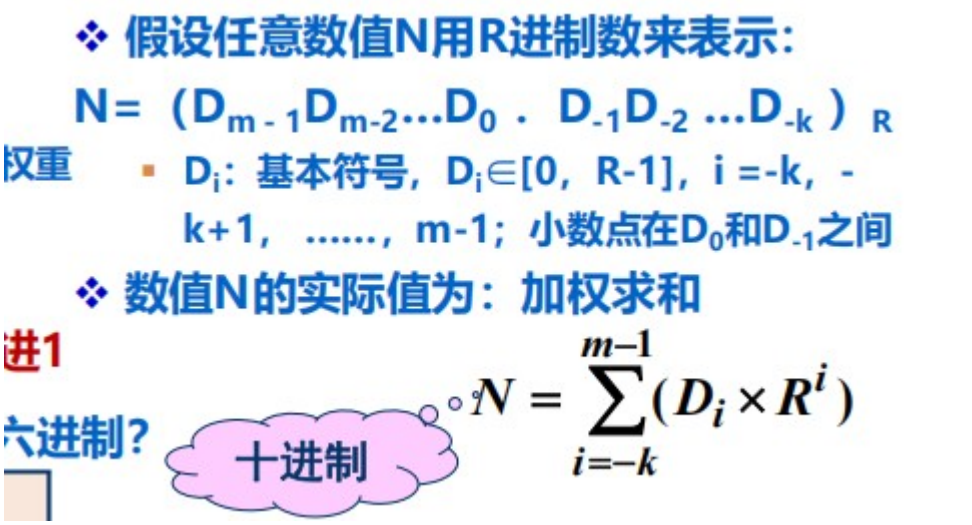
进位计数制

基数 R：允许使用的基本数码个数

▪ 权 Wi：权也称位权，第 i 位上的数码的权重值，即权与数码所处的位置 i 有关

▪ Wi ＝R



进制	基数R	权W_i	数码符号
十进制	R=10	10^i	0～9
二进制	R=2	2^i	0、1
八进制	R=8	8^i	0～7
十六进制	R=16	16^i	0～9、A～F

二进制→八进制

▪ 以小数点为中心分别向两边分组，每三位一组，写出对应的八进制数字（不够位数则在两边加 0 补足 3 位）

❖ 二进制→十六进制

▪ 以小数点为中心分别向两边分组，每四位一组，写出对应的十六进制符号（不够位数则在两边加 0 补足 4 位）

❖ 八进制→二进制

▪ 将每位八进制数展开为 3 位二进制数，整数的最高位 0 和小数的最低位 0 可以略去

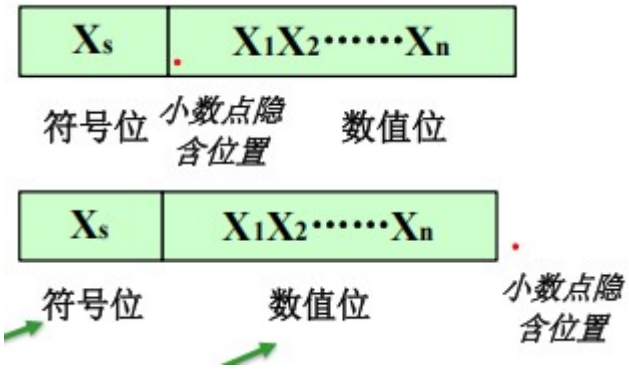
❖ 十六进制→二进制

▪ 将每位十六进制数展开为 4 位二进制数，整数的最高位 0 和小数的最低位 0 可

以略去

如何在计算机内使用二进制来表示十进制数据?
十进制数的编码
❖BCD（Binary Coded Decimal）码：
▪ 使用二进制来编码十进制数字 0～9
❖编码方法：
▪ 一般使用 4 位二进制编码来表示 1 位十进制数字
▪ 在 16 个编码中选用 10 个来表示数字 0～9
▪ 不同的选择构成不同的 BCD 码
❖分类：
▪ 有权码：编码的每一位都有固定的权值，加权求和的值即是表示的十进制数字。如 8421 码、2421 码、5211 码、4311 码、84 -2-1 码等
▪ 无权码：编码的每一位并没有固定的权，主要包括格雷码、余 3 码等
十进制数串的表示方法
数值数据：用于表示数的大小，其值确定，可与数轴上确切的点对应
❖ 非数值数据：没有值大小的概念，表示字符、汉字、图像、声音等信息
❖ 计算机中如何表示数值数据？数值数据的格式要素，必须要考虑三点
▪ 第一，符号如何表示?
▪ 第二，小数点如何表示?
▪ 第三，二进制数字如何排列来表示某个数值?
❖ 机器数：数值数据在计算机中的二进制表示形式。
▪ 机器数的符号位必须被数值化为二进制 0 和 1
▪ 机器数的小数点用隐含规定的方式来表达
▪ 表示的数值范围受计算机字长的限制
▪ 编码方法：不同的排列方式，常见的有原码、反码、补码、移码。
❖ 真值：机器数真正表示的数值数据
❖ 书写：+/－符号 数值（二进制或
十进制）

计算机中参与运算的数值数据有两种：
▪ 无符号数据（Unsigned ）：所有的二进制数据位数均用来表示数值本身，没有正负之分
▪ 带符号数据（Signed）：则其二进制数据位，包括符号位和数值位
❖ 当机器字长相同时，无符号数和有符号数的表示范围是不同的
❖计算机中：小数点的表示方法?
▪ 计算机中没有专用的部件用来表示小数点
▪ 机器数中，小数点及其位置是隐含规定的
▪ 有两种隐含方式：定点和浮点
定点机器数（Fixed Point）：小数点的位置是固定不变的，两种位置：
▪ 定点小数：纯小数，隐含固定在符号位
Xs（最高位）之后，整数位是符号位
▪ 定点整数：纯整数，小数点隐含固定
在最低位之后，最高位为符号位 Xs



浮点机器数：将实数数值大小表示成科学计数法形式

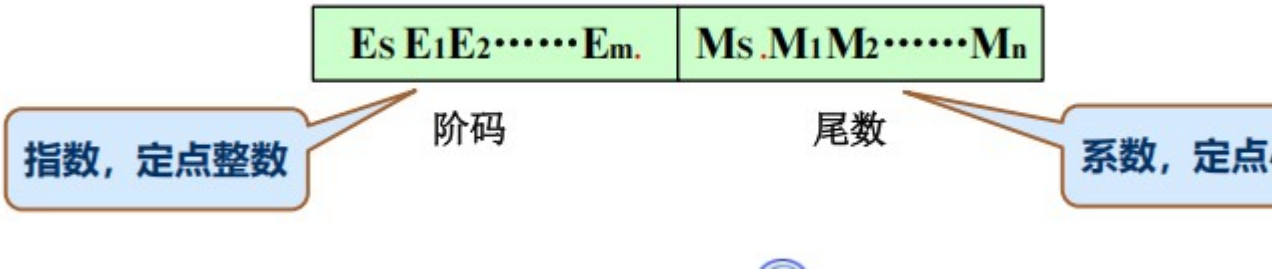
❖ 浮点数 N 由三部分构成：

▪ 阶码的底 R：隐含表示，一般约定为 2、8 或 16

▪ 尾数 M：系数，为定点小数

▪ 阶码 E：R 的指数，为定点整数

❖ 对尾数所表示的定点小数，进行按比例放大（阶码为正数）或者按比例缩小（阶码为负数），数据真值的小数点位置是浮动的——位置由阶码规定



❖ 负精度-δ：机器数在数轴上最靠近原点 0 的负数，计算机能表示的最大负数

❖ 正精度 δ：机器数在数轴上最靠近原点 0 的正数，计算机能表示的最小正数

绝对值称为精度 δ 或分辨率，代表了机器数所能表示的绝对值最小的非零值

❖ 机器数的表示范围：[最小数，最大负数]、0、[最小正数，最大数]

▪ 最大数和和最小数：取决于机器数的编码方式、位数和格式

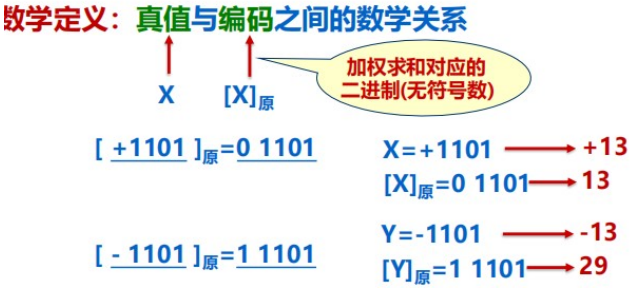
下溢：位于最大负数和最小正数之间的数据（除 0 外），机器无法表示。

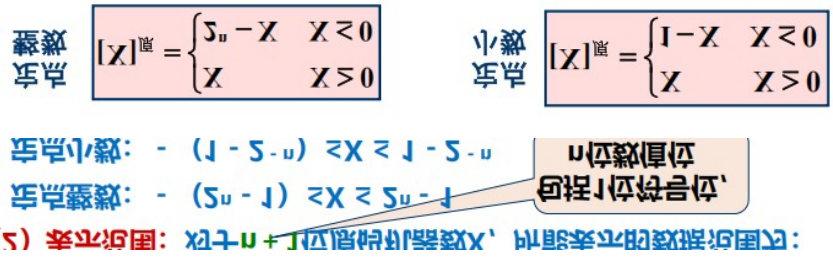
▪ 处理：计算机直接将其视为机器零。

❖ 上溢：当一个数据大于最大（正）数，或者小于最小（负）数时，机器也无法表示，称为上溢，上溢又称溢出。

▪ 处理：计算机置溢出标志位，或者报警。

原码表示法





▪ 使用原码进行加减乘除运算：规则类似于真值的加减乘除运算

▪ 符号位单独处理，不能参加运算，参加运算的是绝对值

▪ 加减运算：

• 根据符号位决定进行的是绝对值的加还是减

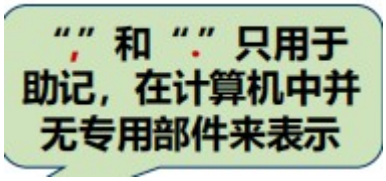
• 进行绝对值的加减运算

• 根据绝对值的大小决定结果（和或者差）的符号

表示方法：最高位为符号位，其他位为数值位

▪ 符号位：0－正数，1－负数

▪ 数值位：与绝对值相同



对于定点整数：

▪ 若 X=+X1X2 .....Xn，则[X]原= 0,X1X2 .....Xn

▪ 若 X=- X1X2 .....Xn，则[X]原= 1,X1X2 .....Xn 。

❖ 对于定点小数：

▪ 若 X=+0.X1X2 .....Xn，则[X]原= 0.X1X2 .....Xn

▪ 若 X=- 0.X1X2 .....Xn，则[X]原= 1.X1X2 .....Xn

补码

定点整数

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

$[X]_{补}= 2^{n+1}+X \quad (\bmod 2^{n+1})$

模2补码

定点小数

$$[X]_{补}=\begin{cases}X & X\geq 0\\ 2+X & X<0\end{cases}$$

$[X]_{补}= 2+X \quad (\bmod 2)$

2的补码：two’s complement

2、表示方法：最高位为符号位，其他位为数值位。

- **符号位：0－正数，1－负数。**
- **数值位：正数时，与绝对值相同；负数时，为绝对值取反后，末位加 1。**

$[X]_{补}= X_0\ X_1\ X_2\ X_3$

权：$-2^3\ \ 2^2\ \ 2^1\ \ 2^0$

真值=加权求和

❖ **对于定点整数：**

- **若$X=+X_1X_2\ldots\ldots X_n$，则$[X]_{补}= 0,X_1X_2\ldots\ldots X_n$**
- **若$X=-X_1X_2\ldots\ldots X_n$，则$[X]_{补}= 1,\overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+1$**

❖ **对于定点小数：**

- **若$X=+0.X_1X_2\ldots\ldots X_n$，则$[X]_{补}= 0.X_1X_2\ldots\ldots X_n$**
- **若$X=-0.X_1X_2\ldots\ldots X_n$，则$[X]_{补}= 1.\overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+0.00\ldots\ldots1$。**

0 的表示：0 的补码表示形式是唯一的，即分别按照正数和负数表示均一致，为全零。

 [+0]补＝ 00...0 [-0]补＝ 00...0

 （2）表示范围：对于 n＋1 位补码机器数 X，它所能表示的数据范围为：

 定点整数：－2n≤X ≤ 2n－1

 定点小数：－1≤X ≤ 1－2－n

（3）补码运算

- **补码的符号位同数值位一起参加运算**
- **可以将减法转化为加法**
- **简化了运算器的结构**

❖ **计算机中的整型数据（int）均用补码来表示**

反码

定点整数

$$[X]_{反}=\begin{cases}X & X\geq 0\\ 2^{n+1}-1+X & X\leq 0\end{cases}\quad(\bmod(2^{n+1}-1))$$

定点小数

$$[X]_{反}=\begin{cases}X & X\geq 0\\ 2-2^{-n}+X & X\leq 0\end{cases}\quad(\bmod(2-2^{-n}))$$

、表示方法：最高位为符号位，其他位为数值位。

- 符号位：0－正数，1－负数。
- 数值位：正数时，与绝对值相同；负数时，为绝对值取反

- 对于定点整数：
  - 若X=+X_1X_2……X_n，则$[X]_{反}$ = 0,X_1X_2……X_n
  - 若X=- X_1X_2……X_n，则$[X]_{反}$= 1, $\overline{X_1}\,\overline{X_2}$…… $\overline{X_n}$
- 对于定点小数：
  - 若X=+0.X_1X_2……X_n，则$[X]_{反}$= 0.X_1X_2……X_n
  - 若X=-0.X_1X_2……X_n，则$[X]_{反}$= 1. $\overline{X_1}\,\overline{X_2}$…… $\overline{X_n}$

- （1）0的表示：0 的反码表示有两种形式
  - $[+0]_{反}$= 00…0　　$[-0]_{反}$= 11…1
- （2）表示范围：对于n＋1位反码机器数X，它所能表示的数据范围为：
  - 定点整数：$-(2^n-1)\leq X\leq 2^n-1$
  - 定点小数：$-(1-2^{-n})\leq X\leq 1-2^{-n}$

移码

定点整数

$$[X]_{移}=2^n+X\quad(\bmod 2^{n+1})$$

2、表示方法：最高位为符号位，其他位为数值位。

- 符号位：1－正数，0－负数。
- 数值位：正数时，与绝对值相同；负数时，为绝对值取反后，末位加 1。

移码=补码的符号位取反

- 对于定点整数：
  - 若X=+X_1X_2……X_n，则$[X]_{移}$= 1,X_1X_2……X_n；
  - 若X=- X_1X_2……X_n，则$[X]_{移}$= 0, $\overline{X_1}\,\overline{X_2}$…… $\overline{X_n}$ + 1。

（1） 0 的表示：0 的移码表示形式是唯一的

 [+0]移＝ 10…0 [-0]移＝ 10…0

 （2）表示范围：对于 n＋1 位移码机器数 X，它所能表示的数据范围为：

 定点整数：－2n≤X ≤ 2n－1

四种定点机器数的对比　　1、表示方法

定点整数

真值	$X=+X_1X_2\ldots\ldots X_n$	$X=-X_1X_2\ldots\ldots X_n$
原码	$[X]_{原}=0\ X_1X_2\ldots\ldots X_n$	$[X]_{原}=1\ X_1X_2\ldots\ldots X_n$
反码	$[X]_{反}=0\ X_1X_2\ldots\ldots X_n$	$[X]_{反}=1\ \overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}$
补码	$[X]_{补}=0\ X_1X_2\ldots\ldots X_n$	$[X]_{补}=1\ \overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+1$
移码	$[X]_{移}=1\ X_1X_2\ldots\ldots X_n$	$[X]_{移}=0\ \overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+1$

定点小数

真值	$X=+0.X_1X_2\ldots\ldots X_n$	$X=-0.X_1X_2\ldots\ldots X_n$
原码	$[X]_{原}=0.X_1X_2\ldots\ldots X_n$	$[X]_{原}=1.X_1X_2\ldots\ldots X_n$
反码	$[X]_{反}=0.X_1X_2\ldots\ldots X_n$	$[X]_{反}=1.X_1X_2\ldots\ldots X_n$
补码	$[X]_{补}=0.X_1X_2\ldots\ldots X_n$	$[X]_{补}=1.\overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+0.00\ldots\ldots1$
移码	$[X]_{移}=1.X_1X_2\ldots\ldots X_n$	$[X]_{移}=0.\overline{X_1}\ \overline{X_2}\ldots\ldots\ \overline{X_n}+0.00\ldots\ldots1$

真值	X=+ 0	X=- 0	
原码	$[X]_{原}=0\ 00\ldots\ldots0$	$[X]_{原}=1\ 00\ldots\ldots0$	不唯一
反码	$[X]_{反}=0\ 00\ldots\ldots0$	$[X]_{反}=1\ 11\ldots\ldots1$	不唯一
补码	$[X]_{补}=0\ 00\ldots\ldots0$		唯一
移码	$[X]_{移}=1\ 00\ldots\ldots0$		唯一

n+1位机器数

	定点整数	定点小数
原码	$-(2^n-1)\le X\le 2^n-1$	$-(1-2^{-n})\le X\le 1-2^{-n}$
反码	$-(2^n-1)\le X\le 2^n-1$	$-(1-2^{-n})\le X\le 1-2^{-n}$
补码	$-2^n\le X\le 2^n-1$	$-1\le X\le 1-2^{-n}$
移码	$-2^n\le X\le 2^n-1$	$-1\le X\le 1-2^{-n}$

❖ $[X]_{原}$→真值?

