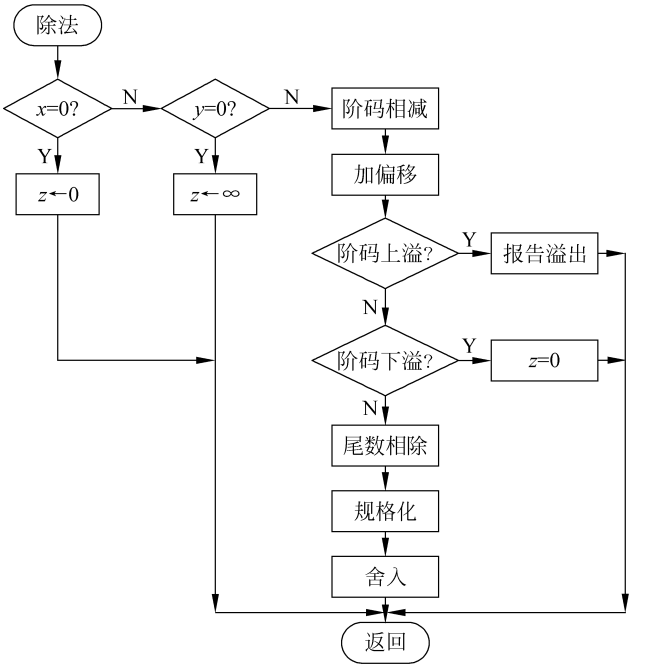
（4）溢出判断及处理。阶码无溢出，浮点乘法运算器输出结果为

$$[x\times y]_{浮}=0110,1.0010$$

所以



(a) 乘法运算的流程



(b) 除法运算的流程

图 3-15　浮点乘(除)运算的流程

$$x \times y = (-0.1110)_2 \times 2^{-10} = (-0.00111)_2$$

验证：$x \times y = (0.1)_2 \times 2^0 \times (-0.111)_2 \times 2^{-1} = (-0.00111)_2$，结果正确。

3.5 逻辑运算

计算机在解决实际问题时，除了需要进行加、减、乘、除等基本的算术运算外，还会遇到许多逻辑问题。常用的逻辑运算有逻辑与、逻辑或、逻辑非、逻辑异或等。

逻辑运算是按位进行，位与位之间没有任何关系，因此其实现比算术运算简单得多。

1. 逻辑非

对某数进行逻辑非运算，就是对该数的每一位进行求反。因此，逻辑非又称求反操作。

假设 $x = x_{n-1}\ x_{n-2} \cdots x_1\ x_0$，则 $z = \bar{x} = \overline{x_{n-1}\ x_{n-2} \cdots x_1\ x_0}$。

例如，若 $x = 10110010$，则 $\bar{x} = 01001101$。

每一位的求反操作用一个非门(反相器)实现即可。

2. 逻辑乘

逻辑乘运算是对参加操作的两数按位进行“与”操作，因此，逻辑乘运算又称逻辑与运算。

若参与运算的两操作数 $x = x_{n-1}\ x_{n-2} \cdots x_1 x_0$，$y = y_{n-1}\ y_{n-2} \cdots y_1 y_0$，则 x 和 y 的逻辑乘写为 $x \cdot y$。

若令 $z = x \cdot y = z_{n-1}\ z_{n-2} \cdots z_1\ z_0$，则 $z_i = x_i \cdot y_i\ (i = 0, 1, \cdots, n-1)$。

例如，设 $x = 10110010$，$y = 10101011$，则

$$\begin{array}{r} 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \quad x \\ \cdot \quad 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1 \quad y \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0 \quad x \cdot y \end{array}$$

即 $x \cdot y = 10100010$。

逻辑乘运算一般用与门实现。由于 $A \cdot B = \overline{A + B}$，因此逻辑乘也可用或非门实现。

3. 逻辑加

逻辑加运算是对参加操作的两数按位进行“或”操作，因此，逻辑加运算也称逻辑或运算。

若参与运算的两操作数 $x = x_{n-1}\ x_{n-2} \cdots x_1\ x_0$，$y = y_{n-1}\ y_{n-2} \cdots y_1\ y_0$，则 x 和 y 的逻辑加写为 $x + y$。

若令 $z = x + y = z_{n-1}\ z_{n-2} \cdots z_1\ z_0$，则 $z_i = x_i + y_i\ (i = 0, 1, \cdots n-1)$。

例如，设 $x = 10110010$，$y = 10101011$，则

$$\begin{array}{r} 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \quad x \\ + \quad 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1 \quad y \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1 \quad x + y \end{array}$$

即 $x + y = 10111011$。

逻辑加运算一般用或门实现。由于 $A + B = \overline{A \cdot B}$，因此逻辑加也可用与非门实现。

4. 逻辑异或

逻辑异或运算是对参加操作的两数按位进行“异或”操作。

若参与运算的两操作数为

$$x = x_{n-1}\ x_{n-2}\cdots\ x_1 x_0$$

$$y = y_{n-1}\ y_{n-2}\cdots\ y_1 y_0$$

x 和 y 的逻辑异或写为 $x \oplus y$。

若令 $z = x \oplus y = z_{n-1}\ z_{n-2}\cdots\ z_1 z_0$，则 $z_i = x_i \oplus y_i\ (i = 0, 1, \cdots, n-1)$。

例如，设 $x = 10110010$，$y = 10101011$，则

$$\begin{array}{rll} & 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0 & x \\ \oplus & 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1 & y \\ \hline & 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1 & x \oplus y \end{array}$$

即 $x \oplus y = 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1$。

逻辑异或可以用来实现模 2 加法运算。

逻辑异或运算可以直接用异或门实现，或用两级与非门实现。

3.6 运算器的基本组成与实例

运算器的主要功能是进行数据运算，其核心部件是算术逻辑单元(ALU)。除此之外，运算器中还包括一定数量用来临时存放数据的寄存器，进行数据选择的多路选择器以及实现内部数据传输的总线等部件。在实际应用中，由于对计算机速度、性能、用途、价格等方面的要求不同，运算器的具体逻辑组织差异很大。例如，参与运算的数据格式不同，定点运算器和浮点运算器的组织结构及复杂程度有很大区别；二进制和十进制运算的逻辑电路也有差异。即使是同一种运算，由于实现方法不同，其运算器的逻辑实现也会有很大差异。例如乘法运算有原码一位乘法、补码一位乘法、阵列乘法等多种实现方法，其相应的逻辑电路差别很大。另外，CPU 的运算指令功能不同，例如是否设置乘、除运算指令，也大大影响运算器的组织结构。本节只是简单地从一般概念上介绍运算器的基本组成及总线结构，给出一些定点运算器(包括 ALU)和浮点运算器的实例。

