第 3 章 运算方法和运算部件

计算机完成的功能通过执行程序来实现，任何程序最终都要转换为机器指令。指令中包含的各种算术逻辑运算能直接在硬件上执行，执行这些运算的硬件称为运算部件。

本章首先分析高级语言和机器指令中涉及的各类运算，然后介绍这些运算用到的核心部件——算术逻辑单元(ALU)的组成与工作原理，在此基础上，对这些运算在计算机内部的实现算法和过程进行详细说明，最后介绍实现这些运算的运算部件。

3.1 高级语言和机器指令中的运算

计算机硬件的设计目标来源于软件需求，高级语言中用到的各种运算，通过编译成底层的算术运算指令和逻辑运算指令实现，这些底层运算指令能在机器硬件上直接被执行。因此，在介绍运算部件的设计之前，有必要先了解一下高级语言程序和机器指令所涉及的一些运算。所有高级语言的运算功能大同小异，一种语言能代表高级语言的总体情况，因此，本书以 C 语言中的运算为例进行说明。同样，指令中涉及的运算也可用某个特定的指令系统来说明，本书主要介绍 MIPS 指令系统中的运算。

3.1.1 C 语言程序中涉及的运算

加、减、乘、除等算术运算是高级语言中必须提供的基本运算。C 语言中除算术运算外，还有以下几类基本运算：按位、逻辑、移位、位扩展和位截断等运算。

1. 按位运算

C 语言中的按位运算包括按位或运算(|)、按位与运算(&)、按位取反运算(～)、按位异或运算(^)等。按位运算的一个重要运用就是实现掩码(masking)操作，通过与给定的一个位模式进行按位与，可以提取所需要的位，然后可以对这些位进行“置 1”“清 0”“1 测试”或“0 测试”等。这里位模式被称为掩码。例如，表达式 0x0F&0x8C 的运算结果为 00001100，即 0x0C。这里通过掩码 0x0F 提取了 0x8C 的低 4 位。

2. 逻辑运算

C 语言中的逻辑运算包括或运算(‖)、与运算(&&)、非运算(!)。这些逻辑运算很容易和按位运算混淆，事实上它们的功能完全不同。逻辑运算是非数值计算，其操作数只有两个逻辑值：True 和 False。通常用非 0 表示 True，全 0 表示 False。

3. 移位运算

C 语言中提供了一组移位运算，有逻辑移位和算术移位两种。

逻辑移位不考虑符号位，总是把高(低)位移出，低(高)位补 0。对于无符号整数应采用逻辑移位，逻辑左移时，若最高位移出的是 1，则发生溢出。

因为带符号整数都用补码表示，所以其移位操作应采用算术移位方式。左移时，高位移出，低位补 0，如果移出的高位不同于移位后的符号位，即左移前、后符号位不同，则发生溢出；右移时，低位移出，高位补符号。

虽然 C 语言没有明确规定应该采用逻辑移位还是算术移位。但是，实际上许多机器和编译器都对无符号整数采用逻辑移位方式，而对带符号整数采用算术移位方式。表达式 x<<k表示对数 x 左移 k 位。事实上，对于左移来说，逻辑移位和算术移位的结果都一样，都是丢弃 k 个最高位，并在低位补 k 个 0。表达式 x>>k 表示对数 x 右移 k 位。

每左移一位，相当于数值扩大一倍，所以左移可能会发生溢出，左移 k 位，相当于数值乘以 2^k；每右移一位，相当于数值缩小一半，右移 k 位，相当于数值除以 2^k。

4. 位扩展和位截断运算

C 语言中没有明确的位扩展运算符，但是在进行数据类型转换时，如果遇到一个短数向长数转换，就要进行位扩展运算了。进行位扩展时，扩展后的数值应保持不变。有两种位扩展方式：0 扩展和符号扩展。0 扩展用于无符号数，只要在短的无符号数前面添加足够的 0 即可。符号扩展用于补码表示的带符号整数。通过在短的带符号整数前添加足够多的符号位来扩展。

考虑以下 C 语言程序代码：

```
1  short si=-32768;
2  unsigned short usi=si;
3  int i=si;
4  unsigned ui=usi;
```

执行上述程序段，并在 32 位大端方式机器上输出变量 si、usi、i、ui 的十进制和十六进制值，可得到各变量的输出结果为：

```
si=-32768   80 00
usi=32768   80 00
i=-32768    FF FF 80 00
ui=32768    00 00 80 00
```

由此可见，－32768 的补码表示和 32768 的无符号整数表示具有相同的 16 位 0/1 序列，分别将它们扩展为 32 位后，得到的 32 位位序列的高位不同。因为前者是符号扩展，高 16 位补符号 1，后者是 0 扩展，高 16 位补 0。

位截断发生在将长数转换为短数时，例如，对于下列代码：

```
1  int i=32768;
2  short si=(short)i;
3  int j=si;
```

在一台 32 位大端方式机器上执行上述代码段时，第 2 行要求强行将一个 32 位带符号

整数截断为16位带符号整数,32768的32位补码表示为00 00 80 00H,截断为16位后变成80 00H,它是-32768的16位补码表示。再将该16位带符号整数扩展为32位时,就变成了FF FF 80 00H,它是-32768的32位补码表示,因此j为-32768。也就是说,原来的i(值为32768)经过截断、再扩展后,其值变成了-32768,不等于原来的值了。

从上述例子可以看出,截断一个数可能会因为溢出而改变它的值。因为长数的表示范围远远大于短数的表示范围,所以当一个长数足够大到短数无法表示的程度,则截断时就会发生溢出。上述例子中的32768大于16位补码能表示的最大数32767,所以就发生了截断错误。这里所说的截断溢出和截断错误只会导致程序出现意外的计算结果,并不导致任何异常或错误报告,因此,错误的隐蔽性很强,需要引起注意。

3.1.2 MIPS指令中涉及的运算

高级语言中的所有运算通过运算指令实现,一个指令系统中涉及运算的指令有很多,表3.1列出了MIPS指令系统中大部分涉及运算的指令。

表3.1 MIPS指令系统中涉及运算的部分指令

指令类型	指令名称	汇编形式举例	含义	所需运算
逻辑运算	and	and $1,$2,$3	$1=$2 & $3	按位与
	or	or $1,$2,$3	$1=$2|$3	按位或
	nor	nor $1,$2,$3	$1=~($2|$3)	按位或非
	and immediate	andi $1,$2,100	$1=$2&100	按位与
	or immediate	ori $1,$2,100	$1=$2|100	按位或
	shift left logical	sll $1,$2,10	$1=$2<<10	逻辑左移
	shift right logical	srl $1,$2,10	$1=$2>>10	逻辑右移
定点算术运算*	shift right arithmetic	sra $1,$2,10	$1=$2>>10	算术右移
	add	add $1,$2,$3	$1=$2+$3	整数加(判溢出)
	subtract	sub $1,$2,$3	$1=$2-$3	整数减(判溢出)
	add immediate	addi $1,$2,100	$1=$2+100	符号扩展、整数加(判溢出)
	sub immediate	subi $1,$2,100	$1=$2-100	符号扩展、整数减(判溢出)
	add unsigned	addu $1,$2,$3	$1=$2+$3	整数加(不判溢出)
	subtract unsigned	subu $1,$2,$3	$1=$2-$3	整数减(不判溢出)
	add immediate unsigned	addiu $1,$2,100	$1=$2+100	0扩展、整数加(不判溢出)
	multiply	mult $2,$3	Hi, Lo=$2×$3	带符号整数乘
	multiply unsigned	multu $2,$3	Hi, Lo=$2×$3	无符号整数乘
	divide	div $2,$3	Lo=$2÷$3 Hi=$2 mod $3	带符号整数除 Lo=商, Hi=余数
	divide unsigned	divu $2,$3	Lo=$2÷$3 Hi=$2 mod $3	无符号整数除 Lo=商, Hi=余数
定点数据传送	load word	lw $1,100($2)	$1=mem[$2+100]	符号扩展并整数加
	store word	sw $1,100($2)	mem[$2+100]=$1	符号扩展并整数加
	load half unsigned	lhu $1,100($2)	$1=mem[$2+100]	符号扩展并整数加,0扩展
	store half	sh $1,100($2)	mem[$2+100]=$1	符号扩展并整数加,符号扩展
	load byte unsigned	lbu $1,100($2)	$1=mem[$2+100]	符号扩展并整数加,0扩展
	store byte	sb $1,100($2)	mem[$2+100]=$1	符号扩展并整数加,符号扩展
	load upper immediate	lui $1,100	$1=100×$2^{16}$	逻辑左移16位