❖ $[X]_{原}=X_S,X_1X_2...X_n$

- $X_S=0$：$X=+X_1X_2...X_n$
- $X_S=1$：$X=-X_1X_2...X_n$

❖ $[X]_{反}$→真值?

❖ $[X]_{反}=X_S,X_1X_2...X_n$

- $X_S=0$：$X=+X_1X_2...X_n$
- $X_S=1$：$X=-\overline{X_1}\,\overline{X_2}\cdots\overline{X_n}$

❖ $[X]_{补}$→真值?

❖ $[X]_{补}=X_S,X_1X_2...X_n$

- $X_S=0$：$X=+X_1X_2...X_n$
- $X_S=1$：$X=-(\overline{X_1}\,\overline{X_2}\cdots\overline{X_n}+1)$

❖ $[X]_{移}$→真值?

❖ $[X]_{移}=X_S,X_1X_2...X_n$

- $X_S=1$：$X=+X_1X_2...X_n$
- $X_S=0$：$X=-(\overline{X_1}\,\overline{X_2}\cdots\overline{X_n}+1)$

浮点数格式

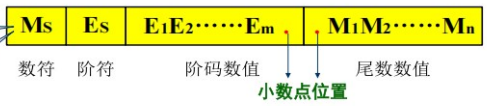
❖ 浮点数N的构成：$N=M\times R^E$

❖ 阶码E（Exponent）、尾数M（Mantissa）和阶码的底R（Radix）

❖ 阶码的底R：是一个常数，隐含规定，一般为2、8或16

为什么将数符放置在首位?

将数符放置在首位，方便比较大小



❖ 浮点数格式要素：

- ①位数：阶码位数+尾数位数→总位数
- ②编码：阶码和尾数采用的机器数编码
- ③排列顺序：数符是否在最高位
- ④其他特殊的编码规则：譬如隐藏位

❖ 尾数M：定点小数

- 尾数的位数：决定了浮点数有效数值的精度
- 尾数的符号：数符，代表了浮点数的正负
- 尾数的编码：一般采用原码和补码表示

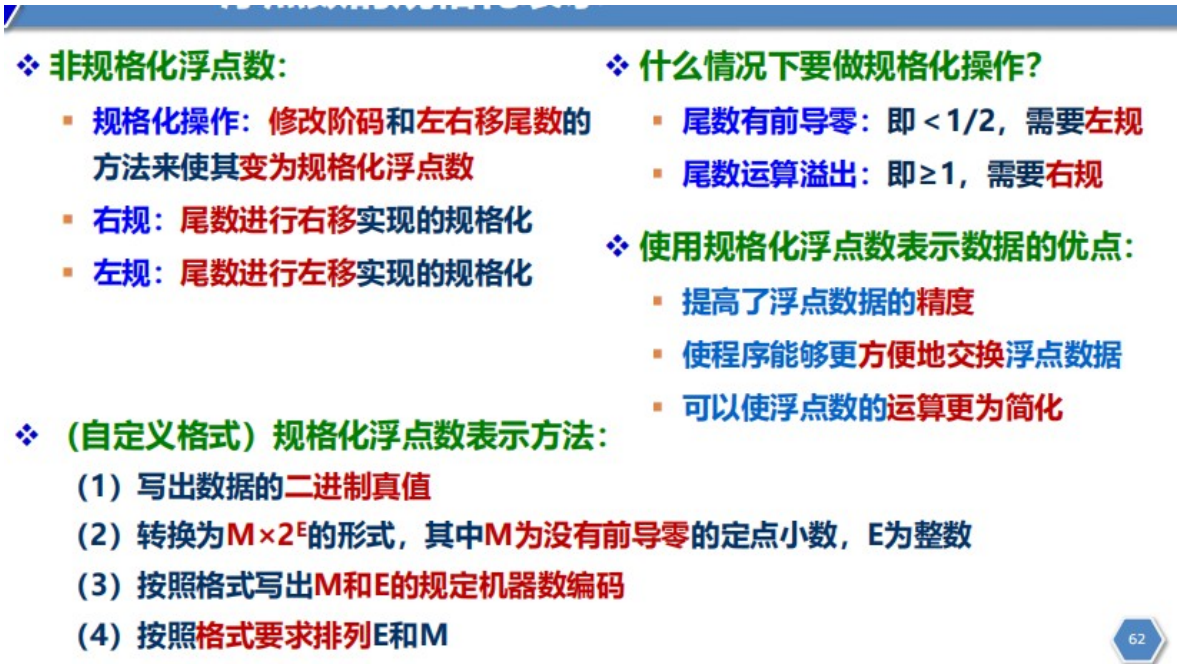
❖ 阶码E：定点整数

- 阶码的位数：多少决定了浮点数的表示范围。
- 阶码的数值：大小决定了数据实际小数点位置与尾数的小数点位置之间的偏移量
- 阶码的符号：阶符，决定了实际小数点位置在尾数的小数点位置的左边还是右边
- 阶码的编码：一般采用移码和补码表示

❖ 阶码的底R：一般为2、8或16，且隐含规定

$0.101110\times2^{-2}=0.0010111$

$N=M\times R^E$



浮点数的运算过程中，尾数发生溢出（超出尾数所能表示的范围），进行右规。
右规一次即可。
阶码发生溢出时，浮点数被判为溢出

1. 一些小结论

由于补码表示的最大值和最小值是非对称的
即最大值的绝对值(0111)的原码比最小值的绝对值的(1000)原码要小
对于 n 位定点整数的机器数则为（n+1）位（算上符号位）
对于一个负数 X 来说
[X]原+[X]补=2^n
Eg.[-2]原=1010；[-2]补=1110，相加为 1000（当成原码并去掉符号位，就是 8）
　负数的补码就是其绝对值的补数
X 的绝对值+[x]补但是[x]补当成原码去掉符号位，为 10000 即 2^（n+1）
还是上个例子。[-2]补=1110=14，2+14=16；
所以，有符号数和无符号数之间的转换规则是：
位模式不变，但是解释这些位的方式改变了
数值数据的表示
进位计数制
不同数制之间的转换
十进制的编码
机器数

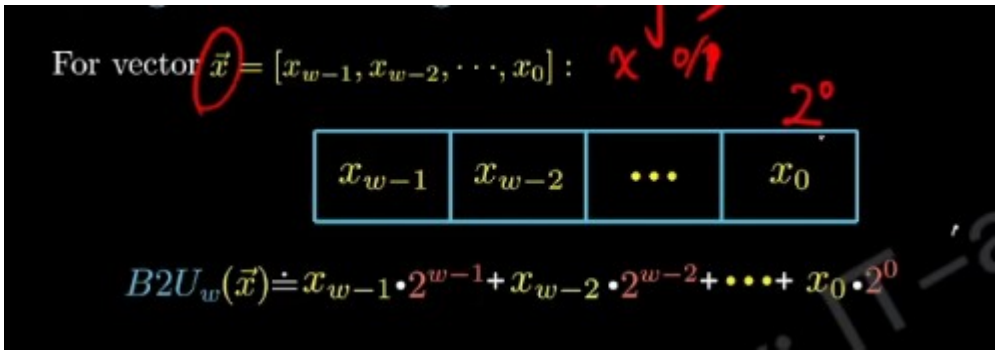
数值数据在计算机中的表示形式称为机器数。机器数的特点是：

- 机器数：数值数据在计算机中的二进制表示形式。
  - 机器数的符号位必须被数值化为二进制0和1
  - 机器数的小数点用隐含规定的方式来表达
  - 表示的数值范围受计算机字长的限制
  - 编码方法：不同的排列方式，常见的有原码、反码、补码、移码。
- 真值：机器数真正表示的数值数据
- 书写：+/ - 符号 数值（二进制或十进制）

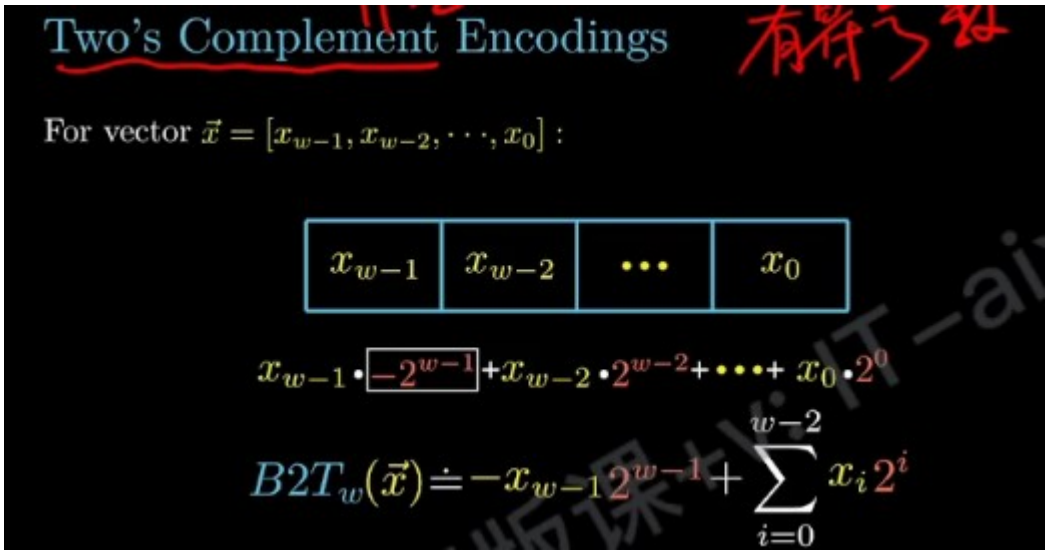
20

定点机器数的表示方法

一个字节由 8 个位组成
我们把这种按照一位一位表示数据的方式称为位模式
对于无符号数编码
在机器中的解释，采用原码表示法（？）



对于有符号数的编码，采用补码



对于相同的位模式，无符号数和有符号数的数值关系如图所示：
（这里的 B2U/B2T 是位向量到无符号数和有符号数的函数映射）
最高位理解成负权重-2^w-1

$$B2U_w = x_{w-1} \cdot 2^w + B2T_w$$

用 T2U 来表示有符号数到无符号数的函数映射

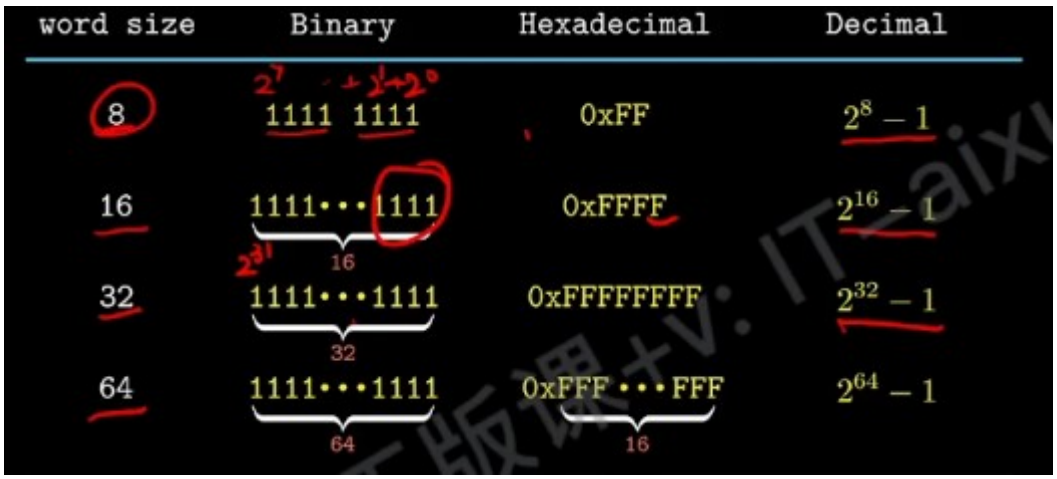
$$T2U_w(x) = \begin{cases} x + 2^w, x < 0 \\ x, x > 0 \end{cases} \text{，即} T2U_w(x) = (x + 2^w)\ mod\ 2^w$$

$$U2T_w(u) = \begin{cases} x, u \le TMax_u \\ x - 2^w, u > TMax_u \end{cases}$$

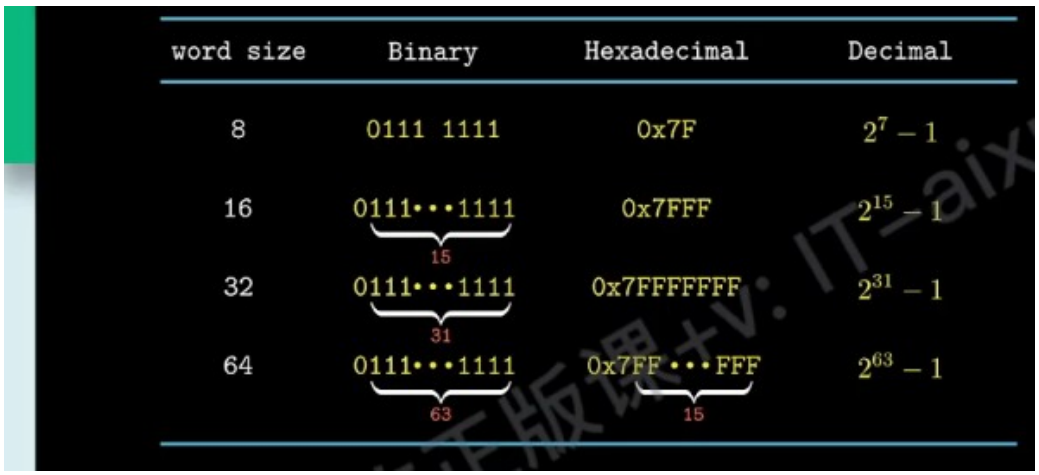
W 位(包含了符号位！)补码表示的数值

$$\omega = -x_{\omega-1}2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i 2^i$$

