



## 第六章 计算机的运算方法

# Contents

-  6.1 无符号数和有符号数
-  6.2 数的定点表示和浮点表示
-  6.3 定点运算
-  6.4 浮点四则运算
-  6.5 算术逻辑单元

# 大 纲

## (一)数制与编码

1. 进位计算制及其数据之间的相互转换
2. 定点数的编码表示

## (二)运算方法和运算电路

1. 基本运算部件  
加法器，算术逻辑部件（ALU）
2. 加/减法运算  
补码加减法运算器，  
标志位的生成
3. 乘除法运算的基本原理，  
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. 带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  
IEEE754标准
2. 浮点数的加/减运算

# 数制与编码

## 1. 进位计算制及其相互转换

**1).** 基 $r$ 数制：用 $r$ 个基本符号（如 $0, 1, 2, \dots, r-1$ ）表示数值 $N$ 。  
 $r$ 为 基数

$$N = D_{m-1} D_{m-2} \dots D_1 D_0 D_{-1} D_{-2} \dots D_{-k}$$

其中， $D_i$ （ $-k \leq i \leq m-1$ ）为基本符号，小数点位置隐含在 $D_0$ 与 $D_{-1}$ 之间。

**2).** 有权基 $r$ 数制：每个 $D_i$ 的单位都赋以固定权值 $w_i$ ， $w_i$ 为 $D_i$ 的权。

如果该数制是逢 $r$ 进位，则有

$$N = \sum_{i=-k}^{m-1} D_i \times r^i$$

其中 $r^i$ 为位权，称该数制为 $r$ 进位数制，简称 $r$ 进制，

# 数制与编码

## 3). 进制转换

(1) R进制数  $\Rightarrow$  十进制数

按“权”展开 (a power of R)

例1:  $(10101.01)_2 = 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-2} = (21.25)_{10}$

例2:  $(307.6)_8 = 3 \times 8^2 + 7 \times 8^0 + 6 \times 8^{-1} = (199.75)_{10}$

例1:  $(3A.1)_{16} = 3 \times 16^1 + 10 \times 16^0 + 1 \times 16^{-1} = (58.0625)_{10}$

(2) 十进制数  $\Rightarrow$  R进制数

整数部分和小数部分分别转换

① 整数(integral part)----“除基取余，上右下左”

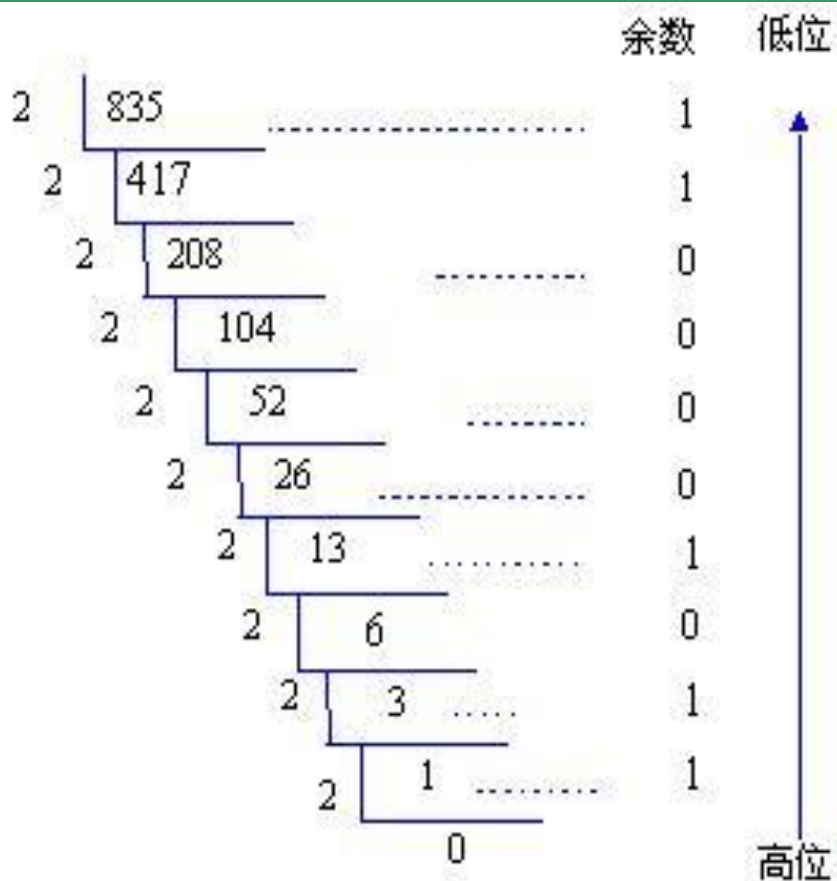
② 小数(fractional part)----“乘基取整，上左下右”

## 二、十进制转换

例1:  $(835.6785)_{10} = (1101000011.1011)_2$

整数-----“除基取余，上右下左”

小数-----“乘基取整，上左下右”



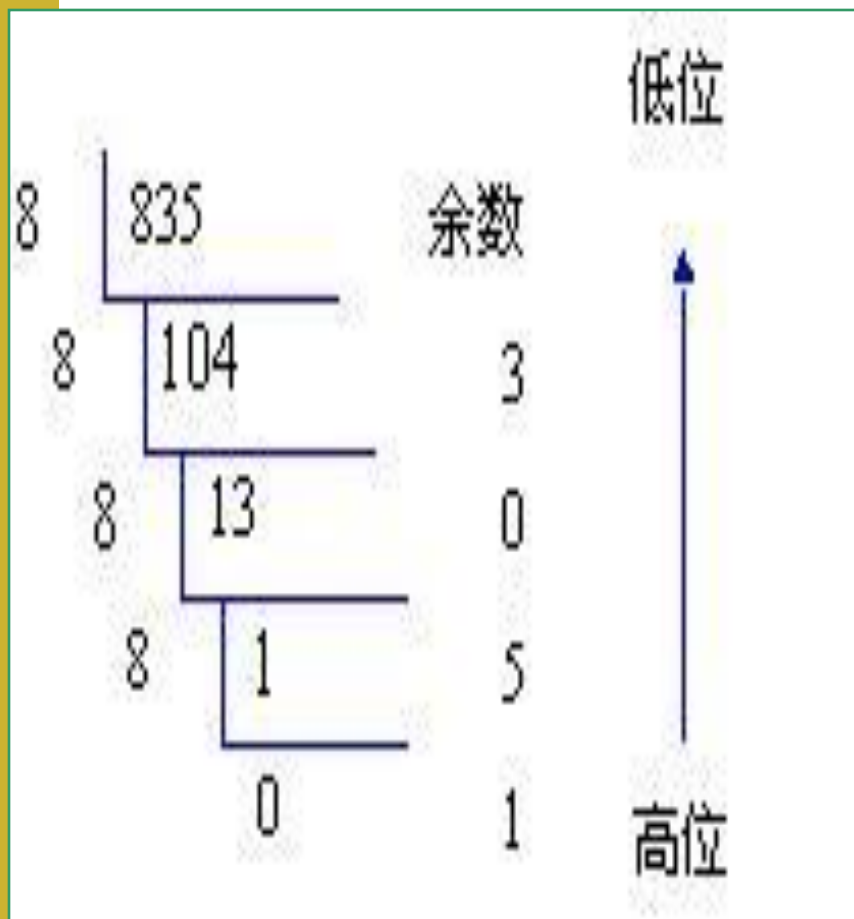
$0.6875 \times 2 = 1.375$	整数部分=1 (高位)
$0.375 \times 2 = 0.75$	整数部分=0 ↓
$0.75 \times 2 = 1.5$	整数部分=1 ↓
$0.5 \times 2 = 1.0$	整数部分=1 (低位)



**例2:  $(835.63)_{10} = (1503.50243...)_{8}$**

整数——“除基取余，上右下左” 小数——“乘基取整，上左下右”

有可能乘积的小数部分总得不到0，此时得到一个近似值。



$0.63 \times 8 = 5.04$	整数部分=5	(高位)
$0.04 \times 8 = 0.32$	整数部分=0	
$0.32 \times 8 = 2.56$	整数部分=2	
$0.56 \times 8 = 4.48$	整数部分=4	
$0.48 \times 8 = 3.84$	整数部分=3	(低位)

### (3) 二/八/十六进制数的相互转换

#### ① 八进制数转换成二进制数

$$(13.724)_8 = (001\ 011\ .\ 111\ 010\ 100)_2 = (1011.1110101)_2$$

#### ② 十六进制数转换成二进制数

$$(2B.5E)_{16} = (00101011\ .\ 01011110)_2 = (101011.0101111)_2$$

#### ③ 二进制数转换成八进制数

$$(0.10101)_2 = (000\ .\ 101\ 010)_2 = (0.52)_8$$

#### ④ 二进制数转换成十六进制数

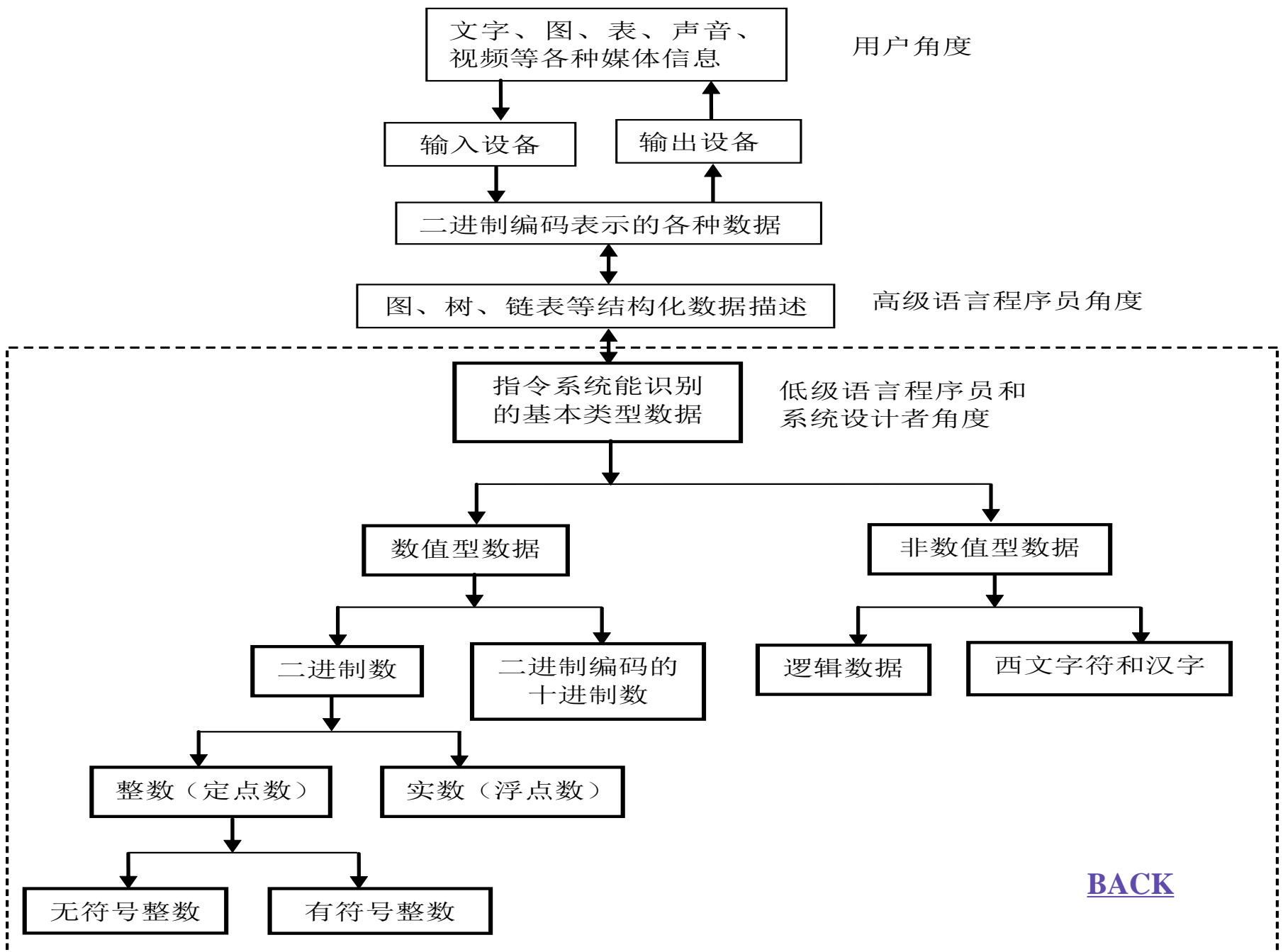
$$(11001.11)_2 = (0001\ 1001\ .\ 1100)_2 = (19.C)_{16}$$



