第二章 数据的表示和运算

2.1 本章大纲要求与核心考点

2.1.1 大纲内容

（一）数制与编码

1. 进位计数制及其数据之间的相互转换
2. 定点数的表示和运算

（二）运算方法和运算电路

1. 基本运算部件：加法器、算数逻辑部件（ALU）
2. 加/减运算：补码加/减运算器，标志位的生成
3. 乘/除运算：乘/除运算的基本原理，乘法电路和除法电路的基本结构

（三）整数的表示和运算

1. 无符号整数的表示和运算
2. 有符号整数的表示和运算

（四）浮点数的表示和运算

1. 浮点数的表示：IEEE 754标准
2. 浮点数的加/减运算

2.1.2 核心考点

本章内容是考研考察的一个重点和难点，往往会有综合应用题出现。

需要重点掌握的内容包括：

- 真值、机器数，定点数的表示及原理
- C 语言中的整型数据，有符号数与无符号数、不同字长整数之间的类型转换
- ALU 的基本组成，标志位的产生，定点数的运算及相关电路，溢出概念与判断方法
- IEEE 754标准浮点数的表示和特点，浮点数的加/减运算方法
- C 语言中的浮点型数据，浮点型与整型、浮点型之间的类型转换，隐式类型转换
- 数据按边界对齐方式的存储，数据按大端和小端方式存储

2.1.3 真题分布

考点	考查次数	
	单项选择题	综合应用题
定点数的表示与运算	10	8
IEEE 754标准浮点数，浮点数的运算	10	3
C语言中各种数据的转换	3	2
数据按边界对齐方式的存储，数据按大小端方式存储	4	0

2.2 数制与编码

计算机的应用领域极其广泛，但不论其应用在什么地方，信息在机器内部的形式都是一致的，采用的是二进制的表达，即均为0和1组成的各种编码。

2.2.1 进位计数制及其相互转换

（一）进位计数制

进位计数制简称“进制”，是人为定义的一种带进位的计数方法，可以用有限的数字符号表示所有的数。定义好的数字符号的个数，称为**基数**；当计数超出基数个数时，就需要向前进位。基数为n的进位计数制，就被称为“n进制”，特点是“逢n进一”。

我们日常生活中最常见的是十进制，使用0~9十个阿拉伯数字，逢十进一；而计算机系统底层的信息，使用的是二进制，也就是只有0和1两个数字，逢二进一。在计算机系统中，也经常使用八进制和十六进制来表示数据。下表是十进制数、二进制数、十六进制数对照表。

书写时，可在十六进制数后面加上“H”，如17DBH 或$(17DB)_{16}$；八进制数后面加上“O”，如372O或$(372)_8$；若在数的后面加上“B”，如10101100B，即表示此数为二进制数，或写成$(10101100)_2$。

十进制数、二进制数、八进制数、十六进制数对照表

十进制数	二进制数	八进制数	十六进制数	十进制数	二进制数	八进制数	十六进制数
0	00000	0	0	16	10000	20	10
1	00001	1	1	17	10001	21	11
2	00010	2	2	18	10010	22	12
3	00011	3	3	19	10011	23	13
4	00100	4	4	20	10100	24	14
5	00101	5	5	21	10101	25	15
6	00110	6	6	22	10110	26	16
7	00111	7	7	23	10111	27	17
8	01000	10	8	24	11000	30	18
9	01001	11	9	25	11001	31	19
10	01010	12	A	26	11010	32	1A
11	01011	13	B	27	11011	33	1B
12	01100	14	C	28	11100	34	1C
13	01101	15	D	29	11101	35	1D
14	01110	16	E	30	11110	36	1E
15	01111	17	F	31	11111	37	1F

计算机系统为什么要采用二进制?

- 使用有两个稳定状态的物理器件就可以表示二进制数的每一位，制造成本比较低。
- 二进制的1和0正好与逻辑值“真”和“假”对应，为计算机实现逻辑运算提供了便利。
- 二进制的编码和运算规则都很简单，通过逻辑门电路能方便地实现算术运算。

（二）不同进制数的相互转换

任意一个数 N，可以用 r 进制表示成下面的形式:

$$N = (d_{n-1}d_{n-2} \ldots d_1d_0.d_{-1}d_{-2} \ldots d_{-m})$$

$$= d_{n-1}r^{n-1} + d_{n-2}r^{n-2} + \ldots + d_1r^1 + d_0r^0 + d_{-1}r^{-1} + d_{-2}r^{-2} + \ldots + d_{-m}r^{-m}$$

$$= \sum d_ir^i$$

其中，r 为**基数**；d 为系数，d_i 代表第 i 位上的数，可以是 0 ~ (r-1) 中的任意一个数字；r^i 叫做第 i 位上的**权值**。n、m 分别代表 N 的整数部分和小数部分的位数。

（1）二进制和八进制、十六进制间的转换

二进制数数位较多，书写不方便，在计算机系统中一般需要进行“缩写”。由于 $2^3=8$， $2^4=16$，从而3位二进制数就对应着一个8进制数、4位二进制数对应着一个16进制数；对于一个小数而言，以小数点为界，整数部分从小数点左侧第一位起向左数，小数部分从小数点右侧第一位起向右数，不够就补0。这样二进制数和八进制数、十六进制数就可以非常方便地互相转换了。

例如，将二进制数1110011101.0010111转换为八进制数为：

左侧补0　　　　　　分界点　　　　　　右侧补0

↓　　　　　　　　　↓　　　　　　　↓

<u>001</u> <u>110</u> <u>011</u> <u>101</u> . <u>001</u> <u>011</u> <u>100</u>

1　　6　　3　　5 .　1　　3　　4

所以 $(1110011101.0010111)_2 = (1635.134)_8$；

同样道理，转换为十六进制数为：

<u>0011</u> <u>1001</u> <u>1101</u> . <u>0010</u> <u>1110</u>

3　　9　　D .　2　　E

所以 $(1110011101.0010111)_2 = (39D.2E)_{16}$；

- 二进制转换为八进制：每数三位就转换成对应的八进制数，位数不够则补0。
- 二进制转换为十六进制：每数四位就转换成对应的十六进制数，位数不够则补0。
- 八进制转换为二进制：每位都转换成对应的3位二进制数。
- 十六进制转换为二进制：每位都转换成对应的4位二进制数。

（2）任意进制数转换为十进制数

任意进制数的各位数码与它的权值相乘，再把乘积相加，即得到相应的十进制数。这种转换方式称为 **按权展开法。**

例如，将二进制数 11011.101 转换为十进制数为：

$(11011.101)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$

$= 27.625$

另一种方法是“**按基数重复相乘/除法**”，需要分整数部分和小数部分分别转换。

整数部分从高到低，将每一位乘以基数值、再加上后一位，进行“重复相乘”：

$(11011)_2 = (((1 \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 = 27$

小数部分从低到高，将每一位除以基数值、再加上前一位，进行“重复相除”：

$(0.101)_2 = ((1 \div 2 + 0) \div 2 + 1) \div 2 + 0 = 0.625$

（3）十进制数转换为二进制数

将十进制数转换为二进制数，一般采用 **基数乘除法。**整数部分和小数部分分别处理，最后将整数部 分与小数部分的转换结果拼接起来。

- 整数部分的转换规则：除2取余，最先取得的余数为数的最低位，最后取得的余数为数的最高位，商为0时结束。 （**即除2取余，先余为低，后余为高**）
- 小数部分的转换规则：乘2取整，最先取得的整数为数的最高位，最后取得的整数为数的最低位，乘积为0或精度满足要求时结束。（**即乘2取整，先整为高，后整为低**）

例如，将十进制数 123.6875 转换为二进制数。

整数部分：

除2得商		余数	
2 |$\underline{123}$	…	1	最低位
2 |$\underline{61}$	…	1	
2 |$\underline{30}$	…	0	
2 |$\underline{15}$	…	1	
2 |$\underline{7}$	…	1	
2 |$\underline{3}$	…	1	
2 |$\underline{1}$	…	1	最高位
2 |$\underline{0}$			

所以 $(123)_{10} = (1111011)_2$

小数部分：

乘积取小数		乘2得积	取整数部分	
0.6875	× 2	= 1.375	1	最高位
0.375	× 2	= 0.75	0	
0.75	× 2	= 1.5	1	
0.5	× 2	= 1	1	最低位

所以 $(0.6875)_{10} = (0.1011)_2$

综合整数和小数部分，得到 $(123.6875)_{10} = (1111011.1011)_2$

另一种方法是“减权定位法”，利用记忆好的2的幂次的十进制表示，从原始数中依次减去所含最大的2的幂次，就可以快速得到对应的结果。例如，对于十进制数123：

十进制数	位权	转换后的结果						
123		2^6	2^5	2^4	2^3	2^2	2^1	2^0
$\underline{-\ 64}$	2^6	1						
59								
$\underline{-\ 32}$	2^5		1					
27								
$\underline{-\ 16}$	2^4			1				
11								
$\underline{-\ 8}$	2^3				1			
3								
$\underline{-\ 2}$	2^1						1	

1

— 1　　　　2^0　　　　　　　　　　　　1

0

所以 $(123)_{10} = (1111011)_2$

这种方法一般在转换很大的十进制数时比较方便。

2.2.2 真值和机器数

在计算机中，如果不加特别的定义，用二进制存储的数都是非负数，不需要加正负号，也就是“无符号数”。

对有符号数而言，符号的“正”、“负”机器本身是无法识别的；不过由于“正”、“负”恰好是两种截然不同的状态，我们可以用“0”表示“正”，用“1”表示“负”，这样符号也被数字化了，并且规定将它放在有效数字的前面，即组成了有符号数。

例如，一个有符号的小数：

+ 0.1011　　在机器中表示为　　0 1 0 1 1

- 0.1011　　在机器中表示为　　1 1 0 1 1

再比如，一个有符号的整数：

+ 1100　　在机器中表示为　　0 1 1 0 0

- 1100　　在机器中表示为　　1 1 1 0 0

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。

- 真值：正、负号加某进制数绝对值的形式，即机器数所代表的实际值。
- 机器数：一个数值数据的机内编码，即符号和数值都数码化的数。常用的有原码和补码表示法等，这几种表示法都将数据的符号数字化，通常用“0”表示“正”，用“1”表示“负”。

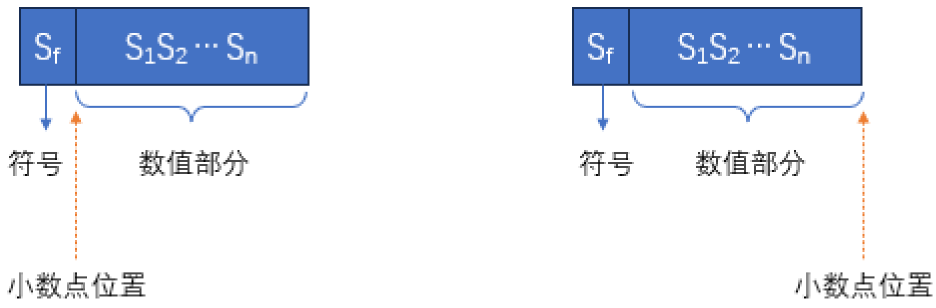
在计算机中，小数点不用专门的器件表示，而是按约定的方式标出。根据小数点位置是否固定，可以分为两种方法表示小数点的存在，即**定点**表示和**浮点**表示。

另外需要考虑的问题是：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。

在现代计算机中，通常用**定点补码整数表示整数**，用定**点原码小数表示浮点数的尾数部分**，用**移码表示浮点数的阶码**部分。

2.2.3 定点数及其编码表示

小数点固定在某一位置的数为定点数，有以下两种格式。



当小数点位于数符和第一数值位之间时，机器内的数为纯小数；当小数点位于数值位之后时，机器内的数为纯整数。采用定点数的机器称为定点机。数值部分的位数n决定了定点机中数的表示范围。

在定点机中，由于小数点的位置固定不变，故当机器处理的数不是纯小数或纯整数时，必须乘上一个比例因子，否则会产生“溢出”。

1. 无符号整数的表示

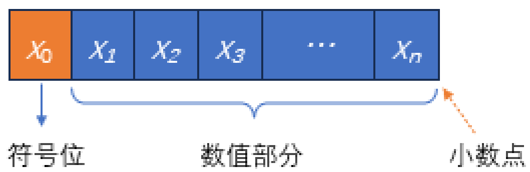
当一个编码的全部二进制位均为数值位时，相当于数的绝对值，该编码表示无符号整数。在字长相同的情况下，它能表示的最大数比带符号整数大。例如，8位无符号整数的表示范围为 $0 \sim 2^8-1$，也就是能表示的最大数为255；而8位带符号整数的最大数是127。通常，在全部是正数运算且不出现负值结果的情况下，使用无符号整数表示。例如，可用无符号整数进行地址运算，或用它来表示指针。

2. 带符号数的表示

最高位用来表示符号位，而不再表示数值位。

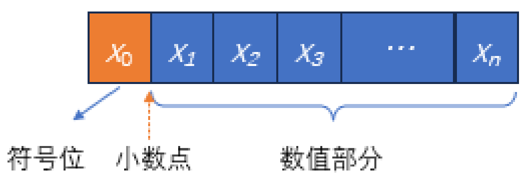
（1）定点整数

约定小数点在有效数值部分最低位之后。数据 $x = x_0x_1x_2 \ldots x_n$ （其中 x_0 为符号位，$x_1 \sim x_n$ 是数值的有效部分，也称尾数），在计算机中的表示形式如图所示：



（2）定点小数

约定小数点在有效数值部分最高位之前。数据 $x = x_0.x_1x_2 \ldots x_n$（其中 x_0 为符号位，$x_1 \sim x_n$ 是尾数，x_1 是最高有效位），在计算中的表示形式如下图所示：



事实上，在计算机中，并没有小数点的表示，只是认为约定了小数点的位置：小数点在最右边的就是定点整数，在最左边的就是定点小数。它们原理相同，只是由于小数点位置不同而可以表示不同范围的数。我们这里重点只考虑定点整数就可以了。

3. 原码、补码、反码和移码

对于有符号的定点数，真实底层的机器数怎样表示，跟选择的编码方式有关。计算机中常用的编码方式有原码、补码、反码和移码。

1. **原码表示法**

用机器数的最高位表示数的符号，其余各位表示数的绝对值。纯小数的原码定义如下:

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

式子中 x 为真值，$[x]_{原}$ 表示原码机器数。

类似，纯整数的原码定义如下:

$$[x]_{原} = \begin{cases} x & 0 \le x < 2^n \\ 2^n - x & -2^n < x \le 0 \end{cases}$$

原码的性质:

- 由**符号位与数的绝对值组成，符号位是0为正、1为负**
- 简单直观，与真值的转换简单
- 0有 ±0 两个编码，即 $[+0]_{原}$ = 00000 和 $[-0]_{原}$ = 10000
- 原码加减运算规则比较复杂，乘除运算规则简单

2. **补码表示法**

纯整数的补码定义为:

$$[x]_{补} = \begin{cases} x & 0 \le x < 2^n \\ 2^{n+1} + x = 2^{n+1} - |x| & -2^n \le x < 0 \end{cases} \quad (\bmod 2^{n+1})$$

这里 n 为整数的位数，真值 x 和补码机器数 $[x]_{原}$ 互为以 2^{n+1} 为模的补数。如果字长为 n+1，那么补码的表示范围为 $-2^n \le x \le 2^n - 1$，比原码多表示了一个数 -2^n。

补码的性质:

- 补码和其真值的关系: $[x]_{补}$ = 符号位 × 2^{n+1} + x
- 0的编码唯一，因此整数补码比原码多1个数，表示 -2^n
- 符号位参与补码加减运算，统一采用加法操作实现
- 将 $[x]_{补}$ 的符号位与数值位一起右移并保持原符号位的值不变，可实现除法功能

例如，当 x = + 1010 时（n = 4），

$[x]_{补}$ = 0, 1010

而当 x = - 1010 时，

$[x]_{补} = 2^{n+1} + x = 100000 - 1010 = 1, 0110$

补码和真值的转换：

- 真值转为补码：对于正数， 与原码的转换方式一样；**对于负数，符号位为1，其余各位由真值“取反加1”得**到。
- 补码转为真值：若符号位为0，真值为正，跟原码的转换一样；若符号位为1，真值为负，其数值部分（绝对值）各位由补码**“取反加1”**得到。

变形补码是采用双符号位的补码表示法，其定义为

$$[x]_{补} = \begin{cases} x & 0 \le x < 2^n \\ 2^{n+2} + x = 2^{n+2} - |x| & -2^n \le x < 0 \end{cases} \quad (\bmod 2^{n+2})$$

变形补码用于算术运算的ALU部件中，双符号位00表示正，11表示负，10和01表示溢出。

3. **反码表示法**

负数的补码可采用“数值位各位取反，末位加1”的方法得到，如果数值位各位取反而末位不加1，那么就是负数的反码表示。正数的反码定义和相应的补码（或原码）表示相同。

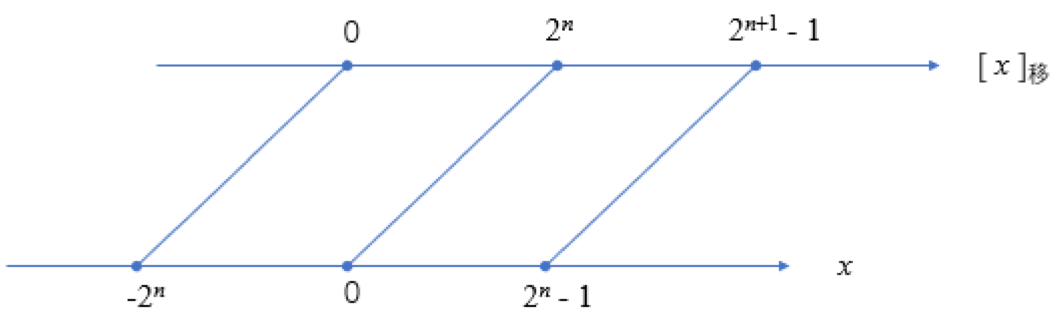
反码表示存在以下几个方面的不足：0的表示不唯一（即存在±0）；表示范围比补码少一个最小负数。反码在计算机中很少使用，通常用作数码变换的中间表示形式。