3.6.1 运算器的基本结构

1. 运算器的基本组成

运算器的基本组成包括实现算术、逻辑运算功能的 ALU，提供操作数据与暂存结果的寄存器组及有关的判断和控制电路等组成部分。

图 3-16 所示为运算器的一种基本组成结构，其 ALU 的输入端增加了一级锁存器。如果 ALU 输入数据的来源较多时，也可以在 ALU 输入端增加多路选择器，例如 AM 2901。

AM 2901 是一个 4 位二进制定点运算器，可以实现定点整数的加、减、移位和逻辑操作，其内部结构如图 3-17 所示。具体组成及功能如下：

(1) ALU 可以完成 3 种算术运算和 5 种逻辑运算。

(2) 有一个包含 16 个通用寄存器的寄存器组。

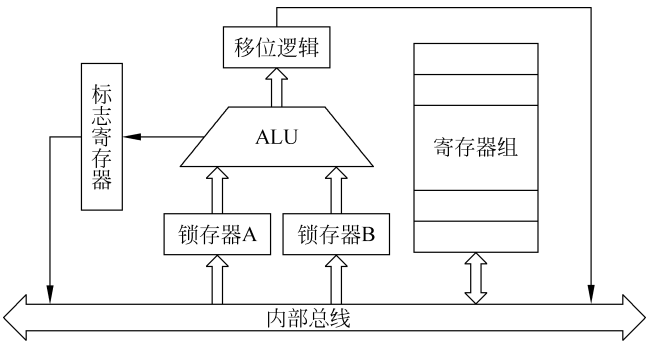


图 3-16　运算器的基本组成结构

(3) 通用寄存器组中的寄存器通过 A 锁存器或 B 锁存器为 ALU 提供操作数，由 A 口地址和 B 口地址对寄存器进行选择。

(4) 在 ALU 的输入端有两个多路选择器选择不同来源的输入数据参与运算。

(5) ALU 运算的结果可以由三态门控制是否输出到 Y。

(6) ALU 的运算结果存入通用寄存器之前有一个移位器，可以实现乘 2(左移)、除 2(右移)操作，也可以直通。

(7) 另有一个乘商寄存器能对自己的内容进行左、右移位，其结果也可以送往 ALU。

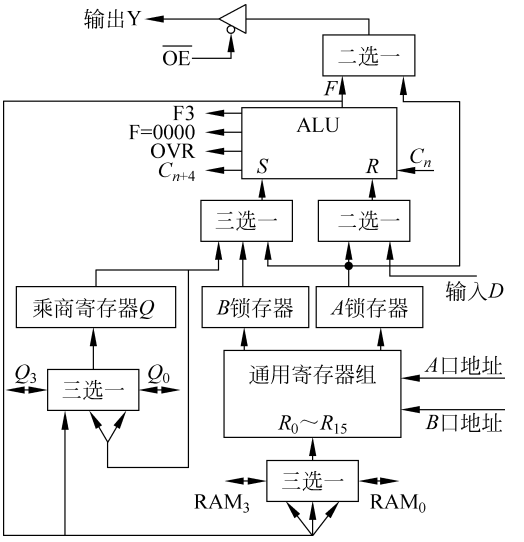
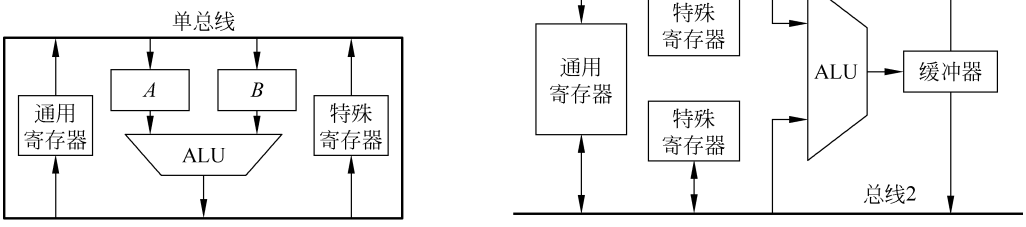


图 3-17　Am2901 运算器的内部结构

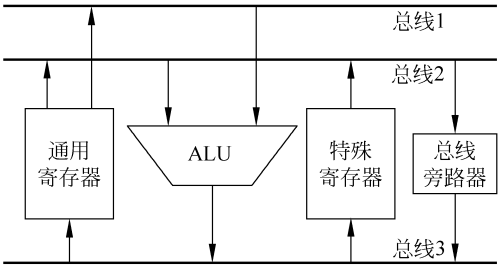
2. 运算器的总线结构

运算器的结构设计，主要是围绕 ALU 和寄存器同数据总线之间如何传送操作数和运算结果进行。组成运算器的各部件之间的连接一般也采用总线结构，这个总线称为 CPU 的内部总线，是 CPU 内部的数据通路。运算器内部总线大体有 3 种基本结构，如图 3-18 所示。

(1) 单总线结构的运算器。单总线结构的运算器如图 3-18(a)。由于所有部件都接到



(a) 单总线结构　　(b) 双总线结构



(c) 三总线结构

图 3-18　运算器的 3 种总线结构

同一总线上，所以数据可以在任何两个寄存器之间，寄存器和 ALU 之间传送。由于同一时间内只能有一个操作数放在单总线上，需要设置 A、B 两个锁存器，两个操作数分两步进入锁存器，然后由 ALU 进行运算，存储运算结果需要在第 3 个时段进行。

这种结构的主要缺点是操作速度较慢。但是由于它只需控制一条总线，控制电路比较简单。

(2) 双总线结构的运算器。双总线结构的运算器如图 3-18(b)所示。两个数据可以分别通过总线 1、总线 2 同时送至 ALU 的两个输入端，由 ALU 的组合逻辑进行运算。由于 ALU 形成操作结果输出时，两条总线都被输入数据占用，ALU 的处理结果不能直接加到总线上去，所以 ALU 输出端需设置缓冲寄存器。假如在总线 1、2 和 ALU 输入端之间各有一个输入缓冲寄存器，可以把两个输入数先放至这两个缓冲寄存器中，此时 ALU 就可以直接把运算结果送至总线 1 或总线 2 上去。总之，双总线结构完成一次运算的操作比单总线结构减少一步。