# 信息的二进制编码

- 计算机的外部信息与内部机器级数据
- 机器级数据分两大类：
  - 数值数据：无符号整数、带符号整数、浮点数（实数）、十进制数
  - 非数值数据：逻辑数（包括位串）、西文字符和汉字
- 计算机内部所有信息都用二进制（即：**0**和**1**）进行编码
- 用二进制编码的原因：
  - 制造二个稳定态的物理器件容易
  - 二进制编码、计数、运算规则简单
  - 正好与逻辑命题对应，便于逻辑运算，并可方便地用逻辑电路实现算术运算

数值数据的表示



# 数值数据的表示

- 数值数据表示的三要素

- 进位计数制
- 定、浮点表示
- 如何用二进制编码

即：要确定一个数值数据的值必须先确定这三个要素。

例如，机器数 01011001 的值是多少？ 答案是：不知道！

- 进位计数制

- 十进制、二进制、十六进制、八进制数及其相互转换

- 定/浮点表示（解决小数点问题）

- 定点整数、定点小数
- 浮点数（可用一个定点小数和一个定点整数来表示）

- 定点数的编码（解决正负号问题）

- 原码、补码、反码、移码（反码很少用）

## 6.1 无符号数和有符号数

### 一、无符号数

寄存器的位数（机器字长）

反映无符号数的表示范围



**8 位**

**0 ~ 255**



**16 位**

**0 ~ 65535**

## 二、有符号数

6.1

### 1. 机器数与真值

真值

带符号的数

+ 0.1011

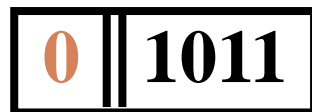
- 0.1011

+ 1100

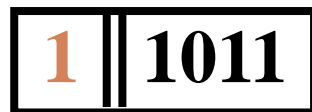
- 1100

机器数

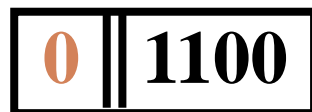
符号数字化的数



小数点的位置



小数点的位置



小数点的位置



小数点的位置

## 2. 原码表示法

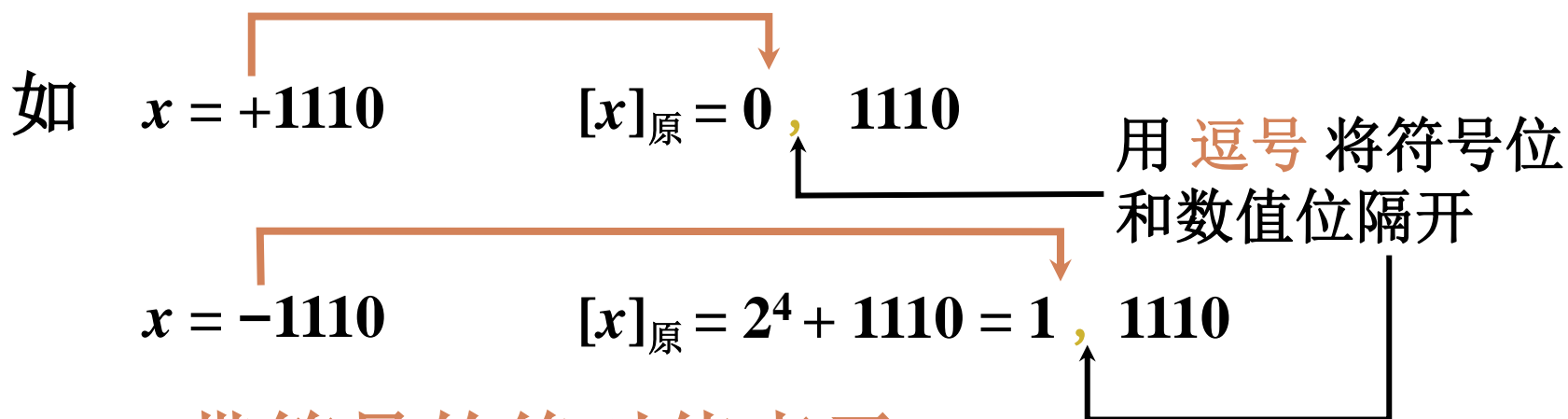
### 6.1

### (1) 定义

整数

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

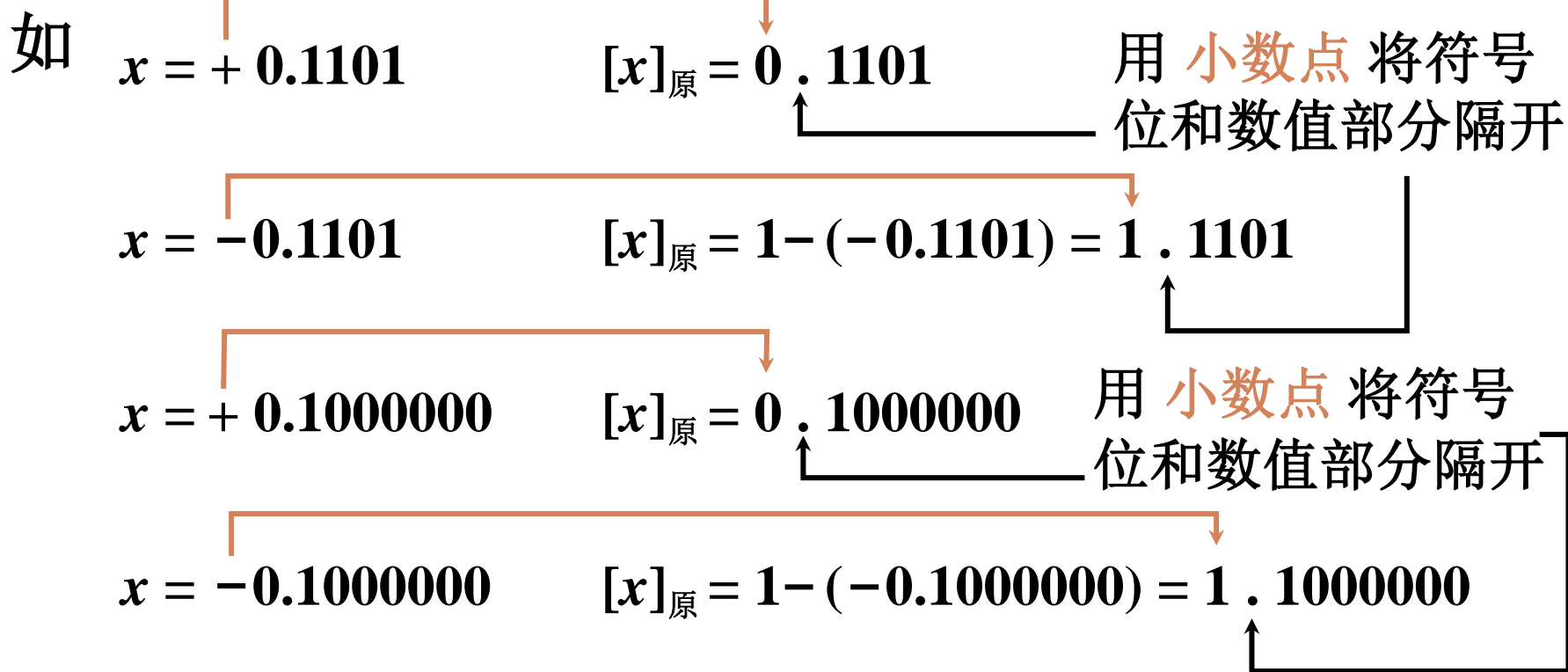
$x$  为真值       $n$  为整数的位数



带符号的绝对值表示

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

$x$  为真值

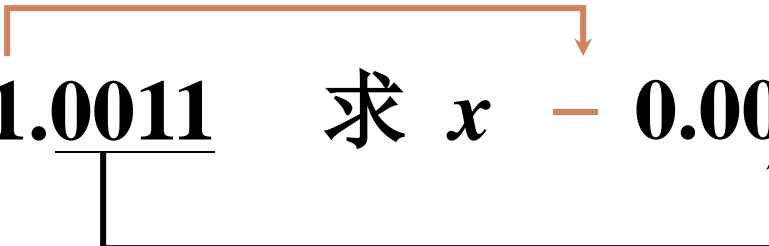




## (2) 举例

6.1

例 6.1 已知  $[x]_{\text{原}} = 1.0011$  求  $x - 0.0011$



解：由定义得

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

例 6.2 已知  $[x]_{\text{原}} = 1,1100$  求  $x - 1100$



解：由定义得

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

## (2) 举例

6.1

例 6.1 已知  $[x]_{\text{原}} = 1.0011$  求  $x - 0.0011$



解：由定义得

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

例 6.2 已知  $[x]_{\text{原}} = 1,1100$  求  $x - 1100$



解：由定义得

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

# 原码的特点：简单、直观

## 6.1

但是用原码作加法时，会出现如下问题：

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

能否 只作加法？

找到一个与负数等价的正数 来代替这个负数  
就可使 减  $\longrightarrow$  加

# 3. 补码表示法

6.1

## (1) 补的概念

- 时钟

逆时针

$$\begin{array}{r} 6 \\ - 3 \\ \hline 3 \end{array}$$

顺时针

$$\begin{array}{r} 6 \\ + 9 \\ \hline 15 \end{array}$$

可见  $-3$  可用  $+9$  代替 减法  $\rightarrow$  加法

$$\begin{array}{r} 6 \\ + 9 \\ \hline 15 \\ - 12 \\ \hline 3 \end{array}$$

称  $+9$  是  $-3$  以  $12$  为模的补数

记作  $-3 \equiv +9 \pmod{12}$

同理  $-4 \equiv +8 \pmod{12}$

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

时钟以  
12为模

# 结论

- 一个负数加上 “**模**” 即得该负数的补数
- 两个互为补数的数 它们绝对值之和即为 **模** 数

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

$$\begin{array}{r}
 \text{11} \quad 1011 \\
 - 1011 \\
 \hline
 0000
 \end{array}
 \qquad
 \begin{array}{r}
 1011 \\
 + 0101 \\
 \hline
 10000
 \end{array}$$