续表

指令类型	指令名称	汇编形式举例	含义	所需运算
浮点算术运算	FP add single	add.s $f2,$f4,$f6	$f2=$f4+$f6	单精度浮点加
	FP subtract single	sub.s $f2,$f4,$f6	$f2=$f4−$f6	单精度浮点减
	FP multiply single	mul.s $f2,$f4,$f6	$f2=$f4×$f6	单精度浮点乘
	FP divide single	div.s $f2,$f4,$f6	$f2=$f4÷$f6	单精度浮点除
	FP add double	add.d $f2,$f4,$f6	$f2=$f4+$f6	双精度浮点加
	FP subtract double	sub.d $f2,$f4,$f6	$f2=$f4−$f6	双精度浮点减
	FP multiply double	mul.d $f2,$f4,$f6	$f2=$f4×$f6	双精度浮点乘
	FP divide double	div.d $f2,$f4,$f6	$f2=$f4÷$f6	双精度浮点除
浮点数据传送	load word corp.1	lwcl $f1,100($2)	$f1=mem[$2+100]	符号扩展并整数加
	store word corp.1	swcl $f1,100($2)	mem[$2+100]=$f1	符号扩展并整数加

注*：add 和 addu 指令的执行结果相同，只是 add 指令需要检测是否发生溢出，而 addu 指令无须判断溢出。addi 和 addiu 除了在溢出判断上有差别外，立即数的扩展也有差别，addi 中的立即数被看成带符号整数，采用符号扩展，而 addiu 中的立即数被看成无符号整数，采用 0 扩展。减法的情况与加法一样，而乘、除法指令都不判断溢出（通过新增的伪指令来实现溢出判断）。

从表 3.1 可以看出，MIPS 指令系统涉及的运算有按位逻辑运算、逻辑移位、算术移位、带符号整数的加减乘除、无符号整数加减乘除、带符号整数的符号扩展、无符号数的 0 扩展、单精度浮点数加减乘除、双精度浮点数加减乘除等。MIPS 指令中没有专门的算术左移指令。因为对于左移来说，逻辑移位和算术移位的结果都一样，都是丢弃 k 个最高位，并在低位补 k 个 0，所以，带符号整数和无符号整数的左移都可用逻辑左移指令实现。很明显，利用 MIPS 提供的这些运算指令完全能够实现 C 语言所需要的各种运算要求。

3.2 基本运算部件

一般情况下，用一个专门的算术逻辑部件（arithmetic and logic unit，ALU）来完成基本逻辑运算和定点数加减运算，各类定点乘除运算和浮点数运算则可利用加法器或 ALU 和移位器来实现，因此基本的运算部件是加法器、ALU 和移位器，而 ALU 的核心部件是加法器。

3.2.1 全加器和加法器

全加器用来实现两个本位数加上低位进位生成一位本位和以及一位向高位的进位。第 i 位的加法运算是指第 i 位的加数 X_i、Y_i 和低位来的进位 C_{i-1} 三者相加，得到本位和 F_i 和第 i 位的进位输出 C_i。

F_i 和 C_i 的逻辑表达式如下：

$$\left.\begin{aligned} F_i &= X_i \oplus Y_i \oplus C_{i-1} \\ C_i &= X_i C_{i-1} + Y_i C_{i-1} + X_i Y_i \end{aligned}\right\} \tag{3-1}$$

式中，F_i、C_i 被分别称为“全加和”和“全加进位”。图 3.1 和图 3.2 分别是“全加和”和“全加进位”生成电路。从图 3.2 可看出，进位 C_{i-1} 到 C_i 的延迟是两级门。

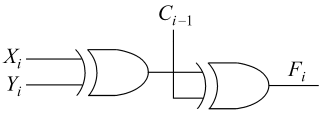


图 3.1　全加和 F_i 的生成

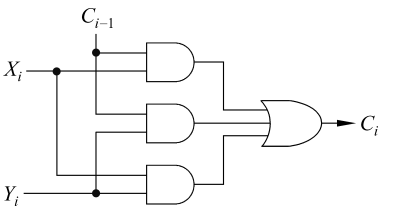


图 3.2　全加进位 C_i 的生成

图 3.3 是全加器的符号表示。将 n 个全加器相连可得 n 位加法器，如图 3.4 所示的加法器实现了两个 n 位二进制数 $X=X_nX_{n-1}\cdots X_1$ 和 $Y=Y_nY_{n-1}\cdots Y_1$ 逐位相加的功能，得到的二进制和为 $F=F_nF_{n-1}\cdots F_1$，进位输出为 C_n。例如，当 $X=11\cdots11$，$Y=00\cdots01$ 时，最后的输出为 $F=00\cdots00$ 且 $C_n=1$。由于只有有限位数，高位自动丢失，所以实际上是在模 2^n 运算系统下的加法运算，可以实现 n 位无符号整数加法和 n 位补码加法。

对于图 3.4 所示的 n 位加法器，X 与 Y 逐位相加，位间进位串行传送，因此称为串行进位方式。众所周知，一块小石头扔进平静的水中，泛起的波纹会向外一圈一圈逐步扩散，串行进位加法器中最低进位 C_0 就像一块小石头，它把进位逐步从低位扩展到最高位，所以，这种串行进位加法器也被称为行波进位加法器(carry ripple adder，CRA)。

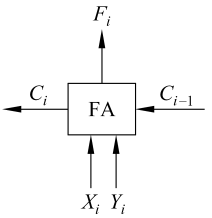


图 3.3　全加器符号

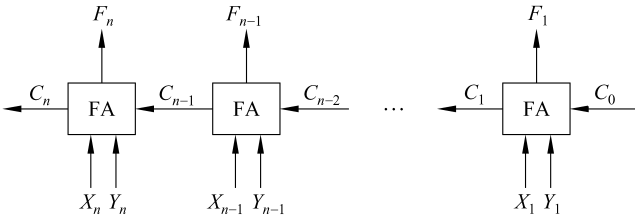


图 3.4　n 位串行进位加法器

这种结构所用元件较少，但进位传递时间较长。从图 3.2 可以看出，全加器内从进位输入到进位输出，共经过 2 级门延迟，所以，n 位串行加法器从 C_0 到 C_n 的延迟时间为 $2n$ 级门延迟。假定一个异或门为 3 级门延迟，则最后一位和数 F_n 的延迟时间为 $2n+1$ 级门延迟。加法运算时间随两个加数位数 n 的增加而增加。当 n 较大时，串行进位的加法器速度将显著变慢。

前面说过，几乎所有的算术运算都要用到 ALU 或加法器，ALU 的核心还是加法器，因此要提高运算速度，加法器的速度非常关键。由于串行进位加法器速度慢的主要原因是进位按串行方式传递，高位进位依赖低位进位。为了提高加法器的速度，必须尽量避免进位之间的依赖关系。

3.2.2　并行进位加法器

由全加器公式(3-1)可知，对于一个 4 位加法器，其进位 C_1、C_2、C_3 和 C_4 的产生条件为

$$C_1=X_1Y_1+(X_1+Y_1)C_0$$

$$C_2=X_2Y_2+(X_2+Y_2)C_1$$

$$=X_2Y_2+(X_2+Y_2)X_1Y_1+(X_2+Y_2)(X_1+Y_1)C_0$$

$$
\begin{aligned}
C_3 &= X_3Y_3 + (X_3 + Y_3)C_2 \\
&= X_3Y_3 + (X_3 + Y_3)[X_2Y_2 + (X_2 + Y_2)X_1Y_1 + (X_2 + Y_2)(X_1 + Y_1)C_0] \\
&= X_3Y_3 + (X_3 + Y_3)X_2Y_2 + (X_3 + Y_3)(X_2 + Y_2)X_1Y_1 + \\
&\quad (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0 \\
C_4 &= X_4Y_4 + (X_4 + Y_4)C_3 \\
&= X_4Y_4 + (X_4 + Y_4)[X_3Y_3 + (X_3 + Y_3)X_2Y_2 + (X_3 + Y_3)(X_2 + Y_2)X_1Y_1 + \\
&\quad (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0] \\
&= X_4Y_4 + (X_4 + Y_4)X_3Y_3 + (X_4 + Y_4)(X_3 + Y_3)X_2Y_2 + (X_4 + Y_4)(X_3 + Y_3) \\
&\quad (X_2 + Y_2)X_1Y_1 + (X_4 + Y_4)(X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0
\end{aligned}
$$

从以上公式来看，每个进位表达式中都含有$(X_i + Y_i)$和X_iY_i，所以，定义两个辅助函数如下：

$$
\left.
\begin{aligned}
P_i &= X_i + Y_i \\
G_i &= X_iY_i
\end{aligned}
\right\} \tag{3-2}
$$

P_i称为进位传递函数，其含义是：当X_i、Y_i中有一个为1时，若有低位进位输入，则一定被传递到高位。这个进位可看作低位进位越过本位直接向高位传递。G_i称为进位生成函数，其含义是：当X_i、Y_i均为1时，不管有无低位进位输入，本位一定向高位产生进位输出。

将P_i、G_i代入前面$C_1 \sim C_4$式中，可得：

$$
\left.
\begin{aligned}
C_1 &= G_1 + P_1C_0 \\
C_2 &= G_2 + P_2G_1 + P_2P_1C_0 \\
C_3 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0 \\
C_4 &= G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0
\end{aligned}
\right\} \tag{3-3}
$$