原码、补码、反码三种编码表示总结如下：

- 三种编码的符号位相同，正数的机器码相同。
- 原码和反码的表示在数轴上对称，二者都存在 ±0 两个零。
- 补码的表示在数轴上不对称，0的表示唯一，补码比原码和反码多表示一个数。
- 负数的反码、补码末位相差1。
- 原码很容易判断大小。而负数的补码和反码很难直接判断大小，可采用这条规则快速判断：对于一个负数，数值部分越大，它的绝对值就越小，所以真值就越大（更靠近0）。

4. **移码表示法**

移码是在真值 x 上加上偏置值 2^n 构成的，相当于 x 在数轴上向正方向偏移了若干单位。



移码定义为：

$$[x]_{移} = 2^n + x \quad (-2^n \le x < 2^n)$$

移码的性质：

- 0的表示唯一， $[+0]_{移} = 2^n + 0 = [-0]_{移} = 2^n - 0 = 100...0$
- 符号位“1”表示正， “0”表示负，这与其他机器数正好相反。

- 一个真值的移码和补码仅差一个符号位，$[x]_{补}$ 的符号位取反即得 $[x]_{移}$，反之亦然。
- 移码全0时，对应真值的最小值- 2^n；移码全1时，对应真值的最大值 2^n -1。
- 保持了数据原有的大小顺序，移码大真值就大，便于进行比较操作。
- 移码常用来表示浮点数的阶码。它只能表示整数。

2.2.4 C 语言中的整型数据类型

1. C 语言中的整型数据简介

C 语言中的整型数据就是定点整数，一般用补码表示。根据位数的不同，可以分为 **字符型(char)**、**短整型(short)**、**整型(int)**、**长整型(long)**。

C 语言中的整型数据，可以分为 **无符号整型** 和 **有符号整型** 两种类型，在定义时只要加上 **signed/unsigned** 就可以明确指定了。

char 是整型数据中比较特殊的一种，其他如 short/int/long 等都默认是带符号整数，但 char 默认是无符号整数。无符号整数（unsigned short/int/long）的全部二进制位均为数值位，没有符号位，相当于数的绝对值。

signed/unsigned 整型数据都是按补码形式存储的，在不溢出条件下的加减运算也是相同的，只是 signed 型的最高位代表符号位，而在 unsigned 型中表示数值位，而这两者体现在输出上则分别是%d 和%u。

2. 有符号数和无符号数的转换

C 语言允许在不同的数据类型之间做类型转换。C 语言的强制类型转换格式为 “TYPE b = (TYPE) a”，强制类型转换后，返回一个具有TYPE类型的数值，这种操作并不会改变操作数本身。

先看由 short 型转换到 unsigned short 型的情况。考虑如下代码片段：

```
short x = -4321;

unsigned short y = (unsigned short)x;
```

执行上述代码后，x = -4321, y = 61215，得到的 y 似乎与原来的 x 没有一点关系。不过将这两个数转化为二进制表示时，我们就会发现其中的规律。

通过本例可知：强制类型转换的结果是保持每位的值不变，仅改变了解释这些位的方式。有符号数转化为等长的无符号数时，符号位解释为数据的一部分，负数转化为无符号数时数值将发生变化。同理，无符号数转化为有符号数时，最高位解释为符号位，也可能发生数值的变化。

3. 不同字长整数之间的转换

另一种常见的运算是在不同字长的整数之间进行数值转换。

先看长字长变量向短字长变量转换的情况。考虑如下代码片段：

```
int x = 165537, u = -34991;                          //int型占用4字节

short y = ( short )x, v = ( short )u;            //short型占用2字节
```

执行上述代码后，x= 165537, y=-31071, u =-34991, v = 30545。x、y、u、v 的十六进制表示分别 是0x000286a1 0x86a1 . 0xffff7751、0x7751。由本例可知：长字长整数向短字长整数转换时，系统把多余的高位部分直接截断，低位直接赋值，因此也是一种保持位值的处理方法。

最后来看短字长变量向长字长变量转换的情况。考虑如下代码片段：

```
short x = -4321;

int y = (int)x;

unsigned short u = (unsigned short)x;

unsigned int v = (unsigned int)u;
```

执行上述代码后，x = -4321, y = -4321, u = 61215, v = 61215。x、y、u、v 的十六进制表示分别是0xef1f. 0xffffef1f、0xef1f、0x0000ef1f。所以，短字长整数向长字长整数转换时，仅要使相应的位值相等，还要对高位部分进行扩展。如果原数字是无符号整数，则进行零扩展，扩展后的高位部分用 0填充。否则进行符号扩展，扩展后的高位部分用原数字符号位填充。其实两种方式扩展的高位部分都可理解为原数字的符号位。

从位值与数值的角度看，前3个例子的转换规则都是保证相应的位值相等，而短字长到长字长的转换可以理解为保证数值的相等。

2.3 运算方法和运算电路

2.3.1 基本运算部件

1. 运算器的基本组成

运算器由算术逻辑单元(ALU)、累加器(AC)、状态寄存器(PSW)、通用寄存器组等组成。

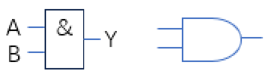
- 算术逻辑单元：完成加、减、乘、除四则运算，与、或、非、异或等逻辑运算。
- 累加器：暂存参加运算的操作数和结果的部件，为 ALU 执行运算提供一个工作区。
- 状态寄存器：也称作标志寄存器，用来记录运算结果的状态信息。
- 通用寄存器组：保存参加运算的操作数和运算结果。

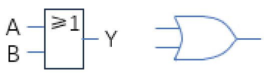
2. 逻辑门电路和逻辑运算（复习）

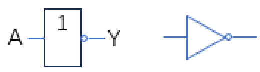
用半导体元器件可以构建出基本的逻辑门电路（与、或、非），能够表示基本的逻辑运算。

与　　或　　非

门电路







真值表

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

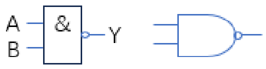
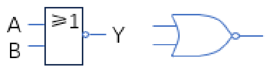
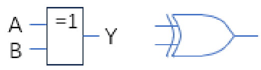
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	Y
0	1
1	0

表达式

$Y = A \cdot B$　　$Y = A + B$　　$Y = \overline{A}$

通过对与门、或门、非门的组合，可以构建出更加复杂的逻辑电路，进行各种复杂的组合逻辑运算。

	与非	或非	异或
门电路			

真值表

与非：

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

或非：

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

异或：

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

表达式　　$Y=\overline{A\cdot B}$　　$Y=\overline{A+B}$　　$Y=A\oplus B$

逻辑运算中的“与”类似于算术中的乘法， “或”类似于算术中的加法，两者组合在一起时，与运算的优先级要更高。逻辑运算满足以下的规则：

交换律	$A\cdot B=B\cdot A$	$A+B=B+A$	$A\oplus B=B\oplus A$	
结合律	$A\cdot(B\cdot C)=A\cdot B\cdot C$	$A+(B+C)=A+B+C$	$A\oplus(B\oplus C)=A\oplus B\oplus C$	
分配律	$(A+B)\cdot C=A\cdot C+B\cdot C$	$(A\cdot B)+C=(A+C)\cdot(B+C)$		
吸收律	$A+(A\cdot B)=A$	$A\cdot(A+B)=A$	$A+(\bar{A}\cdot B)=A+B$	$A\cdot(\bar{A}+B)=A\cdot B$
反演律（德·摩根定律）	$\overline{A\cdot B}=\bar{A}+\bar{B}$	$\overline{A+B}=\bar{A}\cdot\bar{B}$		

3. 全加器（复习）

全加器（Full Adder，FA），是用逻辑门电路实现两个二进制数相加并求出和的组合线路，这称为一位全加器。一位全加器可以处理低位进位，并输出本位加法进位。多个一位全加器进行级联可以得到多位全加器。

一位全加器的真值表如下所示，其中 A 为被加数，B 为加数，相邻低位传来的进位数为 C_{in}，输出本位和为S，向相邻高位输出的进位数为 C_{out} 。

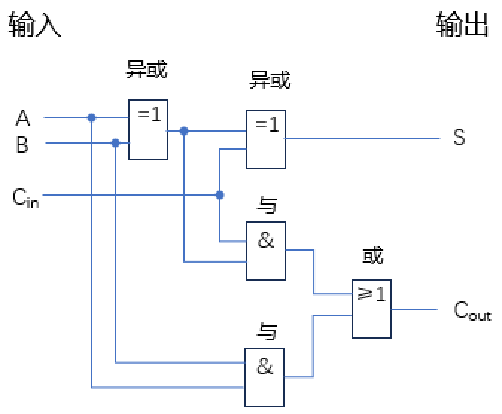
C_{in}	A	B	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

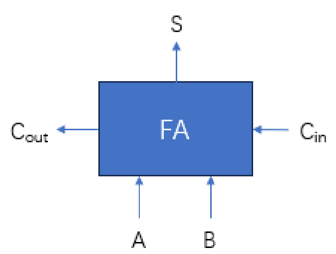
根据真值表，很容易写出一位全加器的输出表达式：

$$\begin{aligned} S &= \bar{A}\,B\,\bar{C}_{in} + A\,\bar{B}\,\bar{C}_{in} + \bar{A}\,\bar{B}\,C_{in} + A\,B\,C_{in} \\ &= A \oplus B \oplus C_{in} \end{aligned}$$

$$\begin{aligned} C_{out} &= A\,B\,\bar{C}_{in} + \bar{A}\,B\,C_{in} + A\,\bar{B}\,C_{in} + A\,B\,C_{in} \\ &= A\,B + (A \oplus B)\,C_{in} \\ &= A\,B + (A + B)\,C_{in} \end{aligned}$$

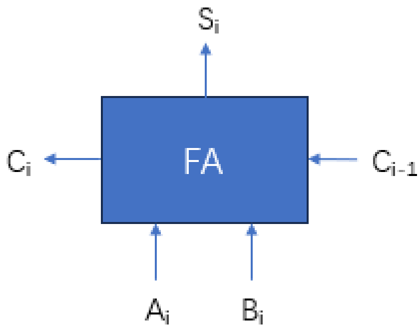
所以，一位全加器可以利用两个异或门、两个与门和一个或门来实现：

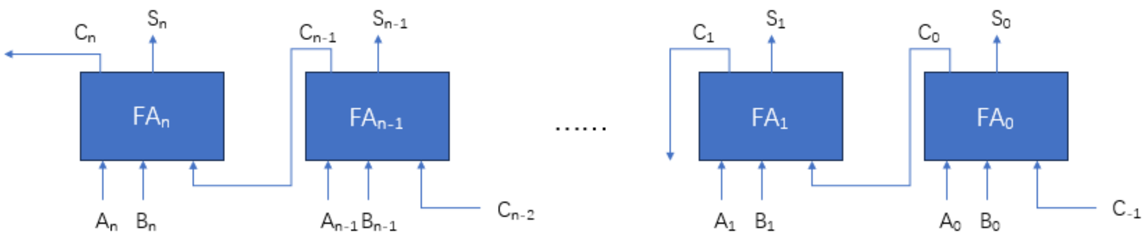




4. 并行加法器

对于 n 位加法器，可以用 n 个全加器（实现两个本位数加上低位进位，生成一个本位和一个向高位的进位）串接起来实现逐位相加，位间进行串行传送，称为 **串行进位加法器。**





这样，一位全加器的输出表达式可以写为：

$$
\begin{aligned}
S_i &= \overline{A}_i B_i \overline{C}_{i-1} + A_i \overline{B}_i \overline{C}_{i-1} + \overline{A}_i \overline{B}_i C_{i-1} + A_i B_i C_{i-1} \\
&= A_i \oplus B_i \oplus C_{i-1}
\end{aligned}
$$

$$
\begin{aligned}
C_i &= A_i B_i \overline{C}_{i-1} + \overline{A}_i B_i C_{i-1} + A_i \overline{B}_i C_{i-1} + A_i B_i C_{i-1} \\
&= A_i B_i + (A_i \oplus B_i) C_{i-1} \\
&= A_i B_i + (A_i + B_i) C_{i-1}
\end{aligned}
$$

在串行进位链中，进位按串行方式传递，高位仅依赖低位进位，因此速度较慢。

为了提高加法器的速度，必须尽量避免进位之间的依赖。引入进位生成函数和进位传递函数，可以使各个进位并行产生，这种以并行进位方式实现的加法器称为 **并行进位加法器**。

在全加器的表达式中可以看到，进位信号 C_i 由两部分组成：

- A_iB_i 与低位无关，可以称为“本地进位”，记作 d_i；
- $(A_i + B_i) C_{i-1}$ 与低位进位 C_{i-1} 有关，可以称为“传递进位”，系数 $(A_i + B_i)$ 称作“传递系数”，记作 t_i。

这样进位信号就可以简写为：

$$C_i = d_i + t_i C_{i-1}$$

以 4 位并行加法器为例，串行进位链的进位表达式就可以写为：

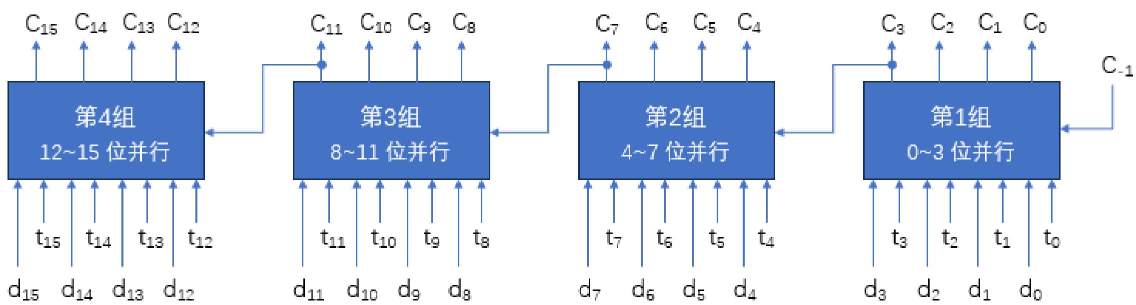
$$C_0 = d_0 + t_0 C_{-1}$$
$$C_1 = d_1 + t_1 C_0$$
$$C_2 = d_2 + t_2 C_1$$
$$C_3 = d_3 + t_3 C_2$$

如果我们将 C_0 的表达式代入 C_1，再依次迭代进 C_2、C_3，那么所有进位信号就只依赖于 C_{-1} 了：

$$C_0 = d_0 + t_0 C_{-1}$$
$$C_1 = d_1 + t_1 C_0 = d_1 + t_1 d_0 + t_1 t_0 C_{-1}$$
$$C_2 = d_2 + t_2 C_1 = d_2 + t_2 d_1 + t_2 t_1 d_0 + t_2 t_1 t_0 C_{-1}$$
$$C_3 = d_3 + t_3 C_2 = d_3 + t_3 d_2 + t_3 t_2 d_1 + t_3 t_2 t_1 d_0 + t_3 t_2 t_1 t_0 C_{-1}$$

并行进位链又称先行进位、跳跃进位，理想的并行进位链就是 n 位全加器的 n 个进位全部同时产生，但实际实现会有困难。一般会使用分组的方式来进行实现，小组内的进位同时产生，小组之间则采用串行进位，这种方式可以总结为“组内并行、组间串行”。

例如，对于 16 位的并行全加器，我们可以 4 位分为一组，得到并行进位链如下：

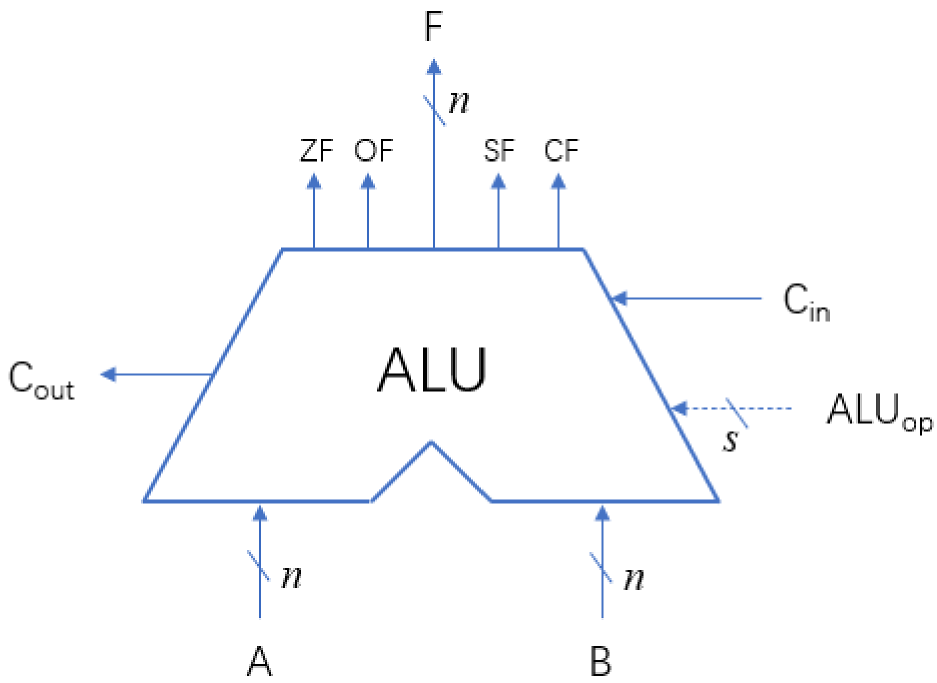


当 n 越来越大时，随着分组的增加，组间串行进位的延迟时间也会越来越长。解决策略是可以采用多重分组，让“小组”间的进位也同时产生，而“大组”间采用串行进位。当有两个层级的分组时，称为“双重分组跳跃进位”；与之对应，之前只有一个层级的分组方式，被称为“单重分组跳跃进位”。