可见  $-1011$  可用  $+0101$  代替

记作  $-1011 \equiv +0101 \pmod{2^4}$

同理  $-011 \equiv +101 \pmod{2^3}$

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

自然去掉

# 6.1


$$-1011 \equiv +0101 \pmod{2^4}$$
$$\begin{array}{r} + 10000 \\ \hline \end{array} \quad \begin{array}{r} + 10000 \\ \hline \end{array}$$
$$+ \mathbf{0101} \equiv + \mathbf{10101}$$

$$\therefore +0101 \equiv +0101 \pmod{2^4}$$

丢掉

可见  $+0101 \rightarrow +0101$

01 ? + 0101  
└─→ - 1011


**0,0101**  $\rightarrow$  **+** **0101**

? **1**,0101  $\rightarrow$   $\underset{\uparrow}{-}$  1011

$$2^{4+1} - 1011 = 100000$$

$$\begin{array}{r} -1011 \\ \hline 1,0101 \end{array}$$

## 用 逗号 将符号位和数值位隔开

 $(\text{mod } 2^{4+1})$

### (3) 补码定义

6.1

整数

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

$x$  为真值

$n$  为整数的位数

如

$$x = +1010$$

$$x = -1011000$$

$$[x]_{\text{补}} = 0,1010$$

用 逗号 将符号位  
和数值位隔开

$$\begin{aligned} [x]_{\text{补}} &= 2^{7+1} + (-1011000) \\ &= 100000000 \\ &\quad - 1011000 \\ \hline &1,0101000 \end{aligned}$$



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

$x$  为真值

如

$$x = +0.1110$$

$$x = -0.1100000$$

$$[x]_{\text{补}} = 0.1110$$

$$[x]_{\text{补}} = 2 + (-0.1100000)$$

$$= 10.0000000$$

$$- 0.1100000$$

---


$$1.0100000$$

用 小数点 将符号位

和数值位隔开

# 补码说明

- 补码最高一位是符号位，符号 0 正 1 负
- 补码表示为： $2 \times \text{符号位} + \text{数的真值}$
- 零的补码只有一个，故补码能表示  $-1$
- 补码能很好地用于加减运算，运算时，符号位与数值位一样参加运算。

# 求特殊数的补码

假定机器数有n位

$$\textcircled{1} [-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = 10\dots0 \text{ (n-1个0)} \quad (\text{mod } 2^n)$$

$$\textcircled{2} [-1]_{\text{补}} = 2^n - 0\dots01 = 11\dots1 \text{ (n个1)} \quad (\text{mod } 2^n) \quad \text{整数补码}$$

$$\textcircled{3} [-1.0]_{\text{补}} = 2 - 1.0 = 1.00\dots0 \text{ (n-1个0)} \quad (\text{mod } 2) \quad \text{小数补码}$$

$$\textcircled{4} [+0]_{\text{补}} = [-0]_{\text{补}} = 00\dots0 \text{ (n个0)}$$

## 特殊数的补码（续）

- 当补码的位数为 $n$ 位时，其模为 $2^n$ ，所以

$$[-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = 10\dots0 \text{ (} n-1 \text{个} 0 \text{)} \pmod{2^n}$$

- 当补码的位数为 $n+1$ 位时，其模为 $2^{n+1}$ ，所以

$$[-2^{n-1}]_{\text{补}} = 2^{n+1} - 2^{n-1} = 2^n + 2^{n-1} = 1\ 10\dots0 \text{ (} n-1 \text{个} 0 \text{)} \pmod{2^{n+1}}$$

这说明同一个真值在不同位数的补码表示中，对应的机器数不同，因此给定编码表示时，一定要明确编码的位数，在机器内部编码的位数就是机器中运算部件的位数。

## (4) 求补码的快捷方式

6.1

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，补码 可用 原码除符号位外  
每位取反，末位加 1 求得

## (5) 举例

## 6.1

例 6.5 已知  $[x]_{\text{补}} = 0.0001$

求  $x$

解：由定义得  $x = +0.0001$

例 6.6 已知  $[x]_{\text{补}} = 1.0001$   $[x]_{\text{补}} \xrightarrow{?} [x]_{\text{原}}$

求  $x$

$$[x]_{\text{原}} = 1.1111$$

解：由定义得

$$\therefore x = -0.1111$$

$$x = [x]_{\text{补}} - 2$$

$$= 1.0001 - 10.0000$$

$$= -0.1111$$

## 例 6.7 已知 $[x]_{\text{补}} = 1,1110$

6.1

求  $x$

解：由定义得

$$x = [x]_{\text{补}} - 2^{4+1}$$

$$= 1,1110 - 100000$$

$$= -0010$$

$$[x]_{\text{补}} \xrightarrow{?} [x]_{\text{原}}$$

$$[x]_{\text{原}} = 1,0010$$

$$\therefore x = -0010$$

当真值为 负 时，原码 可用 补码除符号位外  
每位取反，末位加 1 求得



# 练习 求下列真值的补码

6.1

真值	$[x]_{\text{补}}$	$[x]_{\text{原}}$
$x = +70 = 1000110$	0, 1000110	0, 1000110
$x = -70 = -1000110$	1, 0111010	1, 1000110
$x = 0.1110$	0.1110	0.1110
$x = -0.1110$	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 
$$[x]_{\text{补}} = \begin{cases} x & 1 > x \geq 0 \\ 2 + x & 0 > x \geq -1 \pmod{2} \end{cases}$$

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

## 4. 反码表示法

### 6.1

#### (1) 定义

整数

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

$x$  为真值

$n$  为整数的位数

如

$$x = +1101$$

$$x = -1101$$

$$[x]_{\text{反}} = 0,1101$$

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

$$= 11111 - 1101$$

$$= 1,0010$$

用 逗号 将符号位

和数值位隔开

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

$x$  为真值

如

$$x = +0.1101$$

$$x = -0.1010$$

$$[x]_{\text{反}} = 0.1101$$

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

$$= 1.1111 - 0.1010$$

$$= 1.0101$$

用 小数点 将符号位  
和数值位隔开

## (2) 举例

## 6.1

例6.8 已知  $[x]_{\text{反}} = 0,1110$  求  $x$

解： 由定义得  $x = + 1110$

例6.9 已知  $[x]_{\text{反}} = 1,1110$  求  $x$

解： 由定义得 
$$\begin{aligned} x &= [x]_{\text{反}} - (2^{4+1} - 1) \\ &= 1,1110 - 11111 \\ &= - 0001 \end{aligned}$$

例 6.10 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  $[+0]_{\text{反}} = 0,0000$   $[-0]_{\text{反}} = 1,1111$

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

- 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，原码 = 补码 = 反码
- 对于负数，符号位为 1，其数值部分  
原码除符号位外每位取反末位加 1 → 补码  
原码除符号位外每位取反 → 反码

**例6.11** 设机器数字长为8位（其中一位为符号位）  
 对于整数，当其分别代表无符号数、原码、补码和反码时，对应的真值范围各为多少？

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	$\pm 0$	+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

## 例6.12 已知 $[y]_{\text{补}}$ 求 $[-y]_{\text{补}}$

6.1

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

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

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

即得  $[-y]_{\text{补}}$

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

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

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

即得  $[-y]_{\text{补}}$

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





## 5. 移码表示法

6.1

补码表示很难直接判断其真值大小

如	十进制	二进制	补码	
	$x = +21$	$+10101$	$0,10101$	 错大
	$x = -21$	$-10101$	$1,01011$	
	$x = +31$	$+11111$	$0,11111$	 错大
	$x = -31$	$-11111$	$1,00001$	

$x + 2^5$

$+10101 + 100000 = 110101$		大 正确
$-10101 + 100000 = 001011$		
$+11111 + 100000 = 111111$		大 正确
$-11111 + 100000 = 000001$		

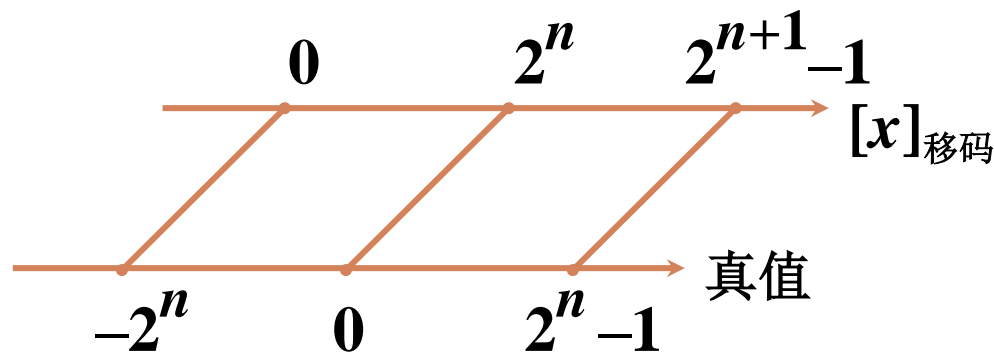
# (1) 移码定义

6.1

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

$x$  为真值,  $n$  为 整数的位数

移码在数轴上的表示



如  $x = 10100$

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

$$x = -10100$$

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

用 逗号 将符号位  
和数值位隔开

## (2) 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  $x = -1100100$

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

$$[x]_{\text{补}} = \mathbf{1},0011100$$

补码与移码只差一个符号位

# (3) 真值、补码和移码的对照表

6.1

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 0	1 0 0 0 0 0	0 0 0 0 0 0	0
- 1 1 1 1 1	1 0 0 0 0 1	0 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

## (4) 移码的特点

6.1

➤ 当  $x = 0$  时  $[+0]_{\text{移}} = 2^5 + 0 = 1,00000$   
 $[-0]_{\text{移}} = 2^5 - 0 = 1,00000$

$\therefore [+0]_{\text{移}} = [-0]_{\text{移}}$

➤ 当  $n = 5$  时 最小的真值为  $-2^5 = -100000$   
 $[-100000]_{\text{移}} = 2^5 - 100000 = 000000$

可见，最小真值的移码为全 0

用移码表示浮点数的阶码

能方便地判断浮点数的阶码大小

例：无符号数在C语言中对应**unsigned short**、**unsigned int(unsigned)**、**unsigned long**，带符号整数表示为**short**、**int**、**long**类型，求以下程序段在一个**32**位机器上运行时输出的结果，并说明为什么。

```
1 int x=-1;
2 unsigned u=2 147 483 648;
3 printf ("x=%u=%d\n", x , x);
4 printf ("u=%u=%d\n", u , u);
```

说明：

- 其中**printf**为输出函数，指示符**%u**、**%d**分别表示以无符号整数和有符号整数形式输出**十进制数**的值，

- $2\ 147\ 483\ 648=2^{31}$

解：因为现代计算机中带符号整数都是用补码表示的，**-1**的补码整数表示为“**11...1**”（共**32**位），当作**32**位无符号数时，因此，十进制表示为  $2^{32}-1=4\ 294\ 967\ 296-1=4\ 294\ 967\ 295$

$2^{31}$ 的无符号整数在计算机中表示为“**10...0**”，当作为有符号整数输出时，其值为最小负数 $-2^{32-1}=-2\ 147\ 483\ 648$

输出结果：

**x= 4 294 967 295=-1**

**u=2 147 483 648=- 2 147 483 648**

例：以下为C语言程序，用来计算一个数组**a**中每个元素的和，当参数**len**为**0**时，返回值应该为**0**，却发生了存储器访问异常。请问这是什么原因造成的？说明如何修改。

```
1 float sum_elements ( float a[ ], unsigned len)
2 {
3     int    i;
4     float result=0;
5     for ( i=0; i<=len-1; i++ )
6         result+=a[ i ] ;
7     return result;
8 }
```

解：存储器访问异常是由于对数组**a**访问时产生了越界错误造成的。循环变量**i**是**int**型，而**len**是**unsigned**型，当**len**为**0**时，执行**len-1**的结果为**32**个**1**，是最大可表示的**32**位无符号数，任何无符号数都比它小，使循环体不断被执行，导致数组访问越界，因而发生存储器访问异常，应当将**len**声明为**int**型。

## 6.2 数的定点表示和浮点表示

小数点按约定方式标出

### 一、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

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

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



## 二、浮点表示

## 6.2

$N = S \times r^j$  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  $r = 2$   $N = 11.0101$

