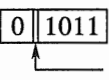
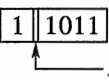
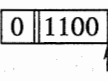
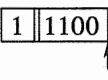


+0.1011 在机器中表示为 $\boxed{0 \parallel 1011}$

 小数点位置

-0.1011 在机器中表示为 $\boxed{1 \parallel 1011}$

 小数点位置

又如,有符号数(整数):

+1100 在机器中表示为 $\boxed{0 \parallel 1100}$

 小数点位置

-1100 在机器中表示为 $\boxed{1 \parallel 1100}$

 小数点位置

把符号“数字化”的数称为机器数,而把带“+”或“-”符号的数称为真值。一旦符号数字化后,符号和数值就形成了一种新的编码。在运算过程中,符号位能否和数值部分一起参加运算?如果参加运算,符号位又需作哪些处理?这些问题都与符号位和数值位所构成的编码有关,这些编码就是原码、补码、反码和移码。

2. 原码表示法

原码是机器数中最简单的一种表示形式,符号位为0表示正数,符号位为1表示负数,数值位即真值的绝对值,故原码表示又称为带符号的绝对值表示。上面列举的4个真值所对应的机器数即为原码。为了书写方便以及区别整数和小数,约定整数的符号位与数值位之间用逗号隔开;小数的符号位与数值位之间用小数点隔开。例如,上面4个数的原码分别是0.1011、1.1011、0,1100和1,1100。由此可得原码的定义。

整数原码的定义为

$$[x]_{\text{原}} = \begin{cases} 0, x & 2^n > x \geq 0 \\ 2^n - x & 0 \geq x > -2^n \end{cases}$$

式中, x 为真值, n 为整数的位数。

例如:

当 $x = +1110$ 时, $[x]_{\text{原}} = 0,1110$

当 $x = -1110$ 时, $[x]_{\text{原}} = 2^4 - (-1110) = 1,1110$

↑
用逗号将符号位和数值部分隔开

小数原码的定义为

$$[x]_{\text{原}} = \begin{cases} x & 1 > x \geq 0 \\ 1 - x & 0 \geq x > -1 \end{cases}$$

式中, x 为真值。

例如:

当 $x = 0.1101$ 时, $[x]_{\text{原}} = 0.1101$

当 $x = -0.1101$ 时, $[x]_{\text{原}} = 1 - (-0.1101) = 1.1101$

根据定义,已知真值可求原码,反之已知原码也可求真值。例如:

当 $[x]_{\text{原}} = 1.0011$ 时,由定义得

$$x = 1 - [x]_{\text{原}} = 1 - 1.0011 = -0.0011$$

当 $[x]_{\text{原}} = 1,1100$ 时,由定义得

$$x = 2^n - [x]_{\text{原}} = 2^4 - 1,1100 = 10000 - 11100 = -1100$$

当 $[x]_{\text{原}} = 0.1101$ 时, $x = 0.1101$

当 $x = 0$ 时

$$[+0.0000]_{\text{原}} = 0.0000$$

$$[-0.0000]_{\text{原}} = 1 - (0.0000) = 1.0000$$

可见 $[+0]_{\text{原}}$ 不等于 $[-0]_{\text{原}}$,即原码中的“零”有两种表示形式。

原码表示简单明了,并易于和真值转换。但用原码进行加减运算时,却带来了许多麻烦。例如,当两个操作数符号不同且要做加法运算时,先要判断两数绝对值大小,然后将绝对值大的数减去绝对值小的数,结果的符号以绝对值大的数为准。运算步骤既复杂又费时,而且本来是加法运算却要用减法器实现。那么能否在计算机中只设加法器,只做加法操作呢?如果能找到一个与负数等价的正数来代替该负数,就可把减法操作用加法代替。而机器数采用补码时,就能满足此要求。

3. 补码表示法

(1) 补数的概念

在日常生活中,常会遇到“补数”的概念。例如,时钟指示 6 点,欲使它指示 3 点,既可按顺时针方向将分针转 9 圈,又可按逆时针方向将分针转 3 圈,结果是一致的。假设顺时针方向转为正,逆时针方向转为负,则有

$$\begin{array}{r} 6 \\ - 3 \\ \hline 3 \end{array} \quad \begin{array}{r} 6 \\ + 9 \\ \hline 15 \end{array}$$

