第5章 数据的表示与运算

电子数字计算机问世至今，其应用范围越来越广泛。但是本质上它仍然是一个信息处理的工具。也就是说，不管计算机的功能多么强大，都是通过对相关信息进行加工处理来完成的。这就涉及一些基本问题：需要计算机处理的信息在计算机中如何表示？对于不同性质的信息计算机如何区别？计算机运算和人工计算的方法一样吗？这些运算是靠什么来完成的？本章将针对上述问题，着重讨论信息的机内表示方法、计算机的运算方法、基本运算部件的逻辑实现等内容。通过讨论向读者介绍计算机中运算器部件的基本工作原理和组成方式。

5.1 计算机中表示信息的基本方法

5.1.1 计算机中常用的信息类型

随着计算机应用领域的不断拓展，现代计算机不仅需要处理传统的科学计算领域的数值问题，而且需要处理大量非数值领域的问题。因此，计算机中处理的信息类型除了数字，还有文字、符号、图形、图像、音频和视频等多种形式。不管信息在计算机外部以何种形式出现，在送到计算机中进行处理时，都必须进行数字化编码，也就是要转换成离散的数字量形式才能由计算机识别，进而在计算机内部进行保存、加工与传送。这样做是由计算机内部采用数字电路的结构特点所决定的。

1. 数字化编码的基本原则

数字化编码的基本原则：用少量、简单的基本符号，按照一定的组合规则，表示出大量复杂、多样的信息。基本符号的种类和其组合规则构成了信息编码的两大要素。像日常生活中熟悉的十进制数，采用0～9十个阿拉伯数码等作为基本符号，通过其特定的组合规则表示出了庞大的数值体系。

那么，什么样的数字化编码适合在计算机中实现呢？早在1945年第一台电子计算机ENIAC的研制过程中，冯·诺依曼对其优缺点进行了深入的分析研究，提出了一台新机器的设计方案。在关于存储程序机设想的科学报告中，他明确指出，电子元件的机器不应采用人们习惯使用的十进制而应采用二进制。因此，目前在电子数字计算机中，广泛采用二进制编码（基二码）来表示各种不同的信息。

2. 基二码的特点

计算机内部采用二进制编码的好处大致可归纳为以下几点。

1）易于物理实现

一位二进制编码仅有“0”“1”两个基本符号，可以很方便地使用具有两个稳定状态的物

理器件实现,例如电位的高、低,脉冲的有、无等。而电子元件的双稳态工作性质和开关特性正好适用。

2) 运算的简易性

二进制的编码、计数和算术运算规则最简单,因此执行基本算术运算也最快。二进制的两个基本符号"0"和"1"正好与逻辑命题的"真"和"假"相对应,为程序中大量的逻辑判断提供了便利的条件。

3) 很高的经济性

与采用十进制表示数据相比,二进制表示使用的器材更少、更经济。

4) 适合逻辑代数的应用

采用二进制就面向着使用二值逻辑,很适合用数字逻辑电路来实现其内部结构,而且特别有利于采用逻辑代数作为数学工具来分析、设计和简化计算机内部大量的逻辑线路。

3. 计算机中常用的信息类型

计算机内采用二进制编码表示所有信息之后所面临的关键问题是,计算机如何对这些形式上相同但含义不同的信息加以区别?解决这个问题的基本思路:针对不同的信息制定出具有不同解释规则的基二码,即所谓的机内数据表示。对于"数据"这个名词,人们习惯于片面地理解为数学中使用的"数"的概念。而在计算机中,"数据"泛指所有需要进行加工处理的信息,不仅包括具有数值的数,而且还包括许多本身并没有数值含义的信息,常见的有逻辑数、字符、汉字、音频信号、视频、图像等。其中,音频、视频信号的表示通常归结为一种新的专门信息类别,称为多媒体,对此本书不做讨论。除了多媒体信息之外,目前计算机中常见的数据大体可以分为数值数据和非数值数据两大类。本章将按照这种分类形式对常用的信息表示方法给以介绍,图 5-1 对这些数据种类进行了概括。本节将着重讨论非数值数据、十进制数据在计算机中的表示方法,二进制数值数据的表示则在后继章节中进行专门的讨论。

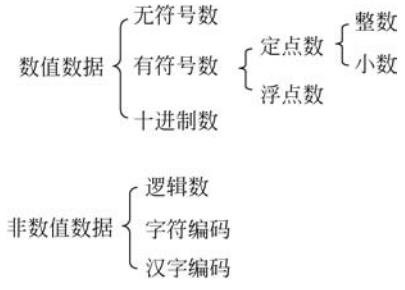


图 5-1 计算机中常用信息综览

5.1.2 非数值数据的表示

1. 逻辑数的表示

逻辑数用二进制的两个基本符号"1"和"0"来表示"是"与"否"或"真"与"假"两种逻辑状态。在这里,"1"和"0"不再被解释成数值的大小,只具有逻辑意义。一个逻辑值需要用一位二进制(bit)表示。为了适应冯·诺依曼机中运算部件以字或字节为单位、多位同时处理的特点,逻辑数通常将多个或一组逻辑值按字节(8 位)或机器字长的规模进行组织,以二进制位串的形式在机内表示。逻辑数虽然形式上和普通的二进制数据一样,但含义是完全不同的,具有按位操作、位与位之间相互独立、无位权、无进位及借位关系等特点。由于逻辑数与数值数据都是一串二进制的 0/1 序列,形式无任何差异,因此计算机中需要通过不同指令来区别它们。例如,逻辑运算指令默认操作数为逻辑数,而算术运算指令默认操作数是数值数据。

逻辑数是计算机中十分重要的数据类型,因为计算机内部硬件实现的基础是逻辑线路,

而逻辑线路能够直接处理的对象就是逻辑数，通常称为逻辑变量。另外，计算机中需要输出的控制信号序列、外部输入计算机的大量状态感应信号等信息形式都可以作为逻辑数据进行发送、测试和处理。为了支持逻辑数的操作，计算机的指令系统中通常都会安排逻辑运算类的指令，像 MIPS 指令集中的 AND(与)、OR(或)、NOR(非或)等，Intel 80x86 微处理器系列的 NOT(非)、XOR(异或)和 TEST(测试)等都是常见的逻辑指令。相应地，在常用的高级语言中也有支持逻辑数据处理的运算类型，像 C 语言就安排有 &(按位 AND)、|(按位 OR)、～(按位 NOT)等运算。逻辑运算的相关内容见本书 5.3.8 节的介绍。

接下来的讨论可能会涉及大量的数据编码，为了方便表示，本书使用常用扩展名字母来表示不同数制的数据。例如，后缀 B 表示这个数是二进制数(Binary)；Q 表示八进制数(Octal)；H 表示十六进制数(Hexadecimai)；D 表示十进制数(Decimal)。十进制是我们日常使用最多的数制，因此书写时经常省略 D，而其他进制的数据书写时后缀一般不可省略。但由于本教材涉及大量对二进制数的讨论，故为简便起见，在不会引起混淆的前提下，书中在给出二进制数据时也经常省略其扩展名。

2. 字符的表示

字符通常指通信行业或人—机交互时大量使用的字母、数字、专用符号等西文信息，是人与计算机进行联系的重要媒介。像书写时常用的英文字母、标点符号、十进制的阿拉伯数码、数学符号等都属于字符范畴。人们日常对字符的表示是基于其字形，但是计算机中的数字电路是无法直接表示出形象符号的，仍然需要借助二进制编码的方式进行表示。当采用二进制编码表示字符时，可把字符看成是计算机中的一种符号数据。对字符进行编码的关键是要保证每个字符都有唯一确定的编码值，这样才能作为识别与使用这些字符的依据。由于西文是一种拼音文字，用有限数量的字母就可以拼写出所有单词，再加上常用的数学符号、标点符号等辅助字符，因此西文字符集的字符总数不是很多，通常每个字符编码使用 7～8 个二进制位表示即可。

1) 字符的交换码标准

有关字符编码的方案很多，目前国际上普遍采用的一种字符编码系统是 ASCII 码，即美国标准信息交换码(American Standard Code for Information Interchange)。它用 7 位二进制编码表示一个特定的字符，因此 ASCII 码字符集总共定义了 128 种字符编码。

在 28 种 ASCII 码字符中，有 95 种字符基本上是日常西文书写时常用的字母或符号，它们的共同特点是“有形”，在把这些字符输入到计算机中时，可在计算机键盘上找到标有这些字母或符号的键，通过敲击这些键就可以把所选的字符送到计算机中。这些字符也可由计算机终端显示输出，或由打印设备进行打印，被称为可显示或可印刷字符。另外 33 种编码则不可以显示或打印，主要用来控制某些外部设备的动作和某些软件的运行方式，以及通信控制等，被称作控制字符。

2) ASCII 码的机内表示

ASCII 码只是一种通用的信息交换码标准，主要解决数字通信系统之间信息传送的相互兼容问题。当用于计算机中表示字符时，还要考虑 ASCII 码的格式与机内数据格式的一致性。由于目前计算机中习惯以字节为单位表示数据，所以 ASCII 码在机内也是用一个字节来进行表示。但是，ASCII 码只占用了 8 个字节中的低 7 位，字节的最高位取值还需要加以确定。对这一位常用的处理方法有如下几种：

(1) 最高位恒为“0”,此时可利用这一位作为 ASCII 码的识别标志。

(2) 最高位用于奇偶校验位,根据奇偶校验技术的需要取值“0”或“1”。

(3) 采用扩展 ASCII 码方案时,最高位也可用作字符编码。例如,将 ASCII 码扩展到 8 位,字符集扩大到 256 种字符。除保留 ASCII 码原有字符之外,可以增加制表符、计算符、希腊字母和拉丁符号等。

3) 字符串的表示

ASCII 机内码解决了单个字符在计算机中的表示问题。但在实际的文字处理等应用场合,使用单个字符的情况并不太多,大量的字符通常都是成串出现的,把这种由连续一串字符组成的数据形式称为字符串。目前,字符串已成为计算机中最常用的数据类型之一,许多计算机的指令系统中都提供了支持字符串存取和处理功能的串操作类型的机器指令,如 Intel 80x86 指令系统中的串处理指令 MOVS、CMPS、SCAS 等。

由于 ASCII 机内码已解决了单个字符在计算机中的表示问题,因此字符串在机内表示时需要解决的关键问题是其在主存中的存放方式。最常用的一种方法称为向量表示法。用它表示字符串时,将字符串存放在主存的连续多个字节单元中,每个字节单元保存串中一个字符的 ASCII 码。在字长较长且主存按字节编址的计算机中,主存单元通常由 2 个(16 位机)或 4 个(32 位机)字节单元组成。此时如采用向量表示法表示字符串,在同一个主存单元中,既可按从低字节单元向高字节单元的顺序存放字符串,也可按从高字节单元向低字节单元的顺序存放字符串。这两种存放方式都很常用,不同的计算机可能会选用其中一种,到底采用哪一种则主要取决于该机主存字节单元的编址顺序。

例如,现在有这样一个字符串:“if (A<B) READ(C)”,其对应的 ASCII 码的十六进制值依次为 69H,66H,20H,28H,41H,3CH,42H,29H,20H,52H,45H,41H,44H,28H,43H,29H。当这个字符串在主存中按从高字节单元向低字节单元的顺序存放时,主存空间占用方式如图 5-2 所示,其中主存单元长度为 32 位,由 4 个字节单元组成。

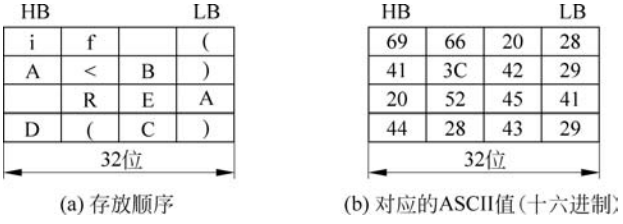


图 5-2 字符串从高到低字节单元存放的方案

采用向量表示法表示字符串时,需要给出串首地址和串长这样两个表征参数来描述字符串在主存中的位置,也可使用特定的结束符来表示字符串的末尾,在给出结束符的情况下串长参数则可省略。向量表示法是最简单、最节省存储空间的字符串存放方法。但是,在进行字符串删除和插入操作时,对被删除或插入的字符后面的剩余字符串部分需要全部重新分配存储空间。在字符串较长的情况下,这将花费较多的时间。

3. 汉字的表示

与国际上广泛采用英文作为文字处理对象的情况有所不同,我国大量使用的语言文字是汉字。计算机要对中文信息进行处理,就必须采用表示西文字符类似的方法,在机内对汉字进行二进制编码。但与西文相比,汉字属大字符集语种,文字数量巨大,其总数可达数万

字，这导致汉字编码所需的二进制位数大大增加。不仅如此，如何使用西文键盘输入汉字？如何将二进制编码表示的汉字输出显示或打印出来？汉字在计算机中如何表示、处理、输入和输出的一系列问题都需要使用专门的方法加以解决。

为了满足计算机对汉字信息的输入、处理、输出等不同工作阶段的特殊需求，汉字处理系统配置了几种性质不同的汉字编码方案，分别用于汉字处理的不同场合。

1）汉字输入码

计算机最早是由西方国家研制的，采用了西文标准键盘作为其最主要的信息输入设备，但经过几十年的发展，到现在绝大多数的计算机仍然继续采用西文标准键盘作为主要的信息输入手段。西文标准键盘的按键排列方式基本上沿袭了西文打字机的键盘布局，一个或两个西文字符对应着一个按键，并充分考虑了盲打指法和速度上的要求，输入英文等西文时使用起来非常方便。但是如果用来输入汉字，由于汉字的数目众多，结构也与西方文字完全不同，因此西文键盘无法像输入英文那样直接提供与之一一对应的关系。

为利用西文标准键盘直接输入汉字，广泛采用的解决方法是为汉字设计专门的输入编码。与二进制编码概念不同的是，汉字输入码的码元（组成编码的基本元素）是西文键盘中的某个按键，每个汉字用一个或几个键的特定组合表示。我国的汉字输入码方案很多，但真正推广应用较好的只有少数几种。能够被广泛接受的汉字输入码方案一般具有易学、易记、效率高（击键次数较少）、重码少、容量大（包含的字数多）等特点。

到目前为止还没有一种方案在所有方面都做到很好。常用的汉字输入码方案大体分以下四类：数字编码、字音编码、字形编码和形音编码。由于汉字输入码仅仅解决汉字的键盘输入问题，因此为与汉字的机内编码相区别，汉字输入码又称为汉字机外码或“外码”。实际上，不管采用哪种汉字输入码方案其实质都是一样的，都是利用键盘进行“手动”输入。比较理想的输入方式是利用语音或图像识别技术“自动”将汉字文本输入到计算机内，并将其自动转换为机内代码表示。目前，已经有手写汉字联机识别输入、汉字扫描输入后自动识别、语音输入汉字等。

2）汉字交换码标准

为了便于不同的信息处理系统之间相互通信以及汉字信息交换的使用需要，早在 1981 年我国就颁布了汉字交换码标准，即《信息交换用汉字编码字符集 基本集》（GB 2312—1980），这个标准称为国标码，又称国标交换码。该标准共收入 6763 个常用汉字，并按其使用频度对这些汉字进行了分类。其中一级汉字 3755 个，属于常用汉字，按汉语拼音排序；二级汉字 3008 个，属于次常用字，因为使用频度降低，所以按偏旁部首排序。此外，一些常用的字母、数字和符号也收入了该标准的字符集，包括英文、俄文、日文平假名与片假名、罗马字母、汉语拼音等共 687 个。

GB 2312—1980 汉字交换码标准为字符集中的每一种汉字或其他字符规定了一个唯一的二进制代码。这些代码以代码表的形式给出，代码表分成 94 个区（对应 94 行），每个区有 94 位（对应 94 列）。表中的行编号称为区号，列编号称为位号，各用 7 位二进制编码表示，形成一个二维数组。表中每个行、列的交叉点上都有一个汉字，交叉点所在位置的行列坐标号合起来就构成了该汉字的编码。其中，7 位区号在左、7 位位号在右，共 14 位。国标码规定区号和位号的取值范围均为十六进制的 21H～7EH，这导致国标码识记起来不很直观。GB 2312—1980 标准还经常采用另一种称为“国标区位码”的表示方式。区位码直接用十进

制的1～94来表示汉字的区号和位号,其对应的二-十进制编码(BCD码)仍以区号在左位号在右的形式表示。与国标码相比,区位码更加直观好记,因此得到了广泛的应用。

3) 汉字机内码

相对于汉字机外码而言,汉字机内码或"内码"是指汉字在计算机内部存储、处理和传送所采用的二进制编码。汉字通过输入码从键盘输入到计算机内部后(或通过语音、手写、扫描等其他手段输入),就以"内码"的形式存在于机内。也就是说,不管汉字采用何种方式输入,都必须转换成机内码才能被计算机存储和处理。因此,汉字机内码与所采用的汉字输入法无关。不同的计算机系统中采用何种汉字机内码方案并没有统一的规定,目前较为广泛的是将国标码或国标区位码用于汉字的机内表示。但与ASCII码机内表示的情况类似,国标码如果直接作为机内码使用,格式上还需要适应计算机内部数据的表示形式。另外,计算机中的中西文信息经常混合在一起进行处理,汉字机内码需要予以特别的标识,才能与ASCII机内码加以区别。现以采用GB 2312—1980方案为例,常用的汉字机内码设定方法如下:

(1) 若ASCII码采用字节最高位恒为"0"的机内表示方案,则通常用两个字节表示一个汉字,两个字节的最高位均设为"1",每个字节的低7位为国标码或国标区位码。

(2) 若ASCII机内码字节的最高位用于奇偶校验位或扩展ASCII码,则需要3个字节表示汉字,其中第一个字节作为汉字的标识符使用。

仍以"红"字为例,其国标码为3A6CH,如采用上述两个字节最高位均为"1"的表示方案,则对应的机内码为BAECH。应当注意,汉字的国标码或区位码是唯一的通用标准编码,而汉字机内码在不同的系统中可能会采用不同的方案实现。

4) 汉字字模码

汉字机内码解决了汉字在计算机内部的编码表示问题。可是,经过计算机处理后的汉字如果需要显示或打印,就必须将二进制编码形式的汉字机内码转换成汉字原来的、可供人们阅读的方块字形式。汉字字模码就是为输出设备输出汉字而设计的字形编码。

汉字的字形主要有两种描述方法:点阵描述和轮廓描述。点阵描述是将汉字的字形用"点"组成的方阵来表示,方阵中有笔画的点位点上黑点,这样我们就可以看到一个用"点"排列出来的汉字。由于汉字的字形较为复杂,因此要能够清晰地描述一个汉字,至少需要16×16左右大小的点阵规模。如果希望"点"出来的汉字更好看,还可以使用更大的点阵,如24×24、32×32,甚至更大。

5.1.3 十进制数据的编码表示

现在我们已经知道计算机内部表示数据的基本方法是采用基二码,而在日常生活中人们习惯使用十进制进行计算,这就导致了计算机内部、外部数据形式的不一致。因此在多数应用环境中,计算机输入数据时通常要将十进制转换成二进制,而输出时又需将二进制转换成十进制。在某些特定的应用领域如商业统计中,需要进行的运算很简单但数据量很大,这样输入输出过程中进制转换所占的时间比例就会很大。从提高机器运行效率的角度出发应尽量减少这种转换。另外,十进制转换成二进制可能会造成数据精度的损失,这在精度要求较高的应用中也是不希望出现的。这些特定场合都对计算机内部能够直接用十进制表示和处理数据提出了要求。为适应这方面的需要,目前大多数通用性较强的计算机中都具有直接表示和处理十进制数据的能力。

1. 十进制数的 BCD 码表示

在计算机内部直接表示十进制数据的基本思路仍然是使其二进制代码化。要使 0～9 这十个阿拉伯数码分别拥有一个唯一的二进制编码，至少需要 4 位二进制码表示。由于 4 位二进制码共有 16 种组合，而表示一位十进制数只需选取其中 10 种，这就使剩下的 6 种变成了冗余的无用组合。如何从 16 种组合中选择 10 种来表示一位十进制数，可供选择的编码方案很多，但比较常用的也只有少数几种。这些编码统称为二进制编码的十进制数(Binary Coded Decimal)，简称 BCD 码或二-十进制码。常用的 BCD 码按照其位权设置方法的区别，可分为有权码、无权码等。

1) 十进制有权码

有权码是指用来表示一个十进制数字的若干二进制数位的每一位都有一个确定的位权，将这些二进制编码中取值为“1”的数位的权值加到一起，就得到了对应的十进制数值。最常用的一种有权码就是 8421 码。8421 码 4 个二进制数位的位权从高到低分别为 8、4、2 和 1，故称 8421 码。这种编码方案选取 4 位二进制编码的前 10 种顺序组合 0000、0001、…、1001 来表示 0～9 这 10 个十进制数字，正好和二进制数的 0～9 取值一致，因此也叫作自然的二进制编码的十进制数(Natural Binary Coded Decimal，NBCD)。8421 码的另一特点是与十进制数 0～9 的 ASCII 码低 4 位取值相同，这使 8421 码与 ASCII 码以及二进制数之间转换很方便。8421 码虽然具有简单直观便于转换的优点，但在计算机内实现算术运算要复杂一些，在某些情况下需要对加法运算的结果进行修正。关于这一点我们将会在运算方法部分进一步讨论。另外几种有权码如 2421 码、5211 码、4311 码等与 8421 码的定义方法类似，表 5-1 列出了这些常用的十进制有权码。

表 5-1　4 位十进制有权码

十进制数	8421 码	2421 码	5211 码	4311 码
0	0000	0000	0000	0000
1	0001	0001	0001	0001
2	0010	0010	0011	0011
3	0011	0011	0101	0100
4	0100	0100	0111	1000
5	0101	1011	1000	0111
6	0110	1100	1010	1011
7	0111	1101	1100	1100
8	1000	1110	1110	1110
9	1001	1111	1111	1111

2) 十进制无权码

另一类常用的二-十进制编码方案是十进制无权码。与有权码表示方法相反，无权码是指用来表示一个十进制数字的 4 个二进制数位的每一位均没有确定的位权。在无权码方案中，使用较多的是余 3 码和格雷码。余 3 码是在 8421 码的基础上加 3(即二进制的 0011)得到，故称“余 3”码。也就是说，余 3 码选取了 4 位二进制编码 16 种组合中的中间 10 种组合来表示一位十进制数。余 3 码的表示方法虽然没有 8421 码直观和转换方便，但在执行十进制加法时，能自动产生出十进制进位信号，有效地简化了进位修正逻辑电路。另外，余 3 码

还具有“自补码”的特性,给十进制减法运算的实现也带来方便。这里所谓的“自补码”是指,如果将某个十进制数的二-十进制编码按位“求反”,就可得到该数相对于9的补码。除了余3码以外,表5-1中的2421码、5211码等有权码也属于“自补码”,而8421码则不具此特性。

格雷码又称循环码。其编码规则是任何两个相邻十进制数字的二-十进制编码只有一位二进制位状态不同,其余3个二进制位的状态必须相同。这样在进行十进制计数时,从一个编码变到下一个编码只有一位二进制位发生状态翻转。这样转换速度达到最快,且有利于得到更加可靠的译码波形,避免了两位以上状态同时翻转带来的不稳定因素,被广泛用于可靠性要求较高的计数电路中,如用来产生平滑改变的控制信号,以及模拟数字转换信号等场合。由于4位二进制组合中6种冗余状态的存在,用格雷码表示十进制数的方案很多。表5-2列出了常用的十进制无权码方案,其中有两种是格雷码。

表5-2　4位十进制无权码

十进制数	余3码	格雷码(1)	格雷码(2)
0	0011	0000	0000
1	0100	0001	0100
2	0101	0011	0110
3	0110	0010	0010
4	0111	0110	1010
5	1000	1110	1011
6	1001	1010	0011
7	1010	1000	0001
8	1011	1100	1001
9	1100	0100	1000

2. 十进制数串的表示

BCD码解决了一位十进制数的机内表示问题。但机内直接采用十进制表示的另一个目的是提高数据的表示范围和运算精度。计算机中的二进制数据由于受字长的限制,可表示的范围和精度是有限的。如果将十进制数转换成二进制进行运算,可能会造成精度的损失。但如果将多位十进制数以数字串的形式直接输入计算机,就可以摆脱二进制机内数据格式的约束。

十进制数串在计算机内主要有以下两种表示形式。

1) 字符串形式

这种表示法把一串十进制数看成一串字符串,采用类似于字符串的表示方法,串中每位十进制数以及正、负号均用对应的ASCII码表示。根据数符的不同安排,此方法又可分为前分隔数字串和后嵌入数字串两种形式。

(1) 前分隔数字串:前分隔数字串表示方法与日常书写顺序相同,是将数符置于十进制数串之前,单独用一个字节来表示。正号用字符“+”的ASCII码(2BH)给出,负号用字符“-”的ASCII码(2DH)给出。例如,十进制数+236可表示为2BH 32H 33H 36H,在内存中占用4个字节;十进制数-2369可表示为2DH 32H 33H 36H 39H,在内存中占用5个字节。

(2) 后嵌入数字串:采用后嵌入数字串方式时,数的符号不再单独用一个字节来表示,

而是嵌入到最低一位数字的 ASCII 码中。具体规定为：正数最低一位数字的 ASCII 编码不变；负数最低一位数字的 ASCII 码的高 4 位由原来的 0011 变为 0111。仍以上面的两个数串为例，+236 此时表示为 32H 33H 36H，在内存中占用 3 个字节；而 −2369 则表示为 32H 33H 36H 79H，在内存中占用 4 个字节。这样，后嵌入数字串方式比前分隔数字串方式少占用一个字节，节省了存储空间。但负数的最低一位数字已不再是 ASCII 码形式，在显示或打印前必须先转换回 ASCII 码。从这一角度看，后嵌入数字串又没有前分隔数字串方式输入输出来得方便。

用字符串方式表示十进制数串方便了十进制数的输入输出，但是对十进制数的机内运算来说却很不方便。因为阿拉伯数字 0～9 的 ASCII 码高 4 位的值并不具有数值意义，必须先去掉转换为二进制数或 BCD 码才能进行算术运算。所以这种方式表示的十进制数串主要应用于非数值计算领域。

2）压缩的十进制数串形式

这种方法在表示十进制数串时，把串中每位十进制数的 ASCII 码高 4 位压缩掉，只用其低 4 位表示（也可看成直接用 8421 码表示）。这样每个十进制数位只需占半个字节的存储空间，一个字节单元就可存放两个十进制数位，比字符串表示方式要少占用一半左右的存储空间。此时十进制数串的符号也占用半个字节并存放在最低数位之后，其值可选用 4 位二进制编码中 8421 码不用的 6 种冗余状态的两种不同值即可。通常用 1100 表示正号，1101 表示负号。这种表示方式规定十进制数串的数值位加符号位之和必须为偶数，如不为偶数应在最高数位之前补一个 0。例如，+123 被表示成 123CH，−12 被表示成 012DH。此表示法指明一个十进制数串时也需给出它在主存中的首地址和十进制数字的位数（又称为位长，不含符号位）。位长为 0 的数其值也为 0。压缩的十进制数串位长可变，许多机器中规定该长度为 0～31，有的甚至更长。另外，它比字符串表示形式节省存储空间，又便于直接完成十进制数的算术运算，是广泛采用的较为理想的表示方法。

虽然采用 BCD 码和十进制数串的方法可以在计算机中直接表示十进制数，但需要占用更多的硬件资源。例如，处理 1000 以内的信息，需要 3 位十进制数，再加上符号位，因而至少需要 $(3+1)\times4=16$ 位的设备量。而对于二进制系统来说，$2^{10}=1024$，只需 10 位的设备量就足够了。因此除非特殊需要，通常情况下计算机内部都用二进制数进行数据的表示和运算。

5.2 定点数的表示

5.2.1 真值与机器数

1. 数符的机内表示

计算机中表示数值数据时，不仅要考虑采用何种进位计数制比较方便，还需要解决数的符号表示问题。数值数据通常有无符号数和有符号数之分。所谓无符号数，就是不考虑数的正负符号，相当于数的绝对值。此时整个机器字长的全部二进制位均可用来表示数值。例如，$X=(0100\ 1010)_2$ 表示无符号数 74；$Y=(1100\ 1101)_2$ 表示无符号数 205。若机器字长为 8 位，则无符号数的表示范围为 0～255。当机器字长为 $n+1$ 位时，无符号数的表示范围是 $0\sim(2^{n+1}-1)$。一般计算机中都设置有一些无符号数的运算和处理指令。如 Intel

8086 中的 MUL 和 DIV 指令就是无符号数的乘法和除法指令,还有一些条件转移指令也是专门针对无符号数的。

然而,在进行数学计算时,大量用到的还是有符号数,即带有正、负号的数。日常生活中通常采用特定的数学符号“+”“-”来表示数的正负,可是这样的符号在计算机中是无法直接表示的。解决的方法又回到了前面讨论的基本思路上,就是把符号二进制数码化,“0”“1”两种代码正好可用来表示数的正、负两种符号。根据这个思想,计算机中在表示有符号数时,一般在最高数值位之前再安排一位专门用来表示数的符号,称为“符号位”,这样就解决了数符的机内表示问题,如图 5-3 所示。

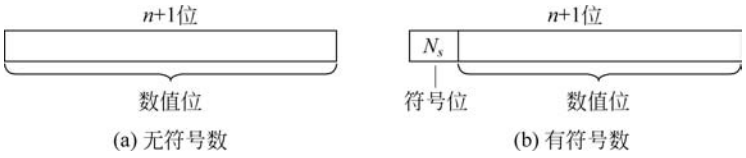


图 5-3　数符的机内表示

注意符号位仅用来表示数符,不具有数值意义。由于无符号数不需要安排符号位,因此在同样机器字长的情况下,比有符号数的表示范围大一倍。例如,设符号位为 0 表示正数,为 1 表示负数,则前面给出的数据用来表示有符号数时,$X=(0100\ 1010)_2$ 的十进制真值为 +74;$Y=(1100\ 1101)_2$ 则对应十进制真值为 -77。在对数的正负号表示作了代码化处理之后,计算机中的数据形式已经和通常的手写形式不一样了。为了加以区别,把计算机内的数据形式称为机器数,而把一般书写形式表示的、带正负符号“+”“-”的数称为机器数的真值。不难看出、无符号数的机器数形式和其二进制真值是一致的,而有符号数的机器数与真值之间的关系则没有这样简单,通常根据符号位与数值位之间的不同编码组合形式又分为原码、反码、补码等不同的表示方式,这个问题将在后续 5.2.3 节中进行详细讨论。

2. 小数点的机内定位

数值数据的机内表示还有一个关键问题需要解决,那就是小数点的定位。这个问题的难点在于小数点在数据格式中无法再用二进制代码表示。我们不妨换一种思路来寻求解决办法,实际上在计算机中通常以“隐含”方式来表示小数点的位置,就是说只要事先约定好小数点的位置,按照这个约定来识别数值即可,数据格式中并不需要明显地给出小数点。目前计算机中采用的小数点表示法分为定点和浮点两种形式。在此,先对比较简单的定点表示法进行讨论,浮点表示法较为复杂,在本章 5.5 节中将进行专门的阐述。

所谓定点表示法指小数点的位置在一台计算机中是事先约定好且固定不变的。通常采用的定位方式有两种:一种将小数点的位置定在最低数值位之后,这导致机内的所有数据均为整数,因此称为定点整数表示法;另一种将小数点的位置定在符号位(最高位)之后,小数点的位置定在这里就意味着机内的所有数据均为小数,因此这种方式称为定点小数表示法。两种表示方式如图 5-4 所示。

定点表示法的缺点是表示范围较小,或者只能表示纯整数(定点整数表示法),或者只能表示纯小数(定点小数表示法),而对于现实计算中普遍采用的带小数形式的数据是无法直接表示的。因此这类数据在运算前需要乘上一定的比例因子,使之变为整数或小数形式,定点计算机内部才会接受。困难的是,对于非常大或非常小的数据,合适的比例因子很难找。

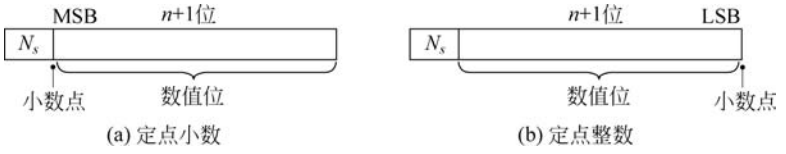


图 5-4　定点数的机内格式

为了讨论方便起见，也为了能够更加清楚地区分符号位和数值位，通常习惯将有符号数的最高数值位用 MSB(Most Significant Bit)表示，称为“最高有效位”；而最低数值位则用 LSB(Last Significant Bit)表示，称为最低有效位。例如，如果我们说“有符号数的最高位”，则一般是指这个数的符号位。而如果说“有符号数的 MSB 位”，则一定是指这个数符号位之后的最高数值位。注意由于位权关系不同，定点小数的 MSB、LSB 和定点整数的 MSB、LSB 所代表的数值是不一样的。在 n 位数值的情况下，定点小数的 1LSB 表示 2^{-n}，而定点整数的 1LSB 就等于 1(2^0)。

5.2.2　常用机器码表示

在将有符号数的正、负符号也用“0”“1”表示之后，就为符号位与数值位之间进行新的编码组合提供了可能性。为了便于在计算机中进行各种运算，特别是最基本的加减运算，计算机中按照有符号数的不同组合规则，提出了不同的机器数编码方案，简称“码制”。因此，机器数也常称为“机器码”。目前计算机中广泛采用的编码方法有原码、补码、反码三种。为了便于区别一个数的真值和对应的各种机器码，本书中用 X 表示真值，$[X]_{原}$ 表示其原码，$[X]_{补}$ 表示其补码，$[X]_{反}$ 表示其反码。

1. 原码表示法

原码是最简单直观的一种机器码表示法，这种方法将机器数的最高位定义为符号位，用“0”表示正号，“1”表示负号；数值部分则与真值保持一致，因此也常被称为“符号-数值表示法”，即原码可看成是由一位符号位与数值位简单的拼接而成，其与二进制真值之间的差别仅仅在于数字的符号被二进制代码化了。这个编码规则可用数学表达式进行定义。

1) 小数的原码表示

如果 n 位二进制小数的真值 $X=\pm 0.X_1X_2\cdots X_n$，则对应的原码定义为

$$[X]_{原}=\begin{cases}X & 0\leqslant X<1\\ 1-X=1+|X| & -1<X\leqslant 0\end{cases} \tag{5-1}$$

例如，$[0.1011010]_{原}=0.1011010$；$[-0.101\ 1010]_{原}=1.1011010$。

书写定点小数时为清楚起见，经常用小数点“.”把符号位和数值隔开。

2) 整数的原码表示

若 X 为 n 位整数，其定点整数原码的形式与定点小数原码完全相同，只是由于小数点的位置变了，原码的定义域和符号位的位权值也变了。定点整数原码定义为

$$[X]_{原}=\begin{cases}X & 0\leqslant X<2^n\\ 2^n-X=2^n+|X| & -2^n<X\leqslant 0\end{cases} \tag{5-2}$$

例如，$[1011010]_{原}=0,1011010$；$[-101\ 1010]_{原}=1,1011010$。

书写定点整数时为清楚起见，经常用逗号“,”把符号位和数值隔开。

3) 零的原码表示

仔细分析原码的定义域可以看出,由于正、负域中都包含了"0"这一点,造成了原码有"+0"和"-0"两种零的表示形式。零的表示不一致性是原码表示法的一个显著缺点,这会给计算机中的判"0"操作带来麻烦。

在学习原码表示法时,不仅要会利用其定义表达式求出原码,还要注意其定义域的边界设定,定义域直接给出了原码表示的范围。

原码表示法很直观,与二进制真值之间的转换非常方便,在计算机中用原码实现乘、除运算的规则简单,但直接用原码进行加减运算相当复杂,需比较两个操作数的符号位,不利于以加减运算为基础的运算部件的实现。

2. 补码表示法

补码表示法的符号位设置与原码相同,而数值部分的表示则与数的正负有关。正数的表示和原码相同,即数值部分与真值保持一致;负数的表示原理则和原码完全不同,采用了真值的"补"作为其补码值。

1) 小数的补码表示

若设真值 $X=\pm 0.X_1X_2\cdots X_n$,则其对应的定点小数补码定义式为

$$[X]_{补}=\begin{cases}X & 0\leqslant X<1\\ 2+X=2-|X| & -1\leqslant X<0\end{cases}\quad(\text{Mod } 2) \tag{5-3}$$

由上述补码定义可看到,补码表示法与原码表示法不同之处在于引入了"模"和"补"的概念。由于上述定义中模=2,因此定点小数补码又称为模2补码。

根据模2补码定义表达式,当 X 为负数时,其补码为

$$\begin{aligned}[X]_{补}&=2+X=2+(-|X|)=2-|X|=10.00\cdots0-0.X_1X_2\cdots X_n\\&=(1.11\cdots1+0.0\cdots01)-0.X_1X_2\cdots X_n\\&=1.11\cdots1-0.X_1X_2\cdots X_n+0.0\cdots01\\&=1.\overline{X}_1\overline{X}_2\cdots\overline{X}_n+0.0\cdots01\end{aligned} \tag{5-4}$$