尾数为纯  
小数

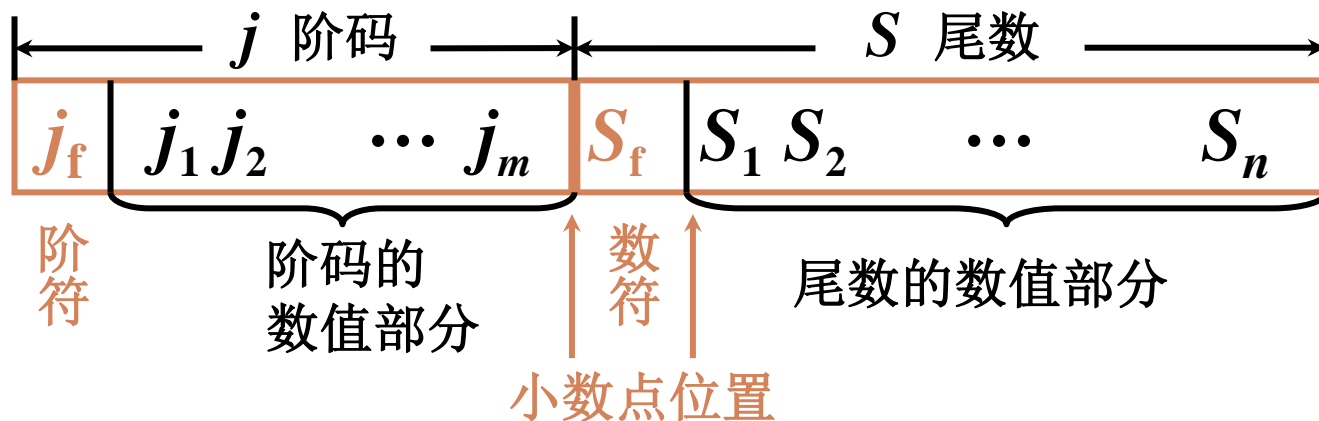
二进制表示  
规格化数

$$\begin{aligned} &= 0.110101 \times 2^{10} \\ &= 1.10101 \times 2^1 \\ &= 1101.01 \times 2^{-10} \\ &= 0.00110101 \times 2^{100} \end{aligned}$$

计算机中  $S$  小数、可正可负  
 $j$  整数、可正可负

# 1. 浮点数的表示形式

## 6.2



$S_f$

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  $m$

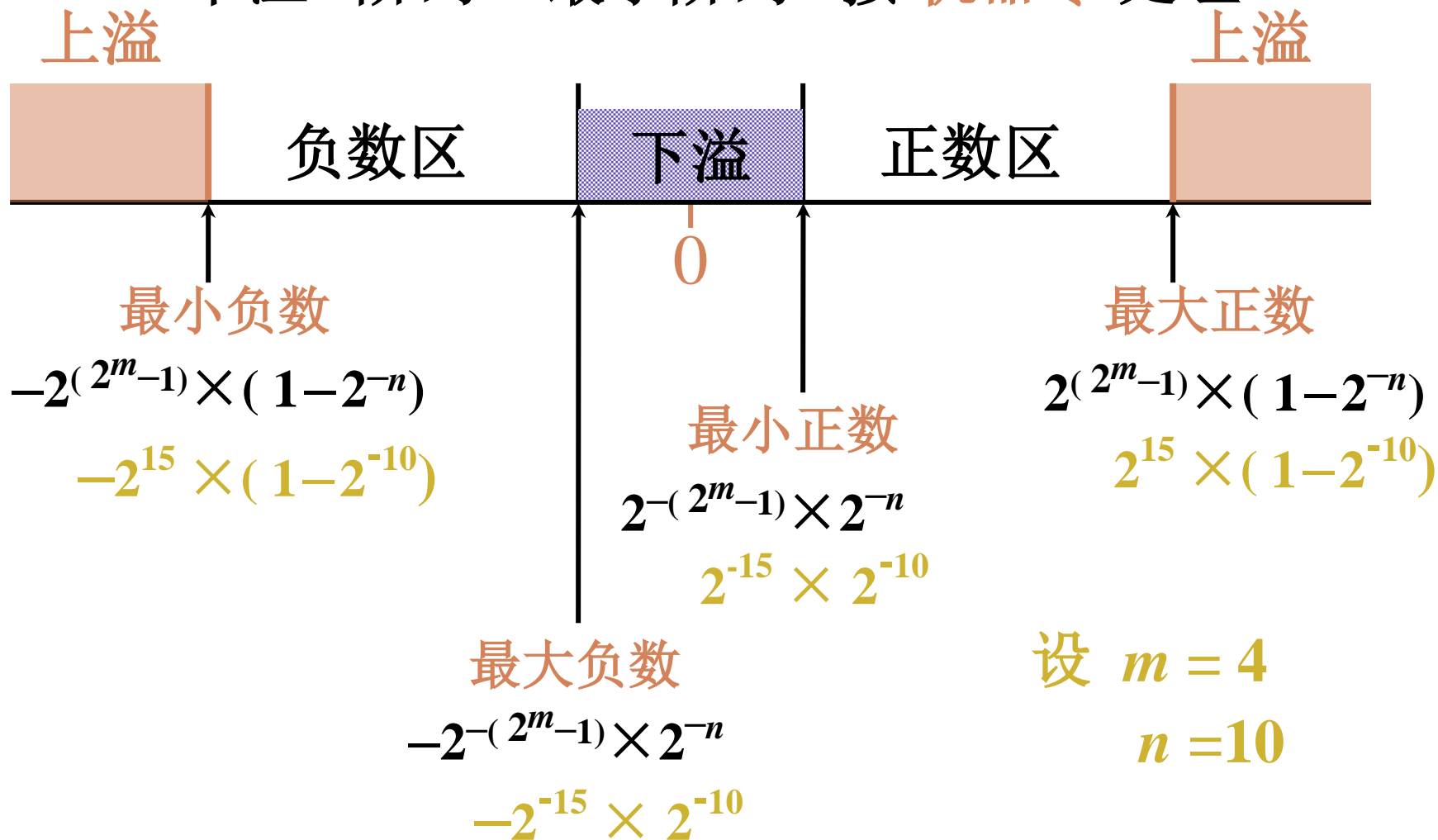
共同表示小数点的实际位置

## 2. 浮点数的表示范围

6.2

上溢 阶码 > 最大阶码

下溢 阶码 < 最小阶码 按 机器零 处理



设机器数字长为 24 位，欲表示  $\pm 3$  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？

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

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

$$2^{15} \times 0.\underbrace{\times \times \times \dots \times \times}_{18\text{位}}$$

$2^4 = 16 \quad m = 4, 5, 6 \dots$

满足 最大精度 可取  $m = 4, n = 18$

### 3. 浮点数的规格化形式

## 6.2

$$|s| \geq 1/r$$

$r = 2$  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

基数不同，浮点数的规格化形式不同

0 . × × × × × × × ×

$2^{-1} 2^{-2} 2^{-3} 2^{-4} \dots$

$r = 4$                       ✓ 0.0101110                      ✓

✓                      ✓ 0.1000101                      ✓

$r = 8$                                       0.0010101                      ✓

## 4. 浮点数的规格化

$r = 2$	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

例如：设  $m = 4$ ,  $n = 10$

6.2

## 尾数规格化后的浮点数表示范围

最大正数  $2^{+1111} \times 0.\underbrace{1111111111}_{2^{-15} \times 2^{-10} = 2^{-25}}, = 2^{15} \times (1 - 2^{-10})$

最小正数  $2^{-1111} \times 0.1\underbrace{0000000000}_{-2^{-15} \times 2^{-10} = -2^{-25}} = 2^{-15} \times 2^{-1} = 2^{-16}$


最大负数  $2^{-1111} \times (-0.1\underbrace{0000000000}_{9 \text{ 个 } 0}) = -2^{-15} \times 2^{-1} = -2^{-16}$

最小负数  $2^{+1111} \times (-0.1\underbrace{1111111111}_{10 \text{ 个 } 1}) = -2^{15} \times (1 - 2^{-10})$

### 三、举例

## 6.2

例 6.13 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 6.14** 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

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

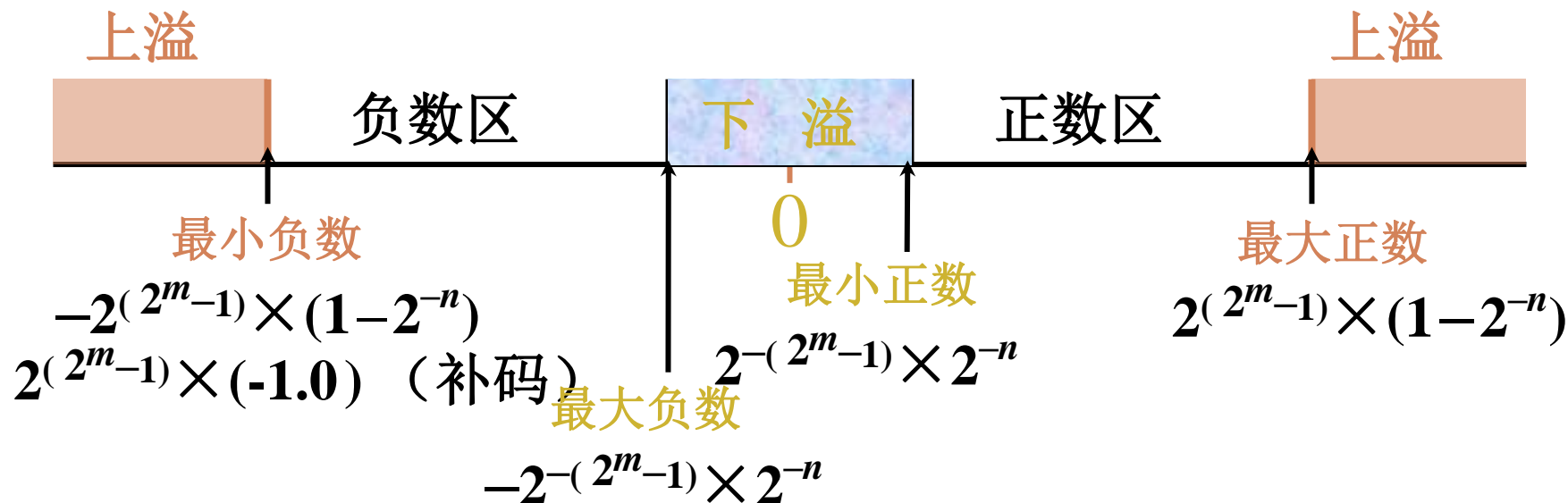
$[x]_{\text{补}} = 0, 0110; 1. 0001100000$

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

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

# 例6.15 写出对应下图所示的浮点数的补码

形式。设  $n = 10$ ,  $m = 4$ , 阶符、数符各取 1 位。



解:	真值	补码
最大正数	$2^{15} \times (1-2^{-10})$	0,1111; 0.1111111111
最小正数	$2^{-15} \times 2^{-10}$	1,0001; 0.0000000001
最大负数	$-2^{-15} \times 2^{-10}$	1,0001; 1.1111111111
最小负数	$-2^{15} \times (1-2^{-10})$	0,1111; 1.0000000001
负数边界	$2^{15} \times (-1.0)$	0,1111; 1.0000000000

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

如  $m = 4$        $n = 10$

当阶码和尾数都用补码表示时，机器零为

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

(阶码 = -16)  $1, 0000; \quad \times.\times\times \dots \times$

当阶码用移码，尾数用补码表示时，机器零为

$0, 0000; \quad 0.00 \dots 0$

有利于机器中“判 0”电路的实现

# “Father” of the IEEE 754 standard

直到80年代初，各个机器内部的浮点数表示格式还没有统一  
因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE754的制定

现在所有计算机都采用IEEE754来表示浮点数

This standard was primarily the work of one person, UC Berkeley math professor William Kahan.