由于时钟的时针转一圈能指示 12 个小时,这“12”在时钟里是不被显示而自动丢失的,即 $15 - 12 = 3$,故 15 点和 3 点均显示 3 点。这样 -3 和 +9 对时钟而言其作用是一致的。在数学上称 12 为模,写作 mod 12,而称 +9 是 -3 以 12 为模的补数,记作

$$-3 \equiv +9 \quad (\text{mod } 12)$$

或者说,对模 12 而言, -3 和 +9 是互为补数的。同理有

$$-4 \equiv +8 \quad (\text{mod } 12)$$

$$-5 \equiv +7 \quad (\text{mod } 12)$$

即对模 12 而言, +8 和 +7 分别是 -4 和 -5 的补数。可见,只要确定了“模”,就可找到一个与负数等价的正数(该正数即为负数的补数)来代替此负数,这样就可把减法运算用加法实现。例如:

设 $A=9, B=5$, 求 $A-B \pmod{12}$ 。

解:

$$A-B=9-5=4 \quad (\text{作减法})$$

对模 12 而言, -5 可以用其补数 $+7$ 代替, 即

$$-5 \equiv +7 \quad (\text{mod } 12)$$

所以 $A-B=9+7=16$ (作加法)

对模 12 而言, 12 会自动丢失, 所以 16 等价于 4, 即 $4 \equiv 16 \pmod{12}$ 。

进一步分析发现, 3 点、15 点、27 点……在时钟上看见的都是 3 点, 即

$$3 \equiv 15 \equiv 27 \quad (\text{mod } 12)$$

也即 $3 \equiv 3+12 \equiv 3+24 \equiv 3 \pmod{12}$

这说明正数相对于“模”的补数就是正数本身。

上述补数的概念可以用到任意“模”上, 如

$$-3 \equiv +7 \quad (\text{mod } 10)$$

$$+7 \equiv +7 \quad (\text{mod } 10)$$

$$-3 \equiv +97 \quad (\text{mod } 10^2)$$

$$+97 \equiv +97 \quad (\text{mod } 10^2)$$

$$-1011 \equiv +0101 \quad (\text{mod } 2^4)$$

$$+0101 \equiv +0101 \quad (\text{mod } 2^4)$$

$$-0.1001 \equiv +1.0111 \quad (\text{mod } 2)$$

$$+0.1001 \equiv +0.1001 \quad (\text{mod } 2)$$

由此可得如下结论。

- 一个负数可用它的正补数来代替, 而这个正补数可以用模加上负数本身求得。
- 一个正数和一个负数互为补数时, 它们绝对值之和即为模数。
- 正数的补数即该正数本身。

将补数的概念用到计算机中, 便出现了补码这种机器数。

(2) 补码的定义

整数补码的定义为

$$[x]_{\text{补}} = \begin{cases} 0, x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 > x \geq -2^n \end{cases} \pmod{2^{n+1}}$$

式中, x 为真值, n 为整数的位数。

例如:

当 $x = +1010$ 时,

$$[x]_{\text{补}} = 0, 1010$$

↑

用逗号将符号位和数值部分隔开

当 $x = -1101$ 时,

$$[x]_{\text{补}} = 2^{n+1} + x = 100000 - 1101 = 1,0011$$

↑

用逗号将符号位和数值部分隔开

小数补码的定义为

$$[x]_{\text{补}} = \begin{cases} x & 1 > x \geq 0 \\ 2+x & 0 > x \geq -1 \end{cases} \pmod{2}$$

式中, x 为真值。

例如:

当 $x = 0.1001$ 时,

$$[x]_{\text{补}} = 0.1001$$

当 $x = -0.0110$ 时,

$$[x]_{\text{补}} = 2+x = 10.0000 - 0.0110 = 1.1010$$

当 $x = 0$ 时,

$$[+0.0000]_{\text{补}} = 0.0000$$

$$[-0.0000]_{\text{补}} = 2 + (-0.0000) = 10.0000 - 0.0000 = 0.0000$$

显然 $[+0]_{\text{补}} = [-0]_{\text{补}} = 0.0000$, 即补码中的“零”只有一种表示形式。

对于小数, 若 $x = -1$, 则根据小数补码定义, 有 $[x]_{\text{补}} = 2+x = 10.0000 - 1.0000 = 1.0000$ 。可见, -1 本不属于小数范围, 但却有 $[-1]_{\text{补}}$ 存在 (其实在小数补码定义中已指明), 这是由于补码中的零只有一种表示形式, 故它比原码能多表示一个“ -1 ”。此外, 根据补码定义, 已知补码还可以求真值, 例如:

若 $[x]_{\text{补}} = 1.0101$

则

$$x = [x]_{\text{补}} - 2 = 1.0101 - 10.0000 = -0.1011$$

若 $[x]_{\text{补}} = 1,1110$

则

$$x = [x]_{\text{补}} - 2^{4+1} = 1,1110 - 100000 = -0010$$

若 $[x]_{\text{补}} = 0.1101$

则

$$x = [x]_{\text{补}} = 0.1101$$

同理, 当模数为 4 时, 形成了双符号位的补码。如 $x = -0.1001$, 对 $(\text{mod } 2^2)$ 而言

$$[x]_{\text{补}} = 2^2 + x = 100.0000 - 0.1001 = 11.0111$$

这种双符号位的补码又称为变形补码, 它在阶码运算和溢出判断中有其特殊作用, 后面有关章节中将详细介绍。

由上讨论可知, 引入补码的概念是为了消除减法运算, 但是根据补码的定义, 在形成补码的过程中又出现了减法。例如:

$$x = -1011$$

$$[x]_{\text{补}} = 2^{4+1} + x = 100000 - 1011 = 1,0101 \quad (6.1)$$

若把模 2^{4+1} 改写成 $2^5 = 100000 = 11111 + 00001$ 时, 则式(6.1)可写成

$$[x]_{\text{补}} = 2^5 + x = 11111 + 00001 + x \quad (6.2)$$

又因 x 是负数, 若 x 用 $-x_1x_2x_3x_4$ 表示, 其中 $x_i (i=1, 2, 3, 4)$ 不为 0 则为 1, 于是式(6.2)可写成

$$\begin{aligned} [x]_{\text{补}} &= 2^5 + x = 11111 + 00001 - x_1x_2x_3x_4 \\ &= 1 \overline{x_1} \overline{x_2} \overline{x_3} \overline{x_4} + 00001 \end{aligned} \quad (6.3)$$

因为任一位“1”减去 x_i 即为 $\overline{x_i}$, 所以式(6.3)成立。

由于负数 $-x_1x_2x_3x_4$ 的原码为 1, $x_1x_2x_3x_4$, 因此对这个负数求补, 可以看作对它的原码除符号位外, 每位求反, 末位加 1, 简称“求反加 1”。这样, 由真值通过原码求补码就可避免减法运算。同理, 对于小数也有同样的结论, 读者可以自行证明。

“由原码除符号位外, 每位求反, 末位加 1 求补码”这一规则同样适用于由 $[x]_{\text{补}}$ 求 $[x]_{\text{原}}$ 。而对于一个负数, 若对其原码除符号位外, 每位求反 (简称“每位求反”), 或是对其补码减去末位的 1, 即得机器数的反码。

4. 反码表示法

反码通常用来作为由原码求补码或者由补码求原码的中间过渡。反码的定义如下:

整数反码的定义为

$$[x]_{\text{反}} = \begin{cases} 0, x & 2^n > x \geq 0 \\ (2^{n+1} - 1) + x & 0 \geq x > -2^n \end{cases} \pmod{(2^{n+1} - 1)}$$

式中, x 为真值, n 为整数的位数。

例如:

当 $x = +1101$ 时,

$$[x]_{\text{反}} = 0, 1101$$

↑

用逗号将符号位和数值部分隔开

当 $x = -1101$ 时,

$$[x]_{\text{反}} = (2^{4+1} - 1) + x = 11111 - 1101 = 1, 0010$$

↑

用逗号将符号位和数值部分隔开

小数反码的定义为

$$[x]_{\text{反}} = \begin{cases} x & 1 > x \geq 0 \\ (2 - 2^{-n}) + x & 0 \geq x > -1 \end{cases} \pmod{(2 - 2^{-n})}$$

式中, x 为真值, n 为小数的位数。

例如:

当 $x = +0.0110$ 时,

$[x]_{\text{反}} = 0.0110$

当 $x = -0.0110$ 时,

$[x]_{\text{反}} = (2 - 2^{-4}) + x = 1.1111 - 0.0110 = 1.1001$

当 $x = 0$ 时,

$[+0.0000]_{\text{反}} = 0.0000$

$[-0.0000]_{\text{反}} = (10.0000 - 0.0001) - 0.0000 = 1.1111$

可见 $[+0]_{\text{反}}$ 不等于 $[-0]_{\text{反}}$, 即反码中的“零”也有两种表示形式。

实际上,反码也可看作是 $\text{mod}(2 - 2^{-n})$ (对于小数)或 $\text{mod}(2^{n+1} - 1)$ (对于整数)的补码。与补码相比,仅在末位差 1,因此有些书上称小数的补码为 2 的补码,而称小数的反码为 1 的补码。

