第 3 章　运算方法和运算部件

早期的计算机是作为计算工具而应用于科学研究和军事领域中的，对数据进行快速运算是促进计算机诞生和早期发展的动力。目前计算机的应用范围大大地扩展了，但是数据在计算机中是如何表示的，怎样进行运算，如何实现运算仍是最最基本的问题，这就是本章要讨论的课题——运算方法和实现运算的部件。

3.1　数据的表示方法和转换

3.1.1　数值型数据的表示和转换

1. 数制

日常生活中，人们广泛使用十进制数，任意一个十进制数$(N)_{10}$可表示为：

$$\begin{aligned}(N)_{10} &= D_m \cdot 10^m + D_{m-1} \cdot 10^{m-1} + \cdots + D_1 \cdot 10^1 + D_0 \cdot 10^0 \\ &\quad + D_{-1} \cdot 10^{-1} + D_{-2} \cdot 10^{-2} + \cdots + D_{-k} \cdot 10^{-k} \\ &= \sum_{i=m}^{-k} D_i \cdot 10^i \end{aligned} \tag{3.1}$$

其中，$(N)_{10}$的下标 10 表示十进制，该数共有 $m+k+1$ 位，且 m 和 k 为正整数；D_i 可以是 0～9这 10 个数码中的任意一个，根据 D_i 在式中所处位置而赋以一个固定的单位值 10^i，称之为权(Weight)。式中的 10 称为基数或“底”。

在计算机中，十进制数的存储和运算都不太方便，于是二进制计数制应运而生。任意一个二进制数可表示为：

$$\begin{aligned}(N)_2 &= D_m \cdot 2^m + D_{m-1} \cdot 2^{m-1} + \cdots + D_1 \cdot 2^1 + D_0 \cdot 2^0 \\ &\quad + D_{-1} \cdot 2^{-1} + D_{-2} \cdot 2^{-2} + \cdots + D_{-k} \cdot 2^{-k} \\ &= \sum_{i=m}^{-k} D_i \cdot 2^i \end{aligned} \tag{3.2}$$

式中，整数部分有 $m+1$ 位，小数部分有 k 位，基数(或底)为 2。

二进制数$(N)_2$按公式展开，可计算得该数的十进制表示。

例 3.1　$(1101.0101)_2 = (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4})_{10}$

$= (8 + 4 + 0 + 1 + 0 + 0.25 + 0 + 0.062\,5)_{10}$

$= (13.312\,5)_{10}$

然而对人来说，二进制数无论是书写或阅读均很不方便，为此经常采用八进制数或十六进制数。

任意一个八进制数可表示为：

$$(N)_8 = \sum_{i=m}^{-k} D_i \cdot 8^i \tag{3.3}$$

式(3.3)中 D_i 可为 0～7 这 8 个数码中的任意一个。

例 3.2 $(15.24)_8 = (1 \cdot 8^1 + 5 \cdot 8^0 + 2 \cdot 8^{-1} + 4 \cdot 8^{-2})_{10}$

$= (8 + 5 + 0.25 + 0.0625)_{10}$

$= (13.312\ 5)_{10}$

任意一个十六进制数可表示为：

$$(N)_{16} = \sum_{i=m}^{-k} D_i \cdot 16^i \tag{3.4}$$

式(3.4)中，D_i 可以是 0～15 共 16 个数中的任一个。为书写和辨认方便，通常用 0～9 和 A～F 分别表示十六进制数 0～9 和 10～15。

例 3.3 $(0D.5) = (0 \cdot 16^1 + 13 \cdot 16^0 + 5 \cdot 16^{-1})_{10}$

$= (0 + 13 + 0.3125)_{10}$

$= (13.312\ 5)_{10}$

二进制数、八进制数、十六进制数和十进制数之间的关系如表 3.1 所示。

表 3.1　二、八、十六和十进制数的对应关系

二进制数	八进制数	十六进制数	十进制数
0 0 0 0	0 0	0	0
0 0 0 1	0 1	1	1
0 0 1 0	0 2	2	2
0 0 1 1	0 3	3	3
0 1 0 0	0 4	4	4
0 1 0 1	0 5	5	5
0 1 1 0	0 6	6	6
0 1 1 1	0 7	7	7
1 0 0 0	1 0	8	8
1 0 0 1	1 1	9	9
1 0 1 0	1 2	A	1 0
1 0 1 1	1 3	B	1 1
1 1 0 0	1 4	C	1 2
1 1 0 1	1 5	D	1 3
1 1 1 0	1 6	E	1 4
1 1 1 1	1 7	F	1 5