5. 算术逻辑单元（ALU）

针对每一种算术运算，都必须有相对应的基本硬件配置，其核心部件就是加法器和寄存器；而当需要完成逻辑运算时，又必须配置相应的逻辑电路。将这些功能结合起来，就构成了“算术逻辑单元”（ALU）。

ALU 是一种能进行多种算术运算和逻辑运算的组合逻辑电路，ALU的核心是“带标志加法器”，基本结构如下所示。



其中 A 和 B 是两个 n 位操作数输入端；C_{in} 是进位输入端；ALU_{op} 是操作控制端，用来控制 ALU 所执行的处理操作。例如，ALU~op~ 选择 Add 运算，ALU 就执行加法运算，输出的结果就是 A 加 B 之和。ALU~op~ 的位数决定了操作的种类。例如，当位数为3时，ALU最多只有8种操作。

F 是结果输出端，此外，还输出相应的标志信息（ZF、OF、SF、CF）；在ALU进行加法运算时，可以得到最高位的进位 C_{out} 。

2.3.2 定点数的移位运算

移位运算根据操作对象的不同，可以分为算术移位和逻辑移位。算术移位针对的是有符号数，逻辑移位针对的是机器码，可以看作无符号数。

1. 算术移位

算术移位的对象是有符号数，有符号数在计算机中采用补码表示。算术移位的特点是，移位后符号位保持不变；空出的位置根据正负和左右移位的情况，决定补 0 还是 1。

- 对于正数，由于 $[x]_{原} = [x]_{补}$ = 真值，因此移位后的空位均补 0。
- 对于负数，算术左移时，高位移出，低位补 0；算术右移时，低位移出，高位补 1。

  可见，不论是正数还是负数，移位后其符号位均不变。

  例如，假设机器字长为 8，$[4]_{补}$ = 0000 0100，$[-4]_{补}$ = 1111 1100；

- 将 4 算术左移一位，就得到了 0000 1000 = $[8]_{补}$；算术右移一位，就得到了 0000 0010 = $[2]_{补}$；
- 将 -4 算术左移一位，就得到了 1111 1000 = $[-8]_{补}$；算术右移一位，就得到了 1111 1110 = $[-2]_{补}$；

对于有符号数，左移一位若不产生溢出，相当于乘以2 （与十进制数左移一位相当于乘以10类似）；右移一位，若不考虑因移出而舍去的末位尾数，相当于除以2。

2. 逻辑移位

逻辑移位不考虑符号位。

移位规则：逻辑左移时，高位移出，低位补 0；逻辑右移时，低位移出，高位补 0 。

2.3.3 定点数的加减运算

加减法运算是计算机中最基本的运算，由于减法可以看成是负值的加法，因此计算机中使用补码表示有符号数之后，可以将减法运算和加法运算合并在一起讨论。

1. 补码的加减运算

补码加减运算的规则简单，易于实现。补码加减运算的公式如下（设机器字长为 n）：

$$[A + B]_{补} = [A]_{补} + [B]_{补} \pmod{2^n}$$

$$[A - B]_{补} = [A]_{补} + [-B]_{补} \pmod{2^n}$$

补码运算的特点如下：

- 按二进制运算规则运算，逢二进一。
- 如果做加法，两数的补码直接相加；如果做减法，则将被减数加上减数的机器负数。
- 符号位与数值位一起参与运算，加、减运算结果的符号位也在运算中直接得出。
- 最终将运算结果的高位丢弃，保留 n 位，运算结果也是补码。

例如，假设机器字长为 8 （n = 8），那么

$[5]_{补}$ = 0000 0101，$[4]_{补}$ = 0000 0100；

$[-5]_{补}$ = 1111 1011，$[-4]_{补}$ = 1111 1100；

$[5 + 4]_{补}$ = 0000 0101 + 0000 0100 = 0000 1001 = $[9]_{补}$；

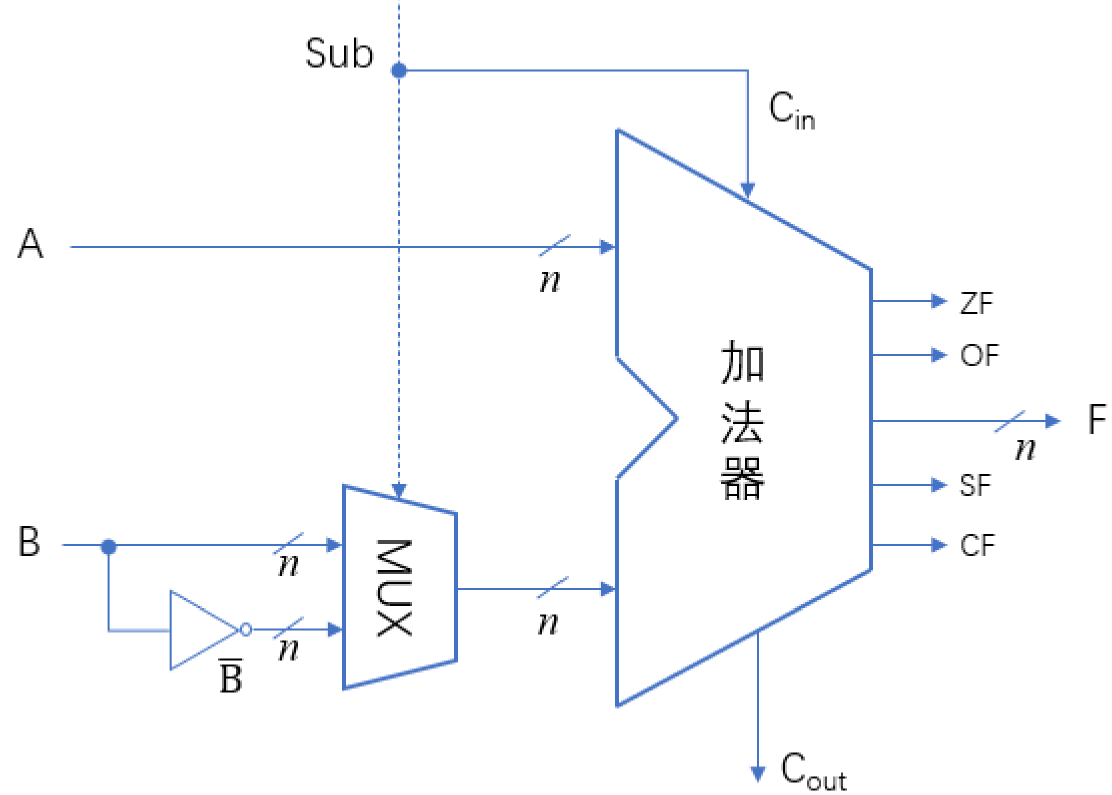
$[5 - 4]_{补}$ = $[5 + (-4)]_{补}$ = 0000 0101 + 1111 1100 = 1 0000 0001 = $[1]_{补}$；

$[4 - 5]_{补}$ = $[4 + (-5)]_{补}$ = 0000 0100 + 1111 1011 = 1111 1111 = $[-1]_{补}$；

$[-5 - 4]_{补}$ = $[-5 + (-4)]_{补}$ = 1111 1011 + 1111 1100 = 1 1111 0111 = $[-9]_{补}$；

2. 补码加减运算电路

利用带标志的加法器电路，可实现补码加减运算。



当控制端信号 Sub 为 0 时，做加法，Sub 控制多路选择器将 ${B}$ 输入加法器，实现 $A + B = [A] *补 + [B]* 补$。当控制端 Sub 为 1 时，做减法，Sub 控制多路选择器将 $\overline{B}$ 的非输入加法器，并将 Sub 作为低位进位送到加法器，实现 $A + \overline{B} + 1 = [A] *补 + [-B]* 补$。

无符号整数的二进制表示相当于正整数的补码表示，因此，该电路同时也能实现无符号整数的加/减运算。可通过标志信息对运算结果进行不同的解释。

- 零标志 ZF = 1 表示结果 F 为 0。不管对于无符号整数还是有符号整数运算，ZF 都有意义。
- 溢出标志 OF = 1 表示有符号数运算时发生溢出。对于无符号数运算，OF 没有意义。
- 符号标志 SF = 1 表示有符号数运算结果为负；有符号数运算结果为正时 SF = 0。对于无符号数运算，SF 没有意义。
- 进/借位标志 CF 表示无符号整数运算时的进/借位，判断是否发生溢出。做加法时，CF = 1 表示结果溢出，因此 CF 等于进位输出 C~out~。做减法时，CF = 1 表示有借位，即不够减，故 CF 等于进位输出 C~out~ 取反。综合可得 CF = Sub ⊕ C~out~。对于有符号数运算，CF 没有意义。

在之前的例子中，如果表示的是无符号数，那么：

$[5 + 4]_{补}$ = 0000 0101 + 0000 0100 = $[5]_{补}$ + $[4]_{补}$ = 0000 1001 = $[9]_{补}$；（加法不溢出）

$[5 - 4]_{补}$ = 0000 0101 + 1111 1100 = $[5]_{补}$ + $[252]_{补}$ = 1 0000 0001 = $[1]_{补}$；（加法溢出、减法不溢出）

$[4 - 5]_{补}$ = 0000 0100 + 1111 1011 = $[4]_{补}$ + $[251]_{补}$= 1111 1111 = $[255]_{补}$；（加法不溢出、减法溢出）

$[-5 - 4]_{补}$ = 1111 1011 + 1111 1100 = $[251]_{补}$ + $[252]_{补}$ = 1 1111 0111 = $[247]_{补}$；（加法溢出）

3. 溢出判别方法

溢出 是指运算结果超出了数的表示范围。通常，大于能表示的最大正数称为正上溢，小于能表示的最小负数称为负上溢。仅当两个符号相同的数相加，或两个符号相异的数相减才可能产生溢出。

在之前的例子中，如果假设机器字长为 4（n = 4），能表示的有符号数范围为 -8 ~ 7，那么就有：

$[5]_{补}$ = 0101，$[4]_{补}$ = 0100；

$[-5]_{补}$ = 1011，$[-4]_{补}$ = 1100；

$[5 + 4]_{补}$ = 0101 + 0100 = 1001 = $[-7]_{补}$；　　（正溢出）

$[5 - 4]_{补}$ = $[5 + (-4)]_{补}$ = 0101 + 1100 = 1 0001 = $[1]_{补}$；

$[4 - 5]_{补}$ = $[4 + (-5)]_{补}$ = 0100 + 1011 = 1111 = $[-1]_{补}$；

$[-5 - 4]_{补}$ = $[-5 + (-4)]_{补}$ = 1011 + 1100 = 1 0111 = $[7]_{补}$；　　（负溢出）

补码加减运算的溢出判断方法有以下 3 种：

（1）采用一位符号位。

由于减法运算在机器中是用加法器实现的，减法可以看作一个正数和一个负数的加法；因此无论是加法还是减法，只要参加操作的两个数符号相同，结果又与原操作数符号不同，就表示结果溢出。

比如上例中，一正一负相加必然不会溢出；两正数相加得到一个负数（符号位为1），则正溢出；两负数相加得到一个正数，则负溢出。

在实际应用中，为了节省时间，通常可以直接判断符号位产生进位 C~s~ 与最高数位的进位 C~1~ 。如果相同说明没有溢出，否则说明发生溢出。溢出标志 OF = C~s~ ⊕ C~1~。

（2）采用双符号位。

运算结果的两个符号位相同，表示未溢出；运算结果的两个符号位不同，表示溢出，此时最高位就代表真正的符号。也就是说，符号位 S~1~S~2~ = 00 表示结果为正数，无溢出； S~1~S~2~ = 11 表示结果为负数，无溢出。 S~1~S~2~= 01 表示结果正溢出； S~1~S~2~ = 10 表示结果负溢出。溢出标志 OF = S~1~ ⊕ S~2~。

比如上例中，如果采用双符号位，机器字长就应该扩展为 5，那么：

$[5]_{补}$ = 00 101，$[4]_{补}$ = 00 100；

$[-5]_{补}$ = 11 011，$[-4]_{补}$ = 11 100；

$[5 + 4]_{补}$ = 00 101 + 00 100 = 01 001 = $[1]_{补}$；　　（正溢出）

$[5 - 4]_{补}$ = $[5 + (-4)]_{补}$ = 00 101 + 11 100 = 1 00 001 = $[1]_{补}$；

$[4 - 5]_{补}$ = $[4 + (-5)]_{补}$ = 00 100 + 11 011 = 11 111 = $[-1]_{补}$；

$[-5 - 4]_{补}$ = $[-5 + (-4)]_{补}$ = 11 011 + 11 100 = 1 10 111 = $[-1]_{补}$；　　（负溢出）

2.2.4 定点数的乘法运算

乘除运算的原理难度较大，考查的概率也较低，做基本了解即可。

在计算机中，乘法运算由累加和右移操作实现。根据机器数的不同，可分为原码一位乘法和补码一位乘法。原码一位乘法的规则比补码一位乘法的规则简单。

1. 原码一位乘法

原码乘法运算的符号位与数值位分开计算。

- 确定乘积的符号位。由两个乘数的符号进行异或运算得到。
- 计算乘积的数值位。两个乘数的数值部分之积，可看作两个无符号数的乘积。

原码一位乘法的基本思路，就是类似竖式乘法的做法，让被乘数 x 分别乘以乘数 y 的每一位，然后再做叠加。不过竖式乘法需要做连加运算，这在电路实现上会有一些困难；改进的做法是，借鉴进制转换的“重复相乘/除法”，对每一位进行迭代计算。

回忆一下二进制数转换成十进制数的重复相乘/除法：

整数部分从高到低，将每一位乘以基数值、再加上后一位，进行“重复相乘”：

$(11011)_2 = (((1 \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 = 27$

小数部分从低到高，将每一位除以基数值、再加上前一位，进行“重复相除”：

$(0.101)_2 = ((1 \div 2 + 0) \div 2 + 1) \div 2 + 0 = 0.625$

所以，两数相乘时，就可以把乘数 y 用这种方式按每一位拆开，并乘以 x 、再逐位叠加就可以了。由于每次乘以 2 就相当于左移一位、除以 2 就相当于右移一位，因此只需要反复迭代这样的 **移位** 和 **加法** 运算就可以很容易地实现乘法了。

以纯小数为例，已知 [x]~原~ = x~0~ . x~1~x~2~...x~n~，[y]~原~ = y~0~ . y~1~y~2~...y~n~，那么

$$
\begin{aligned}
[x]_原 \cdot [y]_原 &= [x]_原 \cdot (y_0 . y_1 y_2 ... y_n) \\
&= [x]_原 \cdot (y_n \cdot 2^{-n} + y_{n-1} \cdot 2^{-(n-1)} + \ldots + y_1 \cdot 2^{-1} + y_0 \cdot 2^0) \\
&= (((([x]_原 \cdot y_n \cdot 2^{-1} + [x]_原 \cdot y_{n-1}) \cdot 2^{-1} + \cdots) \cdot 2^{-1} + [x]_原 \cdot y_1) \cdot 2^{-1} + [x]_原 \cdot y_0
\end{aligned}
$$

原码一位乘法的运算规则如下：

- 被乘数和乘数均取绝对值|x| 和 |y|参加运算，看作无符号数，符号位为 x~0~ ⊕ y~0~。
- 乘数的每一位 y~i~ 乘以被乘数 |x| 得到 |x| · y~i~，将该结果与前面所得的结果相加，作为部分积；初始值为 0。
- 从乘数的最低位 y~n~ 开始判断：若 y~n~ = 1，则部分积加上被乘数 |x|，然后右移一位；若 y~n~ = 0，则部分积加上 0，然后右移一位。
- 重复上一步骤，判断 n 次。

由于参与运算的是两个数的绝对值，因此运算过程中的右移操作均为逻辑右移。

例如，当 x = 0.1101 = (0.8125)~10~，y = 0.1011 = (0.6875)~10~ 时，计算 x · y。

部分积	当前乘数	操作描述
0.0000 + 0.1101	1011	初始状态，部分积为 0； 乘数最低位为 1，需要叠加被乘数
= 0.1101 → 0.0110 + 0.1101	1101	得到的和右移一位，得到新的部分积； 乘数也同时右移一位，部分积移出的 1 填到乘数高位； 乘数最低位为 1，需要叠加被乘数
= 1.0011 → 0.1001 + 0.0000	1110	得到的和右移一位，得到新的部分积； 乘数也同时右移一位，部分积移出的 1 填到乘数高位； 乘数最低位为 0，直接加 0
= 0.1001 → 0.0100 + 0.1101	1111	得到的和右移一位，得到新的部分积； 乘数也同时右移一位，部分积移出的 1 填到乘数高位； 乘数最低位为 1，需要叠加被乘数
= 1.0001 → 0.1000	1111	得到的和右移一位； 乘数所有位都已移出，得到最终结果