双总线结构的特殊寄存器分为两组，它们分别与一条总线交换数据。这样，通用寄存器中的数就可进入到任一组特殊寄存器中去，从而使数据传送更为灵活。

(3) 三总线结构的运算器。三总线结构的运算器如图 3-18(c)所示。三总线结构中，ALU 的两个输入端分别由两条总线提供操作数，而 ALU 的输出则与第 3 条总线相连。这样，算术逻辑操作就可以在一步的控制内完成。由于 ALU 本身有时间延迟，所以打入输出结果的选通脉冲必须考虑到这个延迟。如果某一个操作数不需要运算和修改，不必借助于 ALU，可以通过总线旁路器，把数据从总线 2 传送到总线 3。显然，三总线结构的运算器的特点是操作时间快，但控制较前两种复杂。

3.6.2 多功能算术逻辑运算单元实例

构成运算器的核心部件是算术逻辑运算单元(ALU),它既可以完成算术运算,又可以完成逻辑运算。根据前述加、减、乘、除运算方法的介绍,4 种运算的核心可以归结为加法运算。因此,ALU 的核心是并行加法器。一般的加法器只能完成加、减运算,不能同时完成"与""或""异或"等逻辑运算。如果在普通的加法电路前端增加一级逻辑电路,可以实现多种算术和逻辑运算功能。Intel 74181 就是一种典型的多功能算术逻辑运算单元。

1. 74181 的基本功能及引脚

74181 是一个 4 位多功能算术、逻辑运算单元,可以实现 16 种算术运算和 16 种逻辑运算,其加法器采用先行进位方式。

它可以工作在正逻辑或负逻辑状态,其对应正逻辑的功能引脚如图 3-19 所示。

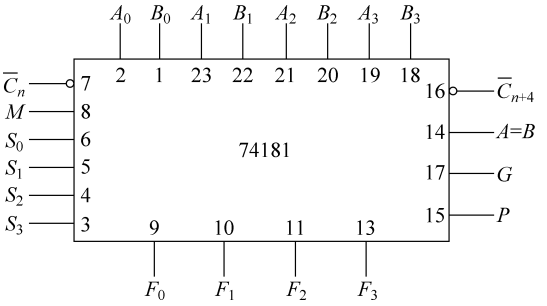


图 3-19 74181ALU 芯片的引脚

(1) $A_3 \sim A_0$、$B_3 \sim B_0$ 分别是参加运算的两个 4 位二进制操作数,$F_3 \sim F_0$ 是运算结果的输出,C_n 是最低位的外来进位,C_{n+4} 是 4 位运算向高位的进位。

(2) M 表示工作方式:$M=0$ 进行算术运算,$M=1$ 进行逻辑运算。

(3) $S_3 \sim S_0$ 用来选择 16 种运算中的某一种。

(4) G 为组进位产生函数,P 为组进位传递函数,是为多级并行进位准备的。详细内容见本节后续介绍。

其基本运算功能如表 3-2 所示。

表 3-2 74181ALU 算术/逻辑运算单元的功能

工作方式输入选择				正逻辑输入与输出	
S_3	S_2	S_1	S_0	逻辑 $M=1$	算术运算 $M=0$ $C_n=1$
0	0	0	0	$\overline{A}$	A
0	0	0	1	$\overline{A+B}$	$A+B$
0	0	1	0	$\overline{A}B$	$A+\overline{B}$
0	0	1	1	逻辑 0	-1
0	1	0	0	$\overline{AB}$	$A+A\overline{B}$
0	1	0	1	$\overline{B}$	$(A+B)+A\overline{B}$
0	1	1	0	$A \oplus B$	$A-B-1$

续表

工作方式输入选择				正逻辑输入与输出	
S_3	S_2	S_1	S_0	逻辑 $M=1$	算术运算 $M=0$ $C_n=1$
0	1	1	1	$A\overline{B}$	$A\overline{B}-1$
1	0	0	0	$\overline{A}+B$	$A+AB$
1	0	0	1	$\overline{A\oplus B}$	$A+B$
1	0	1	0	B	$(A+\overline{B})+AB$
1	0	1	1	AB	$AB-1$
1	1	0	0	逻辑 1	$A+A$
1	1	0	1	$A+\overline{B}$	$(A+B)+A$
1	1	1	0	$A+B$	$(A+\overline{B})+A$
1	1	1	1	A	$A-1$

2. 74181 逻辑实现

(1) 多功能逻辑。为了实现其多功能,在基本加法器的数据输入端增加了多功能函数发生器。其单元电路结构框图如图 3-20 所示。

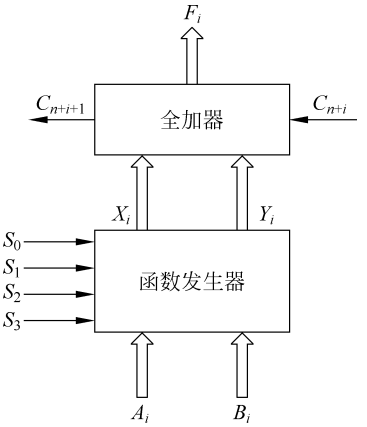


图 3-20　74181 的基本单元电路

图 3-20 中,功能选择控制端 S_3、S_2、S_1、S_0 控制输入端 A_i、B_i,产生不同的 X_i、Y_i。X_i 受 S_1、S_0 控制,Y_i 受 S_3、S_2 控制,其组合函数关系如表 3-3 所示。

表 3-3　X_i、Y_i 与选择控制端和输入端之间的关系

S_0　S_1	Y_i	S_2　S_3	X_i
0　0	$\overline{A}_i$	0　0	1
0　1	$\overline{A}_iB_i$	0　1	$\overline{A}_i+\overline{B}_i$
1　0	$\overline{A}_i\overline{B}_i$	1　0	$\overline{A}_i+B_i$
1　1	0	1　1	$\overline{A}_j$

则

$$X_i=\overline{S_3A_iB_i+S_2A_i\overline{B}_i}$$

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

这样，全法器的和 $F_i = X_i \oplus Y_i \oplus C_{n+i}$ 是受 S_3、S_2、S_1、S_0 控制下的 A_i、B_i 的函数，S_3、S_2、S_1、S_0 控制端的状态不同，得到的 F_i 不同。