[www.cs.berkeley.edu/~wkahan/ieee754status/754story.html](http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html)



**Prof. William Kahan**

# IEEE 754 标准

现代计算机中，浮点数一般采用IEEE制定的国际标准IEEE754标准

符号	阶码（含阶符）	尾数
----	---------	----

符号位： 0—正数， 1—负数。

尾数： 1. xxx.....x

在实际表示中整数位的1省略，称隐藏位（临时浮点数不采用隐藏位方案）

阶码： 移码

由于尾数形式的变化，阶码部分也与一般移码不同，对短实数而言，  
$$[X]_{\text{移}} = 2^7 - 1 + X = 127 + X$$

# IEEE 754 Floating Point Standard

**Single Precision** : ( **Double Precision is similar** )

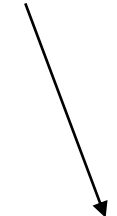
1 bit <b>S</b>	8 bits <b>Exponent</b>	23 bits <b>Significand</b>
----------------	------------------------	----------------------------

- **Sign bit**: 1 表示negative ; 0表示 positive
- **Exponent** (阶码 / 指数) : **全0和全1用来表示特殊值!**
  - SP规格化数阶码范围为0000 0001 (真值: -126) ~ 1111 1110 (真值: 127)
  - bias为127 (single), 1023 (double)
- **Significand** (尾数) : **为什么用127? 若用128, 则阶码范围为多少?**
  - 规格化尾数最高位总是1, 所以隐含表示, 省1位
  - 1 + 23 bits ( single ) , 1 + 52 bits ( double )

**SP**:  $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

**DP**:  $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-1023)}$

0000 0001 (-127) ~  
1111 1110 (126)



主要用在浮点运算单元内部，用于  
减少舍入误差

	单精度数	双精度数	临时实数
字长	32	64	80 128
符号	1	1	1
阶码	8	11	15
偏移量	127	1023	16383
尾数	23+(1)	52+(1)	64 112
最大正数	$2^{127} (2 - 2^{-23})$ $= 2^{128} (1 - 2^{-24})$ $\approx 1.7 * 10^{38}$	$2^{1023} (2 - 2^{-52})$ $= 2^{1024} (1 - 2^{-53})$ $\approx 9 * 10^{307}$	
最小规格化正数	$2^{-126} \approx 1.2 * 10^{-38}$	$2^{-1022} \approx 2.2 * 10^{-308}$	
精度	$2^{-23} \approx 10^{-7}$		

## Ex: Converting Binary FP to Decimal

**BEE00000H** is the hex. Rep. Of an IEEE 754 SP FP number

1	0111 1101	110 0000 0000 0000 0000 0000
---	-----------	------------------------------

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- **Sign:** 1 => negative
- **Exponent:**
  - $0111\ 1101_{\text{two}} = 125_{\text{ten}}$
  - Bias adjustment:  $125 - 127 = -2$
- **Significand:**
$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$
- **Represents:**  $-1.75_{\text{ten}} \times 2^{-2} = -0.4375 \quad (= -4.375 \times 10^{-1})$



# Ex: Converting Decimal to FP

$$-1.275 \times 10^1$$

1. Denormalize: -12.75

2. Convert integer part:

$$12 = 8 + 4 = 1100_2$$

3. Convert fractional part:

$$.75 = .5 + .25 = .11_2$$

4. Put parts together and normalize:

$$1100.11 = 1.10011 \times 2^3$$

5. Convert exponent:  $127 + 3 = 128 + 2 = 1000\ 0010_2$

11000 0010	100 1100 0000 0000 0000 0000
------------	------------------------------

The Hex rep. is C14C0000H

# 0 的表示

How to represent 0?

**exponent**: all zeros

**significand**: all zeros

**What about sign?** Both cases valid.

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

# 如何表示 $+\infty/-\infty$

In FP, 除数为0的结果是  $\pm\infty$ , 不是溢出异常.  $\infty$  : 无穷(infinity)

为什么要这样处理?

- 可以利用 $+\infty/-\infty$ 作比较。 例如:  $X/0 > Y$ 可作为有效比较

How to represent  $+\infty/-\infty$ ?

- **Exponent** : all ones (11111111B = 255)
- **Significand**: all zeros

$+\infty$  : 0 11111111 000000000000000000000000

$-\infty$  : 1 11111111 000000000000000000000000

Operations

$$5 / 0 = +\infty, \quad -5 / 0 = -\infty$$

$$5 + (+\infty) = +\infty, \quad (+\infty) + (+\infty) = +\infty$$

$$5 - (+\infty) = -\infty, \quad (-\infty) - (+\infty) = -\infty \quad \text{etc}$$

# 用**BCD**码表示十进制数

- **BCD码**:用**4**位二进制代码的不同组合来表示一个十进制数码的编码方法。
- **编码思想**: 每个十进数位至少有**4**位二进制表示。而**4**位二进制位可组合成**16**种状态, 去掉**10**种状态后还有**6**种冗余状态。

# 用BCD码表示十进制数

## 编码方案

十进制数 8421

### 1. 十进制有权码

- 每个十进制数位的4个二进制位（称为基2码）都有一个确定的权。8421码是最常用的十进制有权码。也称自然BCD（NBCD）码。

### 2. 十进制无权码

- 每个十进制数位的4个基2码没有确定的权。在无权码方案中，用的较多的是余3码和格雷码。
  - 余3码：8421码+0011
  - 格雷码：任何两个相邻代码只有一个二进制位的状态不同

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

# 用BCD码表示十进制数

## 编码方案

1. 十进制有权码
2. 十进制无权码
3. 其他编码方案（5中取2码、独热码等）

## 符号位的表示：

■ “+”：1100；“-”：1101

■ 例：+236=(1100 0010 0011 0110)<sub>8421</sub>（占2个字节）

- 2369=(1101 0000 0010 0011 0110 1001)<sub>8421</sub>  
(占3个字节)



补0以使数占满一个字节

# BCD码

十进制数	8421	5421	2421	余3码	余3循环码
0	0000	0000	0000	0011	0010
1	0001	0001	0001	0100	0110
2	0010	0010	0010	0101	0111
3	0011	0011	0011	0110	0101
4	0100	0100	0100	0111	0100
5	0101	1000	1011	1000	1100
6	0110	1001	1100	1001	1101
7	0111	1010	1101	1010	1111
8	1000	1011	1110	1011	1110
9	1001	1100	1111	1100	1010

# 逻辑数据的编码表示

- 表示

- 用一位表示 真：1 / 假：0
- N位二进制数可表示N个逻辑数据，或一个位串

- 运算

- 按位进行
- 如:按位与 / 按位或 / 逻辑左移 / 逻辑右移 等

- 识别

- 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器靠指令来识别。



# 西文字符的编码表示

## ● 特点

- 是一种拼音文字，用有限几个字母可拼写出所有单词
- 只对有限个字母和数学符号、标点符号等辅助字符编码
- 所有字符总数不超过256个，使用7或8个二进位可表示

## ● 表示（常用编码为7位ASCII码）

要求必须熟悉数字、字母和空格(SP)的表示

- 十进制数字：0/1/2.../9
- 英文字母：A/B/.../Z/a/b/.../z
- 专用符号：+/-/%/\*/&/.....
- 控制字符（不可打印或显示）

## ● 操作

- 字符串操作，如：传送/比较 等

# 字符串的表示与存储

字符串是指连续的一串字符，它们占据主存中连续的多个字节，每个字节存放一个字符，对一个主存字的多个字节，有按从低位到高位字节次序存放的，也有按从高位到低位字节次序存放的。表示字符串数据要给出串存放的主存起始地址和串的长度。如：

IF A>B THEN READ(C) 就可以有如下不同存放方式：

I	F		A
>	B		T
H	E	N	
R	E	A	D
(	C	)	

A		F	I
T		B	>
	N	E	H
D	A	E	R
	)	C	(

假定每个字  
由 4 个字节组成

# 字符串的表示与存储

按从低位到高位字节次序存放

				31	24	23	16	15	8	7	0
I	F		A	49	46	20	41				
>	B		T	3e	42	20	54				
H	E	N		48	45	4e	20				
R	E	A	D	52	45	41	44				
(	C	)		28	43	29	20				

16进制数据

# 汉字及国际字符的编码表示

- 特点

- 汉字是表意文字，一个字就是一个方块图形。
- 汉字数量巨大，总数超过6万字，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。

- 编码形式

- 有以下几种汉字代码：
  - 输入码：对汉字用相应按键进行编码表示，用于输入
  - 内码：用于在系统中进行存储、查找、传送等处理
  - 字模点阵或轮廓描述：描述汉字字模点阵或轮廓，用于显示/打印

# 汉字的输入码

向计算机输入汉字的方式：

- ① 手写汉字联机识别输入，或者是印刷汉字扫描输入后自动识别，这两种方法现均已达到实用水平。
- ② 用语音输入汉字，虽然简单易操作，但离实用阶段还有距离。
- ③ 利用英文键盘输入汉字：每个汉字用一个或几个键表示，这种对每个汉字用相应按键进行的编码称为汉字“输入码”，又称外码。输入码的码元为按键。是最简便、最广泛的汉字输入方法。

常用的方法有：搜狗拼音、五笔字型、智能ABC、微软拼音等

使用汉字输入码的原因：

- ① 键盘面向西文设计，一个或两个西文字符对应一个按键，非常方便。
- ② 汉字是大字符集，专门的汉字输入键盘由于键多、查找不便、成本高等原因而几乎无法采用。

# 字符集与汉字的内码

问题：西文字符常用的内码是什么？其内码就是ASCII码。

对于汉字内码的选择，必须考虑以下几个因素：

- ① 不能有二义性，即不能和ASCII码有相同的编码。
- ② 尽量与汉字在字库中的位置有关，便于汉字查找和处理。
- ③ 编码应尽量短。

## 国标码（国标交换码）

1981年我国颁布了《信息交换用汉字编码字符集·基本集》(GB2312—80)。该标准选出6763个常用汉字，为每个汉字规定了标准代码，以供汉字信息在不同计算机系统间交换使用

可在汉字国标码的基础上产生汉字机内码

# GB2312-80字符集

- 由三部分组成：

- ① 字母、数字和各种符号，包括英文、俄文、日文平假名与片假名、罗马字母、汉语拼音等共687个
- ② 一级常用汉字，共3755个，按汉语拼音排列
- ③ 二级常用汉字，共3008个，不太常用，按偏旁部首排列

- 汉字的区位码

- 码表由94行、94列组成，行号为区号，列号为位号，各占7位
- 指出汉字在码表中的位置，共14位，区号在左、位号在右

- 汉字的国标码

- 每个汉字的区号和位号各自加上32 (20H)，得到其“国标码”
- 国标码中区号和位号各占7位。在计算机内部，为方便处理与存储，前面添一个0，构成一个字节

# 汉字内码

- 至少需2个字节才能表示一个汉字内码。
  - 由汉字的总数决定！
- 可在GB2312国标码的基础上产生汉字内码
  - 为与ASCII码区别，将国标码的两个字节的第一位置“1”后得到一种汉字内码

例如，汉字“大”在码表中位于第20行、第83列。因此区位码为0010100 1010011，国标码为00110100 01110011，即3473H。前面的34H和字符“4”的ASCII码相同，后面的73H和字符“s”的ASCII码相同，将每个字节的最高位各设为“1”后，就得到其内码：B4F3H (1011 0100 1111 0011B)，因而不会和ASCII码混淆。