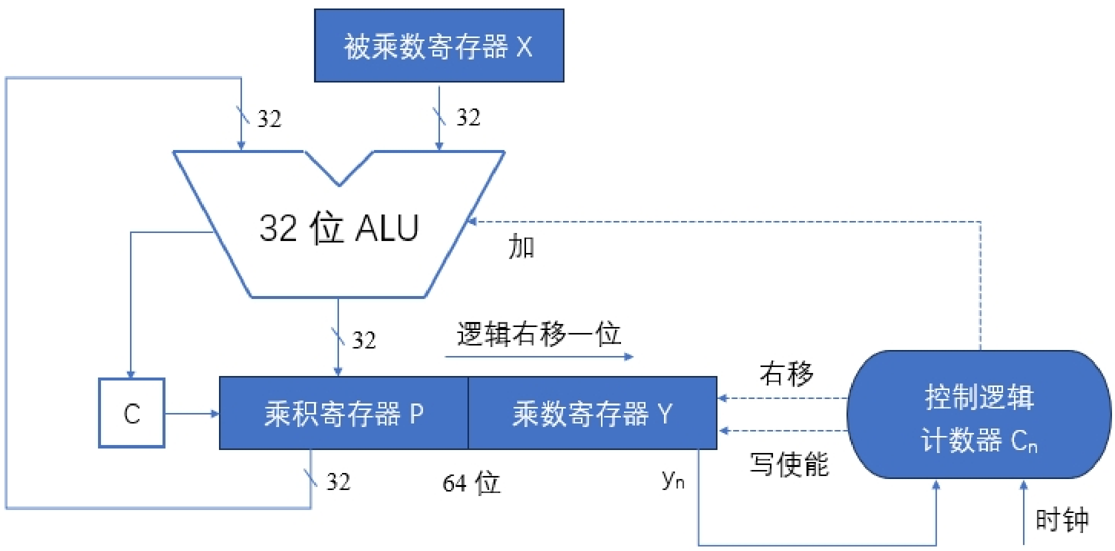
最终的乘积，高位在“部分积”中，低位在“当前乘数”中，所以得到：

x · y = 0.1101 × 0.1011 = 0.10001111 = (0.55859375)~10~

---

2. 无符号数乘法运算电路

下面是实现两个 32 位无符号数乘法运算的逻辑结构图。



X、Y 均为 32 位寄存器，开始时分别用来存放被乘数 x 、乘数 y ； P 也是 32 位寄存器，用来存放乘积（部分积），初始值置 0。部分积和被乘数做无符号数加法时，可能产生进位，因此还需要一个专门的进位位 C。 C~n~ 为计数器，初始值为 32，此后每进行一轮运算就减 1。

每一轮运算都由寄存器 Y 的最低位 y~n~ 来控制具体逻辑：如果 y~n~ = 1，那么就让 ALU 对乘积寄存器 P 和被乘数寄存器 X 的内容做“无符号加法”运算，运算结果送回寄存器 P，进位存放在 C 中；如果 y~n~ = 0 则不做相加。无论是否做了加法，每一轮运算都会对进位位 C、乘积寄存器 P 和乘数寄存器 Y 实现同步的“逻辑右移”；此时，进位位 C 移入寄存器 P 的最高位，寄存器 Y 的最低位 y~n~ 被移出。每次从寄存器 Y 移出的最低位 y~n~ 都被送到控制逻辑，以决定被乘数是否被加到部分积上。

最终，当原始乘数所有位全部被移出时，寄存器 P 和 Y 就分别存放了乘积的高 32 位和低 32 位。在 CPU 的运算器中，就分别对应了累加器 ACC 和乘商寄存器 MQ。

3. 补码一位乘法

原码一位乘法容易理解，规则也比较简单；缺点是符号位不参与计算，需要单独判断。更重要的是，在计算机中为了方便做加减计算，数据一般是用补码来表示的；这就使得如果我们采用原码乘法，需要计算前先把补码转换成原码，计算结束后再把结果的原码转换为补码，增加了额外的运算。

补码一位乘法 是一种带符号数的乘法，采用相加/相减的校正操作，直接计算补码数据的乘积。

补码乘法是直接对补码进行的。对于纯整数，补码表达为：

$$[x]_{补} = \begin{cases} x & 0 \le x < 2^n \\ 2^{n+1} + x = 2^{n+1} - |x| & -2^n \le x < 0 \end{cases} \quad (\mathrm{mod}\ 2^{n+1})$$

而类似的，纯小数补码定义为：

$$[x]_{补} = \begin{cases} x & 0 \le x < 1 \\ 2 + x & -1 \le x < 0 \end{cases} \quad (\mathrm{mod}\ 2)$$

所以，当取不同的正负符号时，补码表达会有所不同，继而影响到逐位相乘叠加的效果。

已知 [x]~补~ = x~0~ . x~1~x~2~...x~n~，[y]~补~ = y~0~ . y~1~y~2~...y~n~，那么需要分不同的情况讨论：

① 被乘数 x 和乘数 y 符号均为正，即 x~0~ = y~0~ = 0 时，

[x]~补~ = x，[y]~补~ = y，所以就有：

$$[x]_{补} \cdot [y]_{补} = x \cdot y = [x \cdot y]_{补}$$

类似原码一位乘法，利用移位和加法的叠加，就可以计算出补码的乘积；这也就是最终计算结果的补码。

② 被乘数 x 为正，乘数 y 为负，即 x~0~ = 0，y~0~ = 1 时，

[y]~补~ = 1. y~1~y~2~...y~n~ = 2 + y，所以：

$$\begin{aligned} y &= [y]_{补} - 2 \\ &= 1.\ y_1 y_2 \ldots y_n - 2 \\ &= 0.\ y_1 y_2 \ldots y_n - 1 \end{aligned}$$

那么两数的乘积就可以写成：

$$\begin{aligned} x \cdot y &= x \cdot (0.\ y_1 y_2 \ldots y_n - 1) \\ &= x \cdot (0.\ y_1 y_2 \ldots y_n) - x \end{aligned}$$

这样一个计算结果，它的补码表示为：

$$[x \cdot y]_{补} = [x \cdot (0.\ y_1\, y_2 ... y_n\ ) - x\ ]_{补}$$
$$= [x\ ]_{补} \cdot (0.\ y_1\, y_2 ... y_n\ ) + [-x\ ]_{补}$$

可以看到，当乘数为负时，可以把乘数补码 [y]~补~ 直接去掉符号位，当成一个正数与 [x]~补~ 相乘；得到的结果再加上 [-x]~补~ 进行校正。所以这种方法也叫做“校正法”。

例如，当 x = 0.1101 = (0.8125)~10~，y = 1.1011 = (-0.3125)~10~ 时，计算 x · y。

我们可以直接计算 0.1101 × 0.1011 = 0.10001111，再加上 [-x]~补~ = 1. 0011，得到：

0.10001111 + 1. 0011 = 1.10111111 = （ -0.25390625)~10~

---

③ 被乘数 x 为负，乘数 y 为正，即 x~0~ = 1，y~0~ = 0 时，

我们可以交换被乘数和乘数，直接按情况②来处理；也可以仔细分析，发现乘数 y 为正数，可以写成

[y]~补~ = 0. y~1~y~2~...y~n~ 的形式，同样可以借鉴情况②中的分析和原码一位乘的方法。当两数的补码相乘时：

$$
\begin{aligned}
[\,x\,]_{补} \cdot [\,y\,]_{补} &= [\,x\,]_{补} \cdot (0.y_1 y_2 \,...\, y_n) \\
&= [\,x\,]_{补} \cdot (y_n \cdot 2^{-n} + y_{n-1} \cdot 2^{-(n-1)} + \ ... + y_1 \cdot 2^{-1}) \\
&= (((([\,x\,]_{补} \cdot y_n \cdot 2^{-1} + [\,x\,]_{补} \cdot y_{n-1}) \cdot 2^{-1} + \cdots) \cdot 2^{-1} + [\,x\,]_{补} \cdot y_1) \cdot 2^{-1} \\
&\to (((x \cdot y_n \cdot 2^{-1} + x \cdot y_{n-1}) \cdot 2^{-1} + \cdots) \cdot 2^{-1} + x \cdot y_1) \cdot 2^{-1} \\
&\to [\,x \cdot y\,]_{补}
\end{aligned}
$$

观察可以发现，与原码一位乘完全类似，补码相乘也可以将乘数展开，逐位进行相乘、右移和叠加。不过需要注意的是，这时由于被乘数 x 是负数，右移时就需要在左侧高位补 1，也就是做算术右移、而不是逻辑右移。

这样一来，算术右移就实现了对真值 x 的“除以 2”操作，最终叠加之后的结果，就是 x · y 的补码了。

例如，当 x = 1.1 = (-0.5)~10~，y = 0.011 = (0.375)~10~ 时，计算 x · y。

部分积	当前乘数	操作描述
0 . 0 + 1 . 1	0 1 <u>1</u>	初始状态，部分积为 0； 乘数最低位为 1，需要叠加被乘数
= 1 . 1 → 1 . 1 + 1 . 1	1 0 <u>1</u>	得到的和算术右移一位，得到新的部分积，高位补 1； 乘数也同时右移一位，部分积移出的 1 填到乘数高位； 乘数最低位为 1，需要叠加被乘数
= 1 1 . 0 → 1 . 1 + 0 . 0	0 1 <u>0</u>	得到的和算术右移一位，得到新的部分积； 乘数也同时右移一位，部分积移出的 0 填到乘数高位； 乘数最低位为 0，直接加 0
= 1 . 1 → 1 . 1	1 0 1	得到的和算术右移一位； 乘数所有位都已移出，得到最终结果

最终的乘积，高位在“部分积”中，低位在“当前乘数”中，所以得到：

x · y = 1.1 × 0.011 = 1.1101 = (- 0.1875)~10~

④ 被乘数 x 和乘数 y 符号均为负，即 x~0~ = y~0~ = 1 时，

通过情况②和③的分析可以看出，当乘数 y 为正时，可以直接按照原码一位乘的方式进行补码乘法，注意需要进行算术右移；而当乘数 y 为负时，则可以先不考虑 y 的符号位，同样按照原码一位乘进行补码乘法，最后的结果要再加上 [-x]~补~ 进行校正。

例如，当 x = 1.1 = (-0.5)~10~，y = 1.011 = (-0.625)~10~ 时，计算 x · y。

我们可以直接计算 1.1 × 0.011 = 1.1101，再加上 [-x]~补~ = 0.1，得到：

1.1101 + 0.1 = 1 0.0101 = （0.3125)~10~

可以看出，如果使用双符号位来表示正负，会更加方便。

⑤ **Booth算法**

以上的 4 种情况需要分别讨论，根据乘数的符号来决定是否需要进行校正，这就导致校正法的逻辑控制电路比较复杂。

如果不考虑操作数的符号，直接用统一的规则来处理所有情况，可以采用 **比较法**。这种方式是 Booth 夫妇首先提出的，所以又叫 **Booth 算法**。

当被乘数 x 和乘数 y 符号任意时，按照之前讨论的校正法规则，可以写出一个统一的计算公式：

$$[x \cdot y]_{补} = [x]_{补} \cdot (0.y_1 y_2 ... y_n) + [-x]_{补} \cdot y_0$$

容易推出，对于纯小数，在 mod 2 的前提下，[-x]~补~ = - [x]~补~，所以可以进一步推导得到：

$$\begin{aligned}
[x \cdot y]_{补} &= [x]_{补} \cdot (0.y_1 y_2 ... y_n) - [x]_{补} \cdot y_0 \\
&= [x]_{补} \cdot \left(y_n \cdot 2^{-n} + y_{n-1} \cdot 2^{-(n-1)} + ... + y_1 \cdot 2^{-1} - y_0\right) \\
&= [x]_{补} \cdot (-y_0 + y_1 \cdot 2^{-1} + y_2 \cdot 2^{-2} + \cdots + y_{n-1} \cdot 2^{n-1} + y_n \cdot 2^{-n}) \\
&= [x]_{补} \cdot [-y_0 + (y_1 - y_1 \cdot 2^{-1}) + (y_2 \cdot 2^{-1} - y_2 \cdot 2^{-2}) + \cdots + (y_n \cdot 2^{-(n-1)} - y_n \cdot 2^{-n})] \\
&= [x]_{补} \cdot [(y_1 - y_0) + (y_2 - y_1) \cdot 2^{-1} + \cdots + (y_n - y_{n-1}) \cdot 2^{-(n-1)} + (0 - y_n) \cdot 2^{-n}]
\end{aligned}$$

令 y~n+1~ = 0，那么就可以得到一个通项系数： d~i~ = y~i+1~ - y~i~ ，上式可以进一步化简为：

$$\begin{aligned}
[x \cdot y]_{补} &= [x]_{补} \cdot \left(d_0 + d_1 \cdot 2^{-1} + \cdots + d_{n-1} \cdot 2^{-(n-1)} + d_n \cdot 2^{-n}\right) \\
&= \left(\left(\left([x]_{补} \cdot d_n \cdot 2^{-1} + [x]_{补} \cdot d_{n-1}\right) \cdot 2^{-1} + \cdots\right) \cdot 2^{-1} + [x]_{补} \cdot d_1\right) \cdot 2^{-1} + [x]_{补} \cdot d_0
\end{aligned}$$

这样一来，补码乘法的计算方式就跟原码一位乘完全一样了，只是被乘数每次乘的不再是乘数 y 的每一位 y~i~，而是变成了 d~i~ = y~i+1~ - y~i~ 。这样就有 1、-1 和 0 三种情况，每一次计算都由 d~i~ 来决定部分积叠加的是 [x]~补~、[-x]~补~ 还是 0；然后再做一位算术右移得到新的部分积。最后一步，需要由 d~0~ = y~1~ - y~0~ 决定是否有叠加项，但不再做位移。

Booth 算法的移位规则如下表所示：

y_i	y_{i+1}	$y_{i+1} - y_i$	操　　作
0	0	0	部分积直接右移一位
0	1	1	部分积加上[x]$_{补}$，再右移一位
1	0	-1	部分积加上[-x]$_{补}$，再右移一位
1	1	0	部分积直接右移一位

Booth 算法的具体运算规则如下：

① 符号位参与运算，运算的数均以补码表示。

② 被乘数一般取 **双符号位** 参与运算，部分积取 **双符号位**，初值为 0，乘数取单符号位。

③ 乘数末尾增加一个“附加位” y~n+l~，初始值为 0。

④ 根据（y~i~，y~i+1~）的取值来确定操作，如上表所示。

⑤ 移位按补码右移规则（算术右移）进行。

⑥ 按照上述算法进行 n + 1 步操作，但第 n + 1 步不再移位，仅根据 y~0~ （符号位）与 y~1~ （第一位数值位）的比较结果做相应的叠加运算。所以总共需要进行 n + 1 次累加和 n 次右移。

例如，当 x = 1.1101 = (-0.1875)~10~，y = 1.1011 = (-0.3125)~10~ 时，计算 x · y。

首先得到 [x]~补~ = 11.1101，[-x]~补~ = 00.0011。具体计算步骤如下：

部分积	当前乘数（y_i）	附加位 y_{i+1}	操作描述
00.0000 + 00.0011	11011	0	初始状态，部分积为 0; (y_i, y_{i+1}) = 10，部分积叠加 [-x]$_{补}$
= 00.0011 → 00.0001 00.0001 → 00.0000 + 11.1101	11011 → 11101 11101 → 11110	 1 1	得到的和逻辑右移一位，得到新的部分积; 乘数也同时右移一位，之前的 y_i 被移出，成为附加位 y_{i+1}; (y_i, y_{i+1}) = 11，部分积叠加 0，继续右移; (y_i, y_{i+1}) = 01，部分积叠加 [x]$_{补}$
= 11.1101 → 11.1110 + 00.0011	11110 → 11111	 0	得到的和逻辑右移一位，得到新的部分积; 乘数也同时右移一位，之前的 y_i 被移出，成为附加位 y_{i+1}; (y_i, y_{i+1}) = 10，部分积叠加 [-x]$_{补}$
= 00.0001 → 00.0000 + 00.0000	11111 → 11111	 1	得到的和逻辑右移一位，得到新的部分积; 乘数也同时右移一位，之前的 y_i 被移出，成为附加位 y_{i+1}; 此时已经移至符号位，(y_0, y_1) = 11，部分积叠加 0
= 00.0000	1111		最后一步不再右移，得到最终结果