可以看出,由负数真值求补码时可以不进行定义中的减法运算,只要直接将补码的符号位置"1",数值部分按位取反,且最低位加1即可。这就是常用的补码转换规则"变反+1"的由来。注意:这里的"+1"应更加确切地理解为"变反+1LSB"。对于定点小数来说,1LSB为 2^{-n} 而不是1。

例如,$[0.1011010]_{补}=0.1011010$;$[-0.1011010]_{补}=1.0100110$。

2) 补码零

对补码的定义进行分析后,可发现补码的正负定义域是不对称的,负数域并不包括"0"这一点,也就是说"0"只包括在正数定义域中,不像原码那样正负定义域中都包含"0"点。这个特点保证补码对于"0"的表示是唯一的,不再有"+0""-0"之分,减少了计算机中判"0"操作的麻烦。

3) 补码表示的下限

由于补码"0"的表示只需一个码点,这就意味着在位数同样多的情况下,补码比原码省出了一个码点,利用这个码点补码可以比原码多表示一个数。不难发现负数补码定义域中比原码多包括了"-1"这一点,即负小数的补码表示下限为-1.00…0,而不是像原码那样只能表示到-0.11…1。

例如，$[-1.0000000]_{补}=1.0000000$。此时补码符号位的“1”有双重含义：既表示负号又表示数值“1”，这两重含义在此并不冲突，故负小数补码表示的下限可以达到整数“－1”这一点。注意，此时“－1”仍属小数补码表示范围，与原码相比这很特别。

4）整数的补码表示

定点整数补码的形式与定点小数补码完全相同，运算特性和转换规则也一样，只是定义域、位权关系和模的取值变了。定点整数补码定义如下。

若 X 表示 n 位整数真值，则对应的补码为

$$[X]_{补}=\begin{cases}X & 0\leqslant X<2^{n}\\ 2^{n+1}+X=2^{n+1}-|X| & -2^{n}\leqslant X<0\end{cases}\quad(\text{Mod } 2^{n+1}) \tag{5-5}$$

由整数补码的定义域不难看出，整数零的表示仍然是唯一的。与原码整数表示法相比，整数补码也可以多表示一个点，下限可达-2^{n}。

5）变形补码

补码在计算机中具体应用时，为了实现一些特殊功能，还经常以“变形”的形式出现，用得比较多的是双符号位补码。这种补码与常规的补码不同之处是设置了两个符号位，称为“变形补码”。变形补码的正数符号位取值“00”，负数符号位取值“11”，对于定点小数来说，由于小数点的默认位置定在符号位与数值位之间，因此设置双符号位意味着变形补码的“模”比模 2 补码扩大了一倍，变成模 4 补码。模 4 补码定义如下：

若设 $X=\pm 0.X_1X_2\cdots X_n$ 代表 n 位小数的真值，则

$$[X]_{补}=\begin{cases}X & 0\leqslant X<2\\ 4+X=4-|X| & -2\leqslant X<0\end{cases}\quad(\text{Mod } 4) \tag{5-6}$$

模 4 补码除了具有双符号位以外，其他特性均与模 2 补码类似。

由上述定义可以看出，同样是定点小数补码，由于模 4 补码的“模”比模 2 补码扩大了一倍，因此其定义域也比模 2 补码扩大了一倍，这就意味着模 4 补码的表示范围也比模 2 补码扩大了一倍。变形补码的这个特性在补码运算的溢出判断环节中起着特殊作用，这一点在 5.3.2 节中将做详细介绍。

补码表示法不如原码表示法直观，与真值之间的转换也不是很方便；但零的表示的唯一性这一特点便于硬件判 0 操作的实现，以及较广的表示范围都是补码表示法超出原码的优点。更为重要的是，补码的加减算法比原码要简单得多（见 5.3.2 节），有利于以加减运算为基础的运算部件的实现。

6）补码与真值的转换关系

上面讨论了由真值转换成补码的方法，但有时可能需要倒过来将补码转换成真值。这种逆转换可以依据补码与真值的转换关系式进行。

基于定点小数的补码与真值转换关系表达式如下，若

$$[X]_{补}=X_0.X_1X_2\cdots X_n\quad(\text{Mod } 2)$$

则

$$X=-X_0+\sum_{i=1}^{n}X_i\times 2^{-i} \tag{5-7}$$

这个转换关系式可以直接由补码定义推导而来，具体过程如下。

假定模 2 补码表示为：$[X]_{补}=X_0.X_1X_2\cdots X_n$，如统一考虑正负数的情况，其定义可

写成：

$$[X]_{补}=2X_0+X \tag{5-8}$$

则

$$X=[X]_{补}-2X_0=X_0.X_1X_2\cdots X_n-2X_0=-2X_0+X_0+0.X_1X_2\cdots X_n$$

$$=-X_0+\sum_{i=1}^{n}X_i\times 2^{-i}$$

当 $X\geqslant 0$ 时，

$$X_0=0,X=-0+\sum_{i=1}^{n}X_i\times 2^{-i}=0.X_1X_2\cdots X_n$$

当 $X<0$ 时，

$$X_0=1$$

$$X=-1+\sum_{i=1}^{n}X_i\times 2^{-i}=-1+0.X_1X_2\cdots X_n=-(1-0.X_1X_2\cdots X_n)$$

$$=-(0.11\cdots1-0.X_1X_2\cdots X_n+0.00\cdots01)=-0.\overline{X}_1\overline{X}_2\cdots\overline{X}_n+0.00\cdots01$$

此关系式在进行补码运算公式推导时会经常用到。

3. 反码表示法

反码表示法在表示负数时将真值按位取反，因此称为“反码”。

1) 小数的反码表示

如果仍用 X 表示真值，则定点小数的反码可定义为

$$[X]_{反}=\begin{cases}X & 0\leqslant X<1\\ (2-2^{-n})+X=(2-2^{-n})-|X| & -1<X\leqslant 0\end{cases}\quad(\text{Mod } 2-2^{-n}) \tag{5-9}$$

从定义表达式可以看出，反码表示法也是一种基于“模”的机器数表示法，但其“模”比补码的模小 1LSB，不是一个整数，即

$(2-2^{-n})=10.00\cdots00-0.00\cdots01=1.11\cdots11+0.00\cdots01-0.00\cdots01=1.11\cdots11$

由上式可看出，反码的模为一串“1”的形式，也称为 1 的补。当需要求负数反码时，据其定义有：

$$[X]_{反}=(2-2^{-n})+X=(2-2^{-n})-|X|$$

$$=1.11\cdots11-0.X_1X_2\cdots X_n=1.\overline{X}_1\overline{X}_2\cdots\overline{X}_n \tag{5-10}$$

即负数反码的转换规则为：符号位置 1，真值的数值部分按位变反。

通过进一步分析我们发现，负数反码和补码之间存在着如下关系：

$$[X]_{补}=[X]_{反}+1\text{LSB} \tag{5-11}$$

即在进行补码转换时可将反码作为中间代码使用，通过反码求补码。

与补码的转换特性类似，式(5-11)给出的转换规则也是双向适用的，也就是当需要把一个负数的反码转换回真值或原码形式时，将其数值部分按位变反即可。注意这个转换过程仍然不包括符号位。

按照定义，反码“0”也有正、负两种表示方法：

$$[+0]_{反}=+0=0.00\cdots0$$

$$[-0]_{反}=(2-2^{-n})+(-0)=1.11\cdots11-0.00\cdots0=1.11\cdots11$$

这个特性也会给机内判 0 带来不便。

反码表示法与补码有许多相似之处，例如同样是模运算，利用反码也可将减法转换为加法，等等。但由于反码的模不是 2 的整幂，因此在发生模溢出（丢掉的是 2 的整幂）时多丢掉了 1LSB，需要对运算结果进行修正，即将丢掉的 1 再加回结果的末位去，此做法称为“循环进位”。这个特点导致反码运算不如补码运算方便，因此反码在计算机中的使用不如补码广泛。

2）整数的反码表示

定点整数反码的形式与定点小数反码完全相同，只是位权关系变了，在表示 n 位整数时模也随之变为 $2^{n+1}-1$。定点整数反码定义如下。

设 X 表示 n 位整数的真值，则

$$[X]_{反}=\begin{cases}X & 0\leqslant X<2^{n}\\ (2^{n+1}-1)+X & -2^{n}<X\leqslant 0\end{cases}\quad(\text{Mod } 2^{n+1}-1)\qquad(5\text{-}12)$$

定点整数反码的运算特性和转换规则也和定点小数反码一样，只是默认的小数点位置不同而已。

4. 三种机器码的比较

在分别对原码、补码和反码进行了讨论之后，我们会发现这三种机器数表示法既有共同点，又有其各自不同的特性。现把它们的异同点归纳如下。

（1）符号位。三种码制相同，最高位都表示符号位，按 0 正 1 负设置。

（2）正数的表示。三种码制相同，符号位置 0，数值部分同真值。

（3）零的表示。原码和反码不唯一，有＋0、－0 之分。补码具有唯一性。

（4）负数的表示。符号位置 1，三种机器码最主要的区别在负数数值部分，表示方法均不相同。

① 原码。数值位同真值，取数的绝对值；

② 反码。数值位为绝对值按位取反；

③ 补码。数值位从右往左逐位寻找第一个 1，这个 1 的左边（高位部分）同反码，右边（包括这个 1 在内的低位部分）同原码。

（5）转换关系。补码与原码、反码与原码之间的转换规则均是双向适用的，三种机器码的符号位均相同，不需转换。真值与机器码的转换在先转换成原码的基础上，把符号位（0、1）再转换成正、负号（＋、－）即可。

（6）表示范围。原码、反码的定义域相同，其正、负域相对零来说是对称的。补码的正、负定义域不对称，正数的表示范围同原码，负数的表示范围较正数多 1 个码点，其最负的数可达－1（定点小数补码），或-2^{n}（定点整数补码），但三种码制的总容量是一样的（n 位可给出 2^{n} 个码点）。

（7）运算特性。补码和反码的运算方法类似，符号位和数值位可作为一个整体看待，一起参加运算；但原码运算时符号位不能参加运算，必须单独进行处理。具体介绍见 5.3 节。

5. 定点表示的特点

通过讨论我们可以看出，无论采用哪一种码制表示定点机器数，都具有如下特点。

（1）表示方法简单，算法也简单，使得定点机结构简单，硬件代价低。

（2）由于计算机的字长有限，定点数的表示范围较小，运算结果容易超出其表示范围，导致溢出出错。

（3）只能表示纯小数或纯整数，需要选择合适的“比例因子”将原始数据变为纯小数或

纯整数形式,比例因子不好选且比较麻烦。

如果想要有效地扩大机器数的表示范围,常用的方法是采用浮点表示法,我们将在5.5节进行深入讨论。

5.3 定点运算

前面我们讨论了定点数常用的表示方法,这一节将在此基础上进一步讨论定点数的算术运算方法及其逻辑实现。通常,运算部件的设计过程分两步完成:首先研制出有效的算法;然后研制与其对应的逻辑实现。算术运算的算法在计算机通常可以用三种方式实现:软件、硬件和固件,或者是这些方法的组合。在这门课程中,运算方法的讨论是作为计算机硬件设计的一个必不可少的中间环节进行的。虽然硬件算法的研制是一项具有开创意义的工作,而算法的逻辑实现可能会涉及更多复杂的工程背景,但是由于我们这门课程是面向初学者的基础课程,因此本书仅原理性地介绍已经成熟的技术,最基本的算法和逻辑实现。计算机中进行定点数的算术运算时,根据其内部采用的机器码形式,可以用原、补、反等各种不同码制的算法进行,不同的算法需要不同配置的运算电路支持,故本书在对每种运算方法进行讨论之后,进一步讨论其所需的硬件实现。但不管采用何种方法进行运算,运算电路的核心部件都是二进制加法器。为了便于运算方法的学习,我们首先来介绍一下基本的加法器电路。

5.3.1 运算部件的基本结构

计算机中采用二进制编码表示数据以后,就可以方便地利用逻辑电路对这些数据进行运算处理。最基本的运算电路虽然已经在数字逻辑课中学习过,但为了更好地理解计算机中实现算术运算的基本原理,并且形成较为完整的机内运算的基本概念,我们仍从一位加法电路开始讨论。

1. 一位二进制加法单元

计算机中最基本的一位二进制加法单元电路是全加器(FA),这种电路可实现两个一位二进制数的相加,以及进位信号的形成和传递。全加器具有3个输入端和2个输出端,包括两个加数输入端 X_i、Y_i 和一个进位输入端 C_i,一个结果输出端 F_i 及一个进位输出端 C_{i+1}。全加器的逻辑符号如图5-5所示。反映全加器输入/输出关系的真值表如表5-3所示。

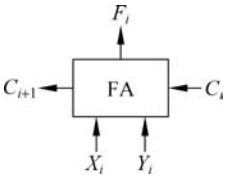


图5-5 全加器的逻辑符号

表5-3 全加器真值表

输入			输出	
X_i	Y_i	C_i	F_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

根据真值表，可得到全加器的典型逻辑表达式：

$$\begin{cases} F_i = X_i \oplus Y_i \oplus C_i \\ C_{i+1} = X_i Y_i + (X_i \oplus Y_i) C_i \text{（或 } C_{i+1} = X_i Y_i + (X_i + Y_i) C_i \text{）} \end{cases} \tag{5-13}$$

这组逻辑表达式反映了全加器的基本组成特点，其逻辑实现方法还有很多种，实际应用时可根据需要转换成“与非”“或非”等其他逻辑形式。

2. *n* 位加法器的基本结构

由于计算机中以机器字长为单位组织数据，因此运算器的核心部件是一个 n 位的二进制加法器。n 位加法器有串行和并行两种基本构成方式。

1）串行加法器

串行加法器的结构很简单，可由一个全加器、一个进位触发器和两个 n 位的移位寄存器 A、B 组成，其中寄存器 A 用作累加器，如图 5-6 所示。

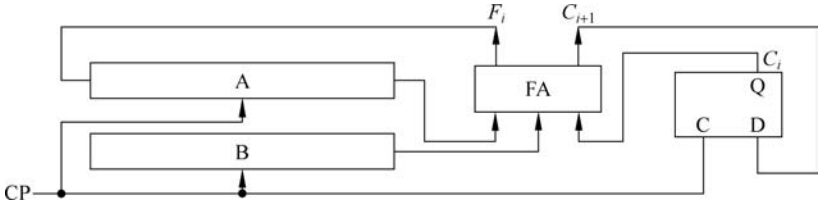


图 5-6　串行加法器

初始时 A、B 中存放着被加数、加数，进位触发器通常初始为 0。加法过程与笔算过程类似，先从最低位开始逐位相加，每加一次 A、B 同时移一位，进位信号保存在触发器中。n 步加法后，累加器 A 中存放着 n 位和，进位触发器中则保留着最高位的进位信号。串行加法器虽然结构非常简单，但加法速度太慢，因此仅用于速度要求不高的简单装置中，计算机中通常并不采用。

2）并行加法器

冯·诺依曼在关于存储程序计算机设想的研究报告中曾经建议，计算机中应采用并行计算方式，即对一个机器字的各位同时进行处理。根据这一思想，目前计算机中均采用并行加法器，一个 n 位的并行加法器可实现 n 位二进制数的同时相加。其最基本的组成逻辑电路如图 5-7 所示。

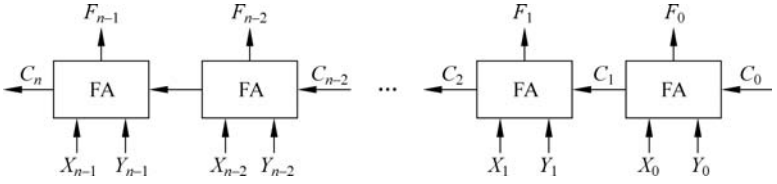


图 5-7　n 位二进制并行加法器

这是一个最基本的并行加法器结构。其显著特点是各位的进位信号由低到高串行传递，连接十分简单。此加法器虽然简单，却可以用作各种算术运算硬件实现的基础。在此基础上我们将进一步讨论计算机中各种算术运算的实现原理。由于使用的普遍性，并行加法器的“并行”两字常被省略。

5.3.2 定点加减运算

计算机中标准的算术运算功能主要包括加、减、乘、除四则运算,有了这四种标准的算术运算,其他所有的数学函数都可以用其组合完成。因此,这四种算术运算在计算机中通常采用硬件实现,同时在计算机指令系统中配有对应的加(ADD)、减(SUB)、乘(MUL)、除(DIV)等算术运算类指令,使用户能通过这些指令调用到硬件的运算功能。而其他的数学函数则可视其复杂程度、开销成本、使用重要性等诸多因素,分别权衡选用软件、硬件或固件方式实现。四则运算中,加、减又是最基本的算术运算,是所有其他运算的基础。计算机中不能没有加减运算,尤其加法运算是任何其他运算所不能替代的。下面我们就来讨论加减运算在计算机中的实现方法。

1. 原码加减运算的基本思想

原码加减运算是指运算前加数和被加数均用原码表示,运算后和也是原码这样一种运算方法,此算法适合在采用原码表示法的计算机中使用。原码表示法虽然简单直观,但其符号位和数值位分开表示,运算时也要分开处理。符号位不参加运算,仅绝对值进行加减,但绝对值到底相加还是相减要根据加/减数的不同符号组合情况决定。如果用 A 表示被加数或被减数的绝对值,B 表示加数或减数的绝对值,S 表示和或差的绝对值,则共有八种判断情况、四种运算组合需要完成,即

$(+A)+(+B)=(+A)-(-B)=+(A+B)=+S$,实际做加法,和为正;

$(-A)+(-B)=(-A)-(+B)=-(A+B)=-S$,实际做加法,和为负;

$(+A)+(-B)=(+A)-(+B)=A-B=\pm S$,实际做减法,够减和为正;不够减和为负,输出为补码形式;

$(-A)+(+B)=(-A)-(-B)=B-A=\pm S$,实际做减法,够减和为正;不够减和为负,输出为补码形式。

由此看来,原码加减运算要预先判断操作数的符号组合,再决定绝对值是加还是减,做减法时情况则更复杂。由于计算机本质上是一个模运算系统,因此不够减时最高位无形中会向上借一个“模”使用,这就不可避免地得到了一个补码差,还需要转换回原码,整个算法很麻烦。因此计算机中很少直接采用原码进行加减运算,即使在原码表示的机器中也是如此,通常都会或多或少地进行变通,比如利用补码来实现原码减法,在此不再做过多的讨论。

2. 补码加法

补码表示法虽不如原码简单直观,但其定义中引入了“模”,这就决定补码比原码更适合计算机内部运算。补码符号位与数值位共同构成一个相对于“模”的编码整体,两者之间存在有机联系,需要一起参加运算。与原码相比,补码加减规则非常简单,易于实现,在计算机中得到了广泛的应用。

1) 补码运算的基本特点

所谓的补码运算是指:

(1) 参与运算的操作数均用补码表示;

(2) 按补码运算规则进行运算;

(3) 补码的符号位与数值位视为一个整体,按同样的规则一起参加运算;

(4) 结果的符号位由运算自动产生;

(5) 运算过程中符号位向高位产生的进位自动丢掉(模溢出)；

(6) 补码运算的结果亦为补码。

2) 补码加法基本公式

补码加法运算规则是以补码加法基本公式的形式给出，即

$$[X+Y]_{补}=[X]_{补}+[Y]_{补} \quad (\text{Mod } M) \tag{5-14}$$

当采用定点小数表示时，$M=2$；当采用定点整数表示时，$M=2^{n+1}$(n 为数值位数)。补码加法基本公式说明，两数的补码相加就可直接得到和的补码。运算过程中不需要单独判断符号位，也不需要区分绝对值做加法还是减法，因此补码加法是计算机中最简单的算法。这也是为什么计算机中普遍采用补码加法的原因。但要注意补码加法运算受“模”的限制，补码加法基本公式仅能保证运算结果不超过补码定义域时是正确的。

补码加法基本公式的正确性可用补码定义加以证明，证明过程按加数的四种组合情况分别进行。以定点小数模 2 补码加法公式为例证明如下：

设 X、Y 为定点小数的真值

(1) 当 $X\geqslant 0$、$Y\geqslant 0$ 时，

$$[X]_{补}+[Y]_{补}=X+Y=[X+Y]_{补} \quad (\text{Mod } 2)$$

(2) 当 $X\geqslant 0$、$Y<0$ 时，

$$[X]_{补}+[Y]_{补}=X+(2+Y)=2+(X+Y)$$

相加结果有两种情况：

$$X+Y\geqslant 0,\text{则 } 2+(X+Y)=X+Y \quad (\text{Mod } 2)$$
$$=[X+Y]_{补} \quad (\text{Mod } 2)$$

此时运算过程中会发生模溢出，丢掉了一个 2。

$$X+Y<0,\text{则 } 2+(X+Y)=[X+Y]_{补}(\text{Mod } 2)$$

此时模 2 与 X、Y 相加共同构成负和的补码。

(3) 当 $X<0$、$Y\geqslant 0$ 时，

$$[X]_{补}+[Y]_{补}=(2+X)+Y=2+(X+Y)$$

证明过程同(2)，有正、负两种可能的结果。

(4) 当 $X<0$、$Y<0$ 时，

$$[X]_{补}+[Y]_{补}=(2+X)+(2+Y)=2+[2+(X+Y)]$$
$$=2+(X+Y) \quad (\text{Mod } 2)$$
$$=[X+Y]_{补} \quad (\text{Mod } 2)$$

此时运算过程中也会发生模溢出，丢掉一个 2。

综合上述四种情况，均证明补码加法基本公式是正确的。

3. 补码减法

1) 补码减法基本公式

与加法类似，补码减法运算规则也由补码减法基本公式给出，即

$$[X-Y]_{补}=[X]_{补}-[Y]_{补} \quad (\text{Mod } M) \tag{5-15}$$

此公式说明，两数的补码相减就可直接得到差的补码。补码减法公式的正确性也能用补码定义进行证明，证明方法与加法相同，在此不再赘述。

虽然通过补码减法公式可方便地进行补码减法运算，但实现时需要用到减法器电路。在机内已经具有加法器的情况下，再专设一个减法器通常认为是没有必要的，因此上述公式

只具有理论上的意义。由于

$$X-Y=X+(-Y)$$

则利用补码加法公式可把减法公式改写成：

$$[X-Y]_{补}=[X+(-Y)]_{补}=[X]_{补}+[-Y]_{补}\quad(\text{Mod } M)\tag{5-16}$$

式(5-16)才是计算机中实际采用的补码减法公式的形式。与加法运算相同，补码减法公式也仅保证运算结果不超过补码定义域时是正确的。

2) 求补运算

进一步对上述两种减法公式进行比较，我们可以得到下述关系：

$$\begin{cases}[X]_{补}+[-Y]_{补}=[X]_{补}-[Y]_{补}\quad(\text{Mod } M)\\ [-Y]_{补}=-[Y]_{补}\quad(\text{Mod } M)\end{cases}\tag{5-17}$$

$[-Y]_{补}$称为$[Y]_{补}$的机器负数，可看成是$[Y]_{补}$的“补”，记作

$$[-Y]_{补}=[\,[Y]_{补}\,]_{补}$$

现在式(5-16)表明，补码减法可以通过加上减数相反数的补码转换成加法进行。这样，机器中只要设一套加法器电路，就可以实现加、减两种运算，大大简化了运算部件的结构。因此我们现在看到的通用计算机中的运算器，其核心部分均只有一个加法器而已。

上述算法实现时有一个关键问题必须解决，那就是如何通过$[Y]_{补}$求$[-Y]_{补}$。我们通常把这个过程看成是一种特殊的运算，叫作“求补”或“变补”。求补运算的规则为：将$[Y]_{补}$连同符号位一起变反，末位加1。在定义了求补运算后，我们可以把一次补码减法过程看成是由一次求补和一次加法联合完成的。特别需要注意，求补运算和负数补码转换操作的区别，求负数补码时符号位的取值是确定的“1”。而求补运算则要连同符号位一起进行。求补运算经常用到，在计算机指令系统中往往设置有相应的求补指令(NEG)。

求补规则的正确性可以定点小数为例进行证明，分两种情况考虑：

设 $[Y]_{补}=Y_0.Y_1Y_2\cdots Y_n$，其中 Y_0 表示符号位

(1) 当 $0\leqslant Y<1$ 时，

$$Y_0=0,[Y]_{补}=[Y]_{原}=Y=0.Y_1Y_2\cdots Y_n,$$

则$[-Y]_{原}=1.\ Y_1Y_2\cdots Y_n$，根据负数原码转换成补码的规则得

$$[-Y]_{补}=1.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB}=\overline{Y}_0.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB}$$

(2) 当 $-1\leqslant Y<0$ 时，$Y_0=1$

$$[Y]_{补}=1.Y_1Y_2\cdots Y_n,$$

根据负数补码转换成原码的规则得

$$[Y]_{原}=1.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB}$$

则

$$[-Y]_{原}=0.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB},$$

根据正数原码转换成补码的规则得

$$[-Y]_{补}=[-Y]_{原}=0.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB}=\overline{Y}_0.\overline{Y}_1\overline{Y}_2\cdots\overline{Y}_n+1\text{LSB}$$

综上所述，不管真值 Y 为正还是为负，均可得出“$[-Y]_{补}$等于$[Y]_{补}$连同符号位一起变反，末位加1”的结论。求补运算可以通过在加法器的一个输入端添加一组异或门电路来简单地实现，具体形式如图5-8所示。增加了求补电路的加法器就变成了加减法器，既可做加法，也可做减法。但是具体做哪种运算，需要在异或门的输入端设置加/减控制信号 M 进行选择。

图5-8的加减法器电路支持双符号位的变形补码加减运算。

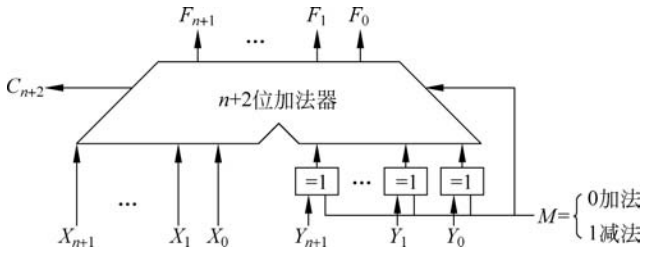


图 5-8　二进制补码加减法器

4. 运算结果的溢出判断

前面讨论补码加减算法时曾提到：补码加减运算基本公式受补码“模”的限制，仅保证运算结果不超出其定义域时是正确的。这就是说，补码加减运算的结果如果超出补码表示范围时会出现错误，计算机中通常把这种错误称为“溢出”(Overflow)。溢出出错产生的根本原因还是由于计算机的字长限制。如果运算过程中出现溢出，再继续运算下去就没有意义了。因此在每步运算完成之后，都要及时判断是否有溢出。如果没有溢出运算可以继续进行；一旦发现溢出则要立即中止运算，转到专门的溢出处理程序。在错误结果得到修正之后，才能重新返回原来的运算过程继续运算。

定点运算的溢出分为正溢出和负溢出两种。当运算结果大于可表示的最大正数时，称为“正溢出”；小于可表示的最小负数时，称为“负溢出”。例如：

$$[0.1101100+0.1011011]_{补}=0.1101100+0.1011011=1.1000111$$

$$[-0.1101100\text{-}0.1011011]_{补}=1.0010100+1.0100101=0.0111001$$

可以发现，溢出出错通常发生在同号两数相加，以及异号两数相减的运算过程中，错误都表现在符号位被反相了。计算机中正是利用这些特征，实现对溢出出错情况的判别。补码加减运算常用的溢出判断方法有以下三种。

1) 根据符号位之间的关系判别

采用补码进行加减运算时，如果同号两数相加，结果的符号位变反了，就可以断定发生了溢出。即出现了“正＋正得负”；或者“负＋负得正”的现象。类似地，如果异号两数相减，结果的符号位与减数相同，也可以断定发生了溢出。这个规律可以用逻辑表达式进行描述。

仍以定点小数补码为例，设$[X]_{补}=X_0.X_1X_2\cdots X_n$，$[Y]_{补}=Y_0.Y_1Y_2\cdots Y_n$，$[F]_{补}=[X]_{补}\pm[Y]_{补}=F_0.F_1F_2\cdots F_n$，其中 X_0、Y_0、F_0 表示补码的符号位；M 为加减控制信号，$M=0$ 表示做加法；$M=1$ 表示做减法。V 表示判别溢出信号，$V=0$ 表示无溢出；$V=1$ 表示溢出，则溢出判别逻辑表达式可归纳如下：

$$V=(\overline{X}_0\overline{Y}_0F_0+X_0Y_0\overline{F}_0)\overline{M}+(\overline{X}_0Y_0F_0+X_0\overline{Y}_0\overline{F}_0)M \tag{5-18}$$

这种方法适用于单符号位补码加减运算时的溢出判别，但需要较复杂的逻辑电路支持。

2) 利用进位信号之间的关系判别

如果进一步对补码加减运算过程进行仔细分析，会发现运算溢出时所产生的进位信号有这样一个规律：当最高数值位(MSB)有向符号位的进位时，符号位不会再产生向更高位的进位；当最高数值位无向符号位的进位时，符号位会产生向更高位的进位。即溢出时这两个相邻的进位信号不会同时出现。设补码加减运算时最高数值位向符号位的进位为 C_{MSB}，符号位向更高位的进位输出为 C_s，则溢出判别逻辑可描述为

$$V = C_s \oplus C_{\text{MSB}} \tag{5-19}$$

这种方法也适用于单符号位补码加减运算时的溢出判别,但与方法 1)相比,所需的逻辑电路要简单得多。

3) 采用双符号位补码判别

在讨论补码表示法时曾经提到变形补码的概念,如果使用双符号位的变形补码进行加减运算,不仅可以很方便地判断溢出,而且还可以避免结果的符号位遭到破坏,因此得到了广泛的应用。双符号位补码在运算前,操作数的两个符号位被定义成相同的,即正数时为 00,负数时为 11。运算后得到的结果也具有两个符号位,这两个符号位在变形补码中所起的作用是不同的。为了便于区别,通常将最高符号位定义为"真符"位,也叫第一符号位,记作 F_{s1}。F_{s1} 之所以叫作真符位,是因为不管是否发生溢出,这个符号位在加减运算后始终保持着结果的正确符号。也就是说,定点小数运算的溢出值只能在 $-2 \sim 2$ 之间,不可能超过 2,因此溢出的数值永远不会破坏到真符位。另一符号位则被叫作第二符号位,记作 F_{s2}。相对于 F_{s1} 而言,F_{s2} 起着辅助的作用。当运算发生溢出后,破坏的是这个符号位。因此运算后,若结果的两个符号位 $F_{s1}F_{s2}$ 仍然相同,表示没有发生溢出;若结果的两个符号位相异,则表示发生了溢出。若 $F_{s1}F_{s2}$ 变成 01,表示发生了正溢出;若 $F_{s1}F_{s2}$ 变成 10,表示发生了负溢出。根据这个变化规律,可以得到使用变形补码判溢出的逻辑表达式:

$$V = F_{s1} \oplus F_{s2} \tag{5-20}$$

与前面讨论的两种溢出判断方法相比,此方法显然更加简单易实现。例如:

$[0.101101 - 0.101101]_{补} = 00.101101 + 11.001010 = 11.110111$,无溢出

$[0.101101 + 0.101101]_{补} = 00.101101 + 00.110110 = 01.100011$,正溢出

$[-0.101101 - 0.110110]_{补} = 11.010011 + 11.001010 = 10.011101$,负溢出

5. 补码加减运算规则

综上所述,我们可以总结出补码加减运算的一般规则。以定点小数变形补码运算为例,模 4 补码加减运算规则如下。

(1) 参加运算的两个操作数均采用双符号位补码。

(2) 操作数的两个符号位与数值位作为一个整体一起参加运算。

(3) 若做加法,两数补码直接相加;若做减法,减数求补后,再与被减数相加。

(4) 运算中产生的模溢出进位信号自动丢掉,不影响运算的正确性。

(5) 结果仍为双符号位补码,其两个符号位均由运算自动产生。

(6) 若结果的两个符号位相同,无溢出,运算结束;若结果的两个符号位相异,有溢出,转溢出处理。最高符号位为结果的正确符号。

定点整数补码加减运算方法与定点小数补码加减运算基本相同,只是补码的模变了,小数点默认在最低数值位(LSB)之后(右边)。

6. 补码加减运算的硬件配置

支持变形补码加减运算的硬件电路框图如图 5-9 所示。

对于 n 位的定点数而言,若采用双符号位补码进行运算,运算器的总位数应达到 $n+2$ 位。由图 5-9 可以看出,补码加减运算的核心部件是二进制并行加法器(见图 5-7)。在加法器的一个输入端配置求补电路,就构成了加减法器(见图 5-8)。为了在运算过程中提供给加减法器稳定可靠的输入,需要配置两个寄存器 A、B。运算前先把两个操作数存放在 A、B 中,然后再进行加减运算。由于原始操作数运算完成后往往就不再需要了,因此可把其中的

一个寄存器 A 设为累加器，用来存放运算结果。这样，补码加减运算器可看成主要由一个加减法器和两个寄存器组成，在此基础上再添加溢出判别、操作控制等辅助逻辑，就构成完整的补码加减运算器电路。

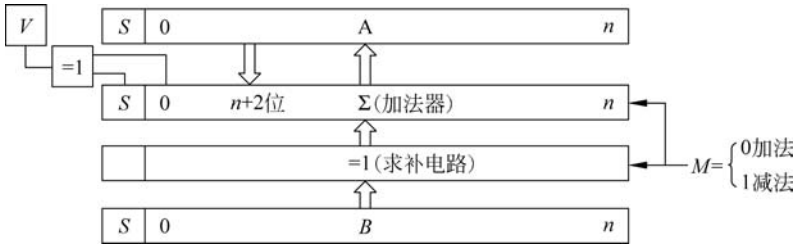


图 5-9　补码加减法运算器

5.3.3　移位运算

1. 什么叫移位运算

计算机中经常需要移动机器数的数位，例如把一个二进制数的各位统统左移一位或若干位，或者右移等，这在计算机中被称作移位操作。因为机器数左移一位相当于乘以 2，右移一位相当于除以 2。基于这一点，我们也把移位操作看成是计算机中的一种特殊的乘除子运算。计算机中有很多场合需要用到移位运算，像利用移位来进行某些测试操作等。因此，不同的计算机中都或多或少地安排有专门的移位类指令。为了便于描述，我们在本书中经常会用符号“→n”表示右移 n 位的操作，其中 $n=1,2,\cdots$，表示移动的位数；同样，左移 n 位的操作用符号“←n”来表示。

2. 移位运算的规则

具体地说，移位操作又分为逻辑移位、算术移位和循环移位三大类。逻辑移位是指移位过程中不特别考虑数据符号位含义的一类移位，包括逻辑左移、逻辑右移，通常用于对无符号数或逻辑数进行移位；循环移位是指机器字首尾相接的移位操作，也分循环左移和循环右移等，通常用来满足一些特殊的需求，如位测试等功能；算术移位则是对有符号数进行的移位，包括算术左移和算术右移，在移位的过程中需要特别注意保持符号位不发生改变。由于操作性质的不同，这三类移位的规则也是不一样的，其区别主要涉及移位后空出位的处理方式上，现分述如下。

1）逻辑移位规则

当机器数左移 n 位或右移 n 位时，必然会使其低 n 位或高 n 位出现空位。由于计算机中机器数的字长往往是固定的，整数高位部分的 0 和小数低位部分的 0 是无法省略不要的，故必须对空出位进行处理。逻辑移位时采取的措施是，不管逻辑左移还是逻辑右移，对移位后的空出位一律补 0；而对于移出位，不管是 1 还是 0，由于超出了机器字长可表示的范围，则一律丢掉。即逻辑移位规则：逻辑左移时，高位移丢，低位补 0；逻辑右移时，低位移丢，高位补 0。逻辑移位最大的特点是移位时不考虑数据的符号位，因此只适用于无符号数或逻辑数。图 5-10 给出了逻辑移位操作示意。在机器指令系统实现时，为了便于判断移出位的值，往往先将移出位保存在进位标志位 CF 中。图中箭头表示移位方向，OPR 表示操作数。

2）循环移位规则

循环移位将机器字的首尾相接进行移位，其移位通路构成了一个封闭的环路。不管是

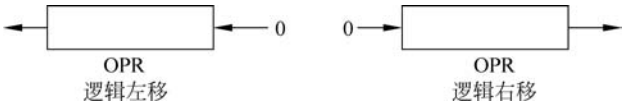


图 5-10　逻辑移位

循环左移还是循环右移,其移出位都会自动填补到空出位中,只是填补的方向不同而已。循环左移时最高位移出填入最低位,而循环右移时最低位移出填入最高位。图 5-11 对循环移位操作进行了示意。实际上计算机中实现循环移位时,往往将移出位自动填补到空出位中的同时,也填补到进位标志位 CF 中,以便需要时能够对移出位方便地进行检测。