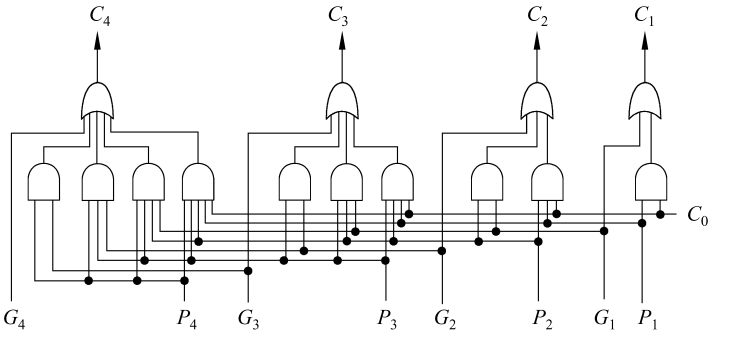
从上述表达式可以看出，C_i仅与X_i、Y_i和C_0有关，相互间的进位没有依赖关系。只要$X_1 \sim X_4$、$Y_1 \sim Y_4$和C_0同时到达，就可几乎同时形成$C_1 \sim C_4$，并同时生成各位的和。

实现上述逻辑表达式(3-3)的电路称为先行进位部件(carry lookahead unit，CLU)。通过这种进位方式实现的加法器称为全先行进位加法器(carry lookahead adder，CLA)。因为各个进位是并行产生的，所以是一种并行进位加法器。图3.5为4位CLU和4位CLA示意图。

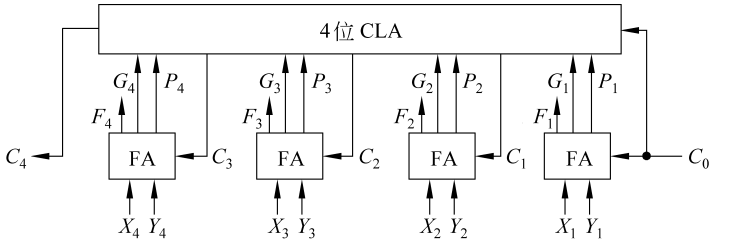
由图3.5可看出，从X_i、Y_i到产生P_i、G_i需要1级门延迟，从P_i、G_i、C_0到产生所有进位$C_1 \sim C_4$需要2级门延迟，产生全部和需要1+2+3=6级门延迟(假定一个异或门等于3级门延迟)。所以4位全先行进位加法器的关键路径长度为6级门延迟。

从式(3-3)可知，更多位数的CLA只会增加逻辑门的输入端个数，而不会增加门的级数，因此，如果用全先行进位方式构建更多位数加法器，从理论上讲，应该还是6级门延迟。但是由于CLA部件中连线数量和输入端个数的增多，使得实现电路中需要具有大驱动信号和大扇入门。因而，当位数较多时，全先行进位实现方式不太现实。例如，对于一个32位全先行进位加法器，其生成C_{32}的与门和或门有多达30多个输入端。

更多位数的加法器可通过将CLU或全先行进位加法器串接起来实现，例如，对于16位加法器，可以分成4位一组，组内为4位先行进位，组间串行进位。为了进一步提高加法器的运算速度，也可以进一步采用组内和组间都并行的进位方式。因为两级先行进位加法器组内和组间都采用先行进位方式，其延迟和加法器的位数没有关系，不会随着位数的增加而



(a) 4位 CLU



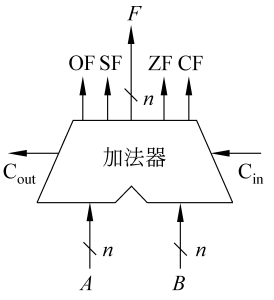
(b) 4位 CLA

图 3.5　4 位 CLU 和 4 位 CLA

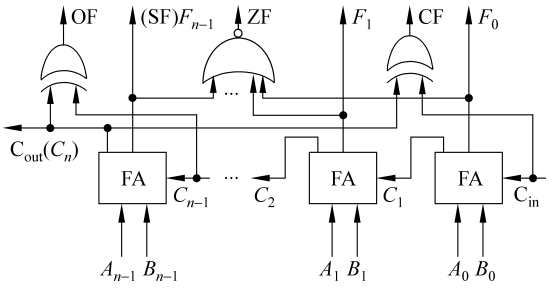
延长时间。所以,计算机内部大多采用两级或多级先行进位加法器。

3.2.3　带标志加法器

n 位无符号数加法器只能用于两个 n 位二进制数相加,不能进行无符号整数的减运算,也不能进行带符号整数的加/减运算。要能够进行无符号整数的加/减运算和带符号整数的加/减运算,还需要在无符号数加法器的基础上增加相应的逻辑门电路,使得加法器不仅能计算和/差,还要能够生成相应的标志信息。图 3.6 是带标志加法器实现电路示意图,其中图 3.6(a)中是符号表示,图 3.6(b)中给出用全加器构成的实现电路。



(a) 带标志加法器符号



(b) 带标志加法器的逻辑电路

图 3.6　用全加器实现 n 位带标志加法器的电路

如图 3.6 所示,溢出标志的逻辑表达式为 $OF=C_n \oplus C_{n-1}$;符号标志就是和的符号,即 $SF=F_{n-1}$;零标志 ZF=1,当且仅当 $F=0$;进位/借位标志 $CF=C_{out} \oplus C_{in}$,即当 $C_{in}=0$ 时,CF 为进位 C_{out},当 $C_{in}=1$ 时,CF 为进位 C_{out} 取反。

需要说明的是，为了加快加法运算的速度，真正的电路一定使用多级先行进位方式，图 3.6(b)主要是为了说明如何从加法运算结果中获得标志信息，因而使用全加器简化了加法器电路。

3.2.4 算术逻辑部件

ALU 是一种能进行多种算术运算与逻辑运算的组合逻辑电路，其核心部件是带标志加法器，多采用先行进位方式。通常用图 3.7 所示的符号来表示。其中 A 和 B 是两个 n 位操作数输入端，C_{in} 是进位输入端，ALUop 是操作控制端，用来决定 ALU 所执行的处理功能。例如，ALUop 选择 add 运算，ALU 就执行加法运算，输出的结果就是 A 加 B 之和。ALUop 的位数决定了操作的种类，例如，当位数为 3 时，ALU 最多只有 8 种操作。

图 3.8 给出了能够完成 3 种运算“与”“或”和“加法”的一位 ALU 结构图。其中，一位加法用一个全加器实现，在 ALUop 的控制下，由一个多路选择器(MUX)选择输出 3 种操作结果之一。这里有 3 种操作，所以 ALUop 至少要有两位。

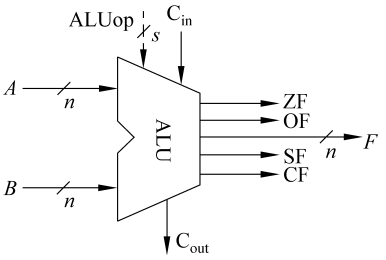


图 3.7 ALU 符号

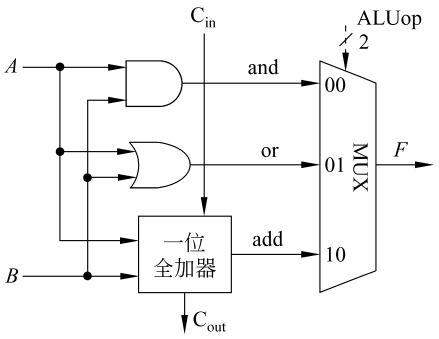


图 3.8 一位 ALU 结构

ALU 中也可实现左(右)移一位和两位的操作，当然也可用一个移位寄存器实现移位。但这两种方式每次都只能固定移动一位或两位，有时移位指令要求一次移动若干位，对于这种一次左移或右移多位的操作，通常用一个在 ALU 之外的桶型移位器实现。桶形移位器不同于普通移位寄存器，它利用大量多路选择器来实现数据的快速移位，移位操作能够一次完成。在 ALU 外单独设置桶型移位器，还可简化 ALU 的控制逻辑，并实现移位和 ALU 运算的并行操作。

3.3 定点数运算

从 3.1 节介绍的有关高级语言和机器指令涉及的运算来看，定点运算主要包括按位逻辑运算、逻辑移位运算、无符号整数的位扩展和截断运算、无符号整数的加减乘除运算、带符号整数的算术移位运算、带符号整数的扩展和截断运算、带符号整数的加减乘除运算等。

按位逻辑运算可用逻辑门电路实现，无符号整数的逻辑移位运算可用专门的移位器实现，带符号整数的移位运算、无符号整数和带符号整数的位扩展运算和截断运算也可用简单电路较容易地实现。