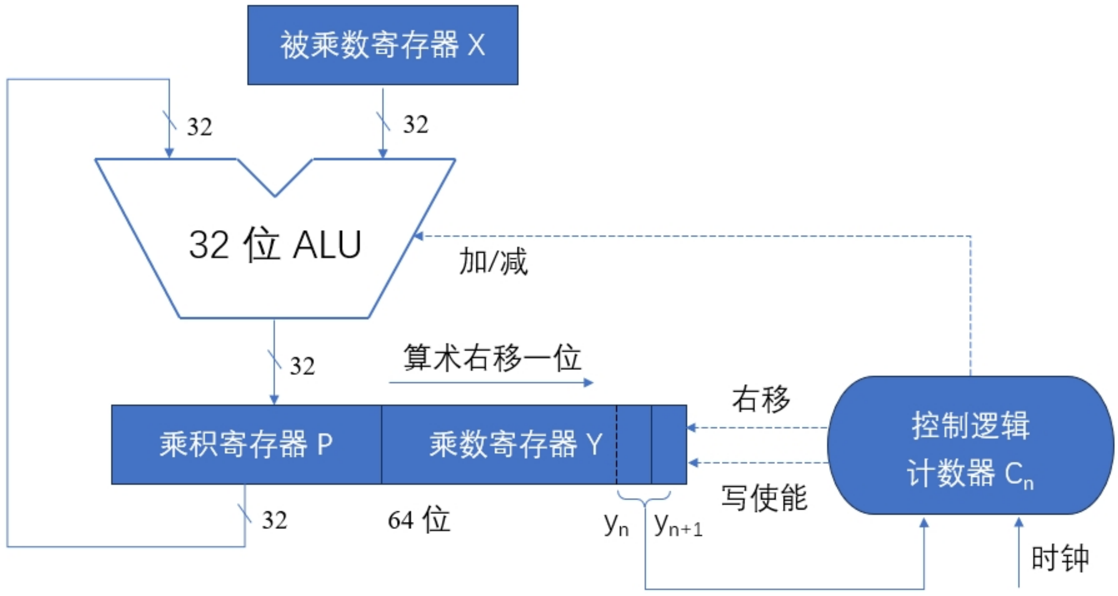
同样，最终的乘积，高位在“部分积”中，低位在“当前乘数”中，所以得到：

x · y = 1.1101 × 1.1011 = 0.00001111 = (0.05859375)~10~

---

4. 补码乘法运算电路

对于有符号数的乘法运算，可以采用补码一位乘。与原码一位乘类似，我们可以用下面的逻辑结构图来实现补码一位乘法的运算电路：



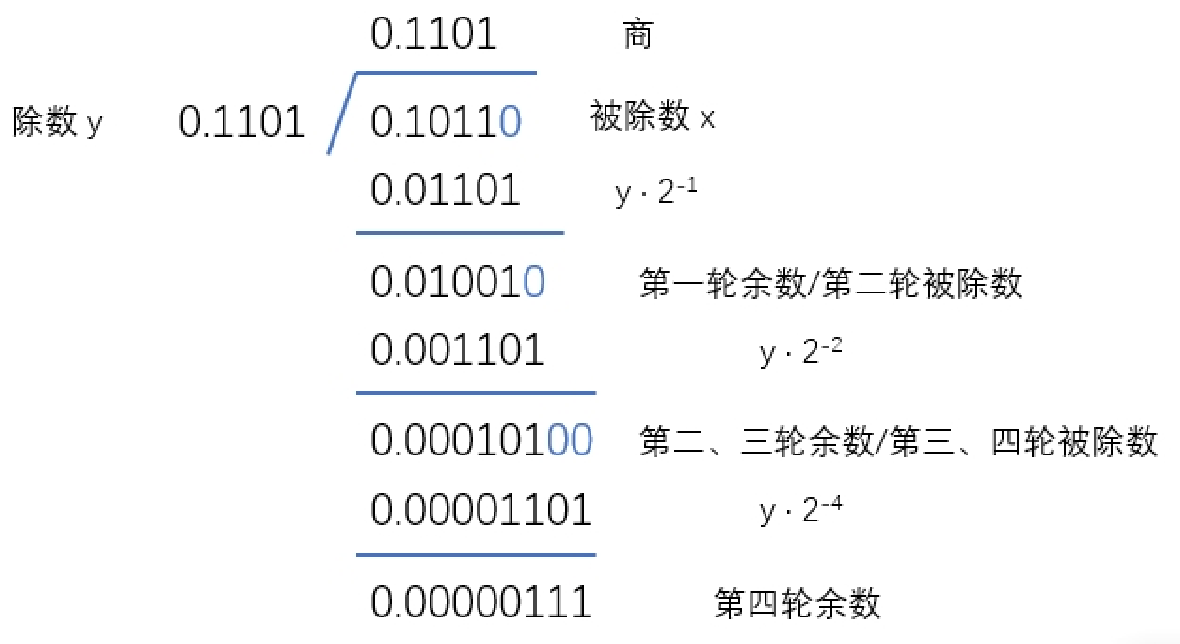
因为是带符号数运算，所以不需要专门的进位位。由于采用 Booth 算法，还需要一个额外的“附加位” y~n+1~。每轮运算，乘积寄存器 P 和乘数寄存器 Y 实现同步的一位“算术右移”，每次从寄存器 Y 移出的最低位成为新的附加位 y~n+1~，它的前一位则成为 y~n~，它们共同决定叠加项是 [x]~补~、[-x]~补~ 还是 0。另外由于 ALU 可以对加减法统一操作，所以电路种只需要选择加/减一项 [x]~补~ 就可以了。

2.2.5 定点数的除法运算

除法是乘法的逆运算，但不像加减法那样可以直接整合。我们可以先从除法的竖式笔算入手，分析一下除法的具体步骤。

例如，当 x = (-0.1011)~2~ = (-0.6875)~10~，y = (0.1101)~2~ = (0.8125)~10~ 时，计算 x / y。

首先可以看出，商的符号为负，余数的符号为负；其次利用竖式计算绝对值的商：



所以可以得到， x / y = - 0.1101（商）... - 0.00000111（余数） = (- 0.8125 ... - 0.02734375)~10~

如果完全按照竖式除法的规则，需要心算上商，本质就是由当前被除数减去除数乘以当前权值（第 i 轮就乘以 2^-i^），够减就上商 1，不够就上商 0 。得到的余数补 0 后再作为下一轮的被除数进行计算。这个过程中，每轮除数要乘以权值 2^-i^，相当于右移 i 位，得到的余数左侧全部是 0；如果被除数 x、除数 y 和商都是 4 位，就需要 8 位数据来保存余数，这就显得有些麻烦。

更加简单的做法是，每轮相减除数不变，把余数左移；这样效果一样，而电路实现会更加简单。当然，代价就是得到的余数是经过左移之后的；n 轮计算完成之后，需要再右移 n 位，也就是乘以 2^-n^ 才是真正的余数。每轮的相减，也可以转换成负数补码的加法。

所以跟乘法类似，除法运算在计算机中，是转换成逐位的“累加-左移”操作来实现的，可以分为原码除法和补码除法。

1. 原码一位除法

原码一位除法和原码一位乘法一样，特点是符号位单独处理。商符由两个操作数的符号位做“异或”形成，减法操作用补码加法实现。

同样以小数为例，已知 [x]~原~ = x~0~ . x~1~x~2~...x~n~，[y]~原~ = y~0~ . y~1~y~2~...y~n~，那么

$$\left[\frac{x}{y}\right]_{原} = (x_0 \oplus y_0).\left(\frac{0.x_1x_2\ldots x_n}{0.y_1y_2\ldots y_n}\right) = (x_0 \oplus y_0).\frac{x^*}{y^*}$$

其中，0 . x~1~x~2~...x~n~ 就是 x 的绝对值，记作 x^ ^*；0 . y~1~y~2~...y~n~ 是 y 的绝对值，记作 y^ ^* 。商符由两数符号位异或得到，商值由两数绝对值相除得到。

小数定点除法对被除数和除数有一定的约束条件：

0 <| 被除数 |≤| 除数 |

实现除法运算时，被除数应不大于除数，并且应该避免被除数和除数为 0。商的位数一般和操作数位数相同。

原码除法中，每轮计算需要用被除数绝对值减去除数绝对值：x^ ^ - *y^ ^*。计算机中减法可以用负数补码的加法实现，所以最终的操作就是：[x^ *^]~补~ + [- y^ ^*]~补~ 。如果结果为正，说明够减，上商 1，结果作为余数直接左移，并作为下一轮被除数；如果结果为负，说明不够减，上商 0，这时的结果并不是真实的余数。

根据对余数的处理方式不同，又可以分为 **恢复余数法** 和 **不恢复余数法（加减交替法）** 两种。

（1）恢复余数法

恢复余数法的特点是：当余数为负时，需要加上除数的绝对值，将其恢复成原本的余数。

由于每次得到的是商的高位，所以每轮计算可以将余数和商同时左移一位；余数加上 [- y^ *^]~补~，判断正负来决定下一位商是 1 还是 0；如果为负，还需要先加上 [y^* ^]~补~ 恢复余数，然后再做左移。

例如，当 x = (-0.1011)~2~ = (-0.6875)~10~，y = (-0.1101)~2~ = (-0.8125)~10~ 时，计算 x / y。

首先看出，商的符号为正，余数的符号为负。并且得到：

x^ ^ *= 0.1011，y^ ^* = 0.1101，[y^ *^]~补~ = 0.1101，[-y^* ^]~补~ = 1.0011

具体计算过程如下：

被除数/余数	商	操作描述
0.1011 + 1.0011	_._ _ _ _	初始值为被除数的绝对值 x^* 减去除数绝对值 y^*，即 + $[-y^*]_{补}$
= 1.1110 + 0.1101	0	余数为负，上商 0 恢复余数，即 + $[y^*]_{补}$
= 0.1011 ← 1.0110 + 1.0011	 0 _	恢复的余数，就是初始的被除数 左移一位，余数和商同时左移 减去除数绝对值 y^*，即 + $[-y^*]_{补}$
= 0.1001 ← 1.0010 + 1.0011	01 01 _	余数为正，上商 1 左移一位，余数和商同时左移 减去除数绝对值 y^*，即 + $[-y^*]_{补}$
= 0.0101 ← 0.1010 + 1.0011	011 011 _	余数为正，上商 1 左移一位，余数和商同时左移 减去除数绝对值 y^*，即 + $[-y^*]_{补}$
= 1.1101 + 0.1101	0110	余数为负，上商 0 恢复余数，即 + $[y^*]_{补}$
= 0.1010 ← 1.0100 + 1.0011	 0110 _	恢复的余数 左移一位，余数和商同时左移 减去除数绝对值 y^*，即 + $[-y^*]_{补}$
= 0.0111	01101	余数为正，上商 1

所以商值为 x^ ^ / *y^ ^* = 0.1101；而余数由于经过了 4 次左移，所以最终还应该做 4 次右移才是真正的余数：0.0111 * 2^-4^ = 0.00000111，另外还要注意余数符号为负，所以最终结果为：

x / y = 0.1101（商）... - 0.00000111（余数）

由上例也可以发现，如果最终商保留 4 位，那么我们需要做 5 次上商，第一次上商其实是商的整数位；由于我们要求 |被除数| ≤ |除数|，因此正常情况下第一位商总是 0。所以对于小数除法而言，可以用它来做溢出判断：当该位为 1 时，表示当前除法溢出，不能进行；当该位为 0 时，当前除法合法，可以进行。

（2）不恢复余数法（加减交替法）

在恢复余数法中，每当余数为负时都需要恢复余数，这就增加了运算量，操作也不规则，电路实现会比较复杂。加减交替法就克服了这一缺点。

加减交替法 又称 **不恢复余数法**，是对恢复余数法的一种改进。

通过分析恢复余数法可以发现，如果把第 i 轮计算的余数记作 R~i~，那么：

- 如果 R~i~ > 0，就上商 1，接下来需要将余数 R~i~ 左移一位，再减去除数绝对值 y^ ^，*即 2R~i~ - y^ ^*；
- 如果 R~i~ < 0，就上商 0，接下来先加上 y^ ^ *恢复余数，再做左移和减法，即 2 (R~i~ + y^* ^) - y^ ^ *= 2R~i~ + y^* ^ 。

这样一来，就不需要额外恢复余数了，每轮计算的规则完全统一起来，只是左移之后再加/减 y^*^ 就可以了；所以把这种方法叫做“加减交替法”，或者“不恢复余数法”。

还是上面的例子，当 x = (-0.1011)~2~ ，y = (-0.1101)~2~ 时，计算 x / y。

同样的步骤，首先看出，商的符号为正，余数的符号为负。并且得到：

x^ ^ *= 0.1011，y^* ^ = 0.1101，[y^ *^]~补~ = 0.1101，[-y^* ^]~补~ = 1.0011

具体计算过程如下：

被除数/余数	商	操作描述
0.1011 + 1.0011	_._ _ _ _	初始值为被除数的绝对值 x^* 减去除数绝对值 y^*，即 + $[-y^*]_补$
= 1.1110 ← 1.1100 + 0.1101	0 0 _	余数为负，上商 0 左移一位，余数和商同时左移 由于余数为负，需要加上除数绝对值 y^*，即 + $[y^*]_补$
= 0.1001 ← 1.0010 + 1.0011	0 1 0 1 _	余数为正，上商 1 左移一位，余数和商同时左移 余数为正，减去除数绝对值 y^*，即 + $[-y^*]_补$
= 0.0101 ← 0.1010 + 1.0011	0 1 1 0 1 1 _	余数为正，上商 1 左移一位，余数和商同时左移 余数为正，减去除数绝对值 y^*，即 + $[-y^*]_补$
= 1.1101 ← 1.1010 + 0.1101	0 1 1 0 0 1 1 0 _	余数为负，上商 0 左移一位，余数和商同时左移 余数为负，加上除数绝对值 y^*，即 + $[y^*]_补$
= 0.0111	0 1 1 0 1	余数为正，上商 1

所以商值为 x^ ^ */ y^* ^ = 0.1101；而余数由于经过了 4 次左移，所以最终还应该做 4 次右移才是真正的余数：0.0111 * 2^-4^ = 0.00000111，另外还要注意余数符号为负，所以最终结果为：

x / y = 0.1101（商）... - 0.00000111（余数）

---

2. 补码一位除法（加减交替法）

与乘法类似，我们同样可以直接使用补码来实现除法操作。补码除法也可以分为恢复余数法和加减交替法，由于加减交替法是改进版本，用得较多，所以我们只讨论加减交替法。

补码一位除法的特点是符号位与数值位一起参加运算，商符自然形成；所有操作数和结果都用补码表示。

二进制除法运算的核心，是“求余数”，本质上就是被除数和除数绝对值的相减；而补码除法是不区分正负统一进行计算的，因此我们需要讨论在不同的正负情况下，如何实现被除数和除数绝对值的相减操作。

被除数 x	除数 y	求余数 $R_i = \|x\| - \|y\|$	余数符号	是否够减 $\|x\| \geq \|y\|$	上商	求新余数 R_{i+1}
+	+	x - y	+	√	1	$2[R_i]_{补} + [-y]_{补}$
			-	×	0	$2[R_i]_{补} + [y]_{补}$
+	-	x + y	+	√	0	$2[R_i]_{补} + [y]_{补}$
			-	×	1	$2[R_i]_{补} + [-y]_{补}$
-	+	x + y	+	×	1	$2[R_i]_{补} + [-y]_{补}$
			-	√	0	$2[R_i]_{补} + [y]_{补}$
-	-	x - y	+	×	0	$2[R_i]_{补} + [y]_{补}$
			-	√	1	$2[R_i]_{补} + [-y]_{补}$

被除数和除数的绝对值相减，得到了余数，接下来就可以根据是否“够减”（即 |x| ≥ |y|）来上商。

需要注意的是，由于商也需要用补码表示，因此当商为正时，够减时上 1，不够减时上 0；而当商为负时需要取反码，够减时上 0，不够减时上 1。补码是原码“取反加1”得到的，所以负商的最后一位还应该再加1，这比较麻烦；如果对精度没有特殊要求，我们一般可以约定商“末位恒置 1”，这种方法操作简单，最大误差为 2^-n^。

对于新余数 R~i+1~ 的计算，可以借鉴原码除法的加减交替法，根据当前余数 R~i~ 和除数 y 的符号，决定加/减 y 的绝对值，这最终可以用加上 [y]~补~ 或者 [-y]~补~ 来实现。事实上，将上一轮的余数 R~i~ 左移之后就得到了下一轮的被除数，再根据被除数、除数的符号关系就可以得到 R~i+1~ 的表达了。

上面表格中，对商值的判断有点繁琐，可以化简如下：

x 和 y 符号	商符	求余数 R_i（绝对值的差）	R_i 和 y 符号	商值	求新余数 R_{i+1}
同号	+	x – y	同号，够减	1	$2[R_i]_{补} + [-y]_{补}$
			异号，不够减	0	$2[R_i]_{补} + [y]_{补}$
异号	-	x + y	同号，不够减	1	$2[R_i]_{补} + [-y]_{补}$
			异号，够减	0	$2[R_i]_{补} + [y]_{补}$

在补码除法中，商符可以在求商的过程中自动生成。

由于要求 |x| ≤ |y|，因此当做第一轮计算时必须”不够减“。如果 x 和 y 同号，那么它们都为正时 R~i~ 应该为负、都为负时 R~i~ 应该为正，即 R~i~ 和 y 必须异号；此时商值为 0，而商符也恰恰要求是正，说明除法运算是合法的。如果 R~i~ 和 y 同号，则说明够减，商值为1，此时发生了溢出。同样，当 x 和 y 异号时，也可以发现商值为 1 时表示”不够减“，是正常情况，这时商符也要求是负的，完全一致。

所以，补码除法中商的符号也可以用来判断溢出。

由于商符自动生成，我们可以把上表进一步化简：

R_i 和 y 符号	商值	求新余数 R_{i+1}
同号	1	$2\,[R_i]_补 + [-y]_补$
异号	0	$2\,[R_i]_补 + [y]_补$

可以总结补码一位除法的运算规则如下：

- 符号位参加运算，除数与被除数均用补码表示，商和余数也用补码表示。
- 如被除数与除数同号，则被除数减去除数；如被除数与除数异号，则被除数加上除数。
- 余数与除数同号，商上1，余数左移一位再减去除数；余数与除数异号， 商上 0，余数左移一位再加上除数。
- 重复执行上一步操作，操作 n 次。
- 如果对商的精度没有特殊要求，一般采用“末位恒置 1”法。

同样的例子，当 x = (-0.1011)~2~ ，y = (-0.1101)~2~ 时，计算 x / y。

首先得到： [x]~补~ = 1.0101，[y]~补~ = 1.0011，[-y]~补~ = 0.1101

具体计算过程如下：

被除数/余数	商	操作描述
1.0101 + 0.1101	_._ _ _ _	初始值为被除数的补码 $[x]_补$ $x_0 = y_0 = 1$，x 和 y 同号，所以 $x - y = [x]_补 + [-y]_补$
= 0.0010 ← 0.0100 + 1.0011	0 0 _	余数为正，与 y 异号，上商 0 左移一位，余数和商同时左移 由于余数和 y 异号，新余数需要加上 y，即 + $[y]_补$
= 1.0111 ← 0.1110 + 0.1101	0 1 0 1 _	余数为负，与 y 同号，上商 1 左移一位，余数和商同时左移 由于余数和 y 同号，新余数需要减去 y，即 + $[-y]_补$
= 1.1011 ← 1.0110 + 0.1101	0 1 1 0 1 1 _	余数为负，与 y 同号，上商 1 左移一位，余数和商同时左移 由于余数和 y 同号，新余数需要减去 y，即 + $[-y]_补$
= 0.0011 ← 0.0110	0 1 1 0 0 1 1 0 _ 0 1 1 0 1	余数为正，与 y 异号，上商 0 左移一位，余数和商同时左移 末尾商恒置 1