# 国际字符集

## 国际字符集的必要性

- ◆ 不同地区使用不同字符集内码，如中文GB2312 / Big5、日文Shift-JIS / EUC-JP等。在安装中文系统的计算机中打开日文文件，会出现乱码。
- ◆ 为使所有国际字符都能互换，必须创建一种涵盖全部字符的多字符集。

## 国际多字符集

- ◆ 通过对各种地区性字符集规定使用范围来唯一定义各字符的编码。
- ◆ 国际标准ISO/IEC 10646提出了一种包括全世界现代书面语言文字所使用的所有字符的标准编码，有4个字节编码(UCS-4)和2字节编码(UCS-2)。
- ◆ 我国（包括香港、台湾地区）与日本、韩国联合制订了一个统一的汉字字符集（CJK编码），共收集了上述不同国家和地区共约2万多汉字及符号，采用2字节编码（即：UCS-2），已被批准为国家标准(GB13000)。
- ◆ Windows操作系统(中文版)已采用中西文统一编码，收集了中、日、韩三国常用的约2万汉字，称为“Unicode”，采用2字节编码，与UCS-2一致。

# UNICODE编码

Unicode是完全双字节表示的多国文字编码体系，编码空间 **0x0000-0xFFFF**。可以表示 **65536** 个字符

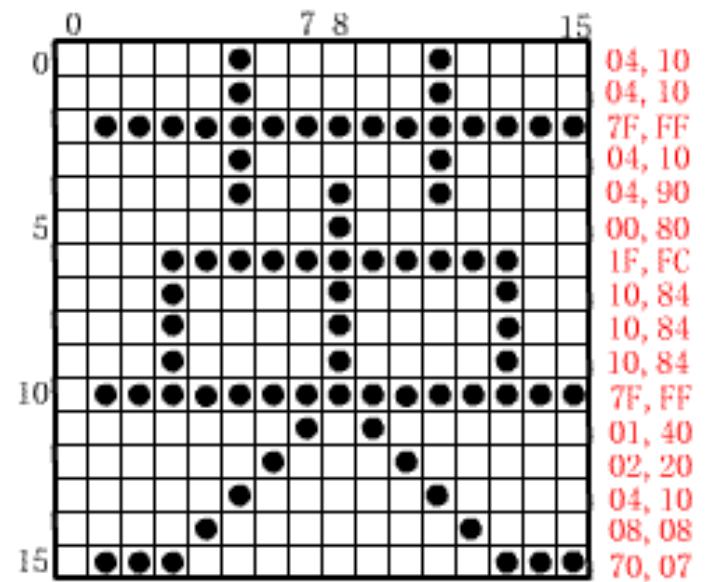
# 汉字的字模点阵码和轮廓描述

- 为便于打印、显示汉字，汉字字形必须预先存在机内
  - 字库 (font): 所有汉字形状的描述信息集合
  - 不同字体 (如宋体、仿宋、楷体、黑体等) 对应不同字库
  - 从字库中找到字形描述信息，然后送设备输出
- 字形主要有两种描述方法：
  - 字模点阵描述 (图像方式)
  - 轮廓描述 (图形方式)
    - 直线向量轮廓
    - 曲线轮廓 (True Type字形)

# 字模点阵描述

- 字模编码

- 字模编码是以点阵方式用来描述汉字字形的代码，它是汉字的输出形式。



# 数据的基本宽度

- 比特 (bit) 是计算机中处理、存储、传输信息的最小单位
- 二进制信息的计量单位是 “字节” (Byte), 也称 “位组”
  - 现代计算机中, 存储器按字节编址
  - 字节是最小可寻址单位 (*addressable unit* )
- 除比特和字节外, 还经常使用 “字” (word)作为单位
- “字” 和 “字长” 的概念不同

# 数据的基本宽度

- “字” 和 “字长” 的概念不同

- “字长” 指数据通路的宽度。

(数据通路指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。因此， “字长” 等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。 )

- “字” 表示被处理信息的单位，用来度量数据类型的宽度。
- 字和字长的宽度可以一样，也可不同。

例如，x86体系结构定义 “字” 的宽度为16位，但从386开始字长就是32位了。

## 6.3 定点运算

### 一、移位运算

#### 1. 移位的意义

15. 米 = 1500. 厘米

小数点右移 2 位

机器用语      15 相对于小数点 左移 2 位

（ 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

## 2. 算术移位规则

6.3

符号位不变

	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6.16

## 6.3

设机器数字长为 8 位（含一位符号位），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
$\leftarrow 1$	0,0110100	+52
$\leftarrow 2$	0,1101000	+104
$\rightarrow 1$	0,0001101	+13
$\rightarrow 2$	0,0000110	+6

## 例6.17

## 6.3

设机器数字长为 8 位（含一位符号位），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
	←1	1,0110100	- 52
	←2	1,1101000	- 104
	→1	1,0001101	- 13
	→2	1,0000110	- 6

补码

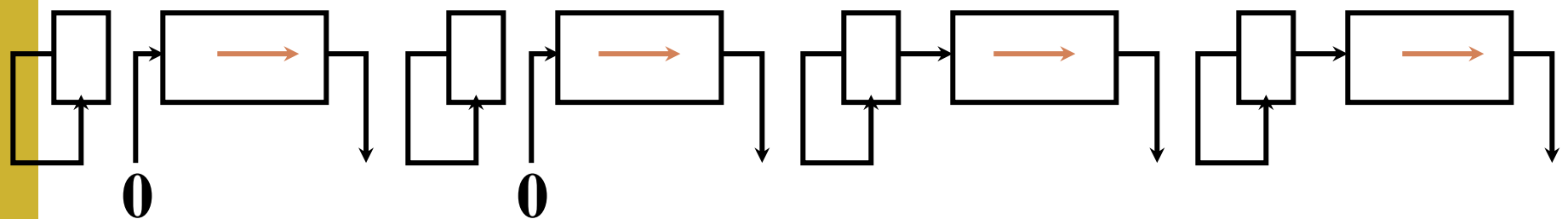
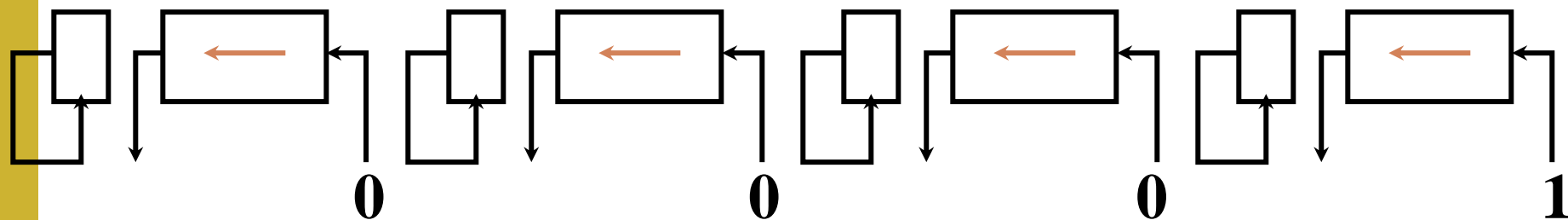
移位操作	机 器 数	对应的真值
移位前	1,1100110	- 26
←1	1,1001100	- 52
←2	1,0011000	- 104
→1	1,110011	- 13
→2	1,111001	- 7

反码

移位操作	机 器 数	对应的真值
移位前	1,1100101	- 26
←1	1,1001011	- 52
←2	1,0010111	- 104
→1	1,110010	- 13
→2	1,111001	- 6

# 3. 算术移位的硬件实现

6.3



(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

# 4. 算术移位和逻辑移位的区别

6.3

算术移位      有符号数的移位

逻辑移位      无符号数的移位

逻辑左移      低位添 0，高位移丢

逻辑右移      高位添 0，低位移丢

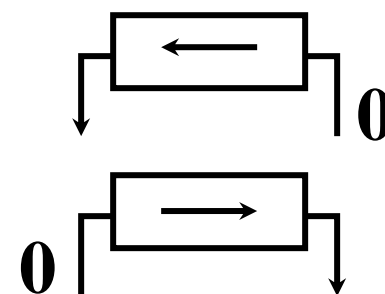
例如      11010011

逻辑左移      10100110

算术左移      10100110

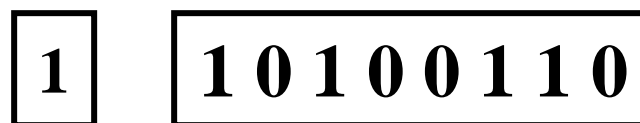
逻辑右移      01011001

算术右移      11011001 (补码)



10110010

高位 1 移丢



## 二、加减法运算

### 6.3

### 1. 补码加减运算公式

#### (1) 加法

**整数**  $[A]_{\text{补}} + [B]_{\text{补}} = [A+B]_{\text{补}} \pmod{2^{n+1}}$

**小数**  $[A]_{\text{补}} + [B]_{\text{补}} = [A+B]_{\text{补}} \pmod{2}$

#### (2) 减法

$$A-B = A+(-B)$$

**整数**  $[A-B]_{\text{补}} = [A+(-B)]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}} \pmod{2^{n+1}}$

**小数**  $[A-B]_{\text{补}} = [A+(-B)]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}} \pmod{2}$

连同符号位一起相加，符号位产生的进位自然丢掉

## 证明

(1) 设  $A > 0, B < 0$

①  $|B| > A$  (  $A+B < 0$  )

$$\begin{aligned}[A+B]_{\text{补}} &= 2 + A + B \\ &= A + 2 + B \\ &= [A]_{\text{补}} + [B]_{\text{补}}\end{aligned}$$

②  $|B| < A$  (  $A+B > 0$  )

$$\begin{aligned}[A+B]_{\text{补}} &= A + B \\ &= A + 2 + B \pmod{2} \\ &= [A]_{\text{补}} + [B]_{\text{补}}\end{aligned}$$

## 2. 举例

6.3

例 6.18 设  $A = 0.1011$ ,  $B = -0.0101$

求  $[A + B]_{\text{补}}$

验证

$$\begin{array}{l} \text{解: } [A]_{\text{补}} = 0.1011 \\ \quad + [B]_{\text{补}} = 1.1011 \\ \hline [A]_{\text{补}} + [B]_{\text{补}} = 10.0110 = [A + B]_{\text{补}} \\ \therefore A + B = 0.0110 \end{array}$$

例 6.19 设  $A = -9$ ,  $B = -5$

求  $[A+B]_{\text{补}}$

验证

$$\begin{array}{l} \text{解: } [A]_{\text{补}} = 1, 0111 \\ \quad + [B]_{\text{补}} = 1, 1011 \\ \hline [A]_{\text{补}} + [B]_{\text{补}} = 11, 0010 = [A + B]_{\text{补}} \\ \therefore A + B = -1110 \end{array}$$