综上所述,三种机器数的特点可归纳如下:

- 三种机器数的最高位均为符号位。符号位和数值部分之间可用“.”(对于小数)或“,”(对于整数)隔开。
- 当真值为正时,原码、补码和反码的表示形式均相同,即符号位用“0”表示,数值部分与真值相同。
- 当真值为负时,原码、补码和反码的表示形式不同,但其符号位都用“1”表示,而数值部分有这样的关系,即补码是原码的“求反加 1”,反码是原码的“每位求反”。

下面通过实例来进一步理解和掌握三种机器数的表示。

例 6.1 设机器数字长为 8 位(其中 1 位为符号位),对于整数,当其分别代表无符号数、原码、补码和反码时,对应的真值范围各为多少?

解:表 6.1 列出了 8 位寄存器中所有二进制代码组合与无符号数、原码、补码和反码所代表的真值的对应关系。

表 6.1 例 6.1 对应的真值范围

二进制代码	无符号数 对应的真值	原码对应 的真值	补码对应 的真值	反码对应 的真值
0 0 0 0 0 0 0 0	0	+0	± 0	+0
0 0 0 0 0 0 0 1	1	+1	+1	+1
0 0 0 0 0 0 1 0	2	+2	+2	+2
⋮	⋮	⋮	⋮	⋮
0 1 1 1 1 1 1 0	126	+126	+126	+126
0 1 1 1 1 1 1 1	127	+127	+127	+127
1 0 0 0 0 0 0 0	128	-0	-128	-127
1 0 0 0 0 0 0 1	129	-1	-127	-126
1 0 0 0 0 0 1 0	130	-2	-126	-125
⋮	⋮	⋮	⋮	⋮
1 1 1 1 1 1 0 1	253	-125	-3	-2
1 1 1 1 1 1 1 0	254	-126	-2	-1
1 1 1 1 1 1 1 1	255	-127	-1	-0

由此可得出一个结论:由于“零”在补码中只有一种表示形式,故补码比原码和反码可以多表示一个负数。

例 6.2 已知 $[y]_{\text{补}}$,求 $[-y]_{\text{补}}$ 。

解:设 $[y]_{\text{补}} = y_0 y_1 y_2 \cdots y_n$

第一种情况 $[y]_{\text{补}} = 0.y_1 y_2 \cdots y_n$ (6.4)

所以 $y = 0.y_1 y_2 \cdots y_n$

故 $-y = -0.y_1 y_2 \cdots y_n$

则 $[-y]_{\text{补}} = 1.\bar{y}_1 \bar{y}_2 \cdots \bar{y}_n + 2^{-n}$ (6.5)

比较式(6.4)和式(6.5),发现由 $[y]_{\text{补}}$ 连同符号位在内每位取反,末位加1,即可得 $[-y]_{\text{补}}$ 。

第二种情况 $[y]_{\text{补}} = 1.y_1 y_2 \cdots y_n$ (6.6)

所以 $[y]_{\text{原}} = 1.\bar{y}_1 \bar{y}_2 \cdots \bar{y}_n + 2^{-n}$

得 $y = -(0.\bar{y}_1 \bar{y}_2 \cdots \bar{y}_n + 2^{-n})$

故 $-y = 0.\bar{y}_1 \bar{y}_2 \cdots \bar{y}_n + 2^{-n}$

则 $[-y]_{\text{补}} = 0.\bar{y}_1 \bar{y}_2 \cdots \bar{y}_n + 2^{-n}$ (6.7)

比较式(6.6)、式(6.7),发现由 $[y]_{\text{补}}$ 连同符号位在内每位取反,末位加1,即可得 $[-y]_{\text{补}}$ 。

可见,不论真值是正(第一种情况)或负(第二种情况),由 $[y]_{\text{补}}$ 求 $[-y]_{\text{补}}$ 都是采用“连同符号位在内,每位取反,末位加1”的规则。这一结论在补码减法运算时将经常用到(详见6.3节有关内容)。

有符号数在计算机中除了用原码、补码和反码表示外,在一些通用计算机中还用另一种机器数——移码表示,由于一些突出的优点,目前它已被广泛采用。

5. 移码表示法

当真值用补码表示时,由于符号位和数值部分一起编码,与习惯上的表示法不同,因此人们很难从补码的形式上直接判断其真值的大小,例如:

十进制数 $x=21$,对应的二进制数为 $+10101$,则 $[x]_{\text{补}}=0,10101$

十进制数 $x=-21$,对应的二进制数为 -10101 ,则 $[x]_{\text{补}}=1,01011$

十进制数 $x=31$,对应的二进制数为 $+11111$,则 $[x]_{\text{补}}=0,11111$

十进制数 $x=-31$,对应的二进制数为 -11111 ,则 $[x]_{\text{补}}=1,00001$

上述补码表示中的“,”在计算机内部是不存在的,因此,从代码形式看,符号位也是一位二进制数。按这6位二进制代码比较大小的话,会得出 $101011 > 010101$, $100001 > 011111$,其实恰恰相反。

如果对每个真值加上一个 2^n (n 为整数的位数),情况就发生了变化。例如:

$x=10101$ 加上 2^5 可得 $10101+100000=110101$

$x=-10101$ 加上 2^5 可得 $-10101+100000=001011$

$x=11111$ 加上 2^5 可得 $11111+100000=111111$

$x = -11111$ 加上 2^5 可得 $-11111 + 100000 = 000001$

比较它们的结果可见, $110101 > 001011$, $111111 > 000001$ 。这样一来, 从 6 位代码本身就可看出真值的实际大小。

由此可得移码的定义

$$[x]_{\text{移}} = 2^n + x \quad (2^n > x \geq -2^n)$$

式中, x 为真值, n 为整数的位数。

其实移码就是在真值上加一个常数 2^n 。在数轴上移码所表示的范围恰好对应于真值在数轴上的范围向轴的正方向移动 2^n 个单元, 如图 6.1 所示, 由此而得移码之称。

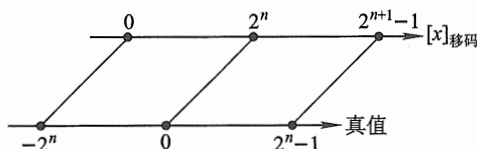


图 6.1 移码在数轴上的表示

例如: $x = 10100$

$$[x]_{\text{移}} = 2^5 + 10100 = 1, 10100$$

↑

用逗号将符号位和数值部分隔开

$x = -10100$

$$[x]_{\text{移}} = 2^5 - 10100 = 0, 01100$$

↑

用逗号将符号位和数值部分隔开

当 $x = 0$ 时

$$[+0]_{\text{移}} = 2^5 + 0 = 1, 00000$$

$$[-0]_{\text{移}} = 2^5 - 0 = 1, 00000$$

可见 $[+0]_{\text{移}}$ 等于 $[-0]_{\text{移}}$, 即移码表示中零也是唯一的。

此外, 由移码的定义可见, 当 $n = 5$ 时, 其最小的真值为 $x = -2^5 = -100000$, 则 $[-100000]_{\text{移}} = 2^5 + x = 100000 - 100000 = 0, 00000$, 即最小真值的移码为全 0, 这符合人们的习惯。利用移码的这一特点, 当浮点数的阶码用移码表示时, 就能很方便地判断阶码的大小 (详见 6.2.4 节)。

进一步观察发现, 同一个真值的移码和补码仅差一个符号位, 若将补码的符号位由“0”改为“1”, 或从“1”改为“0”, 即可得该真值的移码。表 6.2 列出了真值、补码和移码的对应关系。

6.2.2 浮点表示

实际上计算机中处理的数不一定是纯小数或纯整数(如圆周率 3.141 6),而且有些数据的数值范围相差很大(如电子的质量 $9 \times 10^{-28} \text{g}$, 太阳的质量 $2 \times 10^{33} \text{g}$),它们都不能直接用定点小数或定点整数表示,但均可用浮点数表示。浮点数即小数点的位置可以浮动的数,如

$$\begin{aligned} 352.47 &= 3.5247 \times 10^2 \\ &= 3524.7 \times 10^{-1} \\ &= 0.35247 \times 10^3 \end{aligned}$$

显然,这里小数点的位置是变化的,但因为分别乘上了不同的 10 的方幂,故值不变。

通常,浮点数被表示成

$$N = S \times r^j$$

式中, S 为尾数(可正可负), j 为阶码(可正可负), r 是基数(或基值)。在计算机中,基数可取 2、4、8 或 16 等。

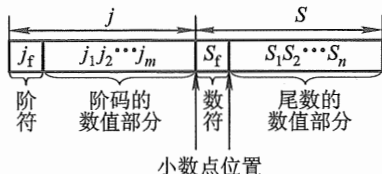
以基数 $r=2$ 为例,数 N 可写成下列不同的形式:

$$\begin{aligned} N &= 11.0101 \\ &= 0.110101 \times 2^{10} \\ &= 1.10101 \times 2^1 \\ &= 1101.01 \times 2^{-10} \\ &= 0.00110101 \times 2^{100} \\ &\vdots \end{aligned}$$