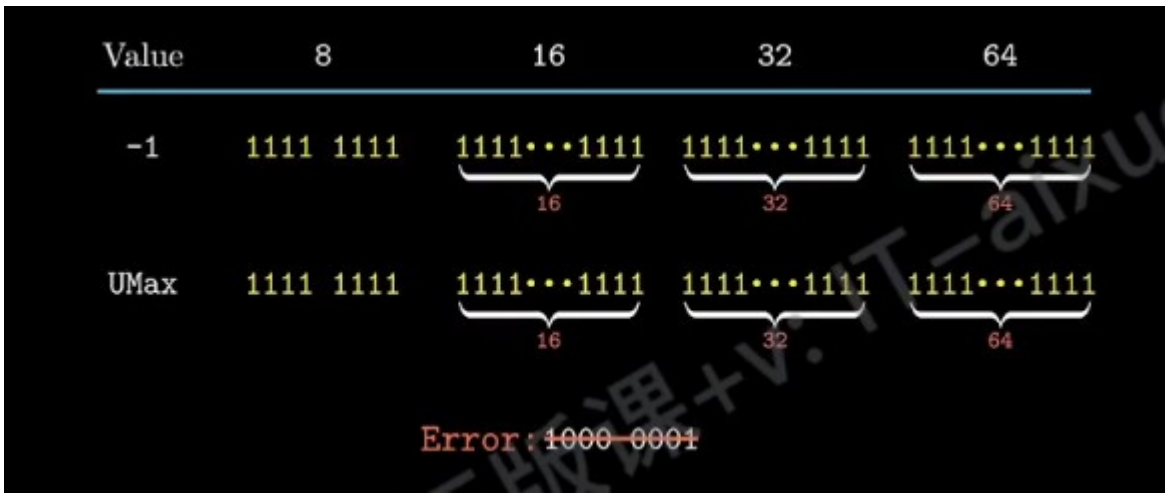
也就是说，把第一位作为负数-2^n 来处理；正数为 0,负数为 1；
你可能已经发现，二进制数的“互补数”编码规则和补码的定义非常的像，其实不是像，而是它们本来就是等价的。也就是“对应正数的互补数”等于“对应正数按位取反再加 1”。下面尝试证明。
原命题精确描述为：相对于 w 位的二进制数，整数 b∈(0,]，即 b∈(0,]，等式 -b = ~b+1 成立。
原码、补码、反码、移码的直接二进制值（机器数的形式值）各自有什么作用?
反码、补码的真值为负数的时候机器数形式值（符号位作为 2^n 而不是正负号）减去对应的 2^(n+1)或者是 2^（n+1）+1 就是这个机器数的真值。
移码是无论正负直接减去 2^(n+1)
而原码没有类似的结果



有符号数最高位为 0（正数），其余为 1



有符号数的-1 和无符号数的最大值一样编码表示



2．真值与补码的转换

真值->补码
正数的补码：去掉+号前面加 0。
负数的补码：1、去掉 - 号前面加 1，从右到左找到第一个 1，左边全部取反。
2、负数的补码是将原码的符号位保持不变,其余各位取反，然后加 1,就得到其补码;
计算一个负数的补码时，通常遵循以下步骤：

1. **取绝对值的二进制表示**：求出绝对值（即对应正数），求出数值的二进制形式。
2. **取反**：将二进制数的每一位取反，即 0 变为 1，1 变为 0。
3. **加 1**：在取反结果的基础上加 1，得到最终的补码表示。

也就说，符号位参与了运算。在第 2 步实际上已经将“符号位为 1“表示成 0-1
实际上，这个符号位啊，你无论是先不管符号位取反再上符号位为 1，还是一起从绝对值取反（从 0 变成 1—），不都是符号位为 1 吗？但参与+1 运算是必须的。
补码->真值
符号位 0 的补码的真值：去掉 0 前面加+号。
符号位 1 的补码的真值：
1、 去掉 1 前面加-号，从右到左找到第一个 1，左边全部取反。
2、负数的原码就是它的补码的补码。
3、 如果补码的符号位为“1”，表示是一个负数，那么补码减 1,再将符号位保持不变其余各位取反

3.对于(X+2^(n+1)) mod (2^(n+1))的解释

对某两个整数 a，b，若它们除以正整数 m 所得的余数相等，则称 a，b 对于模 m 同余，
也就是严格来说，存在整数 k 使得
a−b=km，则称 a,b 对于除数 m 是同余的。一般记做
a≡b(modm)
在模运算中，一个数与它除以“模”后得到的余数是等价的，如 A、B、M 满足 A=B+K×M(K 为整数)，则记为 A≡B(mod M)，即 A、B 各除以 M 后的余数相同。在补码运算中，[A]补-[B]补=[A]补 + M-[B]补，而 M-[B]补 = [-B]补，因此补码可以借助加法运算来实现减法运算。从模运算的角度，补码的表示利用了以下规律：
模运算的同余性质：对于任意整数 a 和模数 m，有 a≡a+km(mod m)，其中 k 为整数。这意味着在模 m 的运算中，数值是循环的，超过 m 会回到起点。
为何常见写法是“x+2^n+1 mod 2^n+1”而不是 x mod 2n+1
数论上，“x 与 x+2^{n+1}x”在模 2^{n+1} 下等价；
但在说明“补码如何把负数映射到二进制的高半区”时，**显式地加上 2^{n+1}** 更能直观反映“负数被搬到了高位=1 的区间”这个事实。

 非负数直接写成“它自己” (因为落在小半区间[0,2^n))；
 负数写成“X + 2^{n+1}” (因为这样就落在大半区间 [2^n,2^{n+1}-1])。
在计算机系统中，n 位定点整数（包括符号位）补码的取值范围是 −2^n 到 2^n - 1。
这意味着总共有 2^{n+1} 个不同的数值可以表示。因此，模数 2^{n+1} 对应于这些数值的总数量。
负数的表示：在模 2^{n+1}的系统中，负数 X 可以表示为 2^{n+1} + X,因为 X 为负数，会落在 2^(n) 到 2^{n+1} - 1 的范围内，符合无符号数的表示范围。
因为模为 2^(n+1), k=1，那么就可以将-2^n-2^n-1 统一成(X+2^(n+1)) mod (2^(n+1)).
减法（加上一个负数）的运算用加法代替了，我们说的补码（反码）实际上是数对于一个模的同余数。
（这里的取模运算，请让我们抛弃符号数，eg.-2 mod 8=6 ,[6]=110,而 [-2]补=1110，当成原码，去掉符号位）
规定：把 0 当成正数，也即+0，这样 0 的编码就变成：0，0000000。那 8 位二进制表示的正数范围仍然是： +0 —— +127。
但是，对于负数就必须要做调整，也即-0 必须要让位---1，0000000 这个编码不能表示-0。我们可以把负数整体向后“挪动 1 位”：只要将 8 位二进制表示的负数范围从：-127 —— -0 变成：-128 —— -1，就能成功解决问题。
-0 的补数是 256-0=256=0x100，由于是“8 位二进制数”，所以-0 的补数是 0x00；‘我们知道“-1=0-1”，所以
-1 的补数是 256-1=255=0xFF；这就好比是"0-1"不够减，"借 256"得到的（这里借的是模数值）。
同理，可以-2~-127 的补数；
注意：这里的补数，就是计算机所讲的补码。
-127 的补数是 256-127=129=0x81；
-1~-127 的补码范围：0xFF~0x81；
负数-128 的补码：
我们知道“-128=0-128”，“-128=-1+-127”所以
-128 的补数是 256-128=128=0x80；所以-128 的补码为 0x80，但是-128 没有原码和反码。
所以人为地规定 10000000 表示-128
总结：
补码补码，就是补上去一个模对应的补数

小结

在一个区间长度为 M 内，统一正负数的常用方法是
X' = (X + M) mod M
其中 M 就是模数。如果你换成一个非 M 的倍数 N，则只有在 N≡0 (mod M)的情况下才能达到同样效果。因此，负数加上的部分必须是 M 的倍数，通常取最小正倍数 M 来实现统一。
可以这么理解：如果一个区间（或集合）恰好包含 m 个整数，并且这 m 个整数构成了一个模 m 的完全剩余系，那么对任意整数 X，用
X mod m

总能得到落在这个集合中的唯一代表。换句话说，只要这个集合包含的元素个数为 m（而且互不同余），那么无论 X 是正数还是负数，都有
X≡Xmodm
而且 Xmodm 就是该模 m 意义下的标准表示。

为什么定点小数的模 2?

至于模数为 2 的说法，这是因为补码表示中，数值的表示是循环的。例如，-1 和 1 - 2^(-n) 之间的差值正好是 2 - 2^(-n)，但由于补码的循环特性，这个差值在模 2 的意义下是等价的。-2^(-n) 并没有被省略，而是补码表示的范围已经包含了从 -1 到 1 - 2^(-n) 的所有可能值。补码的模数为 2 是因为补码表示中，数值的表示是循环的，且循环周期为 2。

关于数的扩展

基本上是这样的。对两类数来说：

- **高位扩展（符号扩展）**：
- **低位扩展（增加精度）**：

定点整数的符号扩展：
对于无符号数，称为零扩展
对于有符号数，称为符号位扩展
在原符号位和数值位中间添加新位，正数都添 0，负数原码添 0，负数反、补码、移码添 1。

$$B2T_w([x_{w-1},x_{w-2},...,x_0]) \quad (1)$$

$$B2T_{w+k}([x_{w-1},\cdots,x_{w-1},x_{w-1},\cdots,x_0]) \quad (2)$$

$$B2T_{w+1}-B2T_w=0$$

定点小数的符号扩展：
在原符号位和数值位后面添加新位，正数都添 0，负数原、补码、移码添 0，负数反码添 1。
低位截断
将 int 类型强制转换为 short 类型，高 16 位被丢弃
将一个 w 位的无符号数，截断成 k 位的方法是，丢弃最高的 w-k 位。截断操作可以对应于取模运算。

$$B2U_w([x_{w-1},x_{w-2},...,x_0]) \ mod \ 2^k = B2U_k([x_{k-1},x_{k-2},...,x_0])$$

截断有符号数
先转换成无符号数，再截断，再转换成有符号数
即

$$T2U_w(x)=\begin{cases} x+2^w, x<0 \\ x, x>0 \end{cases}，即 T2U_w(x)=(x+2^w) \ mod \ 2^w$$

$$B2T_k([x_{k-1},...,x_0])=U2T_w([(B2T_w([x_{w-1},x_{w-2},...,x_0])+2^w) \ mod \ 2^w] \ mod \ 2^k)=U2T_w(\ B2U_w([x_{w-1},x_{w-2},...,x_0]) \ mod \ 2^k)$$

4、其他运算符

布尔运算（位级运算）

~/NOT 非（取反）（单目运算）
&/AND 与
|/OR 或
^/EXCLUSIVE-OR 异或
取反的时候：
因此，可以总结出~按位取反的计算结论是：~n = -(n+1)
例如本例中，~5 = -(5+1)，即~5 = -6

逻辑运算

所有非零参数为 true，参数 0 表示 false
结果只有两个，true 与 false 惰性运算
|| OR
&& AND
! NOT

移位运算

逻辑移位。将移位的数据视为无符号数据，移位的结果知识数据位置发生了变化。移出的数据位一般置入 CF 标志位(进位、借位 标志)
算术移位，对象是有符号数，算术移位的结果在绝对值上进行放大缩小；符号位保持不变。计算机的算术移位指令大多采用补码的 移位规则
循环移位，左移时最高位移入最低位，右移时最高位移入最低为，若和 CF 标志位一起循环，则称为大循环，否则小循环
左移相当于乘以 2，右移相当于除以二
CF: 进位/借位标志。最近的操作使最高位产生了进位。可用来检查无符号操作的溢出。 加法运算时 C=1,则 CF 置 1 表示进位；减法运算时 C=0,CF 置 1 表示借位。CF 只对无符号数运算有意义。

ZF: 零标志。最近的操作得出的结果为 0（全零）则 ZF 置 1。
SF: 符号标志。记录运算结果的符号，它与运算结果的最高位相同。采用补码表示，运算结果为正数置 0，负数置 1。
OF: 溢出标志。最近的操作导致一个补码溢出——正溢出或负溢出。有符号数有意义。有溢出则置 1，否则置 0；
PF:奇偶标志位。操作数 1 的个数为偶数置 1，否则置 0；

左移

逻辑左移，高位移出，低位补 0
左移不区分算术还是逻辑
对于原码的算术左移，符号位不变，高位移出，低位补 0；
当左移移出的数据位为“1”时，发生溢出。
对于补码的算数左移。符号位不变，高位移出，低位补 0；当左移移出的数据位正数为 1 负数为 0 时（符号位与被移出的位不同），发生溢出。一次。为保证不发生溢出，移位数据的最高有效位必须与符号位相同
因此，在硬件补码实现算数左移时，直接将数据的最高有效位移入符号位，当不发生溢出时，不会改变机器数的符号

右移

逻辑右移，低位移出，高位补 0
对于原码的算数右移，符号位不变，低位移出，高位补 0
算数右移
对于补码，符号位不变，低位移出，高位补符号位
对于原码，符号位不变，低位移出，高位补 0

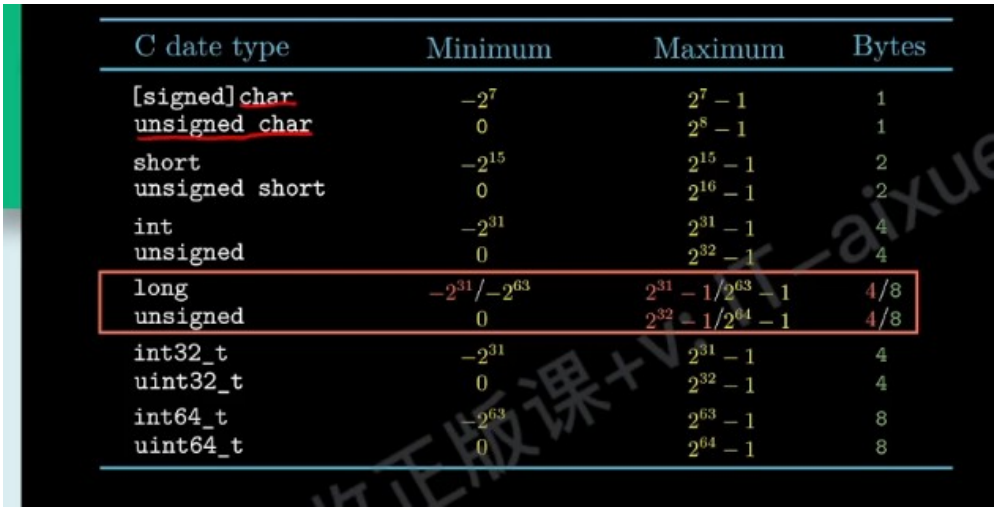
C 语言中一个特性就是支持按位进行布尔运算（位运算）
位运算的用法就是实现掩码运算
特定的位序列
比如说和 0xFF，进行与运算，就是其本身

5、机器数编码与真值

为什么原码、补码、反码、移码 作为机器数的时候，符号位算作 2^n?
关于移码

以 8 位数为例，总共能表示 256 个数，把这些数加上一个固定的值，从而形成新的码值（还是 8 位），这就是移码的定义。例如，可以把有符号数映射到 0 - 255 的段上。
移码的优势：原来的大小关系不变，且一一对应（没有两种表示的 0），很容易判断大小（如判断是最小值或者最大值）
64 为机器上，不同的数据类型，所占的字节数不同，数值范围取值不同
Long 在 64 位机器上占 8 个字节 32 位机器上，long 占 4 个字节
Unsigned 关键字 数字只能为非负数，无符号数

C data type	Min	Hexadecimal	Max	Hexadecimal	Bytes		
[signed] char	-2^7	0x80	2^7-1	0x7F	1		
Unsigned char	0		2^8-1	0xFF	1		
Short	-2^15	0x8000	2^15-1	0x7FFF	2		
Unsigned short	0		2^16-1	0xFFFF	2		
Int	-2^31	0x8(7 个 0)	2^31-1	0x7(7 个 F)	4		
Unsigned int	0		2^32-1	0x(8 个 F)	4		
Long	-2^63	0x8(15 个 0)	2^63-1	0x7(15 个 F)	8		
Unsigned long	0		2^64-1	0x(16 个 F)	8		
Long long							
Int32_t							
Uint 32_t							
Int 64_t							
Uint64_t							
Float							
Double							
Long double							