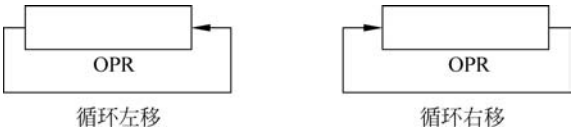


图 5-11　循环移位

另外,还有一种带进位的循环移位,将进位标志位加入循环移位的回路中,可以实现双字长移位时移出位在两个寄存器之间首尾相接的传递。当然,这种操作也可用于其他一些用途,像对机器字中的某一位进行位操作等。图 5-12 示出了带进位循环移位实现双字长移位的过程。

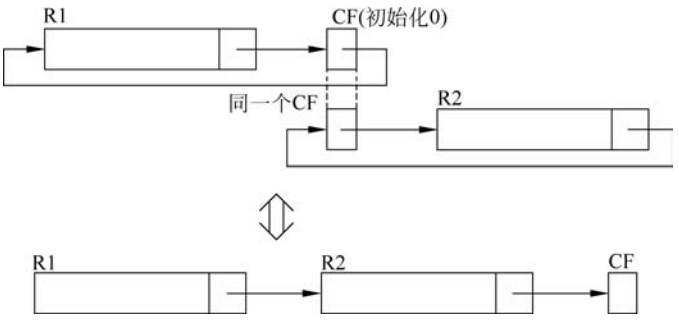


图 5-12　带进位循环移位

3) 算术移位规则

算术移位具体又分为算术左移和算术右移两种。与逻辑移位不同,算术移位的基本规则是要保证移位后符号位取值不变。不仅如此,由于算术移位是针对有符号数的移位操作,因此对采用不同表示法的机器码来说,移位后空出位的填补值要符合机器码编码规则,保证所用码制的正确性。现将常用的原码、补码、反码的算术移位空出位填补规则归纳在表 5-4 中。

表 5-4　原码、补码、反码的算术移位规则

类别	码 制	符号位	右移时 MSB 填补值	左移时 LSB 填补值
正数	原码、补码、反码	0	0	0
负数	原 码	1	0	0
	补 码	1	1	0
	反 码	1	1	1

由表 5-4 可以看出算术移位的规则如下。

(1) 正数：由于正数的原、补、反码表示形式均一样，因此不论是算术左移还是右移，在保证符号位不变的前提下，左、右空出位均补 0。

对于负数，由于原、补、反码的表示形式均不一样，故当机器数移位时，对其空出位的填补规则也各不相同，需要分别考虑。

(2) 负数原码：负数原码的数值部分与正数相同，故移位规则与正数原码完全一致。即符号位不变，不论是左移还是右移，左、右空出位均补 0。

(3) 负数反码：移位规则与原码完全相反，即符号位不变，不论是左移还是右移，左、右空出位均补 1。这是由于其取值与原码正好相反的缘故。

(4) 负数补码：符号位不变，左移时，LSB 位空出补 0，同原码；右移时，MSB 位空出补 1，同反码。这和补码的结构特点是一致的。

表 5-4 给出的算术移位填补规则既适用于定点小数，也适用于定点整数。

与上述规则对应的算术左移和右移操作的硬件示意框图如图 5-13 所示。其中，图 5-13(a)是三种机器码为正数时的移位操作；图 5-13(b)为负数原码的移位操作；图 5-13(c)为负数补码的移位操作；图 5-13(d)为负数反码的移位操作。

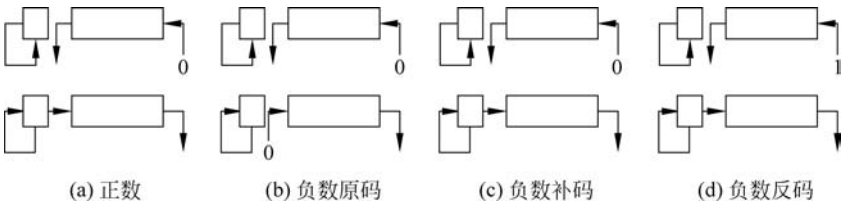


图 5-13　算术左移和右移操作的硬件实现框图

算术移位在实际机器的指令系统中实现时，往往也采用带进位的移位方式，不管是左移还是右移，均先将移出位移入进位标志位中保存起来，以便进一步判断移出位的值，及时发现溢出等错误是否发生。

4) 补码算术移位时的符号延伸(扩展)特性

在对补码进行右移时我们会发现，不论是正数还是负数，右移后对最高数值位(MSB)的填充值都等于其符号位的值，相当于右移时符号位在自身保持不变的前提下，同时移入空出的 MSB 位。这个特性称为补码的符号延伸特性。根据这个特性，我们可以得到如下补码右移公式。

若

$$[X]_{补}=X_0X_1X_2\cdots X_n \quad (\text{Mod } 2 \text{ 或 Mod } 2^{n+1})$$

则

$$\begin{aligned}
\left[\frac{1}{2}X\right]_{补}&=X_0X_0X_1\cdots X_{n-1} \quad X_n \text{ 移出丢掉}\\
\left[\frac{1}{4}X\right]_{补}&=X_0X_0X_0X_1\cdots X_{n-2} \quad X_{n-1}X_n \text{ 移出丢掉}\\
\left[\frac{1}{8}X\right]_{补}&=X_0X_0X_0X_0X_1\cdots X_{n-3} \quad X_{n-2}X_{n-1}X_n \text{ 移出丢掉}\\
&\cdots
\end{aligned} \tag{5-21}$$

这个补码的符号填充规则可一直延伸到所需右移的位数为止。此规则可以用补码和真

值的转换公式进行证明。现以定点小数补码为例证明如下：

求证：若

$$[X]_{补}=X_0.X_1X_2\cdots X_n \quad (\text{Mod } 2)$$

则

$$\left[\frac{1}{2}X\right]_{补}=X_0.X_0X_1\cdots X_{n-1}$$

证明：因为

$$[X]_{补}=X_0.X_1X_2\cdots X_n \quad (\text{Mod } 2)$$

所以

$$X=-X_0+\sum_{i=1}^{n}X_i\times 2^{-i}$$

$$\frac{1}{2}X=-\frac{1}{2}X_0+\frac{1}{2}\sum_{i=1}^{n}X_i\times 2^{-i}=-X_0+\sum_{i=0}^{n}X_i\times 2^{-(i+1)}$$

则在字长不变的条件下

$$\left[\frac{1}{2}X\right]_{补}=X_0.X_0X_1\cdots X_{n-1}$$

使用相同的方法，还可以进一步证出 $\left[\frac{1}{4}X\right]_{补}$、$\left[\frac{1}{8}X\right]_{补}$、…的符号延伸规则，在此不再赘述。

补码的符号延伸特性也可以反过来用于整数补码位数的扩展。例如，将 16 位补码扩展为 32 位补码时，低 16 位取值可保持不变，高 16 位用符号位填充即可(正数为全 0，负数为全 1)。注意 32 位补码的符号位设在 32 位数的最高位，而原来 16 位补码符号位(即 16 位数的最高位)的位置现在变成了扩展后的数值位，不再代表补码符号。该特性也常常被称为补码的符号扩展特性。补码的符号延伸和符号扩展操作如图 5-14 所示。

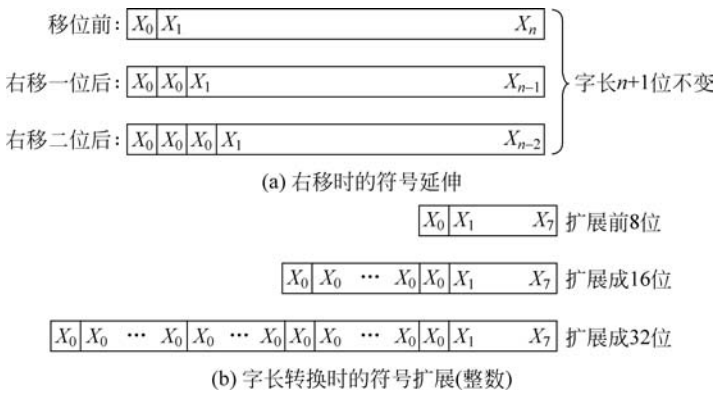


图 5-14　补码的符号延伸和扩展

5) 算术移位的误差及溢出

由于算术移位时操作数的高、低两端有数位移出，因此可能有误差或溢出情况发生。现在我们关心的是如果出现这类情况，将会对移位运算的结果产生什么影响？下面就这一问题进行分析。

对于正数来说，由于原码、补码、反码的表示形式一致，移位操作一致，因此移位后对数

值的影响也是一样的。右移时，如果最低数位丢 1，会产生一定的误差，但对精度的影响有限，不会破坏结果的正确性。左移时，如果最高数位丢 1，会导致结果溢出出错，需要及时进行溢出处理。

对于负数来说，由于原码、补码、反码的表示形式不再一样，故移位后对数值的影响形式也各不相同，现分述如下。

负数原码：右移时，低位丢 1，产生误差影响精度，但结果仍然正确；
左移时，高位丢 1，结果溢出出错。

负数反码：右移时，低位丢 0，产生误差影响精度，但结果仍然正确；
左移时，高位丢 0，结果溢出出错。

负数补码：右移时，低位丢 1，产生误差影响精度，但结果仍然正确；
左移时，高位丢 0，结果溢出出错。

3. 误差的舍入处理方法

通过前面的讨论我们已经看到，算术右移时由于受到机器字长的限制，最低位会有数位移出。如果任由这些移出位自然丢失，就会造成运算误差。在对运算精度要求较高的情况下，为了尽量缩小这些误差，通常在运算后进行相应的舍入处理。所谓舍入是指当运算结果的位数超出机器可表示的位数时，舍去多余位的同时，并按一定的规则对机器数值进行调整的过程。

计算机中采用的舍入方法有许多种，但不管哪种方法，硬件上往往都需要在有效数据字长之外多设置若干位保护位，先把运算过程中右移出的前几位数暂时保存起来，以供舍入判断之用。保护位一般通过增设保护位寄存器实现。常用的舍入方法有如下几种。

1）截断法

截断法也叫截尾，这是一种实现起来最简单的舍入方法。操作方式：当运算结果超出机器字长时，无条件地丢掉超出的位数，只留下正常字长部分，且保留下来的数值不作任何改变。采用截断法引起的误差为单向误差，也就是每次舍入后可能产生的误差都是负误差（对结果精确值起减小作用）。单次误差小于－1LSB。虽然截断法的单次误差并不算大，但是由于是单向误差，经过多次运算后的累积误差可能会很大，所以对运算结果的精度影响较大。

2）末位恒置 1 法

末位恒置 1 法在舍去结果最低位之后超出的数值位的同时，将机器数末位（即 LSB 位）置 1。这种舍入法引入的误差为双向误差，当机器数末位为 0 时，置 1 后可能产生正误差；当机器数末位本来就为 1 时，则可能产生负误差。单次误差小于±1LSB。虽然恒置 1 法单次误差的大小与截断法相同，但是由于是双向误差，经过多次运算后基本上不会产生累积误差，因此对运算结果的精度影响比截断法小得多。恒置 1 法在具体实施时为了进一步减小误差，常采用更加细致的处理方式：当机器数最低位为 1，或移出的位中有 1 时，末位置 1；否则舍去移出位，不做末位置 1 的操作。

3）0 舍 1 入法

0 舍 1 入法相当于十进制计算中的四舍五入法。其基本思想：用将要舍去部分的最高位作为判断标志，以决定做何种舍入操作。如果该位为 0，则“舍”——所做操作同截断法，无条件地丢掉机器数超出字长部分的位数，且保留下来的数值不做任何改变；如果该位为 1，则“入”——在丢掉超出的位数后，对保留部分的最低位加 1。这种舍入法引入的也是双向误差，但单次误差小于±1/2LSB，显然比前两种舍入方法的单次误差小一半，且没有累积

误差,因此精确度最高。不过0舍1入法的操作规则比前两种方法复杂,当用于具体的机器中时,其舍入规则还要考虑所采用的码制。

若机器数用原码表示或者是补码正数时,上述0舍1入法的规则可直接应用。此时"舍"使数据的绝对值变小;"入"使数据的绝对值变大。

但机器数是用补码表示的负数时,0舍1入法的规则需要根据负数补码的编码特点作相应的改变,即若超出部分的最高位为0时舍去;若超出部分的最高位为1,其余各位全为0时舍去;若超出部分的最高位为1,其余各位不全为0时,则在舍去超出部分之后,保留部分的末位+1,作"入"的操作。此时,舍入操作对数据精度的影响与补码正数或原码表示时是相反的,"舍"操作使数据的绝对值变大;而"入"操作则使数据的绝对值变小。

4) 查表舍入法

查表舍入法又称ROM舍入法,因为它用ROM来存放舍入处理表。这种舍入方法可看成是0舍1入法的一种改进方案。通过前面的讨论我们已经看到,0舍1入法虽然具有误差小、精度高的优点,但在舍入时有可能需要多执行一次加1操作,尤其是在浮点运算过程中,这个额外的加1操作可能会引起一系列的连锁反应(详见5.5节),使运算过程复杂化,运算时间延长。查表舍入法则将0舍1入的结果直接保存在ROM单元中,舍入时经过查表就可读得相应的处理结果,不需再做加1运算,显然具有舍入误差小和执行速度快的优点。

4. 移位运算的硬件实现

计算机中实现移位运算通常有以下方法。

1) 通过移位寄存器进行

这种方法使用具有移位功能的寄存器部件实现移位,每次只能移一位,当左移或右移n位时,需要n个时钟周期才能完成。移位速度慢,且移位和加减运算不能同时进行。

2) 数据通路中设置移位器部件

许多计算机CPU的数据通路中,在加减法器的输出端设置一个移位器来完成移位功能。移位器电路实际上相当于一个组合逻辑的多路选择器,三选一的多路选择器可实现将运算结果按位直送、左斜一位传送(左移一位)、右斜一位传送(右移一位)的移位器功能;而五选一的多路选择器可在此基础上再加两路功能,即左斜两位传送(左移二位)、右斜两位传送(右移二位)。五选一移位器的第i位逻辑电路如图5-15所示,其他位的电路实现与此类似。

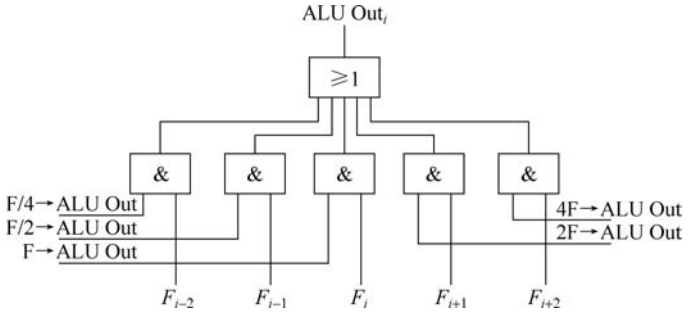


图5-15 五选一移位器第i位的逻辑电路

图5-15中的位序按从左到右依次为0～n。据此思路还可组成更多位同时移位的移位器电路。与移位寄存器相比,组合逻辑的移位器具有多位并行移位,移位操作和加减运算在同一个时钟周期内完成的优点,广泛用于各种计算机的运算器中。

3）桶形移位器

目前微处理器芯片内部的数据通路中，经常可以看到一种叫作“桶形移位器”（Barrel Shifter）的移位部件，这是一种并行度极高的快速移位器，可以实现在数据宽度范围内将一个数据移动任意位的需求。例如，当机器字长 32 位时，一个 32 位的桶形移位器既可实现数据左移一位、右移一位的操作，也可以实现左移 10 位、右移 10 位，或左移 32 位、右移 32 位的操作。

虽然功能强大，但桶形移位器实现原理并不难，仍然基于组合逻辑多路选择器的使用。例如，上述 32 位桶形移位器的设计，可用一个 32 选 1 多路选择器实现将 32 位数据中任意一位移到一个特定位的操作（包括不移位直送），这样 32 位字长共需要 32 个 32 选 1 的多路选择器。具体实现时，由于 32 选 1 规模的多路选择器并不常见，可用多个较小规模的多路选择器多级连接构成。

5.3.4 定点乘法运算

乘法也是最基本的算术四则运算之一，是其他复杂运算的基础。但与加减运算不同，乘法运算虽然也很重要，但在计算机中并不一定非要用硬件直接实现。一般运算器中都少不了加法器，但却可以没有乘法器。这是因为乘法在计算机中是能够替代实现的。在实际的计算机中，有些由硬件乘法器直接完成乘法运算；有些虽然没有乘法器，但乘法指令由硬件执行特定的算法将乘法自动转换为加法-一位操作完成；当然，也有些功能简单的计算机中不设置乘法指令，则可用加法-一位指令通过软件编程来实现。不管是硬件实现还是软件实现，前提都离不了某种乘法算法的支持。在此，我们将原理性地介绍计算机硬件实现乘法时常用的运算方法，以及所需的硬件配置。

1. 原码一位乘法

1）算法推导

与加减运算情况一样，计算机中采用不同的码制需要不同的乘法算法支持。由于原码表示与真值极为相似，其区别只在一个符号位，因此原码乘法与笔算过程也最为接近。作为一种最基本的算法，我们首先讨论原码一位乘法的原理。原码一位乘法直接从笔算过程演变而来，为便于理解，我们先从分析笔算乘法入手，看一下计算机中实现乘法需要解决哪些基本问题。

笔算乘法的规则：两数的绝对值相乘得乘积的绝对值，乘积的符号按“同号相乘得正，异号相乘得负”的规则产生。下面通过一个例子对笔算乘法的具体过程进行分析。为了方便起见，我们对乘法算法的讨论均以定点小数为例。例如，$X=0.1101$，$Y=-0.1011$，则

$X\times Y=-(0.1101\times 0.1011)=-0.10001111$，其笔算乘法竖式如下：

$$
\begin{array}{rl}
0.1101 & \cdots\text{被乘数 } X \\
\times\ \ 0.1011 & \cdots\text{乘数 } Y \\
\hline
1101 & \cdots\text{部分积}_1 \\
1101 & \cdots\text{部分积}_2 \\
0000 & \cdots\text{部分积}_3 \\
+\ \ 1101 & \cdots\text{部分积}_4 \\
\hline
0.10001111 & \cdots\text{乘积 } P
\end{array}
$$

这个竖式反映了笔算乘法的每一步细节，概括起来有如下几点。

(1) 从乘数最低位开始,用一位乘数去乘被乘数,得到一次积(部分积)。

(2) 继续从低位到高位用乘数的每一位去乘被乘数,在 n 位乘数数值的情况下,可以依次得到 n 个部分积。上例中 $n=4$,最后乘得 4 个部分积。

(3) 将所得部分积左斜一位对位相加,积的绝对值位数增长了一倍。上例中两个 4 位的小数相乘,最后得到的乘积为 8 位小数。

现在考虑如果让计算机完全模仿笔算乘法步骤进行计算,将会遇到什么问题?对笔算步骤进行分析后发现,上述算法概括中的第 1、2 两点在计算机中实现都没有困难,正因为每次用乘数的一位去乘被乘数,所以该算法被称叫作"一位乘法"。问题集中在第三点上,机内实现起来有两大困难:其一,常规的加法器只有两个数据输入端,每次只能实现两个数的相加,很难实现多个数同时相加;其二,加法器的位数一般与机器字长相同,很难实现两倍字长的加法。

所以在计算机中实现原码乘法时,不能直接照搬笔算方法,必须进行算法改进。针对上面提出的两个问题,改进思路如下。

(1) 改 n 个部分积同时相加过程为"两两相加—n 次累加"。

(2) 改部分积 $2n$ 位的相加过程为"n 位相加—n 次右移"。由于部分积左斜一位对位相加意味着部分积的最低位并不参加加法运算,故可通过右移一位操作将其从 n 位中移出,使得每次加法只在部分积的高 n 位进行。这样,就把 $2n$ 位的相加过程变成了 n 位的相加过程。

这两点改进的综合效果就是把计算机中 n 位乘法过程转化成了 n 次"加法和移位"操作,使得利用常规的加法器和移位部件就能实现乘法。

至于乘积符号位的产生,则可以根据"同号相乘得正,异号相乘得负"的规则,单独通过对两数的符号位进行逻辑异或求得。

综上所述,定点小数原码一位乘法可描述为:

设被乘数$[X]_{原}=X_0.X_1X_2\cdots X_n$,乘数$[Y]_{原}=Y_0.Y_1Y_2\cdots Y_n$,则

乘积$[P]_{原}=[X\times Y]_{原}=(X_0\oplus Y_0).(X^*\times Y^*)=P_0.P_1P_2\cdots P_nP_{n+1}\cdots P_{2n}$

其中,$X^*=0.X_1X_2\cdots X_n$,$Y^*=0.Y_1Y_2\cdots Y_n$,分别表示 X 和 Y 的绝对值;X_0、Y_0 和 P_0 分别表示$[X]_{原}$、$[Y]_{原}$和$[P]_{原}$的符号位。

乘积的数值部分由两数绝对值相乘而得,其通式为

$$\begin{aligned}X^*\times Y^* &= X^*\times(0.Y_1Y_2\cdots Y_n)=X^*\times(Y_12^{-1}+Y_22^{-2}+\cdots+Y_n2^{-n})\\ &=2^{-1}(Y_1\times X^*+2^{-1}(Y_2\times X^*+2^{-1}(\cdots+2^{-1}(Y_{n-1}\times X^*+\\ &\quad 2^{-1}(Y_n\times X^*+0))\cdots)))\end{aligned}\tag{5-22}$$

令 Z_i 表示第 i 次部分积,则式(5-22)可写成如下递推公式的形式。

$$\begin{cases}Z_0=0\\ Z_1=2^{-1}(Y_n\times X^*+Z_0)\\ Z_2=2^{-1}(Y_{n-1}\times X^*+Z_1)\\ \cdots\\ Z_i=2^{-1}(Y_{n-i+1}\times X^*+Z_{i-1})\\ \cdots\\ Z_n=2^{-1}(Y_1\times X^*+Z_{n-1})\end{cases}\tag{5-23}$$

这组递推公式中，Z_0 表示部分积的初值(=0)，而第 n 步部分积 Z_n 即为最后得到的乘积绝对值

$$P^* = X^* \times Y^* = Z_n$$

2) 原码一位乘运算规则

由上述推导过程可总结出原码一位乘运算规则如下：

(1) 参加运算的两个操作数均为原码，两数符号位不参加运算，取其绝对值做无符号数乘法。

(2) 乘积的符号位单独处理，由两数符号“异或”产生。

(3) 设部分积初值为 0，为防止运算过程中产生暂时性溢出，部分积可采用单符号位参加运算。n 位数相乘，部分积连同符号位应取 $n+1$ 位。

(4) 运算前先检测 0，只要两数任意一个为 0 则乘积为 0，不再进行运算。

(5) 部分积运算，用乘数的最低位作为判断位，若为 1，加被乘数；若为 0，不加(相当于加 0)；运算后部分积逻辑右移一位。

(6) 当乘数数值部分为 n 位时，共作 n 次“加法”和“右移”，最后得到 $2n$ 位的乘积绝对值。

3) 原码一位乘法运算的硬件实现

不难看出上述乘法运算规则很容易实现。通常乘法运算需要三个寄存器、一个加法器和一些辅助电路支持。原码一位乘法运算器框图如图 5-16 所示。

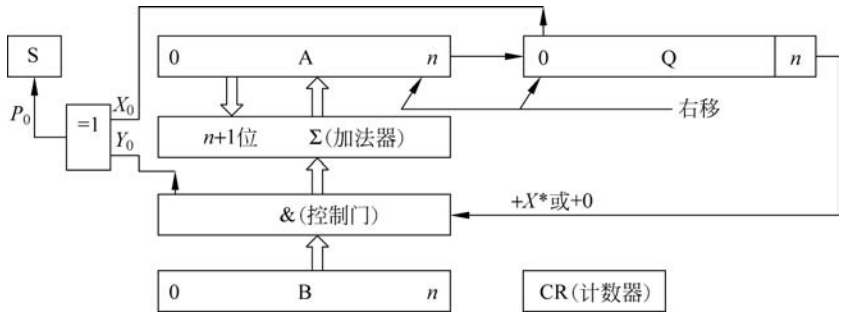


图 5-16　原码一位乘法运算器

由图 5-16 可知，乘法运算器的主体部分由三个寄存器和一个加法器共同组成。部分积寄存器 A 的初值为 0，运算过程中存放部分积的累加值，运算结束后存放乘积的高位部分；寄存器 B 用来存放被乘数；乘数寄存器 Q 在乘法过程中起着关键的作用，Q 最低位的值作为每一步运算的控制信号，控制本步加或者不加被乘数。A、Q 寄存器首尾相接级联在一起，实现部分积和乘数联合右移功能。随着逐次移位的进行，乘数将逐位丢失，Q 寄存器中最终取而代之的是乘积的低 n 位部分。辅助电路主要包括计数器 CR，用来控制加法—右移的次数；乘积符号位产生电路(异或门、符号标志触发器 S)；$n+1$ 个与控制门等。

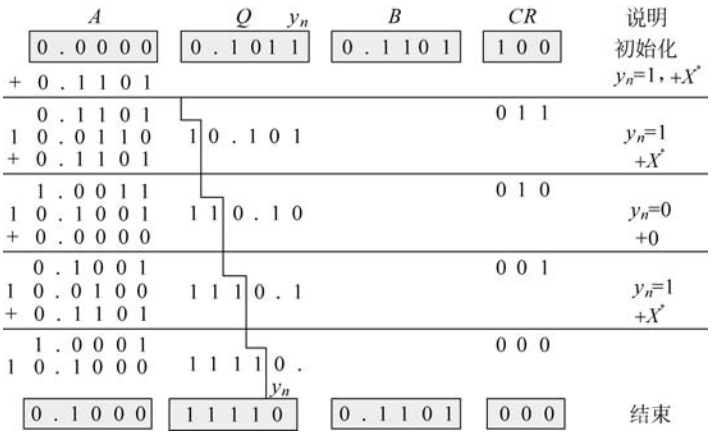
【例 5.1】　$X=0.1101$，$Y=-0.1011$，用原码一位乘法求 $X\times Y$，写出机器运算步骤。

解：

$$[X]_{原}=0.1101,\quad [Y]_{原}=1.1011$$

$$X^*=0.1101\rightarrow \mathrm{B},\quad Y^*=0.1011\rightarrow \mathrm{Q},\quad 0\rightarrow \mathrm{A},\quad 100\rightarrow \mathrm{CR}$$

机器运算步骤如下：



$$P_0 = X_0 \oplus Y_0 = 1 \oplus 0 = 1, \quad P^* = 0.1000\ 11110$$

$$[P]_{原} = [X \times Y]_{原} = 1.1000\ 11110$$

$$X \times Y = -0.1000\ 1111$$

该算法规则同样适用于定点整数，但要把例题中的小数点“.”改为逗号“,”。此时需要注意小数点的隐含位置已和定点小数不一样了，导致乘积最终要向右对齐小数点，这是两种定点数据格式运算时的主要差别。为便于硬件实现，最后一步运算完成后可考虑将乘积右移两位。

2. 补码一位乘法(比较法)

虽然原码乘法算法简单易实现，但因为补码加减法在计算机中得到了普遍应用，因此许多机器都采用补码表示数据，这种情况下若再采用原码乘法就有些不方便，希望能够直接使用补码做乘法。为此，计算机专家们设计出多种补码乘法算法，其中使用较为普遍的是由 Booth 夫妇最先提出来的比较法(也称 Booth 法)。下面将对这种算法进行讨论。

1) 比较法算法推导

我们首先应弄清这样几个问题：补码相乘能否直接套用原码乘法算法？因补码的符号位必须和数值位一起参加运算，而原码乘法特点之一却是符号位不参加运算，故答案显然是否定的。还有，补码相加减能直接得到结果的补码，那么补码相乘是否也能直接得到积的补码呢？如果可以，补码乘法不也很方便吗？这也是我们迫切想要知道的。针对这个问题，我们先来推导一下补码乘积与$[X]_{补}$和$[Y]_{补}$的关系。仍以定点小数补码为例进行讨论。

设被乘数$[X]_{补} = X_0.X_1X_2\cdots X_n$，乘数$[Y]_{补} = Y_0.Y_1Y_2\cdots Y_n$，乘积$[P]_{补} = P_0.P_1P_2\cdots P_{2n}$，运用补码与真值的转换关系式：

当$[Y]_{补} = Y_0.Y_1Y_2\cdots Y_n$时，$Y = -Y_0 + \sum_{i=1}^{n} Y_i \times 2^{-i}$，则

$$X \times Y = X \times \left[-Y_0 + \sum_{i=1}^{n} Y_i \times 2^{-i}\right] = X \times (0.Y_1Y_2\cdots Y_n) - X \times Y_0$$

对等式两边求补码得：

$$\begin{aligned}[X \times Y]_{补} &= [X \times (0.Y_1Y_2\cdots Y_n) - X \times Y_0]_{补} \\ &= [X \times (0.Y_1Y_2\cdots Y_n)]_{补} - [X \times Y_0]_{补}\end{aligned}$$

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

推导结果表明：$[X\times Y]_{补}$等于$[X]_{补}$乘以真值Y，也就是说$[X]_{补}$和$[Y]_{补}$直接相乘并不等于$[X\times Y]_{补}$。通过此推导我们得到了补码乘法的基本公式：

$$[X\times Y]_{补}=[X]_{补}\times Y \tag{5-24}$$

上述推导过程中我们用到了等式：

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

为便于理解，对这一等式证明如下：

(1) 当 $X\geqslant 0$ 时

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

(2) 当 $X<0$ 时

$$\begin{aligned}[X\times(0.Y_1Y_2\cdots Y_n)]_{补}&=2+X\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\\&=2^{n+1}+X\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\\&=2^{n+1}\times(0.Y_1Y_2\cdots Y_n)+X\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\\&=(2^{n+1}+X)\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\\&=(2+X)\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\\&=[X]_{补}\times(0.Y_1Y_2\cdots Y_n)\quad(\text{Mod }2)\end{aligned}$$

证毕。

利用补码乘法的基本公式，我们可以进一步推导比较法算法。

$$\begin{aligned}[P]_{补}&=[X\times Y]_{补}=[X]_{补}\times Y\\&=[X]_{补}\times\Big(-Y_0+\sum_{i=1}^{n}Y_i\times 2^{-i}\Big)\\&=[X]_{补}\times(-Y_0\times 2^0+Y_1\times 2^{-1}+Y_2\times 2^{-2}+\cdots+Y_n\times 2^{-n})\\&=[X]_{补}\times[-Y_0\times 2^0+(Y_1\times 2^0-Y_1\times 2^{-1})+(Y_2\times 2^{-1}-Y_2\times 2^{-2})+\\&\quad\cdots+(Y_n\times 2^{-(n-1)}-Y_n\times 2^{-n})]\\&=[X]_{补}\times[(Y_1-Y_0)\times 2^0+(Y_2-Y_1)\times 2^{-1}+\cdots+(Y_{n+1}-Y_n)\times 2^{-n}],\\&\quad\text{令 }Y_{n+1}=0\\&=[X]_{补}\times\sum_{i=0}^{n}(Y_{i+1}-Y_i)\times 2^{-i}\end{aligned}\tag{5-25}$$

将上式进一步写成部分积递推公式得：

$$\begin{cases}[Z_0]_{补}=0\\[Z_1]_{补}=2^{-1}\{[Z_0]_{补}+(Y_{n+1}-Y_n)\times[X]_{补}\}(\text{初始令 }Y_{n+1}=0)\\\cdots\\[Z_i]_{补}=2^{-1}\{[Z_{i-1}]_{补}+(Y_{n-i+2}-Y_{n-i+1})\times[X]_{补}\}\\\cdots\\[Z_n]_{补}=2^{-1}\{[Z_{n-1}]_{补}+(Y_2-Y_1)\times[X]_{补}\}\\[Z_{n+1}]_{补}=[Z_n]_{补}+(Y_1-Y_0)\times[X]_{补}=[X\times Y]_{补}=[P]_{补}\end{cases}\tag{5-26}$$

式(5-26)中，$[Z_0]_{补}$为部分积的初始值，$[Z_1]_{补}\sim[Z_n]_{补}$依次为各次求得的部分积，$[Z_{n+1}]_{补}=[P]_{补}$则是最终求得的乘积。可以看出，部分积递推公式给出了算法每一步要做

的运算,由于每一步做什么运算都由相邻两位乘数($Y_{i+1}-Y_i$)比较决定,因此这种算法被称作比较法。在这里,相邻两位乘数 Y_i、Y_{i+1} 称为乘法的判断位。Y_{n+1} 是为保持每一步算法的统一而增设的附加位,由于其初值为 0,故增加附加位不会影响运算结果。

2) 比较法运算规则

现将定点小数补码一位乘比较法算法规则总结如下。

(1) 参加运算的数 X、Y 均用补码表示,且符号位都参加运算,乘积的符号位由运算过程自动产生;

(2) 设部分积初值为 0,为防止运算过程中产生暂时性溢出,部分积采用双符号位(模 4 补码)运算,但乘数只需要一位符号位参加运算;

(3) 乘数最低位之后增加一位附加位 Y_{n+1},且初始令 $Y_{n+1}=0$;

(4) 运算前先检测 0,只要 X、Y 两数有任意一个为 0,则乘积为 0,不再进行运算;

(5) 用乘数的最低两位 Y_nY_{n+1} 作为判断位,由于每求一次部分积要右移一位,所以 Y_nY_{n+1} 的比较值($Y_{n+1}-Y_n$)始终决定了每次应执行的操作。

部分积运算规则如下:

$$
\begin{aligned}
&Y_nY_{n+1}=00,\quad [Z_i]_{补}=[Z_{i-1}]_{补}+0, && \text{算术右移 1 位}\\
&Y_nY_{n+1}=01,\quad [Z_i]_{补}=[Z_{i-1}]_{补}+[X]_{补}, && \text{算术右移 1 位}\\
&Y_nY_{n+1}=10,\quad [Z_i]_{补}=[Z_{i-1}]_{补}+[-X]_{补}, && \text{算术右移 1 位}\\
&Y_nY_{n+1}=11,\quad [Z_i]_{补}=[Z_{i-1}]_{补}+0, && \text{算术右移 1 位}
\end{aligned}
$$

(6) 重复 $n+1$ 次比较和运算,移位按补码算术右移规则进行。但只进行 n 次右移,最后一次只运算不移位。

(7) 运算完成后,为避免误差,乘数的最低两位 Y_nY_{n+1} 清 0。

3) 补码一位乘法运算的实现

实现补码一位乘法所需的运算器结构与原码一位乘运算器类似,如图 5-17 所示。该运算器的主体部件仍然是一个加减法器和三个寄存器,各部件的作用基本与原码乘法运算器相同。但是,由于补码的符号位需要参加运算,因此加法器和寄存器的位数都要增加到 $n+2$ 位(双符号位运算)。注意,Q 寄存器中包括了一位附加位 Y_{n+1},存放乘数只需 $n+1$ 位(单符号位参加运算)即可。另外,补码的部分积运算控制电路也比原码要复杂一些。

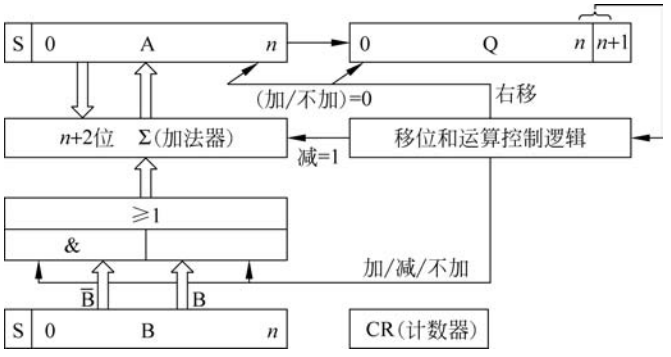


图 5-17 补码一位乘比较法运算器

【例 5.2】 $X=X=0.1101$，$Y=-0.1011$，用补码一位乘比较法求 $X\times Y$，并写出机器运算步骤。

解：

$[X]_{补}=00.1101\rightarrow$B，　$[Y]_{补}=1.0101\ 0\rightarrow$Q，　$[-X]_{补}=11.0011\ 0\rightarrow$A，　$101\rightarrow$CR

机器运算步骤如下：

A	Q　$y_n y_{n+1}$	B	CR	说明
00.0000	1.0101 0	00.1101	101	初始化
+ 11.0011				$y_n y_{n+1}=10$，$+[-X]_{补}$
11.0011			100	第1步
111.1001	11.0101			$y_n y_{n+1}=01$
+ 00.1101				$+[X]_{补}$
00.0110			011	第2步
100.0011	011.010			$y_n y_{n+1}=10$
+ 11.0011				$+[-X]_{补}$
11.0110			010	第3步
111.1011	0011.01			$y_n y_{n+1}=01$
+ 00.1101				$+[X]_{补}$
00.1000			001	第4步
100.0100	0001 1.0			$y_n y_{n+1}=10$
+ 11.0011				$+[-X]_{补}$
11.0111	00010 0		000	第5步
	清0			
11.0111	00010 0	00.1101	000	结束