为了提高数据精度以及便于浮点数的比较,在计算机中规定浮点数的尾数用纯小数形式,故上例中 0.110101×2^{10} 和 $0.00110101 \times 2^{100}$ 形式是可以采用的。此外,将尾数最高位为 1 的浮点数称为规格化数,即 $N = 0.110101 \times 2^{10}$ 为浮点数的规格化形式。浮点数表示成规格化形式后,其精度最高。

1. 浮点数的表示形式

浮点数在机器中的形式如下所示。采用这种数据格式的机器称为浮点机。



浮点数由阶码 j 和尾数 S 两部分组成。阶码是整数,阶符和阶码的位数 m 合起来反映浮点数的表示范围及小数点的实际位置;尾数是小数,其位数 n 反映了浮点数的精度;尾数的符号 S_f

代表浮点数的正负。

2. 浮点数的表示范围

以通式 $N = S \times r^j$ 为例, 设浮点数阶码的数值位取 m 位, 尾数的数值位取 n 位, 当浮点数为非规格化数时, 它在数轴上的表示范围如图 6.2 所示。

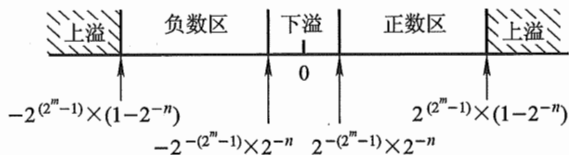


图 6.2 浮点数在数轴上的表示范围

由图中可见, 其最大正数为 $2^{(2^m-1)} \times (1-2^{-n})$; 最小正数为 $2^{-(2^m-1)} \times 2^{-n}$; 最大负数为 $-2^{-(2^m-1)} \times 2^{-n}$; 最小负数为 $-2^{(2^m-1)} \times (1-2^{-n})$ 。当浮点数阶码大于最大阶码时, 称为上溢, 此时机器停止运算, 进行中断溢出处理; 当浮点数阶码小于最小阶码时, 称为下溢, 此时溢出的数绝对值很小, 通常将尾数各位强置为零, 按机器零处理, 此时机器可以继续运行。

一旦浮点数的位数确定后, 合理分配阶码和尾数的位数, 直接影响浮点数的表示范围和精度。通常对于短实数(总位数为 32 位), 阶码取 8 位(含阶符 1 位), 尾数取 24 位(含数符 1 位); 对于长实数(总位数为 64 位), 阶码取 11 位(含阶符 1 位), 尾数取 53 位(含数符 1 位); 对于临时实数(总位数为 80 位), 阶码取 15 位(含阶符 1 位), 尾数取 65 位(含数符 1 位)。

3. 浮点数的规格化

为了提高浮点数的精度, 其尾数必须为规格化数。如果不是规格化数, 就要通过修改阶码并同时左右移尾数的办法, 使其变成规格化数。将非规格化数转换成规格化数的过程称为规格化。对于基数不同的浮点数, 因其规格化数的形式不同, 规格化过程也不同。

当基数为 2 时, 尾数最高位为 1 的数为规格化数。规格化时, 尾数左移一位, 阶码减 1(这种规格化称为向左规格化, 简称左规); 尾数右移一位, 阶码加 1(这种规格化称为向右规格化, 简称右规)。图 6.2 所示的浮点数规格化后, 其最大正数为 $2^{(2^m-1)} \times (1-2^{-n})$, 最小正数为 $2^{-(2^m-1)} \times 2^{-1}$; 最大负数为 $-2^{(2^m-1)} \times 2^{-1}$, 最小负数为 $-2^{(2^m-1)} \times (1-2^{-n})$ 。

当基数为 4 时, 尾数的最高两位不全为零的数为规格化数。规格化时, 尾数左移两位, 阶码减 1; 尾数右移两位, 阶码加 1。

当基数为 8 时, 尾数的最高三位不全为零的数为规格化数。规格化时, 尾数左移三位, 阶码减 1; 尾数右移三位, 阶码加 1。

同理类推, 不难得到基数为 16 或 2^n 时的规格化过程。

浮点机中一旦基数确定后就不再变了, 而且基数是隐含的, 故不同基数的浮点数表示形式完全相同。但基数不同, 对数的表示范围和精度等都有影响。一般来说, 基数 r 越大, 可表示的浮点数范围越大, 而且所表示的数的个数越多。但 r 越大, 浮点数的精度反而下降。如 $r=16$ 的浮