(2) 先行进位。对于每一位全加器，将上述 X_i、Y_i 的表达式带入 $C_{n+i+1} = X_i Y_i + Y_i C_{n+i} + X_i C_{n+i}$，化简后可以得到：

$$C_{n+i+1} = Y_i + X_i C_{n+i}, \quad i = 0,1,2,3$$

其每一位的进位表达式：

$$C_{n+1} = Y_0 + X_0 C_n$$

$$C_{n+2} = Y_1 + X_1 C_{n+1} = Y_1 + Y_0 X_1 C_n$$

$$C_{n+3} = Y_2 + X_2 C_{n+2} = Y_2 + Y_1 X_2 + Y_0 X_1 X_2 + X_0 X_1 X_2 C_n$$

$$C_{n+4} = Y_3 + X_3 C_{n+3} = Y_3 + Y_2 X_3 + Y_1 X_2 X_3 + Y_0 X_1 X_2 X_3 + X_0 X_1 X_2 X_3 C_n$$

74181 的加法器是根据上述 4 个表达式的“与”“或”逻辑实现的，即采用的是先行进位的方式。其内部逻辑电路如图 3-21 所示。

若设

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

$$P = X_0 X_1 X_2 X_3$$

则

$$C_{n+4} = G + PC_n$$

74181ALU 芯片中，G、P 作为组进位传递信号引出，以便于实现多组 ALU 之间的先行进位。

3. 74182CLA 及多级先行进位实现

(1) 74182CLA。74181 ALU 只能实现 4 位二进制的运算，如果要实现更多位的运算，需要多片 74181 进行级联。为了实现多片 74181 之间的先行进位，Intel 公司还提供了 74182 先行进位部件(CLA)。其内部逻辑电路如图 3-22 所示。

对于每一个 74181 来说，有 $C_{n+4} = G + PC_n$，其中 C_{n+4} 是 4 位运算最高位的进位，G、P 为组进位传递信号，它们只与自己的输入数据 X_i、Y_i($i = 0,1,2,3$)有关。假设 4 片 74181 的组进位传递信号分别为 P_3、G_3，P_2、G_2，P_1、G_1，P_0、G_0，每个低位 74181 向高位 74181 的进位信号分别为：

$$C_{n+x} = G_0 + P_0 C_n$$

$$C_{n+y} = G_1 + P_1 C_{n+x} = G_1 + G_0 P_1 + P_0 P_1 C_n$$

$$C_{n+z} = G_2 + P_2 C_{n+y} = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_0 P_1 P_2 C_n$$

由于 C_{n+x}、C_{n+y}、C_{n+z} 表达式中的信号 P_i、G_i 是 4 个 74181 同时输出的，因此 C_{n+x}、C_{n+y}、C_{n+z} 是同时输出的。

对于最高位 74181 向前的进位：

$$G_3 + P_3 C_{n+z} = G_3 + P_3 G_2 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_0 P_1 P_2 P_3 C_n = G^* + P^* C_n$$

可以将 G^*、P^* 引出，便于实现更高级的进位传输。

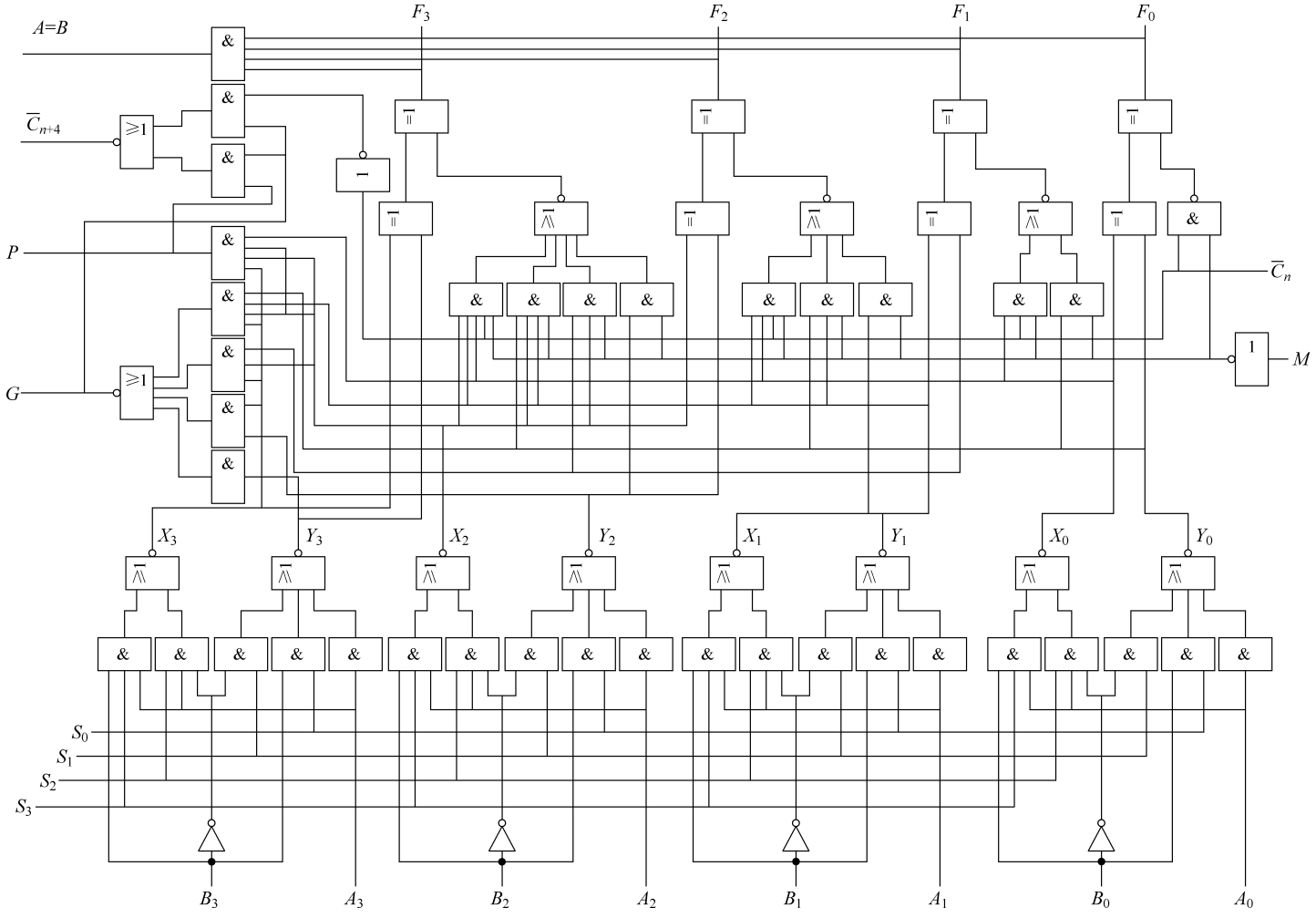


图 3-21　74181ALU 内部逻辑电路(正逻辑)

