

第二章 计算机的运算方法

第二章 计算机的运算方法

- 计算机中数的表示
- 定点运算
- 浮点运算

第二章 计算机的运算方法

- 计算机中数的表示
 - 无符号数和有符号数
 - 定点表示和浮点表示
 - IEEE754标准
 - 算数移位与逻辑移位

(回顾) 十进制整数转换成二进制整数的经典方法

十进制整数转换为二进制整数采用“**除 2 取余，逆序排列**”法。具体做法是：用 2 整除十进制整数，可以得到一个商和余数；再用 2 去除商，又会得到一个商和余数，如此进行，直到商为小于 1 时为止，然后把先得到的余数作为二进制数的低位有效位，后得到的余数作为二进制数的高位有效位，依次排列起来。

以下是两个例题：

The image shows two handwritten examples of converting decimal numbers to binary using the division-by-2 method. Each example consists of a column of division steps, a column of remainders, and a final binary representation.

Example 1: Conversion of 179 to binary

2 179	
2 89	1
2 44	1
2 22	0
2 11	0
2 5	1
2 2	1
2 1	0
0	1

A red arrow points upwards from the bottom remainder (1) to the top remainder (1), indicating the reverse order of reading.

$(179)_{10} = (10110011)_2$

Example 2: Conversion of 206 to binary

2 206	
2 103	0
2 51	1
2 25	1
2 12	1
2 6	0
2 3	0
2 1	1
0	1

A red arrow points upwards from the bottom remainder (1) to the top remainder (0), indicating the reverse order of reading.

$(206)_{10} = (11001110)_2$

无符号数

- 寄存器的位数反映无符号数的表示范围。

8 位

--	--	--	--	--	--	--	--

0 ~ 255

16 位

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

0 ~ 65535

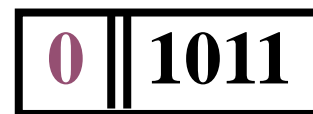
有符号数：真值与机器数

真值：带符号的数

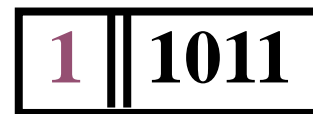
小数 + 0.1011 或 0.1011
- 0.1011

整数 + 1100 或 1100
- 1100

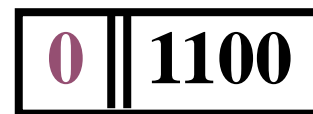
机器数：符号数字化的数



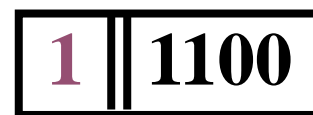
↑ 小数点的位置



↑ 小数点的位置



↑ 小数点的位置



↑ 小数点的位置

注：以后非特殊说明，默认二进制数表示；

二进制数位数不是8的倍数，只是为了讲解方便。

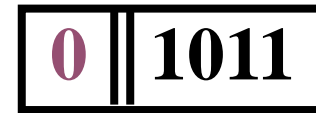
有符号数：真值与机器数

真值：带符号的数

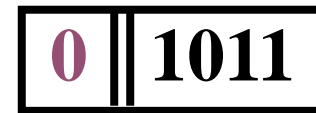
+ 0.1011

+ 1011

机器数：符号数字化的数



小数点的位置



小数点的位置



- 小数和整数在计算机中的表示一模一样？
- 如何表示既有整数又有小数的数据，如+11.01？

原码表示法：整数

带符号的绝对值表示

$x = +1110$ $[x]_{\text{原}} = 0, 1110$ 用逗号将符号位和数值部分隔开

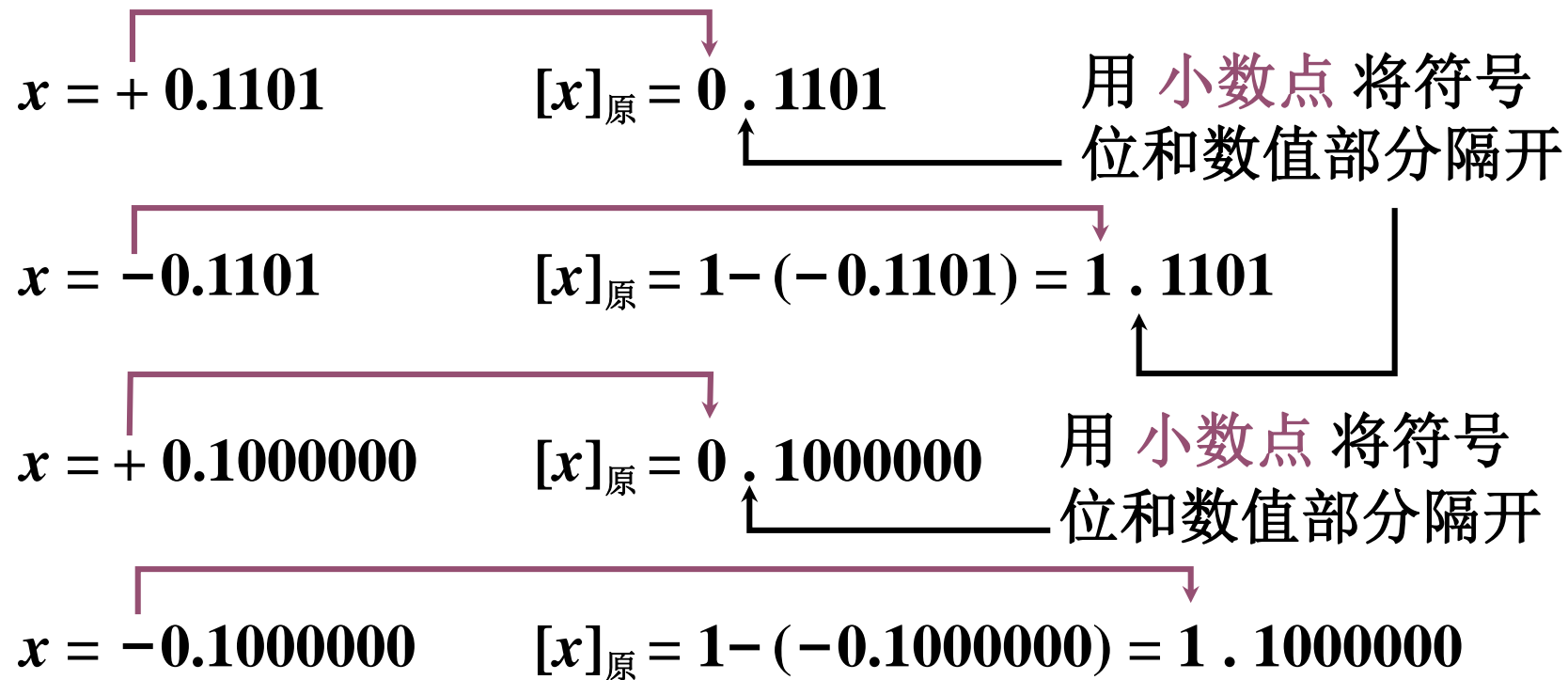
$x = -1110$ $[x]_{\text{原}} = 1, 1110$

$$[x]_{\text{原}} = 2^4 + 1110 = 1, 1110$$

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

x 是真值， n 是数值位数

原码表示法：小数



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

例：根据原码求真值

- 例1. 已知 $[x]_{\text{原}} = 1.0011$, 求 x

解：由定义得

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

- 例2. 已知 $[x]_{\text{原}} = 1,1100$, 求 x

解：由定义得

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

举例

- 例3. 已知 $[x]_{\text{原}} = 0.1101$, 求 x

解: 根据定义 $\because [x]_{\text{原}} = 0.1101 > 0$

$$\therefore x = +0.1101$$

- 例4. 求 $x = 0$ 的原码

解: 设 $x = +0.0000$ 则 $[+0.0000]_{\text{原}} = 0.0000$

$x = -0.0000$ 则 $[-0.0000]_{\text{原}} = 1.0000$

同理, 对于整数 $[+0]_{\text{原}} = 0,0000$

$[-0]_{\text{原}} = 1,0000$

$$\therefore [+0]_{\text{原}} \neq [-0]_{\text{原}}$$

注意: 原码 $x = 0$ 也是要分成小数和整数分别讨论的

口诀1（真值与原码之间转化）

1) 符号位：0正1负

2) 小数分隔符号和数值用小数点 “.”
整数分隔符号和数值用 逗号 “,”

原码的优缺点

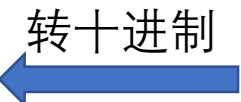
- 优点：简单、直观
- 缺点：1) +0和-0原码不一样

2) 做加减运算时，会出现如下问题：

要求	数1	数2	实际操作	结果符号
减法	正	正	减法	可正可负
加法	正	负	减法	可正可负
加法	负	正	减法	可正可负
减法	负	负	减法	可正可负

- 但实际运算时有时候需要比大小，绝对值做减法，能否只做加法？
找到与负数等价的正数来代替这个负数，就可化减法为加法

原码的缺点——续

十进制		原码									
	3		<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	1				
+	-3	+	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	1	0	0	0	0	0	1	1
1	0	0	0	0	0	1	1				
<hr/>		<hr/>									
	0		<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	1	0	0	0	0	1	1	0
1	0	0	0	0	1	1	0				
		-6									

