

# 第二章 数据的机器级表示

数值数据的表示

非数值数据的表示

数据的宽度和存储

数据的校验码

# 数据的表示

---

- 分以下三个部分介绍
  - **第一讲：数值数据的表示**
    - 定点数的编码表示、整数的表示、无符号整数、带符号整数、浮点数的表示
    - C语言程序的整数类型和浮点数类型
  - **第二讲：非数值数据的表示、数据的存储**
    - 逻辑值、西文字符、汉字字符
    - 数据宽度单位、大端/小端、对齐存放
  - **第三讲：数据的校验码**
    - 奇偶校验

# 课程内容概要

```
/*---sum.c---*/
```

```
int sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

```
/*---main.c---*/
```

```
int main()
{
    int a[1]={100};
    int sum;
    sum=sum(a,0);
    printf("%d",sum);
}
```

数据的表示

数据的运算

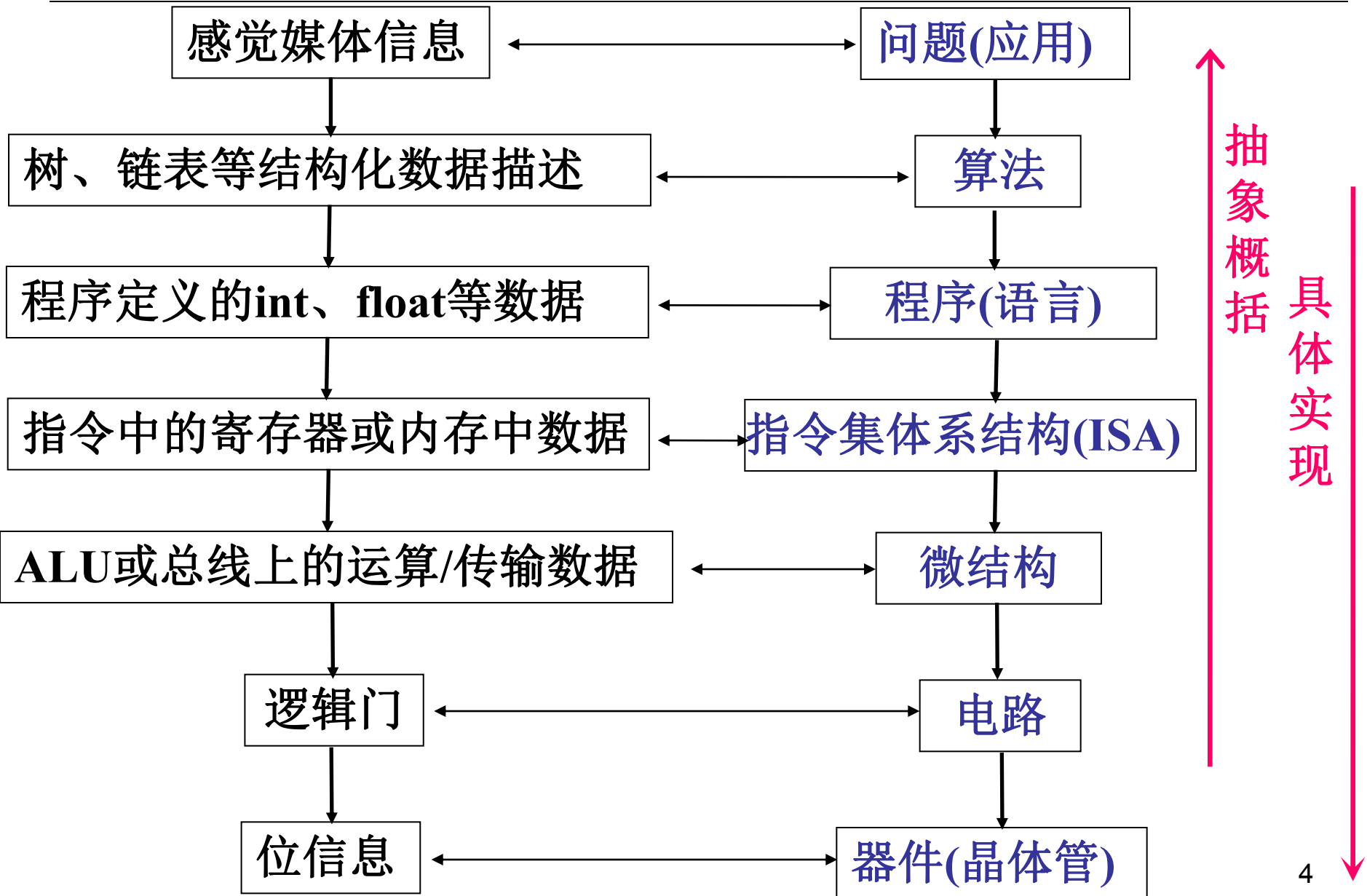