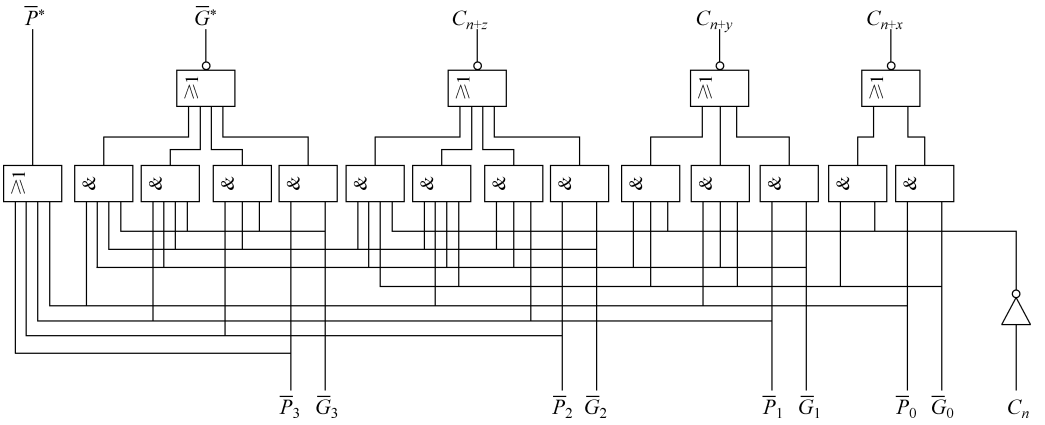


图 3-22　74182CLA 的逻辑电路

（2）多级先行进位 ALU。采用 4 片 74181 和一片 74182 可以组成两级 16 位先行进位的 ALU。用两个 16 位先行进位的 ALU 级联可以组成的 32 位 ALU，其逻辑如图 3-23 所示。

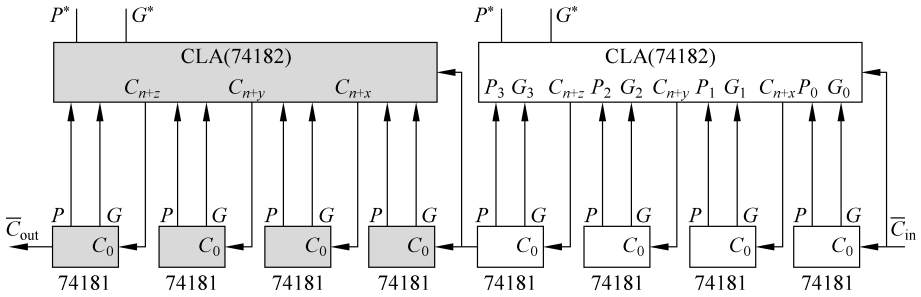


图 3-23　32 位两级先行进位 ALU

74181 和 74182 结合可以组成不同位数和结构的 ALU。位数不同、结构不同，运算所花费的时间不同。表 3-4 列出了多种位数和结构的 ALU 的芯片数目与对应的运算时间。

表 3-4　不同位数和结构的 ALU 运算时间

位数	ALU 结构	总的加法时间/ns	芯片数量	
			74181	74182
4	一级 ALU	21	1	0
8	一级行波 ALU	36	2	0
16	一级行波 ALU	60	4	0
16	两级 ALU	36	4	1
32	一级行波 ALU	110	8	0
32	两级行波 ALU	62	8	2
48	一级行波 ALU	160	12	0
48	两级行波 ALU	101	12	3

续表

位数	ALU 结构	总的加法时间/ns	芯片数量	
			74181	74182
48	三级 ALU	64	12	4
64	一级行波 ALU	210	16	0
64	两级行波 ALU	136	16	4
64	三级 ALU	88	16	5

3.6.3 浮点运算器示例

浮点运算的实现，可以采用专门的浮点运算部件，或者与定点运算一起使用一套运算器。早期的微型计算机系统中浮点运算往往独立于定点运算器，由专门的部件实现。

根据浮点四则运算的规则，浮点运算器通常包括阶码运算器和尾数运算器两大部分。阶码运算器是一个定点运算器，能完成加减运算时的阶码比较，乘除运算时的阶码加减，及规格化时的阶码加 1 或减 1 操作，结构相对简单。尾数运算器是一个定点小数运算器，能完成加减运算时的尾数求和，以及乘除运算时尾数的乘除，另外还应该有快速移位功能，结构相对复杂一点。

1. 80x87 协处理器

80x87 是 Intel 公司为处理浮点数据的算术运算和多种函数计算而设计生产的专用算术处理器。由于它们的算术运算是配合 80x86 进行的，所以又称为协处理器。486SX 以下的微型计算机，80x87 是任选件；而 486DX 及其以上的微型计算机，80x87 已被集成在 CPU 芯片之中。

(1) 80x87 协处理器与 80x86 之间以异步方式进行工作。80x87 相当于 80x86 的一个 I/O 部件，虽然它有自己的指令，但不能作为独立的 CPU 使用，因为写主存的操作是 80x86 完成的。如果 80x86 从内存读取的指令是 80x87 的浮点运算指令，就以输出方式把该指令送到 80x87，80x87 接收后进行译码并执行浮点运算。在 80x87 运算期间，80x86 可取下一条指令予以执行，因而实现了并行工作。如果 80x87 执行浮点运算指令期间 80x86 又取一条 80x87 指令，80x87 给出“忙”的标志信号加以拒绝。

(2) 80x87 协处理器的数据格式。80x87 可处理包括二进制浮点数、二进制整数和压缩十进制数串三大类 7 种不同的数据类型，这些数据类型的格式如图 3-24 所示。

其中，整数最高位为符号位，采用补码表示，有 16、32 和 64 位 3 种格式。浮点数的格式符合 IEEE 754 标准。压缩的十进制数串是特殊形式的整数，80 位的低 72 位表示 18 位十进制数，最高位为符号位。