## 例 6.20 设机器数字长为 8 位（含 1 位符号位） 6.3

且  $A = 15$ ,  $B = 24$ , 用补码求  $A - B$

解:  $A = 15 = 0001111$

$$B = 24 = 0011000$$

$$[A]_{\text{补}} = 0, 0001111 \quad [B]_{\text{补}} = 0, 0011000$$

$$+ [-B]_{\text{补}} = 1, 1101000$$

---

$$[A]_{\text{补}} + [-B]_{\text{补}} = 1, 1110111 = [A - B]_{\text{补}}$$

$$\therefore A - B = -1001 = -9$$

练习 1 设  $x = \frac{9}{16}$   $y = \frac{11}{16}$  用补码求  $x+y$

$$x + y = -0.1100 = -\frac{12}{16} \quad \text{错}$$

练习 2 设机器数字长为 8 位（含 1 位符号位）

且  $A = -97$ ,  $B = +41$ , 用补码求  $A - B$

$$A - B = +1110110 = +118 \quad \text{错}$$

### 3. 溢出判断

### 6.3

#### (1) 一位符号位判溢出

参加操作的 **两个数**（减法时即为被减数和“求补”以后的减数）**符号相同**，其结果的符号与原操作数的符号不同，即为溢出

#### 硬件实现

**最高有效位的进位  $\oplus$  符号位的进位 = 1**      溢出

如

$1 \oplus 0 = 1$	} 有 溢出
$0 \oplus 1 = 1$	
$0 \oplus 0 = 0$	} 无 溢出
$1 \oplus 1 = 0$	

## (2) 两位符号位判溢出

6.3

$$[x]_{\text{补}'} = \begin{cases} x & 1 > x \geq 0 \\ 4 + x & 0 > x \geq -1 \pmod{4} \end{cases}$$

$$[x]_{\text{补}'} + [y]_{\text{补}'} = [x + y]_{\text{补}'} \pmod{4}$$

$$[x - y]_{\text{补}'} = [x]_{\text{补}'} + [-y]_{\text{补}'} \pmod{4}$$

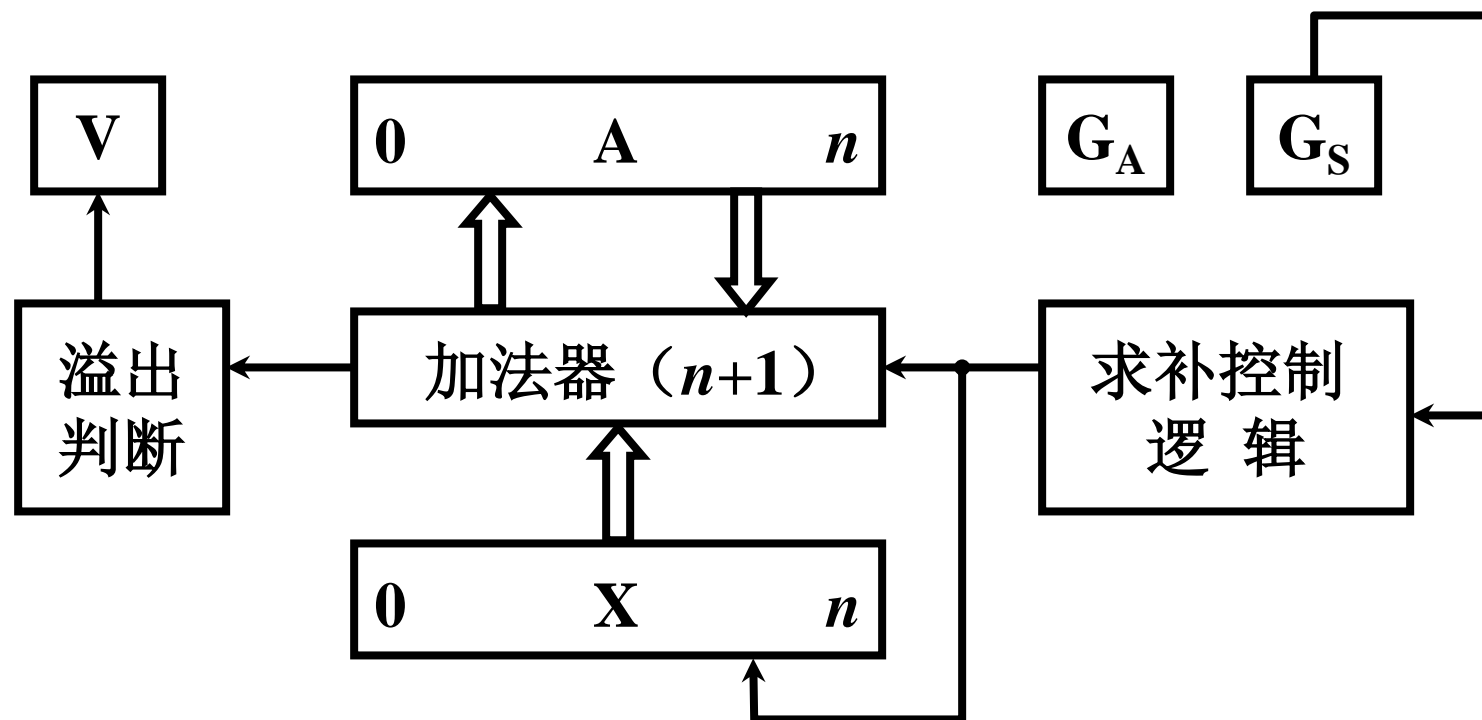
结果的双符号位	相同	未溢出	<b>00</b> , ×××××
			<b>11</b> , ×××××

结果的双符号位	不同	溢出	<b>10</b> , ×××××
			<b>01</b> , ×××××

最高符号位 代表其 真正的符号

## 4. 补码加减法的硬件配置

6.3



A、X 均  $n+1$  位

用减法标记  $G_S$  控制求补逻辑

# 三、乘法运算

## 6.3

### 1. 分析笔算乘法

$$A = -0.1101 \quad B = 0.1011$$

$$A \times B = -0.10001111 \quad \text{乘积的符号心算求得}$$

$$\begin{array}{r} 0.1101 \\ \times 0.1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ 1101 \\ \hline 0.10001111 \end{array}$$

✓ 符号位单独处理

✓ 乘数的某一位决定是否加被乘数

? 4个位积一起相加

✓ 乘积的位数扩大一倍

## 2. 笔算乘法改进

6.3

$$A \cdot B = A \cdot 0.1011$$

$$= 0.1A + 0.00A + 0.001A + 0.0001A$$

$$= 0.1A + 0.00A + 0.001(A + 0.1A)$$

$$= 0.1A + 0.01[0 \cdot A + 0.1(A + 0.1A)]$$

右移一位

$$= 0.1\{A + 0.1[0 \cdot A + 0.1(A + 0.1A)]\}$$

$$= 2^{-1}\{A + 2^{-1}[0 \cdot A + 2^{-1}(A + 2^{-1}(A + 0))]\}$$

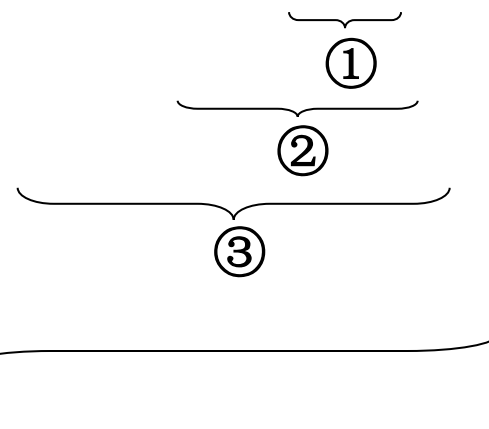
第一步 被乘数  $A + 0$

第二步  $\rightarrow 1$ , 得新的部分积

第三步 部分积  $+$  被乘数

$\vdots$

第八步  $\rightarrow 1$ , 得结果



### 3. 改进后的笔算乘法过程（竖式）

6.3

部分积	乘数	说明
0.0000 0.1101	1011 =	初态，部分积 = 0 乘数为 1，加被乘数
0.1101 0.0110 0.1101	1101 =	→ 1，形成新的部分积 乘数为 1，加被乘数
1.0011 0.1001 0.0000	1 1110 =	→ 1，形成新的部分积 乘数为 0，加 0
0.1001 0.0100 0.1101	11 1111 =	→ 1，形成新的部分积 乘数为 1，加被乘数
1.0001 0.1000	111 1111	→ 1，得结果

- 乘法运算 → 加和移位。 $n = 4$ ，加 4 次，移 4 次
- 由乘数的末位决定被乘数是否与原部分积相加，然后 → 1，形成新的部分积，同时乘数 → 1（末位移丢），空出高位存放部分积的低位。
- 被乘数只与部分积的高位相加

硬件      3个寄存器，具有移位功能  
            一个全加器



## 4. 原码乘法

## 6.3

### (1) 原码一位乘运算规则

以小数为例

$$\text{设}[x]_{\text{原}} = x_0.x_1x_2 \cdots x_n$$

$$[y]_{\text{原}} = y_0.y_1y_2 \cdots y_n$$

$$\begin{aligned}[x \cdot y]_{\text{原}} &= (x_0 \oplus y_0).(0.x_1x_2 \cdots x_n)(0.y_1y_2 \cdots y_n) \\ &= (x_0 \oplus y_0).x^*y^*\end{aligned}$$

式中  $x^* = 0.x_1x_2 \cdots x_n$  为  $x$  的绝对值

$y^* = 0.y_1y_2 \cdots y_n$  为  $y$  的绝对值

乘积的符号位单独处理  $x_0 \oplus y_0$

数值部分为绝对值相乘  $x^* \cdot y^*$

## (2) 原码一位乘递推公式

6.3

$$x^* \cdot y^* = x^*(0.y_1y_2 \dots y_n)$$

$$= x^*(y_12^{-1} + y_22^{-2} + \dots + y_n2^{-n})$$

$$= 2^{-1}(y_1x^* + 2^{-1}(y_2x^* + \dots 2^{-1}(y_nx^* + 0) \dots))$$

The diagram illustrates the recursive structure of the multiplication formula. It shows a large horizontal bracket labeled  $z_n$  at the bottom. Inside this bracket, there are three smaller horizontal brackets. The first is labeled  $z_1$ , the second is labeled  $z_0$ , and the third is labeled  $z_1$ . There are also three dots between the first and second brackets, and three dots between the second and third brackets, indicating a sequence of operations.

$$z_0 = 0$$

$$z_1 = 2^{-1}(y_nx^* + z_0)$$

$$z_2 = 2^{-1}(y_{n-1}x^* + z_1)$$

$\vdots$

$$z_n = 2^{-1}(y_1x^* + z_{n-1})$$

# 例6.21 已知 $x = -0.1110$ $y = 0.1101$ 求 $[x \cdot y]_{\text{原}}$

解：数值部分的运算

部分积	乘数	说明
0.0000	110 <u>1</u>	部分积 初态 $z_0 = 0$
0.1110		
0.1110		
0.0111	011 <u>0</u>	→1, 得 $z_1$
0.0000		
0.0111	0	
0.0011	101 <u>1</u>	→1, 得 $z_2$
0.1110		
1.0001	10	
逻辑右移 0.1000	110 <u>1</u>	→1, 得 $z_3$
0.1110		
1.0110	110	
逻辑右移 0.1011	0110	→1, 得 $z_4$