$[P]_{补}=[X\times Y]_{补}=11.0111\ 0001\ 00$，　$X\times Y=-0.1000\ 1111$

该算法规则同样适用于整数，但关注的焦点仍是定点整数小数点的隐含位置。在此可将小数点约定在乘数的附加位之后，这将意味着整数乘数在运算前被乘了 2。因此为了保证双倍字长整数乘积的小数点对齐，在最后一步运算之后，结果还需要进行算术右移 2 位的特殊处理。

3. 补码两位乘法

1）提高乘法运算速度的方法

前面讨论的原码或补码一位乘比较法都属于最基本的乘法算法，尽管实现细节不同，但其实质都属于逐位循环迭代算法，把一次乘法转换成 n 次累加和移位来实现，完成一次乘法所需时间直接取决于数据的位数。当字长等于 n 时，意味着乘法运算时间是加减时间的 n 倍左右，比加减运算慢得多。因此，如何提高乘法运算速度就成为运算器设计时研究的主要问题之一。

提高乘法执行速度有两条常见的思路如下。

（1）对算法进行改进，或研制更高效的算法。常用的改进方法是通过合并乘法的运算步骤来实现乘法加速。常见的有两位乘法——将一位乘两步合并一步完成，可使乘法速度提高一倍左右；三位乘法——将一位乘三步合一步，可使速度提高两倍左右；以及更多位同时运算的乘法等。这种方法较为传统，主要希望通过提高算法本身的并行性、减少迭代次数来解决速度问题，虽然可以取得一定效果，但无法从根本上改变多次迭代的相乘过程。

（2）用硬件阵列直接实现乘法。如在许多计算机中，都安排有大规模集成电路的阵列乘法器模块。这是大规模集成电路技术支持下的产物。阵列乘法虽然也需要特定的算法支

持,但算法的作用相对淡化。主要依靠大量硬件电路的分级堆砌来实现乘法,每步运算不再需要分时使用唯一的一个加法器串行进行,而是通过多级加法器并行完成。因此可保证乘法在一个时钟周期内结束。显然这是一种较为理想的实现方法,但需要大规模的使用硬件电路,不过这一点在目前的集成电路技术水平支持下已经不成问题。

下面先对第一种方法进行讨论,第二种方法将在5.3.6节讨论。

2) 补码两位乘比较法

补码两位乘比较法可以简单地理解为将补码一位乘比较法算法的两步运算合并为一步来做。利用前面介绍的补码一位乘比较法部分积递推公式(5-26)可方便地推导出补码两位乘比较法算法公式。即

设

$$\text{被乘数}\quad [X]_{补}=X_0.X_1X_2\cdots X_n$$
$$\text{乘\quad 数}\quad [Y]_{补}=Y_0.Y_1Y_2\cdots Y_n$$

由补码一位乘比较法算法得:

$$[Z_{i+1}]_{补}=2^{-1}\{[Z_i]_{补}+(Y_{n-i+1}-Y_{n-i})\times[X]_{补}\}$$

$$\begin{aligned}[Z_{i+2}]_{补}&=2^{-1}\{[Z_{i+1}]_{补}+(Y_{n-i}-Y_{n-i-1})\times[X]_{补}\}\\&=2^{-1}\{2^{-1}\{[Z_i]_{补}+(Y_{n-i+1}-Y_{n-i})\times[X]_{补}\}+(Y_{n-i}-Y_{n-i-1})\times[X]_{补}\}\\&=2^{-2}\{[Z_i]_{补}+(Y_{n-i+1}+Y_{n-i}-2Y_{n-i-1})\times[X]_{补}\}\end{aligned}\tag{5-27}$$

由式(5-27)不难得出以下结论:

补码两位乘比较法的部分积运算可以通过相邻的三位乘数 $Y_{i-1}Y_iY_{i+1}$ 作为判断位进行比较决定,每次运算后算术右移两位。

现将定点小数补码两位乘比较法运算的算法规则总结如下。

(1) 参加运算的数均用补码表示,符号位一起参加运算,乘积的符号位由运算过程自动产生;

(2) 部分积初值为0,为防止运算过程中产生暂时性溢出,部分积采用三位符号位(模8变形补码)运算;

(3) 乘数最低位之后增加一位附加位 Y_{n+1},且初始令 $Y_{n+1}=0$;

(4) 运算前先检测0:只要 X、Y 两数有任意一个为0,则乘积为0;

(5) 用乘数的最低三位 $Y_{n-1}Y_nY_{n+1}$ 作判断位,由于每求一次部分积要右移两位,所以乘数最低三位的比较值($Y_{n+1}+Y_n-2Y_{n-1}$)始终决定了每次应执行的操作。

部分积运算规则如下:

$Y_{n-1}Y_nY_{n+1}=000$, $[Z_{i+2}]_{补}=[Z_i]_{补}+0$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=001$, $[Z_{i+2}]_{补}=[Z_i]_{补}+[X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=010$, $[Z_{i+2}]_{补}=[Z_i]_{补}+[X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=011$, $[Z_{i+2}]_{补}=[Z_i]_{补}+2[X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=100$, $[Z_{i+2}]_{补}=[Z_i]_{补}+2[-X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=101$, $[Z_{i+2}]_{补}=[Z_i]_{补}+[-X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=110$, $[Z_{i+2}]_{补}=[Z_i]_{补}+[-X]_{补}$, 算术右移2位

$Y_{n-1}Y_nY_{n+1}=111$, $[Z_{i+2}]_{补}=[Z_i]_{补}+0$, 算术右移2位

(6) 设乘数数值部分为 n 位，

当 n 为奇数时，乘数设 1 位符号位，做 $\frac{n+1}{2}$ 次运算和移位，最后一步右移 1 位；

当 n 为偶数时，乘数设 2 位符号位，做 $\frac{n}{2}+1$ 次运算，$\frac{n}{2}$ 次移位，最后一步不移位。

(7) 运算完成后，为避免误差，乘数的最低三位 $Y_{n-1}Y_nY_{n+1}$ 应清 0。

不难看出除细节不同以外，补码两位乘比较法的基本规则与补码一位乘比较法是一致的。其所需的运算器结构也与补码一位乘比较法运算器基本相同，只需添加符号位并适当修改控制、移位等功能即可，在此不再赘述。

【例 5.3】 已知 $X=0.110011$，$Y=-0.101010$，用补码两位乘比较法求 $X\times Y$。

解：

$[X]_{补}=000.110011\rightarrow$B，　$[Y]_{补}=11.010110\ 0\rightarrow$Q，　$[-X]_{补}=111.001101$

$[2X]_{补}=001.100110$，　$[-2X]_{补}=110.011010$，　$0\rightarrow$A，　$100\rightarrow$CR

机器运算步骤如下：

	A	Q	B	说明
	000.000000	11.0101100	000.110011	CR=100
	+ 110.011010			$+[-2X]_{补}$
	110.011010			
→2	111.100110	1011.01011		CR=011
	+ 001.100110			$+[2X]_{补}$
丢掉→1	001.001100			
→2	000.010011	0010 11.010		CR=010
	+ 000.110011			$+[X]_{补}$
	001.000110			
→2	000.010001	100010 11.0		CR=001
	+ 111.001101			$+[-X]_{补}$
	111.011110	10001000.0	清0	CR=000
	111.011110	100010000	000.110011	结束

$[P]_{补}=[X\times Y]_{补}=111.011\ 110\ 100\ 010\ 000$，　$X\times Y=-0.100\ 001\ 011\ 11$

由于补码一位乘法和补码两位乘法均采用的是 Booth 算法（比较乘法），因此算法规则上有许多相似性。对于初学者来说，往往容易在运算次数、符号位数等细节问题上发生混淆，为了帮助大家记忆，现将这两种乘法算法需要注意的问题统一列于表 5-5 中，以供参考比较。

表 5-5　比较乘法运算总结

乘法类型	符号位			累加次数	移位		
	参加运算	部分积	乘数		方向	次数	每次位数
补码一位乘法	是	2	1	$n+1$	右	n	1
补码两位乘法	是	3	2（n 为偶数）	$\frac{n}{2}+1$	右	$\frac{n}{2}$	2
			1（n 为奇数）	$\frac{n+1}{2}$	右	$\frac{n+1}{2}$	2（最后一次移 1 位）

说明：n 为乘数数值部分的位数

5.3.5　定点除法运算

除法是乘法的逆运算，与乘法运算的处理思想相似，在计算机中可以将 n 位除法转化

为 n 次“减法-移位”的迭代过程实现。

1. 原码除法运算

1) 原码恢复余数除法

与乘法情况类似,在计算机中做除法仍然是原码比补码方便。让我们先分析一下笔算除法过程,明确计算机中实现除法需要解决的问题。为方便起见,仍以定点小数为例讨论除法算法。若 $X=-0.1001$,$Y=0.1010$,求 $X\div Y$,则笔算竖式如下:

```
                0.1110      商
0.1010 √0.1001 0            被除数X
       -  101 0             除数Y
        0.0100 00           部分余数
       -   10 10
        0.0001 100          部分余数
       -    1 010
        0.0000 0100         余数
```

最后得 $X\div Y=-0.1110$,$R^*=0.0000\ 0010$(R^* 表示余数的绝对值)。

我们对上述计算过程进行分析,可以总结出笔算除法的基本步骤。

(1) 首先比较除数和被除数的大小:若除数小于或等于被除数,够除,该位商上“1”,从被除数中减去除数,得到一次余数;若除数大于被除数,不够除,该位商上“0”,被除数保持不变,继续作为余数使用。由于除法还要继续进行,所得余数并不是最后的余数,因此称为“部分余数”;

(2) 将被除数的低位部分补充进来参加运算(被除数位数较多时),或在部分余数的最低位补“0”(被除数位数较少时),再与除数进行比较相除,直至除尽或商的位数满足要求为止。

如果想在计算机中直接实现上述笔算过程,需要解决以下问题。

(1) 需要在运算器中专门设置比较器线路。

(2) 每次上商后都要进行右斜对位相减,需要 $2n$ 位的减法器线路。

这些问题如果直接用硬件电路来解决,将会大大增加硬件的代价,并不合算。常用的解决方案仍是从算法上进行改进,改进思路如下:①先试着做减法,如果够减(部分余数为正),商上 1;如果不够减(部分余数为负),商上 0,并将减掉的除数加回去(恢复余数)。由于事先并不能确定是否真正需要做减法,因此这步操作叫作“试减”;②将除数右斜对位改为部分余数左移,将实际上并不参加运算的部分余数高位移出去,只留低 n 位参加减法;③减法由 $+[-Y^*]_{补}$ 转化为加法实现。

这样,就把除法过程转化成了 n 次“减法和移位”操作,用常规的 n 位加减法器和移位电路就能实现。这种最直观的原码除法算法叫作“恢复余数除法”。综上所述,定点小数原码除法的基本规则可描述为:

设被除数 $[X]_{原}=X_0.X_1X_2\cdots X_n$,除数 $[Y]_{原}=Y_0.Y_1Y_2\cdots Y_n$,则当满足条件 $0<X<Y$ 时,

$$[Q]_{原}=[X\div Y]_{原}=(X_0\oplus Y_0).(X^*\div Y^*)=Q_0.Q_1Q_2\cdots Q_n$$

其中,X^* 和 Y^* 分别表示 X 和 Y 的绝对值,X_0、Y_0 和 Q_0 分别表示 $[X]_{原}$、$[Y]_{原}$ 和 $[Q]_{原}$ 的符号位。

由于够减和不够减的情况是随机出现的,故恢复余数除法会使除法运算的操作步数不固定,导致控制电路比较复杂,而且在恢复余数时,要多做一次加法,降低了除法的执行速

度。因此，原码恢复余数除法在计算机中很少采用。

2）原码加减交替除法（原码不恢复余数除法）

原码恢复余数除法中导致算法控制不均衡的恢复余数操作能否省去呢？为了寻找这个问题的答案，我们先来看看下面的推导过程。

（1）在恢复余数除法中，若第 $i-1$ 次求商所得的部分余数为 R_{i-1}，且 $R_{i-1}\geqslant 0$，则第 i 次试减操作可描述为：$R_i=2R_{i-1}-Y^*$；

（2）若够减，部分余数 $R_i=2R_{i-1}-Y^*\geqslant 0$，本次商 1，部分余数左移 1 位得 $2R_i$；

若不够减，部分余数 $R_i=2R_{i-1}-Y^*<0$，本次商 0，应恢复余数

$$R_i'=R_i+Y^*$$

然后再左移一位得 $2R_i'$。

（3）进行第 $i+1$ 次操作：

若前一步够减

$$R_{i+1}=2R_i-Y^*$$

若前一步不够减

$$R_{i+1}=2R_i'-Y^*=2(R_i+Y^*)-Y^*=2R_i+Y^*$$

上式表明，当某步试减操作不够减时，并不需要立即恢复余数，只要下一步试减直接做加法即可，其结果与先恢复余数、然后左移一位再减 Y^* 是等效的。由于不需要恢复余数，且加减运算交替地进行，故称此算法为“原码加减交替除法”或“不恢复余数除法”。由此可知原码加减交替除法的试减规则可由下面新部分余数的通式表示：

$$R_{i+1}=2R_i+(1-2Q_i)Y^* \tag{5-28}$$

式中，Q_i 为第 i 次上的商，若部分余数为正，则 $Q_i=1$，$R_{i+1}=2R_i-Y^*$，部分余数左移一位，下一次继续减除数；若部分余数为负，则 $Q_i=1$，$R_{i+1}=2R_i+Y^*$，部分余数左移一位，下一次加除数。原码不恢复余数除法是对恢复余数除法的一种改进，它减少了不均衡的加法时间，且运算次数固定，使控制过程得以简化，故在计算机中得到了广泛应用。

3）原码加减交替除法的运算规则

综上所述，原码加减交替除法的运算规则可总结如下：

（1）参加运算的操作数均为原码，两数符号位不参加运算，取其绝对值做无符号数除法。

（2）商的符号位单独处理，由两数符号“异或”产生。

（3）除法运算前先检测 0：只要 X、Y 有任意一个为 0，不进行实际运算。若 X^* 为 0，则商为 0；若 Y^* 为 0，按非法除数处理。

（4）运算前应满足条件：$X^*<Y^*$，否则按溢出处理。

（5）设商的初值为 0（被除数取单倍字长时）或为被除数低位部分（被除数取双倍字长时）。部分余数的初值为被除数（被除数取单倍字长时）或为被除数高位部分（被除数取双倍字长时）。

（6）为防止运算过程中产生暂时性的溢出，部分余数可采用单符号位，且符号位初值为 0（取 X^*）。

（7）部分余数运算：首先试减：$X^*+[-Y^*]_{补}$；

若部分余数为正，商上 1，余数和商联合左移一位，减除数；

若部分余数为负，商上 0，余数和商联合左移一位，加除数；

若求 n 位商，以上的上商、移位和运算过程重复 n 步。

(8) 第 $n+1$ 步。

若余数为正,商上 1,商单独左移一位,余数不移位,运算结束;

若余数为负,商上 0,商单独左移一位,余数不移位,当结果需要余数时,最后一步需恢复余数(余数不移位直接加除数)。

此规则在具体运用时应注意以下几点。

(1) 本算法上商是利用左移后对最低空出位进行填补(0 或 1)完成的,因此最后一步运算(第 $n+1$ 步)时余数和商不再联合移位,此时余数保持不动,而商需要单独左移一位,以便填入最后一位商。

(2) 因运算过程中不断地进行左移,故算法得到的最后一步余数只是余数的低 n 位。为和真正的余数区别,我们称其为"机器余数",用 R_n 表示,意为第 n 步机器运算所得余数。最终的余数应为绝对值形式:$R^*=R_n\times 2^{-n}$。

(3) 虽然该算法规定运算前需满足 $X^*<Y^*$ 的条件,但如果 $X^*\geqslant Y^*$,算法仍然可以正常进行,只是第一次上的商为 1,表示溢出到整数位的数值。这个 1 可作为溢出判断条件,在运算步骤中第一次上商后增加判溢出操作。

(4) 由于加减交替除法中缺少对部分余数判"0"的步骤,因此算法运行到中间的某一步已除尽时(部分余数为全 0),算法不会自动停止,而是继续按既定步数运行完。

4) 原码加减交替除法的实现

除法运算所需电路与乘法运算类似,对乘法电路略加修改,就能实现除法的功能。原码加减交替除法运算器的框图如图 5-18 所示。

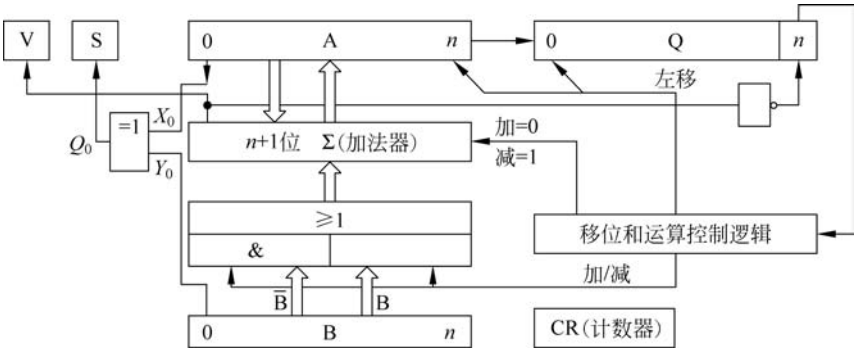


图 5-18 原码加减交替除法运算器

由图 5-18 可知,原码加减交替除法运算器的主体部分仍然是一个加减法器和三个寄存器。辅助电路除计数器 CR、商符产生电路外,还设有溢出标志 V。A 寄存器的初值是被除数,运算过程中为部分余数,运算结束后存放着机器余数;B 寄存器中存放着除数;Q 寄存器用来存放商。Q 在除法过程中仍然起着关键的作用,它和 A 首尾相接实现联合左移一位的功能。左移时 Q 的最低位用来上商。每次上的商同时作为下一次运算做加法还是做减法的控制信号使用。随着逐次联合左移,最终 Q 中存放着商的 $n+1$ 位绝对值。

【例 5.4】 $X=0.1001$,$Y=0.1100$,用原码加减交替除法求 $X\div Y$。

解:

$[X]_{原}=0.1001$;$[Y]_{原}=0.1100$;$X^*=0.1001\rightarrow$A;$Y^*=0.1100\rightarrow$B

$0<X^*<Y^*$;$[-Y^*]_{补}=1.0100$;$101\rightarrow$CR;$0\rightarrow$Q

机器运算步骤如下：

A	Q q_n	B	CR	说明
0.1001	0.0000	0.1100	101	初始化
+1.0100				$+[-Y^*]_{补}$
1.1101			100	$R<0$
1← 1.1010	0.0000			$q_n=0$
+0.1100				$+Y^*$
0.0110			011	$R>0$
1← 0.1100	0.0001			$q_n=1$
+1.0100				$+[-Y^*]_{补}$
0.0000			010	$R>0$
1← 0.0000	0.0011			$q_n=1$
+1.0100				$+[-Y^*]_{补}$
1.0100			001	$R<0$
1← 0.1000	0.0110			$q_n=0$
+0.1100				$+Y^*$
1.0100			000	$R<0$
+0.1100				$+Y^*$
0.0000				
1←	0.1100			$q_n=0$
0.0000	0.1100	0.1100	000	结束

$Q_0 = X_0 \oplus Y_0 = 0 \oplus 0 = 0$，$Q^* = (X \div Y)^* = 0.1100$，$[Q]_{原} = [X \div Y]_{原} = 0.1100$

$X \div Y = 0.1100$，$R^* = 0.0000 \times 2^{-4} = 0.0000\ 0000$

本例反映了原码加减交替除法的硬件操作过程，同时也示意出该算法运行到中间的某一步除尽时并没有自动停止，而是继续按既定步数运行完。这是由于算法中缺少对部分余数判“0”的步骤导致的。

该除法算法规则同样适用于整数，只要把例题中的小数点“.”改为逗号“,”即可。但此时被除数应采用双倍字长参加运算，否则当单字长被除数的绝对值小于除数时，得不到整数商。

2. 补码加减交替除法（补码不恢复余数除法）

1）补码加减交替除法的推导

原码除法与原码乘法一样，也比较简单直观易实现。但是在补码表示的机器中，仍然希望能够直接使用补码做除法。普遍采用的补码除法算法还是加减交替除法，其基本思想与原码加减交替除法一致，但被除数和除数都要用补码参加运算，求得的商和余数也是补码形式。加减交替除法在用于补码时不如原码直观，需要解决以下问题。

（1）试减及上商规则。

我们从原码除法的讨论中已知，上商时比较被除数（部分余数）和除数的操作改为试减实现。如果够减（够除）商 1；如果不够减（不够除）商 0。原码除法中试减可简单地通过被除数和除数的绝对值直接相减完成。但两个数的补码进行试减时，由于符号位要参加运算，情况就变得复杂了。当参加运算的两个数的补码符号任意时，进行试减前必须先判断两个数的符号：

- 若两数同号，商为正，试减时应做减法。减得的新部分余数与除数同号，表示够减商为 1；新部分余数与除数异号，表示不够减商为 0。
- 若两数异号，商为负，试减时应做加法才能实现绝对值的相减。得到的新部分余数与除数异号，表示够减商为 0（1 的反码商）；新部分余数与除数同号，表示不够减商为 1（0 的反码商）。

将上述分析结果加以总结我们发现，补码除法时正商和负商判断够减的条件正好相反，

因此无法再通过判断是否够减来决定上商。但是我们也发现了新的上商规律,即不管是否够减,也不管商的正负,补码除法运算均有:

- 当被除数(部分余数)与除数同号时,商上1,左移一位后减除数。
- 当被除数(部分余数)与除数异号时,商上0。左移一位后加除数。

上述规律我们可用一个求新部分余数$[R_{i+1}]_{补}$的通式描述如下:

$$[R_{i+1}]_{补}=2[R_i]_{补}+(1-2Q_i)\times[Y]_{补} \tag{5-29}$$

式中,Q_i表示第i步的商。若商上"1",下一步操作为部分余数左移一位减去除数;若商上"0",下一步操作为部分余数左移一位加上除数。将这个通式与原码新部分余数通式(5-29)比较可发现,两式中商对下一步加减运算的控制都是一样的,区别仅在于部分余数和除数的形式是否为补码。但要注意,原码算法和补码算法中负商的1、0取值的含义是完全不同的。

(2) 商符的产生。

由于运算前算法要求满足$X^*<Y^*$的条件,所以第一次试减后一定不够减,则上商操作的结果为:①若被除数与除数同号,则不够减时部分余数与除数异号,第一次商上0,正好等于正商的符号位;②若被除数与除数异号,则不够减时部分余数与除数同号,第一次商上1,正好等于负商的符号位。

由此可知商符在第一次试减后自动形成,且与数值位的上商规则相同。

(3) 商的校正。

从前面的分析已知:当商为负时,算法是以反码形式上商的,而按补码除法运算要求,最终应该得到商的补码,两者编码之间相差2^{-n}(即1LSB)。为了得到补码商,通常在最后一步上商时,采用商的末位"恒置1"的舍入方法进行处理。即最后一步商"恒置1",不再根据部分余数运算的结果决定商值。这种方法把反码商简单地修正为近似的补码商,产生的误差仅在$\pm2^{-n}$的范围内,运算后不再需要对商进一步校正,实现起来十分方便。

2) 补码加减交替除法的规则

根据上述的算法推导,补码加减交替除法的运算规则可归纳如下:

(1) 操作数均为补码,两数符号位参加运算,商符由运算自动产生;

(2) 运算前先检测0:只要X、Y有任意一个为0,不进行实际运算;若X为0,则商为0;若Y为0,按非法除数处理;

(3) 运算前应满足条件:$X^*<Y^*$,否则按溢出处理;

(4) 设商的初值为0(被除数取单倍字长)或为被除数低位部分(被除数取双倍字长)。部分余数的初值为被除数(被除数取单倍字长)或为被除数的高位部分(被除数取双倍字长);

(5) 为防止运算过程中产生暂时性的溢出,部分余数可采用单符号位或双符号位运算;

(6) 部分余数运算:首先试减

若X、Y同号,$[X]_{补}+[-Y]_{补}$;若X、Y异号,$[X]_{补}+[Y]_{补}$。

然后上商:

若部分余数与Y同号,商上1,余数和商联合左移一位,下一步做减法;

若部分余数与Y异号,商上0,余数和商联合左移一位,下一步做加法;

若求n位商,则上商、移位和运算过程重复n步。

(7) 第$n+1$步:

余数不移位,商单独左移一位恒置1,若不需要求余数,除法结束;

若需要求余数则继续判断：

余数与 X 同号，除法结束；

余数与 X 异号，最后一步需恢复余数，恢复方法如下：

若余数与 Y 同号，做减法；

若余数与 Y 异号，做加法；

补码加减交替除法需要注意的事项与原码加减交替除法类似，最后一步运算也需要特殊处理，且恢复余数时的运算与部分余数运算一致（同号减，异号加）；所得机器余数为补码形式；在 $X^* \geqslant Y^*$ 时补码除法也可正常进行，只是第一次上的商为溢出到整数位的数值而不是补码的符号位。此时商可以使用双符号位，其真符位通过 X、Y 的符号位“异或”产生，并可利用双符号位的“异或”进行溢出判断。

【例 5.5】 $X=0.1101$，$Y=0.1011$，用补码加减交替除法求 $X \div Y$ 的值。

解：

$$[X]_{补}=00.1101 \rightarrow \mathrm{A}；[Y]_{补}=00.1011 \rightarrow \mathrm{B}；$$

$$[-Y]_{补}=11.0101；101 \rightarrow \mathrm{CR}；0 \rightarrow \mathrm{Q}$$

采用算后判溢出方法，机器运算步骤如下：

A	Q q_n	B	CR	说明
00.1101	00.0000	00.1011	101	初始化
+11.0101				X, Y同号, $+[-Y]_{补}$
00.0010			100	R, Y同号
1← 00.0100	00.0001			商1
+11.0101				$+[-Y]_{补}$
11.1001			011	R, Y异号
1← 11.0010	00.0010			商0
+00.1011				$+[Y]_{补}$
11.1101			010	R, Y异号
1← 11.1010	00.0100			商0
+00.1011				$+[Y]_{补}$
00.0101			001	R, Y同号
1← 00.1010	00.1001			商1
+11.0101				$+[-Y]_{补}$
11.1111			000	R, Y异号，恢复余数
+00.1011				$+[Y]_{补}$
00.1010				
1←	01.0011			
00.1010	01.0011	00.1011	000	结束

$Q_s=X_s \oplus Y_s=0 \oplus 0=0$

$$[Q]_{补}=[X \div Y]_{补}=01.0011，[R]_{补}=00.1010$$

$$V=Q_s \oplus Q_0=0 \oplus 1=1，\text{结果溢出出错}$$

$$X \div Y=+1.0011，R=00.1010 \times 2^{-4}=0.0000\ 1010$$

本例题示意出补码加减交替除法算后判溢出的实现方式，就是运算开始之前不判溢出，直接进行运算。商设双符号位，通过运算得到的是商的第二符号位。第一符号位（真符位）直接由 X、Y 的符号位异或形成。运算完成后可通过变形补码判溢出的方法来判断运算结果是否发生溢出。本题的运行结果说明加减交替除法算法在有溢出发生时仍然能够正确运行。

补码加减交替除法规则也适用于整数。但被除数同样需采用双倍字长参加运算，否则不能保证得到的是整数商。

至此，我们看到原、补码加减交替除法的基本思想是一样的，只是采用了不同的码制实

现而已,因此两种算法之间有许多可比之处。现将其规则中涉及的主要问题列于表 5-6 中,以便形成对照,帮助大家区别和记忆。

表 5-6　原码、补码加减交替除法运算规则对照表

除法类型	符号位参与运算	加减次数	移位		备　　注
			方向	次数	
原码加减交替法	否	$n+1$ 或 $n+2$	左	n	若最终余数为负,需恢复余数
补码加减交替法	是	$n+1$ 或 $n+2$	左	n	末位恒置 1,若最终余数与被除数异号,需恢复余数

说明:n 为除数数值部分的位数。

5.3.6　阵列乘除法器

随着大规模集成电路技术的日益成熟,使用并行的阵列运算电路实现高速的乘除法器已得到广泛流行,下面将对其实现原理进行讨论。

1. 阵列乘法器

1) 无符号数阵列乘法

为了进一步提高乘法运算的速度,可采用大规模的阵列乘法器来实现。下面以两个 5 位无符号数相乘为例来说明阵列乘法的基本原理。与传统的逐位循环迭代算法相比,这种阵列乘法似乎实现了算法的回归,基本上可看成是笔算过程的大规模集成电路实现。

现在让我们重新分析笔算乘法过程,设两个不带符号的二进制整数:

$$A=\sum_{i=0}^{m-1}a_i\times 2^i,\ B=\sum_{j=0}^{n-1}b_j\times 2^j$$

则

$$P=A\times B=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(a_i\times b_j)\times 2^{(i+j)}=\sum_{k=0}^{m+n-1}P_k\times 2^k \tag{5-30}$$

当 $m=n=5$ 时,$A\times B$ 可用如下竖式算出:

$$\begin{array}{rrrrrrrrrrr}
 & & & & a_4 & a_3 & a_2 & a_1 & a_0 & =A \\
 & & & \times & b_4 & b_3 & b_2 & b_1 & b_0 & =B \\
\hline
 & & & & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_0 & & \\
 & & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
 & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
+\ a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & & \\
\hline
P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & =P
\end{array}$$

这是大家所熟悉的笔算乘法竖式,式中一位被乘数和一位乘数相乘的结果 $P_k=P_{ij}=a_i\times b_j$ 称为“位积”,由于一位二进制乘法规则和“逻辑与”的规则完全相同,因此位积可通过简单的与门产生,而多个位积则可通过一个与门阵列形成。求出位积后,将所有位积加起来就得到了最后的乘积。故两个无符号数相乘的过程可以用一个与门阵列加上一个加法阵列实现。

图 5-19 是位积产生电路的示意图，在这个例子中，它是一个 5×5 的与门阵列，当 A、B 同时输入后，这个与门阵列可同时输出各个位积。

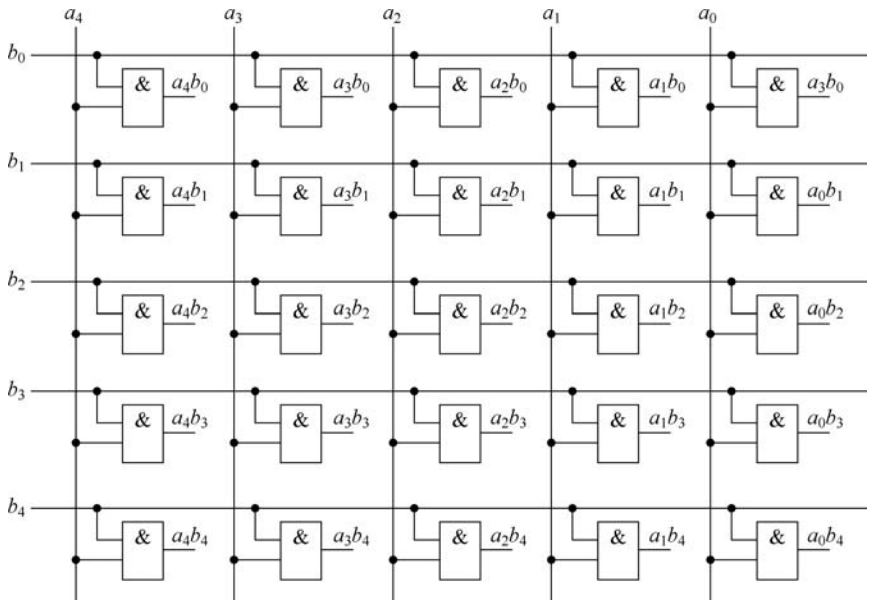


图 5-19　5×5 位积产生电路

图 5-20 所示为两个 5 位无符号数相乘所需的 5×5 的乘法阵列，其中 FA 表示全加器，在图中作为一位加法单元电路使用。图中的位积 a_ib_j 可通过上面图 5-18 的与门阵列产生。

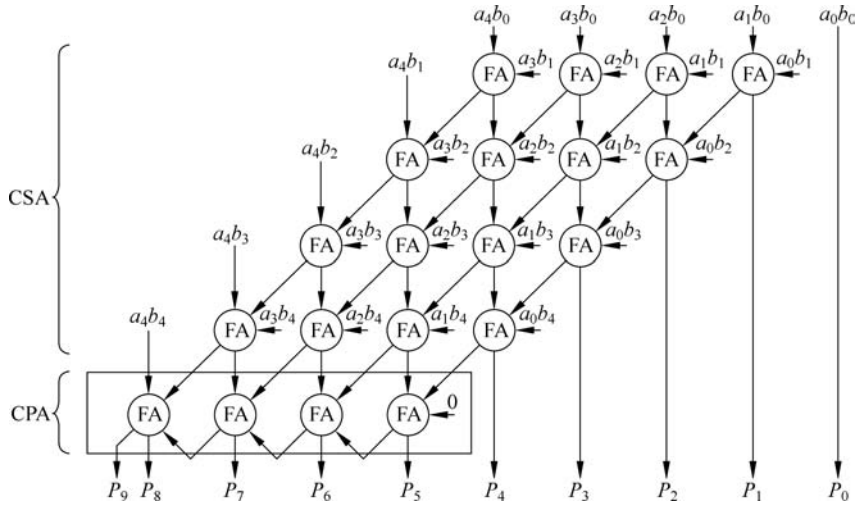


图 5-20　5×5 位绝对值相乘的阵列乘法器

图 5-20 中的 FA 构成了 5 个 5 位的二进制并行加法器，通过 5 级加法实现 5 次部分积的纵向左斜对位相加操作，并得到最后的 10 位乘积。

2）保留进位加法

仔细观察图 5-20 可以发现，图中 FA 进位输入/输出端的连接方法和最基本的并行加

法器(见图5-7)的进位连接形式有所不同。在阵列乘法器中,由于使用了多级加法器,如果还采用串行连接的进位方式,进位传递时间会对线路的运算速度产生较为显著的影响。为了减少加法的进位传递时间,阵列乘法器中经常用到一种被称为"保留进位加法"的快速相加技术,相应的加法器则称为保留进位加法器(Carry Save Adder,CSA)。

所谓保留进位加法是指:本位的两个数相加,产生的进位信号并不马上横向传递到本级加法器的高一位来参加本级的加法,而是暂时予以保留,以便纵向参加下一级加法器高一位的加法,即图5-20中进位信号的左斜传递相加过程。由图可见,保留进位加法非常适合多级阵列加法过程的快速进位传递。由于这种加法器也是用全加器FA实现的,为了便于区别,我们把前面5.1节中介绍的普通加法器称为进位传递加法器(Carry Parameter Adder,CPA)。但是我们也从图5-20看到,最后一级加法器无法再采用CSA而只能采用CPA,以便完成最终的求和过程。

如想提高最后一级CPA的速度,可采用另一种称为"先行进位"的快速进位技术。保留进位与先行进位两种快速技术的联合应用,可使阵列乘法器内部的相加过程基本不受进位信号传递时间的影响。因此尽管阵列乘法器中的加法阵列规模很大,但运算速度却很快,只需经历几级加法时延就可完成整个乘法过程。关于先行进位的原理我们将在本章的5.4节详细讨论。

以上是一个无符号数乘法阵列的例子,以此阵列为基础,外围配上相应的符号产生电路(异或门),就可以实现原码阵列乘法器。

3) 带求补器的阵列乘法器

如果希望用上述无符号数阵列乘法器实现补码乘法,则需要在无符号数阵列乘法器的外围再添加三个求补器,如图5-21所示。

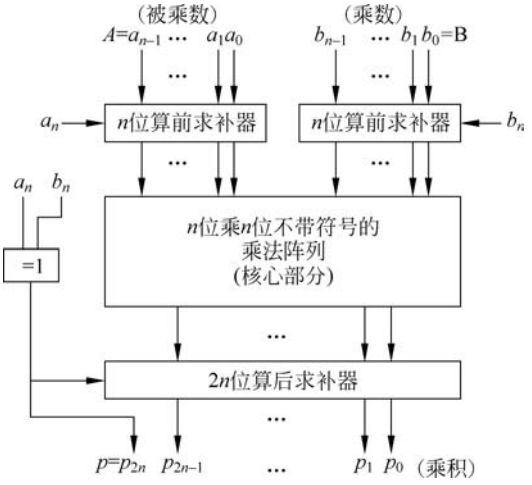


图5-21 $(n+1)\times(n+1)$位带求补器的阵列乘法器

两个算前求补器在相乘之前先将两个操作数的补码变成无符号整数,然后通过无符号数阵列乘法器进行乘法计算,最后通过算后求补器把运算结果转换回补码形式,乘积的符号也要通过外加的异或门电路产生。积符在输出的同时,还要作为控制信号来控制算后求补器的工作(同号相乘得正,结果不求补;异号相乘得负,结果求补)。