(3) 80x87 的内部结构。80x87 的内部结构如图 3-25 所示。它不仅仅是一个浮点运算器，还包括了执行数据运算所需的全部控制线路。由总线控制逻辑部件、数据接口与控制部件、浮点运算部件 3 个主要功能模块组成。

在浮点运算部件中，分别设置了阶码(指数)运算部件和尾数运算部件。还有加速移位操作的移位器。它们通过指数总线和尾数总线与 8 个 80 位字长的寄存器相连。这些寄存器按“先进后出”的栈操作方式工作，也可以按寄存器的编号访问某一个寄存器。

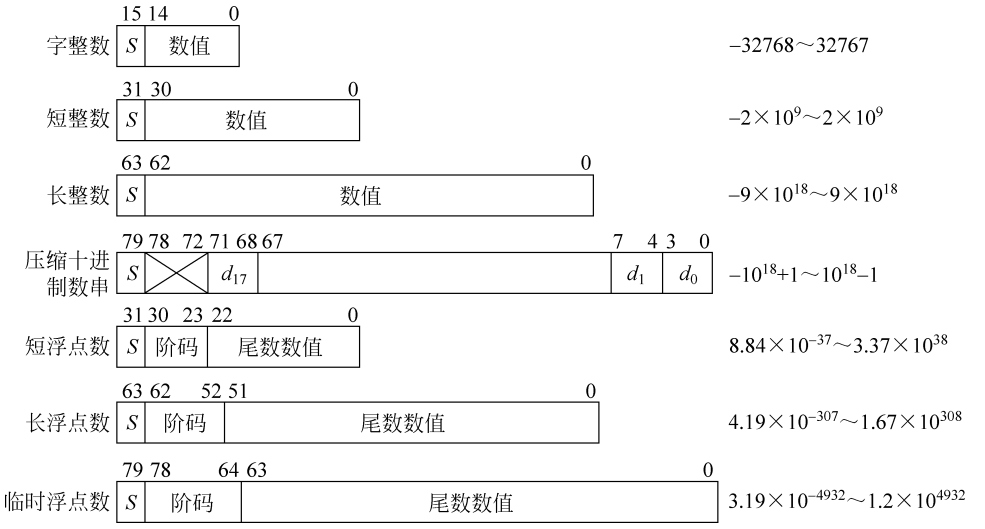


图 3-24　80x87 的数据格式

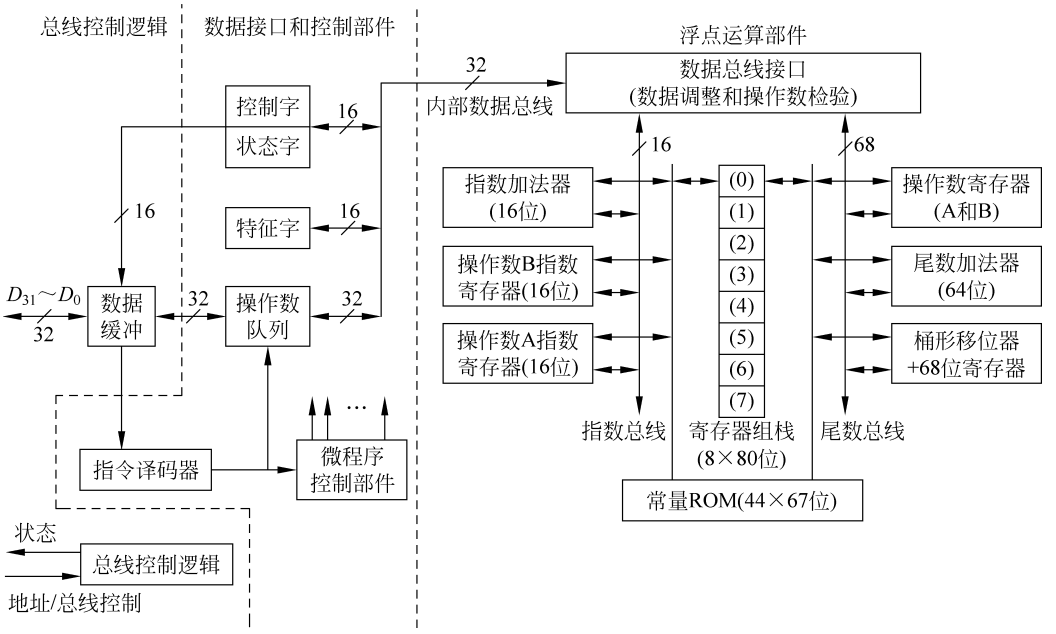


图 3-25　80x87 浮点运算器的内部结构

全部数据在 80x87 中均以 80 位的临时浮点数的形式表示。80x87 从主存取数或向主存写数时，用 80 位的临时浮点数与其他数据类型自动转换。

2. 奔腾 CPU 的浮点运算器

奔腾 CPU 的浮点运算器包含在芯片内，CPU 的整体结构如图 3-26 所示。浮点运算部件采用流水线设计，内有专用的加法器、乘法器和除法器，有 8 个 80 位寄存器组成的寄存器堆，内部数据总线为 80 位宽。浮点部件可以支持 IEEE 754 标准的单精度和双精度格式的浮点数，也使用临时实数的 80 位浮点数。对于浮点数的取整、加法、乘法等操作，采用了新的算法，并用硬件来实现，其执行速度约为 80486 的 10 倍。