2. 不同数制间的数据转换

(1) 二进制数、八进制数和十六进制数之间的转换

八进制数和十六进制数是从二进制数演变而来的，由 3 位二进制数组成 1 位八进制数，4 位二进制数组成 1 位十六进制数。对于一个兼有整数和小数部分的数，以小数点为界，对小数点前后的数分别分组进行处理，不足的位数用 0 补足，对整数部分将 0 补在数的左侧，对小数部分将 0 补在数的右侧。这样数值不会发生差错。

假如从二进制数转换到八进制数，则以 3 位为 1 组(用下划线表示)。

例 3.4 $(1\ 101.010\ 1)_2=(001\ 101.010\ 100)_2=(15.24)_8$

假如从二进制数转换到十六进制数，则以 4 位为 1 组。

例 3.5 $(1\ 1101.0101)_2=(0001\ 1101.0101)_2=(1D.5)_{16}$

从八进制数或十六进制数转换到二进制数，只要顺序将每一位数写成 3 位或 4 位即可。

例 3.6 $(15.24)_8=(001\ 101.010\ 100)_2=(1101.0101)_2$

八进制数与十六进制数之间的转换，可将二进制数作为中间媒介进行转换。

(2) 二进制数转换成十进制数

利用上面讲到的公式 $(N)_2=\sum_{i=m}^{-k}D_i\cdot 2^i$ 进行计算。可参阅前面的例子，在此不再重复。

(3) 十进制数转换成二进制数

通常要对一个数的整数部分和小数部分分别进行处理，各自得出结果后再合并。

对整数部分，一般采用除 2 取余数法，其规则如下。

将十进制数除以 2，所得余数(0 或 1)即为对应二进制数最低位的值。然后对上次所得的商除以 2，所得余数即为二进制数次低位的值，如此进行下去，直到商等于 0 为止，最后得出的余数是所求二进制数最高位的值。

例 3.7 将$(105)_{10}$转换成二进制。

	余数	结果
2 ⌊105		
2 ⌊52	1	最低位
2 ⌊26	0	
2 ⌊13	0	⋮
2 ⌊6	1	
2 ⌊3	0	
2 ⌊1	1	
0	1	最高位

得出$(105)_{10}=(1101001)_2$

对小数部分，一般用乘 2 取整数法，其规则如下。

将十进制数乘以 2，所得乘积的整数部分即为对应二进制小数最高位的值，然后对所余的小数部分乘以 2，所得乘积的整数部分为次高位的值，如此进行下去，直到乘积的小数部分为 0，或结果已满足所需精度要求为止。

例 3.8 将$(0.312\ 5)_{10}$和$(0.312\ 8)_{10}$转换成二进制数(要求 4 位有效位)。

①

结果		0.3125×2
最高位	0	.6250×2
⋮	1	.2500×2
	0	.5000×2
最低位	1	.0000

得出：$(0.312\ 5)_{10}=(0.0101)_2$

②

结果		0.3128×2
最高位	0	.6256×2
⋮	1	.2512×2
	0	.5024×2
最低位	1	.0048

得出：$(0.312\ 8)_{10}=(0.0101)_2$

例 3.8 中①最后一次乘积的小数部分为 0，转换成精确的二进制数；例 3.8 中②最后一次乘积的小数部分不为 0，因此舍去了更低位的值，现取 4 位有效位，其误差$<2^{-4}$。

当一个数既有整数部分又有小数部分时，分别进行转换后再进行拼接，如有数$(105.312\ 5)_{10}$，则根据前面的计算，得出$(105.312\ 5)_{10}=(1101001.0101)_2$。