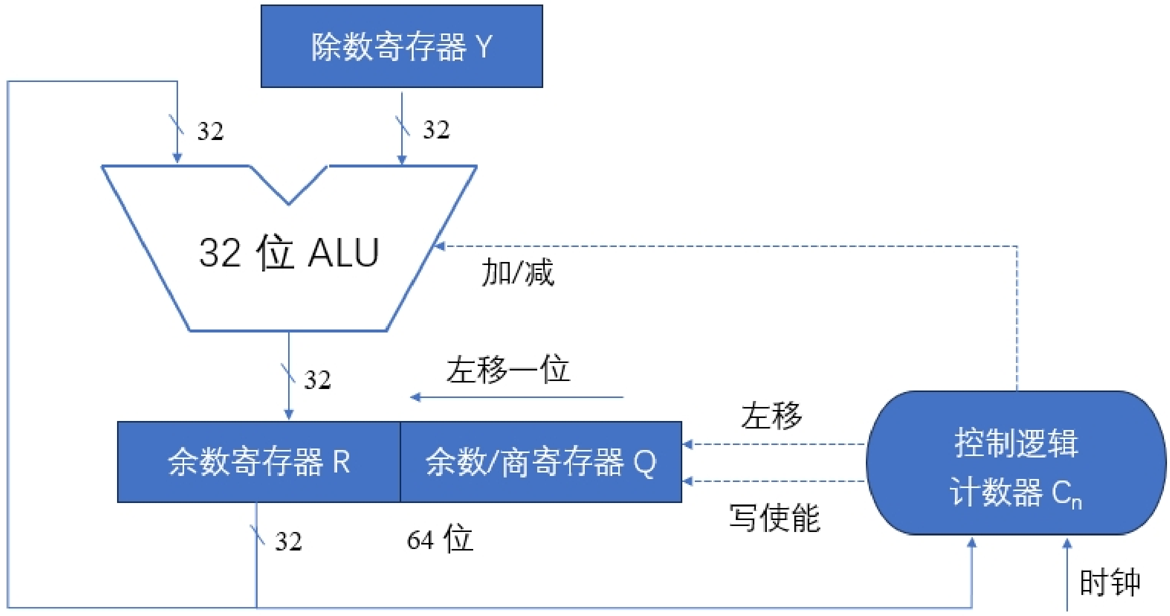
所以商值为

x / y = 0.1101

---

3. 除法运算电路

n 位定点数的除法运算，可以看作用一个 2n 位的被除数 x，去除以一个 n 位的除数 y。这就需要对被除数进行扩展。下图是一个 32 位除法运算的逻辑结构图。



由于被除数 x 是 64 位，所以需要两个 32 位的寄存器 R 和 Q 来存放。初始时，寄存器 R 存放扩展被除数的高位部分，寄存器 Q 存放扩展被除数的低位部分。除数 y 则放在除数寄存器 Y 中。

跟乘法运算电路类似，ALU 也是除法器的核心部件，每轮计算中都要对余数寄存器 R 和除数寄存器 Y 的内容做加/减运算，运算结果送回寄存器 R。开始时，首先取 R 中的数据，也就是 x 的高 32 位，根据符号关系与 y 进行加/减运算，得到的余数写回 R 中；之后每轮计算，寄存器 R 和 Q 实现同步左移，左移时，Q 的最高位移入 R 的最低位，Q 中空出的最低位用来上商。所以 Q 也叫做 余数/商寄存器。每次都会根据余数的符号，交给控制逻辑电路来判断当前商为 1 还是 0，并控制 ALU 的加/减信号。

2.4 浮点数的表示和运算

2.4.1 浮点数的表示

计算机中处理的数，不一定都是纯小数或者纯整数。对于一个同时有整数部分和小数部分的数，我们可以把它拆开，分别用定点整数和定点小数来表示；这样的缺点在于需要手动进行调整组合，往往要编程来调节小数点的位置。而且有些数据的数值范围相差很大，比如太阳的质量是 1.989 × 10^30^ kg，而电子的质量是 9.110 × 10^-31^ kg，如果直接写成 *定点整数 + 定点小数* 的形式就需要很大的机器字长，造成有效数位的浪费。

更好的方式是，小数点不再固定，使用 ” 浮点数 “ 来表示。

浮点数 就是小数点的位置可以浮动的数。例如：

365.242 = 3.65242 × 10^2^

= 365242.0 × 10^-3^

= 0.365242 × 10^3^

这其实就是”科学计数法“的思路。通常，浮点数被表示为下面的形式：

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

上式中，S 称作 **尾数**（可正可负），j 称作 **阶码**（可正可负），r 是 **基数**。在计算机中基数一般取 2，也可以取 4、8 或 16 等。

例如，一个二进制数 N = 11.0101，那么它可以写成各种不同的浮点形式：

N = 11.0101

= 1.10101 × 2^1^

= 0.110101 × 2^10^

= 11010.1 × 2^-11^

= 0.00110101 × 2^100^

...

为了提高数据精度，并且便于比较，在计算机中规定浮点数的尾数用 **纯小数形式** 表示。所以上面的 0.110101 × 2^10^ 和 0.00110101 × 2^100^ 两种形式都可以在计算机中表示 N。不过很明显，后一种形式对有效数位是一种浪费，所以将尾数最高位为 1 的浮点数称为 **规格化数**。

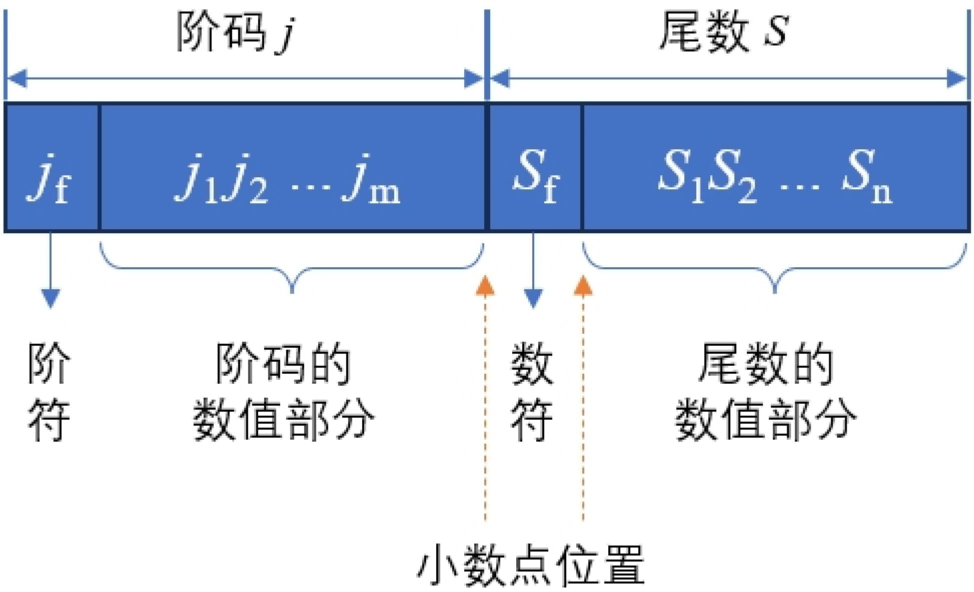
这样，N 就有了唯一的规格化浮点形式：

N = 0.110101 × 2^10^

浮点数表示为规格化形式，精度是最高的。

1. 浮点数的表示格式

在计算机中，浮点数的格式如下图所示。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。

- 阶码是纯整数，阶符和阶码值合起来决定了小数点的实际位置；阶码值的位数 m 再结合阶符，可以反映浮点数的表示范围。
- 尾数是纯小数，数符 S~f~ 代表了浮点数的正负，而尾数值则是有效数位，位数 n 反映了浮点数的精度。

2. 浮点数的表示范围

假设浮点数 N 的阶码 j 数值部分有 m 位，尾数 S 数值部分有 n 位。

阶码是纯整数，尾数是纯小数，它们可以各自选择编码方式。对于非规格化的浮点数，如果阶码和尾数都用原码表达，各自的取值范围如下：

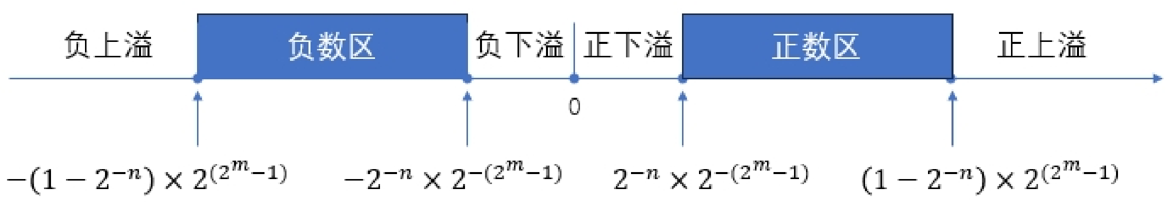
	阶码 *j*（m + 1 位原码）		**尾数 *S*（n + 1 位原码）**			
	最小	最大	最小	绝对值最小（不为 0 时）		最大
				负	正	
真值	$-(2^m-1)$	2^m-1	$-(1-2^{-n})$	-2^{-n}	2^{-n}	$1-2^{-n}$
机器码	$\underbrace{111\cdots1}_{m+1个1}$	$0\underbrace{11\cdots1}_{m个1}$	$1.\underbrace{11\cdots11}_{n个1}$	$1.\underbrace{00\cdots0}_{n-1个0}1$	$0.\underbrace{00\cdots0}_{n-1个0}1$	$0.\underbrace{11\cdots1}_{n个1}$

那么阶码 j 和尾数 S 组合之后，能表示的浮点数最大范围就是：

		尾数 *S* 和阶码 *j* 的取值	**浮点数 *N* 的值**
正数	最大	$S = 1-2^{-n}$，$j = 2^m-1$	$(1-2^{-n})\times2^{(2^m-1)}$
	最小	$S = 2^{-n}$，$j = -(2^m-1)$	$2^{-n}\times2^{-(2^m-1)}$
0		$S = 0$，$j = -(2^m-1)$	0
负数	最大	$S = -2^{-n}$，$j = -(2^m-1)$	$-2^{-n}\times2^{-(2^m-1)}$
	最小	$S = -(1-2^{-n})$，$j = 2^m-1$	$-(1-2^{-n})\times2^{(2^m-1)}$

- 正数最大为 $(1-2^{-n})\times2^{2^m-1}$，最小为 $2^{-n}\times2^{-(2^m-1)}$；
- 负数最大为 $-2^{-n}\times2^{-(2^m-1)}$，最小为 $-(1-2^{-n})\times2^{2^m-1}$；
- 当 S = 0，j = - (2^m^ - 1) 时，可以表示 0 值。

在数轴上表示出来，如下图所示：



可以看到，原码是关于原点对称的，所以浮点数的表示范围也是关于原点对称的。

当运算结果大于能表示的最大正数时，称为正上溢；小于最小负数时。称为负上溢：两者统称 **上溢**。由于尾数的溢出可以通过移位、增加阶码来调整，因此上溢的本质就是 **阶码大于最大阶码**，这时机器会停止计算，进行中断溢出处理。

当运算结果在 0 至最小正数之间时，称为正下溢；在 0 至最大负数之间时，称为负下溢，统称 **下溢**。同样道理，下溢的本质是 **阶码小于最小阶码**，这时溢出的数值绝对值非常小，通常可以将尾数各位直接强置为 0，按 ” **机器零** “ 来处理，机器可以继续正常运行。

类似地，如果阶码和尾数都用补码表达，各自的取值范围如下：

	阶码 j（m + 1 位补码）		尾数 S (n + 1 位补码)			
	最小	最大	最小	绝对值最小（不为 0 时）负	绝对值最小（不为 0 时）正	最大
真值	-2^m	$2^m - 1$	- 1	-2^{-n}	2^{-n}	$1 - 2^{-n}$
机器码	$1\underbrace{00\cdots0}_{m个0}$	$0\underbrace{11\cdots1}_{m个1}$	$1.\underbrace{00\cdots0}_{n个0}$	$1.\underbrace{11\cdots1}_{n个1}$	$0.\underbrace{00\cdots0}_{n-1个0}1$	$0.\underbrace{11\cdots1}_{n个1}$

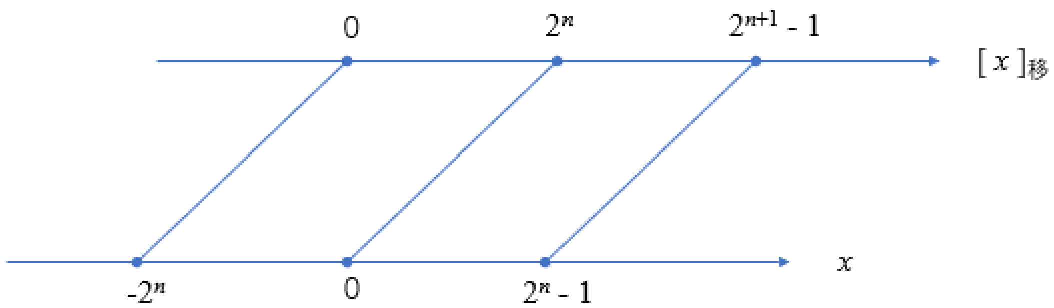
具体应用当中，阶码和尾数到底应该选择哪种编码方式呢？

很明显，采用补码的优势在于方便做加减运算，但是在表示尾数时，数值范围就无法相对原点对称了。而且对于浮点数来说，尾数是不能直接做加减的，需要先把阶数 ” 对齐 “ ，所以使用补码好处并不明显；反倒是在做乘除运算时，阶码需要直接做加减操作。

所以我们可以发现，阶码可以用补码表示；而尾数用原码和补码表示都可以，用原码会更好一点。

进一步分析会发现，其实只要尾数 S = 0，无论阶码 j 取什么值都应该表示 0 值。不过浮点数的尾数是受精度限制的，只要阶码没有达到最小值，就还可以通过继续减小阶码、让尾数的数值左移，这样就能使得之前丢弃的低位重新成为有效数位。所以只有当阶码已经是最小值时，尾数 S = 0，才能说明当前的数值已经足够小了，可以认为就是 0；这时的尾数本质上是一个 ” 接近 0 的很小的数 “，那它的正负就还是有意义的，所以原码的 ± 0 也有了独特的含义。

不过这样一来，” 机器零 “ 的机器码不全为 0，而是 1 0 0 ... 0；这会对实现计算机的 ” 判 0 “ 电路造成一些麻烦。所以我们可以考虑把阶码做个调整，让它能够用 ” 全 0 “ 的机器码来表示真正的 0 值。考虑到阶码范围 ” 关于原点对称 “ ，我们要表示的阶码真值范围还是 -2^m^ ~ 2^m^ - 1，所以要让全 0 的机器码对应最小的真值 -2^m^，只要整体做一个 2^m^ 的偏移就可以了——这就是 “移码” 的编码方式。



$$[x]_{移} = 2^n + x \quad (-2^n \le x < 2^n)$$

用移码来表示阶码还有一个好处，就是方便进行阶数的比较和对齐，简称 “**对阶**” 。这在进行浮点数加减运算时非常重要，尾数只有在阶数相同的时候才能做加减。如果用补码表示阶码，那么可以通过阶码相减来确定哪个阶码大；而采用移码则更加简单，直接比较两个阶码的二进制大小关系就可以了，这在电路上更加容易实现。

阶码采用移码、尾数采用原码编码后，各自的取值范围如下：

	阶码 j（m + 1 位移码）		尾数 S (n + 1 位原码)			
	最小	最大	最小	绝对值最小（不为 0 时）		最大
				负	正	
真值	-2^m	$2^m - 1$	$-(1 - 2^{-n})$	-2^{-n}	2^{-n}	$1 - 2^{-n}$
机器码	$\underbrace{0\,0\,0 \cdots 0}_{m+1\text{个}0}$	$\underbrace{1\,1\,1 \cdots 1}_{m+1\text{个}1}$	$1.\underbrace{1\,1 \cdots 1}_{n\text{个}1}$	$1.\underbrace{0\,0 \cdots 0}_{n-1\text{个}0}1$	$0.\underbrace{0\,0 \cdots 0}_{n-1\text{个}0}1$	$0.\underbrace{1\,1 \cdots 1}_{n\text{个}1}$

一旦确定了浮点数的位数，那么如何分配阶码和尾数的位数，将直接影响到浮点数的表示范围和精度。整体来说，阶码位数越多，说明浮点数的表示范围越大；而尾数位数越多，说明浮点数的精度越高。

3. 浮点数的规格化

由于规格化数的精度最高，所以当一个非零的浮点数不是规格化数时，应该通过左右移动尾数、并同时修改阶码的方法，将它转换为规格化数。把一个非规格化数转换成规格化数的过程，叫做 **规格化**。

规格化的本质类似于 “科学计数法” 的表达，通过保证尾数最高数位上是一个有效值，尽可能多地保留有效数字的尾数，从而提高精度。

规格化可以分为 “**左规**” 和 “**右规**” 两种。以基数 r = 2 为例：

- 左规：向左规格化。当运算结果尾数的最高数位不是有效位，即出现 0.0...01... 的形式时，需要向左规格化。左规时，尾数左移一位，阶码减 1；
- 右规：向右规格化。当运算结果尾数的小数点左侧出现有效位，即整数部分不为 0 时，需要向右规格化。右规时，尾数右移一位，阶码加 1；需要右规时，只需进行一次。

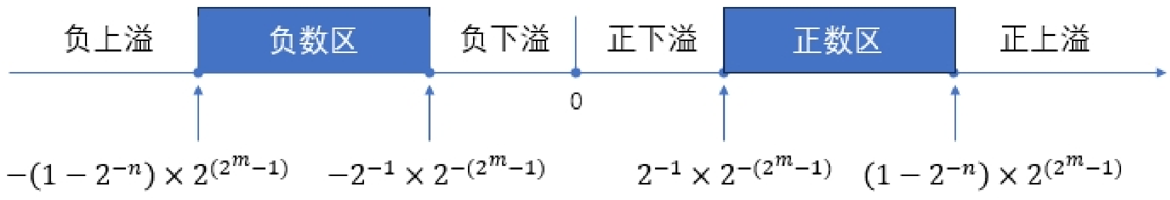
当基数不同时，规格化的原则会有相应的改变。比如，当基数 r = 4 时，阶码每次加/减 1，就相当于多乘/除以 4，也就是左/右移 2 位。所以左规就是尾数左移 2 位，阶码减 1；右规是尾数右移 2 位，阶码加 1。尾数的最高 2 位不全为 0 的数，就是规格化数。