更好的实现补码阵列乘法的方法是利用加减法阵列电路直接实现 Booth 算法，完成两个补码的比较相乘过程。也就是说，可以利用先进的阵列技术来实现传统的逐位循环迭代算法，其好处是将分时使用一套加法器电路进行 n 次迭代运算，变为同时使用多级加法电路一次完成整个运算，使一次乘法运算时间由 n 个时钟周期缩短到在一个时钟周期内完成，其速度上的改善之大可想而知。限于篇幅在此不再做更深入的讨论。

2. 阵列除法器

与乘法情况相似，对除法运算进行加速的较理想方法是使用阵列除法器。阵列除法器的基本思想仍然是使用并行的多级加减法硬件电路来完成传统的逐位循环迭代算法，以赢得速度上的优势。根据所用算法的不同，阵列除法器也具有多种类型，但较常见的还是加减交替除法的阵列除法器。下面将详细介绍无符号数加减交替除法阵列除法器的组成原理。

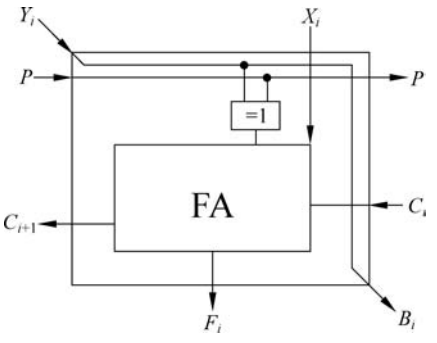


图 5-22　可控加减法单元

1）可控加减法单元

可控加减法单元(Controllable Adder Subtracter, CAS)是实现一位加减运算的硬件电路，其基本结构由一个全加器 FA 再加上一个求补用的异或门组成，如图 5-22 所示。图中异或门的两个输入端之一用来输入加数或减数，另一端接控制信号 P。

CAS 单元的输入/输出关系可表示如下：

$$\begin{cases} F_i = A_i \oplus (B_i \oplus P) \oplus C_i \\ C_{i+1} = A_i C_i + (A_i + C_i)(B_i \oplus P) \end{cases} \tag{5-31}$$

当 $P=0$ 时，输入的操作数不求补，控制做加法。上式就等于求和公式：

$$\begin{cases} F_i = A_i \oplus B_i \oplus C_i \\ C_{i+1} = A_i B_i + (A_i + B_i) C_i \end{cases} \quad \text{(同 FA 逻辑表达式)}$$

当 $P=1$ 时，对输入的操作数进行求补(变反+1)，控制做减法；则式(5-31)等于求差公式：

$$\begin{cases} F_i = A_i \oplus \overline{B}_i \oplus C_i \\ C_{i+1} = A_i \overline{B}_i + (A_i + \overline{B}_i) C_i \end{cases}$$

当最低位进位输入 $C_0=1$ 时，通过上述求差公式可完成减数求补并与被减数相加的操作。在做减法时，C_i 表示借位输入，C_{i+1} 表示借位输出。

2）除法阵列

将 $n+1$ 个 CAS 按串行进位方式连接在一起，就组成一个 $n+1$ 位的串行进位加减法器，可实现一次试减运算。若需要求 m 位商，就需要 $m+1$ 级 $n+1$ 位的串行进位加减法器组成加减法阵列来实现整个除法过程。在此 n 表示除数的数值位数，m 表示商的数值位数，图 5-23 给出了一个无符号数加减交替阵列除法的例子。

图中设被除数 $X^*=0.X_1X_2X_3X_4X_5X_6$(双倍字长)；除数 $Y^*=0.Y_1Y_2Y_3$，商 $Q^*=0.Q_1Q_2Q_3$，余数 $R^*=0.00R_3R_4R_5R_6$。这里 $n=m=3$，使用了 4 级 4 位的串行进位加减法器，沿纵向右斜对位构成除法阵列。由于原码加减交替除法的第一步试减肯定做减法，故

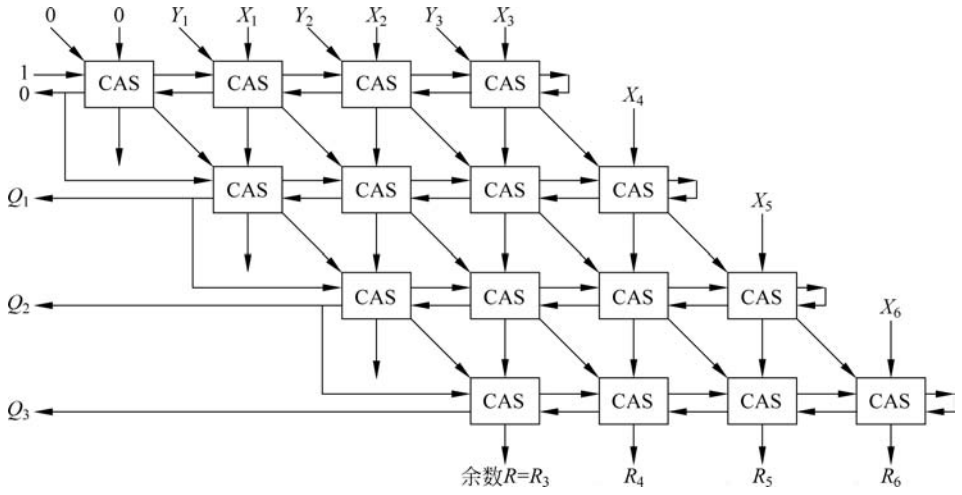


图 5-23　无符号数加减交替阵列除法器

控制信号 P 的初值恒为 1,其他各级加减法器输出的商值直接连到控制端作为 P 信号使用。经过分析可知,加减法器的最高位进位输出信号正好和该步试减要上的商值一致,因此通过各级加减法器的最高位进位输出端能够得到商的各位。

3) 阵列除法器运算举例

设 $X^*=0.100101$,$Y^*=0.101$,求 $Q^*=X^*\div Y^*$,则阵列除法过程如下:

第一级:试减,

$$P=1,R_1=X^*+[-Y^*]_{补}=0.100+1.011=\underset{\downarrow}{0}\ 1.111$$
$$C_4$$

在此 R_i 表示第 i 次部分余数。因为 $X^*<Y^*$,所以第一步一定不够减,最高位进位 $C_4=0$,此进位输出用作下一步的 P 信号,即 $Q_0=0$,$P=0$。

第二级:$R_2=R_1+Y^*=1.111+0.101=10.100$,$C_4=1$,够减 $Q_1=1$,$P=1$。

第三级:$R_3=R_2+[-Y^*]_{补}=0.100+1.011=01.111$,$C_4=0$,不够减 $Q_2=0$,$P=0$。

第四级:$R_4=R_3+Y^*=1.111+0.101=10.100$,$C_4=1$,够减 $Q_3=1$,除法结束。

最后得:$Q^*=0.101$,$R^*=0.000100$。

与传统的逐位循环迭代运算过程相比,本例中每一步运算都是电平操作,不需要脉冲控制,故整个阵列除法过程可在一个时钟周期内完成。

5.3.7　十进制运算

在某些特定的应用领域中,需要在计算机内部能够直接表示和处理十进制数据,为适应这方面的要求,目前大多数通用性较强的计算机内部都具有直接表示和处理十进制数据的能力。下面我们将讨论计算机中常用的十进制加法的基本原理和硬件实现。

1. 十进制加法单元

5.1 节中曾经讨论了计算机中表示十进制数的基本方法,即采用各种 BCD 码对十进制数据二进制代码化。那么,在计算机中进行十进制运算时,操作数显然是以 BCD 码的形式参加运算的,这就要求运算部件能够直接处理 BCD 码数据。十进制运算的核心部件是十进

制加法器，一位十进制加法单元是组成十进制加法器的基本元件，将进行重点讨论。

计算机中实现十进制加法的基本思想是：先将一位 BCD 码看成四位二进制数，利用二进制加法器进行加法，然后对所得到的和进行修正，转换成 BCD 码形式的和。原理性示意框图见图 5-24。

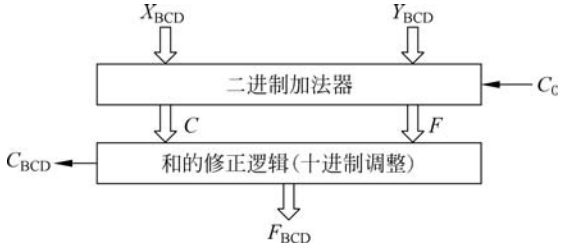


图 5-24　十进制加法单元组成框图

由图 5-24 可知，一位十进制加法单元可以看成是由二进制加法器和十进制修正逻辑两部分组成。由于对二进制加法和的修正一般通过加/减某个常数完成，因此十进制修正逻辑也可以用一级二进制加法器来实现。这样，整个十进制加法单元基本上可以看成是由两级二进制加法器再配以辅助逻辑组成的。二进制加法器的基本组成原理已在本节的开头进行过介绍，在此我们将十进制加法单元的主要设计任务集中在对二进制加法和以及进位信号的十进制修正上(也叫十进制调整)。下面主要介绍常用的 8421 码和余 3 码加法单元的十进制修正方法和逻辑实现。

1) 8421 码加法单元

十进制数采用 8421 码表示时，操作数的形式与 4 位二进制数相同，因此两个 8421 码可以直接进行二进制加法。但是需要解决以下两个基本问题：

(1) 相加后的和有可能进入 8421 码的 6 种无用编码组合(非码)，需要修正成正确的 8421 码。

(2) 两个 8421 码看成两个 4 位的二进制数进行相加，产生进位的基数为 16 而不是 10，导致加法过程不能自动产生出十进制的进位信号，需要设置专门的逻辑电路来实现十进制进位。

为了解决这两个基本问题，我们先将 8421 码加法的修正规则总结如下：

(1) 两个 8421 码直接进行 4 位二进制加法。

(2) 当二进制的和≤9 时，8421 码的和同二进制加法和，无须修正。

(3) 当 10≤二进制加法和≤19 时，需要进行加 6 修正，以产生 8421 码的和及十进制进位。

8421 码和的修正关系如表 5-7 所示。

表 5-7　8421 码和与二进制和的修正关系表

十进制数	8421 码和					校正前的和					校正关系
	C_{BCD}	F_{BCD4}	F_{BCD3}	F_{BCD2}	F_{BCD1}	C_4	F_4	F_3	F_2	F_1	
0～9	0	0	0	0	0	0	0	0	0	0	不校正
			…					…			
	0	1	0	0	1	0	1	0	0	1	

续表

十进制数	8421 码和					校正前的和					校正关系
	C_{BCD}	F_{BCD4}	F_{BCD3}	F_{BCD2}	F_{BCD1}	C_4	F_4	F_3	F_2	F_1	
10	1	0	0	0	0	0	1	0	1	0	
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	+6 校正
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	

由表 5-13 可以综合出 8421 码的进位生成逻辑：

$$C_{BCD}=C_4+F_4F_3+F_4F_2 \tag{5-32}$$

此逻辑产生本位 8421 码向高位传递的十进制进位信号，该信号同时也用来控制 8421 码和的+6 修正过程。一位 8421 码加法单元如图 5-25 所示，由一级 4 位二进制加法器、一级简化的二进制+6 加法器和进位产生逻辑三部分组成。

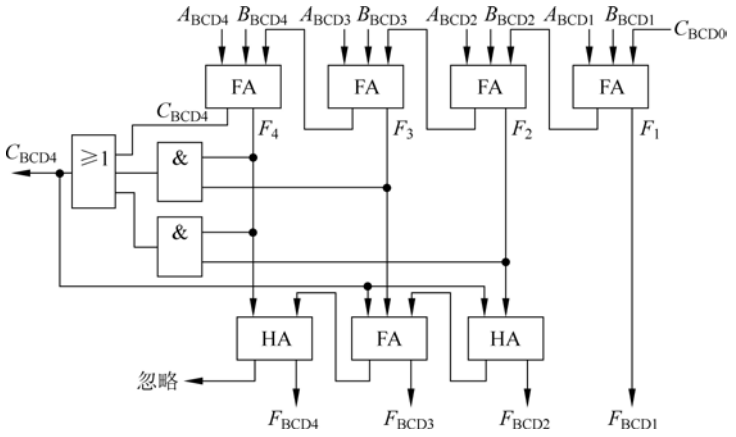


图 5-25　8421 码加法单元

2）余 3 码加法单元

与 8421 码相比，余 3 码加法的最大好处是能够通过二进制加法自动产生十进制的进位信号，不用再另外设置进位逻辑，只需要对和进行修正即可，实现起来比较方便。

余 3 码加法的修正规则为：

(1) 两个余 3 码先直接进行 4 位二进制加法。

(2) 如果无进位，减 3 修正，即加上 0011 的补码 1101。

(3) 如果有进位，加 3 修正，即加上 0011。

余 3 码和的修正关系如表 5-8 所示。

表 5-8　余 3 码和与二进制和的修正关系表

十进制数	余 3 码和					校正前的和					校正关系
	C_E	F_{E4}	F_{E3}	F_{E2}	F_{E1}	C_4	F_4	F_3	F_2	F_1	
0	0	0	0	1	1	0	0	1	1	0	−3 校正
1	0	0	1	0	0	0	0	1	1	1	
2	0	0	1	0	1	0	1	0	0	0	
3	0	0	1	1	0	0	1	0	0	1	
4	0	0	1	1	1	0	1	0	1	0	
5	0	1	0	0	0	0	1	0	1	1	
6	0	1	0	0	1	0	1	1	0	0	
7	0	1	0	1	0	0	1	1	0	1	
8	0	1	0	1	1	0	1	1	1	0	
9	0	1	1	0	0	0	1	1	1	1	
10	1	0	0	1	1	1	0	0	0	0	+3 校正
11	1	0	1	0	0	1	0	0	0	1	
12	1	0	1	0	1	1	0	0	1	0	
13	1	0	1	1	0	1	0	0	1	1	
14	1	0	1	1	1	1	0	1	0	0	
15	1	1	0	0	0	1	0	1	0	1	
16	1	1	0	0	1	1	0	1	1	0	
17	1	1	0	1	0	1	0	1	1	1	
18	1	1	0	1	1	1	1	0	0	0	
19	1	1	1	0	0	1	1	0	0	1	

由表可看出，二进制加法产生的进位信号与余 3 码十进制进位信号完全一致，即 $C_4 = C_E$，不再需要附加其他逻辑电路。同时我们还可利用该进位信号控制二进制加法和的修正过程，即 $C_4 = 0$，减 3 修正；$C_4 = 1$，加 3 修正。

一位余 3 码加法单元如图 5-26 所示。其组成特点可看成是由一级 4 位的二进制加法器和一级简化了的二进制＋3、－3 加法器两部分组成。

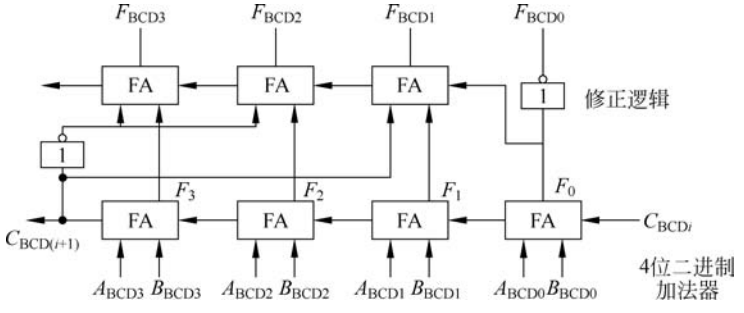


图 5-26　余 3 码加法单元

注意，一位 BCD 码加法单元中，由第二级修正加法器产生的最高位进位信号无用，任其自然丢失即可。

2. *n* 位十进制加法器

如果要进行 n 位的十进制加法,可考虑以一位十进制加法为基础,按以下规则进行。

(1) 各位 BCD 码内部的相加过程按一位 BCD 码加法及修正规则进行。

(2) n 位 BCD 码并行相加,位与位之间的十进制进位信号串行传递。

根据上述规则,n 位十进制并行加法器的组成如图 5-27 所示。

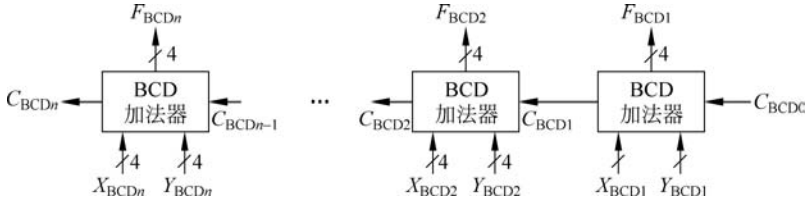


图 5-27 n 位十进制并行加法器

图中为突出 n 位十进制加法器的整体结构,淡化了位内的逻辑细节,把每位加法单元看成一个元素,用逻辑符号方框表示。n 位加法器需要用 n 个加法单元并行结构,位与位之间的进位信号则串行连接。这样就构成了一个最简单的 n 位十进制并行加法器。

5.3.8 基本的逻辑运算

计算机中除了最基本的加减乘除算术四则运算以外,经常还需要进行许多逻辑操作,如与、或、非、异或等,这些逻辑操作统称为逻辑运算。逻辑运算把操作对象看成无数值意义及位权关系的逻辑数,因此运算可以按布尔代数的规则进行。但是需要注意计算机中的逻辑数都是具有 n 位字长的机器数,而不是单个的逻辑变量。这使得逻辑运算具有按位并行进行、位与位之间无进位、借位等关系的特点。

大多数计算机中都支持四种最基本的逻辑运算:逻辑非、逻辑或、逻辑乘和逻辑异。其他更复杂的逻辑运算可以通过基本运算的逻辑组合实现。

1. 逻辑非(取反)

逻辑非是一种一元的逻辑运算,其所做的操作就是按位求反。逻辑非的运算符与布尔代数一致,通常用“－”号或“¬”号表示。计算机中一般安排“求反”或者“逻辑非”指令来提供逻辑非运算功能,如 80x86 系列机中的“NOT”指令。一位二进制数的逻辑非运算真值表如下:

X_i	F_i
0	1
1	0

对于一个 n 位的逻辑数 $X=X_0X_1\cdots X_n$,若求:$F=/X=F_0F_1\cdots F_n$,则运算规则可描述为:

$$F_i=/X_i \tag{5-33}$$

2. 逻辑或(逻辑加)

逻辑加也叫逻辑或,是一种二元逻辑运算,所做的操作为按位“或”。逻辑加的运算符通常用“或”运算符“∨”或者“＋”号表示。逻辑加运算可用相应的指令来实现,如 MIPS 机和

80x86 系列机中的 OR 指令。一位二进制数的逻辑加运算真值表为：

X_i ∨ Y_i	F_i
0 0	0
0 1	1
1 0	1
1 1	1

设两个逻辑数 $X=X_0X_1\cdots X_n$，$Y=Y_0Y_1\cdots Y_n$，若 $F=X\vee Y=F_0F_1\cdots F_n$，则其逻辑或运算规则可描述为：

$$F_i=X_i\vee Y_i \tag{5-34}$$

3. 逻辑乘(逻辑与)

逻辑乘也是一种常见的二元运算，又叫逻辑与，对应的操作为按位“与”。运算符用“∧”或“•”号表示。一位二进制数的逻辑乘真值表为：

X_i ∧ Y_i	F_i
0 0	0
0 1	0
1 0	0
1 1	1

若设 $X=X_0X_1\cdots X_n$，$Y=Y_0Y_1\cdots Y_n$，$F=X\wedge Y=F_0F_1\cdots F_n$，则逻辑乘运算规则可描述为：

$$F_i=X_i\wedge Y_i \tag{5-35}$$

4. 逻辑异(按位加)

与逻辑乘和逻辑加相比，逻辑异是一种稍稍复杂一点的二元运算。其操作就是按位“异或”。运算符可用异或操作常用的符号“⊕”表示。一位二进制数的逻辑异运算真值表如下：

X_i ⊕ Y_i	F_i
0 0	0
0 1	1
1 0	1
1 1	0

其运算规则可描述为：若

$$X=X_0X_1\cdots X_n,\quad Y=Y_0Y_1\cdots Y_n,\quad F=X\oplus Y=F_0F_1\cdots F_n$$

则

$$F_i=X_i\oplus Y_i \tag{5-36}$$

由真值表可以看出，一位逻辑异运算的结果与一位算术加法和的取值完全相同，只是不产生进位，故逻辑异运算又经常被称为按位加。

利用上述的基本逻辑运算，可以进一步实现任意的组合逻辑运算。

5. 逻辑运算的实现

由于计算机的运算部件本身就是采用逻辑电路搭建的，因此计算机内并不需要设置专

门的逻辑运算器,只要在算术运算部件的基础上附加少量的逻辑电路就能顺便实现逻辑运算功能。这种既能完成算术运算又能完成逻辑运算功能的部件称为“算术逻辑运算单元”,是运算器的核心部件,通常缩写为“ALU”。关于ALU的详细讨论,见5.4.2节。

5.4 定点运算器的实现

在介绍定点加、减、乘、除四则运算各种算法的过程中,针对每种算法都给出了对应硬件电路的实现方案。但是在实际的计算机中,不可能为每种运算都安排一套特定的线路,而是将拟采用的各种算法所需要的全部硬件整合在一起,组成一个多功能的部件来实现全部运算功能。这一节我们主要围绕具有完整运算能力的定点运算器设计原理进行讨论。

5.4.1 加法器的进位技术

从5.3节的讨论中已经了解到,不管运算部件的结构怎样复杂,其核心线路始终是加法器。因此关于运算器的讨论仍然从最基本的加法线路开始。由于加法运算是计算机中最基本的运算,所有算术运算均要借助于加法来完成,所以加法器的速度对整个计算机的运算速度有着至关重要的影响,而且随着字长的增加,进位速度对加法时间的影响也愈发明显。下面将通过对进位原理的分析,探讨实现加法器快速进位的方法。

1. 行波进位加法器

二进制并行加法器根据其进位传递方式又可分为串行进位加法器和并行进位加法器两种。其中,串行进位加法器也叫行波进位加法器(Ripple Carry Adder),是最简单的并行加法器。n位行波进位加法器由n个全加器组成,全加器的进位输入/输出端按照由低位到高位的顺序直接相连,构成链式的串行进位结构,因此串行进位线路通常也叫作“串行进位链”。图5-7所示的二进制并行加法器就是行波进位加法器。串行进位加法器的名称很容易和串行加法器混淆,讨论时应注意区别。

行波进位加法器的加法速度受到进位串行传递过程的限制。下面让我们围绕一个4位的行波进位加法器,来具体分析一下进位传递对加法时间的影响。根据5.3.1节给出的全加器真值表(见表5-3),行波进位加法器中各位的进位输出表达式可表示成

$$C_{i+1}=X_iY_i+(X_i\oplus Y_i)C_i \tag{5-37}$$

仔细对式(5-37)进行分析可发现,式中的第一项X_iY_i反映了本位产生进位输出的条件,而第二项中的“异或因子”($X_i\oplus Y_i$)则反映了前一位进位到本位进位输出的传递条件。为了简化全加器的进位逻辑,我们现在定义两个进位函数:

令
$$g_i=X_iY_i\text{——进位生成函数(Carry Generate)} \tag{5-38}$$
$$p_i=X_i\oplus Y_i\text{——进位传递函数(Carry Propagate)} \tag{5-39}$$

将这两个进位函数代入式(5-37)可得

$$C_{i+1}=g_i+p_iC_i \tag{5-40}$$

由此可写出4位行波进位加法器每位的进位逻辑如下:

$$\begin{cases}C_1=g_0+p_0C_0\\C_2=g_1+p_1C_1\\C_3=g_2+p_2C_2\\C_4=g_3+p_3C_3\end{cases} \tag{5-41}$$

这组进位逻辑表达式清楚地示意出行波进位加法器串行进位链路中高位进位依赖于低一位进位信号的传递关系。为了便于分析进位延迟时间，若采用与非逻辑实现式(5-41)，则每位全加器 FA 内部大约经历两级与非门的延迟产生进位输出信号，如图 5-28 所示。

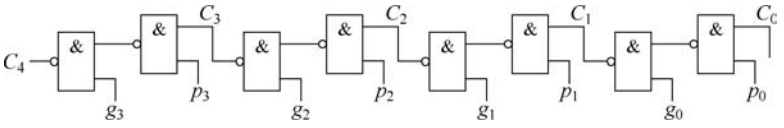


图 5-28　四位串行进位链

若设图 5-28 中与非门的延迟时间为 1ty，且忽略产生进位函数所需的时延，则可看出一位全加器从低位进位输入到本位进位输出最长需时 2ty，每增加 1 位全加器就增加 2ty 的进位延迟时间，而 4 位加法器最长则需 4×2ty＝8ty 进位时延。由此不难推出，n 位加法器的最长进位时间为 $2n$ty。当 $n=16$ 时，最长进位时间可达 32ty。注意，这里对进位延迟时间的计算只是一种粗略的估算，意在给大家一个原理性的时间概念，精确的进位时间计算是和实际所用门电路的具体参数密切相关的，在此不做深入讨论。

现在再看看没有进位传递时并行加法器所需的加法时延。假设一个异或门的延迟时间为 1.5ty，根据全加器输出逻辑表达式，则从加数 X_i、Y_i 输入到全和 F_i 输出仅需要两级异或门延迟时间 3ty，且与加法器位数的多少无关(各位加数并行输入，各位和并行输出)。但在行波进位加法器位数较多时，并行相加的这种速度优势并不能很好地发挥出来，如 16 位字长的情况下，最长进位时延(32ty)与并行加法所需时延(3ty)是十分不相适应的，由此可知串行进位传递时间是影响加法器速度的主要因素。如果想提高加法速度，必须设法改变进位信号串行传递的方式。

2. 先行进位加法器

1) 先行进位原理

如何才能突破加法器的串行进位结构呢？由 4 位行波进位加法器的进位表达式(5-41)已经看到，串行进位的特点是每位进位信号的产生都对低一位进位有依赖关系。如果设法打破这种依赖关系，就能有效地提高进位速度。仍以 4 位加法器为例，让我们试着将串行进位表达式中的前一位进位信号用其表达式直接代入，然后加以整理，就得到了一组新的进位表达式：

$$\begin{cases} C_1 = g_0 + p_0 C_0 \\ C_2 = g_1 + p_1 g_0 + p_1 p_0 C_0 \\ C_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0 \\ C_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0 \end{cases} \tag{5-42}$$

这组表达式中，各位进位信号都直接由进位函数和最低位进位输入产生，而进位函数是直接由加数生成的，与低一位进位信号无关，这就打破了串行进位之间的依赖关系。这种直接由原始操作数产生进位信号的技术，叫作先行进位(Carry Look Ahead，CLA)或并行进位、同时进位、超前进位等。一个由与或非-与非逻辑实现的 4 位先行进位线路，如图 5-29 所示。

这种先行进位线路可通过 MSI 芯片实现，简称为 CLA 部件。以图 5-29 的线路为例，再来估算一下先行进位的延迟时间。假设图中与或非门的时延为 1.5ty，与非门时延仍为

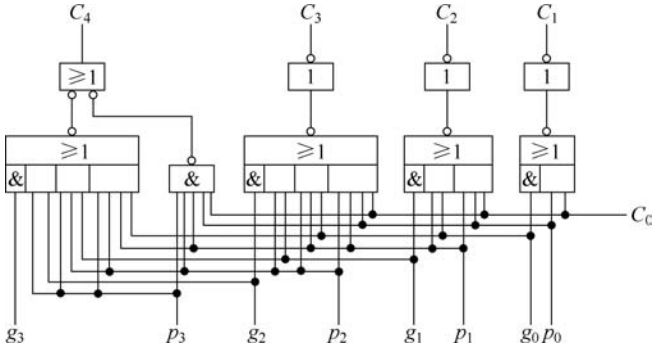


图 5-29　四位先行进位线路(CLA)

1ty,并忽略产生进位函数的时延,则产生进位输出共需两级门 2.5ty 左右的时间。而且这个延迟时间是个常数,对各位进位都一样,与加法器的位数无关。可见此进位时延与 3ty 左右的并行加法时间是相适应的。前面我们已经计算过 4 位串行进位最长约需 8ty 的时延,相比之下先行进位加速了两倍多,加速作用相当明显,而且位数越多加速作用越明显。例如,当加法器增加到 8 位时,串行进位最长达 16ty 时延,而并行进位时间仍然保持 2.5ty 左右不变,此时进位速度提高了 5 倍多。

2) 先行进位加法器

先行进位线路的逻辑结构较为复杂,故无法再直接使用全加器来搭建先行进位加法器。通常一个完整的先行进位加法器可看成是由进位函数产生线路(进位生成/传递部件)、求和线路(求和部件)及先行进位线路(CLA 部件)三部分构成。4 位先行进位加法器结构框图如图 5-30 所示。

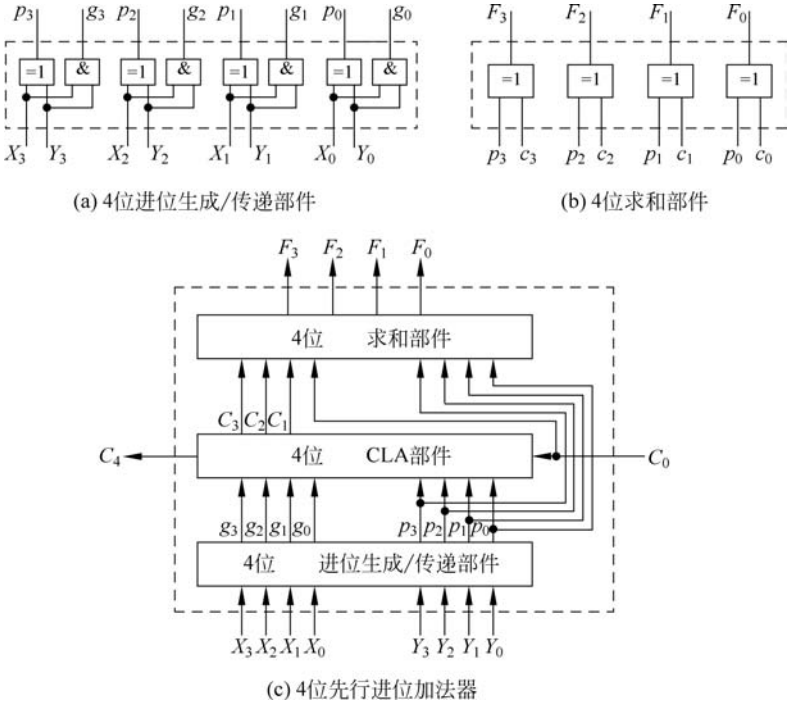


图 5-30　4 位先行进位加法器

3. 多级先行进位技术

先行进位技术虽然能有效改善加法器的进位速度，但却由于门电路扇入系数的影响，使加法器的位数受到限制。我们仔细观察 4 位先行进位表达式(5-42)就可发现，由于此时进位信号直接由操作数产生，使得进位表达式中的输入因子逐位增多，C_4 表达式中与门的输入已经达到了 5 个，因此先行进位加法器的规模一般最多只能到 8 位。如果想将先行进位技术用于位数更多的加法器中，必须采取其他措施。

1) 成组先行-级联进位

作为并行进位与串行进位的折中方案，成组先行-级联进位技术适合用来构建位数较多的先行进位加法器。实现的基本思想：将 n 位并行加法器分成若干组，每组若干位。组内采用先行进位方式，组间则采用串行进位(通常称为级联方式)。例如，采用成组先行-级联进位技术搭建一个 16 位的并行加法器，若每 4 位分为一组，共分 4 组。各组通用的进位表达式如下：

$$\begin{cases} C_{n+1} = g_n + p_n C_n \\ C_{n+2} = g_{n+1} + p_{n+1} g_n + p_{n+1} p_n C_n \\ C_{n+3} = g_{n+2} + p_{n+2} g_{n+1} + p_{n+2} p_{n+1} g_n + p_{n+2} p_{n+1} p_n C_n \\ C_{n+4} = g_{n+3} + p_{n+3} g_{n+2} + p_{n+3} p_{n+2} g_{n+1} + p_{n+3} p_{n+2} p_{n+1} g_n + p_{n+3} p_{n+2} p_{n+1} p_n C_n \end{cases} \tag{5-43}$$

式中，n 表示各组下标的增量，在 4 分组的情况下，$n=0,4,8,12$。这组进位表达式中，C_{n+1}、C_{n+2}、C_{n+3} 三个信号仅在组内传递，而 C_{n+4} 则为小组进位信号，它虽由组内先行进位电路产生，但却在组间进行传递。例如，将 $n=4$ 代入这组通式可得第二组的进位表达式如下：

$$\begin{cases} C_5 = g_4 + p_4 C_4 \\ C_6 = g_5 + p_5 g_4 + p_5 p_4 C_4 \\ C_7 = g_6 + p_6 g_5 + p_6 p_5 g_4 + p_6 p_5 p_4 C_4 \\ C_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 C_4 \end{cases} \tag{5-44}$$

其中，C_4 为第一组向本组传递的小组进位信号，C_8 为本组向第三组传递的小组进位信号。由这组进位表达式可以看出，采用成组先行-级联进位方案时，各组的进位信号虽然能够通过组内的先行进位逻辑产生，但依赖于前一组的小组进位信号。16 位成组先行-级联进位框图如图 5-31 所示。

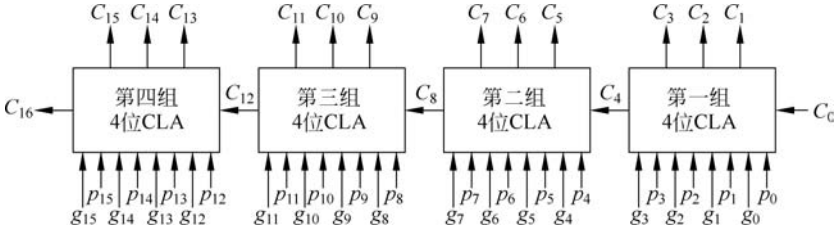


图 5-31　16 位成组先行-级联进位框图

由图 5-31 可看出，成组先行-级联进位是一个二级的进位结构，其组内为第一级进位，用 4 位 CLA 部件实现；组间则构成第二级进位，用级联方式实现。再让我们进一步分析一下成组先行-级联进位线路的速度。还是以图 5-31 为例，已知组内先行进位线路的时延为

2.5ty,16位分为4组的情况下最长进位时延可达2.5ty×4=10ty。这个时延虽然比先行进位的2.5ty长了许多,可是比16位串行进位最长需32ty还是快了2倍多,加速作用仍然十分明显,因此成组先行-级联进位技术在加法器位数不太多时仍不失为一种较好的进位加速方案。但如果在加法器位数很多时仍采用这种技术,进位加速作用会明显下降。如32位字长仍采用4位一组的分组方案时,组数增加到8,最长进位延迟时间则达2.5ty×8=20ty,加速作用已不再明显。

2) 多级先行进位

在对加法器进行分组的基础上,如果不仅组内先行进位,且组间也采用先行进位技术,就构成了二级先行进位加法器。仍以16位加法器、4位一组的分组方案为例来进行讨论。此时小组进位信号C_{n+4}不再由组内的先行进位线路直接产生,而是在组间专门设一级先行进位线路来产生出小组的进位信号。对组间进位信号的通式(5-43)进行分析:

$$C_{n+4}=g_{n+3}+p_{n+3}g_{n+2}+p_{n+3}p_{n+2}g_{n+1}+p_{n+3}p_{n+2}p_{n+1}g_n+p_{n+3}p_{n+2}p_{n+1}p_nC_n$$

可看出式中前4项为本组产生进位的条件,而第五项则反映了低一组进位信号通过本组传递到高一组的条件。因此我们定义两个小组进位函数:

$$\begin{cases}G_i=g_{n+3}+p_{n+3}g_{n+2}+p_{n+3}p_{n+2}g_{n+1}+p_{n+3}p_{n+2}p_{n+1}g_n & \text{小组进位生成函数}\\ P_i=p_{n+3}p_{n+2}p_{n+1}p_n & \text{小组进位传递函数}\end{cases} \tag{5-45}$$

函数的下标i表示组号,在16位加法器中,$i=0(n=0),1(n=4),2(n=8),3(n=12)$。

将小组进位函数代入小组进位信号表达式中可得:

$$\begin{cases}C_4=G_0+P_0C_0\\ C_8=G_1+P_1G_0+P_1P_0C_0\\ C_{12}=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0\\ C_{16}=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0\end{cases} \tag{5-46}$$