(4) 十进制数转换成八进制数

参照十进制数转换成二进制数的方法，将基数 2 改为 8，即可实现转换。

例 3.9 将$(13.312\ 5)_{10}$转换成八进制数，处理过程如下：

整数部分转换

8	13	余数
8	1	5
	0	1

$(13)_{10}=(15)_8$

小数部分转换

	0.3125×8
2	$.5000\times8$
4	$.0000$

$(0.312\ 5)_{10}=(0.24)_8$

得出 $(13.312\ 5)_{10}=(15.24)_8$

3. 数据符号的表示

数据的数值通常以正(+)负(−)号后跟绝对值来表示，称之为“真值”。在计算机中正负号也需要数字化，一般用 0 表示正号，1 表示负号。正号有时可省略，如用$(01001)_2$或$(1001)_2$表示$(+9)_{10}$，$(11001)_2$表示$(-9)_{10}$。

3.1.2 十进制数的编码与运算

1. 十进制数位的编码与运算

在计算机中采用 4 位二进制码对每个十进制数位进行编码。4 位二进制码有 16 种不同的组合，从中选出 10 种来表示十进制数位的 0～9，有多种方案可供选择，下面介绍最常用的几种。

(1) 有权码

表示一位十进制数的二进制码的每一位有确定的权。一般用 8421 码，其 4 个二进制码的权从高到低分别为 8、4、2 和 1。用 0000，0001，…，1001 分别表示 0，1，…，9，每个数位内部满足二进制规则，而数位之间满足十进制规则，故称这种编码为“以二进制编码的十进制(binary coded decimal，BCD)码”。

在计算机内部实现 BCD 码算术运算，要对运算结果进行修正，对加法运算的修正规则如下。

如果两个一位 BCD 码相加之和小于或等于$(1001)_2$，即$(9)_{10}$，不需要修正；如相加之和大于或等于$(10)_{10}$，要进行加 6 修正，并向高位进位，进位可以在首次相加(例 3.10③)或修正时(例 3.10②)产生。

例 3.10 ① 1+8=9

```
  0 0 0 1
+ 1 0 0 0
---------
  1 0 0 1
```

不需要修正

② 4+9=13

```
  0 1 0 0
+ 1 0 0 1
---------
  1 1 0 1
+ 0 1 1 0   修正
---------
1 0 0 1 1
```

进位

③ 9+7=16

```
  1 0 0 1
+ 0 1 1 1
---------
1 0 0 0 0
+ 0 1 1 0   修正
---------
1 0 1 1 0
```

进位

另外几种有权码，如 2421，5211，4311 码（如表 3.2 所示），也是用 4 位二进制码表示一个十进制数位，但 4 位二进制码之间不符合二进制规则。这几种有权码有一特点，即任何两个相加之和等于$(9)_{10}$的二进制码互为反码。例如，在 2421 码中，0(0000)与 9(1111)，1(0001)与8(1110)，…，互为反码。

表 3.2　4 位有权码

十进制数	8421 码	2421 码	5211 码	4311 码
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 0	0 0 1 1	0 0 1 1
3	0 0 1 1	0 0 1 1	0 1 0 1	0 1 0 0
4	0 1 0 0	0 1 0 0	0 1 1 1	1 0 0 0
5	0 1 0 1	1 0 1 1	1 0 0 0	0 1 1 1
6	0 1 1 0	1 1 0 0	1 0 1 0	1 0 1 1
7	0 1 1 1	1 1 0 1	1 1 0 0	1 1 0 0
8	1 0 0 0	1 1 1 0	1 1 1 0	1 1 1 0
9	1 0 0 1	1 1 1 1	1 1 1 1	1 1 1 1

(2) 无权码

表示一个十进制数位的二进制码的每一位没有确定的权。用得较多的是余 3 码(Excess-3 Code)和格雷码(Gray Code)，格雷码又称“循环码”。

余 3 码是在 8421 码基础上，把每个编码都加上 0011 而形成的（如表 3.3 所示），其运算规则如下。

当两个余 3 码相加不产生进位时，应从结果中减去 0011；产生进位时，应将进位信号送入高位，本位加 0011。

例 3.11　$(28)_{10}+(55)_{10}=(83)_{10}$

```
   0 1 0 1      1 0 1 1     (28)10
+) 1 0 0 0   1  1 0 0 0     (55)10
-------------------------
   1 1 1 0      0 0 1 1     低位向高位产生进位，高位不产生进位。
-) 0 0 1 1   +) 0 0 1 1     低位+3，高位-3。(0011)2=(3)106
-------------------------
   1 0 1 1      0 1 1 0
```