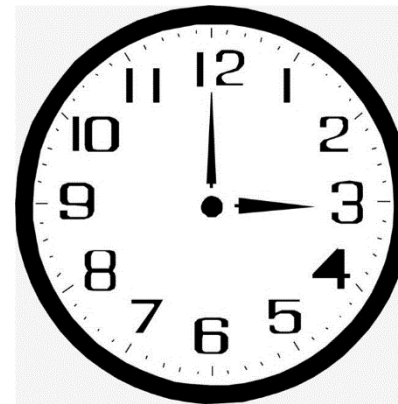
原码运算 不等于 十进制运算，尤其是当符号参与运算。

补数表示法

- 小明从下午5点学习到凌晨3点，一共学了多少小时？

- 补的概念：时钟以12为模

- 逆时针： $5 - 2 = 3$
- 顺时针： $5 + 10 = 3 + 12$



- 可见 -2 可用 $+10$ 代替

- 称 $+10$ 是 -2 （以 12 为模）的补数

- 记作 $-2 \equiv +10 \pmod{12}$

同理 $-4 \equiv +8 \pmod{12}$

$-5 \equiv +7 \pmod{12}$

减法 \longrightarrow 加法

补数——续

计数器（模 16） $1011 \longrightarrow 0000 ?$

$$\begin{array}{r} 1011 \\ - 1011 \\ \hline 0000 \end{array}$$

$$\begin{array}{r} 1011 \\ + 0101 \\ \hline 10000 \end{array}$$

可见 -1011 与 $+0101$ 作用等价

记作 $-1011 \equiv +0101 \pmod{2^4}$

同理 $-011 \equiv +101 \pmod{2^3}$

$-0.1001 \equiv +1.0111 \pmod{2^1}$

自然去掉

- 结论（真值的绝对值小于模）
 - 一个负数加上“模”即得该负数的补数
 - 一个正数和一个负数互为补数时，绝对值之和即为模数

补数——续

正数的补数即为其本身

对于时钟：3点、15点、27点都是3点 \longrightarrow

$$\begin{aligned} 3 &\equiv 15 \equiv 27 \pmod{12} \\ 3 &\equiv 3+12 \equiv 3+24 \equiv 3 \pmod{12} \end{aligned}$$

同理： $+0101 \equiv +0101 + 2^4 \equiv +0101 \pmod{2^4}$

前面已证明： $-1011 \equiv -1011 + 2^4 \equiv +0101 \pmod{2^4}$

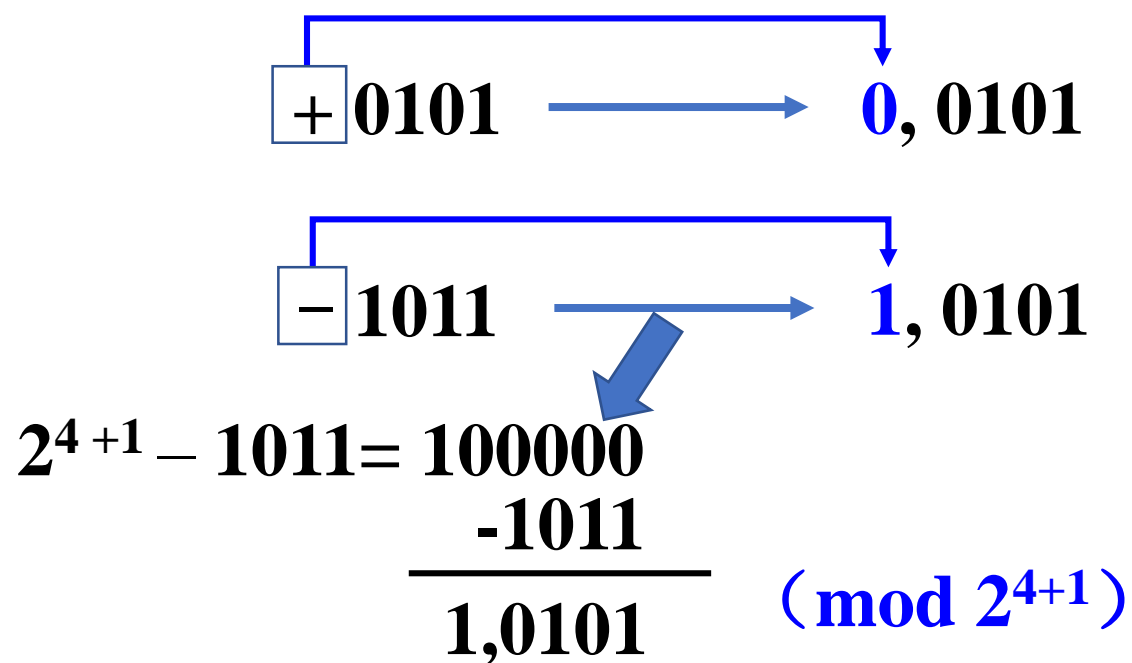
问题： $+0101$ 究竟是 -1011 的补数还是 $+0101$ 的补数呢？



补数——续

问题：+ 0101 究竟是-1011的补数还是+0101的补数呢？

解决办法：前面加一个0或1来区别原真值的正负：



补码表示法：二进制整数

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

其中： x 为真值， n 为二进制整数的位数

$$x = -1011000$$

$$x \text{ 的补数} = -1011000 + 2^7 \quad \text{检验上式为什么是 } 2^{n+1}?$$

$$= 0101000$$

$$[x]_{\text{补}} = 1,0101000$$

2^7

$$x = +0101000$$

$$[x]_{\text{补}} = 0,0101000$$



用 逗号 将符号位
和数值部分隔开

$$[x]_{\text{补}} = 2^{7+1} + (-1011000)$$

$$= 100000000$$

$$- \quad 1011000$$

$$\hline 1,0101000$$

补码表示法：二进制（纯）小数

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

其中： x 为真值

$$x = + 0.1110$$
$$[x]_{\text{补}} = 0.1110$$

用 小数点 将符号位
和数值部分隔开

$$x = - 0.1100000$$
$$\begin{aligned} [x]_{\text{补}} &= - 0.1100000 + 2 \\ &= 10.0000000 \\ &\quad - 0.1100000 \\ \hline &= 1.0100000 \end{aligned}$$

求补码的快捷方式

设 $x = -1010$ 时

$$\begin{aligned} \text{则 } [x]_{\text{补}} &= 2^{4+1} - 1010 = 11111 + 1 - 1010 \\ &= 100000 &= 11111 + 1 \\ &\quad - 1010 &\quad - 1010 \\ \hline &= 1,0110 &\quad \boxed{10101} + 1 \\ & &= 1,0110 \end{aligned}$$

$$\text{又 } [x]_{\text{原}} = \boxed{1,1010}$$

当真值为 负 时，原码和补码的互相转换的快捷方法（特例除外）

方法1（经典方法）：原码除符号位外，每位取反，末位加 1

方法2（扫描法）：符号位不变，自右向左从最先遇到的1的左边开始，每位取反。注意：扫描法只针对负数

口诀2（原码与补码之间互相转化）

1) 正数不变

2) 负数使用扫描法：符号位不变，自右向左从最先遇到的1的左边开始，每位取反

说明：口诀对整数和小数都是通用的

注意特例：原码的-0、补码的最小值这两种情况不能进行双向转化表示。

举例：已知小数补码求真值

已知 $[x]_{\text{补}} = 1.0001$ ，求 x 。

$$\begin{aligned}\text{解：由定义得 } x &= [x]_{\text{补}} - 2 \\ &= 1.0001 - 10.0000 \\ &= -0.1111\end{aligned}$$

$$\begin{aligned}\text{或：} [x]_{\text{补}} &\rightarrow [x]_{\text{原}} & [x]_{\text{原}} &= 1.1111 \\ & & \therefore x &= -0.1111\end{aligned}$$

**强烈推荐
扫描法**

当真值为**负**时，已知补码求原码的快捷方法：

补码除符号位外，每位取反，末位加 1（需要记住）

补码除符号位外，末位减 1，再每位取反（不常用）

已知补码求真值（双向转换，注意特例不适用）

当真值为 负 时，根据原码求补码的快捷方法（特例除外）

- 原码除符号位外，每位取反，末位加 1
- 自右向左从最先遇到的1左边开始，每位取反（符号位除外）

逆运算（也是特例除外）：

当真值为 负 时，根据补码求原码的快捷方法

- 补码除符号位外，每位取反，末位加 1
- 自右向左从最先遇到的1左边开始，每位取反（符号位除外）

例子：求下列真值的补码

真值	$[x]_{\text{补}}$	$[x]_{\text{原}}$
$x = -70 = -1000110$	1, 0111010	1,1000110
$x = -0.1110$	1.0010	1.1110
$x = \boxed{0.0000}$ $[+0]_{\text{补}} = [-0]_{\text{补}}$	$\boxed{0.0000}$	0.0000
$x = \boxed{-0.0000}$	$\boxed{0.0000}$	1.0000
$x = -1.0000$	1.0000	不能表示