由这组表达式可以看出,组间先行进位的逻辑结构与组内先行进位线路完全相同,其输入都依赖于进位函数和最低位进位,输出均为进位信号。因此,我们可以使用通用的先行进位部件CLA来构成组间的先行进位线路。但是,为了便于CLA的连接,组内先行进位线路通常不再直接输出小组进位信号,而是输出一对小组进位函数P、G,然后通过小组进位函数进一步产生组间进位信号。由此构成了一种适用于分组结构的先行进位线路,叫作"成组先行进位部件",简称为BCLA(Block Carry Look Ahead)。4位BCLA部件的输入/输出逻辑关系如下:

$$\begin{cases}C_{n+1}=g_n+p_nC_n\\ C_{n+2}=g_{n+1}+p_{n+1}g_n+p_{n+1}p_nC_n\\ C_{n+3}=g_{n+2}+p_{n+2}g_{n+1}+p_{n+2}p_{n+1}g_n+p_{n+2}p_{n+1}p_nC_n\\ G=g_{n+3}+p_{n+3}g_{n+2}+p_{n+3}p_{n+2}g_{n+1}+p_{n+3}p_{n+2}p_{n+1}g_n\\ P=p_{n+3}p_{n+2}p_{n+1}p_n\end{cases} \tag{5-47}$$

式中,n仍表示不同组的下标增量,$C_{n+1}\sim C_{n+3}$为组内进位输出;P、G为组间进位函数输出。通过P、G函数,可在高一级先行进位线路中进一步组合出进位信号来。对于不同层次的先行进位线路来说,由式(5-47)所描述的BCLA部件是通用的,可用来实现不同分级的

先行进位加法器。为了便于区别，通常将基于操作数 X_i、Y_i 的最低级进位函数 p_i、g_i 称为一级进位函数，其逻辑结构见式(5-38)和式(5-39)，以及图 5-30(a)；而小组进位函数 P、G 则称为二级进位函数，见式(5-45)；三级进位函数在本书中通常用 P^*、G^* 表示……当级数更多时可依次类推。

目前已有很多通用的 BCLA 芯片可供使用，非常方便。使用时请注意 BCLA 和 CLA 两种进位部件的区别。用 BCLA 和 CLA 部件相结合构成的 4-4-4-4 分组的 16 位二级先行进位加法器逻辑框图如图 5-32 所示。

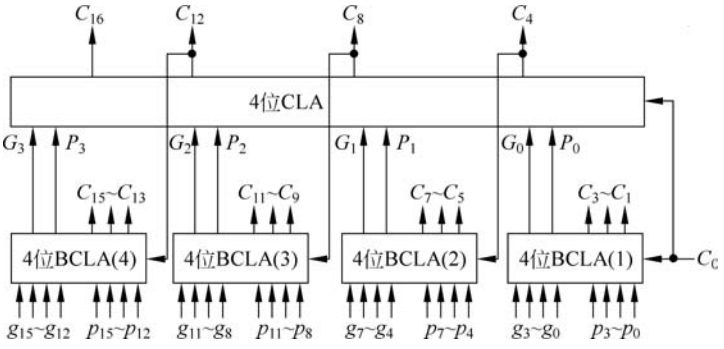


图 5-32　16 位二级先行进位加法器

该 16 位加法器的组内先行进位由 4 个 4 位的 BCLA 部件产生，组间先行进位由 1 个 4 位的 CLA 部件承担。图中最长进位传递顺序依次为：$(X_i、Y_i、C_0)\rightarrow(p_i、g_i)\rightarrow(C_1\sim C_3、P_i、G_i)\rightarrow(C_4、C_8、C_{12}、C_{16})\rightarrow(C_5\sim C_7、C_9\sim C_{11}、C_{13}\sim C_{15})$。若仍然忽略 p_i、g_i 的形成时间，则该二级先行进位线路所需的最长进位时间为 $3\times 2.5\text{ty}=7.5\text{ty}$，比成组先行-级联进位的 10ty 时延又快了 2.5ty。有趣的是，在此最高位进位 C_{16} 并不是最后产生的，而是“先行”于第 2～4 组组内进位信号产生。按照同样的思想，在加法器位数更多时还可进一步实现三级先行进位，或多级先行-级联进位方案，但是当先行进位线路的级数增加时，最长进位传递途径将变得较为复杂，这会使先行进位的加速作用遭到削弱，因此实际应用时先行进位线路的级数不宜过多。

5.4.2　算术逻辑单元

1. 算术逻辑单元的基本结构

加法器只是计算机中最基本的算术运算线路。在加法器的基础上，配以求补电路可以进一步实现加减运算，配以少量的逻辑电路可以完成各种逻辑运算，再加上适当的控制电路就可以实现对不同运算的选择控制。这种同时兼有算术、逻辑运算功能并具备简单的选择控制能力的电路称为算术逻辑单元，简称 ALU(Arithmetic Logic Unit)。ALU 是运算器的核心部件，其基本结构可看成是由加法器、逻辑函数发生器、功能选择电路等三部分组成。常用来表示 ALU 的逻辑符号及内部结构框图如图 5-33 所示。

通常，人们习惯于用一个梯形符号表示 ALU，梯形符号的宽边进一步分为两部分，表示 ALU 的两个输入端，窄边则表示 ALU 的输出，梯形符号的侧边通常标注 ALU 所需的运算控制信号。两个操作数输入 ALU 后，在运算控制信号的作用下，逻辑函数发生器根据运算类型产生出所选的逻辑函数，然后送到加法器进行指定的算术/逻辑运算，最后输出运算结果。

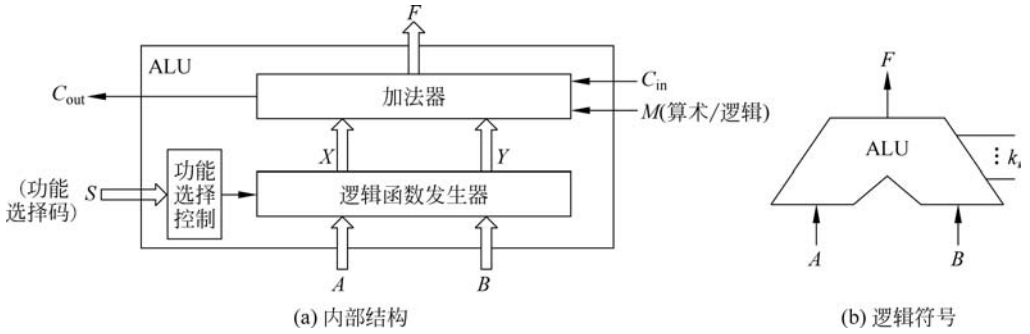


图 5-33　ALU 结构示意

2. 典型 ALU 芯片实例

1) 74181 芯片功能

作为一种经典的 ALU 芯片,74181 是一个 4 位的二进制多功能算逻运算部件,它具有 16 种可供选择的算术运算或逻辑运算,支持正、负两种逻辑电平的输入/输出,芯片规模虽小功能却十分强大,经常用来实现多位的 ALU 电路。74181 的逻辑符号如图 5-34 所示。

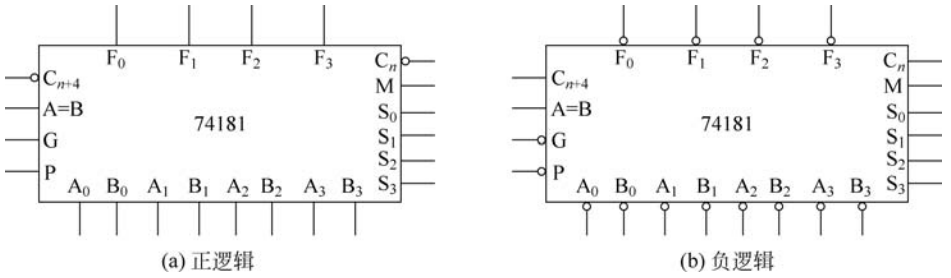


图 5-34　74181 芯片的逻辑符号

图 5-34 中,A_3～A_0、B_3～B_0 为输入的两个 4 位操作数,F_3～F_0 为输出的 4 位结果;C_n 表示最低位的进位输入,C_{n+4} 表示最高位的进位输出;G、P 分别为小组进位生成函数和小组进位传递函数输出;M 为工作方式选择信号,M=L(低电平)表示算术运算,M=H(高电平)表示逻辑运算;S_3～S_0 为 4 位功能选择码,其 16 种组合(2^4)共能选择 16 种逻辑或者算术运算。74181 ALU 算术/逻辑运算功能见表 5-9。

表 5-9　74181 ALU 算术/逻辑运算功能

工作选择 $S_3S_2S_1S_0$	负逻辑			正逻辑		
	逻辑运算 ($M=1$)	算术运算($M=0$) $C_n=0$(无进位)	算术运算($M=0$) $C_n=1$(有进位)	逻辑运算 ($M=1$)	算术运算($M=0$) $\overline{C}_n=1$(无进位)	算术运算($M=0$) $\overline{C}_n=0$(有进位)
L L L L	$F=\overline{A}$	$F=A$ 减 1	$F=A$	$F=\overline{A}$	$F=A$	$F=A$ 加 1
L L L H	$F=\overline{AB}$	$F=AB$ 减 1	$F=AB$	$F=\overline{A+B}$	$F=A+B$	$F=(A+B)$加 1
L L H L	$F=\overline{A}+B$	$F=A\overline{B}$ 减 1	$F=A\overline{B}$	$F=\overline{A}B$	$F=A+\overline{B}$	$F=(A+\overline{B})$加 1
L L H H	$F=1$	$F=$减 1	$F=0$	$F=0$	$F=$减 1	$F=0$
L H L L	$F=\overline{A+B}$	$F=A$ 加$(A+\overline{B})$	$F=A$ 加$(A+\overline{B})$加 1	$F=\overline{AB}$	$F=A$ 加 $A\overline{B}$	$F=A$ 加 $A\overline{B}$ 加 1
L H L H	$F=\overline{B}$	$F=AB$ 加$(A+\overline{B})$	$F=AB$ 加$(A+\overline{B})$加 1	$F=\overline{B}$	$F=(A+B)$加 $A\overline{B}$	$F=(A+B)$加 $A\overline{B}$ 加 1

续表

工作选择 $S_3S_2S_1S_0$	负逻辑			正逻辑		
	逻辑运算($M=1$)	算术运算($M=0$) $C_n=0$(无进位)	算术运算($M=0$) $C_n=1$(有进位)	逻辑运算($M=1$)	算术运算($M=0$) $\overline{C}_n=1$(无进位)	算术运算($M=0$) $\overline{C}_n=0$(有进位)
L H H L	$F=\overline{A\oplus B}$	$F=A$ 减 B 减 1	$F=A$ 减 B	$F=A\oplus B$	$F=A$ 减 B 减 1	$F=A$ 减 B
L H H H	$F=A+\overline{B}$	$F=A+\overline{B}$	$F=(A+\overline{B})$加	$F=A\overline{B}$	$F=A\overline{B}$ 减 1	$F=A\overline{B}$
H L L L	$F=\overline{A}B$	$F=A$ 加$(A+B)$	$F=A$ 加$(A+B)$加 1	$F=\overline{A}+B$	$F=A$ 加 AB	$F=A$ 加 AB 加 1
H L L H	$F=A\oplus B$	$F=A$ 加 B	$F=A$ 加 B 加 1	$F=\overline{A\oplus B}$	$F=A$ 加 B	$F=A$ 加 B 加 1
H L H L	$F=B$	$F=A\overline{B}$ 加$(A+B)$	$F=A\overline{B}$ 加$(A+B)$加 1	$F=B$	$F=(A+\overline{B})$加 AB	$F=(A+\overline{B})$加 AB 加 1
H L H H	$F=A+B$	$F=A+B$	$F=(A+B)$加 1	$F=AB$	$F=AB$ 减 1	$F=AB$
H H L L	$F=0$	$F=A$ 加 A^*	$F=A$ 加 A 加 1	$F=1$	$F=A$ 加 A^*	$F=A$ 加 A 加 1
H H L H	$F=A\overline{B}$	$F=AB$ 加 A	$F=AB$ 加 A 加 1	$F=A+\overline{B}$	$F=(A+B)$加 A	$F=(A+B)$加 A 加 1
H H H L	$F=AB$	$F=A\overline{B}$ 加 A	$F=A\overline{B}$ 加 A 加 1	$F=A+B$	$F=(A+\overline{B})$加 A	$F=(A+\overline{B})$加 A 加 1
H H H H	$F=A$	$F=A$	$F=A$ 加 1	$F=A$	$F=A$ 减 1	$F=A$

注：* 表示 A 加 $A=2A$，算术左移一位。

表 5-9 中分别给出了正、负逻辑定义下的 16 种逻辑运算和 16 种算术运算，而算术运算针对最低位有、无进位信号又分别对应 16 种运算组合。为了避免二意性，功能选择码 $S_3\sim S_0$ 采用 H(高)、L(低)电平的组合表示。注意表中列出的算术运算中，很多加数以两个原始输入数据 A、B 的组合逻辑函数形式出现。为了进一步区分逻辑运算与算术运算，表中我们用“+”号表示逻辑加，而算术加则用汉字“加”来表示。除此之外，74181 还具有比较输出功能，当 $F_3\sim F_0$ 输出 0 时，A=B 端输出高电平。但是比较功能只在异或、减法等特定的运算时有意义。

2) 74181 芯片内部结构

74181 芯片内部的逻辑结构可看成由逻辑函数发生器(功能发生器)、求和线路、先行进位线路、比较线路四部分组成，如图 5-35 所示。

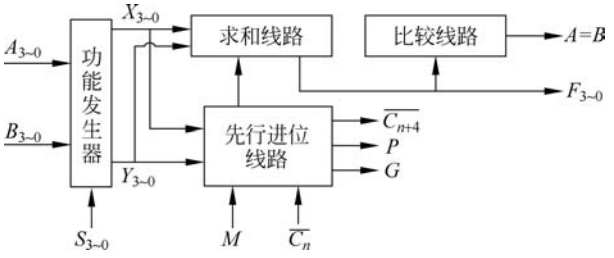


图 5-35　74181 芯片内部结构(正逻辑)

(1) 逻辑函数发生器。

逻辑函数发生器是 74181 的输入部件，其作用是按照功能码 $S_3\sim S_0$ 的选择，对原始输入数据 $A_3\sim A_0$、$B_3\sim B_0$ 进行逻辑组合，产生出 $X_3\sim X_0$、$Y_3\sim Y_0$ 两路逻辑函数输出。X_i、Y_i 再作为求和线路或进位线路的输入进行下一步的运算。函数发生器的逻辑表达式如式(5-48)所示。

$$\begin{cases} X_i=\overline{S_2A_i\overline{B_i}+S_3A_iB_i} \\ Y_i=\overline{A_i+S_0B_i+S_1\overline{B_i}} \end{cases} \tag{5-48}$$

74181 内部函数发生器设计的一大特点是,当进行加法算术运算时,X_i、Y_i 既是加数,又是进位函数 g_i、p_i,即有

$$\begin{cases}\text{进位生成函数：}g_i = X_i \cdot Y_i = Y_i \\ \text{进位传递函数：}p_i = X_i + Y_i = X_i\end{cases} \tag{5-49}$$

当把 X_i、Y_i 的不同输入组合代入式(5-49)时,会发现 $g_i = Y_i$、$p_i = X_i$ 的关系总是成立。这个设计十分巧妙,大大简化了 74181 内部的逻辑线路。

(2) 求和电路。

求和电路由两级异或门组成,以正逻辑为例,其等效的逻辑表达式为

$$F_i = X_i \oplus Y_i \oplus (C_{n+i} + M) \tag{5-50}$$

当 $M = H$ 时,选择逻辑运算,$F_i = X_i \oplus Y_i \oplus 1 = X_i \odot Y_i$,此时求和电路对低一位进位信号进行封锁,相当于一组同或门;

当 $M = L$ 时,选择算术运算,$F_i = X_i \oplus Y_i \oplus C_{n+i}$,进行全加操作。

(3) 先行进位电路。

74181 芯片兼有 CLA、BCLA 两种先行进位功能,以芯片的规模 4 位分组,同时支持先行-级联进位和多级先行进位方式。一方面小组进位信号 C_{n+4} 通过相应的引脚输出,可供组间级联进位使用。另一方面还输出小组进位函数 P、G,与外接的 BCLA 部件配合可实现组间先行进位。

(4) 比较电路。

74181 内部设置了一个比较门,当 $F_3 \sim F_0$ 为全 0 时,其输出端 $A = B$ 输出有效的高电平比较信号。正逻辑表示的 74181 内部线路如图 5-36 所示。

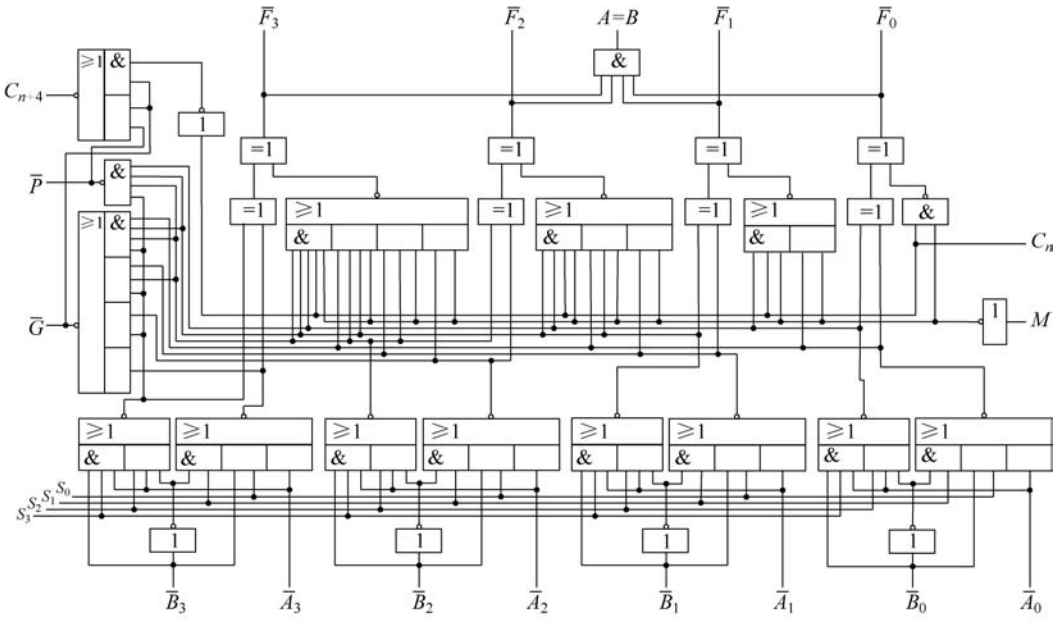


图 5-36　正逻辑表示的 74181ALU 内部逻辑线路

3. 多位 ALU 线路的实现

1）多位先行-级联进位的 ALU

将多片 74181 芯片的进位输入/输出端串行连接起来可以构成多位的先行-级联进位 ALU 线路。以正逻辑为例，一个用 4 片 74181 构成的 16 位先行-级联进位 ALU 线路如图 5-37 所示。

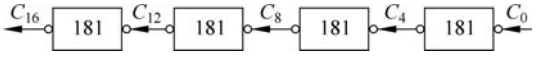


图 5-37　用 74181 芯片构成的 16 位先行-级联进位 ALU(正逻辑)

为了突出其进位结构，图中与进位无关的引脚连线全部省略。该 ALU 以 74181 为单位 4 位一组分组，组内进位由芯片内部的先行进位线路实现，组间进位则利用 74181 的进位输入端 C_n 和进位输出端 C_{n+4} 串联实现。

将 4 片 74181 芯片的 $A=B$ 输出端并联，还可实现 16 位的比较输出线路，如图 5-38 所示。

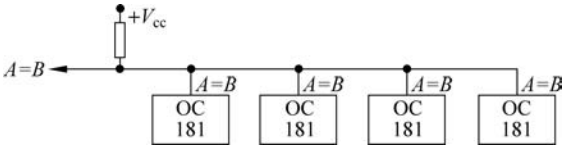


图 5-38　用 74181 芯片实现的 16 位比较线路

2）74182 BCLA 芯片

74182 是与 74181 配套的通用 4 位成组先行进位部件，74181 和 74182 配合使用可以构成字长更长的多级先行进位 ALU。与 74181 对应，74182 也支持正、负两种逻辑应用。74182 的逻辑符号图如图 5-39 所示。

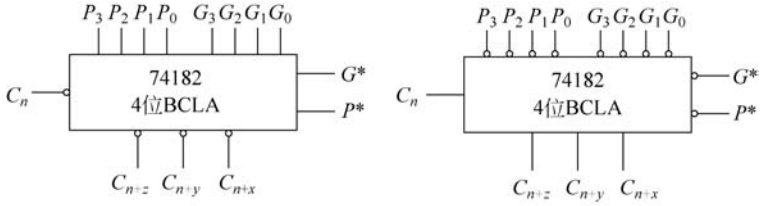


图 5-39　74182 四位 BCLA 芯片的逻辑符号

由于是通用的 BCLA 部件，74182 的两组进位函数输入引脚 $P_3 \sim P_0$、$G_3 \sim G_0$ 可根据需要接入任意级别的进位函数；C_n 为最低位进位输入；3 个进位输出信号的位序由高到低排列为 C_{n+z}、C_{n+y}、C_{n+x}，由于通用不给出具体位序；P^* 为成组进位生成函数，G^* 为成组进位传递函数，是 74182 产生的高一级进位函数输出，用来支持更高级的先行进位结构。

为了减少延迟时间，74182 内部的逻辑结构采用了进位信号输出与进位函数输入反相的方式。负逻辑表示的 74182 的内部逻辑如图 5-40 所示。

3）二级先行进位的 ALU

4 片 74181 与 1 片 74182 联合可构成 16 位二级先行进位的 ALU，如图 5-41 所示。16 位 ALU 分为 4 组，组内先行进位仍由 74181 内部实现，但组间不再使用 C_{n+4} 产生小组进

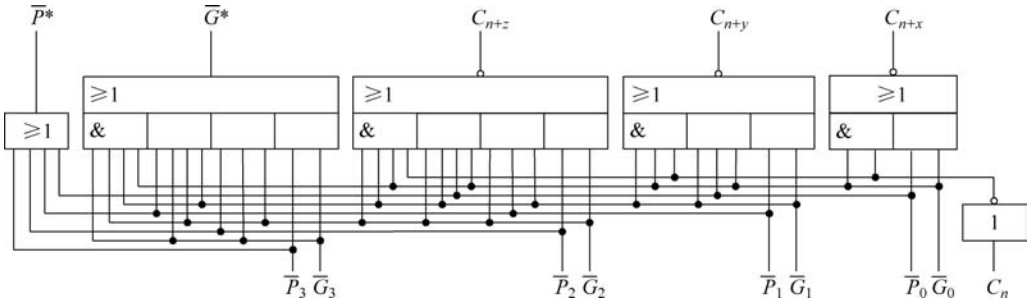


图 5-40　负逻辑表示的 74182 BCLA 内部逻辑线路

位信号,而是将 74181 的小组进位函数 P、G 信号送往 74182,再通过 74182 产生先行进位的组间进位信号。注意 74182 的最低进位输入端 C_n 与 74181 最低进位输入端同名,且一起与最低位进位输入信号 C_0 连接,这是由二级先行进位逻辑要求的。

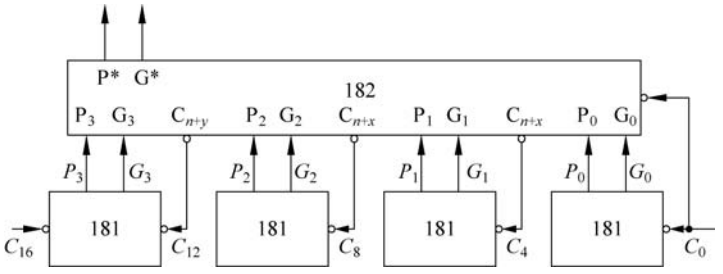


图 5-41　16 位二级先行进位的 ALU(正逻辑)

4) 二级先行-级联进位的 ALU

如果 ALU 的位数多于 16 位,分组可分级进行。如 32 位 ALU 可按 4 位一小组、4 小组一大组划分,共分两个 16 位的大组。大组内采用二级先行进位结构,大组间则利用 74181 的 C_{n+4} 输出端构成级联进位。由 8 片 74181 和 2 片 74182 组成的 32 位二级先行-级联进位的 ALU 如图 5-42 所示。

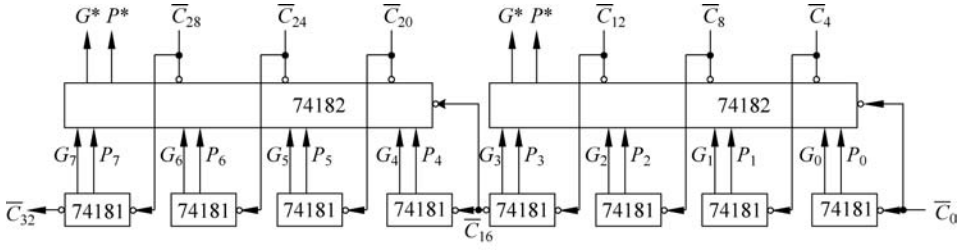


图 5-42　32 位二级先行-级联进位的 ALU(正逻辑)

5) 三级先行进位的 ALU

对于位数更多的 ALU 电路,还可采用三级先行进位结构来提高进位速度。例如,64 位的 ALU 可按 4 位一小组、16 位一大组分组,共分 16 个小组,用 16 片 74181 实现;4 个大组的内部用 4 片 74182 构成二级先行进位线路;4 个大组间再用 1 片 74182 实现第三级的先行进位,如图 5-43 所示。

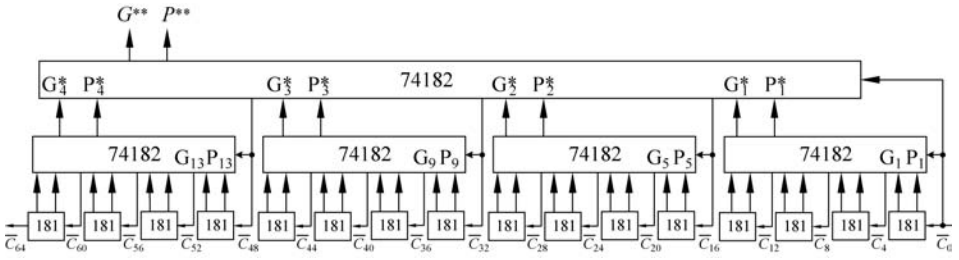


图 5-43　64 位三级先行进位的 ALU(正逻辑)

5.4.3　定点运算器的基本结构

定点运算器主要用来完成定点数据的运算操作。ALU 只是运算器的核心部件,要组成一个完整的运算器,还必须在 ALU 的基础上配以相应的寄存器、暂存器、移位器、多路选择器、内部总线等部件,才能支持全部的处理操作。归纳起来,一个完整的运算器应具有下述基本功能。

(1) 完成对数据的算术运算和逻辑运算,这是运算器的首要功能。这些功能由 ALU 承担,它在给出运算结果的同时,还给出结果的某些特征,如是否溢出,有无进位输出,结果是否为零、为负等。这些特征信息通常被保存在特定的触发器中,作为计算机运行的状态条件。ALU 做何种运算由相应指令执行时提供的运算控制信号指定。

(2) 暂存参加运算的原始数据和中间结果,这项功能通过运算器内部的一组寄存器完成。当前的计算机中都把这些寄存器设置成可以被汇编语言直接编程访问的形式,称为通用寄存器组。

(3) 有些运算器中设置有一个能自行左右移位的专用寄存器,称为乘商寄存器,或者设置专门的移位器部件,以支持传统的乘除运算所需的移位操作以及移位指令的执行。当然乘商寄存器的移位也可通过移位器实现。

(4) ALU 的输入/输出与通用寄存器组等部件间的相互连接通常通过几组多路选择器电路实现,以便从多个数据来源中选择一路送往 ALU 的某个输入端进行运算。运算器内部的数据传送通过其内部总线完成。

(5) 运算器还作为 CPU 内部传送数据的主要通路存在,要考虑与计算机中其他几个功能部件的连接和协同运行问题,就必须包括接收外部数据输入和送出运算结果的接口电路。因此运算器也经常称为 CPU 的数据通路。

运算器结构的合理性十分重要,将直接关系到计算机系统的性能。例如,运算器并行运算的位数取决于机器字长,通常是 16 位、32 位或者 64 位,涉及系统的处理能力;完成一次加法运算所用的时间限定了 CPU 工作周期的长度,运算器内部包含的寄存器个数决定了 CPU 读写存储器的频率,这些都影响到系统的运行速度。由此可见,运算器的组成是计算机设计时的一项重点工作,需要认真对待。由于运算器通常作为 CPU 内部的数据通路出现,因此对其结构控制原理的深入讨论将见第 6 章,这一章我们仅围绕运算器组织的基本逻辑实现加以讨论。

为了结构上的规整性,目前定点运算器内部一般也采用总线结构,称为 CPU 内部总线。注意内部总线与系统总线属于不同的总线层次,需要加以区别。一切送到运算器处理的

信息以及处理结果,不管其确切含义如何,均被看作是“数据”,故内部总线只需考虑数据的传输,按机器字长设定即可,不再需要像系统总线那样细分为数据线、地址线、控制线等类型。

运算器的结构方式,按其总线的设置情况,大体可分为以下几种。

1) 单总线结构的运算器

这种运算器内部通过一条总线连接所有部件的输入/输出,其基本结构如图 5-44 所示。

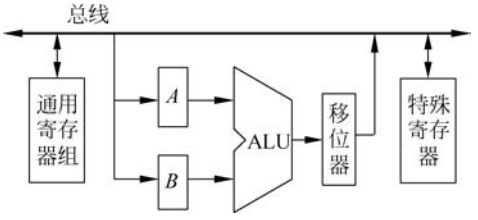


图 5-44 单总线结构的运算器数据通路

采用单总线结构时,需要特别考虑 ALU 的输入/输出端与总线的连接问题。由于 ALU 是一个组合逻辑部件,两个输入以及输出结果 3 路信号几乎同时要求使用总线,怎样利用唯一的总线进行传输是关键。常用的解决办法是在 ALU 的输入/输出端设置相应的暂存器。单总线结构中需要两个暂存器才能协调 ALU 输入/输出端同时使用总线的矛盾。至于暂存器的具体位置,可将其安排在两个输入端,如图 5-44 中所示的 A、B 暂存器。也可以一个放在输入端,另一个放在输出端,可视具体情况而定。由于总线传输的基本原则是“分时”使用,故单总线结构的运算器完成一次加法运算需要 3 个 CPU 时钟周期的时间,即两个时钟送操作数到 ALU 输入端的暂存器,一个时钟回送运算结果到累加器或指定的其他寄存器。

这种连接方式具有结构简单、规整的优点,其主要缺点是运算速度慢,通路中唯一的一条单总线构成了影响性能的主要瓶颈。

2) 双总线结构的运算器

与单总线结构相比,这种运算器内部多了一条总线,因此 ALU 的输入/输出端只需要设一个暂存器。这个暂存器可放在 ALU 的一个输入端,也可放在输出端,如图 5-45 所示。此时运算器完成一次加法运算需要两个 CPU 时钟周期的时间,一个时钟周期送操作数到 ALU,一个时钟周期回送运算结果,运算速度比单总线结构快了一个时钟周期。性能指标有所改善。两条总线之间的传输可通过 ALU 进行,或通过设置专用的总线连接器实现。

3) 三总线结构的运算器

三总线结构的运算器内部具有 3 条总线,如图 5-46 所示。此时可通过不同的总线同时为 ALU 送数并回送运算结果,运算器完成一次加法运算只需要一个 CPU 时钟周期的时间,运算速度不再受总线分时使用的限制,可以看出在三总线结构的运算器中总线已不是影响运算器性能的主要因素。

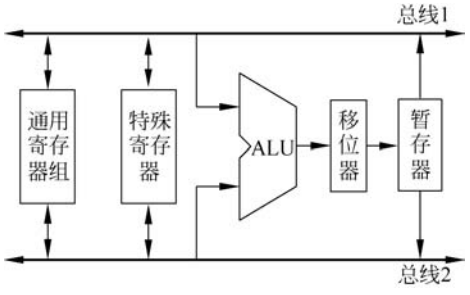


图 5-45 双总线结构的运算器数据通路

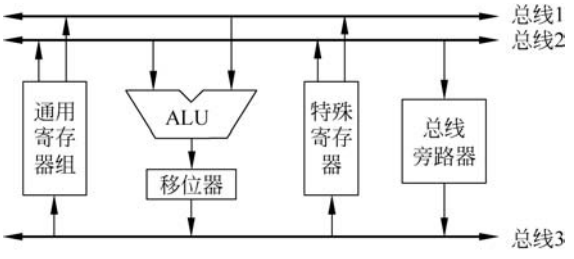


图 5-46 三总线结构的运算器数据通路

5.5 浮点数的表示与运算

本书 5.2 节曾讨论了定点数的表示方法，定点数的一个缺点是表示范围很小，只能表示纯小数或纯整数这样简单的数据形式。而在日常计算中，原始数据不一定正好是纯整数或者纯小数，即便正好是整数或小数，也不一定能满足机内定点数的表示范围要求。因此，原始数据在输入计算机前必须乘上一定的比例因子，统一进行“放大”或“缩小”，变成约定的定点数形式才能进行运算。在处理完成后，运算结果又要根据比例因子还原成实际的数值。如果在运算过程中出现结果“溢出”情况，还需要重新调整比例因子，这增加了编程工作量和机内进行“溢出”处理的次数，带来了很多的麻烦。尤其在科学计算这样的领域中，运算时涉及的数值常常非常大或非常小，如果仍用定点数进行计算，很难兼顾运算对数值范围和精度的双重要求。因此，计算机中引入了另一种数据的表示方法——浮点表示法。

5.5.1 浮点数的基本格式

浮点表示法对应于自然科学领域常用来表示实数的“科学记数法”，即把一个数表示成“数值×指数”的形式，对于数值范围过大的数来说，用这种方法表示起来既科学又简洁。例如，电子的质量可以表示为 9×10^{-28} 克，太阳的质量可以表示为 2×10^{33} 克，等等。使用这种方法表示的数据，相当于其小数点的位置可随指数的不同而变化，在一定范围内自由浮动，因此称为浮点数。浮点数可以书写成：

$$N = M \times R^{E} \tag{5-51}$$

式中，M 称为浮点数的尾数(Mantissa)，一般用带符号的 n 位小数表示，常为原码或补码，其形式同定点小数；E 叫作浮点数的阶码(Exponent)，一般为带符号的 k 位整数，常用移码或补码表示，其形式同定点整数；R 是浮点数阶码的基数或称阶的底，在计算机中取值固定，通常可取 2、4、8、16 等，最常见的情况是取 $R=2$。在此我们提到了移码表示法，这是一种经常用来表示浮点数阶码的数据表示方法，我们随后将予以介绍。浮点数在机内的一般格式如图 5-47 所示。

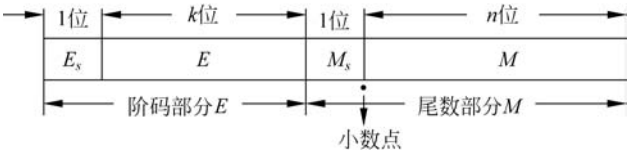


图 5-47 浮点数的一般格式

图中浮点数可看成是由阶码和尾数两部分组成，阶码部分包括阶码的符号位(E_s)和阶码的数值位(k 位整数)；尾数部分包括浮点数的符号位(M_s)和尾数的数值位(n 位小数)，尾数的小数点隐含约定在浮点数的符号位之后、尾数的最高数值位之前；由于阶码的基 R 是一个固定的常数，并不一定需要明显表示出来，因此采用隐含方式进行约定，在整个浮点数格式中不出现。为了讨论起来方便，在此我们约定对应上述格式的浮点机器码书写形式为：$E_s,E_1E_2\cdots E_k$；$M_s.M_1M_2M_3\cdots M_{n-1}M_n$。为清楚起见，格式中阶码与尾数之间用分号分开，阶符和阶值之间用逗号分开(同定点整数格式的描述形式)，数符和尾值之间用小数点分开(同定点小数格式的描述形式)。

5.5.2 浮点阶的移码表示

1. 移码的定义

前面我们已经介绍过,浮点数的阶码用带符号的整数表示,其形式同定点整数。因此从理论上来说,前面介绍过的任何一种定点机器数表示方法均应该能够用来表示浮点数的阶码。但在计算机中进行计算时,经常会遇到比较两数大小的操作。而两个浮点数进行比较,首先要比较阶码的大小。如果我们用常见的原、补、反等机器码来表示浮点数的阶码,由于其正、负域的定义不一样,因此当两个阶码的正、负不一致时,或者两个负阶进行比较时,就会不太方便。基于这个原因,目前在大多数通用计算机中,采用一种特殊的编码方式——移码表示法——来表示浮点数的阶码。

移码由真值 E 加上一个偏置常数 2^k 构成。加上这个偏置常数后,相当于将 E 在数轴上向正方向偏移了 2^k 个单位,这就是"移码"一词的由来。类似地,移码也可称为增码或偏码。