表 3.3　4 位无权码

十进制数	余 3 码	格雷码(1)	格雷码(2)
0	0 0 1 1	0 0 0 0	0 0 0 0
1	0 1 0 0	0 0 0 1	0 1 0 0
2	0 1 0 1	0 0 1 1	0 1 1 0
3	0 1 1 0	0 0 1 0	0 0 1 0
4	0 1 1 1	0 1 1 0	1 0 1 0
5	1 0 0 0	1 1 1 0	1 0 1 1
6	1 0 0 1	1 0 1 0	0 0 1 1
7	1 0 1 0	1 0 0 0	0 0 0 1
8	1 0 1 1	1 1 0 0	1 0 0 1
9	1 1 0 0	0 1 0 0	1 0 0 0

格雷码的编码规则：任何两个相邻编码只有 1 个二进制位不同，而其余 3 个二进制位相同。其优点是从一个编码变到下一个相邻编码时，只有 1 位发生变化，用它构成计数器时可得到更好的译码波形（见第 6 章）。格雷码的编码方案有多种，表 3.3 给出两组常用的编码值。

2. 数字串在计算机内的表示与存储

主要有两种形式：

(1) 字符形式。即一个字节存放一个十进制数位或符号位，存放的是 0～9 十个数字和正负号的 ASCII 编码值（有关 ASCII 编码请参阅第 9 章）。

例如，+123 的编码为2B 31 32 33，占用 4 个连续的字节，这里的 2B，31，32 和 33 是用十六进制形式给出的编码，2B 表示正号，31，32 和 33 分别表示数字 1，2 和 3。−123 在主存中为 2D 31 32 33，其中 2D 为负号。

这种表示方式运算起来很不方便，因为它的高 4 位不具有数值的意义，它主要用在非数值计算的应用领域中。

(2) 压缩的十进制数形式。用一个字节存放两个十进制数位，既节省了存储空间，又便于完成十进制数的算术运算。其值用 BCD 码或 ASCII 码的低 4 位表示。符号位也占半个字节并放在最低数字位之后，其值可从 4 位二进制码中的 6 种冗余状态中选用。例如，用 B(11)表示正号，D(13)表示负号。并规定数字和符号位个数之和必须为偶数，否则在最高数字之前补一个 0。例如，+123 被表示成12 3B(2 个字节)，−12 被表示成01 2D(2 个字节)。

3.2 带符号的二进制数据在计算机中的表示方法及加减法运算

在计算机中表示的带符号的二进制数称为“机器数”。机器数有三种表示方式：原码、补码和反码。

为讨论方便，先假设机器数为小数，符号位放在最左面，小数点置于符号位与数值之间。数的真值用 X 表示。

3.2.1 原码、补码、反码及其加减法运算

1. 原码表示法

机器数的最高位为符号位，0 表示正数，1 表示负数，数值跟随其后，并以绝对值形式给出。这是与真值最接近的一种表示形式。

原码的定义：

$$[X]_{原}=\begin{cases}X & 0\leqslant X<1\\ 1-X=1+|X| & -1<X\leqslant 0\end{cases} \tag{3.5}$$

即$[X]_{原}$＝符号位＋$|X|$。

例 3.12 $X=+0.1011$，　　$[X]_{原}=01011$；

$X=-0.1011$，　　$[X]_{原}=11011$。

由于小数点位置已默认在符号位之后，书写时将其省略了。

根据定义，当 $X=-0.1011$ 时，$[X]_{原}=1-(-0.1011)=1.1011$。

数值零的真值有 $+0$ 和 -0 两种表示形式，$[X]_{原}$ 也有两种表示形式：

$$[+0]_{原}=00000,\quad [-0]_{原}=10000。$$

数的原码与真值之间的关系比较简单，其算术运算规则与大家已经熟悉的十进制运算规则类似，当运算结果不超出机器能表示的范围时，运算结果仍以原码表示。它的最大缺点是在机器中进行加减法运算时比较复杂。例如，当两数相加时，先要判别两数的符号，如果两数是同号，则相加；两数是异号，则相减。而进行减法运算又要先比较两数绝对值的大小，再用大绝对值减去小绝对值，最后还要确定运算结果的正负号。下面介绍的用补码表示的数在进行加减法运算时可避免这些缺点。