点数,因其规格化数的尾数最高三位可能出现零,故与其尾数位数相同的 $r=2$ 的浮点数相比,后者可能比前者多三位精度。

6.2.3 定点数和浮点数的比较

定点数和浮点数可从如下几个方面进行比较。

① 当浮点机和定点机中数的位数相同时,浮点数的表示范围比定点数的大得多。

② 当浮点数为规格化数时,其相对精度远比定点数高。

③ 浮点数运算要分阶码部分和尾数部分,而且运算结果都要求规格化,故浮点运算步骤比定点运算步骤多,运算速度比定点运算的低,运算线路比定点运算的复杂。

④ 在溢出的判断方法上,浮点数是对规格化数的阶码进行判断,而定点数是对数值本身进行判断。例如,小数定点机中的数,其绝对值必须小于1,否则“溢出”,此时要求机器停止运算,进行处理。为了防止溢出,上机前必须选择比例因子,这个工作比较麻烦,给编程带来不便。而浮点数的表示范围远比定点数大,仅当“上溢”时机器才停止运算,故一般不必考虑比例因子的选择。

总之,浮点数在数的表示范围、数的精度、溢出处理和程序编程方面(不取比例因子)均优于定点数。但在运算规则、运算速度及硬件成本方面又不如定点数。因此,究竟选用定点数还是浮点数,应根据具体应用综合考虑。一般来说,通用的大型计算机大多采用浮点数,或同时采用定、浮点数;小型、微型及某些专用机、控制机则大多采用定点数。当需要做浮点运算时,可通过软件实现,也可外加浮点扩展硬件(如协处理器)来实现。

6.2.4 举例

例 6.3 设浮点数字长16位,其中阶码5位(含1位阶符),尾数11位(含1位数符),将十进制数 $+\frac{13}{128}$ 写成二进制定点数和浮点数,并分别写出它在定点机和浮点机中的机器数形式。

解:令 $x = +\frac{13}{128}$

其二进制形式 $x = 0.0001101000$

定点数表示 $x = 0.0001101000$

浮点数规格化表示 $x = 0.1101000000 \times 2^{-11}$

定点机中 $[x]_{\text{原}} = [x]_{\text{补}} = [x]_{\text{反}} = 0.0001101000$

浮点机中

$[x]_{\text{原}}:$

1	0011	0	1101000000
---	------	---	------------

 或写成1,0011;0.1101000000

$[x]_{\text{补}}:$

1	1101	0	1101000000
---	------	---	------------

 或写成1,1101;0.1101000000

$[x]_{\text{反}}:$

1	1100	0	1101000000
---	------	---	------------

 或写成1,1100;0.1101000000

例 6.4 将十进制数-54 表示成二进制定点数和浮点数,并写出它在定点机和浮点机中的机器数形式(其他要求同上例)。

解:令 $x = -54$

其二进制形式

$$x = -110110$$

定点数表示

$$x = -0000110110$$

浮点数规格化表示

$$x = -(0.1101100000) \times 2^{110}$$

定点机中

$$[x]_{\text{原}} = 1,0000110110$$

$$[x]_{\text{补}} = 1,1111001010$$

$$[x]_{\text{反}} = 1,1111001001$$

浮点机中

$$[x]_{\text{原}} = 0,0110; 1.1101100000$$

$$[x]_{\text{补}} = 0,0110; 1.0010100000$$

$$[x]_{\text{反}} = 0,0110; 1.0010011111$$

例 6.5 写出对应图 6.2 所示的浮点数的补码形式。设图中 $n = 10, m = 4$, 阶符、数符各取 1 位。

解:

真值

补码

最大正数 $2^{15} \times (1 - 2^{-10})$ 0,1111; 0.1111111111

最小正数 $2^{-15} \times 2^{-10}$ 1,0001; 0.0000000001

最大负数 $-2^{-15} \times 2^{-10}$ 1,0001; 1.1111111111

最小负数 $-2^{15} \times (1 - 2^{-10})$ 0,1111; 1.0000000001

计算机中浮点数的阶码和尾数可以采用同一种机器数表示,也可采用不同的机器数表示。

例 6.6 设浮点数字长为 16 位,其中阶码为 5 位(含 1 位阶符),尾数为 11 位(含 1 位数符),写出 $-\frac{53}{512}$ 对应的浮点规格化数的原码、补码、反码和阶码用移码,尾数用补码的形式。