因此，对于无符号整数和带符号整数的运算，主要是加、减、乘、除运算方法以及对应的

运算部件。计算机内部带符号整数用补码表示,所以带符号整数的运算就是补码运算。

浮点数由一个定点小数和一个定点整数表示,大多数机器采用 IEEE 754 标准来表示浮点数,IEEE 754 标准用定点原码小数表示尾数,因而浮点数运算涉及原码定点小数的加、减、乘、除运算。

3.3.1 补码加减运算

若两个补码表示的 n 位定点整数为$[x]_{补}=X_{n-1}X_{n-2}\cdots X_0$,$[y]_{补}=Y_{n-1}Y_{n-2}\cdots Y_0$,则$[x+y]_{补}$和$[x-y]_{补}$的运算表达式如下:

$$\left.\begin{aligned}[x+y]_{补}&=[x]_{补}+[y]_{补}\quad(\mathrm{mod}\ 2^n)\\ [x-y]_{补}&=[x]_{补}+[-y]_{补}\quad(\mathrm{mod}\ 2^n)\end{aligned}\right\}\tag{3-4}$$

式(3-4)的正确性可以从补码的编码规则得到证明。从式(3-4)中可看出,在补码表示方式下,无论 x、y 是正数还是负数,加、减运算统一采用加法来处理,而且$[x]_{补}$和$[y]_{补}$的符号位(最高有效位)可以和数值位一起参与运算,加、减运算结果的符号位也在求和运算中直接得出,这样,可以直接用 3.2 节介绍的加法器实现$[x]_{补}+[y]_{补}(\mathrm{mod}\ 2^n)$和$[x]_{补}+[-y]_{补}(\mathrm{mod}\ 2^n)$。最终运算结果的高位丢弃,保留低 n 位,相当于对和数取模 2^n。因此,实现减法的主要工作在于求$[-y]_{补}$。

根据第 2 章介绍的补码运算的特点可知,一个数的负数的补码可以由其补码"各位取反,末位加 1"得到。即已知一个数的补码表示为 Y,则这个数负数的补码为 $\overline{Y}+1$,因此,只要在原加法器的 Y 输入端,加 n 个反向器以实现各位取反的功能,然后加一个 2 选 1 多路选择器,用一个控制端 Sub 来控制选择将原码 Y 输入到加法器还是将 $\overline{Y}$ 输入到加法器,并将控制端 Sub 同时作为低位进位送到加法器,如图 3.9 所示。该电路可实现补码加减运算。当控制端 Sub 为 1 时,做减法,实现 $X+\overline{Y}+1=[x]_{补}+[-y]_{补}$;当控制端 Sub 为 0 时,做加法,实现 $X+Y=[x]_{补}+[y]_{补}$。

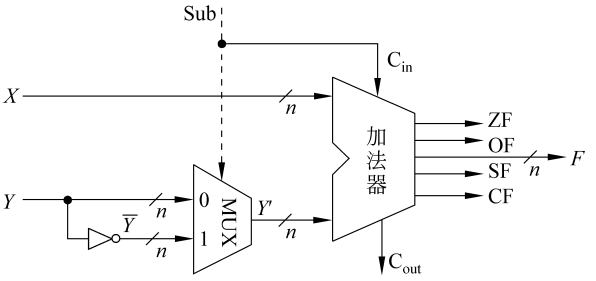


图 3.9 补码加减运算部件

图 3.9 所示电路可以实现整数加、减运算,其中的加法器就是图 3.6 所示的带标志加法器。因为无符号整数相当于正整数,而正整数的补码表示等于其二进制表示本身,所以,无符号整数的二进制表示相当于正整数的补码表示,因此,该电路同时也能实现无符号整数的加/减运算。对于带符号整数 x 和 y 来说,图中 X 和 Y 就是 x 和 y 的补码表示;对于无符号整数 x 和 y 来说,图中 X 和 Y 就是 x 和 y 的二进制表示。

可通过标志信息来区分带符号整数运算结果和无符号整数运算结果。

零标志 ZF=1 表示结果 F 为 0。不管作为无符号整数还是带符号整数来运算,ZF 都

有意义。

符号标志 SF 表示结果的符号，即 F 的最高位。对于无符号数运算，SF 没有意义。

进/借位标志 CF 表示无符号数加/减运算时的进/借位。加法时，若 CF＝1 则表示无符号数加法溢出；减法时，若 CF＝1 则表示有借位，即不够减。因此，加法时 CF 就应等于进位输出 C_{out}；减法时 CF 就应将进位输出 C_{out} 取反来作为借位标志。综合起来，可得 CF＝Sub⊕C_{out}。对于带符号整数运算，CF 没有意义。

溢出标志 OF＝1 表示带符号整数运算时结果发生了溢出。对于无符号整数运算，OF 没有意义。

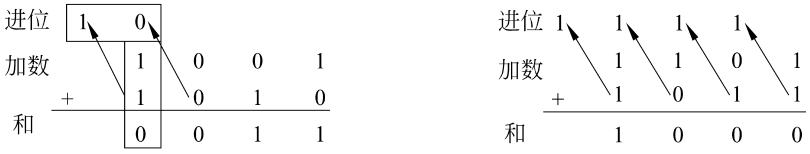
对于 n 位补码整数，它可表示的数值范围为 $-2^{n-1}\sim 2^{n-1}-1$。当运算结果超出该范围，则结果溢出。补码溢出判断方法有多种，先看两个例子。

例 3.1 用 4 位补码计算－7－6 和－3－5 的值。

解：$[-7]_{补}=1001$　$[-6]_{补}=1010$　$[-3]_{补}=1101$　$[-5]_{补}=1011$

$[-7-6]_{补}=[-7]_{补}+[-6]_{补}=1001+1010=0011(+3)$

$[-3-5]_{补}=[-3]_{补}+[-5]_{补}=1101+1011=1000(-8)$



因为 4 位补码的可表示范围为－8～＋7，而－7－6＝－13＜－8，所以，结果 0011(＋3)一定发生了溢出，是一个错误的值。考查－7－6 的例子后，发现以下两种现象：

(1) 最高位和次高位的进位不同；

(2) 和的符号位和加数的符号位不同。

对于例子－3－5，结果 1000(－8)没有超出范围，因而没有发生溢出，是一个正确的值。此时，最高位的进位和次高位的进位都是 1，没有发生第(1)种现象，而且和的符号和加数的符号都是 1，因而也没有发生第(2)种现象。

通常根据上述两种现象是否发生来判断有无溢出。因此，有以下两种溢出判断逻辑表达式。

(1) 若符号位产生的进位 C_n 与最高数值位向符号位的进位 C_{n-1} 不同，则产生溢出，即

$$OF=C_{n-1}\oplus C_n$$

(2) 若两个加数的符号位 X_{n-1} 和 Y_{n-1} 相同，且与和的符号位 F_{n-1} 不同，则产生溢出，即

$$OF=X_{n-1}Y_{n-1}\overline{F}_{n-1}+\overline{X}_{n-1}\overline{Y}_{n-1}F_{n-1}$$

根据上述溢出判断逻辑表达式，可以很容易实现溢出判断电路。图 3.6(b)中 OF 的生成采用了上述第(1)种方法。

*3.3.2 原码加减运算

浮点数多采用 IEEE 754 标准，其尾数用原码表示，故在浮点数加减运算中涉及原码加减运算。原码加减运算规则如下。

(1) 比较两个操作数的符号，对加法实行“同号求和，异号求差”，对减法实行“异号求

和,同号求差”。

(2) 求和时,数值位相加,若最高位产生进位则结果溢出。和的符号位取被加数(或被减数)的符号。

(3) 求差时,被加数(或被减数)数值位加上加数(或减数)数值位的补码。若最高数值位产生进位,则说明加法结果为正,所得数值位正确,差的符号位取被加数(被减数)的符号;若最高数值位没有产生进位,则说明加法结果为负,得到的是数值位的补码形式,需要对结果求补,还原为绝对值形式的数值位。

3.3.3 原码乘法运算

原码作为浮点数尾数的表示形式,需要计算机能实现定点原码小数的乘法运算。根据每次部分积是一位相乘得到还是两位相乘得到,可以有原码一位乘法和原码两位乘法,根据原码两位乘法的原理推广,可以有原码多位乘法。

1. 原码一位乘法

用原码实现乘法运算时,符号位与数值位分开计算,因此,原码乘法运算分为两步。

(1) 确定乘积的符号位。由两个乘数的符号异或得到。

(2) 计算乘积的数值位。乘积的数值部分为两个乘数的数值部分之积。

原码乘法算法描述如下:已知$[x]_{原}=X_0.X_1\cdots X_n$,$[y]_{原}=Y_0.Y_1\cdots Y_n$,则$[x\times y]_{原}=Z_0.Z_1\cdots Z_{2n}$,其中$Z_0=X_0\oplus Y_0$,$Z_1\cdots Z_{2n}=(0.X_1\cdots X_n)\times(0.Y_1\cdots Y_n)$。