假设一个整数的数据类型有 w 位，用向量 x 表示

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \cdots, x_0]$

$$B2U_w(\vec{x}) \doteq x_{w-1} \cdot 2^{w-1} + x_{w-2} \cdot 2^{w-2} \ldots$$

B2U 的意思是 binary to unsigned
对于有符号数的编码采取补码的方式
$$B2T_w(\vec{x}) \doteq x_{w-1} \cdot -2^{w-1} + x_{w-2} \cdot 2^{w-2} \ldots$$

这里最高位为 1 时表示负数，最高位为 0 时表示非负数
最高位也叫做符号位
关于符号位，需要理解负权重的概念，而不能简单的当成一个负号
记住对于无符号数，最高位的 1 表示的不是权重
也就是说，-2^3=-8;
-1 在有符号数全为 1，但在无符号数为最大值
如果在计算的时候一个是有符号数，一个是无符号数，
C 语言会隐式地将有符号数转换为无符号数

6、溢出与取模运算的数学关系

设我们有两个 w 位无符号整数 a 和 b，其和为 S，即
S = a + b
由于二进制表示限定为 w 位，系统只能表示 0 到 2w−1 内的数。因此，任何 S 超出这一范围的部分都会被丢弃。用数学语言描述就是：实际存储的结果 R 为
R=S mod 2^w
也就是说，在加法运算中，丢弃溢出位的操作等同于计算 S 对 2^w 取模。这正是计算机硬件中处理无符号整数溢出的方式
很多人可能会疑惑，既然阶码字段有 8 位，那么为什么不选择 2^7 = 128 作为偏移量？原因在于：

- **对称分布的需求**：以 127 为偏移量，可以使得存储阶码为 127 时实际指数为 0。这种设计使得正指数和负指数在规格化数中分布较为对称，便于硬件实现和算法设计。
- **历史约定与标准化**：IEEE754 标准规定的偏移量就是 127，它的选择经过广泛验证并且与现有的硬件架构和软件设计高度契合。采用 127 而非 128 使得编码与译码的过程更加简便，同时避免了 0 不对称的问题。
- 具体来说，设若采用 128 作为偏移量，则当存储阶码 E 为 128 时，实际指数为 0；但这会使得整个指数范围整体右移，从而使得可表示的最小负指数与最大正指数的数量不均衡，不符合 IEEE754 对于规格化数的要求。选择 127 则能使得 0 成为一个中心值，既便于表达接近 1 的数值，又能合理覆盖较大范围的指数。

5.3. 关于阶码真值为 -127 和 128
一般我们正常探讨的阶码区间是 -126D ～ 127D，而真值 -127D 的阶码为 0000 0000B，真值 128D 的阶码为 1111 1111B。

5.3.1. 阶码真值为 -127
当阶码全为 0 ，尾数不全为 0，表示 非规格化小数 ，用来表示比最小绝对值还要小的数，即
尾数码 隐含的最高位不是 1，而是 0；
阶码真值 固定为 -126，而非 -127；
当阶码全为 0 ，尾数全为 0，表示 真值 +/- 0 ；

5.3.2. 阶码真值为 128

1. 当**阶码全为 1** ，**尾数全为 0**，表示 正负无穷大 +/- ∞
2. 当**阶码全为 1** ，**尾数不全为 0**，表示非数值 NaN（Not a Number）

如 0/0，∞-∞ 等非法运算的结果即为 NaN
汉字输入法程序：将汉字输入计算机转换成汉字内码
1、 输入—汉字输入码，也称作外码， 由西文字符编码而成（拼音编码、字形编码等）
2、 存储、交换和检索—内码 ，两个字节表示。汉字内码在计算机中是唯一的。（区位码、国标码、汉字内码 2 个字节最高位均为“1”）
3、 交换—交换码，是指不同的具有汉字处理功能的计算机系统之间在交换汉字信息所用的代码标准。
4、 输出——字形码 点阵表示法 字模码：用于汉字的显示和打印
汉字字库———所有汉字的字模点阵代码按顺序集中存放根据内码查表
字形检索程序：将汉字的机内代码转化为汉字的字形码

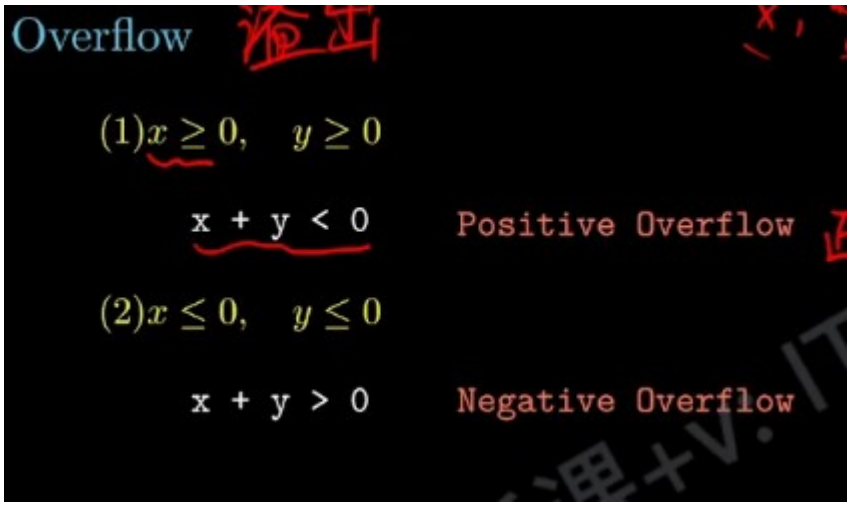
整数的运算
计算值：x+y
实际值：计算机中实际上保存的结果

$$For\ x\ and\ y\ ,x+{}_{w}^{u}y=\begin{cases}x+y,x+y<2^{w}\\x+y-2^{w},2^{w}\leq x+y<2^{w+1}\end{cases},\text{其中}{}_{w}^{u}y\text{表示}w\text{位的无符号整型}$$

溢出时，计算值=实际值+2^w（这里相当于是原码！）
如何判定运算结果是否发生了溢出

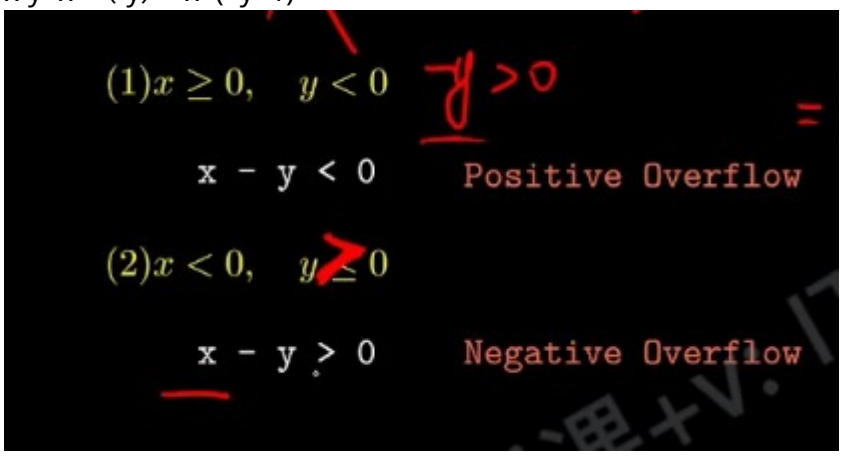
```
Int uadd_ok(unsigned x,unsigned y)
{
unsigned sum=x+y;
If (sum>=x)
Return 1;
else
return 0;
}
```

有符号整数
为了避免数据大小的扩张，最终结果将截断为 w 位；即实际上数字只占 w-1 位
两个异号的数字肯定不会溢出
两个同号的才有可能溢出



正溢出/上溢
负溢出/下溢
减法转换成加法来做

x-y=x+（-y）=x+(~y+1)



$$For\ x\ and\ y, x +_w^t y = \begin{cases} x+y-2^{w-1}-2^{w-1} = x+y-2^w, 2^{w-1} \leq x+y, \text{正溢出} \\ x+y, -2^{w-1} \leq x+y < 2^{w-1} \\ x+y+2^w, x+y < -2^{w-1}, \text{负溢出} \end{cases}$$

对于正溢出，正溢出实际上是让符号位从 1 变成 0，计算值=实际值+2^w,它没被截断，最高位由 0 变成了 1，这两个数本来就没有-2^（w-1)的权重结果有了，那么实际值就要加上 2^（w-1），又因为补码的定义，又得减去 2^(w-1)
对于负溢出，计算值=实际值-2^w，这里因为原来两个最高位符号位从 1 变成 0，两个数本来有-2^w 的权重结果没了。那么实际值就要减去个-2^w.
如何判断溢出?
两个正数相加，得到结果为负，正溢出
两个负数相加，得到结果≥0，负溢出

加法逆元
$For\ x, 0 \leq x < 2^w, x + x' = x' + x = 0$

y-x->y+x'.
对于无符号整型来说，x+x'=2^w=0

$$x' = \begin{cases} x, x = 0 \\ 2^w - x, x > 0 \end{cases}$$

对于补码表示的有符号数的逆元比较简单

$$x' = \begin{cases} -x, x > TMin_w \\ TMin_w, x = TMin_w \end{cases}$$

由于补码最大值比最小值的绝对值要小
即 0111<1000
关于 最小值的逆元要通过负溢出 的方法来实现
即
$T\min_w + T\min_w = -2^w = 0$

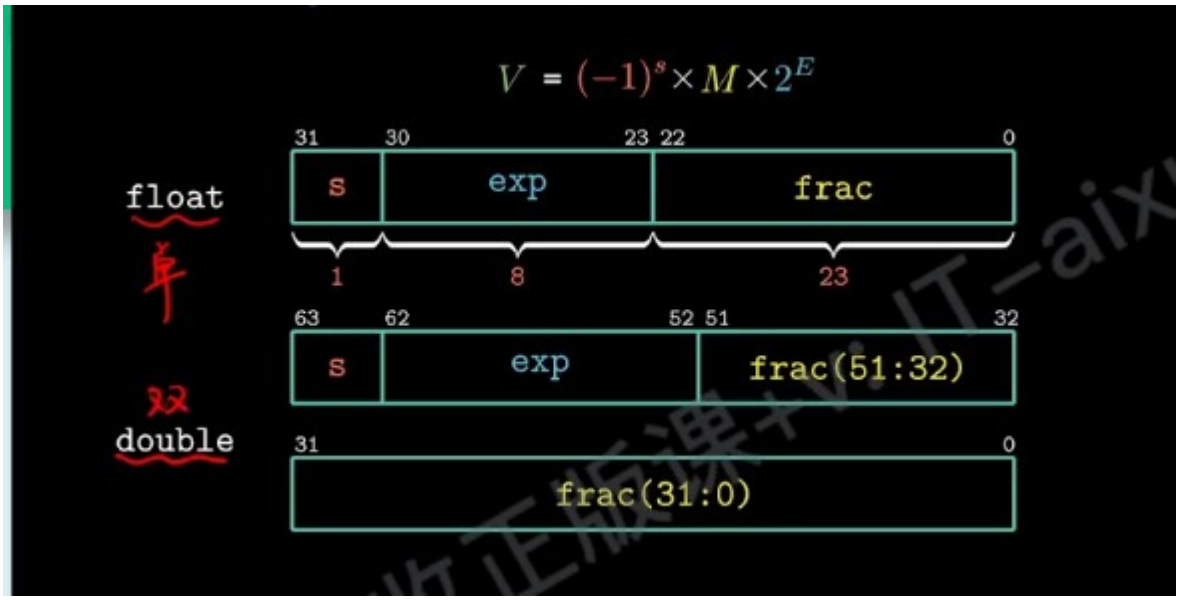
X、y 都为 w 位，乘积结果可能为 2w 位，c 语言定义无符号数乘法所产生的结果是 w 位
因此，z=xy mod 2^w
对于有符号数，我们有 z=U2T(xymod2^w)
X'y' mod 2^w=(xy) mod 2^w.即有符号整数和无符号整数高位截断后结果的位级表示相同
整数的乘法往往利用移位运算（拆解成与多个 2^k 相乘再相加减）
整数的除法遇到除不尽的情况，总是向 0 舍入
但是，移位的时候会导致向下舍入。为此，我们要在移位之前加入偏置。
即如果要右移 k 位，在低 k 位加上 2^k-1, 向 k+1 产生一个进位，实现了向 0 取整。
即用整除实现了向上取整
举个例子：
K=0
1100111111001100 -12340
K=4 1111110011111100 -772 而 12340/2^k=-771.25;
加上偏置，1100111111011011，右移四位有 1111110011111101,即-771。

$$division\ by\ a\ power\ of\ 2$$

$$(x < 0 ? x + (1 << k) - 1 : x) >> K$$

但是，除以 2 不能推广到除以任意整数
浮点数

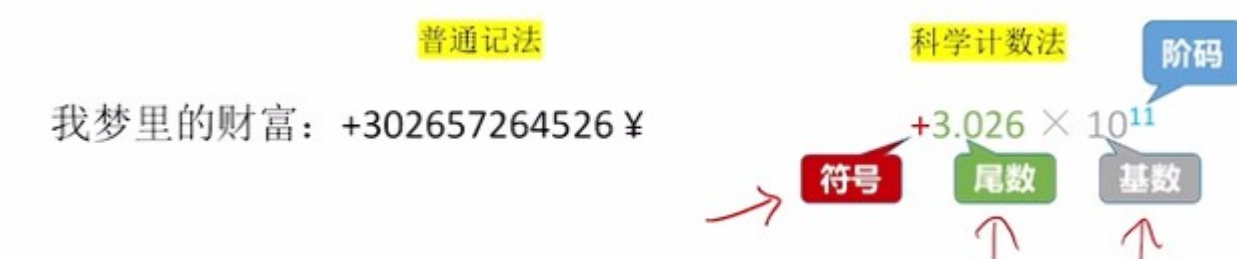
类型	总　位数	尾数	阶码	真值计算		
短　实数	32	24	8	$N=(-1)^{M_s}\times(1.M_1M_2...M_n)\times2^{E-127}$		
长　实数	64	53	11	$N=(-1)^{M_s}\times(1.M_1M_2...M_n)\times2^{E-1023}$		
临　时实数	80	65	15			



符号位 阶码 尾数

7、IEEE754

Float32bit
Double64bit

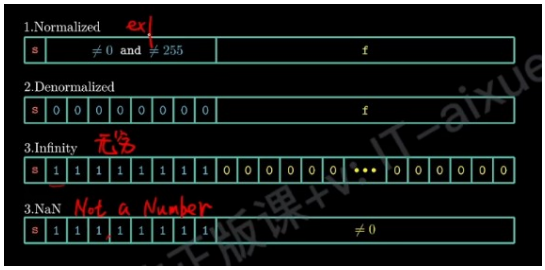


符号：决定数值的正负性
尾数：影响数值的精度。尾数的位数越多，精度越高
阶码：反映小数点的实际位置
基数：K进制通常默认基数为K
规格化：确保尾数的最高位非0数位刚好在小数点之前