2. 补码表示法

机器数的最高位为符号位，0 表示正数，1 表示负数，其定义如下：

$$[X]_{补}=\begin{cases}X & 0\leqslant X<1\\ 2+X=2-|X| & -1\leqslant X<0\end{cases}\tag{3.6}$$

即 $[X]_{补}=2\cdot$符号位$+X\quad(\mathrm{mod}\ 2)$

此处 2 为十进制数，即为二进制的 10。

例 3.13　$X=+0.1011$，则 $[X]_{补}=0.1011$

$X=-0.1011$，则 $[X]_{补}=2+X=2+(-0.1011)=1.0101$

数值零的补码表示形式是唯一的，即：

$$[+0]_{补}=[-0]_{补}=0.0000$$

这可根据补码定义计算如下：

当 $X=+0.0000$ 时，　$[X]_{补}=0.0000$。

当 $X=-0.0000$ 时，　$[X]_{补}=2+X=10.0000+0.0000=10.0000=0.0000\quad \mathrm{mod}\ 2$

当补码加法运算的结果不超出机器范围时，可得出以下重要结论：

(1) 用补码表示的两数进行加法运算，其结果仍为补码。

(2) $[X+Y]_{补}=[X]_{补}+[Y]_{补}$。

(3) 符号位与数值位一样参与运算。

今举例说明之。

例 3.14　设 $X=0.1010$，$Y=0.0101$，两数均为正数，则有：

$$[X+Y]_{补}=[0.1010+0.0101]_{补}=[0.1111]_{补}=0.1111$$

$$[X]_{补}+[Y]_{补}=0.1010+0.0101=0.1111$$

即 $[X+Y]_{补}=[X]_{补}+[Y]_{补}=0.1111$

例 3.15　设 $X=0.1010$，$Y=-0.0101$，X 为正，Y 为负，则有：

$$[X+Y]_{补}=[0.1010+(-0.0101)]_{补}=0.0101$$

$$[X]_{补}+[Y]_{补}=0.1010+[-0.0101]_{补}=0.1010+(2-0.0101)$$

$$=2+0.0101=0.0101\quad \mathrm{mod}\ 2$$

即 $[X+Y]_{补}=[X]_{补}+[Y]_{补}=0.0101$

例 3.16 设 $X=-0.1010, Y=0.0101$，X 为负，Y 为正，则有：

$$[X+Y]_{补}=[-0.1010+0.0101]_{补}=[-0.0101]_{补}=1.1011$$

$$[X]_{补}+[Y]_{补}=[-0.1010]_{补}+[0.0101]_{补}=1.0110+0.0101=1.1011$$

即 $[X+Y]_{补}=[X]_{补}+[Y]_{补}=1.1011$

例 3.17 设 $X=-0.1010, Y=-0.0101$，X, Y 均为负数，则有：

$$[X+Y]_{补}=[-0.1010+(-0.0101)]_{补}=[-0.1111]_{补}=1.0001$$

$$[X]_{补}+[Y]_{补}=1.0110+1.1011=10+1.0001=1.0001 \quad \text{mod } 2$$

即 $[X+Y]_{补}=[X]_{补}+[Y]_{补}=1.0001$

在例 3.14～例 3.17 中，包括了 X、Y 各为正负数的各种组合，证实了当运算结果不超出机器所能表示的范围时，$[X+Y]_{补}=[X]_{补}+[Y]_{补}$。

以下再举两个特例。

例 3.18 设 $X=-0.0000$，$Y=-0.0000$

$$[X]_{补}+[Y]_{补}=[X+Y]_{补}=(2+0.0000)+(2+0.0000)=4+0.0000$$

$$=0.0000 \quad \text{mod } 2$$

说明两个负零相加，最后得正零，再次说明了补码零在机器中的表示形式是唯一的。

例 3.19 设 $X=-0.1011$，$Y=-0.0101$，则有：

$$[X+Y]_{补}=[X]_{补}+[Y]_{补}=1.0101+1.1011=11.0000$$

$$=1.0000 \quad \text{mod } 2$$

而 $X+Y$ 的真值 $=-0.1011+(-0.0101)=-1.0000$，为 -1。由此说明一个数的补码值的范围在 -1 和 $(1-2^{-n})$ 之间（假设数值部分为 n 位）。