可以不管小数点,事实上在机器内部也没有小数点,只是约定了一个小数点的位置,小数点约定在最左边就是定点小数乘法,约定在右边就是定点整数乘法。因此,两个定点小数的数值部分之积可以看成是两个无符号数的乘积。

下面是一个手算乘法的例子,以此可以推导出两个无符号数相乘的计算过程。

$$
\begin{array}{rl}
0.1\ 0\ 1\ 1 & \text{被乘数 } X=0.X_1X_2X_3X_4=0.1011\\
\times\ 0.1\ 1\ 0\ 1 & \text{乘数 } Y=0.Y_1Y_2Y_3Y_4=0.1101\\
\hline
\quad 1\ 0\ 1\ 1 & \cdots\cdots X\times Y_4\times 2^{-4}\\
\quad 0\ 0\ 0\ 0 & \cdots\cdots X\times Y_3\times 2^{-3}\\
\quad 1\ 0\ 1\ 1 & \cdots\cdots X\times Y_2\times 2^{-2}\\
1\ 0\ 1\ 1 & \cdots\cdots X\times Y_1\times 2^{-1}\\
\hline
0.1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 &
\end{array}
$$

由此可知,$X\times Y=\sum_{i=1}^{4}(X\times Y_i\times 2^{-i})=0.10001111$。

从上述手算乘法过程可以看出,两个无符号数相乘具有以下几个特点。

(1) 用乘数Y的每一位依次去乘以被乘数得$X\times Y_i$,$i=4,3,2,1$。若$Y_i=0$,则得0;若$Y_i=1$,则得X。

(2) 把(1)中求得的各项结果$X\times Y_i$在空间上向左错位排列,即逐次左移,可以表示为$X\times Y_i\times 2^{-i}$。

(3) 对(2)中求得的结果求和,这就是两个无符号数的乘积。

计算机中两个无符号数相乘,类似于手算乘法。但为了提高效率,作了相应改进。主要

的改进措施有以下几方面。

(1) 每将乘数 Y 的一位乘以被乘数得 $X\times Y_i$ 后，就将该结果与前面所得的结果累加，得到 P_i，称之为部分积。因为没有等到全部计算后一次求和，所以减少了保存每次相乘结果 $X\times Y_i$ 的开销。

(2) 在每次求得 $X\times Y_i$ 后，不是将它左移与前次部分积 P_i 相加，而是将部分积 P_i 右移一位与 $X\times Y_i$ 相加。

(3) 对乘数中为 1 的位执行加法和右移运算；对为 0 的位只执行右移运算，而不需执行加法运算。

因为每次进行加法运算时，只需要将 $X\times Y_i$ 与部分积中的高 n 位进行相加，低 n 位不会改变，因此，只需用 n 位加法器就可实现两个 n 位数的相乘。

上述思想可以写成如下数学推导过程：

$$\begin{aligned}X\times Y&=X\times(0.Y_1Y_2\cdots Y_n)\\&=X\times Y_1\times 2^{-1}+X\times Y_2\times 2^{-2}+X\times Y_3\times 2^{-3}+\cdots+X\times Y_n\times 2^{-n}\\&=\underbrace{2^{-1}(2^{-1}(2^{-1}\cdots 2^{-1}(2}_{n\text{个}2^{-1}})^{-1}(0+X\times Y_n)+X\times Y_{n-1})+\cdots+X\times Y_2)+X\times Y_1)\end{aligned}$$

上述推导过程具有明显的递归性质，其递推公式为

$$P_{i+1}=2^{-1}(P_i+X\times Y_{n-i})\quad(i=0,1,2,3,\cdots,n-1)\tag{3-5}$$

设 $P_0=0$，无符号数乘法过程可以归结为循环地计算下列算式的过程。

$$\begin{aligned}P_1&=2^{-1}(P_0+X\times Y_n)\\P_2&=2^{-1}(P_1+X\times Y_{n-1})\\&\vdots\\P_n&=2^{-1}(P_{n-1}+X\times Y_1)\end{aligned}$$

上述推导过程中的 P_i 称为部分积，每一步迭代过程如下。

(1) 取乘数的最低位 Y_{n-i} 判断。

(2) 若 Y_{n-i} 为 1，则将上一步迭代部分积 P_i 与 X 相加；若 Y_{n-i} 为 0，则什么也不做。

(3) 右移一位，产生本次部分积 P_{i+1}。

部分积 P_i 和 X 进行无符号数相加，可能会产生进位，因而需要有一个专门的进位位 C。整个迭代过程从乘数最低位 Y_n 和 $P_0=0$ 开始，经过 n 次“判断—加法—右移”循环，直到求出 P_n 为止。P_n 就是最终的乘积。假定每次循环在一个时钟周期内完成，则 n 位乘法需要用 n 个时钟周期来完成。图 3.10 是实现两个 32 位无符号数乘法的逻辑结构图。

图 3.10 中被乘数寄存器 X 用于存放被乘数；乘积寄存器 P 开始时置初始部分积 $P_0=0$，结束时存放的是 64 位乘积的高 32 位；乘数寄存器 Y 开始时置乘数，结束时存放的是 64 位乘积的低 32 位；进位触发器 C 保存加法器的进位信号；计数器 C_n 存放循环次数，初值是 32，每循环一次，C_n 减 1，当 $C_n=0$ 时，乘法运算结束；ALU 是乘法器核心部件，在控制逻辑的控制下，对乘积寄存器 P 和被乘数寄存器 X 的内容进行“加”运算，在“写使能”控制下运算结果被送回乘积寄存器 P，进位位存放在 C 中。

每次循环都要对进位位 C、乘积寄存器 P 和乘数寄存器 Y 实现同步“右移”，此时，进位信号 C 移入寄存器 P 的最高位，寄存器 P 的最低位移出到寄存器 Y 的最高位，寄存器 Y 的最低位移出，0 移入进位位 C 中。从最低位 Y_n 开始，逐次把乘数的各个数位 Y_{n-i} 移到寄存器 Y

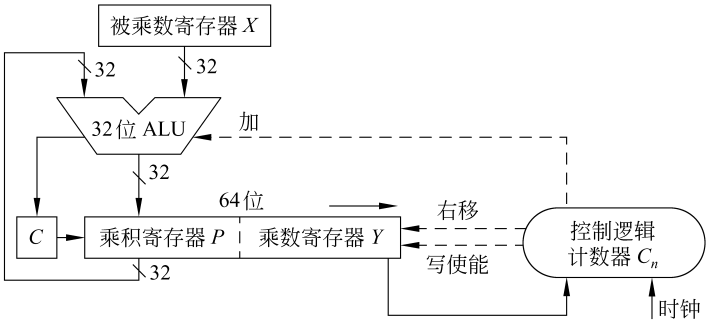


图 3.10　实现 32 位无符号数乘法运算的逻辑结构

的最低位上。因此,寄存器 Y 的最低位被送到控制逻辑以决定被乘数是否“加”到部分积上。

对于原码定点小数的乘法运算,只要根据上述无符号数的乘法运算得到乘积的数值部分,然后再加上符号位,就可以得到最终原码表示的乘积。需要补充说明一点,当被乘数或乘数中至少有一个为全 0 时,结果直接得 0,不需要再进行乘法运算。

例 3.2　已知$[x]_{原}=0.1101$,$[y]_{原}=0.1011$,用原码一位乘法计算$[x\times y]_{原}$。

解:先采用无符号数乘法计算 1101×1011 的乘积,原码一位乘法过程如下。

C	P	Y	说明
0	0 0 0 0	1 0 1 1	$P_0=0$
	+1 1 0 1		Y_4=1, $+X$
0	1 1 0 1		C,P和 Y同时右移一位
0	0 1 1 0	1 1 0 1	得 P_1
	+1 1 0 1		Y_3=1,$+X$
1	0 0 1 1		C,P和 Y同时右移一位
0	1 0 0 1	1 1 1 0	得 P_2
	+0 0 0 0		Y_2=0, 不做加法 (加 0)
0	1 0 0 1	1 1 1 0	C,P和 Y同时右移一位
0	0 1 0 0	1 1 1 1	得 P_3
	+1 1 0 1		Y_1=1, $+X$
1	0 0 0 1		C,P和 Y同时右移一位
0	1 0 0 0	1 1 1 1	得 P_4

符号位为 $0\oplus 0=0$,因此,$[x\times y]_{原}=0.10001111$。

2. 原码两位乘法

对于 n 位原码一位乘法来说,需要经过 n 次“判断—加法—右移”循环,运算速度较慢。如果对乘数的每两位取值情况进行判断,使每步求出对应于该两位的部分积,则可将乘法速度提高一倍。这种方法被称为原码两位乘法,只需在原码一位乘法的基础上增加少量的逻辑线路,就可实现原码两位乘法。

考查乘数每两位的组合以及对应的求部分积的操作情况,归纳如下:

若 $Y_{i-1}Y_i=00$,则 $P_{i+1}=2^{-2}(P_i+0)$

若 $Y_{i-1}Y_i=01$,则 $P_{i+1}=2^{-2}(P_i+X)$

若 $Y_{i-1}Y_i=10$,则 $P_{i+1}=2^{-2}(P_i+2X)$

若 $Y_{i-1}Y_i=11$,则 $P_{i+1}=2^{-2}(P_i+3X)$

对于上述“+0”和“+X”的情况，与前面原码一位乘法一样计算即可；对于“+$2X$”，可通过X左移1位来实现；对于“+$3X$”，则以$4X-X$代替$3X$，在本次运算中只执行$-X$，而+$4X$则延迟到下一次执行。因此，这种情况下，部分积可以由下式得到：$P_{i+1}=2^{-2}(P_i+3X)=2^{-2}(P_i-X+4X)=2^{-2}(P_i-X)+X$。“$-X$”用$+[-X]_{补}$实现。因为下一次部分积已右移了两位，所以，上次未完成的“+$4X$”已变成“+X”。可用一个触发器T记录是否需要下次执行“+X”，若是，则$1\rightarrow T$。因此，实际操作中用Y_{i-1}、Y_i和T三位来控制乘法操作。

3.3.4 补码乘法运算

补码作为机器中带符号整数的表示形式，需要计算机能实现定点补码整数的乘法运算。根据每次部分积是一位相乘得到还是两位相乘得到，有补码一位乘法和补码两位乘法。

1. 补码一位乘法

A.D.Booth提出了一种补码相乘算法，可以将符号位与数值位合在一起参与运算，直接得出用补码表示的乘积，且正数和负数同等对待。这种算法被称为布斯(Booth)乘法。

因为补码用来表示带符号整数，机器字长都是字节的倍数，所以考查偶数位的补码定点整数的乘法运算。即假定$[x]_{补}$和$[y]_{补}$是两个偶数位补码定点整数，$[x\times y]_{补}$的布斯乘法递推公式推导如下。

设：$[x]_{补}=X_{n-1}\cdots X_1X_0$，$[y]_{补}=Y_{n-1}\cdots Y_1Y_0$，根据补码定义，可得到真值$y$的计算公式如下。

$$\begin{aligned}y&=-Y_{n-1}2^{n-1}+\sum_{i=0}^{n-2}Y_i2^i\\&=-Y_{n-1}2^{n-1}+Y_{n-2}2^{n-2}+\cdots+Y_12^1+Y_02^0\\&=-Y_{n-1}2^{n-1}+Y_{n-2}2^{n-1}-Y_{n-2}2^{n-2}+\cdots+Y_12^2-Y_12^1+Y_02^1-Y_02^0\\&=(Y_{n-2}-Y_{n-1})2^{n-1}+(Y_{n-3}-Y_{n-2})2^{n-2}+\cdots+(Y_0-Y_1)\ 2^1+(0-Y_0)\ 2^0\\&=\sum_{i=0}^{n-1}(Y_{i-1}-Y_i)\ 2^i\end{aligned}$$

这里假设$Y_{-1}=0$。因此，

$$[x\times y]_{补}=\left[x\times\sum_{i=0}^{n-1}(Y_{i-1}-Y_i)\ 2^i\right]_{补}\tag{3-6}$$

与推导无符号数乘法算法一样，可以不考虑小数点位置，只要最终的乘积约定好小数点位置就可以了。因此，式(3-6)的右边可以通过乘以2^{-n}来变换成以下形式：

$$\left[x\times\sum_{i=0}^{n-1}(Y_{i-1}-Y_i)\ 2^{-(n-i)}\right]_{补}\tag{3-7}$$

将式(3-7)展开后，得到如下递推公式：

$$[P_{i+1}]_{补}=[2^{-1}(P_i+(Y_{i-1}-Y_i)\ x)]_{补}\quad(i=0,1,2,\cdots,n-1)\tag{3-8}$$

此公式中P_i为上次部分积，P_{i+1}为本次部分积。令$[P_0]_{补}=0$，则有：

$$\begin{aligned}&[P_1]_{补}=[2^{-1}(P_0+(Y_{-1}-Y_0)\times x)]_{补}\\&\vdots\\&[P_{n-1}]_{补}=[2^{-1}(P_{n-2}+(Y_{n-3}-Y_{n-2})\times x)]_{补}\\&[P_n]_{补}=[2^{-1}(P_{n-1}+(Y_{n-2}-Y_{n-1})\times x]_{补}\end{aligned}\tag{3-9}$$

比较式(3-6)和式(3-9),可以得出结论:$[x\times y]_{补}=2^{n}[P_n]_{补}$,因此,只要将最终部分积$[P_n]_{补}$的小数点约定到最右边就行了。

由递推公式(3-8)可知,在求得$[P_i]_{补}$后,根据对乘数中连续两位Y_iY_{i-1}的判断,就可求得$[P_{i+1}]_{补}$。

若$Y_i\ Y_{i-1}=01$,则$[P_{i+1}]_{补}=[2^{-1}(P_i+x)]_{补}$。

若$Y_i\ Y_{i-1}=10$,则$[P_{i+1}]_{补}=[2^{-1}(P_i-x)]_{补}$。

若$Y_i\ Y_{i-1}=00$或11,则$[P_{i+1}]_{补}=[2^{-1}(P_i+0)]_{补}$。

上述式子$[2^{-1}(P_i\pm x)]_{补}$可通过执行$[P_i]_{补}+[\pm x]_{补}$后右移一位实现。此时,采用的是补码右移方式,即带符号整数的算术右移。

根据上述分析,归纳出补码乘法运算规则如下:

(1) 乘数最低位增加一位辅助位$Y_{-1}=0$;

(2) 根据$Y_i\ Y_{i-1}$的值,决定是“$+[x]_{补}$”“$-[x]_{补}$”还是“$+0$”;

(3) 每次加减后,算术右移一位,得到部分积;

(4) 重复第(2)和第(3)步n次,结果得$[x\times y]_{补}$。

图3.11是实现32位补码一位乘法的逻辑结构图,和图3.10所示的无符号数乘法电路的逻辑结构很类似,只是部分控制逻辑不同。

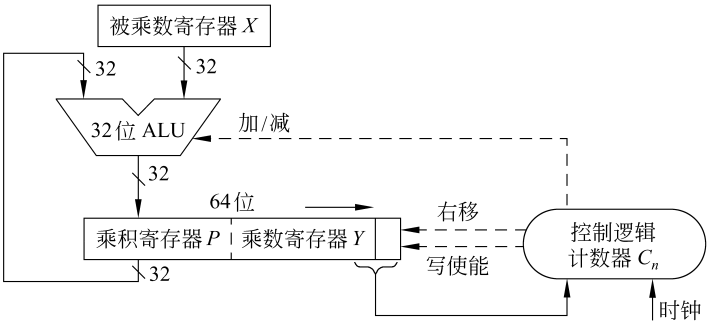


图3.11　实现补码一位乘法的逻辑结构

例3.3　已知$[x]_{补}=1\ 101$,$[y]_{补}=0\ 110$,要求用布斯乘法计算$[x\times y]_{补}$。

解:$[-x]_{补}=0\ 011$,布斯乘法过程如下。

P	Y	Y_{-1}	说　明
0 0 0 0	0 1 1 0	0	设$Y_{-1}=0$, $[P_0]_{补}=0$
0 0 0 0	0 0 1 1	0	$Y_0Y_{-1}=00$, P、Y直接右移一位 得$[P_1]_{补}$
+0 0 1 1			$Y_1Y_0=10$, $+[-x]_{补}$
0 0 1 1			P、Y同时右移一位
0 0 0 1	1 0 0 1	1	得$[P_2]_{补}$
0 0 0 0	1 1 0 0	1	$Y_2Y_1=11$, P、Y直接右移一位 得$[P_3]_{补}$
+1 1 0 1			$Y_3Y_2=01$, $+[x]_{补}$
1 1 0 1			P、Y同时右移一位
1 1 1 0	1 1 1 0	0	得$[P_4]_{补}$

因此,$[x\times y]_{补}=1110\ 1110$。

验证：$x=-011\text{B}=-3$，$y=+110\text{B}=6$，$x\times y=-001\ 0010\text{B}=-18$，结果正确。

布斯乘法的算法过程为 n 次“判断－加减－右移”循环，从流程图可以看出，在布斯乘法中，遇到连续的 1 或连续的 0 时，可跳过加法运算直接进行右移操作，因此，布斯算法的运算效率较高。

2. 补码两位乘法

补码乘法也可以采用两位乘的方法，把乘数分成两位一组，根据两位代码的组合决定加或减被乘数的倍数，形成的部分积每次右移两位。补码两位乘的方法可以用布斯乘法过程来推导。