移码偏置常数的选取很有讲究,理论上一般考虑将一个正、负域对称表示的阶码真值,全部偏移到正数域用移码表示出来。这样做的好处很明显,偏移后移码值的大小直接反映了阶码真值的大小,比较时不用再考虑正、负数的区别。对于 $k+1$ 位阶码来说,总共需要 2^{k+1} 个无符号整数才能把 2^{k+1} 个真值全部表示出来。在 2^{k+1} 个阶码的真值中,有 2^k 个正数,2^k 个负数,正、负域基本上对称分布。而在 2^{k+1} 个无符号整数中,居于中间的两个数是:2^k-1(二进制码 0111…11)和 2^k(二进制码 1000…00)。如果希望阶码的正数和负数基本均匀地分布在 2^{k+1} 个无符号整数区间内,则可以选择这两个居中数中任何一个作为移码的偏置常数。出于习惯,偏置常数原理上一般取 2 的整幂(2^k),因此,也经常能见到将移码称为"余 2^k 码"的情况。

根据上述讨论,当浮点数的阶码由一位符号位和 k 位数值位构成的整数表示时,对应真值 E 的移码可定义为

$$[E]_{移}=2^k+E \quad -2^k \leqslant E < 2^k \tag{5-52}$$

在此,偏置常数 2^k 正好等于移码符号位的位权,这使得移码的符号位取值正好与原、补、反码相反。如果用 E_s 表示移码的符号位,则有正数时 $E_s=1$;负数时 $E_s=0$。图 5-48 是基于上述定义的移码和真值映射关系图。

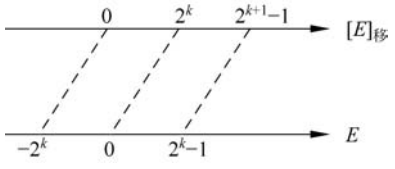


图 5-48 移码和真值的映射关系

由图可知,移码表示的几何意义为:将 k 位数在数轴上向右平移了 2^k 个位置,刚好把其下限 -2^k 移到了移码数轴上的 0 点,使得一个原本正负域取值的数被映射成可全部在正数域表示的,且由小到大顺序排列的移码。当移码用于实际的机器中时,也可能会根据具体需要选取 2^k-1 作为偏置常数。

2. 移码的特点

通过进一步观察可以看出移码具有以下特点。

(1) 移码符号位的取值与原、补、反码符号位的取值正好相反。

(2) 移码的符号位与数值位是一个从 0 到全 1 的连续编码的整体,不可分开考虑。正因为如此,很多系统中并不特别强调移码的符号位。

(3) 移码把跨越正负域的真值范围全部映射到一个正数域,所以两移码比较大小时,可

视为无符号数直接对数值进行比较。当浮点数阶码采用移码表示时，大大简化了两个阶码的比较操作。

（4）与补码表示法类似，真值0在移码中的表示形式也是唯一的，即

$$[0]_{移}=[+0]_{移}=[-0]_{移}=100\cdots000 \tag{5-53}$$

这有利于简化浮点数的判0操作。

（5）移码与补码之间存在符号位相反、数值位相同的简单转换关系。

正因为具有上述特点，使得移码广泛用来表示浮点数的阶码。采用移码表示的好处除了便于比较浮点数的大小之外，还能够简化浮点机器零的表示，有利于判零电路的实现。相关内容我们将在后面的章节中详细介绍。

3. 移码与补码的关系

对移码和补码进行更深入的比较我们会进一步发现，移码和补码有许多相似之处，但也有明显的区别。有以下几点。

（1）移码和补码的表示范围一样，负数域也可以表示到-2^k，比原码多表示一个码点，且正负数表示范围不对称。

（2）移码和补码的符号位取值相反，数值部分取值相同。

（3）移码和补码一样，零的表示唯一。但移码0的符号位取值为1，与补码0的符号位取值相反。

这些特点使得移码和补码之间转换起来非常方便。通常移码只用于表示整数，且只用于表示浮点数的阶码部分。

【例5.6】 设机器数字长为8位（含1位符号位），当其分别代表整数的无符号数、原码、反码、补码和移码时，对应的十进制真值各为多少？写出该机器数的全部二进制代码及其代表不同机器码时对应的十进制真值。

解： 根据题意，表5-10列出了8位二进制代码的所有组合及其解释为无符号数、原码、反码、补码和移码时所代表的十进制真值。

表5-10　8位机器数代表不同码制时的真值表示范围

二进制代码	无符号数 对应的真值	原码 对应的真值	反码 对应的真值	补码 对应的真值	移码 对应的真值
0000 0000	0	+0	+0	0	−128
0000 0001	1	+1	+1	+1	−127
0000 0010	2	+2	+2	+2	−126
…	…	…	…	…	…
0111 1110	126	+126	+126	+126	−2
0111 1111	127	+127	+127	+127	−1
1000 0000	128	−0	−127	−128	0
1000 0001	129	−1	−126	−127	+1
1000 0010	130	−2	−125	−126	+2
…	…	…	…	…	…
1111 1101	253	−125	−2	−3	+125
1111 1110	254	−126	−1	−2	+126
1111 1111	255	−127	−0	−1	+127

通过表5-10可以更加深入地理解“机器数”的含义。同一串二进制代码,如果用来表示不同的机器码,所对应的真值是截然不同的。

5.5.3 浮点表示法

当计算机中采用浮点表示法时,会涉及许多定点表示法中不需要考虑的复杂问题,下面让我们来看看这些问题在浮点表示法中是如何解决的。

1. 规格化的浮点数

1) 浮点数的规格化表示

如果不做特殊规定,一个浮点数的表示形式并不是唯一的。例如,二进制数101.1101可以表示为1.01 1101×2^2,也可以表示为0.101 1101×2^3、0.0101 1101×2^4,等等。但为提高运算精度,通常希望充分利用尾数的有效位数。因此,计算机中往往对浮点数进行规格化的规定,即要求尾数的最高位必须是一个有效数值。满足这种规定的浮点数称为规格化浮点数,对应地则把不满足这种规定的浮点数称为非规格化浮点数。规格化浮点数的尾数M的绝对值通常应在下列范围内:

$$\frac{1}{R} \leqslant |M| < 1 \tag{5-54}$$

如果取$R=2$,则有:$\frac{1}{2} \leqslant |M| < 1$。受此表示范围的限制,在用原码表示尾数时,规格化浮点数尾数的最高数值位总是为1,即$M_{MSB}=1$。在用补码表示尾数时,为了简化规格化数的判断条件,对其表示范围作了一些小小的调整,规定尾数的最高数值位与符号位取值不同时为规格化浮点数。即

$$M_s \oplus M_{MSB} = 1 \tag{5-55}$$

此时,规格化正尾数的表示范围为$\frac{1}{2} \leqslant M < 1$,应有0.1XXX…XX的表示形式;规格化负尾数的表示范围为$-1 \leqslant M < -\frac{1}{2}$,应有1.0XXX…XX的表示形式。注意:规格化补码尾数与规格化原码尾数有三点重要的区别:

(1) 原码规格化浮点尾数的表示范围正、负域对称,而补码规格化浮点尾数的表示范围正、负域不对称,不可再用式(5-54)的形式进行描述。

(2) 对于原码来说,$-\frac{1}{2}$是规格化尾数;而对于补码来说,这不再是一个规格化尾数。-1原码无法表示;而补码则能代表一个规格化尾数。

(3) 原码规格化数与补码规格化数的判断条件不一样,原码的判断条件为$M_{MSB}=1$;而补码的判断条件则为$M_s \oplus M_{MSB}=1$。

2) 浮点数的规格化处理

在采用规格化浮点数表示的机器中,浮点数以规格化形式出现并进行传送和存储,参加运算的初始数据也是规格化数,但运算后结果可能超出规格化数的表示范围。此时需要同时调整浮点数尾数和阶码的大小,将其变回规格化数的形式。这一操作过程叫作浮点数的规格化处理,一般通过对尾数进行左/右移实现。在尾数移位的同时,阶码也需要进行增量或减量。具体操作又分为左规和右规两种,下面分别加以介绍。

(1) 右规。浮点运算有可能发生尾数溢出，即尾数的数值超出小数表示范围冲到了数符位中。这在定点运算中是不允许的，属于溢出出错。但在浮点运算时这不算真正的溢出，可通过向右进行规格化的操作，将溢出的尾数调整回正常表示范围中来，简称为右规。由于尾数用小数表示，如果发生溢出只可能破坏到小数点左边一位，因此右规时只需将尾数右移一位，阶码加 1 即可。

(2) 左规。如果运算后尾数的绝对值小于规格化尾数值时，可通过向左进行规格化的操作，将尾数调整回规格化尾数的表示范围中来，简称为左规。与右规不同的是，左规过程中尾数可能需要左移多位才能回到规格化数的范围。此时尾数每左移一位，阶码减 1，直到符合规格化数的判断条件为止。

2. 浮点机器零的表示

在浮点表示的计算机中，运算后需要将结果作为 0 处理的情况会比较复杂，主要有以下几种。

(1) 真值为 0(真 0)。

(2) 尾数为 0，阶码不为 0 且未达最小值(最负阶)。

(3) 阶码已达最小值，尾数仍为非规格化数($0 \leqslant |M| < 1/2$)形式。

(4) 阶码已小于可表示的最小值，尾数仍不为 0。

除真 0 以外，不论出现上述哪种情况，都说明数值已经小到无法用规格化浮点数表示出来的地步，在采用规格化浮点数表示的机器中只能作为 0 处理(当然，在允许非规格化数表示的机器中，情况 3 可不看作 0)。而这种 0 和真值 0 是有区别的，故称为“机器 0”。为了保证浮点表示时 0 的形式上的统一性，通常规定机器零的标准格式为：尾数为 0，阶码为最小值(最负阶)。另外，为了便于判零操作的实现，希望机器零的代码为一串零的形式，因此许多计算机中规定浮点数的阶码用移码表示，尾数用补码表示。

3. 浮点数的表示范围

不管是定点数还是浮点数，所谓数的表示范围实际上指的只是数值可达的上、下限边界。实际上，机器数的每个值都只对应数轴上的一个点，点与点之间是离散的，而不是一段连续的区间。定点整数的各个点在数轴上的分布是均匀的；而浮点数的各个点在数轴上的分布是不均匀的，越靠近数轴的原点，两个相邻数之间的距离就越近。虽然浮点数表示范围的大小主要取决于阶码的表示范围，但是如果要确定出表示范围的上下限，则不仅要考虑阶码的位数和所用的码制，也要考虑尾数的位数和码制。

描述浮点数表示范围的关键方法是找准几个边界点。假设阶码用移码表示，尾数用补码表示(阶移尾补格式)，则对于规格化浮点数来说，当阶码达到移码可表示的最大值、尾数达到补码可表示的最大值时，该浮点数为可表示的最大正数，对应的机器码为 1,11…1；0.111…11，十进制形式为

$$N_{\text{最大正数}} = (1 - 2^{-n}) \times 2^{2^k - 1} \tag{5-56}$$

当阶码达到移码可表示的最小值、尾数达到规格化正数补码最小值时，该浮点数为最小正数，机器码为 0,00…0；0.100…00，十进制形式为

$$N_{\text{最小正数}} = 2^{-1} \times 2^{-2^k} \tag{5-57}$$

这样，我们就确定了阶移尾补规格化浮点数的正数可表示范围

$$2^{-1}\times 2^{-2^k}\sim(1-2^{-n})\times 2^{2^k-1} \tag{5-58}$$

同样,当阶码达到移码可表示的最大值、尾数达到补码可表示的最小值(最负数)时,该浮点数为最小负数(即绝对值最大的负数),对应的机器码为 1,11…1; 1.000…00,写成十进制形式为

$$N_{\text{最小负数}}=-1\times 2^{2^k-1} \tag{5-59}$$

当阶码达到移码可表示的最小值、尾数达到规格化补码可表示的负数最大值时,该浮点数为可表示的最大负数(即绝对值最小的负数),对应的机器码为 0,00…0; 1.011…11,写成十进制形式为

$$N_{\text{最大负数}}=-(2^{-1}+2^{-n})\times 2^{-2^k} \tag{5-60}$$

则阶移尾补规格化浮点数的负数可表示范围也确定出来了,即

$$-1\times 2^{2^k-1}\sim-(2^{-1}+2^{-n})2^{-2^k} \tag{5-61}$$

将上述正、负数表示范围进行综合,我们可以详细地分三段写出阶移尾补规格化浮点数的表示范围

$$-1\times 2^{2^k-1}\sim-(2^{-1}+2^{-n})2^{-2^k},\quad 0,\quad 2^{-1}\times 2^{-2^k}\sim(1-2^{-n})\times 2^{2^k-1} \tag{5-62}$$

注意,阶移尾补格式的规格化浮点数正负可表示范围是不对称的。如果不要求详细地表明正、负数的可表示范围,上述范围也可粗略地描述为

$$-1\times 2^{2^k-1}\sim(1-2^{-n})\times 2^{2^k-1} \tag{5-63}$$

这种表示方法更清楚地反映出阶移尾补规格化浮点数正负域的不对称。为了直观起见,我们可以在数轴上标出这个范围,如图 5-49(a)所示。

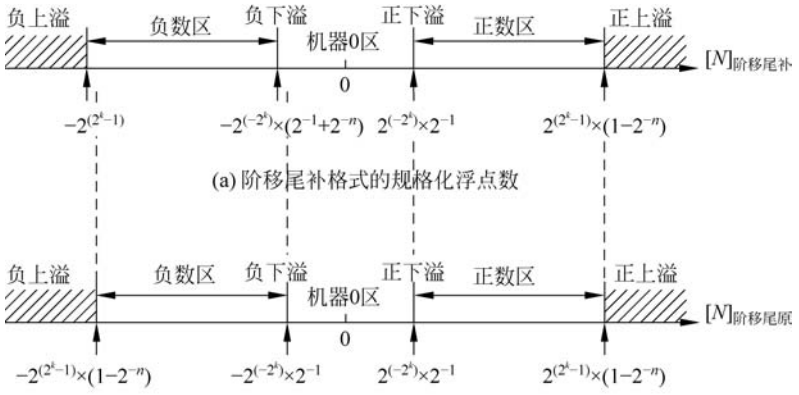


图 5-49　规格化浮点数的表示范围

采用不同的码制来表示浮点数时,由于各码制的定义域不一样,会导致表示范围的边界点不同。例如,阶移尾原规格化浮点数的表示范围为

$$-(1-2^{-n})\times 2^{2^k-1}\sim-2^{-1}\times 2^{-2^k},\quad 0,\quad 2^{-1}\times 2^{-2^k}\sim(1-2^{-n})\times 2^{2^k-1} \tag{5-64}$$

或粗略地用上下限表示成:

$$-(1-2^{-n})\times 2^{2^k-1}\sim(1-2^{-n})\times 2^{2^k-1} \tag{5-65}$$

上述表示范围对应的数轴如图 5-49(b)所示。将该图(b)与图(a)对比可知,当采用阶

移尾原格式时，规格化浮点数的正数域与阶移尾补格式相同；而负数域则稍微右移了一个点（即缩小了一个点），与正数域对称。

从图 5-49 还可以看出，浮点数的表示范围由正数域、机器 0 和负数域三个区域组成。比较特殊的是机器 0 区域。这段区域中只有数轴中心的坐标原点表示真 0，而其他部分都是由于机器表示范围的限制才被当成 0 看待的。换句话说，由几何表示来看真 0 与机器 0 的区别，就在于真 0 是数轴上的一个点，而机器 0 则对应于数轴上 0 点附近的一段区间。

当浮点运算结果超出浮点表示范围时，需要及时发现错误并进行纠正。浮点数的溢出判断方法与定点数不同，是以阶码是否超出其表示范围来界定的，具体又分为阶码上溢和阶码下溢两种情况。当运算结果太大以至阶码大于浮点数可表示的最大阶时，称为浮点数的上溢。此时如果数据为正，为正上溢；如果数据为负，则为负上溢，如图 5-49 所示数轴上左、右两边的上溢区。不管是正上溢还是负上溢，都属于溢出出错。运算结果一旦上溢，计算机必须立即停止运算，置溢出标志，并转溢出中断程序进行处理。一般所说的浮点溢出错误，均是指发生上溢的情况。另一方面，运算结果也有可能太小，其表现形式为运算结果非 0，但阶码小于最小阶（最负阶）。这种情况叫作浮点数的下溢。此时如果数据为正，称为正下溢；数据为负则称为负下溢。下溢时，浮点数的值小到接近于零且无法正常表示，跳转到了图 5-48 所示的机器 0 区间中去（0 点的左边为负下溢区，右边为正下溢区）。故运算时如出现下溢，计算机一般不看作出错，仅作为机器零处理即可。

浮点数判溢出的方法还与阶码的码制有关，如用补码或移码表示阶码，则设置双符号位来判溢出十分方便，具体实现方法见移码加减运算（见 5.5.5 节）的讨论以及定点补码加减运算关于溢出判断方法（见 5.3.2 节）的介绍。

【例 5.7】 某机字长 32 位，其浮点数采用规格化阶移尾补格式，其中阶码 8 位（含 1 位阶符），尾数 24 位（含 1 位数符）。请写出该浮点数的表示范围，并指出可表示的最小正数和绝对值最小的负数值各是多少？

解：让我们根据题意找出该浮点数表示范围中正、负域的边界点，为便于确定先写出这些边界点的机器码，再写成更加直观的十进制真值形式。

	机器码	真值
最大正数	1,111 1111；0.$\underbrace{111\cdots11}_{23\text{ 个 }1}$	$(1-2^{-23})\times2^{127}$
规格化最小正数	0,000 0000；0.$\underbrace{100\cdots00}_{22\text{ 个 }0}$	$2^{-1}\times2^{-128}$
规格化最大负数	0,000 0000；1.$\underbrace{011\cdots11}_{22\text{ 个 }1}$	$-(2^{-1}+2^{-23})\times2^{-128}$
最小负数	1,111 1111；1.$\underbrace{000\cdots00}_{23\text{ 个 }0}$	-1×2^{127}

则该浮点数的表示范围为 $-1\times2^{127}\sim(1-2^{-23})\times2^{127}$，其中可表示的规格化最小正数为 $2^{-1}\times2^{-128}$，绝对值最小的规格化负数为 $-(2^{-1}+2^{-23})\times2^{-128}$。由此例可看出阶移尾补格式的规格化浮点数正、负表示范围不对称的情况。

若想在格式不变的情况下进一步扩大浮点数的表示范围，可采用非规格化浮点数表示。但与规格化数相比，虽然其正、负数表示范围有所扩大，但总的表示范围不会得到改善，其效

果并不比增加阶码位数来得明显,反而会使有效数位的利用率降低,由此引起的精度损失变大。因此计算机中一般均采用规格化浮点数。作为非规格化浮点数的举例,如果仍采用阶移尾补格式,非规格化最小正数的机器码可达0,00…0;0.000…01,十进制形式为

$$N_{\text{非规格化最小正数}}=2^{-n}\times 2^{-2^k} \tag{5-66}$$

非规格化最大负数的机器码为0,00…0;1.111…11,十进制形式为

$$N_{\text{非规格化最大负数}}=-2^{-n}\times 2^{-2^k} \tag{5-67}$$

则阶移尾补非规格化浮点数的表示范围为

$$-1\times 2^{2^k-1}\sim -2^{-n}\times 2^{-2^k},0,2^{-n}\times 2^{-2^k}\sim (1-2^{-n})\times 2^{2^k-1} \tag{5-68}$$

由式(5-68)可以看出,阶移尾补非规格化浮点数总的表示区间大小与阶移尾补规格化浮点数相同,并没有扩大。非规格化浮点数扩展的范围集中在0点附近,使得机器0区间缩小了一点,但由于只是尾数的范围变大了,阶码的范围并没有变,因此下溢区缩小的范围很有限。

从上述的讨论中可以看出,浮点数表示范围的大小主要取决于阶码的位数,阶码的位数越多,指数项取值越大,表示范围也越大,反之亦然。因此通过调整阶码的位数,可以有效地调整浮点表示的范围。但是在字长不变的情况下,阶码位数增加的同时会减少尾数的位数,导致表示精度降低。在计算机中具体运用时应权衡范围与精度两方面的利益。

4. 浮点数的隐藏位表示

采用规格化浮点数表示时,我们会发现非0规格化尾数的最高数值位取值其实是确定的,即绝对值必定为"1"。既然这样,计算机中是否还需要保存这一位呢?答案显然是否定的。许多机器在保存浮点数时对这一位数值进行了省略,即把浮点数存入内存前,通过尾数的左移强行把该位去掉。但去掉并不意味着就不要了,而是隐含约定该位数值总是在小数点之后,居于尾数的最高位。也就是说,该位仍然存在,只是"躲藏"起来不出现而已。这种处理方案被形象地称为"隐藏位"技术。采用隐藏位的好处是在尾数位数不增加的情况下,能多表示一位尾数,有利于提高数据表示的精度。

例如,10位字长的浮点数$[N]_{\text{阶移尾原}}=1,011;1.10101$在不采用隐藏位方案时尾数只有5位;而在同样字长的情况下,如果具有隐藏位,恢复成正常浮点数格式后就可表示为$[N]_{\text{阶移尾原}}=1,011;1.110101$,尾数达到了6位,精度也随之增加了1位。这个例子只是对隐藏位方案作的原理性示意,实际的隐藏位方案在不同的机器中实现时可能会有不同的规定,应灵活运用。当然,隐藏位只能在存储和传送时"隐藏",运算时仍然需要参加。因此,在采用隐藏位方案的机器中,如果需要从内存取出浮点数执行运算,则运算前必须先恢复隐藏位,而运算器的位数也要随之增加1位。

引入隐藏位概念后,如果不加说明,同一种浮点数格式可能会被理解为两种不同的取值,带隐藏位的或不带隐藏位的。为了避免二意性,也为了讨论起来更加方便,本书在此约定:除非特别说明,书中所有与浮点数格式相关的原理性讨论、例题、习题等均不考虑隐藏位。

5. 浮点基数的选择

我们前面对浮点数的讨论,都是以阶的基$R=2$为基础进行的,这是最简单的情况。而在计算机中,实际上还经常选取$R=4,8,16$等作为阶的基数。选择不同的基数对浮点数特

性的改变起着重要的作用，它既影响浮点数的表示精度，也影响数值的表示范围。采用较大的 R 值，在阶码位数相同的情况下，可以有效地扩大浮点数的表示范围，但此时即使尾数位数不变，表示精度也会下降。下面我们就来深入讨论一下不同的基数对浮点数的影响。

1）基数对规格化尾数的影响

在计算机中，不论阶的基数怎样选取，浮点数的阶码和尾数仍然保持二进制形态，运算也仍然按二进制规则进行，但此时尾数的基值已经随着阶码基数的不同而改变了。在浮点表示中，一般阶码与尾数的基数是保持一致的。在以 R 进制为基数的浮点数中，当尾数的位数为 n 位二进制时，就相当于 R 进制的尾数有 n' 个数位，其中 $n'=n/\lceil\log_2 R\rceil$。因此如果进行移位操作，就不能再按二进制进行了。而规格化表示也是和基数 R 有关系的，由式(5-54)已知，规格化浮点数的尾数 $|M|$ 应在 $\left(\frac{1}{R},1\right)$ 范围内，如果 R 改变，规格化数的定义也随之改变。具体分析如下：

(1) $R=4$ 时，两位尾数相当于一位四进制码，规格化尾数的绝对值小于或等于 1/4，即原码尾数的最高两位不全为 0 时是规格化数，补码则要求最高两位尾数至少有一位与符号位不同。阶码+1 或−1，尾数要对应地右移或左移两位。

(2) $R=8$ 时，三位尾数相当于一位八进制码，规格化尾数的绝对值小于或等于 1/8，原码尾数的最高三位不全为 0 时是规格化数，补码规格化尾数的最高三位至少有一位与符号位不同。阶码+1 或−1，尾数要对应地右移或左移三位。

(3) $R=16$ 时，四位尾数相当于一位十六进制码，规格化尾数的绝对值小于或等于 1/16，原码尾数的最高四位不全为 0 时是规格化数，补码规格化尾数中最高四位至少有一位与符号位不同。阶码+1 或−1，尾数要对应地右移或左移四位。

依此类推，不难得到 $R=2^n$ 时的情况，尾数每左移(或右移)$\lceil\log_2 R\rceil$ 位，阶码将减 1(或加 1)。

又如现在尾数 $M=0.0001X\cdots X$，对于 $R=2$ 的浮点数来说，这是一个非规格化数，需要进行规格化操作；而对于 $R=16$ 的浮点数来说，这已是一个规格化数了，无须再进行规格化操作。由此看来选取较大的基数在运算时可以减少规格化操作的次数和移位的次数，有利于提高运算速度。

2）基数对表示范围的影响

浮点基数的选取对其表示范围影响很大。一般而言，基数大能使可表示数的个数增加，表示范围增大。例如，对于 32 位的浮点数来说，如果阶码 7 位(含 1 位阶符)，尾数 25 位(含 1 位数符)，采用阶移尾补形式，则

当 $R=2$ 时，可表示的浮点数范围为 $-1\times 2^{63}\sim(1-2^{-24})\times 2^{63}$；

当 $R=8$ 时，可表示的浮点数范围为 $-1\times 8^{63}\sim(1-2^{-24})\times 8^{63}$；

当 $R=16$ 时，可表示的浮点数范围 $-1\times 16^{63}\sim(1-2^{-24})\times 16^{63}$。

可知，随着基数的变大，浮点数表示范围有沿数轴向正负两边扩展的趋势。

3）基数对表示精度的影响

基数的选取对浮点数表示精度也会有显著的影响。这里所谓的精度是指一个数所含有效数值位的位数。随着 R 的增大，浮点数尾数的粒度会随之变粗，即数轴上各点的排列更稀疏，表示精度随之下降。仍以上述格式的 32 位浮点数为例来看一下精度的具体变化

情况。

(1) 当$R=2$时,可表示24位二进制尾数,规格化最小正数为$2^{-1}\times2^{-64}$,尾数的最高位为有效数值。

(2) 当$R=8$时,24位二进制尾数相当于8位八进制数,规格化最小正数为$8^{-1}\times8^{-64}$,尾数的最高两位为0。

(3) 当$R=16$时,24位二进制尾数相当于6位十六进制数,规格化最小正数为$16^{-1}\times16^{-64}$,尾数的最高三位为0。

通过上例我们看出,基数越大,规格化尾数的最高位利用率越低,数据在数轴上的分布密度越稀,精度受影响越大。可知基数增大对浮点数表示范围有着正面的影响,但对表示精度却带来负面的影响。为了扬长避短,在机器字长较长的巨、大、中型机中,浮点数的基数宜取大些;而在字长较短的微、小型机中,浮点基数宜取小些。如PDP-11机浮点基数为2,而IBM 370机的浮点基数取16,虽然两者的短浮点数具有同样的格式,但由于所用基数不同,IBM 370的表示范围比PDP-11大,当然相对误差也较大。

最后可以对浮点表示法的基本特性进行如下总结:浮点数的表示范围主要由阶码的位数决定,有效数字的精度主要由尾数的位数决定,基数的大小则对浮点数的表示范围和精度均有影响。阶码的位数越多,表示的范围越大;尾数的位数越多,可表示的精度越高;基数越大,表示范围扩大的越多,但精度却随之单调下降。这些特性都是由浮点数本身的结构决定的。

6. 定点表示法与浮点表示法的比较

为了加深对不同数据表示法性能的理解,我们对同样字长的定点数和浮点数进行一下简单的比较,概括地说明两种表示法在以下几方面有所不同:

(1) 由于浮点数加入了指数部分(即阶码),表示范围得到了扩充,因此比定点数表示范围大得多。

(2) 浮点数的有效精度比同样字长的定点数低。一般来说,机器字长越长,可表示的有效位数就越多,精度就越高。但由于浮点数的阶码占去了一部分位数,可表示的有效数位减少了,因此精度也随之下降。从这一点上看,浮点表示法的实质是以牺牲精度为代价来扩大数据的表示范围的。

(3) 溢出判断的方法不同,浮点运算以阶码上溢为溢出标志、而定点数则是以数值本身是否超出表示范围对符号位造成破坏为溢出标志的。

(4) 浮点数由阶码和尾数两部分组成,比定点数结构复杂,导致运算步骤增多,运算速度下降,运算线路也比定点运算器复杂。

(5) 由于表示范围大,浮点数在运算时能够有效减少运算结果产生溢出的次数,减轻编程时选择比例因子的工作量。

总的看来,浮点数在数据表示范围、溢出处理和编程的方便性上均优于定点数,而在表示精度、运算速度及运算的复杂性等方面又不如定点数,因此,在确定机内数据表示方式时应根据具体应用方向综合考虑。

7. 浮点表示法对计算机性能结构的影响

虽然浮点表示法在表示范围等方面具有较为明显的优势,但表示方法不如定点数简单直观,运算规则和硬件结构都较为复杂,实现成本也随之增加。这些因素均说明并不是所有

的计算机都适合采用浮点数。机内是否采用浮点表示，在计算机设计时必须统筹安排、全面权衡、合理配置。按照对浮点功能的设置方式，通常可以将计算机分为以下几种。

1）定点机

定点计算机中只能表示和处理定点数，它的指令系统不支持任何浮点功能，只能通过执行软件程序来实现对实数的处理。机器结构简单，成本低廉，适用于规模较小、运算功能要求不高的低档微型计算机及某些专用机、控制机结构。

2）定点机＋浮点选件

为了增强硬件的实数处理能力，许多性能较好的定点微、小型计算机中都配有浮点协处理器选件。浮点协处理器是一种专门用来对计算机中的浮点数进行处理的 LSI 芯片，其内部主要包括浮点运算部件和浮点处理部件两部分，可在定点机原有指令系统的基础上执行一套附加的浮点指令，完成对定点机浮点功能的扩展。之所以叫作“协处理器”，是因为这种处理部件虽然本身也有自己的指令集，但功能专一，无法单独工作，只能和主 CPU 协同运行。流传较广的有与 80x86 CPU 配套的 80x87 系列浮点协处理器。计算机系统在配置了浮点协处理器选件之后，可使浮点运算的速度大大提高。

3）浮点机

浮点计算机本身就具有浮点运算指令和浮点运算器，实数处理速度和处理能力都十分强大，通用的大、中型机、高档微型机多采用浮点机结构。

【例 5.8】 设计一个浮点数格式，要求必须满足下列条件。

（1）所用的机器码位数最少；

（2）可表示的十进制数范围：

负数 $-10^{38}\sim-10^{-38}$；　正数 $+10^{-38}\sim+10^{38}$

（3）可达到的精度：7 位十进制数据。

解：（1）浮点数表示范围的设计主要是确定阶码位数，采用试探法求解：

由 $2^{10}>10^3$，　可得 $(2^{10})^{12}>(10^3)^{12}$，　即 $2^{120}>10^{36}$

又因 $2^7>10^2$，　所以 $2^7\times2^{120}>10^2\times10^{36}$，　即 $2^{127}>10^{38}$

同理可得　$2^{-127}<10^{-38}$

由上推导可知：阶码取 8 位（含 1 位阶符），且用补码或移码表示时，对应的浮点数取值范围为 $2^{-128}\sim 2^{+127}$，可以满足题意要求。

（2）浮点数精度的设计主要是确定尾数的位数。

由于 $10^7\approx2^{23}$，故尾数的数值部分可取 23 位，加 1 位数符位达 24 位。

最终该浮点数至少取 32 位，其中阶码 8 位（含 1 位阶符），尾数 24 位（含 1 位数符）。

5.5.4　IEEE 754 浮点标准

通过上面对浮点数表示方法和原理的讨论，我们对浮点数有了一个基本的认识。为了进一步加深对浮点表示法的理解，现在以广泛流行的 IEEE 754 国际标准浮点数为例，来看看实际的浮点数格式。

1. IEEE 浮点格式

IEEE 754 浮点标准由 IEEE（电气和电子工程师协会）于 1985 年提出，主要为便于不同计算机系统之间的数据交换、协同工作及软件移植，浮点数的格式应该有统一的标准定义而

设计。IEEE 754 标准浮点数的格式与我们前面 5.5.1 节介绍的浮点数基本格式有一些差别,如图 5-50 所示。

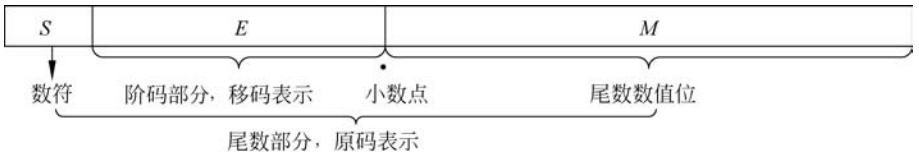


图 5-50 IEEE 754 标准的浮点数格式

为了便于讨论,我们把上述 IEEE 754 标准浮点数的格式书写为

$$S,E_sE_1E_2\cdots E_k;.M_1M_2\cdots M_n$$

概括起来,IEEE 754 标准对浮点数的格式作了如下规定(以 32 位为例)。

(1) 表示格式由数符 S、阶码 E、尾数 M 三部分组成。

(2) 数符位 S 占 1 位,设在浮点格式的最左边(最高位)。S 的取值按原码符号位的规定,$S=0$ 表示正数,$S=1$ 表示负数。

(3) 阶码 E 为 8 位整数,用移码表示,偏置常数为+127,阶的基为 2。

(4) 尾数值 M 为 23 位小数,用原码表示。

(5) 非 0 规格化尾数的最高有效位恒为 1,采用隐藏位方案,约定该位总是躲藏在尾数小数点的左边(即位权为 2^0,是一位整数)。因此尾数值实际上可达 24 位(1 位隐藏位+23 位小数位)。

IEEE 754 标准中设定了三种常用的浮点数格式,其数位的具体分配如表 5-11 所示。

表 5-11 IEEE 754 标准中的三种浮点数

类型	数符 S	阶码 E	隐藏位	尾数值 M	总位数	偏置常数	
						十六进制	十进制
短实数	1	8	有	23	32	7FH	+127
长实数	1	11	有	52	64	3FFH	+1023
临时实数	1	15	无	64	80	3FFFH	+16383

表中短实数又称为单精度浮点数,长实数又称为双精度浮点数,它们都采用了隐藏位方案,这样实际上又增加了一位尾数。临时实数又称为扩展双精度浮点数,它不含隐藏位,常用于中间计算过程,以减少相应的精度损失。

2. 极端值的特殊规定

IEEE 754 标准用阶码 E 的两个极端值 0 和全 1 表示一些特殊的情况,主要有下面几点:

(1) 阶码 E 为 0 且尾数 M 为 0 表示浮点 0。$S=0$ 表示+0,$S=1$ 表示−0。

(2) 阶码 E 为 0 且尾数 M 不为 0,若隐藏位=0,表示非规格化数。$S=0$ 表示正非规格化数,$S=1$ 表示负非规格化数。

(3) 阶码 E 为全 1 且尾数 M 为 0 表示∞,$S=0$ 表示+∞,$S=1$ 表示−∞。

(4) 阶码 E 为全 1 且尾数 M 不为 0,表示非数(即不是一个数),非数常用来通知各种异常条件。

3. 舍入处理方法

IEEE 754 标准中还给出了四种可供选择的舍入处理方法。

(1) 就近舍入：运算结果可被舍入成最接近的可表示数，这是该标准默认的舍入方式，其实质类似于前面介绍的“0 舍 1 入”法。例如，假设运算结果比规定的尾数位数多出 5 位，若多余位取值是 10010，已超过尾数最低有效位(LSB)权值的一半，故最低有效位加 1，相当于“入”；若多余位取值是 01111，则简单的截尾即可，相当于“舍”；若多余位取值是 10000 这种严格中点的情况，则需要根据最低有效位的取值来决定取舍。若最低有效位＝0 则截尾；最低有效位＝1 则作加 1 操作。

(2) 朝 0 舍入：就是简单的截尾，丢掉多余位后不作任何进一步的处理。其舍入后果总是使结果的绝对值变小(单向负误差)，朝向数轴原点趋近。

(3) 朝＋∞舍入：朝正无穷大方向向上舍入。若尾数为正数，只要舍去的多余位不全为 0，做最低有效位加 1 操作；若尾数为负数，则简单的截尾。

(4) 朝－∞舍入：结果朝负无穷大方向向下舍入。处理方法正好与朝＋∞舍入相反，对正数来说，做简单的截尾；对负数而言，只要多余位不全为 0，做最低有效位加 1 操作。