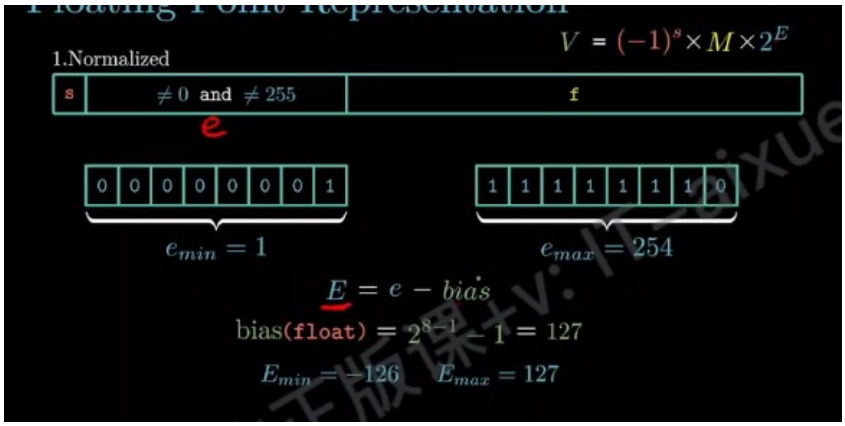
阶码确定范围
尾数影响精度

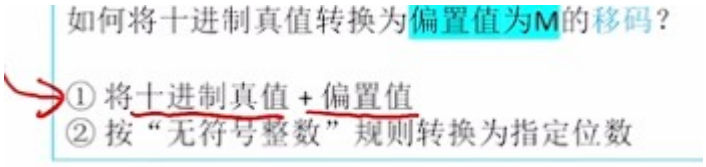
$$V=(-1)^S\times M\times 2^E$$

尾数用原码定点小数表示
阶码用移码，规定偏置值为 127
s=0,正数；s=1 负数；M 为小数部分，尾数部分，frac。E 为指数部分，阶码，exp。
Emin=1.emax=254，对半分，所以要减去 254/2=127；
第一位总是 1，因此没必要显示出来，所以尾数部分要加 1

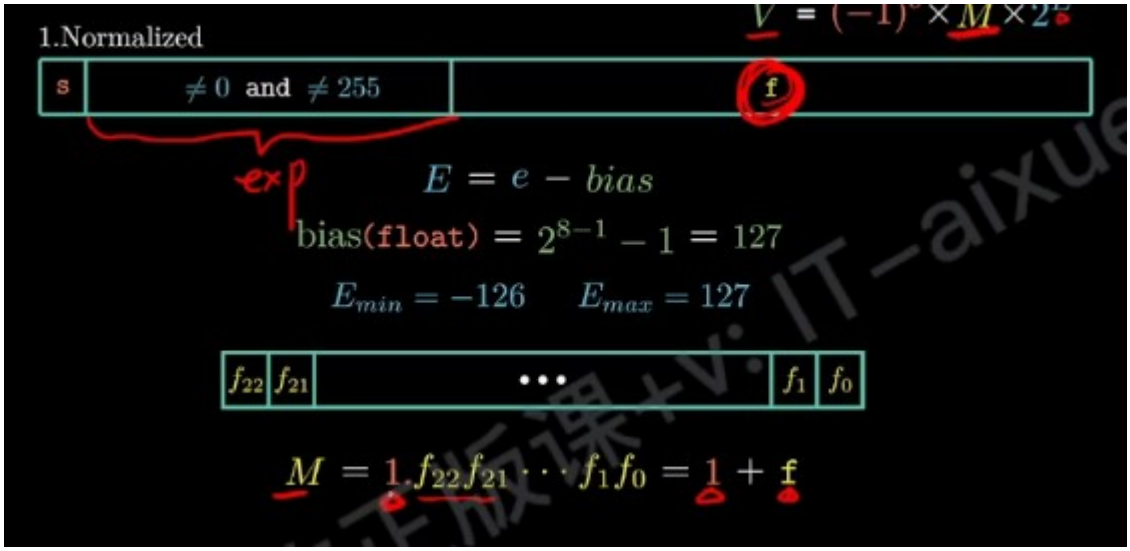


单精度记得减去 127e-bias

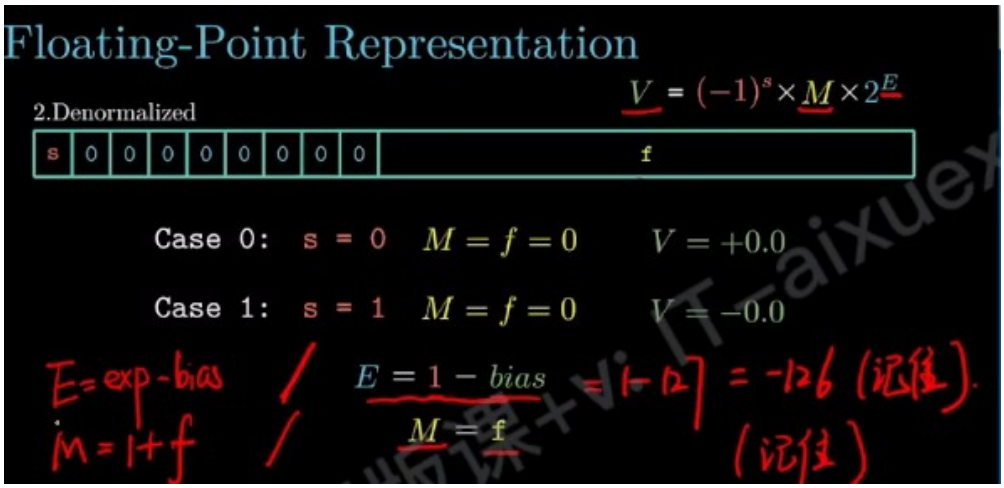




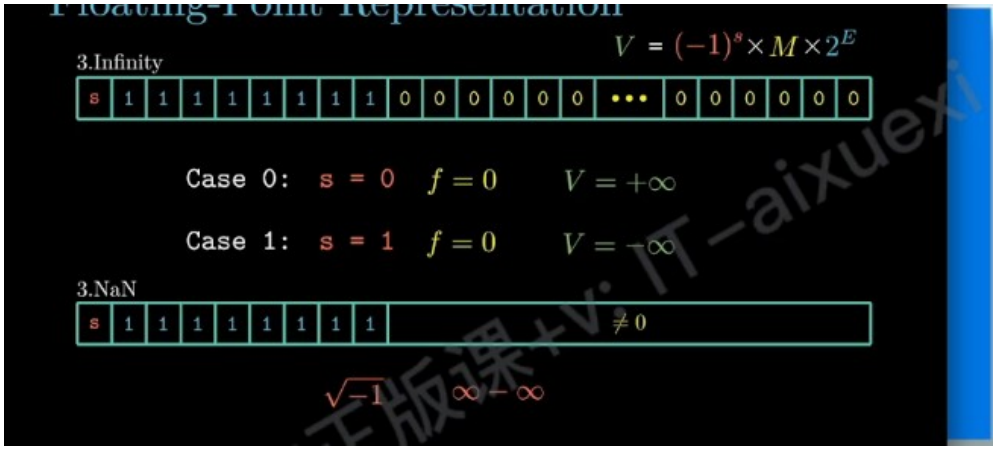
小数点前的 1 省略了（隐藏了 1）



1、 当阶码字段二进制位不全为 0，也不全为 1 时，（0 和 255），规格化
2、当阶码字段全为 0，非规格化.1、尾数部分也全为 0，则当 s=0，表示正 0，s=1，负 0；
表示非常接近 0 的数；如果尾数不为 0，则 E=1-bias，M=f 不加 1



3、当阶码字段全为 1，特殊值；当尾数为 0 时，s=0，正无穷；s=1，负无穷；当尾数不为 0 时，表示 NaN 表示不是一个数



单精度为例

Ms	E	M	意义
0/1	0	0	±0
0/1	0	非0	非规格化数
0/1	1~254	任意	规格化数
0/1	255	0	±无穷大
0/1	255	非0	NaN

NaN: 非数（not a number）

跳读	M	E	2^M
0-	00……00	00000000	1
绝对小非规非大量	11……11	00000000	1
绝对小非规非小量	00……00	00000000	1
绝对小非规	……	……	1
绝对小非规	……	……	1
绝对小非规小量	00……00	11111111	1
绝对小非规大量	11……11	01111111	1
∞-	00……00	11111111	1
NaN	10……00	11111111	1
NaN	……	11111111	1
NaN	……	11111111	1
NaN	11……11	11111111	1

跳读	M	E	2^M
0+	00……00	00000000	0
绝对小非规非小量	00……00	00000000	0
绝对小非规非大量	11……11	00000000	0
绝对小非规	……	……	0
绝对小非规	……	……	0
绝对小非规小量	00……00	11111111	0
绝对小非规大量	11……11	01111111	0
∞+	00……00	11111111	0
NaN	10……00	11111111	0
NaN	……	11111111	0
NaN	……	11111111	0
NaN	11……11	11111111	0

- 阶码：是2^n-1的移码，$E=[E']_{移}=2^n-1+E'$
- 隐藏位：单、双精度浮点数，尾数规格化的“1”，放置于整数位，并将其隐藏；临时实数无隐藏位
- 符号位：尾数的符号位Ms在最高位

隐藏位,位于整数位

类型	真值计算
单精度	规格化：$N=(-1)^{Ms}\times(1.M_1M_2...M_n)\times2^{E-127}$ 非规格化：$N=(-1)^{Ms}\times(0.M_1M_2...M_n)\times2^{-126}$
双精度	规格化：$N=(-1)^{Ms}\times(1.M_1M_2...M_n)\times2^{E-1023}$ 非规格化：$N=(-1)^{Ms}\times(0.M_1M_2...M_n)\times2^{-1022}$

❖ 例3.10　若X和Y均是IEEE 754标准的单精度浮点数。
- （1）　若X=-135.625，求X的规格化浮点数表示。
- （2）　若浮点数Y的存储形式为41360000H，求Y的真值。

❖ （1）真值→IEEE 754规格化单精度浮点数
❖ ①写出数据的二进制真值　　$X=(-10000111.101)_2$
❖ ②转换为$M'\times2^{E'}$的形式：$M'=\pm1.\times\times...\times$，E'为整数
- $X=-1.0000111101\times2^7$　　$M'=-1.0000111101$　　$E'=+7$

❖ ③写出尾数M'的原码，尾数数值隐藏（舍去）整数位的“1”
- $M_S=1$　　尾数数值M=000 0111 1010 0000 0000 0000

❖ ④写出阶码E'的移码E：$E=[E']_{移}=127+E'=127+7=134=1000\ 0110B$
❖ ⑤按照格式排列E和M：即Ms E M
- $[X]_{浮}=$ 1 10000110 000 0111 1010 0000 0000 0000B=C307A000H

72

❖例3.10 （2）浮点数→真值

阶码8位，2^n-1的移码

- ①浮点数的二进制编码：按照格式拆分，写出M_S、阶码E和尾数数值M
  - $[Y]_{浮}$=41360000H= 0100 0001 0011 0110 0000 0000 0000 0000 B
  - M_S=0　　E=1000 0010 B=130　　M=011 0110 0000 0000
  - 阶码编码E不是全0也不是全1，可判定Y是规格化浮点数
- ②写出尾数的真值M'：M_S=0，表明是正数　添加隐藏位1
  - M'=+1. 011 0110 0000 0000 0000 0000B
- ③求出阶码的真值E'：Y是单精度浮点数，故E'=E-127=130-127=3
- ④按照$M'\times2^{E'}$，求出浮点数的真值
- $Y=M'\times2^{E'}=+1.\ 011\ 0110\ 0000\ 0000\ 0000\ 0000\times2^3$
  $=+1011.011B=+11.375$

添加隐藏位1

数符+

尾数数值位23位原码

73

但是，浮点数只能近似表示实数运算

1、 向上舍入

2、 向下舍入

3、 向零舍入

4、 四舍六入五取偶

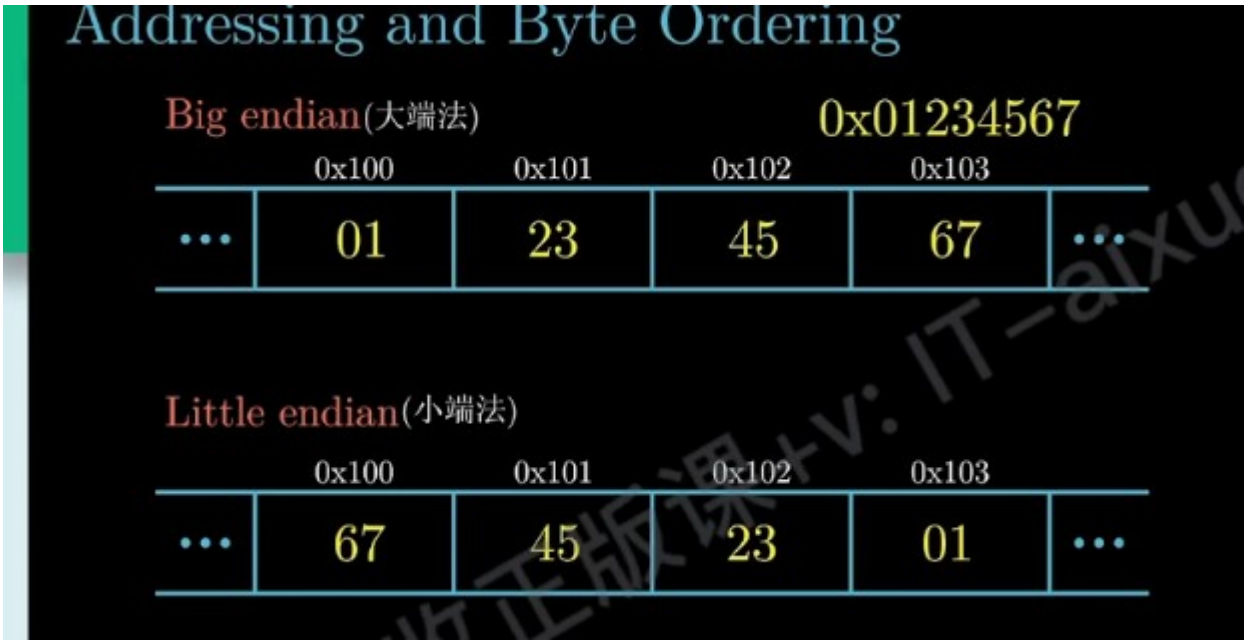
Int ->float 同样 32 位不会发生溢出但是会失去精度

Int/float->double 不发生溢出，可以保留精度

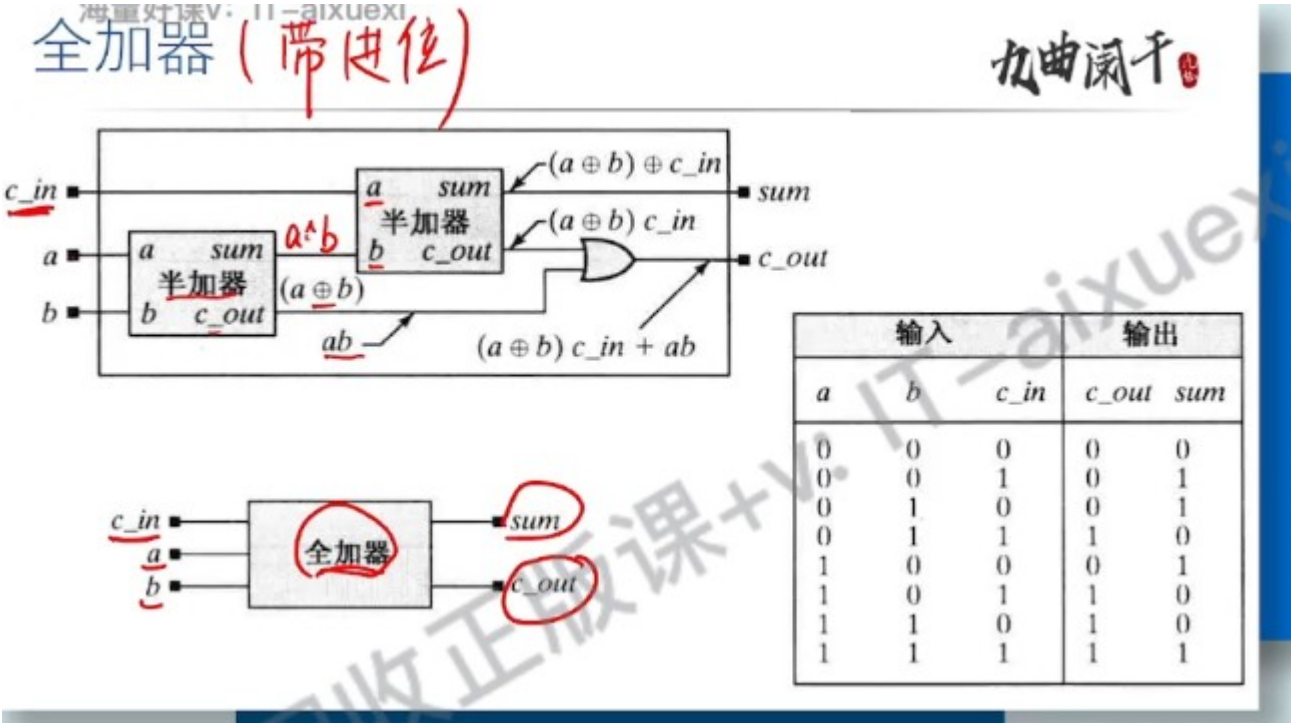
Double->float float 数值更小会溢出；float 精度更小会舍入