假设用布斯乘法已经求得部分积$[P_i]_{补}$，则部分积$[P_{i+1}]_{补}$和$[P_{i+2}]_{补}$可分别写为

$$[P_{i+1}]_{补}=2^{-1}([P_i]_{补}+(Y_{i-1}-Y_i)[x]_{补}) \tag{3-10}$$

$$[P_{i+2}]_{补}=2^{-1}([P_{i+1}]_{补}+(Y_i-Y_{i+1})[x]_{补}) \tag{3-11}$$

把式(3-10)代入式(3-11)中，可以得到：

$$\begin{aligned}[P_{i+2}]_{补}&=2^{-1}(2^{-1}([P_i]_{补}+(Y_{i-1}-Y_i)[x]_{补})+(Y_i-Y_{i+1})[x]_{补})\\&=2^{-2}([P_i]_{补}+(Y_{i-1}+Y_i-2Y_{i+1})[x]_{补})\end{aligned} \tag{3-12}$$

从式(3-12)可看出，由乘数中相邻 3 位 Y_{i+1}、Y_i、Y_{i-1}的组合作为判断依据，可以跳过$[P_{i+1}]_{补}$的计算步骤，即从$[P_i]_{补}$直接求得$[P_{i+2}]_{补}$。补码两位乘法运算过程与布斯乘法相似，因此称为改进布斯算法(modified booth algorithm，MBA)，也称为基 4 布斯算法。因为字长总是 8 的倍数，所以补码的位数 n 应是偶数，因此，总循环次数为 $n/2$。该算法可将部分积数目压缩一半，从而提高运算速度。

*3.3.5 快速乘法器

乘法是数字信号处理中重要的基本运算。在图像、语音、加密等数字信号处理领域，乘法器扮演着重要的角色，并在很大程度上决定着系统性能。乘法器也是处理器中进行数据处理的关键部件，大约 1/3 是乘法运算。因此，有必要考虑实现高速乘法运算。前面介绍的原码两位乘法和补码两位乘法(MBA)，通过一次判断两位乘数来提高乘法速度。同理，可以采用一次判断更多位乘数的乘法，但是多位乘法运算的控制复杂度呈几何级数增长，实现难度很大。随着大规模集成电路技术的飞速发展，出现了采用硬件叠加或流水处理的快速乘法器件，如阵列乘法器就是其中之一。

图 3.12 是用手算进行两个 4 位无符号数相乘的示意图。在手算算式中，每个 X_iY_j($i=1\sim4$，$j=1\sim4$)都是由两个 1 位的二进制数相乘得到的。第一行为 X_iY_4($i=1\sim4$)；第二行为 X_iY_3($i=1\sim4$)；第三行为 X_iY_2($i=1\sim4$)；第四行为 X_iY_1($i=1\sim4$)。所以，每个 X_iY_j($i=1\sim4$，$j=1\sim4$)可以用一个“与”门实现。每行都向左错一位，最终将权相等的位的积相加，形成最终的乘积 $P=P_7P_6P_5P_4P_3P_2P_1P_0$。

在计算机内，用组合逻辑线路可以构成一个实现上述执行过程的乘法器。如图 3.13 所示，该乘法器为阵列结构形式，故称为阵列乘法器(array multiplier)。图中实现了 $X\times Y$，其中 $X=X_1X_2X_3X_4$，$Y=Y_1Y_2Y_3Y_4$。X 和 Y 是无符号数。一位乘积 X_iY_j 可以用一个两输入端的“与”门实现。每一次加法操作用一个全加器实现。2^i 和 2^j 的因子所蕴含的移位由全加器的空间错位来实现。与门和全加器的功能可用一个单元组合起来，称为一个细胞模块，在图中用一个方框来表示。

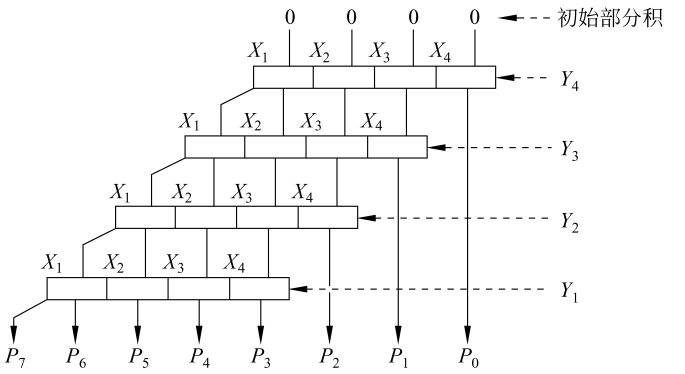


图 3.12　4 位无符号数的手算过程

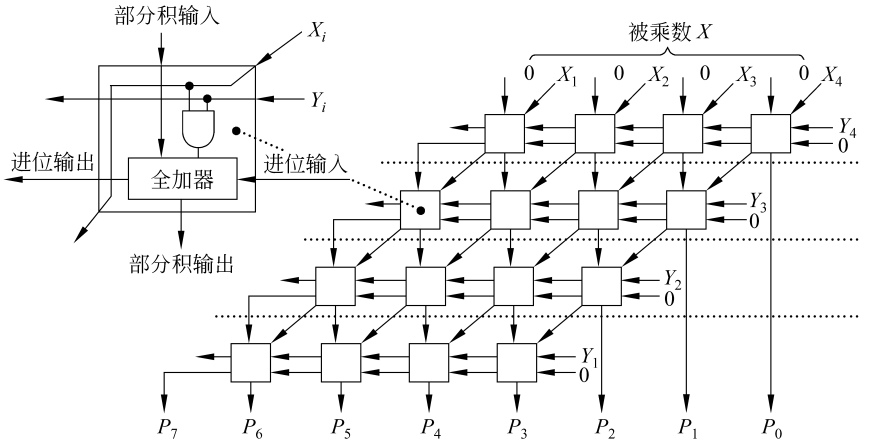


图 3.13　4×4 位基于 CRA 的阵列乘法器

阵列乘法器基于移位与求和算法，每一行中被乘数与乘数中的某一位相乘，产生一组部分积。即每一行由乘数的每一数位 $Y_j(j=1,2,3,4)$ 控制得到本级的部分积 $X_i 2^{(4-i)} \times Y_j(i=1,2,3,4)$。而每一斜列则由被乘数的每一数位 $X_i(i=1,2,3,4)$ 控制，即为 $X_i \times Y_j 2^{(4-j)}(j=1,2,3,4)$。如此求出全部部分积，最后对所有部分积求和得到乘积，整个电路的延迟取决于用于求和的加法阵列结构。

图 3.13 中采用的是基于行波进位加法器(carry ripple adder，CRA)的阵列乘法器，采用串行进位，每一级部分积的生成不仅依赖上一级的部分积，还依赖于上一级的最终进位，因而运算速度慢。为加快运算速度，加法阵列可改用基于进位保留加法器(carry save adder，CSA)方式的结构，如图 3.14 所示。CSA 将本级进位与本级和一样同时输出至下一级，而不是向前传递到本级的下一位，因而求和速度快，且向下级传递的速度与字长无关。

阵列乘法器结构规范，标准化程度高，有利于布局布线，适合用超大规模集成电路实现，且能获得较高的运算速度，其乘法速度仅取决于逻辑门和加法器的传输延迟。随着集成电路价格的不断下降，阵列乘法器在某些数字系统中也被大量使用，例如在数字信号处理系统中受到重视。

阵列乘法器至少要做 $O(N)$ 次加法，速度较慢。为了进一步提高速度，部分积求和电路

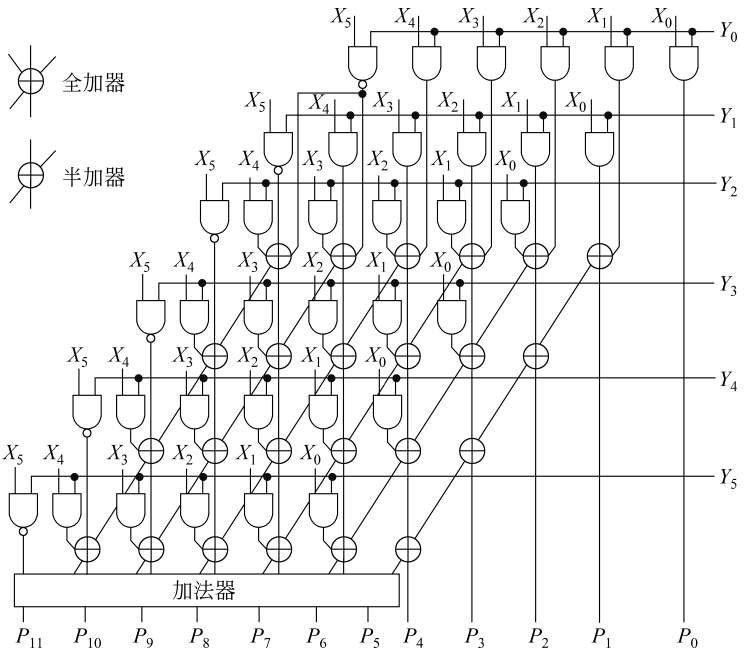


图 3.14　6×6 位基于 CSA 的阵列乘法器

可采用树形结构。树形结构可以减少求和级数，是提高乘法运算速度的一种方法。1961 年 C.S.Wallac 提出的华莱士树(Wallace tree，WT)结构是其中最著名的一种，它对 16 位以上的乘法运算尤其适用。WT 结构将全部部分积按列分组，每列对应一组加法器，各列同时相加，前一列进位传至后一列，生成新的部分积阵列；按同样的方法化简新的阵列，直至只剩两行部分积，最后用高速加法器求和得到最终乘积。WT 结构只需做 $O(\log N)$次加法，因而运算速度快。可将 MBA 和 WT 结合起来进一步加快乘法速度，MBA 用来减少部分积个数，而 WT 用来缩短部分积求和时间。