对于减法运算，因为 $[X-Y]_{补}=[X+(-Y)]_{补}=[X]_{补}+[-Y]_{补}$，所以计算时，可以先求出 $-Y$ 的补码，然后再进行加法运算，这样在用逻辑电路实现加减法运算时，可以只考虑用加法电路，而不必设置减法电路。

图 3.1 为实现加法运算的逻辑示例。

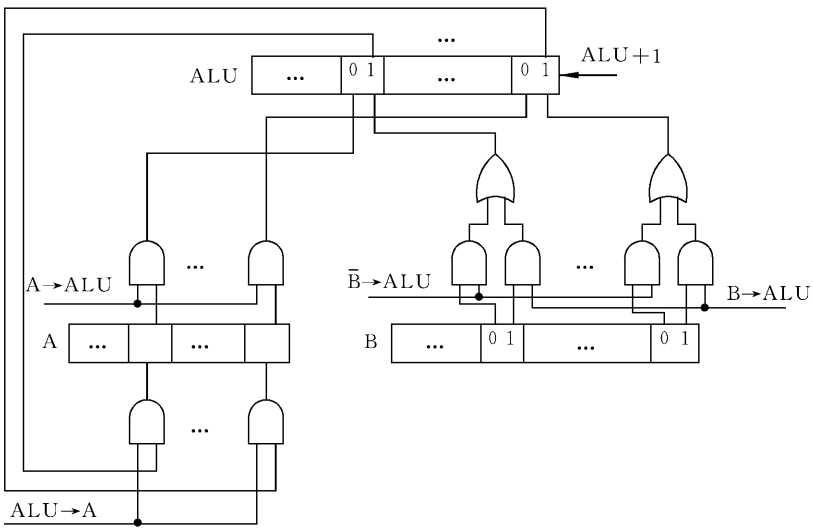

图 3.1 实现加法运算的逻辑示例

在图 3.1 中，被加数(或被减数)X 和加数(或减数)Y 分别存放在 A 寄存器和 B 寄存器中。当执行加法运算时，执行$[X]_{补}+[Y]_{补}$，将$[X]_{补}$和$[Y]_{补}$从 A 寄存器和 B 寄存器送到加法器的两个输入端(加法器及其快速进位逻辑电路参阅第 2 章)。当执行减法运算时，执行$[X]_{补}+[-Y]_{补}$，将运算结果保存在 A 寄存器中。从$[Y]_{补}$形成$[-Y]_{补}$有简便的方法：先将$[Y]_{补}$的各位取反(即 0→1，1→0)，然后在最低位加 1，即可得$[-Y]_{补}$。

假设 $Y=0.1100$，则 $-Y$ 的真值应等于 -0.1100。根据上述方法，先取 Y 数中各位的反值，得 1.0011，然后在最低位加 1，即 $1.0011+0.0001=1.0100$。该值正好是 $-Y$ 的补码。

在逻辑电路中，ALU 由多个全加器及其他电路组成。每个全加器有三个输入端，其中一个接收从低位来的进位信号，而最低位恰好没有进位信号输入，因此可利用来作为“+1”信号，于是可归纳出以下控制信号。

当执行加法时，应提供的控制信号有：

A→ALU，B→ALU(从 B 寄存器的各触发器的 1 端输出)，ALU→A。

当执行减法时，应提供的控制信号有：

A→ALU，$\overline{B}$→ALU(从 B 寄存器的各触发器的 0 端输出)，ALU+1，ALU→A。

其中，ALU+1 操作可以与加法操作同时进行，所以总共只需要进行一次加法运算。

补码的英文表示为 two's complement。

当前大部分计算机字长为 32 位/64 位，一般符号位取 1 位，数值部分取 31 位/63 位。ALU 和寄存器都为 32 位/64 位，最高位产生的进位自动丢弃，满足补码定义中有关“mod 2”的运算规则，不必另行处理。这在运算结果不超出机器能表示的数的范围时，结果是正确的。超出机器数范围的情况称之为溢出，将在 3.2.2 小节介绍。

3. 反码表示法

机器码的最高位为符号，0 表示正数，1 表示负数。

反码的定义：

$$[X]_{反}=\begin{cases}X & 0\leqslant X<1\\ 2-2^{-n}+X & -1<X\leqslant 0\end{cases} \tag{3.7}$$