解:设 $x = -\frac{53}{512} = -0.000110101 = 2^{-11} \times (-0.1101010000)$

$$[x]_{\text{原}} = 1,0011; 1.1101010000$$

$$[x]_{\text{补}} = 1,1101; 1.0010110000$$

$$[x]_{\text{反}} = 1,1100; 1.0010101111$$

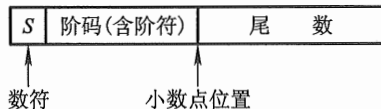
$$[x]_{\text{阶移,尾补}} = 0,1101; 1.0010110000$$

值得注意的是,当一个浮点数尾数为 0 时,不论其阶码为何值;或阶码等于或小于它所能表示的最小数时,不管其尾数为何值,机器都把该浮点数作为零看待,并称之为“机器零”。如果浮点数的阶码用移码表示,尾数用补码表示,则当阶码为它所能表示的最小数 2^{-m} (式中 m 为阶码的位数)且尾数为 0 时,其阶码(移码)全为 0,尾数(补码)也全为 0,这样的机器零为 000...0000,

全零表示有利于简化机器中判“0”电路。

6.2.5 IEEE 754 标准

现代计算机中,浮点数一般采用 IEEE 制定的国际标准,这种标准形式如下:



按 IEEE 标准,常用的浮点数有三种:

	符号位 S	阶码	尾数	总位数
短实数	1	8	23	32
长实数	1	11	52	64
临时实数	1	15	64	80

其中, S 为数符,它表示浮点数的正负,但与其有效位(尾数)是分开的。阶码用移码表示,阶码的真值都被加上一个常数(偏移量),如短实数、长实数和临时实数的偏移量用十六进制数表示分别为 7FH、3FFH 和 3FFFH(见附录 6A.1)。尾数部分通常都是规格化表示,即非“0”的有效位最高位总是“1”,但在 IEEE 标准中,有效位呈如下形式。

$$1 \blacktriangle \text{ffff} \cdots \text{fff}$$

其中 \blacktriangle 表示假想的二进制小数点。在实际表示中,对短实数和长实数,这个整数位的 1 省略,称隐藏位;对于临时实数不采用隐藏位方案。表 6.3 列出了十进制数 178.125 的实数表示。

表 6.3 实数 178.125 的几种不同表示

实数表示	数 值		
原始十进制数	178.125		
二进制数	10110010.001		
二进制浮点表示	$1.0110010001 \times 2^{11}$		
短实数表示	符号	偏移的阶码	有效值
	0	$00000111 + 01111111$ $= 10000110$	0110010001000000000000 $\uparrow 1_{\blacktriangle}(\text{隐含的})$

6.3 定点运算

定点运算包括移位、加、减、乘、除几种。

6.3.1 移位运算

1. 移位的意义

移位运算在日常生活中常见。例如,15 m 可写成 1 500 cm,单就数字而言,1 500 相当于数 15 相对于小数点左移了两位,并在小数点前面添了两个 0;同样 15 也相当于 1 500 相对于小数点右移了两位,并删去了小数点后面的两个 0。可见,当某个十进制数相对于小数点左移 n 位时,相当于该数乘以 10^n ;右移 n 位时,相当于该数除以 10^n 。

计算机中小数点的位置是事先约定的,因此,二进制表示的机器数在相对于小数点作 n 位左移或右移时,其实质就是该数乘以或除以 $2^n(n=1,2,\cdots,n)$ 。

移位运算称为移位操作,对计算机来说,有很大的实用价值。例如,当某计算机没有乘(除)法运算线路时,可以采用移位和加法相结合,实现乘(除)运算。

计算机中机器数的字长往往是固定的,当机器数左移 n 位或右移 n 位时,必然会使其 n 位低位或 n 位高位出现空位。那么,对空出的空位应该添补 0 还是 1 呢? 这与机器数采用有符号数还是无符号数有关。对有符号数的移位称为算术移位。

2. 算术移位规则

对于正数,由于 $[x]_{原}=[x]_{补}=[x]_{反}$ = 真值,故移位后出现的空位均以 0 添之。对于负数,由于原码、补码和反码的表示形式不同,故当机器数移位时,对其空位的添补规则也不同。表 6.4 列出了三种不同码制的机器数(整数或小数均可),分别对应正数或负数移位后的添补规则。必须注意的是:不论是正数还是负数,移位后其符号位均不变,这是算术移位的重要特点。

表 6.4 不同码制机器数算术移位后的空位添补规则

真值	码 制	添补代码
正数	原码、补码、反码	0
负数	原 码	0
	补 码	左移添 0
	反 码	右移添 1
		1

由表 6.4 可得出如下结论。