由小数补码定义
$$[x]_{\text{补}} = \begin{cases} x & 1 > x \geq 0 \\ 2 + x & 0 > x \geq -1 \pmod{2} \end{cases}$$

$$[-1]_{\text{补}} = 2 + x = 10.0000 - 1.0000 = 1.0000$$

请大家做一下雨课堂练习题

反码表示法：二进制整数

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

其中： x 为真值， n 为二进制数位

$$x = +1101$$

$$[x]_{\text{反}} = 0,1101$$



用 逗号 将符号位
和数值部分隔开

$$x = -1101$$

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

$$= 11111 - 1101$$

$$= 1,0010$$



反码表示法：二进制小数

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

其中： x 为真值， n 为二进制数位数


$$x = -0.1010$$

$$[x]_{\text{反}} = (2-2^{-4}) - 0.1010$$

$$= 1.1111 - 0.1010$$

$$= 1.0101$$

用 小数点 将符号位

和数值部分隔开 

例子：已知反码求真值，0的反码

- 已知 $[x]_{\text{反}} = 1,1110$ ，求 x

解：由定义得 $x = [x]_{\text{反}} - (2^{4+1} - 1)$

$$= 1,1110 - 11111$$
$$= -0001$$

- 求 0 的反码

解：设 $x = +0.0000$ ， $[+0.0000]_{\text{反}} = 0.0000$

$$x = -0.0000, [-0.0000]_{\text{反}} = 1.1111$$

同理，对于整数 $[+0]_{\text{反}} = 0,0000$ ， $[-0]_{\text{反}} = 1,1111$

$$[+0]_{\text{反}} \neq [-0]_{\text{反}}$$

口诀3（原码与反码一一映射）

1) 正数不变

2) 负数：符号位不变，数值位按位取反

注意：

1) 原码与反码是一一映射，它们所能表示的数的范围一致。

2) 位数相同，补码也比反码所能表示的数的范围更广。

三种机器数的小结

- 最高位为符号位，**书写上**用 “,” （整数）或 “.” （小数）将数值部分和符号位隔开
- 对于**正数**：符号位为0，原码 = 补码 = 反码
- 对于**负数**：补码 \leftarrow 原码的方法：
 - 除符号位外，每位取反，末位加 1
 - 扫描法：符号位不变，自右向左从最先遇到的1保持不变，从这个1的左边开始，每位取反
- 对于**负数**：补码=反码+1（特例：最小负数不适用）
反码=原码按位取反（符号位除外）

三种机器数的小结

- 注意口诀不适合特例（2种情况：原码-0和补码的最小负数不能用口诀），补码比原码表示的范围广，比原码多了一个最小的负数。

1) 比如8位寄存器（含符号位1位）存储整数

- 该寄存器所能表示的补码范围是-128~127，其中的-128没有原码。
- 补码的0唯一，原码分+0和-0，都对应补码的0。

2) 比如8位寄存器（含符号位1位）存储小数，则该寄存器所能表示的补码范围是-1 ~ $(1-2^{-7})$ ，其中的-1没有原码。

例子：机器数的真值

- 设机器数字长为8位(其中1位为符号位);对于整数,当其分别代表无符号数、原码、补码和反码时,对应的十进制真值范围如下:

二进制代码	无符号数 对应的真值	原码对应 的真值	补码对应 的真值	反码对应 的真值
00000000	0	+0	± 0	+0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
⋮	⋮	⋮	⋮	⋮
01111111	127	+127	+127	+127
10000000	128	-0	-128	-127
10000001	129	-1	-127	-126
⋮	⋮	⋮	⋮	⋮
11111101	253	-125	-3	-2
11111110	254	-126	-2	-1
11111111	255	-127	-1	-0

请大家做一下雨课堂练习题

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

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

<I> $[y]_{\text{补}} = 0 \cdot y_1 y_2 \cdots y_n$

$$y = 0 \cdot y_1 y_2 \cdots y_n$$

$$-y = -0 \cdot y_1 y_2 \cdots y_n$$

$$[-y]_{\text{补}} = 1 \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_n} + 2^{-n}$$

<II> $[y]_{\text{补}} = 1 \cdot y_1 y_2 \cdots y_n$

$$[y]_{\text{原}} = 1 \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_n} + 2^{-n}$$

$$y = -(0 \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_n} + 2^{-n})$$

$$-y = 0 \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_n} + 2^{-n}$$

$$[-y]_{\text{补}} = 0 \cdot \overline{y_1} \overline{y_2} \cdots \overline{y_n} + 2^{-n}$$

$[y]_{\text{补}}$ 连同符号位在内，
每位取反，末位加1，
即得 $[-y]_{\text{补}}$

注意：本例题不适用 $y = -1.00 \dots 0$

移码表示法

- 补码表示很难直接判断其真值大小

如	十进制	二进制	补码	
	$x = +21$	+10101	0,10101	错
	$x = -21$	-10101	1,01011	大
	$x = +31$	+11111	0,11111	错
	$x = -31$	-11111	1,00001	大
以上 $x + 2^5$		+10101 + 100000 = 110101		大 正确
		-10101 + 100000 = 001011		大 正确
		+11111 + 100000 = 111111		大 正确
		-11111 + 100000 = 000001		大 正确

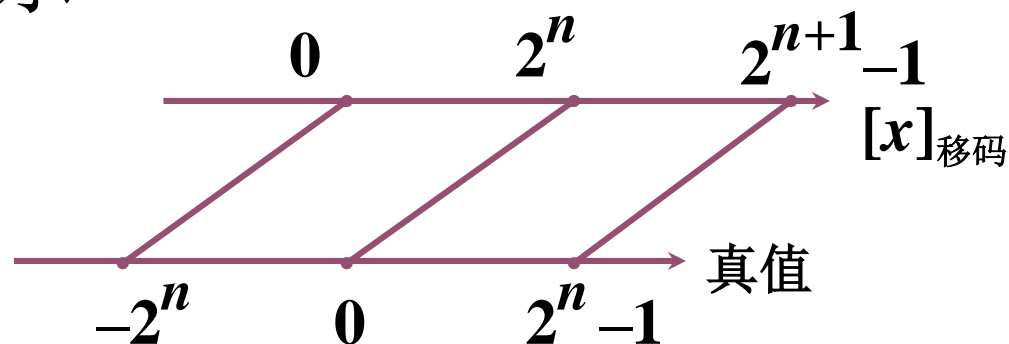
移码表示法：二进制整数

- 定义

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

其中： x 为真值， n 为 整数的位数

- 移码在数轴上的表示



- 例：

$$x = 10100$$

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

$$x = -10100$$

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

用 逗号 将符号位
和数值位隔开

移码表示法：二进制整数

- 移码定义

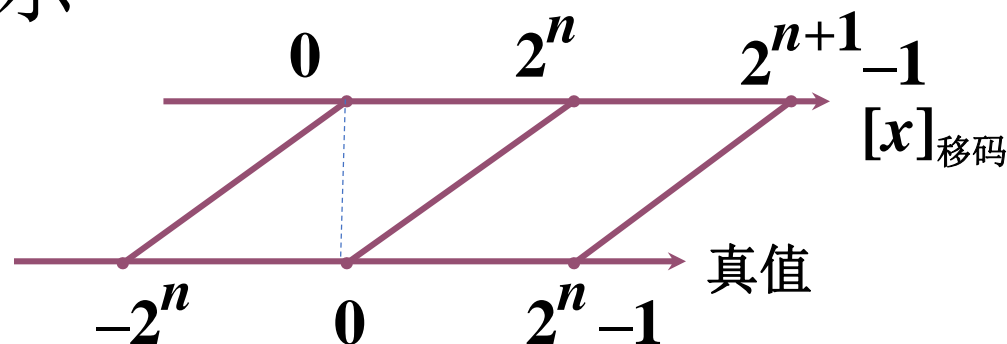
$$[x]_{\text{移}} = 2^n + x \quad (2^n > x \geq -2^n)$$

其中： x 为真值， n 为 整数的位数

小数的移码定义呢？



- 移码在数轴上的表示



- 例：

$$x = 10100$$

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

$$x = -10100$$

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

用 逗号 将符号位
和数值位隔开

移码和补码的比较

设 $x = +1100100$

$$[x]_{\text{移}} = 2^7 + 1100100 = \mathbf{1},1100100$$

$$[x]_{\text{补}} = \mathbf{0},1100100$$

设 $x = -1100100$

$$[x]_{\text{移}} = 2^7 + (-1100100) = \mathbf{0},0011100$$

$$[x]_{\text{补}} = 2^{7+1} - 1100100 = \mathbf{1},0011100$$

补码与移码只差一个符号位

口诀4（补码与移码）

无论正负，移码与补码关系都是：

只有符号位对调（0变1，1变0）

- 补码与移码一一映射，所能表示的数的范围一致。
- 原码与反码一一映射，所能表示的数的范围一致。
- 补码和移码表示范围都比原码和反码广，多了一个最小的负数。
- 补码与移码一一映射的关系仅限于整数范围，因为移码只表示整数。

真值、补码和移码的对照表

真值 x ($n=5$)	$[x]_{\text{补}}$	$[x]_{\text{移}}$	$[x]_{\text{移}}$ 对应的 十进制整数
- 1 0 0 0 0	1 0 0 0 0	0 0 0 0 0	0
- 1 1 1 1 1	1 0 0 0 1	0 0 0 0 1	1
- 1 1 1 1 0	1 0 0 0 1 0	0 0 0 0 1 0	2
⋮	⋮	⋮	⋮
- 0 0 0 0 1	1 1 1 1 1 1	0 1 1 1 1 1	31
± 0 0 0 0 0	0 0 0 0 0 0	1 0 0 0 0 0	32
+ 0 0 0 0 1	0 0 0 0 0 1	1 0 0 0 0 1	33
+ 0 0 0 1 0	0 0 0 0 1 0	1 0 0 0 1 0	34
⋮	⋮	⋮	⋮
+ 1 1 1 1 0	0 1 1 1 1 0	1 1 1 1 1 0	62
+ 1 1 1 1 1	0 1 1 1 1 1	1 1 1 1 1 1	63