用原码表示的尾数 S，经过规格化处理后，绝对值应该满足 2^-1^ ≤ |S| ≤ 1。它的取值范围如下：

	规格化数的尾数 S (n + 1 位原码)			
	正数		负数	
	最大	最小	最大	最小
真值	$1 - 2^{-n}$	2^{-1}	-2^{-1}	$-(1 - 2^{-n})$
机器码	$0.\underbrace{1\,1 \cdots 1}_{n\text{个}1}$	$0.1\underbrace{0 \cdots 0}_{n-1\text{个}0}$	$1.1\underbrace{0 \cdots 0}_{n-1\text{个}0}$	$1.\underbrace{1\,1 \cdots 1}_{n\text{个}1}$

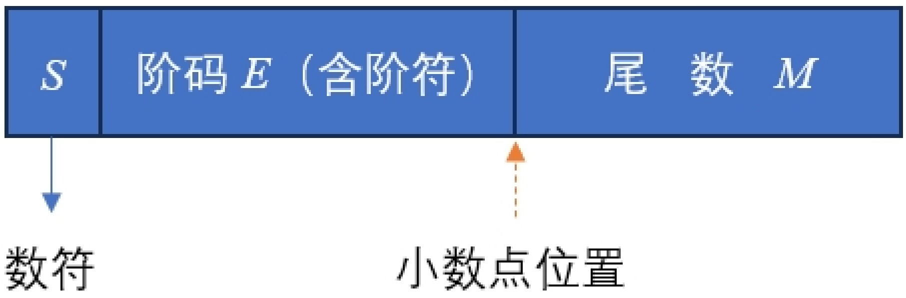
- 正数为 0. 1×...× 的形式，最大值为 0. 11...1 = 1 - 2^-n^，最小值为 0. 10...0 = 2^-1^；
- 负数为 1. 1×...× 的形式，最大值为 1. 10...0 = -2^-1^，最小值为 1. 11...1 = - (1 - 2^-n^)。

这样，浮点数规格化后的取值范围被缩小了。在数轴上表示为：



4. IEEE 754 标准浮点数

在现代计算机中，浮点数的格式一般采用 IEEE 制定的国际标准。IEEE 754 标准规定的浮点数形式为：



- S 为数符，直接表示浮点数的正负，它与尾数所表示的有效数位是分开的。
- 阶码 E 包含了阶符，用移码来表示，不过这里移码的偏移量不是 2 的整次幂，而是要再减去 1。假设阶码 E 的位数为 m + 1，那么偏移量就是 2^m^ - 1。
- 尾数 M 是原码表示的纯小数。

浮点数的位数不同，可以表示不同的数值范围和精度。IEEE 标准中常用的浮点数有三种：短浮点数（单精度）、长浮点数（双精度）和临时浮点数（延伸双精度）。

浮点数类型	数符 S	阶码 E	尾数 M	总位数	偏置值	
					十六进制	十进制
短浮点数	1	8	23	32	7FH	127
长浮点数	1	11	52	64	3FFH	1023
临时浮点数	1	15	64	80	3FFFH	16383

IEEE 标准默认的基数为 2。以短浮点数（单精度浮点数）为例，总共 32 位：最高 1 位是数符位；后面的 8 位是阶码域（指数部分），偏置值为 2^8-1^ - 1 = 127，即阶码部分的 8 位无符号数值减去 127，就得到了真正的阶码 E；最后的 23 位是尾数域（小数部分）。

- IEEE 标准用 **规格化数** 表示一般的数值，这时尾数 M 的最高有效位一定为 1；于是 IEEE 标准规定这个最高有效位的 1 可以省略，并且将它隐藏放在整数位上，称为 “**隐藏位**”。这样就可以多出一位有效数位，从而提高了精度。例如，短浮点数的 23 位尾数，可以表示 24 位有效数字；临时浮点数不采用隐藏位的方案。
- IEEE 标准使用 **非规格化数** 表示 0 附近的很小的数。这时阶码 E 所有位全部为 0；尾数 M 不为 0，且没有隐藏位，或者说隐藏位为 0。

除此之外，IEEE 标准还规定了几种特殊情况：

- 当阶码 E 所有位全为 0，并且尾数 M 也为 0 时，表示 **0 值**（浮点数 0）。根据数符不同可以分别表示 ± 0。
- 当阶码 E 所有位全为 1，并且尾数 M 为 0 时，表示 **无穷大**。根据数符不同可以分别表示 ±∞。
- 当阶码 E 所有位全为 1，并且尾数 M 不为 0 时，表示这不是一个数（NaN）。

以 32 位的单精度浮点数为例，所有的机器码和对应的取值范围如下：

类型	数符 S	实际阶码 E	阶码域（8位）	尾数 M（23位）	浮点数值
零	0	—	0000 0000	000 0000 0000 0000 0000 0000	0.0
最小的非规格化数	0	-126	0000 0000	000 0000 0000 0000 0000 0001	$2^{-23} \times 2^{-126}$
最大的非规格化数	0	-126	0000 0000	111 1111 1111 1111 1111 1111	$(1 - 2^{-23}) \times 2^{-126}$
最小的规格化数	0	-126	0000 0001	000 0000 0000 0000 0000 0000	1.0×2^{-126} $\approx 1.18\times 10^{-38}$
中间的普通规格化数	0	0	0111 1111	000 0000 0000 0000 0000 0000	1.0×2^{0}
	0	1	1000 0000	100 0000 0000 0000 0000 0000	1.5×2^{1}
最大的规格化数	0	127	1111 1110	111 1111 1111 1111 1111 1111	$(2 - 2^{-23}) \times 2^{127}$ $\approx 3.4\times 10^{38}$
无穷	0	—	1111 1111	000 0000 0000 0000 0000 0000	$+\infty$
NaN	0	—	1111 1111	非 0	NaN

例如，对于十进制数 178.125，把它写成 IEEE 标准的短浮点数。

我们需要分整数部分和小数部分，首先转换成二进制数的表示；然后写成类似 “科学计数法” 的二进制浮点数表达。

数的表示	数值
原始十进制数	178.125
二进制数	10110010. 001
二进制浮点表示	$1.0110010001 \times 2^{111}$

这是一个正数，符号位为 0；然后从二进制浮点表达中得到阶码和尾数。将 8 位二进制阶码加上偏移量 127，尾数隐藏整数位的 1 后补成 23 位，就是最终符合 IEEE 标准的 32 位短浮点数。

数符 S	阶码域（8位）	尾数 M（23位）
0	0000 0111 + 0111 1111 = 1000 0110	011 0010 0010 0000 0000 0000

最后，我们也可以总结一下计算机中数据的定点、浮点表示的区别：

- 数值的表示范围不同。对于相同的字长，浮点表示法所能表示的数值范围远大于定点表示法。
- 数值精度不同。对于相同的字长，浮点数虽然扩大了数的表示范围，但精度降低了。
- 数据的运算不同。浮点数包括阶码和尾数两部分，运算时不仅要做尾数的运算，还要做阶码的运算；而且运算结果要做规格化，所以浮点数运算比定点数运算复杂。
- 溢出问题。在定点数运算中，当运算结果超出数的表示范围时，发生溢出；在浮点数运算中, 当运算结果超出尾数表示范围时，不一定发生溢出；只有规格化后，阶码超出所能表示的范围时，才发

生溢出。

5. C 语言中的浮点数据类型和类型转换

在 C 语言中，用 float 类型来表示 IEEE 标准中的 32 位单精度浮点数，用 double 类型来表示 64 位双精度浮点数。long double 类型对应扩展的双精度浮点数，具体的格式会随编译器和处理器架构的不同而改变。

在进行不同类型数据的混合运算时，遵循的原则是 “类型提升”，即由位数较少的类型（比如 char、short）向位数更多的类型（比如 int、long、double）转换。这时数据范围会扩大、精度也变大，因此不会溢出、也不会损失精度，实现了 “无损转换”。类型转换以 char → int → long → double 和 float → double 最为常见，从前到后范围和精度都从小到大、从低到高，转换过程没有损失。

例如，long 类型与 int 类型一起运算，需先将 int 类型转换为 long 类型，然后再进行运算，结果为 long 类型；如果 float 类型和 double 类型一起运算，需要先将 float 转换为 double 类型再进行运算，结果为 double 类型。所有这些转换都是系统自动进行的，这种转换称为 **隐式类型转换**。C 语言中的隐式转换有算术转换、赋值转换及输出转换。

在 C 语言程序里，数据类型除隐式转换外，也可以显式地进行转换（强制类型转换）。整型之间、浮点类型之间、整型和浮点型之间都可以进行转换。我们需要注意转换过程中数据的取值范围是否发生了变化（是否溢出）、精度是否发生了缺失。精度的缺失主要针对浮点数而言。

具体来说：

- 从 int 或者 float 转换为 double 时，因为 double 的表示范围更大、有效位数更多，因此可以无损转换；
- 从 double 转换为 float 时，因为 float 表示的范围更小，所以可能发生溢出；另外有效数位也减少了，所以可能发生舍入，从而丢失精度；
- 从 float 或 double 转换为 int 时，因为 int 的表示范围更小，所以可能发生溢出；另外 int 没有小数部分，所以数据会仅保留整数部分而丢失精度；
- 从 int 转换为 float 时，表示的范围扩大了，不会发生溢出；但 int 有 32 位，而 float 最多只有 24 位有效数位，所以可能发生数据的舍入，从而丢失精度。

2.4.2 浮点数的加/减运算

浮点数运算的特点，是阶码运算和尾数运算分开进行。浮点数加/减运算可以分为 5 步进行：

（1）对阶

对阶的目的是使两个操作数的小数点位置对齐，使两个数的阶码相等。先求阶差，然后以 “小阶向大阶看齐” 的原则，将阶码小的尾数右移一位（基数为2），阶码加1，直到两个数的阶码相等为止。

（2）尾数求和

将对阶后的尾数，按定点数加/减运算规则运算。

（3）规格化

IEEE 754 规格化尾数的形式为 ±1.×…×，所以当计算结果为非规格化数时，需要进行规格化处理。

- 左规：当结果为 ±0.0…01x…x 时，需进行左规。尾数每左移一位，阶码减 1。可能需要左规多次，直到将第一位 1移到小数点左边。
- 右规：当结果为 ±1x.x…x 时，出现了尾数的溢出，需进行右规。尾数右移一位，阶码加 1。当尾数右移时，最高位 1 被移到小数点前一位作为隐藏位；当最后一位移出时，要考虑舍入。

左规一次相当于乘以2，右规一次相当于除以2；需要右规时，只需进行一次。

（4）舍入

在对阶和尾数右规时，尾数右移可能会将低位丢失，影响精度，IEEE 754有以下4种舍入方式：

- 就近舍入：舍入为最近的那个数，类似于 “四舍五入”，一般被叫做 “ 0 舍 1 入” 法；如果被舍入的值恰好是 100...0 形式，选择舍入为最近的偶数；
- 正向舍入：向 +∞ 方向舍入，即取右边那个数，也叫 “向上舍入”；
- 负向舍入：向 -∞ 方向舍入，即取左边那个数，也叫 “向下舍入”；
- 截断：朝 0 方向舍入，即取绝对值较小的那个数。

（5）溢出判断

浮点数的溢出，并不是以尾数溢出来判断的；尾数溢出可以通过右规操作得到纠正。运算结果是否溢出，主要看结果的指数是否发生了溢出，因此是由阶码来判断的。

- 若一个正阶码超出了最大允许值（127 或 1023），则发生上溢，产生异常；
- 若一个负阶码超出了最小允许值（-149 或 -1074），则发生下溢，通常把结果按机器零处理。

例如，两个数 x = 29/32 × 2^7^，y = 5/8 × 2^5^，用浮点加法计算 x + y。假设浮点数的阶码和尾数均用补码表示，且阶码为 5 位（含 2 位阶符），尾数为 7 位（含 2 位数符）。

首先，将浮点数写成下面的规格化二进制形式：

x = 0.11101 × 2^111^，y = 0.101 × 2^101^

具体计算过程如下：

（1）对阶

阶码相减 00, 111 - 00, 101 = 00, 010，说明 x 的阶码比 y 的大 2，需要将 y 的尾数右移两位，阶码加 2：

y = 0.00101 × 2^111^

（2）尾数求和

尾数相加 00.11101 + 00.00101 = 01.00010

（3）规格化

运算结果的尾数出现溢出，需要进行右规：尾数右移一位，阶码加 1：

1.00010 × 2^111^ = 0.100010 × 2^1000^，即计算结果为 01, 000; 00, 10001

（4）舍入

结果的尾数用补码表示为：00 10001，不需要舍入。

（5）溢出判断

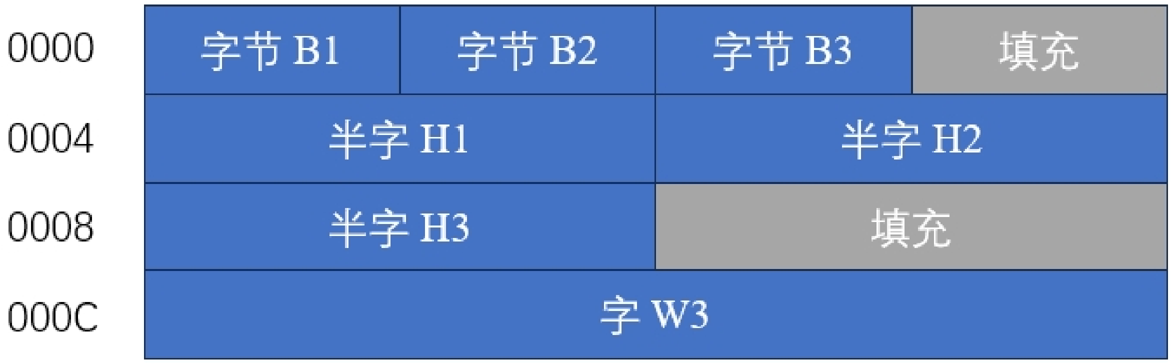
结果的阶码用补码表示为：01 000，由于阶符为 01，说明结果溢出。

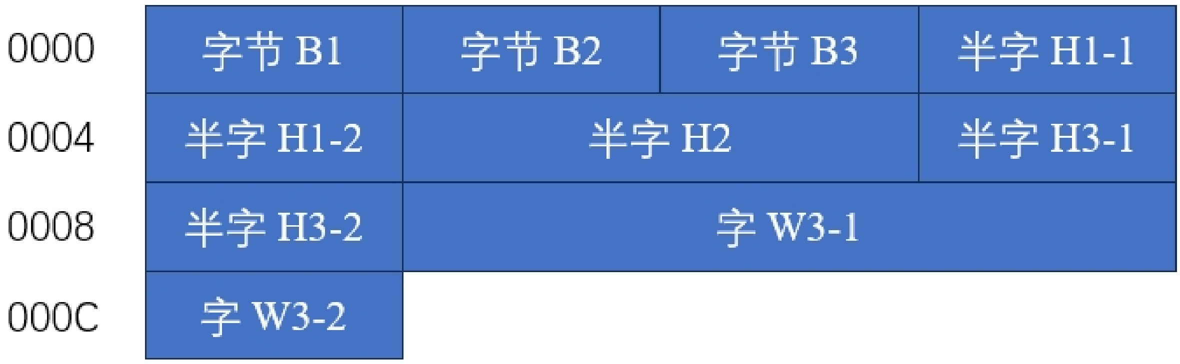
2.5 数据的存储和排列

2.5.1 数据按“边界对齐”方式存储

可以假设字长为 32 位，可按字节、半字、字寻址。在对准边界的 32 位计算机中，半字地址是 2 的整数 倍，字地址是 4 的整数倍，当所存数据不满足此要求时，可填充一个或多个空白字节。这种存储方式称为 “**边界对齐**” 。这样无论所存的数据是字节、半字还是字，均可一次访存取出。虽然浪费了一些存储空间，但可提高存取速度。

数据不按边界对齐方式存储时，半字长或字长的数据可能在两个存储字中，此时需要两次访存，并对高低字节的位置进行调整后才能取得所需数据，从而影响系统的效率。





在 C 语言的 struct 类型中，边界对齐方式存储有两个重要要求：

（1）每个成员按其类型的方式对齐，比如 char 类型的对齐值为 1，short 为 2，int 为 4（单位为字节）。

（2）struct 的长度必须是成员中最大对齐值的整数倍（不够就补空字节），以便在处理 struct 数组时保证每项都满足边界对齐的条件。

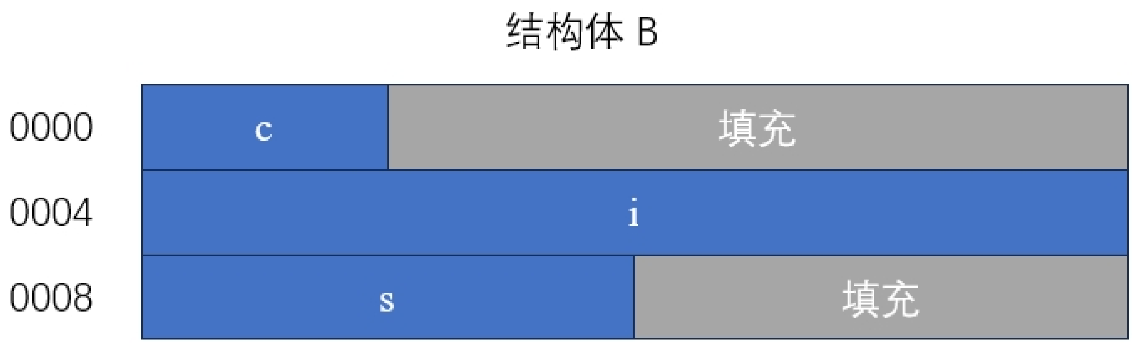
例如，下面是两个成员完全一样的结构体：

```
struct A {
    int i;
    char c;
    short s;
}
struct B {
    char c;
    int i;
    short s;
}
```