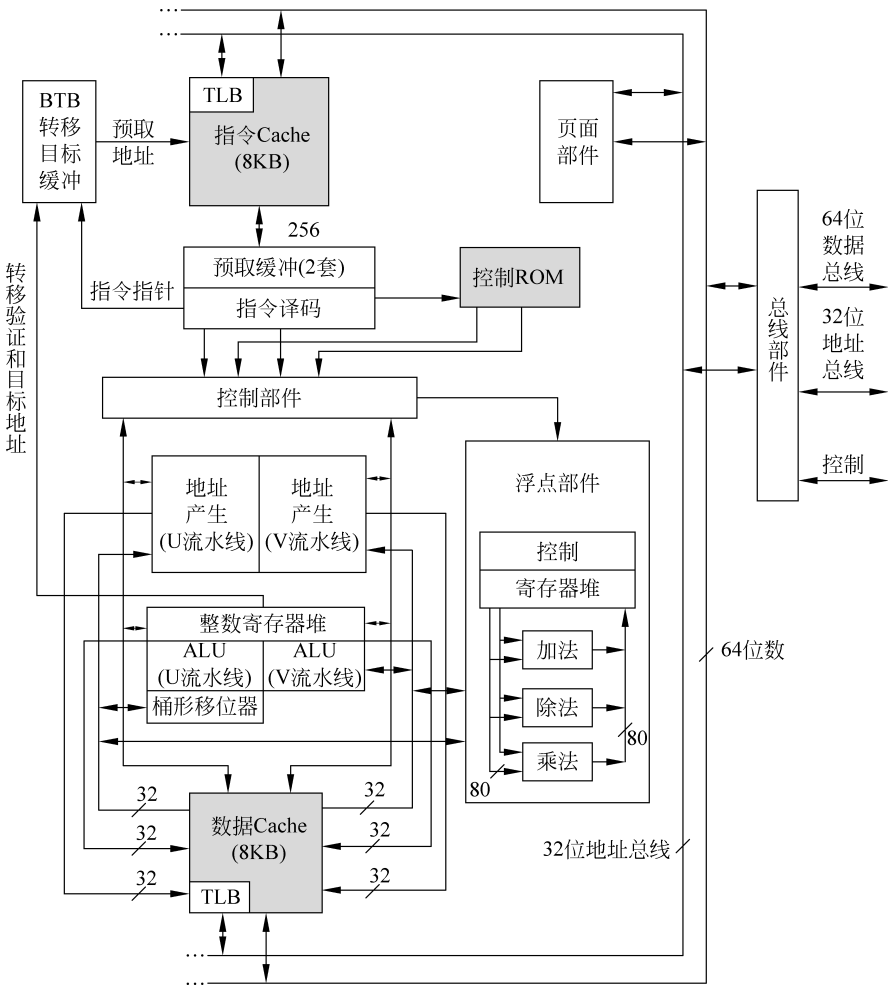


图 3-26　Pentium CPU 的内部结构

本 章 小 结

运算器是计算机处理数据的功能部件，它可以进行算术运算和逻辑运算，其核心是 ALU。运算器的逻辑结构与机器指令系统的运算功能，数据的表示形式和运算方法等有关。

定点加减运算通常采用补码实现。补码的加减运算规则使得计算机中的减法转化为加法实现。运算过程中符号位可以当做二进制数与数值部分一起进行运算。大大地简化了硬件的设计。

定点数的乘法运算可以采用原码或补码实现，具体有原码一位乘法、补码一位乘法（Booth 算法）等串行乘法算法。定点数的除法运算也可以采用原码和补码实现，具体有恢复余数法和不恢复余数（加减交替）法等除法算法。它们的逻辑电路都是在基本的加减法器的基础上增加一些左右移位和控制电路实现，运算速度较慢。为了提高运算方法，常采用阵列乘法器和阵列乘法器。

浮点数的阶码和尾数均是定点数，浮点运算可以在定点运算器的基础上实现。一般，浮点运算器由阶码运算部件和尾数运算部件两部分构成。

无论是定点加减运算还是乘除运算，其核心部件都是基本的二进制加减法器。为了提高运算速度，在加减运算器中采用了先行进位的措施；对于复杂的浮点加减、乘除运算，引入流水线技术。

习 题 3

一、基础题

1. 运算器的主要功能是进行(　　)。

A. 逻辑运算　　B. 算术运算

C. 逻辑运算和算术运算　　D. 只做加法算术/逻辑运算

2. 运算器虽由许多部件组成，但核心部件是(　　)。

A. 算术逻辑运算单元　　B. 多路开关

C. 数据总线　　D. 累加寄存器

3. 大部分计算机内的减法是通过(　　)实现。

A. 将被减数加到减数中　　B. 从被减数中减去减数

C. 补数相加　　D. 从减数中减去被减数

4. 补码加/减法是指(　　)。

A. 操作数用补码表示，两尾数相加/减，符号位单独处理

B. 操作数用补码表示，符号位和尾数一起参加运算，结果的符号与加/减数相同

C. 操作数用补码表示，符号位一起参与运算，减某数用加某数的机器负数代替，结果的符号在运算中形成

D. 操作数用补码表示，由数符决定两尾数的操作，符号位单独处理

5. 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

6. 当定点运算发生溢出时，应(　　)。

A. 向左规格化　　B. 向右规格化

C. 发出出错信息　　D. 舍入处理

7. 加法器采用先行进位的目的是(　　)。

A. 优化加法器的结构　　B. 节省器材

C. 加速传递进位信号　　D. 实现减法运算

8. 设机器字长8位(含1位符号位)，若机器数DAH为补码，则算术左移一位的结果为(　　)，算术右移一位的结果为(　　)。

A. F4H，6DH　　B. B4H，EDH

C. B5H，EDH　　D. B4H，6DH

9. 原码乘法是(　　)。

A. 用原码表示操作数，数值部分直接相乘，符号位单独处理

B. 用原码表示操作数，然后两数直接相乘(包括符号位)

C. 被乘数用原码表示，乘数取绝对值，然后相乘

D. 乘数用原码表示，被乘数取绝对值，然后相乘

10. 浮点加减中对阶的原则是(　　)。

A. 将较小的一个阶码调整到与较大的一个阶码相同

B. 将较大的一个阶码调整到与较小的一个阶码相同

C. 将被加数的阶码调整到与加数的阶码相同

D. 将加数的阶码调整到与被加数的阶码相同

11. 设浮点数的尾数为纯小数，用原码表示，最高位为符号位，则(　　)是规格化的数。

A. 1.101101　　B. 0.001101　　C. 1.011011　　D. 0.000110

12. 定点运算器用来进行(　　)。

A. 十进制数加法运算

B. 定点数运算

C. 浮点数运算

D. 既进行定点数运算也进行浮点数运算

13. 74181ALU 可具有(　　)。

A. 16 种算术运算功能

B. 16 种算术运算功能和 16 种逻辑运算功能

C. 16 种逻辑运算功能

D. 4 位乘法运算和除法运算功能

14. 4 片 74181 和 1 片 74182 组成 16 位 ALU，其进位信号传递是(　　)。

A. 组内、组间均为串行进位　　B. 组内并行进位，组间串行进位

C. 组内串行进位，组间并行进位　　D. 组内、组间均为并行进位

15. 已知 x 和 y，试用变形补码计算 $x+y$，并指出结果是否溢出。假设机器字长为 6 位。

(1) $x=11011, y=10101$。

(2) $x=10110, y=01010$。

(3) $x=10100, y=01001$。

16. 已知 x 和 y，试用变形补码计算 $x-y$，并指出结果是否溢出。假设机器字长为 6 位。

(1) $x=11011, y=-10101$。

(2) $x=-10110, y=01010$。

(3) $x=-10100, y=-01001$。

17. 设某加法器的输入数据为 $A=A_3\ A_2\ A_1\ A_0$ 和 $B=B_3\ B_2\ B_1\ B_0$，进位链小组信号为 $C_4\ C_3\ C_2\ C_1$，低位来的进位信号为 C_0，请分别按下述两种方式写出 $C_4\ C_3\ C_2\ C_1$ 的逻辑表达式。

