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：

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

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：

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

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　$$\begin{aligned}(1101.0101)_2 &= (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} \\ &\quad + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4})_{10} \\ &= (8 + 4 + 0 + 1 + 0 + 0.25 + 0 + 0.0625)_{10} \\ &= (13.3125)_{10}\end{aligned}$$

对人来说，二进制数的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：

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

D_i 可为 0～7 这 8 个数码中的任意一个。

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

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

$= (13.3125)_{10}$

任意一个十六进制数可表示为：

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

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

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

$= (13.3125)_{10}$

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（在例中用下划线表示）。

例 3.4 $(\underline{1}\ \underline{101}.\underline{010}\ \underline{1})_2=(\underline{001}\ \underline{101}.\underline{010}\ \underline{100})_2=(15.24)_8$

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 $(\underline{1}\ \underline{1101}.\underline{0101})_2=(\underline{0001}\ \underline{1101}.\underline{0101})_2=(1D.5)_{16}$

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 $(15.24)_8=(\underline{001}\ \underline{101}.\underline{010}\ \underline{100})_2=(1101.0101)_2$

八进制数与十六进制数之间的转换可以用二进制数作为中间媒介进行。

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
$2\lfloor\underline{105}$		
$2\lfloor\underline{52}$	1	最低位
$2\lfloor\underline{26}$	0	
$2\lfloor\underline{13}$	0	
$2\lfloor\underline{6}$	1	⋮
$2\lfloor\underline{3}$	0	
$2\lfloor\underline{1}$	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$(0.3125)_{10}$和$(0.3128)_{10}$转换成二进制数（要求 4 位有效位）。

①

结果		0.3125×2
最高位	0	$.6250\times2$
	1	$.2500\times2$
⋮	0	$.5000\times2$
最低位	1	.0000

得出$(0.312\,5)_{10}=(0.0101)_2$

②

结果		0.3128×2
最高位	0	$.6256\times2$
	1	$.2512\times2$
⋮	0	$.5024\times2$
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$(105.3125)_{10}$，则根据前面的计算，得出$(105.3125)_{10}=(1101001.0101)_2$。

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	
8	1	余数 5
0		余数 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

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码”（binary coded decimal，BCD）。

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

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 码的基础上，把每个编码都加上 0011 而形成的(如表 3.2 所示)。

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

0101	1011	$(28)_{10}$
+) 1000	1 1000	$(55)_{10}$
1110	0011	低位向高位产生进位，高位不产生进位。
−) 0011	+) 0011	低位+3，高位−3。$(0011)_2=(3)_{10}$
1011	0110	

格雷码的编码规则是：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 3.2 给出两组常用的编码值。

表 3.2　4 位无权码

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

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 3 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：

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

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，书写时可以将其保留或省略。

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，其原码也有两种表示形式：

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

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：

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

即

$$[X]_{补}=2\cdot 符号位+X \mod 2$$

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，　$[X]_{补}=2+X=10.0000+0.0000=10.0000=0.0000 \mod 2$

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。

下面举例说明之。

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：

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

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

$$=2+0.0101=0.0101 \quad \bmod 2$$

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 $X=-0.1010$，$Y=-0.0101$，两数均为负数，则有：