移码的特点

续前表, $n=5$

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

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

$$[+0]_{\text{移}} = [-0]_{\text{移}}$$

最小真值 $-2^5 = -100000$ 对应的移码为 $2^5 - 100000 = 000000$

最小真值的移码为全 0

可用**移码思想**表示浮点数的阶码, 便于判断浮点数阶码大小

讲解雨课堂练习题（预习效果检测）

第二章 计算机中数的表示

- 计算机中数的表示
 - 无符号数和有符号数
 - 定点表示和浮点表示
 - IEEE754标准
 - 算数移位与逻辑移位

定点表示

- 小数点按约定方式标出
- 定点表示



或



定点机

小数定点机

整数定点机

原码 $-(1 - 2^{-n}) \sim +(1 - 2^{-n})$

$-(2^n - 1) \sim +(2^n - 1)$

补码 $-1 \sim +(1 - 2^{-n})$

$-2^n \sim +(2^n - 1)$

反码 $-(1 - 2^{-n}) \sim +(1 - 2^{-n})$

$-(2^n - 1) \sim +(2^n - 1)$

注意：n是数值部分的位数

浮点表示

$N = S \times r^j$ 浮点数的一般形式

S 尾数 j 阶码 r 基数（基值）

计算机中 r 取 2、4、8、16 等

当 $r = 2$ $N = 11.0101$ 二进制表示

✓ $= 0.110101 \times 2^{10}$ 规格化数

$= 1.10101 \times 2^1$

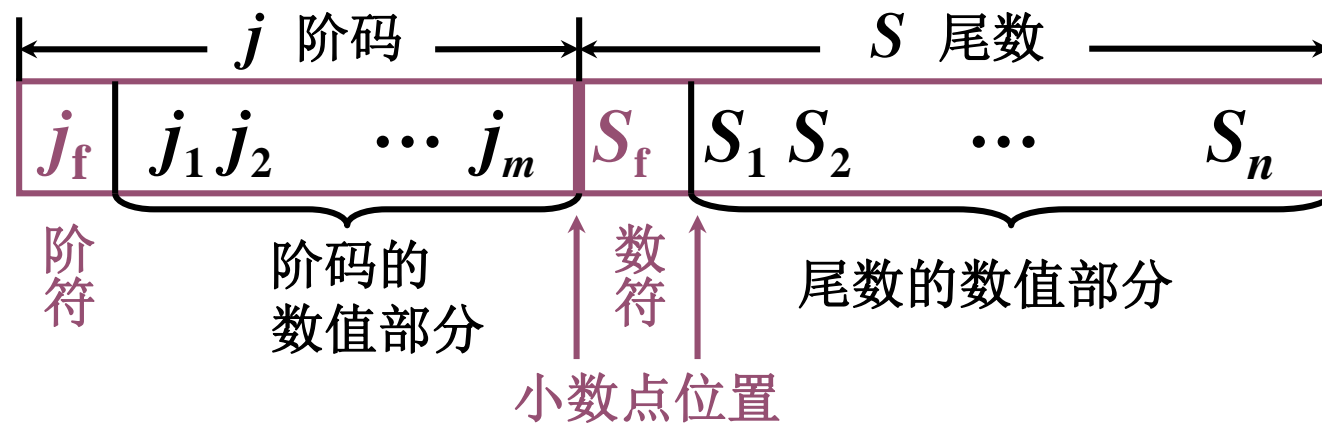
$= 1101.01 \times 2^{-10}$

✓ $= 0.00110101 \times 2^{100}$

计算机中 S 小数、可正可负

j 整数、可正可负

浮点数的表示形式



S_f 代表浮点数的符号

n 其位数反映浮点数的精度

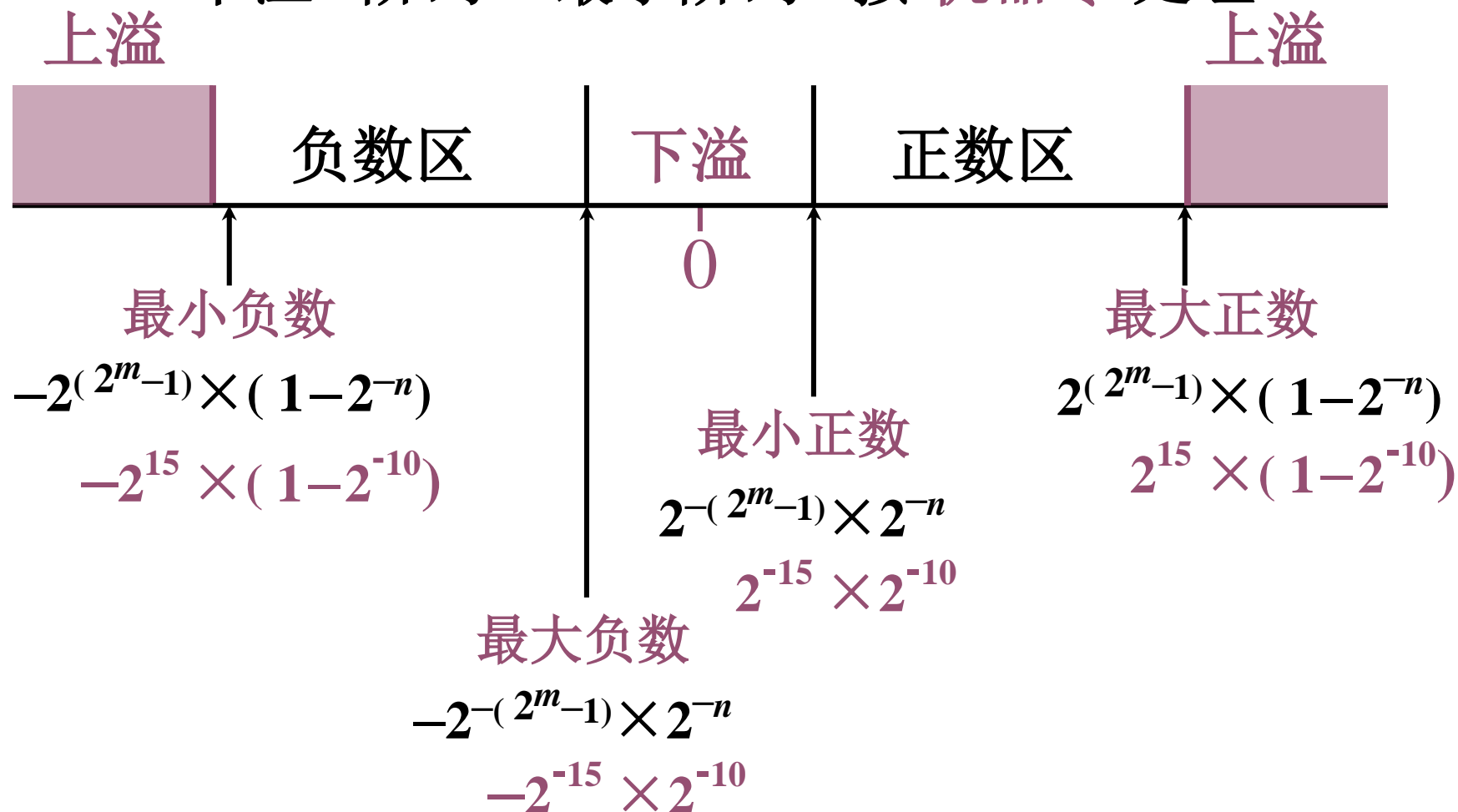
m 其位数反映浮点数的表示范围

j_f 和 m 共同表示小数点的实际位置

浮点数（**原码**）的表示范围（唐书P230）

上溢 阶码 > 最大阶码

下溢 阶码 < 最小阶码 按 **机器零** 处理



设 $m = 4$
 $n = 10$

练习

- 设**原码表示的**机器数字长为 24 位，欲表示 ± 3 万的十进制数，试问在保证数的最大精度的前提下，除阶符、数符各取 1 位外，阶码、尾数各取几位？

根本原理：
阶码的位数决定范围
尾数的位数决定精度

解： $\because 2^{14} = 16384 \quad 2^{15} = 32768$

\therefore **15** 位二进制数可反映 ± 3 万之间的十进制数

$$2^{15} \times 0.\underbrace{\times \times \times \dots \times \times \times}_{15\text{位}}$$

$m = 4, 5, 6, \dots$

满足 **最大精度** 可取 $m = 4, n = 18$

- 浮点数的规格化形式

基数不同，浮点数的规格化形式不同

$r = 2$ 尾数最高位为 1

$r = 4$ 尾数最高 2 位不全为 0

$r = 8$ 尾数最高 3 位不全为 0

- 浮点数的规格化

特别注意：尾数移位不同于小数点的移位，所以阶码加减有所不同

$r = 2$ 左规 尾数左移 1 位，阶码减 1

右规 尾数右移 1 位，阶码加 1

$r = 4$ 左规 尾数左移 2 位，阶码减 1

右规 尾数右移 2 位，阶码加 1

$r = 8$ 左规 尾数左移 3 位，阶码减 1

右规 尾数右移 3 位，阶码加 1

基数 r 越大，可表示的浮点数的范围越大

基数 r 越大，浮点数的精度降低

- 规格化数的定义

$$r = 2 \quad \frac{1}{2} \leq |S| < 1$$

- 规格化数的判断

$S > 0$	规格化形式	$S < 0$	规格化形式
真值	$0.1 \times \times \dots \times$	真值	$-0.1 \times \times \dots \times$
原码	$0.\boxed{1} \times \times \dots \times$	原码	$1.\boxed{1} \times \times \dots \times$
补码	$\boxed{0.1} \times \times \dots \times$	补码	$\boxed{1.0} \times \times \dots \times$
反码	$0.1 \times \times \dots \times$	反码	$1.0 \times \times \dots \times$