## 例6.21 结果

6.3

① 乘积的符号位  $x_0 \oplus y_0 = 1 \oplus 0 = 1$

② 数值部分按绝对值相乘

$$x^* \cdot y^* = 0.10110110$$

$$\text{则 } [x \cdot y]_{\text{原}} = 1.10110110$$

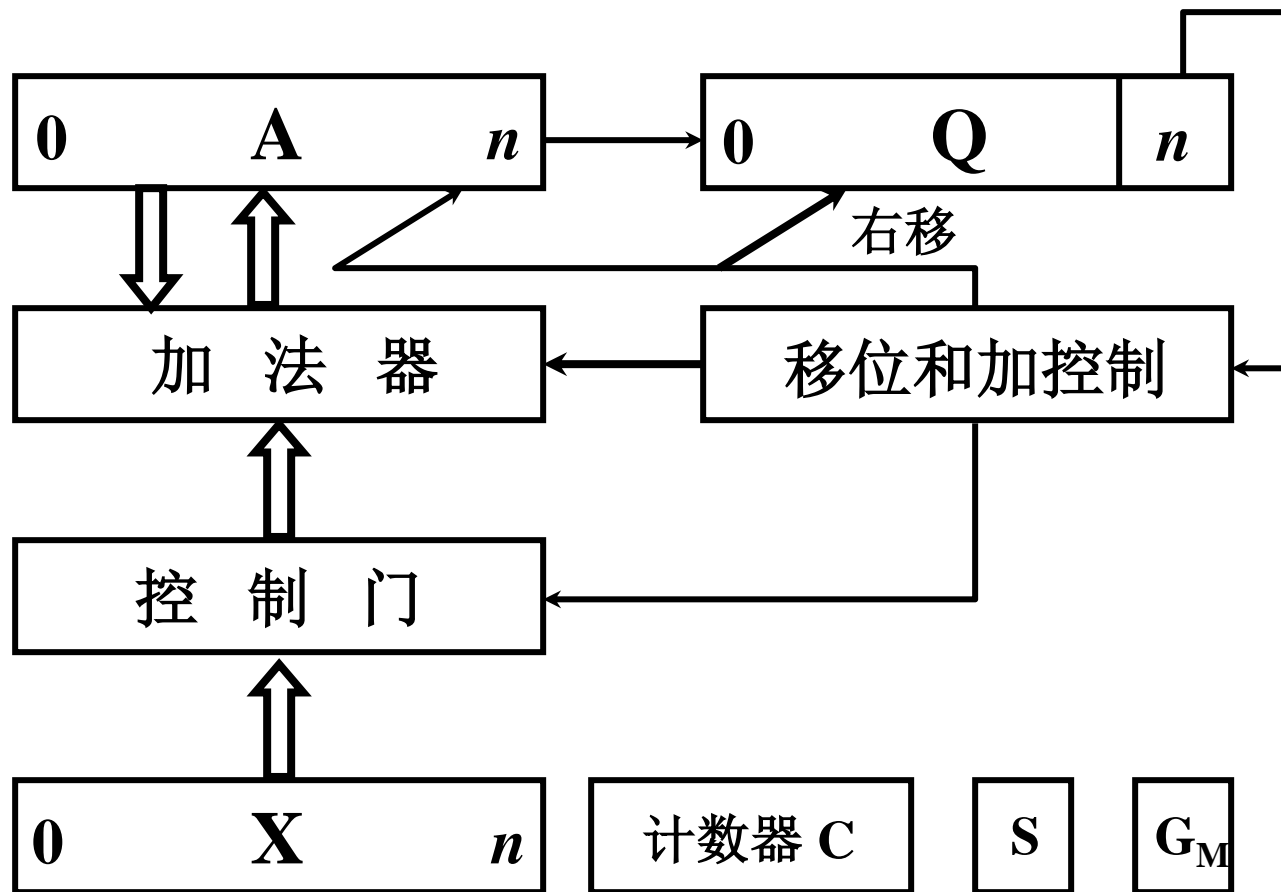
特点      绝对值运算

用移位的次数判断乘法是否结束

逻辑移位

### (3) 原码一位乘的硬件配置

6.3



A、X、Q 均  $n+1$  位

移位和加受末位乘数控制

## (4) 原码两位乘

6.3

原码乘

符号位 和 数值位 部分 分开运算

两位乘

每次用 乘数的 2 位判断 原部分积  
是否加 和 如何加 被乘数

乘数 $y_{n-1}y_n$	新的部分积
0 0	加 “0” $\rightarrow 2$
0 1	加 1 倍的被乘数 $\rightarrow 2$
1 0	加 2 倍的被乘数 $\rightarrow 2$
1 1	加 3 倍的被乘数 $\rightarrow 2$

$$\begin{array}{r} 3? \quad 4 \quad 100 \\ -1 \quad -01 \\ \hline 3 \quad 11 \end{array}$$

先 减 1 倍 的被乘数  
再 加 4 倍 的被乘数

# (5) 原码两位乘运算规则

6.3

乘数判断位 $y_{n-1}y_n$	标志位 $C_j$	操作内容
0 0	0	$z \rightarrow 2, y^* \rightarrow 2, C_j$ 保持 “0”
0 1	0	$z+x^* \rightarrow 2, y^* \rightarrow 2, C_j$ 保持 “0”
1 0	0	$z+2x^* \rightarrow 2, y^* \rightarrow 2, C_j$ 保持 “0”
1 1	0	$z-x^* \rightarrow 2, y^* \rightarrow 2, \text{置 “1” } C_j$
0 0	1	$z+x^* \rightarrow 2, y^* \rightarrow 2, \text{置 “0” } C_j$
0 1	1	$z+2x^* \rightarrow 2, y^* \rightarrow 2, \text{置 “0” } C_j$
1 0	1	$z-x^* \rightarrow 2, y^* \rightarrow 2, C_j$ 保持 “1”
1 1	1	$z \rightarrow 2, y^* \rightarrow 2, C_j$ 保持 “1”

共有操作

$+x^*$

$+2x^*$

$-x^*$

$\rightarrow 2$

实际操作

$+ [x^*]_{\text{补}}$

$+ [2x^*]_{\text{补}}$

$+ [-x^*]_{\text{补}}$

$\rightarrow 2$  补码移

例6.22 已知  $x = 0.111111$   $y = -0.111001$  求  $[x \cdot y]_{\text{原}}$

6.3

解：数值部分的运算

补码右移

补码右移

	乘 数	$C_j$	说 明
000.000000	00.11100 <u>1</u>	<u>0</u>	初态 $z_0 = 0$
000.111111			$+ x^*, C_j = 0$
000.111111			
000.001111	11 0011 <u>10</u>	<u>0</u>	$\rightarrow 2$
001.111110			$+ 2x^*, C_j = 0$
010.001101	11		
000.100011	0111 00 <u>11</u>	<u>0</u>	$\rightarrow 2$
111.000001			$- x^*, C_j = 1$
111.100100	0111		
111.111001	000111 <u>00</u>	<u>1</u>	$\rightarrow 2$
000.111111			$+ x^*, C_j = 0$
000.111000	000111		



① 乘积的符号位  $x_0 \oplus y_0 = 0 \oplus 1 = 1$

② 数值部分的运算

$$x^* \cdot y^* = 0.111000000111$$

$$\text{则 } [x \cdot y]_{\text{原}} = 1.111000000111$$

特点      绝对值的补码运算

用移位的次数判断乘法是否结束

算术移位

## (6) 原码两位乘和原码一位乘比较

6.3

	原码一位乘	原码两位乘
符号位	$x_0 \oplus y_0$	$x_0 \oplus y_0$
操作数	绝对值	绝对值的补码
移位	逻辑右移	算术右移
移位次数	$n$	$\frac{n}{2}$ ( $n$ 为偶数)
最多加法次数	$n$	$\frac{n}{2} + 1$ ( $n$ 为偶数)

思考  $n$  为奇数时，原码两位乘 移？次 最多加？次

# 5. 补码乘法

6.3

(以小数为例)

设 被乘数  $[x]_{\text{补}} = x_0.x_1x_2 \dots x_n$

$$[2^{-1}x]_{\text{补}} = x_0.\textcolor{brown}{x}_0x_1x_2 \dots x_n = 2^{-1}[x]_{\text{补}}$$

$$[-\textcolor{brown}{0.110}]_{\text{补}} = 1.010$$

$$[2^{-1}(-\textcolor{brown}{0.110})]_{\text{补}} = [-\textcolor{brown}{0.0110}]_{\text{补}} = 1.\textcolor{brown}{1}010$$

乘数  $[y]_{\text{补}} = y_0.y_1y_2 \dots y_n$

$$y = [y]_{\text{补}} - 2y_0$$

$$= y_0.y_1y_2 \dots y_n - 2y_0$$

$$= 0.y_1y_2 \dots y_n - y_0$$

$y$ 为正时 $y_0 = 0$

$y$ 为负时 $y_0 = 1$

## (1) 补码一位乘运算规则

### ① 校正法

$$[x \cdot y]_{\text{补}}$$

$$= [x (0.y_1 \cdots y_n -y_0)]_{\text{补}}$$

$$= [x]_{\text{补}} (0.y_1 \cdots y_n) - [x]_{\text{补}} \cdot y_0$$

$$= [x]_{\text{补}} (y_1 2^{-1} + y_2 2^{-2} + \cdots + y_n 2^{-n}) - [x]_{\text{补}} \cdot y_0$$

$$= [x]_{\text{补}} (y_1 2^{-1} + y_2 2^{-2} + \cdots + y_n 2^{-n}) + [-x]_{\text{补}} \cdot y_0$$

和原码一位乘法相同，部分积右移时采用算术移位

用校正法求 $[x \cdot y]_{\text{补}}$ 是用 $[x]_{\text{补}}$ 乘以 $[y]_{\text{补}}$ 的数值位，最后再根据 $[y]_{\text{补}}$ 的符号 $y_0$ 做一次校正

$\left\{ \begin{array}{l} y \text{ 为负, 加 } [-x]_{\text{补}} \\ y \text{ 为正, 不加} \end{array} \right.$

### ③ Booth 算法（被乘数、乘数符号任意） 6.3

设  $[x]_{\text{补}} = x_0 \cdot x_1 x_2 \cdots x_n$      $[y]_{\text{补}} = y_0 \cdot y_1 y_2 \cdots y_n$

$[x \cdot y]_{\text{补}}$

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

$$= [x]_{\text{补}} (0 \cdot y_1 \cdots y_n) - [x]_{\text{补}} \cdot y_0$$

$$= [x]_{\text{补}} (y_1 2^{-1} + y_2 2^{-2} + \cdots + y_n 2^{-n}) - [x]_{\text{补}} \cdot y_0$$

$$2^{-1} = 2^0 - 2^{-1}$$

$$= [x]_{\text{补}} (-y_0 + y_1 2^{-1} + y_2 2^{-2} + \cdots + y_n 2^{-n})$$

$$2^{-2} = 2^{-1} - 2^{-2}$$

$$= [x]_{\text{补}} [-y_0 + (y_1 - y_1 2^{-1}) + (y_2 2^{-1} - y_2 2^{-2}) + \cdots + (y_n 2^{-(n-1)} - y_n 2^{-n})]$$

$$= [x]_{\text{补}} [(y_1 - y_0) + (y_2 - y_1) 2^{-1} + \cdots + (y_n - y_{n-1}) 2^{-(n-1)} + (0 - y_n) 2^{-n}]$$

$$= [x]_{\text{补}} [(y_1 - y_0) + (y_2 - y_1) 2^{-1} + \cdots + (y_{n+1} - y_n) 2^{-n}]$$

附加位  $y_{n+1}$

## ④ Booth 算法递推公式

6.3

$$[z_0]_{\text{补}} = 0$$

$$[z_1]_{\text{补}} = 2^{-1} \{ (y_{n+1} - y_n) [x]_{\text{补}} + [z_0]_{\text{补}} \} \quad y_{n+1} = 0$$

⋮

$$[z_n]_{\text{补}} = 2^{-1} \{ (y_2 - y_1) [x]_{\text{补}} + [z_{n-1}]_{\text{补}} \}$$

$$[x \cdot y]_{\text{补}} = [z_n]_{\text{补}} + (y_1 - y_0) [x]_{\text{补}}$$

最后一步不移位

如何实现  
 $y_{i+1} - y_i$  ?

$y_i$	$y_{i+1}$	$y_{i+1} - y_i$	操作
0	0	0	$\rightarrow 1$
0	1	1	$+ [x]_{\text{补}} \rightarrow 1$
1	0	-1	$+ [-x]_{\text{补}} \rightarrow 1$
1	1	0	$\rightarrow 1$