4. 主要表示特点概括

综上所述，IEEE 754 标准浮点数主要有下述特点：

(1) 符号位放在浮点数格式的最高位，便于判 0 和判断符号操作的实现。

(2) 机器零为一串 0 的形式(E 和 M 均为零)，便于判 0。

(3) 两浮点数进行大小的比较时，可不必区分阶码位和数值位，视同两定点数比较一样对待。

(4) 设置隐藏位提高尾数表示精度，且隐藏的数值为 1 而不是通常的 2^{-1}。

(5) 阶的移码采用 2^k-1 而不是通常的 2^k 作为偏置常数，通过对 0 或全 1 极端阶码值的充分利用，增加了表示的多样性。

现以单精度浮点数格式为例，32 位的浮点规格化数的真值可表示为

$$N=(-1)^S\times(1.M)\times 2^{E-127} \tag{5-69}$$

其中，S 代表符号位，$S=0$ 表示正数，$S=1$ 表示负数；E 为阶码的移码；M 为尾数的小数部分；小数点前面的 1 为隐藏位。由于正常的阶码其移码值 E 可取 1～254，则短浮点数阶码的真值范围为－126～＋127。

【例 5.9】 写出 IEEE 754 标准短浮点数和长浮点数的表示范围，并进一步指出其规格化可表示的最小正数、最大负数为多少？

解：(1) 短浮点数。

根据 IEEE 754 标准短浮点数的格式规定，其阶码 8 位，移码取值范围为 0～255。由于正常阶码的移码值 E 只能取 1～254，则短浮点数阶码的真值范围为－126～＋127。短浮点数的尾数 23 位，则

$$\text{最大正数}=2^{127}\times(2-2^{-23}),\quad \text{最小正数}=2^{-126}$$

$$\text{最大负数}=-2^{-126},\quad \text{最小负数}=-2^{127}\times(2-2^{-23})$$

IEEE 754 标准短浮点数表示范围为：$-2^{127}\times(2-2^{-23})\sim 2^{127}\times(2-2^{-23})$，用 10 的幂可近似表示为：$10^{-38}\sim10^{38}$。

(2) 长浮点数。

长浮点数的格式规定阶码11位,移码取值范围为0～2047。而正常阶码的移码值E可取1～2046,则长浮点数阶码的真值范围为－1022～＋1023。长浮点数的尾数52位,其规格化数的真值可表示为$N=(-1)^S\times(1.M)\times2^{E-1023}$,则

$$\text{最大正数}=2^{1023}\times(2-2^{-52}),\quad \text{最小正数}=2^{-1022}$$

$$\text{最大负数}=-2^{-1022},\quad \text{最小负数}=-2^{1023}\times(2-2^{-52})$$

IEEE 754标准长浮点数表示范围为:$-2^{1023}\times(2-2^{-52})\sim 2^{1023}\times(2-2^{-52})$。

通过上述例题可知,在使用IEEE 754标准浮点数时,需要特别注意隐藏位的存在。整数部分的"1"被隐含表示,在格式中不留任何痕迹。

5.5.5 规格化浮点加减运算

在讨论了浮点数的表示方法之后,让我们进一步讨论浮点数的算术四则运算方法。由于浮点数的表示方法比定点数复杂得多,因此运算方法也相对复杂。出于习惯,我们对于浮点算法的讨论仍然按先加减、后乘除的顺序展开。由于计算机中普遍采用规格化浮点数,因此对浮点算法的讨论主要围绕规格化浮点运算展开。所谓规格化浮点运算是指,参加运算的操作数为规格化浮点数,运算完成后结果也是规格化浮点数。

1. 浮点加减运算基本规则

首先对浮点数阶码和尾数两部分进行分析,可知有效数值是由尾数给出的,则加减运算应在尾数部分进行。这里遇到的第一个问题是加减运算要求操作数的小数点对齐,虽然所有浮点数格式中尾数的小数点位置是一致的,但这并不能代表其小数点的实际位置。浮点数小数点的实际位置是由阶码的大小决定的。当两浮点数阶码不等时,其小数点的实际位置是不一样的,也就是小数点未对齐,此时尾数无法直接进行加减运算。故浮点加减运算的第一步必须使参加运算的两浮点数阶码相等,这一操作称为"对阶"。对阶之后,尾数就可以进行加减运算了。由于尾数的表示方式与定点小数完全相同,因此其加减运算规则也与定点小数完全相同。尾数运算完成后,还可能会涉及运算结果的规格化、精度的舍入、溢出判断等一系列问题需要进一步解决。另外,由于浮点运算比定点加减运算更费时间,因此希望尽量简化运算过程,当操作数为0时,可以考虑不实施运算。按照上面的分析,可以将浮点加减运算步骤归纳为下面几步:

(1) 0检测,判断操作数是否为0;

(2) 对阶,使两数的小数点位置对齐;

(3) 尾数运算,两尾数按定点加减运算规则求和(差);

(4) 结果规格化,若运算后尾数不再是规格化形式,进行规格化处理;

(5) 舍入,为尽量避免精度损失,对尾数右移时移出的保护位进行舍入;

(6) 溢出判断,运算过程中对可能发生溢出出错的操作及时判溢出。

现设有两个浮点数X和Y,分别表示为$X=M_x\times R^E$,$Y=M_y\times R^E$,若求$Z=X\pm Y$,则浮点加减运算的基本规则可用通式描述为:

$$Z=X\pm Y=\begin{cases}(M_x\pm M_y\times 2^{-(Ex-Ey)})\times 2^{Ex}, & E_x\geqslant E_y\\ (M_x\times 2^{-(Ey-Ex)}\pm M_y)\times 2^{Ey}, & E_x<E_y\end{cases}\tag{5-70}$$

式中，$2^{-(Ex-Ey)}$ 和 $2^{-(Ey-Ex)}$ 称为移位因子，反映对阶时尾数右移的位数。

下面让我们详细讨论每一步运算的具体操作方法。

2. 浮点加减运算步骤

1）0 操作数检测

运算前先分别对两操作数 X、Y 进行判 0 操作

若 $X=0$，则令 $X+Y=Y$；$X-Y=-Y$；运算结束；

若 $Y=0$，则令 $X+Y=X-Y=X$；运算结束。

2）对阶

对阶的目的是使两操作数的小数点对齐，即使得两数的阶码相等。对阶前，先要进行阶码比较，确定两数的阶码是否一致。由于运算器中通常不设比较电路，因此阶码的比较是通过“求阶差”操作完成的。阶差可描述为

$$\Delta E = E_x - E_y \tag{5-71}$$

若 $\Delta E=0$，说明 $E_x=E_y$，尾数可直接进行加减；

若 $\Delta E\neq 0$，则意味着 $E_x\neq E_y$，需要先对阶，阶码对齐后才能进行运算。

求阶差时的减法运算采用何种算法与阶码所用的码制有关，若阶码用补码表示，则可使用定点整数补码加减法进行计算。即

$$[\Delta E]_{补}=[E_x]_{补}+[-E_y]_{补} \tag{5-72}$$

其次要解决阶码按怎样的规则对齐的问题。若大阶向小阶看齐尾数需要左移，对规格化数来说会引起尾数溢出，显然不可取；而小阶向大阶看齐则需要尾数右移，虽然可能损失精度但对尾数的正确性没有大的影响，故规定对阶的规则为：小阶向大阶看齐。

对阶的操作过程一般采用逐步逼近的方式，小阶每+1，其尾数右移一位，$\Delta E+1$ 或 -1 向 0 趋近一步，反复进行到 $\Delta E=0$ 为止。为了尽量减少精度损失，对阶时尾数最低位移出的多余位可通过在运算器中设置保护位寄存器的方法，先把移出的前几位数值暂时保存起来，以便左规时再补入最低有效位，或供舍入判断之用。对阶过程具体描述如下：

$\Delta E>0$，$E_x>E_y$，E_y+1，$M_y\rightarrow 1$，$\Delta E-1$，重复到 $\Delta E=0$ 为止。此时 $E_z=E_x$；

$\Delta E<0$，$E_x<E_y$，E_x+1，$M_x\rightarrow 1$，$\Delta E+1$，重复到 $\Delta E=0$ 为止。此时 $E_z=E_y$。

3）尾数相加减

对阶完成后即可进行尾数运算，按定点小数加减运算规则求 $M_z=M_x\pm M_y$。

当尾数用补码表示时有：

$$[M_z]_{补}=[M_x\pm M_y]_{补}=\begin{cases}[M_x]_{补}+[M_y]_{补} & \text{求和}\\ [M_x]_{补}+[-M_y]_{补} & \text{求差}\end{cases}$$

4）结果规格化

采用规格化表示时，虽然参加运算的操作数都是浮点规格化形式，但运算后结果有可能超出规格化数的表示范围，此时应对结果进行规格化处理，将其变回到规格化形式。浮点加减运算结果可能发生两种非规格化情况：

（1）尾数发生溢出，此时只要右规一次，尾数右移 1 位，阶码+1 即可；

（2）尾数非 0，未发生溢出，也不满足规格化判断条件。此时需要左规，而且还有可能需要左规多位。左规时尾数每左移 1 位，阶码−1，重复进行到满足规格化条件为止。随着尾

数逐位左移,可以考虑将保护位逐位补入尾数的最低位。规格化处理的具体规则参见前面5.5.3节的详细介绍。

5) 舍入

对结果进行规格化处理后,可能还会剩余有若干保护位不能补充回正式的尾数位中,需要舍去。为了把结果的精度损失降为最低,通常采用特定的舍入规则对尾数进行舍入。常用的舍入方法在5.3.3节误差处理的讨论中作过介绍,运算时可根据具体情况选用。如无特殊要求我们在此约定:本节对浮点运算结果的舍入处理均采用"0舍1入"法。

6) 溢出判断

规格化浮点数溢出的概念及其处理方法在讨论浮点数表示范围时曾作过介绍(见5.5.3节),一般以阶码是否溢出作为判断条件。浮点加减运算过程中有可能导致阶码上溢的操作是右规和舍入。由于右规过程中尾数右移一位时阶码加1,若阶码已为最大值再加1就会上溢。而在采用"0舍1入"法时,如果需要"入"则尾数末位加1。若尾数已达最大值,加1后会使尾数溢出,引起右规……接下来的情形就和右规时发生阶码上溢的情形一样了。因此当这两步操作进行时,需要及时地判溢出。注意,溢出判断操作不是在浮点加减运算结束时完成,而是在运算过程中需要的步骤立即进行。

3. 浮点加减运算举例

【例5.10】 设 $X=2^{-1010}\times(-0.1010\ 1000)$,$Y=2^{-1000}\times(+0.1110\ 1101)$,并设阶符2位,阶值4位,数符2位,尾值8位,均采用补码表示,求 $X\pm Y$。

解:由

$$X=2^{-1010}\times(-0.1010\ 1000),\quad Y=2^{-1000}\times(+0.1110\ 1101)$$

得

$$[X]_{补}=11,0110;\ 11.0101\ 1000\neq 0,\quad [Y]_{补}=11,1000;\ 00.1110\ 1101\neq 0$$

(1) 对阶。

$$[\Delta E]_{补}=[E_x]_{补}-[E_y]_{补}=11,0110+00,1000=11,1110$$

$\Delta E=-2$,$E_x<E_y$,则$[X]_{补}$的尾数右移2位,阶码加2,使 $E_x=E_y$,即

$$[X]'_{补}=11,1000;11.11010110$$

(2) 尾数相加减。

$$[M_x]'_{补}+[M_y]_{补}=11.11010110=00.11000011$$

$$\underline{+\ 00.11101101}$$

$$00.11000011$$

$$[M_x]'_{补}+[-M_y]_{补}=11.11010110=10.11101001$$

$$\underline{+\ 11.00010011}$$

$$10.11101001$$

即

$[X+Y]_{补}=11,1000;\ 00.1100\ 0011$——已是规格化数

$[X-Y]_{补}=11,1000;\ 10.1110\ 1001$——数符位=10,尾数溢出需右规

(3) 结果规格化。

右规后:

$$[X-Y]_{补}=11,1001;\ 11.0111\ 0100(1)$$

移出一位保护位,阶符=11,未溢出。

(4) 舍入处理。

采用“0 舍 1 入”法，按照负数补码舍入规则，应舍，则

$$[X-Y]_{补}=11,1001;\ 11.0111\ 0100$$

(5) 溢出判断。

本题在阶码可能发生溢出的(3)、(4)两步均未溢出，结果正确。

最后得 $[X+Y]_{补}=1,1000;\ 0.1100\ 0011$

$[X-Y]_{补}=1,1001;\ 1.0111\ 0100$

对应的真值为 $X+Y=2^{-1000}\times(+0.1100\ 0011)$

$X-Y=2^{-0111}\times(-0.1000\ 1100)$

此例中浮点数的阶码和尾数都采用了补码表示，因此对阶和加减运算使用定点补码加减算法即可。若阶码用移码表示，则对阶时求阶差需要用到移码加减算法。移码加减运算规则将在讨论浮点乘除运算时介绍。

5.5.6 规格化浮点乘除运算

与浮点加减运算不同，浮点乘除运算时不要求操作数的小数点对齐，阶码和尾数两部分可分别进行运算，运算过程中两者不发生关系。若两个浮点数相乘，阶码相加，尾数相乘；两个浮点数相除，阶码相减，尾数相除。当然也需要考虑算后的结果规格化、舍入、判溢出，以及算前的判 0 等操作。因此，浮点乘除运算步骤一般需要考虑以下几步：

(1) 0 检测，判断操作数是否为 0。

(2) 阶码运算，阶码相加(减)，求乘积或商的阶码。

(3) 尾数运算，尾数相乘(除)，求乘积或商的尾数。

(4) 结果规格化。

(5) 舍入。

(6) 溢出判断。

其中，(4)、(5)、(6)三步操作方法同浮点加减运算。

1. 移码加减运算

两个浮点数乘除的过程中，阶码需要进行加减运算。如果阶码用补码表示，则可按定点整数补码加减规则进行，这在前面我们已经介绍过。但是目前许多机器中使用移码来表示阶码，如 IEEE 754 标准采用阶移尾原的浮点数格式。这就对移码加减算法的探讨提出了要求。现在我们关心的是，两移码相加减能否直接得到和或差的移码？下面针对这一问题进行算法分析。

1) 算法推导

设两个 k 位的二进制整数 E_x 和 E_y，其移码可定义为

$$[E_x]_{移}=2^k+E_x \qquad -2^k\leqslant E_x<2^k$$

$$[E_y]_{移}=2^k+E_y \qquad -2^k\leqslant E_y<2^k$$

则

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

$$=[E_x+E_y]_{移}+2^k \tag{5-73}$$

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

$$=[E_x-E_y]_{移}-2^k \tag{5-74}$$

可见直接用移码进行相加,得到的结果并非移码,而是比和(差)的移码增加(减少)了2^k,需要对运算结果进行$+2^k$(-2^k)修正才能得到正确的移码。由于偏置常数2^k正好是移码符号位的位权,因此修正的方法很简单,无论$+2^k$还是-2^k,都只需要将结果的符号位变反即可。尽管如此,仍然希望能有更加简便的算法可供使用。一种常用的改进算法是利用补码与移码符号位相反的简单转换关系,通过加减补码来实现移码加减运算。算法推导如下。

设同样位数的补码为:

$$[E_y]_{补}=2^{k+1}+E_y \quad (\text{Mod } 2^{k+1})$$

则

$$\begin{aligned}[E_x]_{移}+[E_y]_{补}&=2^k+E_x+2^{k+1}+E_y=2^{k+1}+2^k+E_x+E_y\\&=2^k+E_x+E_y \quad (\text{Mod } 2^{k+1})\\&=[E_x+E_y]_{移}\end{aligned} \tag{5-75}$$

$$\begin{aligned}[E_x]_{移}-[E_y]_{补}&=[E_x]_{移}+[-E_y]_{补}=2^k+E_x+(2^{k+1}-E_y)\\&=2^{k+1}+2^k+E_x-E_y=2^k+(E_x-E_y) \quad (\text{Mod } 2^{k+1})\\&=[E_x-E_y]_{移}\end{aligned} \tag{5-76}$$

由上述推导结果可知,和或差的移码可通过移码与补码相加减直接获得。而运算前把移码转换为补码时,只要将其符号位变反即可。

2) 溢出判断

与补码加减运算一样,两个移码相加减结果也存在溢出出错的可能。那么移码运算又如何判断溢出呢?一种常用的方法是采用双符号位移码进行运算,但移码最高符号位的意义与变形补码的真符位有所不同。在此规定,无论正负移码的最高符号位(E_{s1})初始时均为0,而第二符号位(E_{s2})才表示移码的正负。因此运算开始前,移码两个符号位的组合含义为:

$E_{s1}E_{s2}=00$——表示负数;$E_{s1}E_{s2}=01$——表示正数。

运算完成后,对E_{s1}的取值进行判断,若E_{s1}仍然为0,表示无溢出,两符号位的组合含义不变;若E_{s1}为1,则表示发生了溢出。进一步可以根据两个符号位的组合值来确定溢出的方向,此时若用移码表示阶码则有:

- 若$E_{s1}E_{s2}=10$,阶码正溢出(上溢),E_{s2}由1破坏成0,溢出出错;
- 若$E_{s1}E_{s2}=11$,阶码负溢出(下溢),E_{s2}由0破坏成1,看作机器0。

移码采用双符号位参加运算时,对应的补码也要用双符号位参加运算。

2. 浮点乘除基本规则

设两浮点数$X=Mx\times RE$,$Y=My\times RE$,若求$Zp=X\times Y$和$Zq=X\div Y$,则浮点乘除运算的基本规则可描述为

$$Z_p=X\times Y=(M_x\times M_y)\times R^{E_x+E_y} \tag{5-77}$$

$$Z_q=X\div Y=(M_x\div M_y)\times R^{E_x-E_y} \tag{5-78}$$

3. 浮点乘除运算步骤

现在,让我们进一步讨论浮点乘除运算每一步的具体操作方法。

1) 0操作数检测

运算前首先检测是否$X=0$? $Y=0$?

若$X=0$,则令$Z_p=0$(乘法);或$Z_q=0$(除法),运算结束;

若 $Y=0$，则令 $Z_p=0$（乘法）；或 $Z_q=\pm\infty$（除法），此时也可视为运算异常，转非法除数处理，运算结束。

若 X、Y 均不为 0，则可继续进行下一步运算。

2）阶码相加减

若求 $X\times Y$，　　$E_p=E_x+E_y$

若求 $X\div Y$，　　$E_q=E_x-E_y$

如果阶码用补码表示，就按定点整数补码加减算法及溢出判断规则进行运算；若阶码用移码表示，则按移码加减算法和溢出判断规则进行运算。注意，阶码加减运算的结果有可能发生溢出，需要及时判断并转溢出处理。

3）尾数相乘除

若求 $X\times Y$，　　$M_p=M_x\times M_y$

若求 $X\div Y$，　　$M_q=M_x\div M_y$

如果尾数用补码表示，按定点小数补码乘除算法进行运算；如果尾数用原码表示，则要按定点小数原码乘除规则进行运算。有关算法（如加减交替除法）曾经在讨论定点乘除运算时介绍过（见 5.3.4 节、5.3.5 节）。注意按照定点乘法算法求出的乘积是双倍长度的，超长部分应先予以保留，以备接下来的左规或舍入操作使用。由于接下来会对结果进行规格化处理，因此除法过程可不必先判溢出。为了提高商的精度，可多上几位商用作保护位。

4）结果规格化

由于尾数是纯小数，相乘后结果不可能溢出，故不需右规。但乘积有可能变为非规格化数，此时需要左规。除法开始前若尾数满足 $M_x^*<M_y^*$（M_x^*、M_y^* 表示尾数的绝对值）的条件，结果不需右规，但可能需要左规；如果不满足 $M_x^*<M_y^*$ 的条件，则尾数会发生溢出，左右规都有可能需要进行。

5）舍入处理

不论乘法还是除法，在规格化操作完成后，都需要对尾数剩余的保护位进行舍入处理。按照本教材的约定，舍入操作均采用 0 舍 1 入法实现。

6）判断溢出

浮点乘除运算时溢出出错主要可能发生在阶码运算、结果规格化以及舍入处理等步骤，必须及时进行溢出判断，以免造成更大的错误。溢出判断方法与浮点加减运算相同，在此不再赘述。

4. 浮点乘除运算举例

为了使大家加深对浮点乘除算法的理解，下面举例说明其运算过程。

【例 5.11】 已知 $X=2^{-1001}\times(-0.1010\ 1001)$，$Y=2^{+0101}\times(+0.1001\ 1111)$，假设浮点数采用阶移尾补格式，阶值 4 位，尾值 8 位，求 $Z=X\times Y$。

解：$[X]_{\text{阶移尾补}}=0,0111;\ 1.0101\ 0111\neq 0$

$[Y]_{\text{阶移尾补}}=1,0101;\ 0.1001\ 1111\neq 0$

（1）阶码相加。

由于阶码用移码表示，因此采用移码加法较为方便。

$[E_z]_{\text{移}}=[E_x]_{\text{移}}+[E_y]_{\text{补}}=00,\ \ 0111=00,1100,\ \ E_{s1}=0,\ \ $ 无溢出

$$\begin{array}{r} +\ 00,0101 \\ \hline 00,1100 \end{array}$$

(2) 尾数相乘。

原则上讲可选任意一种定点补码乘法算法进行尾数运算,但为了加快运算速度,本题采用补码两位乘比较法。机器运算步骤同定点运算过程故省略。

$$[M_z]_{补} = 111.1001\ 0111\ 0000\ 1001\ 000$$

$$[Z]_{阶移尾补} = 00,1100;\ 111.1001\ 0111\ 0000\ 1001\ 000$$

(3) 结果规格化。

$$M_s \oplus M_{MSB} = 1 \oplus 1 = 0,$$

需要左规,即

$$[Z]_{阶移尾补} = 00,1011;\ 111.0010\ 1110\ 0001\ 0010\ 00$$

(4) 舍入处理。

取结果字长与原始操作数 X、Y 相同(单倍字长),采用 0 舍 1 入法,按负数补码的舍入规则,保护位的最高位为 0,应"舍",故

$$[Z]_{阶移尾补} = 00,1011;\ 111.0010\ 1110\underbrace{0001\ 0010\ 00}_{保护位}$$

$$= 00,1011;\ 111.0010\ 1110$$

(5) 溢出判断。

本题在阶码可能发生溢出的(1)、(4)两步均未溢出,结果正确。

最终结果为 $[Z]_{阶移尾补} = 0,1011;\ 1.0010\ 1110$

$$Z = X \times Y = 2^{-0101} \times (-0.1101\ 0010)$$

【例 5.12】 用浮点除法计算 $\left[2^{9} \times \left(-\frac{19}{32}\right)\right] \div \left[2^{-5} \times \left(+\frac{21}{32}\right)\right]$,假设浮点数采用阶移尾补格式,阶值 4 位,尾值 5 位,求 $Z = X \div Y$。

解:这道题需要先把十进制操作数转换为规格化二进制真值形式,再进一步转换为阶移尾补的浮点规格化机器数格式,然后才可进行机器运算。

$$X = \left[2^{9} \times \left(-\frac{19}{32}\right)\right]_{10} = [2^{+1001} \times (-0.10011)]_2 \neq 0$$

$$Y = \left[2^{-5} \times \left(+\frac{21}{32}\right)\right]_{10} = [2^{-0101} \times (+0.10101)]_2 \neq 0$$

$$[X]_{阶移尾补} = 1,1001,1.01101, [Y]_{阶移尾补} = 0,1011,0.10101$$

(1) 阶码相减。

采用移码减法进行运算

$$[E_z]_{移} = [E_x]_{移} + [-E_y]_{补} = 01,1001 = 01,1110, E_{s1} = 0, 无溢出$$

$$\begin{array}{r} +\ 00,0101 \\ \hline 01,1110 \end{array}$$

(2) 尾数相除。

采用补码加减交替除法,机器运算步骤同定点计算过程故省略。

$$[M_z]_{补} = 11.00011$$

$$[Z]_{阶移尾补} = 01,1110;11.00011$$

(3) 结果规格化。

商的双符号位为 11，尾数无溢出，不需右规；$M_s \oplus M_{MSB}=1 \oplus 0=1$，已是规格化数，不需左规。

(4) 舍入处理。

商的字长已与原始操作数 X、Y 相同，无保护位，不需舍入。

(5) 溢出判断。

本题在阶码可能发生溢出的(1)、(4)两步均未溢出，结果正确。

最终结果为

$$[Z]_{\text{阶移尾补}}=1,1110;\ 1.00011$$

$$Z=X\div Y=[2^{+1110}\times(-0.11101)]_2=\left[2^{14}\times\left(-\frac{29}{32}\right)\right]_{10}$$

5.5.7 浮点运算的实现

1. 浮点运算器的基本配置

分析浮点四则运算算法可以发现，浮点运算分阶码运算和尾数运算两部分进行，阶码只需要加减运算，尾数则需要加、减、乘、除四则运算。由此可知浮点运算器可由两个定点运算部件组成，一个是阶码部件，用来完成阶码加、减以及对阶时求阶差、尾数右移次数的控制，规格化时阶码的调整等操作。另一个是尾数部件，用来完成尾数的四则运算以及舍入、结果规格化判断和溢出判断等操作。一个基本的浮点运算器框图如图 5-51 所示。

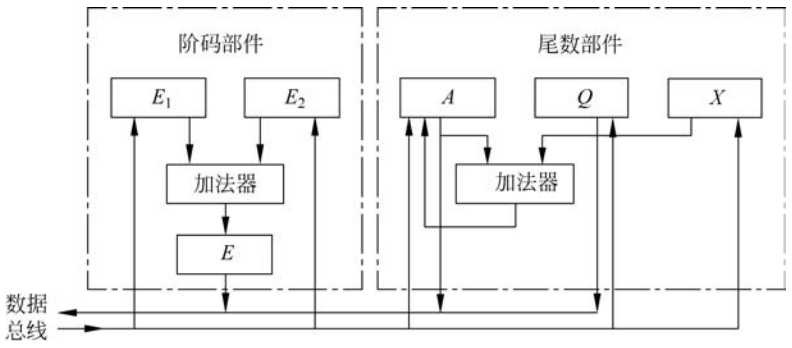


图 5-51　浮点运算器基本结构框图

图 5-51 示意出了由定点加减运算器配上阶差计数器 E 组成的阶码部件和由定点四则运算器实现的尾数部件，通过总线松散连接构成完整浮点运算部件的方式。虽然简单但基本上体现了浮点运算器的结构特点。

2. 具有高速乘除法器的浮点运算器

为了进一步提高乘除法运算速度，目前的浮点运算器中还经常设置阵列式的高速乘除法器，如图 5-52 所示。

图 5-52 示意的浮点运算器除了寄存器配置之外，阶码运算部分仍然由一个定点加减法器实现，而尾数运算部分则由一个定点加减法器和一个高速的定点乘除法器共同构成。这个浮点运算器的内部也采用了双总线连接方式，两条总线之间的传送则需要通过添加总线连接器来实现。

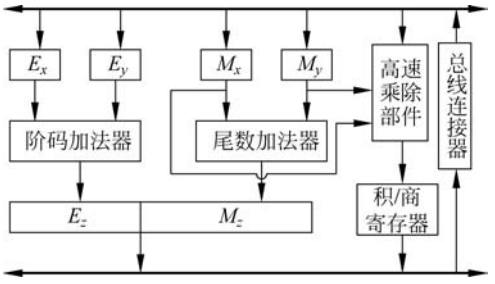


图 5-52　设置有高速乘除法器的浮点运算器框图

思考题与习题

1. 用十六进制写出大写字母“F”、小写字母“a”和星号“*”的 ASCII 码。当最高位用作偶校验位时,写出它们的 ASCII 机内码字节。

2. 汉字“大”和“小”的国标区位码分别为 2083 和 4801,要求:

(1) 分别写出这两个字对应的国标码;

(2) 若采用汉字两个字节的最高位均设为“1”的机内表示方案,分别写出这两个字的机内码形式。

3. 用向量表示法,在 32 位字长的存储器中,用 ASCII 码分别按左→右(大端方式)和右→左(小端方式)的顺序表示下列字符串:

(1) WHAT IS THIS?

(2) THIS IS A DISK.

4. 用以下形式表示十进制数 5862。

(1) 二进制数;(2) 8421 码;(3) 余 3 码;(4) 2421 码。

5. 用前分隔数字串表示法、后嵌入数字串表示法和压缩的十进制数串表示法表示下列十进制数,设存储器按字节编址。

+1980;　−76543;　+254;　−1992

6. 有两位 NBCD 码编码的十进制整数置于寄存器 A 中,可以通过一个加法器网络将其直接转换成二进制整数。试用半加器、全加器电路画出该加法器网络。

7. 设 X 为整数,$[X]_{补}=1,X_1X_2X_3X_4X_5$,若要求 $X<-16$,试问 $X_1\sim X_5$ 应取何值?

8. 已知数的补码表示,求数的原码与真值。

$[X_1]_{补}=0001\ 1010$　　$[X_2]_{补}=1001\ 1010$　　$[X_3]_{补}=1111\ 0001$

9. 讨论若 $[X]_{补}>[Y]_{补}$,是否有 $X>Y$?

10. 设 $[X]_{补}=a_0.a_1a_2a_3a_4a_5a_6$,其中 a_i 取 0 或 1,若要 $X>-0.5$,求 $a_0,a_1,a_2,\cdots,a_6$ 的取值。

11. 当十六进制数 9AH、80H 和 FFH 分别表示原码、补码、反码、移码和无符号数时,对应的十进制真值各为多少(设机器数采用一位符号位)?

12. 设机器数字长为 16 位,写出下列各种情况下它能表示的数的范围。机器数采用一位符号位,答案均用十进制 2 的幂形式表示。

(1) 无符号整数；

(2) 原码表示的定点小数；

(3) 补码表示的定点小数；

(4) 原码表示的定点整数；

(5) 补码表示的定点整数。

13. 设机器字长为8位(含1位符号位)，分整数和小数两种情况讨论真值 X 为何值时，$[X]_{补}=[X]_{原}$ 成立。

14. 设机器数字长为8位(含1位符号位)，对下列各机器数算术左移一位、两位，算术右移一位、两位，并讨论结果是否正确。

$$[X_1]_{原}=0.001\ 1010；\quad [X_2]_{原}=1.110\ 1000$$

$$[Y_1]_{补}=0.101\ 0100；\quad [Y_2]_{补}=1.110\ 1000$$

$$[Z_1]_{反}=1.010\ 1111；\quad [Z_2]_{反}=1.110\ 1000$$

15. 设有符号数 $[Y]_{原}=[Y]_{反}=[Y]_{补}=1,011\ 0010$，分别对这个8位字长的机器数进行算术左移1位、2位，算术右移1位、2位，逻辑左移1位、2位，逻辑右移1位、2位的操作，比较两种移位运算的区别，并分析结果的真值变化、误差及溢出情况。

16. 设 $X_1=0.011\ 1000\ 010$，$Y_1=-0.011\ 1000\ 010$；$X_2=0.011\ 1001\ 100$，$Y_2=-0.011\ 1001\ 100$。

分别用原码和补码表示，如果只要求8位字长，请采用截断法、恒置1法和0舍1入法对每一个操作数进行舍入，并对舍入结果进行比较。

17. 设机器数字长为8位(含1位符号位)，用补码加减运算规则计算下列各题，并指出是否溢出。

(1) $X=-17/32$，$Y=19/64$，求 $X-Y$；

(2) $X=-21/32$，$Y=-67/128$，求 $X+Y$；

(3) $X=97$，$Y=-54$，求 $X-Y$；

(4) $X=118$，$Y=-36$，求 $X+Y$。

18. 用原码一位乘法和补码一位乘比较法、两位乘比较法计算 $X\times Y$。

(1) $X=0.110\ 111$，$Y=-0.101\ 110$

(2) $X=-0.010\ 111$，$Y=-0.010\ 101$

19. 用原码加减交替除法和补码加减交替除法计算 $X\div Y$。

(1) $X=-0.10101$，$Y=0.11011$

(2) $X=13/32$，$Y=-27/32$

20. 设机器字长为16位(含1位符号位)，若一次移位需1μs，一次加法需1μs，试问原码一位乘法、补码一位乘法、原码加减交替除法和补码加减交替除法最多各需多少时间？

21. 分别用8421码加法和余3码加法求 $57+48=?$ $316+258$，要求列出竖式计算过程。

22. 用预加6方案设计一位8421码加法单元，并以设计好的加法单元为模块。进一步设计一个4位的8421码加法器。

23. (1) 设计一个一位的余3码加法器，并分析其修正规律；

(2) 用8位并行二进制加法器实现2位余3码加法，试提出你的方案。

24. 有下列16位字长的逻辑数(八进制表示):

$$A=000\ 377;\ B=123\ 456;\ C=054\ 321$$

试计算:$X_1=(B\oplus C)\cdot A$;$X_2=/(/B\cdot /C)+A$;$X_3=(A\oplus B)+/(A\cdot C)$

25. 设4位二进制加法器进位信号为$C_4C_3C_2C_1$,最低位进位输入为C_0;输入数据为$A_3A_2A_1A_0$和$B_3B_2B_1B_0$;进位生成函数为$g_3g_2g_1g_0$,进位传递函数为$p_3p_2p_1p_0$;请分别按下述两种方式写出$C_4C_3C_2C_1$的逻辑表达式:

(1) 串行进位方式;(2) 并行进位方式。

26. 设机器字长为32位,用与非门和与或非门设计一个并行加法器(假设与非门的延迟时间为10ns,与或非门的延迟时间为15ns),要求完成32位加法时间不得超过0.2μs。画出进位线路逻辑框图及加法器逻辑框图。

27. 设机器字长为16位,分别按4、4、4、4和3、5、3、5分组后:

(1) 画出两种分组方案的成组先行-级联进位线路框图,并比较哪种方案运算速度快。

(2) 画出两种分组方案的二级先行进位线路框图,并对这两种方案的速度进行比较。

(3) 用74181和74182画出成组先行-级联进位和二级先行进位的并行进位线路框图。

28. 假设某机字长为32位,加法器的输入数据为a_i、b_i,第1级进位函数为g_i、p_i,第2级进位函数为G_i、P_i,第3级进位函数为G_i^*、P_i^*,进位信号为C_i,其中$i=0,1,2,\cdots$,从低位到高位递增。

(1) 请写出g_i、p_i的原理性逻辑表达式;

(2) 假设第1级为4位分组,加法器采用二级先行-级联进位方案,请写出C_4、G_0、P_0的原理性逻辑表达式。

(3) 请用74181和74182为该机设计一个并行加法器。

29. 浮点数的格式为:阶码6位(含1位阶符),尾数10位(含1位数符)。按下列要求分别写出正数和负数的表示范围,答案均用2的幂形式的十进制真值表示。

(1) 阶原尾原非规格化数;

(2) 阶移尾补规格化数;

(3) 按照(2)的格式,写出-27/1024和7.375的浮点机器数。

30. (1) 将十进制数138.75转换成32位的IEEE 754短浮点数格式,并用十六进制缩写表示。

(2) 将IEEE 754短浮点数C1B7 0000H转换成对应的十进制真值。

31. 设浮点数字长为32位,欲表示-6万~6万的十进制数,在保证数的最大精度条件下,除阶符、数符各取一位外,阶码和尾数各取几位?按这样分配,该浮点数溢出的条件是什么?

32. 对于尾数为40位的浮点数(不包括符号位在内),若采用不同的机器数表示,试问当尾数左规或右规时,最多移位次数各为多少?

33. 按机器补码浮点运算步骤计算$[X\pm Y]_{补}$。

(1) $X=2^{-011}\times 0.101\ 100$,$Y=2^{-010}\times(-0.011\ 100)$

(2) $X=2^{101}\times(-0.100\ 101)$,$Y=2^{100}\times(-0.001\ 111)$

34. 设浮点数阶码取3位,尾数取6位(均不包括符号位),要求阶码用移码运算,尾数

用原码运算，计算 $X\times Y$ 和 $X\div Y$，结果保留 1 倍字长。

(1) $X=2^{100}\times 0.100\ 111$，$Y=2^{011}\times(-0.101\ 011)$

(2) $X=2^{101}\times(-0.101\ 101)$，$Y=2^{001}\times(-0.111\ 100)$

35. 设数的阶码 3 位，尾数 6 位，均不含符号位；阶码用移码表示，尾数用补码表示；阶的基为 2。用浮点算法计算 $X+Y$、$X-Y$、$X\times Y$、$X\div Y$，结果要求为规格化数。已知：$X=2^{-2}\times 11/16$；$Y=2^{3}\times(-15/16)$。

36. 假定在一个 8 位字长的计算机中，定点整数用单字长表示，其中带符号整数用补码表示(1 位符号位)；浮点数用双字长表示，阶码为 8 位移码(包括 1 位符号位)，尾数用 8 位原码(包括 1 位符号位)。运行如下类 C 程序段：

```
int   x1 = - 124;
int   x2 = 116;
unsigned int y1 = x1;
float   f1 = x1;
int   z1 = x1  +  x2;
int   z2 = x1  -  x2;
```

(1) 执行上述程序段后，所有变量的值在该计算机内的数据表示形式各是多少？所有变量的值对应的十进制形式各是多少？

(2) 在该计算机中，无符号整数、有符号整数和规格化浮点数的表示范围各是什么？(要求：用十进制 2 的幂形式表示)

(3) 执行上述程序段后，哪些运算语句的执行结果发生了溢出？