(1) 串行进位方式。

(2) 并行进位方式。

18. 已知 $x=1011, y=-0101$，求 $[x]_{补}$、$[x/2]_{补}$、$[x/4]_{补}$、$[-x]_{补}$，$[y]_{补}$、$[y/2]_{补}$、$[y/4]_{补}$。假设机器字长为 5 位。

19. 设下列数据长 8 位，包括一位符号位，补码表示，分别写出每个数据右移或左移两

位后的结果，并给出真值验证移位和乘除运算间的关系。

(1) 01101100。

(2) 10010011。

(3) 11100110。

(4) 10001110。

20. 用原码一位乘法计算 $x \times y$。

(1) $x=11011, y=-11111$。

(2) $x=-11010, y=-01110$。

21. 用原码加减交替法计算 x/y。

(1) $x=10101, y=-11011$。

(2) $x=-10101, y=11011$。

22. 设浮点数的阶码(包括阶符)占 4 位，尾数占 7 位(包括数符)，阶码和尾数均采用补码表示。试用浮点运算规则计算 $x+y$、$x-y$(要求写出详细运算步骤，并进行规格化)。

(1) $x=0.110101\times 2^{-001}, y=-0.010010\times 2^{001}$。

(2) $x=(-0.011010)\times 2^{-010}, y=0.100101\times 2^{-001}$。

23. 设浮点数的阶码 4 位(包含阶符 1 位)，尾数 7 位(包含数符 1 位)，尾数采用原码表示，阶码采用移码表示。试用浮点运算规则计算下列各题。

(1) $x=2^3\times(13/16), y=2^4\times(-7/16)$，求 $x\times y$。尾数乘法采用原码一位乘法。

(2) $x=2^3\times(-13/16), y=2^5\times(15/16)$，求 x/y。尾数除法采用加减交替法。

二、提高题

1. 在定点二进制运算器中，减法运算一般通过(　　)来实现。

A. 原码运算的二进制减法器　　B. 补码运算的二进制减法器

C. 补码运算的十进制加法器　　D. 补码运算的二进制加法器

2. 下列说法正确的是(　　)。

A. 采用变形补码进行加减运算可以避免溢出

B. 只有定点运算才有可能溢出，浮点运算不会溢出

C. 只有带符号数的运算才有可能溢出

D. 将两个同符号数相加有可能溢出

3. 在定点运算中产生溢出的原因是(　　)。

A. 运算过程中最高位产生了进位或借位

B. 参加运算的数据超出了机器所能的表示范围

C. 运算结果超出了机器所能表示的范围

D. 寄存器的位数太少，不得不舍弃最低有效位

4. 下溢是指(　　)。

A. 运算结果的绝对值小于机器所能表示的最小绝对值

B. 运算结果小于机器所能表示的最小负数

C. 运算结果小于机器所能表示的最小正数

D. 运算结果的最低有效位产生的错误

5. 在双符号位判断溢出的方案中，出现负溢出时，双符号位应当为(　　)。

A. 00　　B. 01　　C. 10　　D. 11

6. 采用规格化的浮点数是为了(　　)。

A. 增加数据的表示范围　　B. 方便浮点运算

C. 防止运算时数据溢出　　D. 提高数据的表示精度

7. 在串行进位的加法器中,影响加法器运算速度的关键因素是(　　)。

A. 门电路的级延迟　　B. 元器件速度

C. 进位传递延迟　　D. 各位加法器速度的不同

8. 用 8 片 74181 和两片 74182 可组成(　　)。

A. 组内并行进位、组间串行进位的 32 位 ALU

B. 二级先行进位结构的 32 位 ALU

C. 组内先行进位、组间先行进位的 16 位 ALU

D. 三级先行进位结构的 32 位 ALU

9. 下列逻辑部件中,不包括在运算器内的是(　　)。

A. 累加器　　B. 运算状态寄存器

C. ALU　　D. 指令寄存器

10. 下面关于浮点运算器的描述正确的是(　　)。

A. 浮点运算器可使用两个独立的定点运算部件实现

B. 阶码部件需要实现加、减、乘、除 4 种运算

C. 阶码部件只进行阶码相加、相减和比较操作

D. 尾数部件只进行乘法和除法运算

11. 分析图 3-2 完成加减法运算的过程,指出其时间延迟的主要原因,给出提高运算速度的改进措施。

12. 用补码乘法(Booth 算法)计算 $x \times y$。

(1) $x=11011, y=-11111$。

(2) $x=-11010, y=-01110$。

13. 已知 $x=0.10101, y=-0.11011$,用原码加减交替法计算 $[x/y]_{原}$,并还原成真值。

14. 用浮点运算规则计算 $x+y$、$x-y$。假设阶码 4 位(含 1 位阶符),采用移码表示;尾数 7 位,含 1 位数符,采用原码表示,整数部分隐含 1,即

$$X=(1.M)\times 2^{E-K}$$

要求写出详细运算步骤,并进行规格化。

(1) $x=0.110101\times 2^{-001}, y=-0.010010\times 2^{+001}$。

(2) $x=(-0.011010)\times 2^{-010}, y=0.100101\times 2^{-001}$。

15. 设浮点数的阶码 4 位(包括阶符),用移码表示;尾数 7 位(包括数符),用补码表示。试按浮点运算规则计算下列各式。

(1) $[2^3\times(13/16)]\times[2^4\times(-9/16)]$,尾数乘法采用 Booth 算法。

(2) $[2^{-2}\times(13/32)]\times[2^3\times(15/16)]$,尾数除法采用加减交替法。

16. 用 74181 和 74182 芯片构成一个 32 位的 ALU,采用多级分组并行进位,要求速度尽可能快。试画出相应的逻辑框图。

17. 设计一个余 3 编码的十进制加法器单元电路。