Float 、double->int 1、向 0 舍入 2、发生溢出

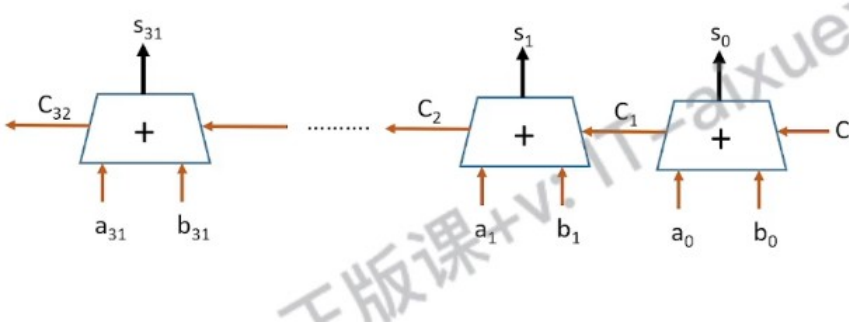
大端法与小端法



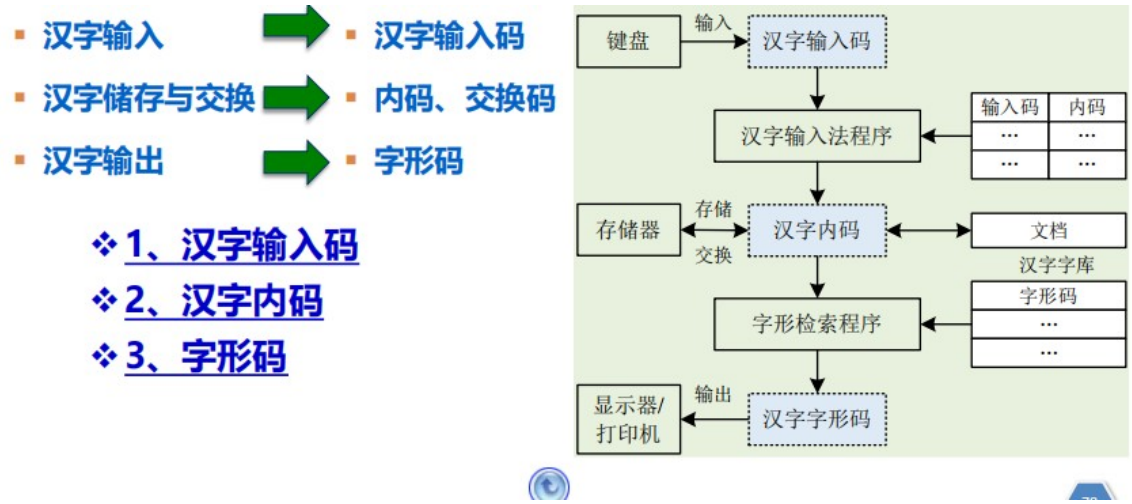
全加器



串行进位加法器（行波进位加法器）



先行进位加法器
乘法运算
除法运算



❖ 汉字输入：在西文标准键盘上，通过不同的字符组合，将汉字输入计算机，转换为汉字内码的过程
❖ 汉字输入码：也称外码，由不同的西文字符组合成的编码
❖ 汉字输入码种类：
▪
数字编码：区位码、国标码、电报码等
▪
拼音编码：搜狗拼音、微软拼音等
▪
字形编码：五笔字型输入法、郑码
▪
音形编码：自然码、钱码
汉字内码：用于汉字信息的存储、交换、检索等操作的机内代码
▪ 一般采用两个字节表示，汉字内码在计算机中是唯一的
❖ 汉字内码涉及以下几种编码：
❖ （1）区位码
▪ 汉字编码的国家标准：GB2312-1980《信息交换用汉字编码字符集·基本集》

▪ 共收录了 6763 个汉字、682 个图形符号
• 一级常用汉字：3755 个，按拼音字母顺序排列
• 二级次常用汉字：3008 个，按偏旁部首排列
▪ 两个字节编码汉字，每个字节各取 7 位，可编码 128×128＝16384 个字符
▪ 将所有的汉字及符号组成一个 94×94 的方阵，行→“区”，列→“位”
▪ 每个汉字与符号都位于某个区的某个位上，区号和位号→“区位码”
▪ 例如：“中”字位于 54 区 48 位→“中”字的区位码即为“5448 ❖ （2）国标码
▪ 国标码=区位码+2020H，占用两个字节
▪ 例如“中”字：区位码=5448
• 区码和位码转化为 16 进制，=3630H
• 国标码=3630H+2020H=5650H
❖ （3）汉字内码
▪ 西文字符：七位的 ASCII 码，当用一个字节表示时，最高位为“0”
▪ 汉字内码：汉字内码 = 汉字国标码 + 8080H，2 个字节的最高位均为“1”
• 例如“中”字的机内码=5650H+8080H=D6D0H
▪ 文本文件中储存的是汉字内码
汉字处理系统中字形信息：①用活字或文字版的母体字形形式；②点阵表示法、矢量表示法等形式，最基本的，应用也最广泛的是字模码
▪ 字模码：用点阵形式表示与存储汉字字形代码，它是汉字的输出形式，用于汉字的显示和打印
▪ 根据汉字输出的要求不同，点阵的规模也不同：
• 简易型汉字：16×16， 32 字节/汉字
• 普及型汉字：24×24， 72 字节/汉字
• 提高型汉字：32×32，128 字节/汉字
• 精密型汉字：48×48，288 字节/汉字
▪ 汉字字库：将所有汉字的字模点阵代码按顺序集中存放，构成了汉字库
▪ 字形检索程序：将汉字的机内代码转化为汉字

补码加减

❖补码的加法运算公式：

▪ $[X+Y]_{补} = [X]_{补} + [Y]_{补}$

证明：

$[X]_{补} = 2^{n+1} + X \pmod{2^{n+1}}$

$[Y]_{补} = 2^{n+1} + Y \pmod{2^{n+1}}$

$[X]_{补} + [Y]_{补} = 2^{n+1} + X + \cancel{2^{n+1}} + Y \pmod{2^{n+1}}$

$= 2^{n+1} + (X + Y) \pmod{2^{n+1}}$

$= [X+Y]_{补} \pmod{2^{n+1}}$

❖补码的减法运算公式：

▪ $[X-Y]_{补} = [X+(-Y)]_{补}$

$= [X]_{补} + [-Y]_{补}$

❖ 任意两数之和的补码 = 两数的补码之和

❖ 任意两数之差的补码 = 被减数的补码与减数相反数的补码之和

求补运算：$[Y]_{补} \rightarrow [-Y]_{补}$

- 求补规则：将$[Y]_{补}$包括符号位在内每一位取反，末位加1。
- 若$[Y]_{补} = Y_S, Y_1 \ldots\ldots Y_n$，则：
  - $[-Y]_{补} = \overline{Y_S}\ \overline{Y_1} \ \ldots\ldots\ \overline{Y_n} + 1$
- 若$[Y]补 = Y_S.Y_1 \ldots\ldots Y_n$，则：
  - $[-Y]_{补} = \overline{Y_S}\ \overline{Y_1} \ \ldots\ldots\ \overline{Y_n} + 0.0 \ldots\ldots 01$

- 溢出：运算结果超出机器数的表示范围
- 计算机硬件必须具备：
  - 能够检测运算结果是否发生溢出；
  - 能够指示运算结果是否发生溢出；
- 对于加减运算，可能发生溢出的情况：
  - 同号（两数）相加
  - 异号（两数）相减

Why?

- 补码加减运算判溢方法：
  - 单符号位判溢
  - 进位判溢
  - 双符号位判溢

- 一定会发生溢出的情况：
  - 正数相加，且运算结果符号位为1
  - 负数相加，且运算结果符号位为0
  - 正数 - 负数，且运算结果符号位为1
  - 负数 - 正数，且运算结果符号位为0

18

(1) 单符号位判溢方法

$\overline{ADD}/SUB$ =0：做加法

$\overline{ADD}/SUB$ =1：做减法

- 【例4.3】的4种情况：

$[X]_{补} = X_S X_1 \ldots\ldots X_n$

$[Y]_{补} = Y_S Y_1 \ldots\ldots Y_n$

$[X \pm Y]_{补} = F_S F_1 \ldots\ldots F_n$

运算	操作数X	X_S	操作数Y	Y_S	运算结果F	F_S	溢出情况
X+Y	正数	0	正数	0	负数	1	正溢出
X+Y	负数	1	负数	1	正数	0	负溢出
X-Y	正数	0	负数	1	负数	1	正溢出
X-Y	负数	1	正数	0	正数	0	负溢出

正数相加：结果符号位为1

负数相加：结果符号位为0

正数 - 负数：结果符号为1

负数 - 正数：结果符号为0

$$V = \overline{ADD}/\overline{SUB}(\overline{X_S}\ \overline{Y_S} F_S + X_S Y_S \overline{F_S}) + \overline{ADD}/SUB(\overline{X_S} Y_S F_S + X_S \overline{Y_S}\ \overline{F_S})$$

V=1：溢出

20

(3) 双符号位判溢方法

- X和Y采用双符号位补码参加运算，正数的双符号位为00，负数的双符号位为11；当运算结果的两位符号F_{S1} F_{S2}不同时（01或10），发生溢出。

$$\begin{array}{r} X_S\ X_S\ X_1\ X_2\ \cdots\cdots X_n \\ +\ Y_S\ Y_S\ Y_1\ Y_2\ \cdots\cdots Y_n \\ \hline F_{S1}\ F_{S2}\ F_1\ F_2\ \cdots\cdots F_n \end{array} \qquad \begin{array}{r} X_S\ X_1\ X_2\ \cdots\cdots X_n \\ +\ Y_S\ Y_1\ Y_2\ \cdots\cdots Y_n \\ \hline F_S\ F_1\ F_2\ \cdots\cdots F_n \end{array}$$

$F_{S1} = X_S \oplus Y_S \oplus C_S$

$F_{S2} = F_S$

- $V = F_{S1} \oplus F_{S2}$
- $V = F_{S1} \oplus F_{S2} = X_S \oplus Y_S \oplus C_S \oplus F_S$
- $F_{S1}\ F_{S2}=01$，则正溢出；$F_{S1}\ F_{S2}=10$，则负溢出。

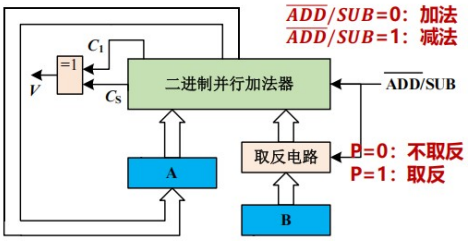
核心部件：一个普通的二进制并行加法器。

A：累加器，存放$[X]_{补}$

B：寄存器，存放$[Y]_{补}$

取反电路：

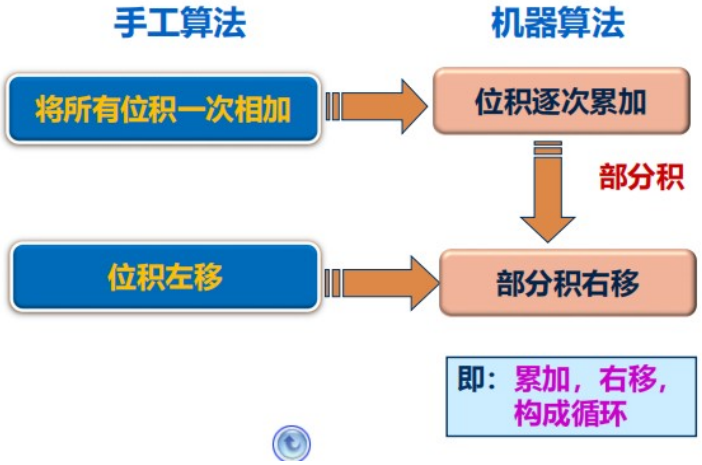
- 控制信号P=1，取反电路工作；
  P=0：不工作（不取反）
- P= $\overline{ADD}/SUB$



工作原理：

- $\overline{ADD}/SUB$ =0：补码加法器，B寄存器→并行加法器（P=0）；
- $\overline{ADD}/SUB$ =1：补码减法器，B取反→并行加法器（P=1），且并行加法器的最低位的进位=1，即B取反加1；相当于[X]补加[-Y]补，→减法运算

乘法实现



算法步骤：

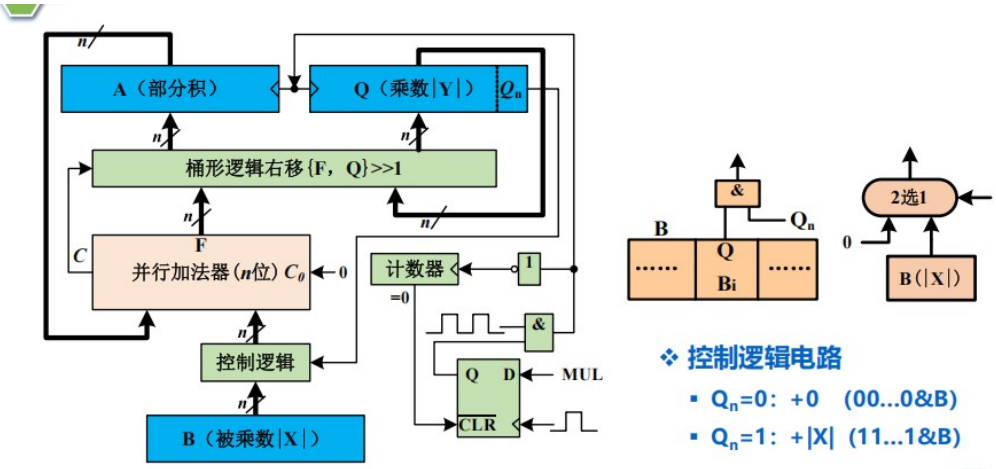
- $[X]_{原}=X_S\ X_1\ X_2\ \ldots\ldots X_n$
- $[Y]_{原}=Y_S\ Y_1\ Y_2\ \ldots\ldots Y_n$
- $P=X\cdot Y$，P_S是积的符号