即：$[X]_{反}=(2-2^{-n})\cdot$符号位$+X\bmod(2-2^{-n})$，其中 n 为小数点后的有效位数。

例 3.20 $X=+0.1011$ $(n=4)$，则$[X]_{反}=0.1011$

$X=-0.1011$ $(n=4)$，则$[X]_{反}=2-2^{-4}+(-0.1011)=1.0100$

即：当 X 为正数时，$[X]_{反}=[X]_{原}$；当 X 为负数时保持$[X]_{原}$符号位不变，而将数值部分取反。反码运算是以 $2-2^{-n}$ 为模，所以，当最高位有进位而丢掉进位(即 2)时，要在最低位+1。

例 3.21 $X=0.1011$，$Y=-0.0100$，则有：

$[X]_{反}=0.1011$，$[Y]_{反}=1.1011$

$[X+Y]_{反}=[X]_{反}+[Y]_{反}=0.1011+1.1011=\underline{1}0.0110$ $\quad \bmod(2-2^{-4})$

其中，最高位 1 丢掉，并要在最低位加 1。所以，得$[X+Y]_{反}=0.0111$

例 3.22 $X=0.1011$，$Y=-0.1100$，则有：

$[X]_{反}=0.1011$，$[Y]_{反}=1.0011$

$[X+Y]_{反}=0.1011+1.0011=1.1110$(其真值为 -0.0001)

反码零有两种表示形式：

$$[+0]_{反}=0.0000,\ [-0]_{反}=1.1111$$

反码运算在最高位有进位时，要在最低位+1，此时要多进行一次加法运算，增加了复杂性，又影响了速度，因此很少采用。

反码的英文表示为 one's complement。

从以上讨论可见，正数的原码、补码和反码的表示形式是相同的，而负数则各不相同。

4. 数据从补码和反码表示形式转换成原码

仿照原码转换成补码或反码的过程再重复执行一遍，即可还原成原码形式。

(1) 将反码表示的数据转换成原码。

转换方法：符号位保持不变，正数的数值部分不变，负数的数值部分取反。

例 3.23 设$[X]_{反}=0.1010$，则$[X]_{原}=0.1010$，真值 $X=0.1010$。

例 3.24 设$[X]_{反}=1.1010$，则$[X]_{原}=1.0101$，真值 $X=-0.0101$。

(2) 将补码表示的数据转换成原码。

例 3.25 设$[X]_{补}=0.1010$，则$[X]_{原}=0.1010$，真值 $X=0.1010$。

例 3.26 设$[X]_{补}=1.1010$，则$[X]_{原}=1.0110$，真值 $X=-0.0110$。

(3) 原码和补码、反码之间相互转换的实现。

在图 3.1 中，给出适当的控制命令，即可实现转换。假设被转换的数 X 存放在 B 寄存器中，结果保留在 A 寄存器中，要求将$[X]_{原}$转换成$[X]_{补}$。

首先判 B 寄存器的符号位 f_B：

如 $f_B=0$，发控制命令 B→ALU，ALU→A，在 A 中得$[X]_{补}$；

如 $f_B=1$，发控制命令 $\overline{B}$→ALU，ALU+1，ALU→A，并令 $f_A=f_B$，于是在 A 中得$[X]_{补}$。

如果在 B 寄存器中存放的是$[X]_{补}$，则可用同样方法转换成$[X]_{原}$。

至于$[X]_{原}$转换成$[X]_{反}$，就更简单了，只要不发控制信号 ALU+1(其余信号与补码转换相同)，即可得$[X]_{反}$。

例 3.27 $[X]_{原}=1.1010$，则$[X]_{反}=1.0101$。$[X]_{反}$最低位加 1，即可得$[X]_{补}=1.0110$。

在计算机中，当用串行电路按位将原码转换成补码形式时(或反之)，经常采取以下方法，自低位开始转换，从低位向高位，在遇到第 1 个 1 之前，保持各位的 0 不变，第 1 个 1 也不变，以后的各位按位取反，最后保持符号位不变，经历一遍后，即可得到补码。

5. 整数的表示形式

设 $X=X_n\cdots X_2X_1X_0$，其中 X_n 为符号位。

(1) 原码

$$[X]_{原}=\begin{cases}X & 0\leqslant X<2^n\\ 2^n-X=2^n+|X| & -2^n<X\leqslant 0\end{cases}\tag{3.8}$$

(2) 补码