但两者在内存中占据的空间却不同。这是因为结构体成员是按定义的先后顺序排列的，编译器要使它们在空间上对齐，所以应该有：

每个成员存储的起始地址 % 该成员的长度 = 0

同时，还需要让结构体的长度是最大成员长度的整数倍。



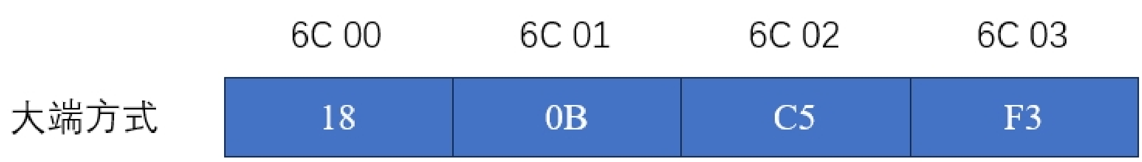
按边界对齐方式，相对于不按边界对齐方式是一种空间换时间的思想。

2.5.2 数据的“大端方式”和“小端方式”存储

在存储数据时，通常用 **最低有效字节（LSB）** 和 **最高有效字节（MSB）** 来分别表示数据的低位和高位。例如，在 32 位机器中定义了一个 int 类型的变量 i，机器数为 18 0B C5 F3H，那么它的 MSB = 18H，LSB = F3H。

现代计算机基本都采用字节编址，也就是每个地址编号对应存放 1 个字节。不同类型的数据占用的字节数不 同，而程序中对每个数据只给定一个地址。变量 i 占据连续的四个字节，它们各有一个内存地址，而变量 i 的地址就是开始的那个字节的地址。假设 i 的地址为 6C 00H，那么 i 具体存放的四个字节的地址就是：6C 00H、6C 01H、6C 02H、6C 03H。而具体每个字节存放什么内容，可以有不同的定义方式。

多字节数据都存放在连续的字节序列中，根据数据中各字节在连续字节序列中的排列顺序不同，分为两种排列方式：**大端方式**（big endian）和 **小端方式**（little endian）。



- 大端方式：先存储高位字节，后存储低位字节。高位字节存储在低位地址中，字中的字节顺序和原序列相同。
- 小端方式：先存储低位字节，后存储高位字节。低位字节存储在低位地址中，字中的字节顺序和原序列相反。

在阅读以小端方式存储的机器代码时，要注意字节是按相反顺序显示的。

2.6 章节练习

一、单项选择题

1. 【2018真题】冯·诺伊曼结构计算机中的数据采用二进制编码表示，其主要原因是（ ）。

   I. 二进制的运算规则简单　　　　II. 制造两个稳态的物理器件较容易

   III. 便于用逻辑门电路实现算术运算

   A. 仅 I、II　　B. 仅 I、III　　C. 仅 II、III　　D. I、II 和 III

   答案： D

2. 【 2015真题】由 3个“1”和 5个“0”组成的8位二进制补码，能表示的最小整数是 （ ）。

   A． -126　　B． -125　　C． -32　　D． -3

   答案：B

3. 【2022真题】32 位补码所能表示的整数范围是 （ ）。

   A. $-2^{32}\sim2^{31}-1$　　B. $-2^{31}\sim2^{31}-1$　　C. $-2^{32}\sim2^{32}-1$　　D. $-2^{31}\sim2^{32}-1$

   答案：B

4. 【2021真题】已知带符号整数用补码表示，变量 x、y、z 的机器数分别为 FFFDH、FFDFH、7FFCH，下列结论中，正确的是 （ ）。

   A. 若x、y和z为无符号整数,则z<x<y　　B. 若x、y和z为无符号整数,则x<y<z

   C. 若x、y和z为带符号整数,则x<y<z　　D. 若x、y和z为带符号整数,则y<x<z

   答案：D

5. 【2016真题】有如下 C 语言程序段

```
short si = -32767;
unsigned short usi = si;
```

执行上述两条语句后， usi 的值为 （ ）。

A． -32767　　B． 32767　　C． 32768　　D． 32769

答案：D

6. 【2019真题】考虑以下 C 语言代码：

```
unsigned short usi = 65535;
short si = usi;
```

执行上述程序段后， si 的值是 （ ）。

A．-1　　B．-32767　　C．-32768　　D．-65535

答案：A

7. 【2012真题】假定编译器规定 int 和 short 型长度分别为 32 位和 16 位，执行下列 C 语言语句：

```
unsigned short x=65530;
unsigned int y=x;
```

得到 y 的机器数为 （ ）。

A． 0000 7FFAH B． 0000 FFFAH C． FFFF 7FFAH D． FFFF FFFAH

答案：B

8. 【2009真题】一个 C 语言程序在一台 32 位机器上运行。程序中定义了三个变量 x、 y 和 z，其中 x 和 z 为 int 型， y 为 short 型。当 x=127， y=-9 时，执行赋值语句 z=x+y 后， x、 y 和 z 的值分别是 （ ）。

A． x=0000007FH， y=FFF9H， z=00000076H B． x=0000007FH， y=FFF9H， z=FFFF0076H

C． x=0000007FH， y=FFF7H， z=FFFF0076H D． x=0000007FH， y=FFF7H， z=00000076H

答案：D

9. 【2018真题】整数x的机器数为1101 1000，分别对x进行逻辑右移1位和算术右移1位操作，得到的机器数各 是 （ ）。

A.1110 1100、1110 1100 B. 0110 1100、1110 1100

C.1110 1100、0110 1100 D. 0110 1100、 01101100

答案：B

10. 【2013真题】某字长为 8 位的计算机中，已知整型变量 x、y的机器数分别为$[x]_{补}=1\quad 1110100$，$[y]_{补}=1\quad 0110000$。若整型变量 $z=2*x+y/2$，则 z 的机器数为 （ ）。

A. 1 1000000 B. 0 0100100 C. 1 0101010 D. 溢出

答案：A

11. 【2018真题】假定带符号整数采用补码表示，若 int 型变量 x 和 y 的机器数分别是 FFFF FFDFH 和 0000 0041H，则 x、y 的值以及 x-y 的机器数分别是 （ ）。

A.x=-65，y=41，x-y 的机器数溢出 B.x=-33，y=65，x-y 的机器数为FFFF FF9DH

C.x=-33，y=65，x-y 的机器数为FFFF FF9EH D. x = -65，y = 41，x-y 的机器数为FFFF FF96H

答案：C

12. 【2016真题】某计算机字长为 32 位，按字节编址，采用小端（Little Endian）方式存放数据。假定有一个 double 型变量，其机器数表示为 1122 3344 5566 7788H，存放在 0000 8040H 开始的连续存储单元中，则存储单元 0000 8046H 中存放的是 （ ）。

A． 22H B． 33H C． 66H D． 77H

答案：A

13. 【2018真题】某32位计算机按字节编址，采用小端(Little Endian)方式。若语句 “inti=0;” 对应指令的机器代码为 “C7 45 FC 00 00 00 00”，则语句 “int i = -64;” 对应指令的机器代码是 （ ）。

A.C7 45 FC C0 FF FF FF B.C7 45 FC 0C FF FF FF C.C7 45 FC FF FF FF C0 D.C7 45 FC FF FF FF 0C

答案：A

14. 【2012真题】某计算机存储器按字节编址，采用小端方式存放数据。假定编译器规定 int 型和 short

型长度分别为 32 位和 16 位，并且数据按边界对齐存储。某 C 语言程序段如下：

```
struct{
    int a;
    char b;
    short c;
} record;
record.a=273;
```

若 record 变量的首地址为 0xC008，则地址 0xC008 中内容及 record.c 的地址分别为 （ ）。

A. 0x00、 0xC00D　　B. 0x00、 0xC00E　　C. 0x11、 0xC00D　　D. 0x11、 0xC00E

答案：D

15. 【2020真题】在按字节编址，采用小端方式的 32 位计算机中，按边界对齐方式为以下 C语言结构型变量a分配存储空间。

```
Struct record{
    short    x1;
    int    x2;
} a;
```

若a的首地址为 2020 FE00H，a的成员变量x2的机器数为1234 0000H，则其中34H所在存储单元的地址是 （ ）。

A. 2020 FE03H　　B. 2020 FE04H　　C.2020 FE05H　　D. 2020 FE06H

答案：D

16. 【2012真题】 float 类型（即 IEEE754 单精度浮点数格式）能表示的最大正整数是 （ ）。

A． $2^{126}-2^{103}$　　B． $2^{127}-2^{104}$　　C． $2^{127}-2^{103}$　　D． $2^{128}-2^{104}$

答案：D

17. 【2013真题】某数采用 IEEE 754 单精度浮点数格式表示为C640 0000H，则该数的值是 （ ）。

A. -1.5×2^{13}　　B. -1.5×2^{12}　　C. -0.5×2^{13}　　D. -0.5×2^{12}

答案：A

18. 【2014真题】float 型数据常用 IEEE 754 单精度浮点格式表示。 假设两个 float 型变量 x 和 y 分别存放在32位寄存器 f_1 和 f_2 中，若(f_1)=CC90 0000H, (f_2)= B0C0 0000H, 则 x 和 y 之间的关系为 （ ）。

A. x<y且符号相同　　B. x<y且符号不同　　C. x>y且符号相同　　D. x>y且符号不同

答案：A

19. 【2015真题】下列有关浮点数加减运算的叙述中，正确的是 （ ）。

Ⅰ．对阶操作不会引起阶码上溢或下溢

Ⅱ．右规和尾数舍入都可能引起阶码上溢

Ⅲ．左规时可能引起阶码下溢

Ⅳ．尾数溢出时结果不一定溢出

A．仅Ⅱ、Ⅲ　　B． 仅Ⅰ、Ⅱ、Ⅳ　　C．仅Ⅰ、Ⅲ、Ⅳ　　D． Ⅰ、Ⅱ、Ⅲ、Ⅳ

答案： D

20. 【2018真题】IEEE 754 单精度浮点格式表示的数中，最小的规格化正数是 （ ）。

A.$1.0x2^{-126}$ B. $1.0x2^{-127}$ C.$1.0x2^{-128}$ D.$1.0x2^{-149}$

答案：A

21. 【2020真题】已知带符号整数用补码表示，float 型数据用 IEEE 754 标准表示，假定变量 x 的类型只
可能是 int 或 float，当 x 的机器数为 C800 0000H时，x 的值可能是 （ ）。

A.$-7x2^{27}$ B.-2^{16} C. 2^{17} D. $25x2^{27}$

答案：A

22. 【2021真题】下列数值中,不能用 IEEE 754 浮点格式精确表示的是 （ ）。

A.1.2 B.1.25 C.2.0 D.2.5

答案：A

23. 【2022真题】-0.4375 的 IEEE 754 单精度浮点数表示为（ ）。

A. BEE0 0000H B. BF60 0000H C. BF70 0000H D. C0E0 0000H

答案：A

二、综合应用题

1. 【2020真题】有实现 x*y 的两个 C 语言函数如下:

```
unsigned umul (unsigned x, unsigned y){return x*y; }
int imul (int x,int y) {return x* y;}
```

假定某计算机 M 中 ALU 只能进行加运算和逻辑运算。请回答下列问题。

（1）若 M 的指令系统中没有乘法指令，但有加法、减法和位移等指令，则在 M 上也能实现上述两个函数中的乘法运算，为什么?

（2）若 M 的指令系统中有乘法指令，则基于 ALU、位移器、寄存器以及相应控制逻辑实现乘法指令时，控制逻辑的作用是什么?

（3）针对以下三种情况：a）没有乘法指令；b）有使用ALU和位移器实现的乘法指令；c）有使用阵列乘法器实现的乘法指令，函数 umul() 在哪种情况下执行时间最长？哪种情况下执行的时间最短？说明理由。

（4）n 位整数乘法指令可保存 2n 位乘积，当仅取低 n 位作为乘积时，其结果可能会发生溢出。当 n=32、$x=2^{31}-1$、y=2 时，带符号整数乘法指令和无符号整数乘法指令得到的 x*y 的 2n 位乘积分别是什么（用十六进制表示）？此时函数 umul() 和 imul() 的返回结果是否溢出？对于无符号整数乘法运算，当仅取乘积的低 n 位作为乘法结果时，如何用 2n 位乘积进行溢出判断?

答案：

（1）编译器可以将乘法运算转换为一个循环代码段，在循环代码段中通过比较、加法、移位等指令实现乘法运算。

（2）控制逻辑的作用为： 控制循环次数，控制加法和移位操作。

（3）a）最长， c）最短。

对于 a）， 需要用循环代码段（软件）实现乘法操作，因而需反复执行很多条指令， 而每条指令都需要取指令、译码、取数、执行并保存结果，所以执行时间很长； 对于 b）和 c）， 都只要用一条乘法指令实现乘法操作，不过， b）中的乘法指令需要多个时钟周期才能完成，而 c）中的乘法指令可以在一个时钟周期内完成， 所以 c）执行时间最短。

(4) 当 n=32、 x=2^{31}-1、 y=2 时，带符号整数和无符号整数乘法指令得到的 64 位乘积都为 0000 0000 FFFF FFFEH。

函数 imul 的结果溢出，而函数 umul 结果不溢出。对于无符号整数乘法，若乘积高 n 位全为 0，则不溢出，否则溢出。

2. 【2017真题】 已知

$$f(n)=\sum_{i=0}^{n}2^i=2^{n+1}-1=\underbrace{11\ldots1}_{n+1\text{ 位}}\mathrm{B}$$

计算 f(n) 的 C 语言函数 f1 如下:

```
int f1(unsigned n){
    int sum=1, power=1;
    for(unsigned i=0;i<=n-1;i++){
        power *= 2;
        sum += power;
    }
    return sum;
}
```

将 f1 中的 int 都改为 float， 可得到计算 f(n) 的另一个函数 f2。假设 unsigned 和 int 型数据都占 32 位， float 采用 IEEE 754 单精度标准。请回答下列问题。

(1) 当 n=0 时， f1 会出现死循环，为什么? 若将 f1 中的变量 i 和 n 都定义为 int 型，则 f1 是否还会出现死循环? 为什么?

(2) f1(23) 和 f2(23) 的返回值是否相等? 机器数各是什么（用十六进制表示）?

(3) f1(24)和 f2(24)的返回值分别为 33 554 431 和 33 554 432.0， 为什么不相等?

(4) f(31)=2^32^-1， 而 f1(31) 的返回值却为 -1，为什么? 若使 f1(n) 的返回值与 f(n) 相等，则最大的 n 是多少?

(5) f2(127) 的机器数为 7F80 0000H， 对应的值是什么? 若使 f2(n) 的结果不溢出，则最大的 n 是多少? 若使 f2(n) 的结果精确（无舍入）， 则最大的 n 是多少?

答案:

(1) 由于 i 和 n 是 unsigned 型，故 “i<=n-1” 是无符号数比较； n=0 时， n-1 的机器数为全 1，值是 2^{32}-1，为 unsigned 型可表示的最大数，条件 “i<=n-1” 永真，因此出现死循环。

若 i 和 n 改为 int 类型，则不会出现死循环。因为 “i<=n-1” 是带符号整数比较， n=0 时， n-1 的值是-1，当 i=0 时条件 “i<=n-1” 不成立，此时退出 for 循环。

(2) f1(23)与 f2(23)的返回值相等。

f(23) = 2^{23+1}-1 = 2^{24}-1，它的二进制形式是 24 个1。 int 占 32 位，没有溢出。 float 有 1 个符号位， 8 个指数（阶码）位， 23 个底数（尾数）位， 23 个底数位可以表示 24 位的底数。所以两者返回值相等。

f1(23)的机器数是 00FF FFFFH；f2(23)的机器数是 4B7F FFFFH。显而易见前者是 24 个 1，即 0000 0000 1111 1111 1111 1111 1111 1111 $_{(2)}$ ，后者符号位是 0，指数位为 $23+127_{(10)}$ = 1001 0110 $_{(2)}$ ，底数位是 111 1111 1111 1111 1111 1111 $_{(2)}$ 。

（3）当 n=24 时， f(24) = 1 1111 1111 1111 1111 1111 1111 B，而 float 型数只有 24 位有效位，舍入后数值增大，所以 f2(24) 比 f1(24) 大 1。

（4）f(31) 已超出了 int 型数据的表示范围，用 f1(31) 实现时得到的机器数为 32 个 1，作为 int 型数解释时其值为-1，所以 f1(31) 的返回值为-1。

因为 int 型最大可表示数是 0 后面加 31 个 1，故使 f1(n)的返回值与 f(n)相等的最大 n 值是 30。

（5） f2 返回值为 float 型，7F80 0000H 中数符为0，阶码全为1，尾数为0。IEEE 754 标准用“阶码全 1、尾数全 0”表示无穷大。所以机器数 7F80 0000H 对应的值是+∞。

当 n=126 时， f(126) = 2^{127}-1 = 1.1…1×2^{126}，对应阶码为 127+126=253，尾数部分舍入后阶码加 1，最终阶码为 254，是 IEEE 754 单精度格式表示的最大阶码。故使 f2 结果不溢出的最大 n 值为 126。

当 n=23 时， f(23) 为 24 位 1， float 型数有 24 位有效位，所以不需舍入，结果精确。所以使 f2 获得精确结果的最大 n 值为 23。