① 符号位单独处理 $Ps=Xs \oplus Ys$

② 绝对值进行数值运算 $|P|=|X|*|Y|$

③ 初始部分积为0，从乘数最低位开始累加、右移：Yi=1，部分积加|X|，Yi=0，部分积加0，累加结果右移一位，得新部分积。

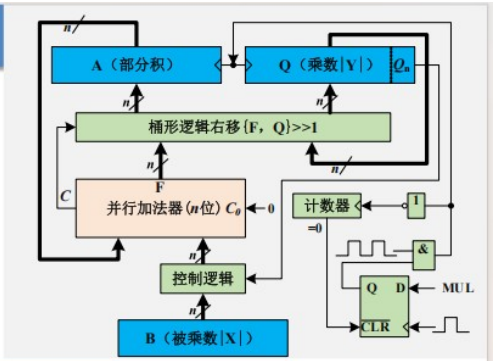
④ 累加右移n次，即i=n，n-1，…，2，1



- 控制逻辑电路
  - Q_n=0：+0 (00...0&B)
  - Q_n=1：+|X| (11...1&B)

3、原码乘法的硬件实现

- A：累加寄存器
- B：被乘数寄存器
- Q：乘数寄存器
- 桶形移位器：F、Q联合逻辑右移
  - A←{C，F_1...F_{n-1}} Q←{F_n,Q_1...Q_{n-1}}

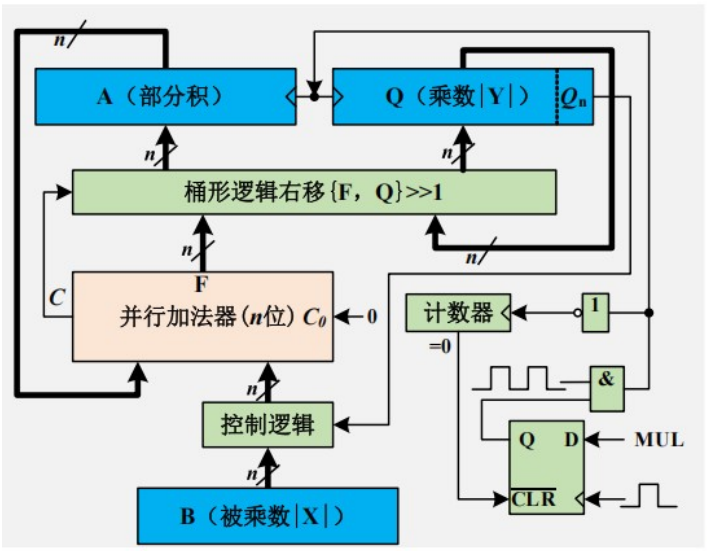


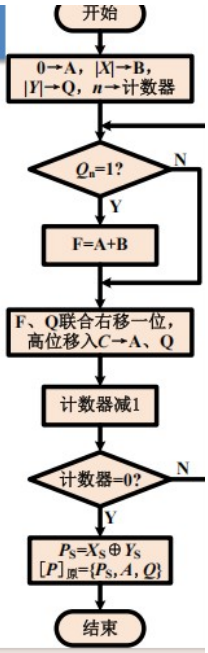
初始：
- A=0
- B=|X|
- Q=|Y|
- 计数器=n

累加、右移n次

结果：
- A=乘积绝对值高位
- B =|X|
- Q=乘积绝对值低位
- 计数器=0

原码一位乘法流程：





除法

- X=+0.1011，Y=-0.1101，求X÷Y
- 手工除法：
  - 判断被除数是否大于除数，决定商0/1。
  - 若商1，则减除数（得部分余数），再添加一位数或者添加0
  - 若商0，则直接添加一位数或者添加0
- 改进手工算法→适合机器运算：
  - 计算机通过做减法测试来实现判断：结果大于等于0，表明够减，商1；结果小于0，表明不够减，商0。
  - 计算机将部分余数左移一位，再直接与不右移的除数相减。
- 即：减（+除数相反数的补码）
  左移，构成循环

整数　小数　部分余数

```
        0.1101
0.1101 ) 0.10110
         1101
        10010
         1101
        10100
         1101
         0111
```

是2n位除以n位

- $[X]_{原}=X_S.X_1X_2\cdots\cdots X_n$，$[Y]_{原}=Y_S.Y_1Y_2\cdots\cdots Y_n$
- Q：X÷Y的商，Q_S：商的符号，R：X÷Y的余数，R_S：余数的符号
- 原码除法运算的规则是：

①$Q_S = X_S \oplus Y_S$ ，$R_S = X_S$，　$|Q| = (|X|-|R|) \div |Y|$

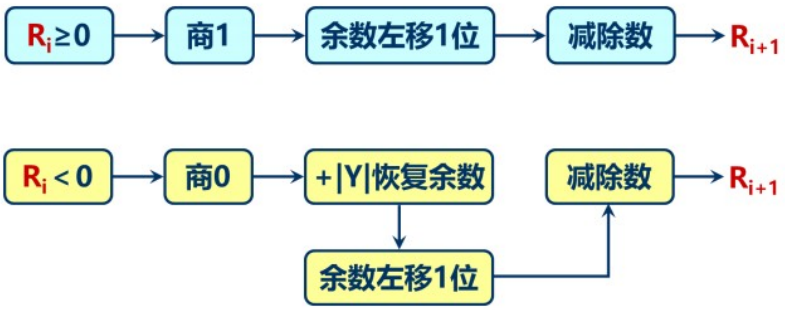
②余数和被除数、除数均采用双符号位；初始余数为|X|，初始商值为全0。

③部分余数减去|Y|（通过$+[-|Y|]_{补}$来实现）：

- 结果符号位为0：够减，上商1，余数左移一位
- 结果符号位为1：不够减，上商0，先加|Y|恢复余数，然后余数左移一位

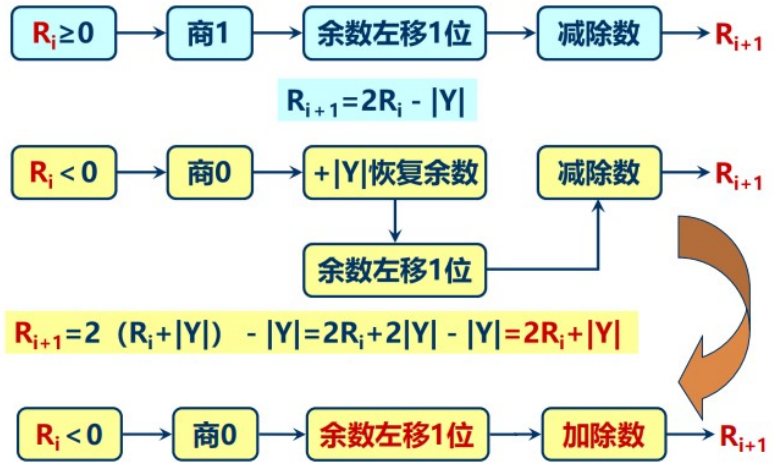
④循环操作步骤③，共做n+1次，最后一次不左移

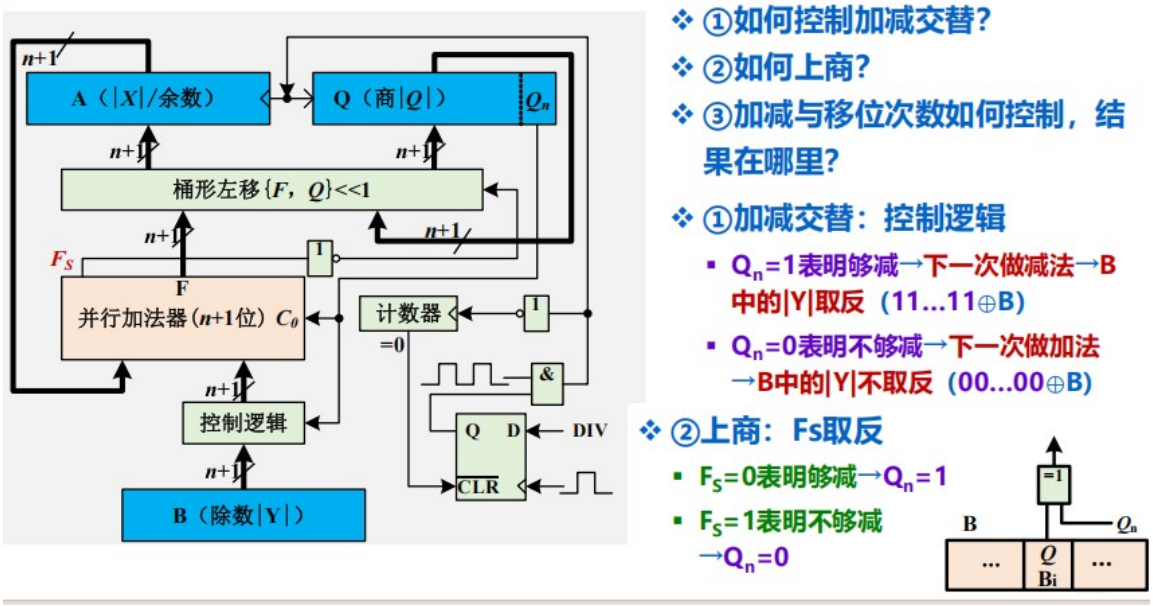
- 若最后一次上商0，则必须+|Y|恢复余数；
- 若为定点小数除法，余数则为最后计算得到的余数右移n位的值



不恢复余数算法

- 又称为加减交替法：当不够减时，不恢复余数，用加上除数（+|Y|）的办法来继续求下一位商，其他操作不变。
- 加减交替法的规则如下：
  - 余数为正：商上1，求下一位商的办法，是余数左移一位，再减去除数；
  - 余数为负：商上0，求下一位商的办法，是余数左移一位，再加上除数。
  - 若最后一次上商为0，而又需得到正确余数，则最后一次仍需恢复余数。
- Why?

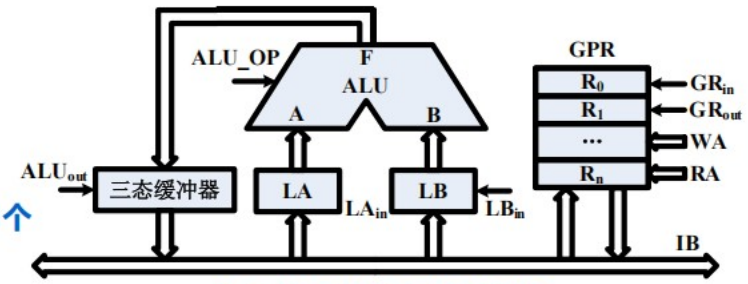

Ri≥0
商1
余数左移1位
减除数
Ri+1
Ri+1=2Ri - |Y|
Ri < 0
商0
+|Y|恢复余数
减除数
Ri+1
余数左移1位
Ri+1=2（Ri+|Y|） - |Y|=2Ri+2|Y| - |Y|=2Ri+|Y|
Ri < 0
商0
余数左移1位
加除数
Ri+1


n+1
A（|X|/余数）
Q（商|Q|）
Qn
n+1
n+1
桶形左移{F，Q}<<1
n+1
n+1
FS
1
F
并行加法器（n+1位）C0
计数器
1
=0
&
n+1
控制逻辑
Q
D
DIV
n+1
CLR
B（除数|Y|）
❖ ①如何控制加减交替?
❖ ②如何上商?
❖ ③加减与移位次数如何控制，结果在哪里?
❖ ①加减交替：控制逻辑
▪ Qn=1表明够减→下一次做减法→B中的|Y|取反（11...11⊕B)
▪ Qn=0表明不够减→下一次做加法→B中的|Y|不取反（00...00⊕B)
❖ ②上商：Fs取反
▪ FS=0表明够减→Qn=1
▪ FS=1表明不够减→Qn=0
=1
B
Qn
...
Q
Bi
...

定点运算器

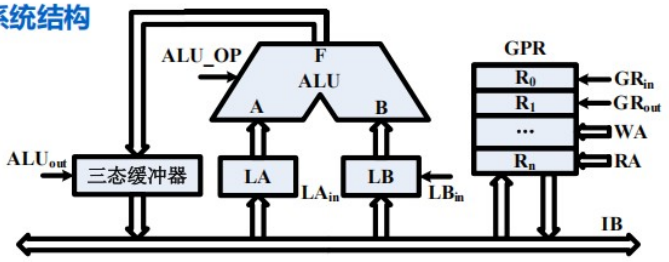
❖构成：

- 内部总线IB
- ALU
- 2个暂存器：LA和LB
- 通用寄存器堆GPR：n+1个



- ❖ LA_{in}=1：IB上数据写入LA
- ❖ LB_{in} =1：IB上数据写入LB
- ❖ ALU_OP：功能码/运算码，选择ALU执行的运算
- ❖ ALU_{out}=1：ALU的运算结果（即输出端F）→IB
- ❖ RA：执行读操作的寄存器地址
- ❖ WA：执行写操作的寄存器地址
- ❖ GR_{out}=1：读出RA指定的寄存器数据送到IB上（Reg[RA]→IB）
- ❖ GR_{in}=1：将IB上的数据写入由写地址WA选中的寄存器（IB→Reg[WA]）

数据通路：指数据在硬件系统结构上流动的过程与控制步骤

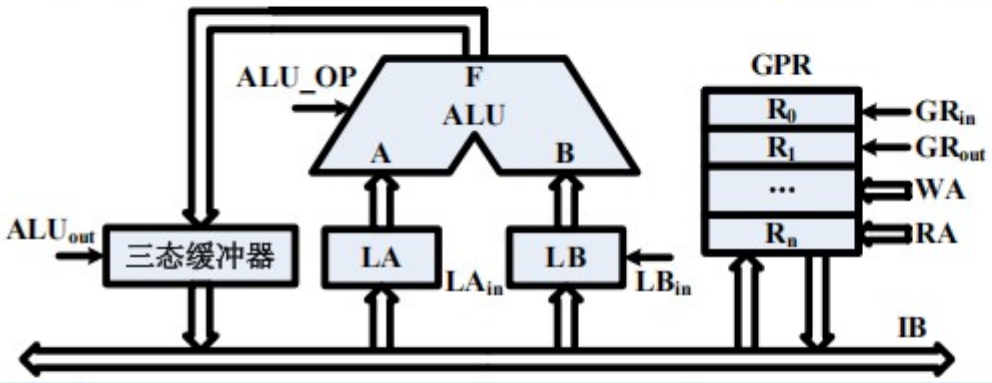


在总线上传送数据的数据通路：需要一个CPU周期

- 源部件将数据送上总线；目的部件从总线上接收数据
- 任何要把数据送上总线的源部件，其数据输出端必须以三态输出的形式与总线连接，并由使能信号控制
- 对于某个总线系统来说，各个总线源部件的数据输出使能信号不能同时有效：任何时刻只能允许一个部件将数据送上总线，以保证分时享用总线

（1）单总线结构

❖ 【例4.10】假设在单总线运算器中，完成一条指令ADD R_3，R_2作$R_2+R_7\rightarrow R_3$，则分析该操作的数据通路，需要几个CPU周期？

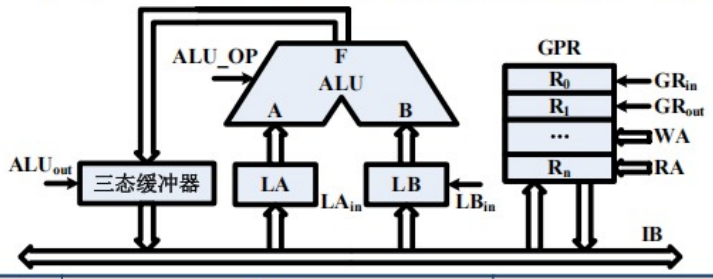


❖ 需要

步骤	操作	发送的信号
1	$R_2\rightarrow LA$	$RA=2$，$GR_{out}=1$，$LA_{in}=1$
2	$R_7\rightarrow LB$	$RA=7$，$GR_{out}=1$，$LB_{in}=1$
3	ALU执行F=A+B运算，结果$F\rightarrow R_3$	ALU_OP=加法功能码，$ALU_{out}=1$，

（1）单总线结构

❖ 【例4.10】假设在单总线运算器中，完成一条指令ADD R_3，R_2，R_7的运算操作$R_2+R_7\rightarrow R_3$，则分析该操作的数据通路，需要几个CPU周期？