3.3.6　原码除法运算

在进行定点数除法运算前，首先要对被除数和除数的取值和大小进行相应的判断，以确定除数是否为 0、商是否为 0、是否溢出或为不确定的值 NaN。通常的判断操作如下。

(1) 若被除数为 0、除数不为 0，或者定点整数除法时|被除数|＜|除数|，则说明商为 0，余数为被除数，不再继续执行。

(2) 若被除数不为 0、除数为 0，对于整数，则发生“除数为 0”异常；对于浮点数，则结果为无穷大。

(3) 若被除数和除数都为 0，对于整数，则发生除法错异常；对于浮点数，则有些机器产生一个不发信号的 NaN，即 quiet NaN。

只有当被除数和除数都不为 0，并且商也不可能溢出(如补码中最大负数除以－1) 时，才进一步进行除法运算。

原码作为浮点数尾数的表示形式，需要计算机能实现定点原码小数的除法运算。因此，本节接下来介绍原码除法运算。除法运算与乘法运算很相似，都是一种移位和加减运算的

迭代过程,但比乘法运算更加复杂。下面以两个定点正数为例,说明手算除法步骤。

假定被除数 $X=10011101$,除数 $Y=1011$,以下是这两个数相除的手算过程:

```
                1110   商
除数 1011 ) 10011101   被除数
            1011
            -----
             10001
             1011
            -----
              01100   <-- 中间余数
              1011
             -----
               0011
               0000
             -----
               0011   余数
```

从上述过程和结果来看,手算除法的基本要点如下。

(1) 被除数与除数相减,若够减,则上商为1;若不够减,则上商为0。

(2) 每次得到的差为中间余数,将除数右移后与上次的中间余数比较。用中间余数减除数,若够减,则上商为1;若不够减,则上商为0。

(3) 重复执行第(2)步,直到求得的商的位数足够为止。

计算机内部的除法运算与手算算法一样,通过被除数(中间余数)减除数来得到每一位商。

原码除法运算与原码乘法运算一样,要将符号位和数值位分开来处理。商的符号为相除两数符号的"异或"值,商的数值为两数绝对值之商。因此,以下考虑定点正整数和定点正小数的除法运算。如图3.15所示是一个32位除法逻辑结构示意图。

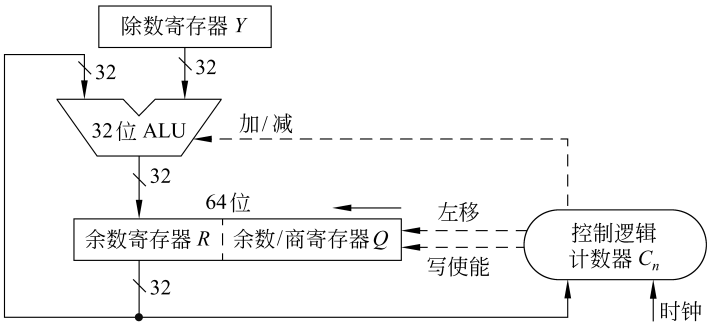


图3.15　32位除法运算逻辑结构

图3.15中除数寄存器 Y 存放除数;余数寄存器 R 开始时置被除数的高32位,作为初始中间余数 R_0 的高位部分,结束时存放的是余数;余数/商寄存器 Q 开始时置被除数的低32位,作为初始中间余数 R_0 的低位部分,结束时存放的是32位商,在运算过程中,Q 中存放的并不是商的全部位数,而是部分为被除数或中间余数,部分为商,只有到最后一步才是商的全部位数;计数器 C_n 存放循环次数,初值是32,每循环一次,C_n 减1,当 $C_n=0$ 时,除法运算结束;ALU是除法器核心部件,在控制逻辑控制下,对于余数寄存器 R 和除数寄存器 Y 的内容进行"加/减"运算,在"写使能"控制下运算结果被送回寄存器 R。

每次循环都要对寄存器 R 和 Q 实现同步"左移",左移时,Q 的最高位移入 R 的最低

位，Q 中空出的最低位上被上“商”。从低位开始，逐次把商的各个数位左移到 Q 中。每次由控制逻辑根据 ALU 运算结果的符号位来决定上商为 0 还是 1。

由图 3.15 可知，两个 32 位数相除，必须把被除数扩展成一个 64 位数。推而广之，n 位定点数的除法，实际上是用一个 $2n$ 位的数去除以一个 n 位的数，得到一个 n 位的商。因此需要进行被除数的扩展。

定点正整数和定点正小数的除法运算，都可以用图 3.15 所示的除法逻辑来实现。只是被除数扩展的方法不太一样，此外，导致溢出的情况也有所不同。

(1) 对于两个 n 位定点正整数相除的情况，即当两个 n 位无符号数相除时，只要将被除数 X 的高位添 n 个 0 即可。即 $X=X_{n-1}X_{n-2}\cdots X_0$ 变成 $X=0\cdots0\ X_{n-1}X_{n-2}\cdots X_0$。显然，对被除数预置时，$R$ 寄存器中为全 0，Q 寄存器中为被除数 X。这种方式通常称为单精度除法，其商的位数一定不会超过 n 位，因此不会发生溢出。

(2) 对于两个 n 位定点正小数相除的情况，即当两个作为浮点数尾数的 n 位原码小数相除时，只要在被除数 X 的低位添加 n 个 0 即可。即将 $X=0.X_{n-1}X_{n-2}\cdots X_0$ 变成 $X=0.X_{n-1}X_{n-2}\cdots X_00\cdots0$，显然，扩展为 $2n$ 位后，R 寄存器中为被除数 X，Q 寄存器中为全 0。

(3) 对于一个 $2n$ 位的数与一个 n 位的数相除的情况，则无须对被除数 X 进行扩展，这种情况下，商的位数可能多于 n 位，因此，有可能发生溢出。采用这种方式的机器，其除法指令给出的被除数在两个寄存器或一个双倍字长寄存器中，这种方式通常称为双精度除法。

综合上述几种情况，可把定点正整数和定点正小数归结在统一的假设下，并将其统称为无符号数的除法。因而，可以假定：除法运算时，被除数 X 为 $2n$ 位，除数 Y 和商 Q 都为 n 位。本书后续对无符号数除法和原码定点小数除法的算法描述也都基于这个假设。

参考手工除法过程，得到计算机中两个无符号数除法的运算步骤和算法要点如下。

(1) 操作数预置。在确认被除数和除数都不为 0 后，将被除数(必要时进行 0 扩展)置于余数寄存器 R 和余数/商寄存器 Q 中，除数置于除数寄存器 Y 中。

(2) 做减法试商。根据 $R-Y$ 的符号来判断两数大小。若为正，则上商 1；若为负，则上商 0。

(3) 上商为 0 时恢复余数。把减掉的除数再加回来，恢复原来的中间余数。

(4) 中间余数左移，以便继续试商。手算除法中，每次试商前，除数右移后，与中间余数进行比较。在计算机内部进行除法运算时，除数在除数寄存器中不动，因此，需要将中间余数左移，将左移结果与除数相减以进行比较。左移时中间余数和商一起左移，Q 的最低位空出，以备上商。

上述算法要点(3)中，采用了“上商为 0 时恢复余数”的方式，所以，把上述这种方法称为“恢复余数法”。也可以不这样做，而是在下一步运算时把当前多减的除数补回来。这种方法称为“不恢复余数法”，又称“加减交替法”。根据余数恢复方式的不同，有“恢复余数除法”和“不恢复余数除法”两种。

1. 恢复余数除法

假定被除数 X 为 $2n$ 位，除数 Y 和商 Q 都为 n 位，X、Y 和 Q 分别表示为：$X=X_{2n-1}X_{2n-2}\cdots X_nX_{n-1}\cdots X_0$，$Y=Y_{n-1}Y_{n-2}\cdots Y_0$，$Q=Q_{n-1}Q_{n-2}\cdots Q_0$，则恢复余数除法的算法步骤如下。

第 1 步，$R_1=X-Y$，若 $R_1<0$，则上商 $Q_n=0$，同时恢复余数，即 $R_1=R_1+Y$；若 $R_1\geqslant$