原码 不论正数、负数，第一数位为1

补码 符号位和第一数位不同

- 特例

$$S = -\frac{1}{2} = -0.100 \dots 0$$

$$[S]_{\text{原}} = 1.100 \dots 0$$

$$[S]_{\text{补}} = \boxed{1.1}00 \dots 0$$

$\therefore [-\frac{1}{2}]_{\text{补}}$ 不是规格化的数

但 $[-\frac{1}{2}]_{\text{原}}$ 是规格化的数

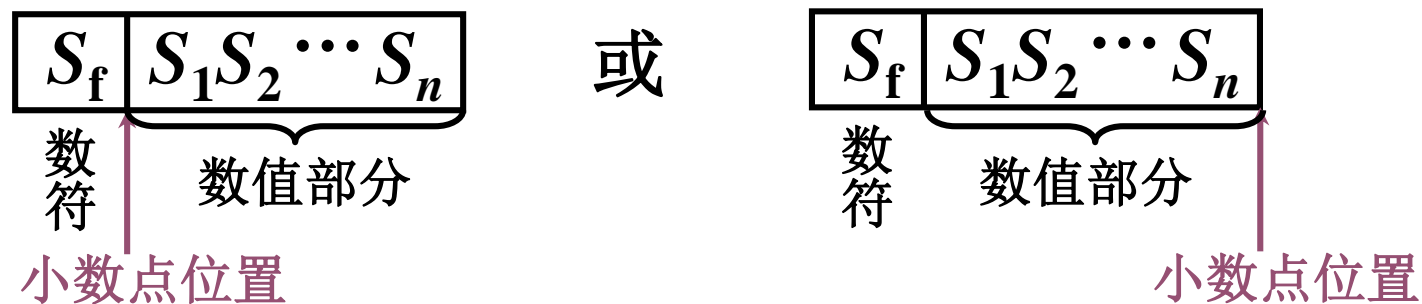
$$S = -1$$

$$[S]_{\text{补}} = \boxed{1.0}00 \dots 0$$

$\therefore [-1]_{\text{补}}$ 是规格化的数

表示范围：原码对称，补码不对称

- 小数点按约定方式标出
- 定点表示



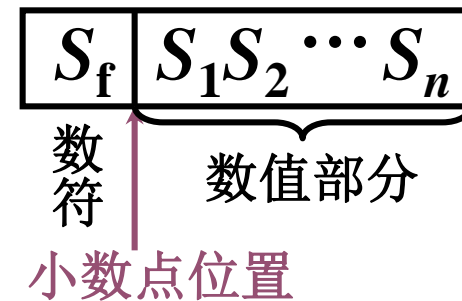
定点机	小数定点机	整数定点机
原码	$-(1 - 2^{-n}) \sim +(1 - 2^{-n})$	$-(2^n - 1) \sim +(2^n - 1)$
补码	$-1 \sim +(1 - 2^{-n})$	$-2^n \sim +(2^n - 1)$
反码	$-(1 - 2^{-n}) \sim +(1 - 2^{-n})$	$-(2^n - 1) \sim +(2^n - 1)$

尾数规格化范围：原码对称，补码不对称

- 规格化只针对浮点数尾数进行

（而尾数其实就是定点小数）

- 规格化后的定点小数表示范围（分两段）



表示形式

小数定点机

原码 $-(1 - 2^{-n}) \sim -\frac{1}{2}$ \cup $+\frac{1}{2} \sim +(1 - 2^{-n})$

补码 $-1 \sim -\frac{1}{2} - 2^{-n}$ \cup $+\frac{1}{2} \sim +(1 - 2^{-n})$

（注：原码到补码的转换中，负数部分通过红色箭头标注了 -2^{-n} 的调整量）

m和n的含义见PPT46页

- 例13. 设 $m = 4$, $n = 10$, $r = 2$, 求尾数规格化后的浮点数表示范围（阶码与尾数都是原码表示）

$$\text{最大正数} \quad 2^{0,1111} \times (+0.\underbrace{1111111111}_{10 \text{ 个 } 1}) = 2^{15} \times (1 - 2^{-10})$$

$$\text{最小正数} \quad 2^{1,1111} \times (+0.\underbrace{1000000000}_{9 \text{ 个 } 0}) = 2^{-15} \times 2^{-1} = 2^{-16}$$

$$\text{最大负数} \quad 2^{1,1111} \times (-0.\underbrace{1000000000}_{9 \text{ 个 } 0}) = -2^{-15} \times 2^{-1} = -2^{-16}$$

$$\text{最小负数} \quad 2^{0,1111} \times (-0.\underbrace{1111111111}_{10 \text{ 个 } 1}) = -2^{15} \times (1 - 2^{-10})$$

m和n的含义见PPT46页

- 例13（发散思维）. 设 $m = 4$, $n = 10$, $r = 2$, 求尾数规格化后的浮点数表示范围（阶码与尾数都是补码表示）

本题重要

最大正数 $2^{0,1111} \times (+0.1111111111) = 2^{15} \times (1 - 2^{-10})$
即：补码是0.1111111111

最小正数 $2^{1,0000} \times (+0.1000000000) = 2^{-16} \times 2^{-1} = 2^{-17}$
即：补码是0.1000000000

最大负数 $2^{1,0000} \times (-0.10000000001) = -2^{-16} \times (2^{-1} + 2^{-10})$
即：补码是1.0111111111

最小负数 $2^{0,1111} \times (-1.0000000000) = -2^{15} \times 1$
即：补码是1.0000000000

- 例14. 将 $+\frac{19}{128}$ 写成二进制定点数、浮点数及在定点机和浮点机中的机器数形式。其中数值部分均取 10 位，数符取 1 位，浮点数阶码取 5 位（含1位阶符）。

解： 设 $x = +\frac{19}{128}$

二进制形式 $x = 0.0010011$

定点表示 $x = 0.0010011\ 000$

浮点规格化形式 $x = 0.1001100000 \times 2^{-10}$

定点机中 $[x]_{\text{原}} = [x]_{\text{补}} = [x]_{\text{反}} = 0.0010011000$

浮点机中 $[x]_{\text{原}} = 1, 0010; 0. 1001100000$

$[x]_{\text{补}} = 1, 1110; 0. 1001100000$

$[x]_{\text{反}} = 1, 1101; 0. 1001100000$

- 例15. 将 -58 表示成二进制定点数和浮点数，并写出它在定点机和浮点机中的三种机器数及阶码为移码、尾数为补码的形式（其他要求同上例）。

解： 设 $x = -58$

二进制形式 $x = -111010$

定点表示 $x = -0000111010$

浮点规格化形式 $x = -(0.1110100000) \times 2^{110}$

定点机中

$[x]_{\text{原}} = 1, 0000111010$

$[x]_{\text{补}} = 1, 1111000110$

$[x]_{\text{反}} = 1, 1111000101$

浮点机中

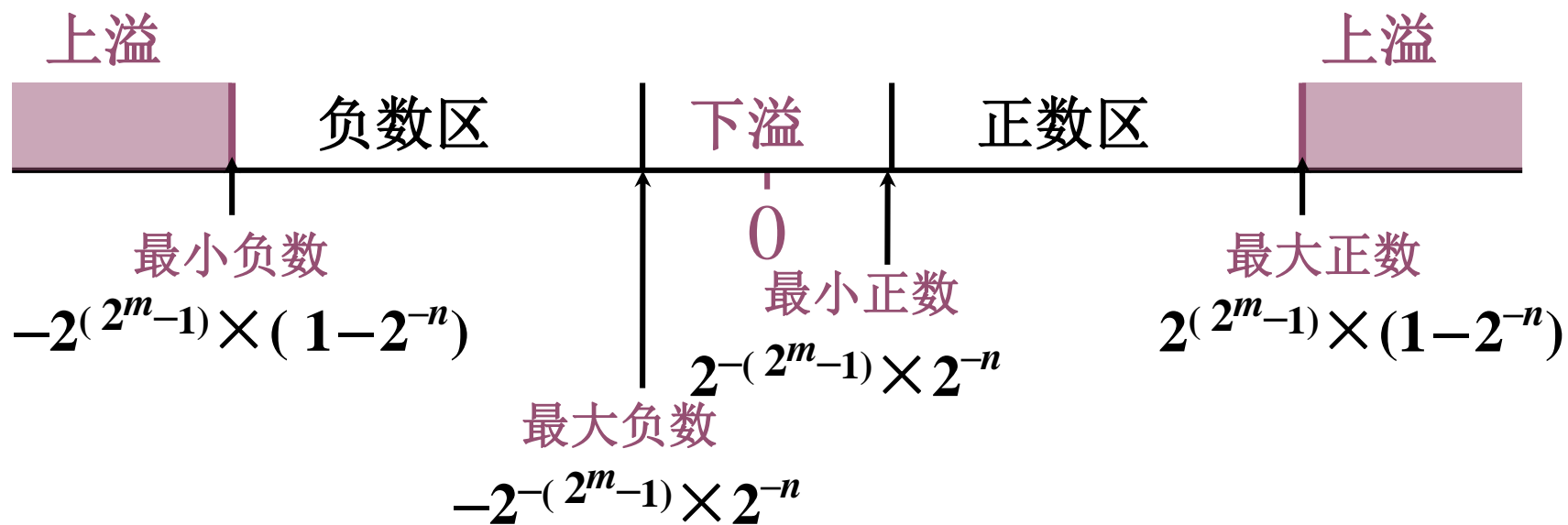
$[x]_{\text{原}} = 0, 0110; 1. 1110100000$