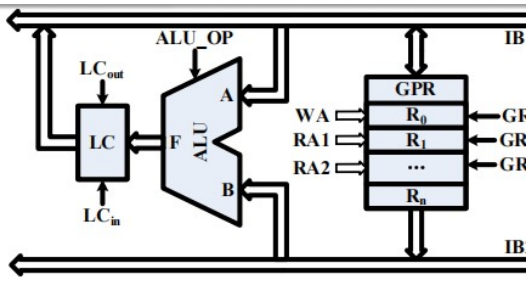


❖ 需要3个CPU周期

步骤	操作	发送的信号
1	$R_2\rightarrow LA$	$RA=2$，$GR_{out}=1$，$LA_{in}=1$
2	$R_7\rightarrow LB$	$RA=7$，$GR_{out}=1$，$LB_{in}=1$
3	ALU执行F=A+B运算，结果$F\rightarrow R_3$	ALU_OP=加法功能码，$ALU_{out}=1$，$GR_{in}=1$，$WA=3$

❖构成：

- 2条内部总线：IB1、IB2
- ALU
- 暂存器：LC
- 通用寄存器堆GPR：n+1个

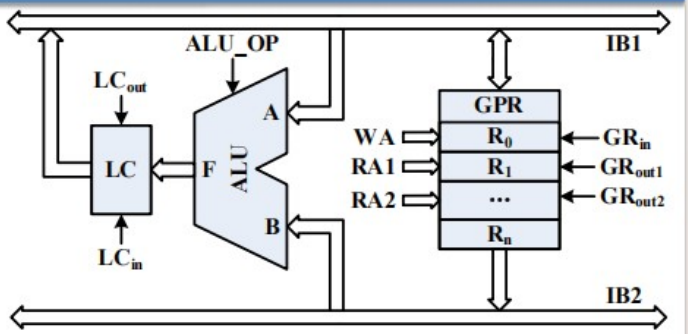


- ❖ LC_{in}=1： ALU运算结果写入LC，即F→LC
- ❖ LC_{out} =1：LC中数据送上总线IB1，即LC→IB1
- ❖ ALU_OP：ALU功能码/运算码
- ❖ RA1：执行读出数据到IB1的寄存器地址
- ❖ RA2：执行读出数据到IB2的寄存器地址
- ❖ WA：执行写操作的寄存器地址
- ❖ GR_{out1}=1：读出RA1指定的寄存器数据送到IB1上，即Reg[RA1]→IB1
- ❖ GR_{out2}=1：读出RA2指定的寄存器数据送到IB2上，即Reg[RA2]→IB2
- ❖ GR_{in}=1：将IB1上的数据写入由写地址WA选中的寄存器，即IB1→Reg[WA]

（2）双总线结构

❖ 【例4.11】假设在双总线运算器中，同样完成指令ADD R_3，R_2，R_7的运算操作$R_2+R_7→R_3$，分析数据通路，需要几个CPU周期?

❖ 需要2个CPU周期



步骤	操作	发送的信号
1	R_2→IB1，R_7→IB2，F=A+B，F→LC	RA1=2，RA2=7，ALU_OP=加法功能码，GR_{out1}=1，GR_{out2}=1，LC_{in}=1
2	LC→R_3	LC_{out}=1，GR_{in}=1，WA=3

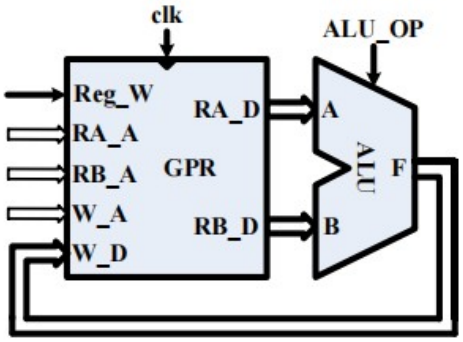
❖ 思考：如果LC暂存器输出端连接到IB2上，能否完成上述操作?

❖ 总结：在分析总线结构的运算器数据通路时，基本的原则：在一个CPU周期内，某条总线上的数据必须是唯一的，且不能保留至下一个CPU周期。

❖ 特征：按照运算操作的过程与控制需求，部件之间由专用的独立线路进行连接

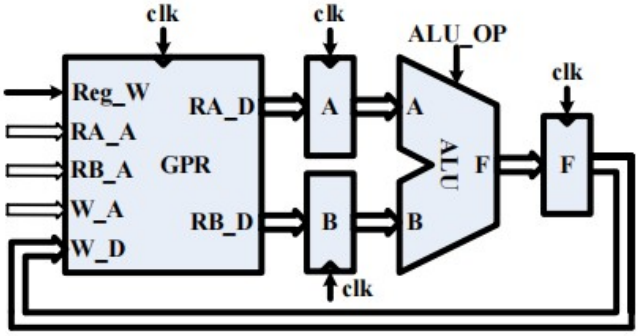
❖ 特点：线路比较复杂，但是传输效率高，适用于流水线CPU

❖ 单周期CPU的运算器



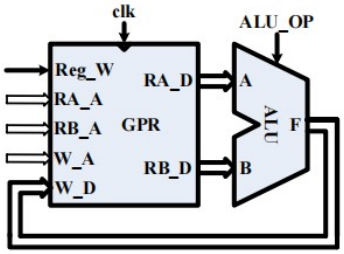
❖ 每条指令只需一个CPU周期完成

❖ 多周期CPU的运算器



❖ 每条指令需要多个CPU周期完成

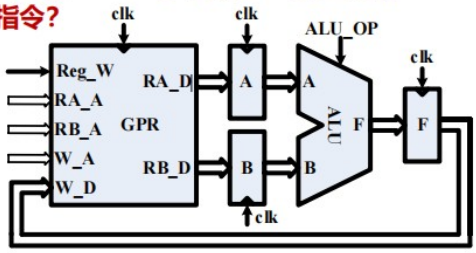
❖ 单周期CPU的运算器



- 只需1个clk周期：
  - RA_A=2，RB_A=7，W_A=3
  - ALU_OP=加法功能码
  - Reg_W=1，在clk的边沿完成 $R_2+R_7\rightarrow R_3$操作

❖ 执行ADD R_3，R_2，R_7指令？

❖ 多周期CPU的运算器



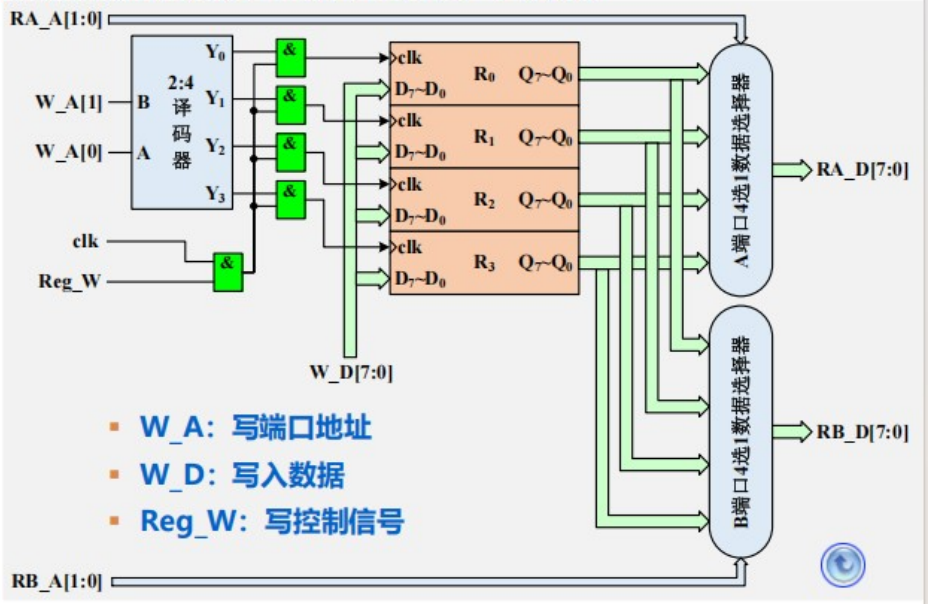
❖ 需3个节拍（clk周期）：

- 第1个clk（读寄存器）：$R_2\rightarrow$A，$R_7\rightarrow$B：RA_A=2，RB_A=7
- 第2个clk（执行）：A+B→F：ALU_OP=加法功能码
- 第3个clk（写回）：F→R_3：W_A=3，Reg_W=1

❖ 特征：能够对一组寄存器同时读出或者写入多个数据

❖ 三端口寄存器堆
可以同时进行
- 读A端口
- 读B端口
- 写入数据

❖ 寄存器堆：4×8位
- R_0~R_3，地址2位
- RA_A：A口地址
- RA_D：A口读出数据
- RB_A：B口地址
- RB_D：B口读出数据



- W_A：写端口地址
- W_D：写入数据
- Reg_W：写控制信号

❖ 标志寄存器FR：又称为状态寄存器　　PSW：程序状态字
- 用来保存ALU操作结果的某些状态，可作为外界对运算结果进行分析的依据，也可以用于判断程序是否要转移的条件

❖ 最基本的5种运算结果标志：

①ZF：结果为零标志
- 运算结果为全0，ZF置1
- 运算结果不全为0， ZF置0

$$ZF = \overline{F_n + F_{n-1} + \cdots\cdots + F_0}$$

②CF：进位/借位标志位，CF标志只对无符号数运算有意义
- 加法运算时：C=1则CF置1（表示有进位），否则置0
- 减法运算时：C=0则CF置1（表示不够减，有借位），否则置0

最高位的进位

$$CF = \overline{\overline{ADD}/SUB} \bullet C + \overline{\overline{ADD}/SUB} \bullet \overline{C}$$

③OF：溢出标志，反映有符号数加减运算所得结果是否溢出；OF标志只对带符号数运算有意义。

- 运算溢出：OF=1
- 运算没有溢出：OF=0

$$OF = \overline{ADD/SUB}(\overline{X_S}\,\overline{Y_S}F_S + X_S Y_S \overline{F_S}) + \overline{ADD/SUB}(\overline{X_S} Y_S F_S + X_S \overline{Y_S}\,\overline{F_S})$$

$$OF = C_1 \oplus C_S \qquad OF = F_{S1} \oplus F_{S2}$$

④SF：符号标志，记录运算结果的符号。

- 为运算结果的最高位 $SF = F_n$
- 运算结果为正数时：SF=0；为负数时：SF=1。（无溢出时才正确）

⑤PF：奇偶标志，反映运算结果中“1”的个数的奇偶性

- 结果中“1”的个数为偶数：PF=1
- 结果中“1”的个数为奇数：PF=0

$$PF = \overline{F_n \oplus F_{n-1} \oplus \cdots\cdots \oplus F_0}$$

❖ 假设两个浮点数X和Y

$$X = M_X \times 2^{E_X} \qquad Y = M_Y \times 2^{E_Y}$$

❖ 对齐小数点：使得阶码小的那个数变得与阶码大的那个数的阶码

❖ 然后：对尾数（有效数位）做加减运算

$$Z = X \pm Y = (M_X \bullet 2^{(E_X - E_Y)} \pm M_Y) \times 2^{E_Y} \qquad E_X \leq$$

(2) 对阶：

- 目标是：**对齐小数点**
- 原则是：**小阶对大阶**
  - 求阶差$\Delta E = E_X - E_Y$，若$\Delta E \neq 0$，即$E_X \neq E_Y$时需要对阶。
  - 若$\Delta E > 0$，则$E_X > E_Y$，M_Y每右移一位，$E_Y + 1$，直至 $E_Y = E_X$
  - 若$\Delta E < 0$，则$E_X < E_Y$，M_X每右移一位，$E_X + 1$，直至$E_X = E_Y$

(3) 尾数相加减

- 若做减法，先将操作数Y的尾数取相反数，再与Y的尾数相加
- 对阶时，右移多出的数位不丢弃（因为一般设置位数更宽的ALU进行计算，同时设置更宽的寄存器来存放中间运算结果）

浮点加减运算步骤

(4) 结果规格化：尾数运算的结果可能出现两种非规格化情况：

- 尾数溢出：需要右规（1次），即尾数右移1位，阶码 + 1
- $|\text{尾数}| < 2^{-1}$：需要左规，即尾数左移1位，阶码 - 1，左规可能多次，直到尾数变为规格化形式。

- 先判断是否溢出，如果没有，再判断是否有前导零

(5) 舍入：将尾数多余的位数舍去，变成规定的浮点数格式

- 舍入策略：舍去多余位数的方法，简单的舍入策略：
  - 截断法：多余的全舍去
  - 0舍1入法：多余位的最高位为1时，在尾数最低位上+1，否则舍去多余位
  - 末位恒置1：始终在舍入结果的最末位置1

❖IEEE 754标准：规定了4种可选的舍入模式

①向上舍入（总是朝+∞）：为正数时，只要多余位不全为0，就向最低有效位进1；为负数时，则采用简单的截断法

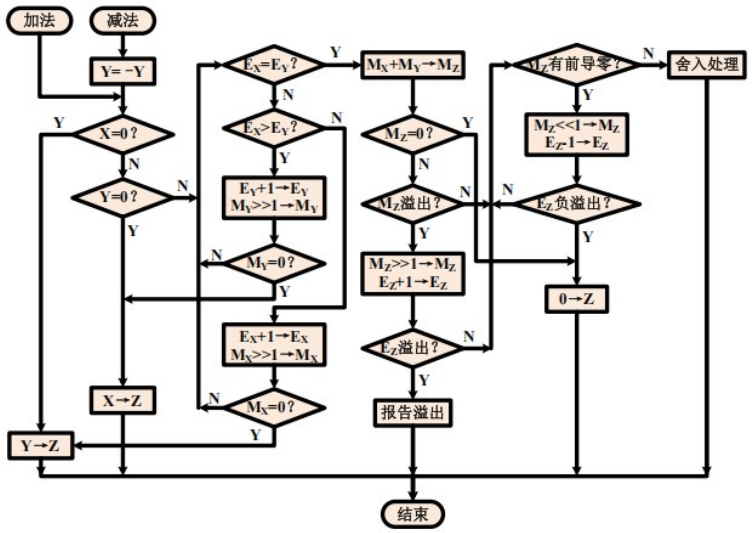
②向下舍入（总是朝 - ∞）：为正数时，只要多余位不全为0，就简单地截尾为负数时，则向最低有效位进1

③向0舍入：即朝数轴的原点方向舍入，就是无论正数还是负数，都采用简单的截尾，从而使得绝对值总是变小。这种方法容易累积误差

④就近舍入：即舍入到最接近的数，“四舍五入”。多余位=$R_1R_2……R_n$

- 当R_1=0：截尾（即舍去）；
- 当R_1=1,且$R_2…R_n$不全为0：向最低有效位进1；
- 当$R_1R_2……R_n$=10……0：即等于一半（中点），则若最低有效位为0就截尾，若最低有效位为1就进1，以使得最低有效位总是为0

❖中间结果设置了：保护位(guard)、舍入位(round)、粘滞位(sticky)



浮点加减运算例题

(2) 尾数相加:

$[M_Z]_{补}$ = 1.1000001 (101)

	$[M_X]_{补}$	11.0110101
+	$[M_Y]_{补}$	00.0001100(101)
	$[M_X+M_Y]_{补}$	11.1000001(101)

(3) 结果规格化:

- 首先判断: 是否溢出?
  - 因为双符号位为11, 没有溢出
- 再检查: 有无前导零?
  - 因为$[M_Z]_{补}$符号位和最高有效位相同=1.1, 故有一个前导零, 须左规一位
- 左规一位: 尾数左移1位, 阶码-1
  - $[M_Z]_{补}$ = 1.0000011 (01)
  - $[E_Z]_{补}$ = 00, 001 + 11, 111= 00, 000

(4) 舍入: 按照0舍1入法, 尾数多余位舍去

结果为: $[X+Y]_{浮}$ = 0, 000 1.0000011