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

$$[X]_{补}+[Y]_{补}=1.0110+1.1011=11.0001=1.0001 \quad \bmod 2$$

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 $X=-0.0000$，$Y=-0.0000$

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

$$=0.0000 \quad \bmod 2$$

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：

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

$$=1.0000 \quad \bmod 2$$

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。

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

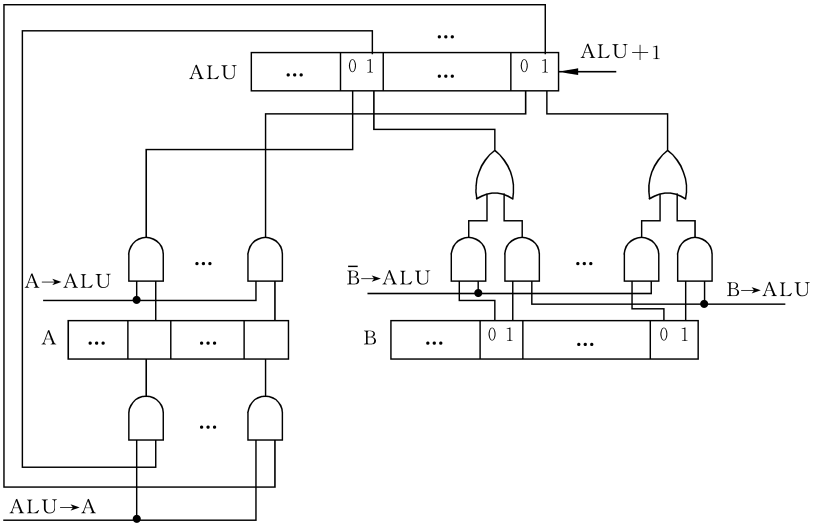

图 3.1　实现加法运算的逻辑示例

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:将 $[Y]_{补}$ 的各位取反(即 0→1,1→0),并在最低位+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,取 Y 数中各位的反值,得 1.0011,并在最低位+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 3 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,$\bar{B}$→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 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:

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

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

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

$X=-0.1011 \quad (n=4)$，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：

$[X]_{反}=0.1011$，$[Y]_{反}=1.1011$

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

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 $X=-0.0110$。

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，在遇到第一个 1 之前，保持各位的 0 不变，第一个 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

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

2）补码

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

3）反码

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

3.2.2 加减法运算的溢出处理

当运算结果超出机器数所能表示的范围时，称为溢出。显然，两个异号数相加或两个同号数相减，其结果是不会溢出的。仅当两个同号数相加或者两个异号数相减时，才有可能发生溢出的情况，一旦溢出，运算结果就不正确了，因此必须将溢出的情况检查出来。

今以 4 位二进制补码整数加法运算为例说明如下。

例 3.27 ① 9+5=14

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

② (−9)+(−5)=−14

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

③ 12+7=19(溢出)

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

④ (−12)+(−7)=−19(溢出)

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

⑤ 14−1=13

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

⑥ −14+1=−13

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

在上例中，①、②、⑤和⑥得出正确结果，③和④为溢出。

现以 f_A，f_B 表示两操作数(A、B)的符号位，f_S 为结果的符号位。符号位 f_A、f_B 直接参与运算，它所产生的进位以 C_f 表示。在以 2^{n+1} 为模的运算中符号位有进位，并不一定表示溢出，在例 3.27 中的②和⑤即是这种情况。假如用 C 来表示数值最高位产生的进位，那么 $C=1$ 也不一定表示溢出，例 3.28 中的②和⑤仍属这种情况。究竟如何判断溢出，实现时有多种方法可供选择，采用其中一种方法即可，现将判别溢出的几种方法介绍如下。

(1) 当符号相同的两数相加时，如果结果的符号与加数(或被加数)不相同，则为溢出。例 3.27 中③和④，溢出条件为 $\bar{f}_A\bar{f}_B f_S+f_A f_B\bar{f}_S$。在计算机中判溢出的逻辑电路如图 3.2 所示，图 3.2(a)和图 3.2(b)是两种不同逻辑电路，但其结果是相同的。

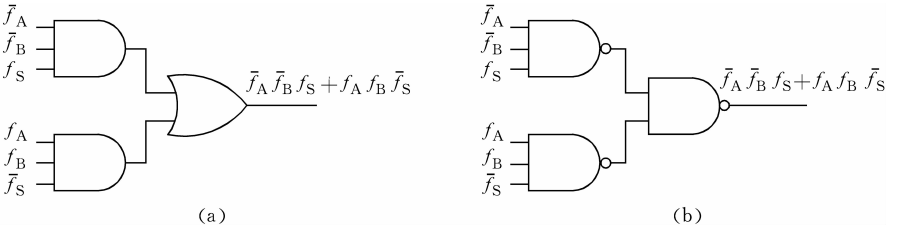

图 3.2 判溢出的逻辑图(之一)

(2) 当任意符号两数相加时，如果 $C=C_f$，运算结果正确，其中 C 为数值最高位的进位，C_f 为符号位的进位。如果 $C\neq C_f$，则为溢出，所以溢出条件为 $C\oplus C_f$。其逻辑电路如图 3.3