$[x]_{\text{补}} = 0, 0110; 1. 0001100000$

$[x]_{\text{反}} = 0, 0110; 1. 0001011111$

$[x]_{\text{阶移、尾补}} = 1, 0110; 1. 0001100000$

- 例16. 写出对应下图所示的原码表示的浮点数范围对应的补码形式。设 $n = 10$, $m = 4$, 阶符、数符各取1位。



已知:

真值

最大正数 $2^{15} \times (1-2^{-10})$

最小正数 $2^{-15} \times 2^{-10}$

最大负数 $-2^{-15} \times 2^{-10}$

最小负数 $-2^{15} \times (1-2^{-10})$

解:

补码

0,1111; 0.1111111111

1,0001; 0.0000000001

1,0001; 1.1111111111

0,1111; 1.0000000001

机器零

- 当浮点数尾数为 0 时，不论其阶码为何值，按机器零处理
- 当浮点数阶码小于它所表示的最小数时，按机器零处理

例如 $m = 4$, $n = 10$ 时

说法一

当阶码和尾数都用补码表示时，机器零为：

$\times, \times \times \times \times; 0.00 \dots 0$

或者阶码 < -16 ，按照机器零处理

说法二

当阶码用移码，尾数用补码表示时，机器零为

$0, 0000; 0.00 \dots 0$

有利于机器中“判 0”电路的实现

溢出定义

- 溢出的概念（唐书P238）

计算机中，这种超出机器字长的现象就叫溢出。

- 定点数溢出

补码定点加减运算判断溢出的两种方法：

1) 用一位符号位判断溢出

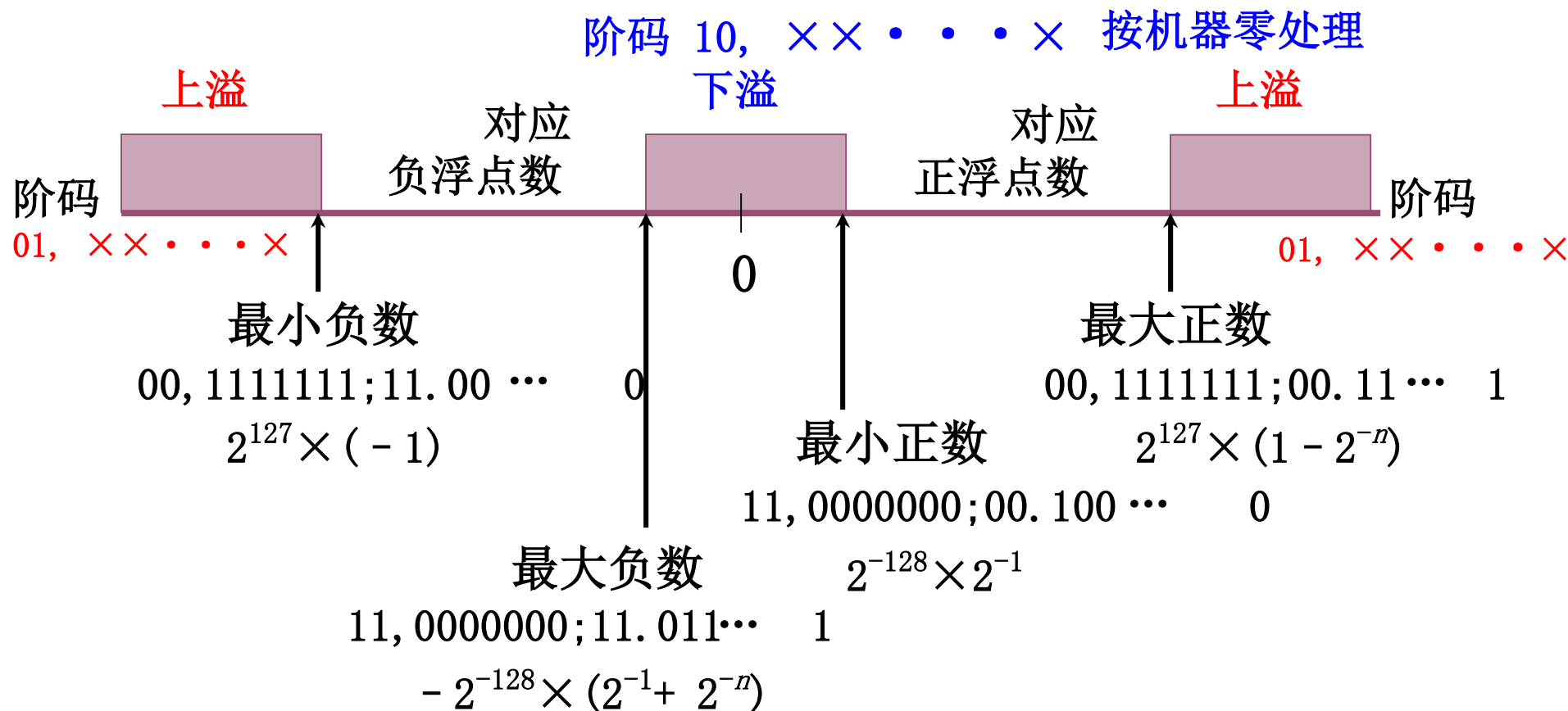
2) 用两位符号位判断溢出

- 浮点数溢出

当浮点数阶码大于最大阶码时，称为上溢，此时机器停止运算，进行中断处理；当浮点数阶码小于最小阶码时，称为下溢，此时溢出的数绝对值很小，尾数强制清零，按机器零处理。⁶²

溢出判断（重点）

设机器数为补码，尾数为规格化形式，并假设阶符取 2 位，阶码的数值部分取 7 位，数符取 2 位，尾数的数值部分取 n 位，则该补码在数轴上的表示为



唐书教材中的一些问题

• P230 正确

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

• P232-233

值得注意的是,当一个浮点数尾数为0时,不论其阶码为何值;或阶码等于或小于它所能表示的最小数时,不管其尾数为何值,机器都把该浮点数作为零看待,并称之为“机器零”。如果浮点数的阶码用移码表示,尾数用补码表示,则当阶码为它所能表示的最小数 2^{-m} (式中 m 为阶码的位数) 且尾数为0时,其阶码(移码)全为0,尾数(补码)也全为0,这样的机器零为000...0000,全零表示有利于简化机器中判“0”电路。

错误

改正1: 去掉“等于或”

改正2: -2^m (式中 m 为阶码中数值部分的位数)

• 唐书P223中的证明

小数补码的定义为

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

式中 x 为真值。

例如，当 $x=0.1001$ 时， $[x]_{\text{补}}=0.1001$

当 $x=-0.0110$ 时，

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

当 $x=0$ 时，

$$[+0.0000]_{\text{补}} = 0.0000$$

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

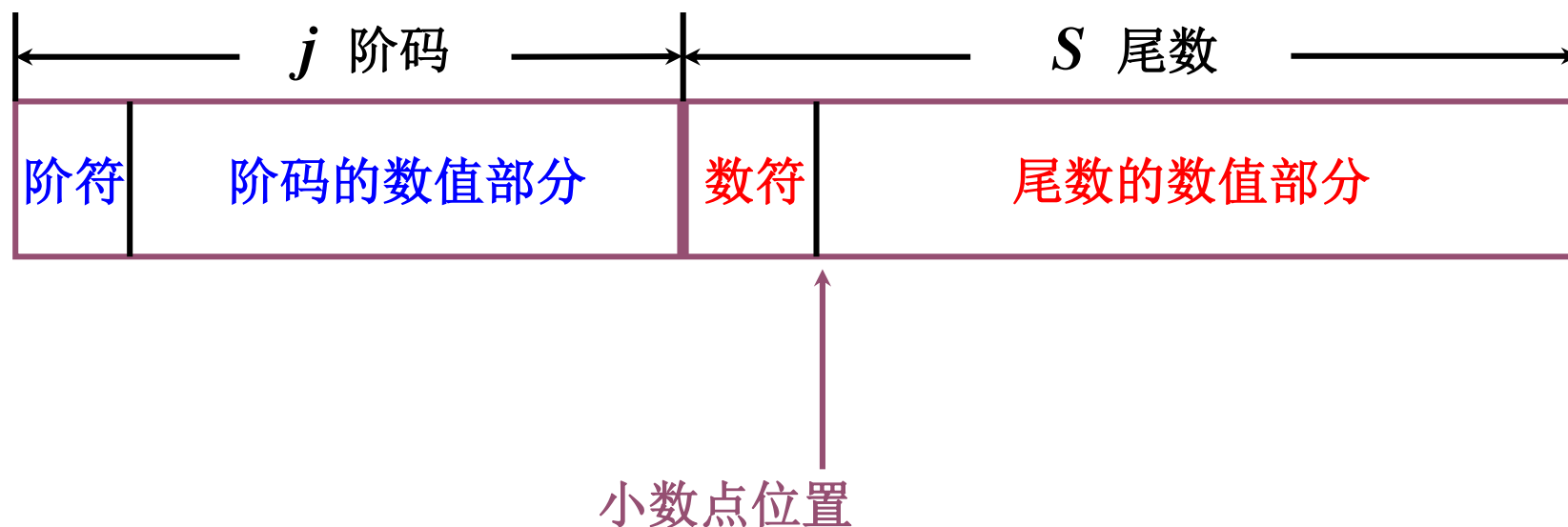
总之：补码中的零只有一种表示形式

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

第二章 计算机中数的表示

- 计算机中数的表示
 - 无符号数和有符号数
 - 定点表示和浮点表示
 - IEEE754标准
 - 算数移位与逻辑移位

浮点数的表示形式（回顾）



例：假设一个浮点数的阶码和尾数共16位，并且都用补码表示，则：

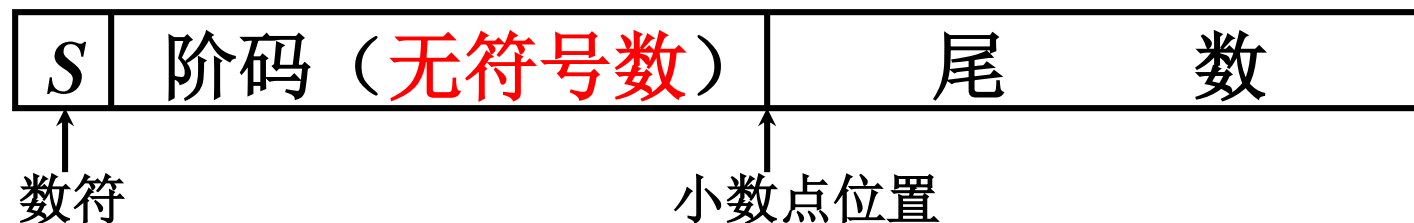
$$(0.75 \times 2^{-16})_{10} = (2^{1,0000} \times \underline{0.1100000000})_2$$

IEEE754简介

- 目前主流CPU都用IEEE754格式表示浮点数
- IEEE754
 - William Kahan从1976年开始为Intel设计(1989获图灵奖)
 - 1985年成为浮点运算的统一标准
 - 具有快速、易于实现、精度损失小等特点
 - 相比前面的浮点数表示更具有先进性，用于C语言中的float，double



IEEE 754 标准



尾数为规格化表示

非“0”的有效位最高位为“1”（隐含）

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

临时实数又称为扩展精度浮点数，它没有隐含位，尾数真值就等于 64 位尾数数值

IEEE 754浮点数标准

- 单精度 (32-bit)

31	30	29	28	27	26	25	24	23	22 ~ 0										
s	8位指数（无符号数）								23位尾数（无符号数）										

- 双精度 (64-bit)

63	62	61	60	59	58	57	56	55	54	53	52	51~0									
s	11位指数（无符号数）											52位尾数（无符号数）									

IEEE754浮点数：单精度为例

单精度浮点数值分类

1. 规格化的



2. 非规格化的



3. 特殊值

3a. 无穷大



3b. NaN(Not a Number)



IEEE 754浮点数：单精度为例



	指数	尾数	表示对象	IEEE754到十进制的换算方法
非规格化	0	0	0	规定（符号位不同，存在+0.0和-0.0）
	0	非0	正负非规格化数	正负非规格化数 = $(-1)^s \times (\text{尾数}_2) \times 2^{(0 - 126)}$ (s代表符号位，0为正数，1为负数)
规格化	[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^s \times (1 + \text{尾数}_2) \times 2^{(\text{指数} - 127)}$
特殊值	255	0	正负无穷 (inf)	规定
	255	非零	NaN	规定

IEEE754浮点数：真值转二进制

• 例题

- 将十进制 -0.75 转为单精度 IEEE754 格式二进制

解：

根据十进制小数转二进制小数算法： $-0.75_{10} = -0.11_2$

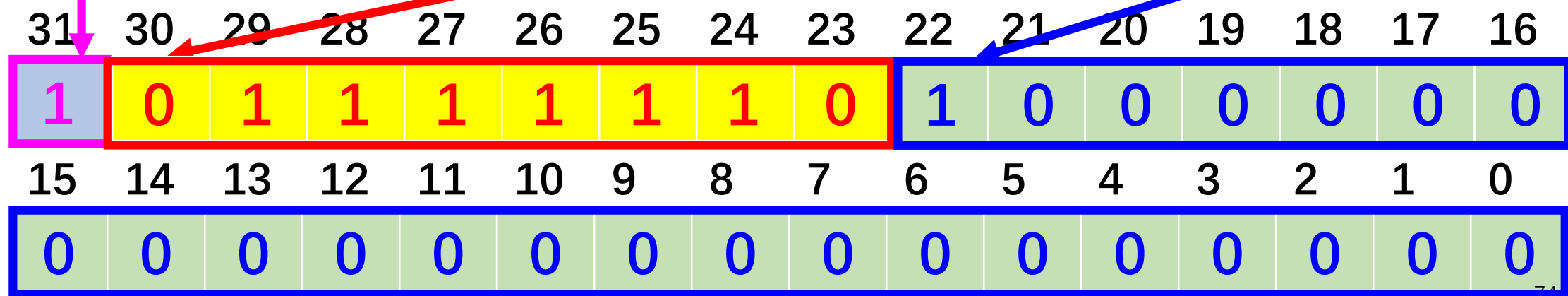
二进制 $-0.11 = -\underline{1}.1 \times 2^{-1}$

①符号位：1

②指数部分： $(-1+127)_{10} = (126)_{10} = (01111110)_2$

③尾数部分： 0.1_2

注意：小数点左边的1会被隐藏起来



十六进制：0xBF400000

IEEE754浮点数：二进制转真值

- 例题

- 将二进制IEEE754浮点数表示转换为十进制浮点数（空白为0）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- 解：

符号位为1，（IEEE754）指数字段为129，尾数字段为 $2^{-2} = 0.25$ ，还原成十进制：

$$\begin{aligned} -2^{(\text{指数} - 127)} \times (1 + \text{尾数}) &= -2^{(129 - 127)} \times (1 + 0.25) \\ &= -2^2 \times 1.25 \\ &= -5.0 \end{aligned}$$

IEEE 754浮点数：真值转二进制

IEEE754相关网址: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Tools & Thoughts

IEEE-754 Floating Point Converter

Translations: de

This page allows you to convert between the decimal representation of numbers (like "1.02") and the binary format used by all modern CPUs (IEEE 754 floating point).

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	-1	2 ⁻¹	1.5
Encoded as:	1	126	4194304
Binary:	<input checked="" type="checkbox"/>	<input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

You entered

Value actually stored in float:

Error due to conversion:

Binary Representation

Hexadecimal Representation

+1

-1

Update

There has been an update in the way the number is displayed. Previous version would give you the represented value as a possibly rounded decimal number and the same number with the increased pre float. Now the original number is shown (either as the number that was entered, or as a possibly rounded decimal string) as well as the actual full precision decimal number that the float value is represented. A nice example to see this behaviour. The difference between both values is shown as well, so you can easier tell the difference between what you entered and what you get in IEEE-754.

Translations: [de](#)

This page allows you to convert between the decimal representation of numbers (like "1.02") and the binary format used by all modern CPUs (IEEE 754 floating point).

IEEE 754浮点数：单精度为例



指数	尾数	表示对象	换算方法
0	0	0	规定（符号位不同，存在+0.0和-0.0）
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{-126}$ (S代表符号位，1为负数，0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

IEEE 754浮点数：正负浮点数（即规格化）

十进制科学记数法：

782300=7.823×10⁵

0.00012=1.2×10⁻⁴

10000=1×10⁴

- 正负浮点数 = $(-1)^S \times (1 + \text{尾数}_2) \times 2^{(\text{指数} - 127)}$
- 尾数前加1？
 - 考虑到科学计数法，小数点前要求是1，这个1称为前导数。为了打包更多的位到数中，就在二进制表示中省略了前导数，默认小数点前有1。
 - 有效位数：隐含的1加上尾数共有多少位。对单精度来说，有效位数是 24 位（隐含的1和 23 位尾数）；对双精度来说，是 53 位（1 + 52）。
 - 由于 0（和非规格化数）没有前导数，所以被赋予特殊的指数 0，硬件不会给它附加 1

指数	尾数	表示对象	换算方法
0	0	0	规定
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{(0 - 126)}$ (S代表符号位, 1为负数, 0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

IEEE 754浮点数：正负浮点数的偏移

- 正负浮点数 = $(-1)^S \times (1 + \text{尾数}_2) \times 2^{(\text{指数} - 127)}$
- 指数 - 127?
 - 使用移码的思想（偏移值是127）。二进制表示中的指数部分是又一种移码（不同于唐书定义），可以直接进行大小比较。如果两个数的符号相同，那么具有更大二进制指数的数就更大。
 - 对于真值而言，其实际的“指数”范围： $[1-127: 254-127] = [-126: 127]$

指数	尾数	表示对象	换算方法
0	0	0	规定
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{(0 - 126)}$ (S代表符号位, 1为负数, 0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

IEEE 754浮点数：正负非规格化数

- 正负非规格化数 = $(-1)^S \times (\text{尾数}_2) \times 2^{(0 - 126)}$
- 什么是非规格化数？
 - 规格化数：科学计数法中整数部分前导不等于0 的数称为规格化数；
 - 非规格化数：整数部分前导为 0 的数
- 非规格化数的绝对值比规格化正负浮点数绝对值更小
 - 对于正负浮点数来说，若二进制指数部分为1，则对应的十进制真值指数部分为 -126，和非规格化数相同。但浮点数尾数有前导1，导致浮点数绝对值更大。

十进制科学记数法：
782300=7.823×10⁵
0.00012=1.2×10⁻⁴
10000=1×10⁴

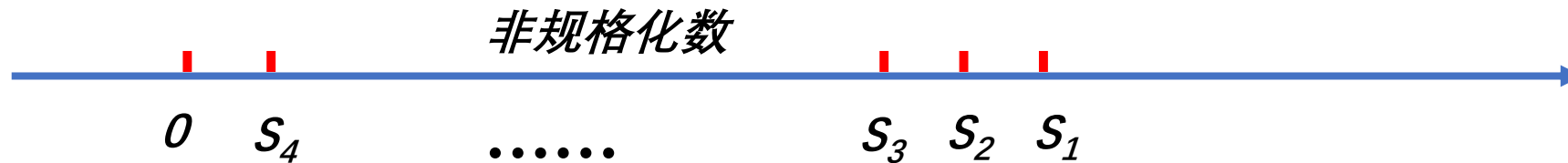
指数	尾数	表示对象	换算方法
0	0	0	规定
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{(0 - 126)}$ (S代表符号位, 1为负数, 0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

IEEE 754单精度浮点数：正负非规格化数

- 正负（规格化）浮点数 = $(-1)^S \times (1 + \text{尾数}_2) \times 2^{(\text{指数} - 127)}$
 - 正负非规格化数 = $(-1)^S \times (\text{尾数}_2) \times 2^{(0 - 126)}$
- 尾数23位

最小正浮点数： $S_2 = (1 + 0_2) * 2^{(1 - 127)} = 2^{-126}$

第二小正浮点数： $S_1 = (1 + 0.0...01_2) * 2^{(1-127)} = 2^{-126} + \underline{2^{-149}}$



最大非规格化数： $S_3 = 0.1...11_2 * 2^{(0 - 126)} = (1 - 2^{-23}) * 2^{-126} = 2^{-126} - 2^{-149}$

最小非规格化正数： $S_4 = 0.0...01_2 * 2^{(0 - 126)} = 2^{-23} * 2^{-126} = \underline{2^{-149}}$

IEEE 754浮点数：正负非规格化数

问题1：判断 $(-1.0) * 2^{-127}$ 是否属于IEEE754规格化浮点数

答案：不是

问题2：为什么偏移是127？不是128？

答案：数值的表示范围和精度折中，
选择127相对来说表示的数范围更广。

指数	尾数	表示对象	IEEE754换算成十进制方法
0	0	0	规定
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{(0 - 126)}$ (S代表符号位，1为负数，0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

IEEE 754 重要结论

- IEEE 754表示的数在数轴上是不均匀的
- 越靠近0， IEEE 754表示的数越密集

指数	尾数	表示对象	换算方法
0	0	0	规定
0	非0	正负非规格化数	正负非规格化数 = $(-1)^S * (\text{尾数}_2) * 2^{(0 - 126)}$ (S代表符号位, 1为负数, 0为正数)
[1: 254]	任意	正负浮点数	正负浮点数 = $(-1)^S * (1 + \text{尾数}_2) * 2^{(\text{指数} - 127)}$
255	0	正负无穷 (inf)	规定
255	非零	NaN	规定

第二章 计算机中数的表示

- 计算机中数的表示

- 无符号数和有符号数
- 定点表示和浮点表示
- IEEE754标准
- 算数移位与逻辑移位

移位运算

- 移位的意义

$$15\text{ m} = 1500\text{ cm}$$

小数点右移 2 位

机器用语 15 相对于小数点 左移 2 位 (注意: 这里是数字移动, 不是小数点)

(小数点不动)

左移 绝对值扩大

右移 绝对值缩小

- 在计算机中, 移位与加减配合, 能够实现乘除运算

算术移位规则（重要）

总原则：无论正负，算术移位，符号位不变

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

- 例17. 设机器数字长为 8 位（含 1 位符号位），写出 $A = +26$ 时，三种机器数左、右移一位和两位后的表示形式及对应的真值，并分析结果的正确性。

解： $A = +26 = +11010$

则 $[A]_{\text{原}} = [A]_{\text{补}} = [A]_{\text{反}} = 0,0011010$

移位操作	机 器 数	对应的真值
	$[A]_{\text{原}} = [A]_{\text{补}} = [A]_{\text{反}}$	
移位前	0,0011010	+26
左移一位	0,0110100	+52
左移两位	0,1101000	+104
右移一位	0,0001101	+13
右移两位	0,0000110	+6

- 例18. 设机器数字长为 8 位（含 1 位符号位），写出 $A = -26$ 时，三种机器数左、右移一位和两位后的表示形式及对应的真值，并分析结果的正确性。

解： $A = -26 = -11010$

原码

移位操作	机 器 数	对应的真值
移位前	1,0011010	- 26
左移一位	1,0110100	- 52
左移两位	1,1101000	- 104
右移一位	1,0001101	- 13
右移两位	1,0000110	- 6

无论正负，算术移位，符号位不变

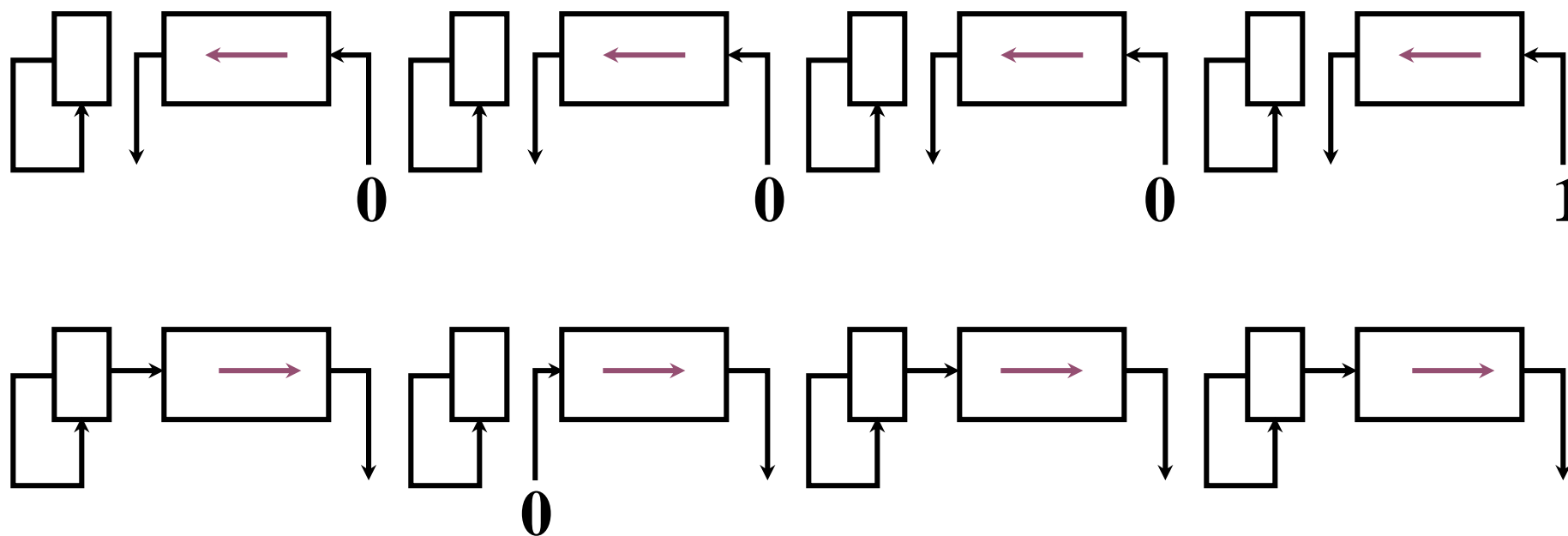
补码

移位操作	机 器 数	对应的真值
移位前	1,1100110	– 26
左移一位	1,1001100	– 52
左移两位	1,0011000	– 104
右移一位	1,1110011	– 13
右移两位	1,1111001	– 7

反码

移位操作	机 器 数	对应的真值
移位前	1,1100101	– 26
左移一位	1,1001011	– 52
左移两位	1,0010111	– 104
右移一位	1,1110010	– 13
右移两位	1,1111001	– 6

3. 算术移位的硬件实现

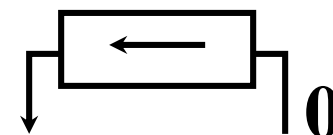


	(a) 真值为正	(b) 负数的原码	(c) 负数的补码	(d) 负数的反码
← 丢 1	出错	出错	正确	正确
→ 丢 1	影响精度	影响精度	影响精度	正确

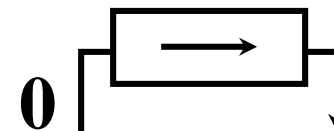
算术移位和逻辑移位的区别

算术移位是有符号数的移位，其特点是符号位不变
逻辑移位是无符号数的移位，其特点是左或右补0

逻辑左移 低位添 0，高位移丢



逻辑右移 高位添 0，低位移丢



例如

01010011

10110010

逻辑左移

10100110

逻辑右移

01011001

算术左移

00100110

算术右移

11011001 (补码)

为了防止高位弄丢，可以设计出用硬件 C_y 来存储高位。

高位 1 移丢