如果程序处理的是图像、视频、声音、文字等数据，那么，

(1) 如何获得这些数据？

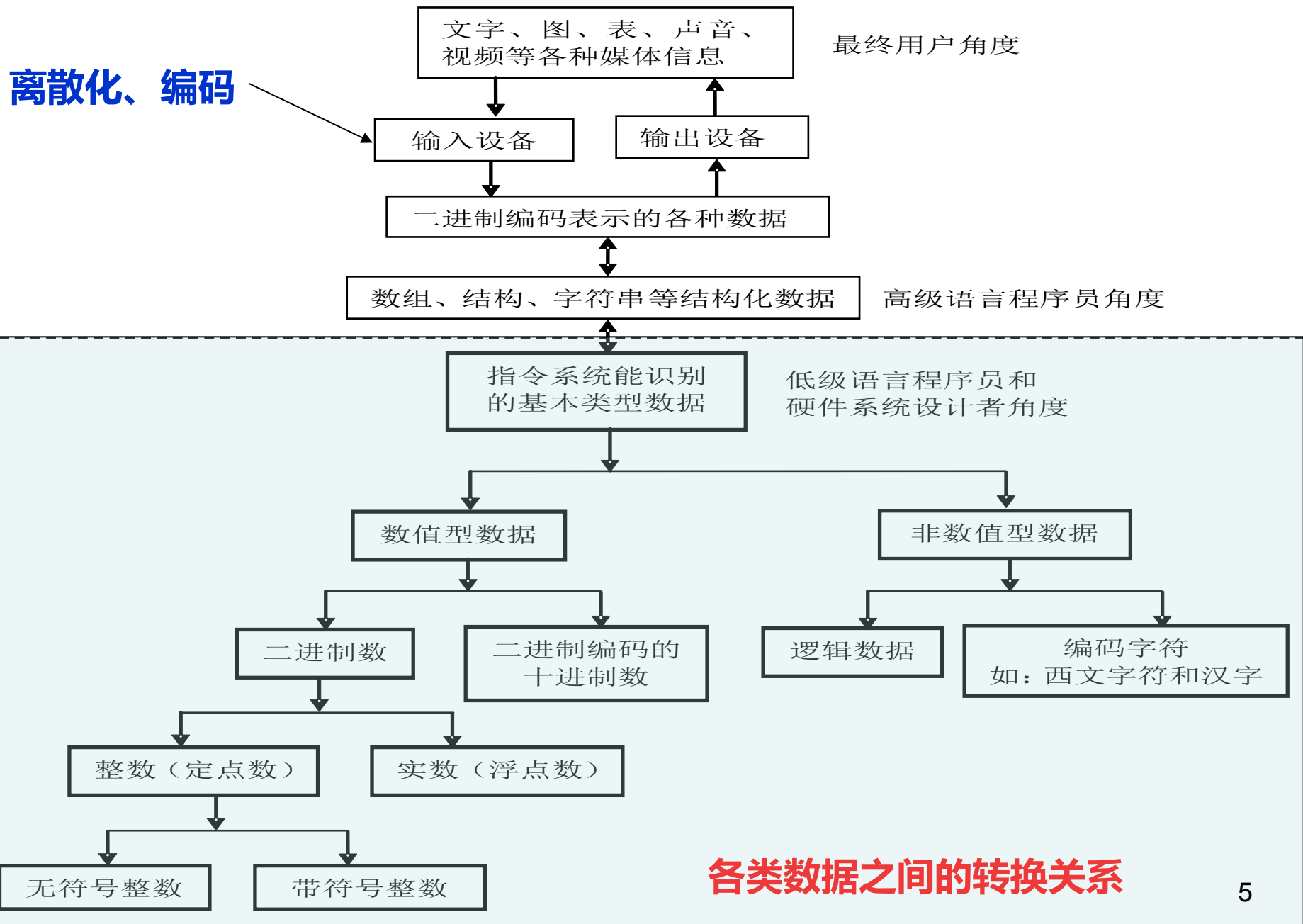
(2) 如何表示这些数据？

(3) 如何处理这些数据？

# “转换”的概念在数据表示中的反映



## 离散化、编码



# 数码相机成像

---

过程:光线从镜头这样的光学系统进入相机,光聚焦在CCD (Charge-coupled Device,电荷耦合元件)或CMOS (Complementary Metal Oxide Semiconductor互补金属氧化物半导体),进行滤色、感光(光电转化成电信号),经处理器加工,按照一定的排列方式将拍摄物体“分解”成了一个一个的像素点,这些像素点以模拟图像信号的形式转移到“模数(A/D)转换器”上,将每个像素上光电信号转变成数码信号,再经DSP(Digital Signal Processing)图像处理器处理成数码图像,处理成真正的图像,之后压缩存储到相机的存储介质中,通过电脑处理和显示器的电光转换,或经打印机打印便形成影象。

# 数值数据的表示

- 数值数据表示的三要素

- 进位计数制: 十进制、二进制、十六进制、八进制数
- 定点、浮点数表示: 定点整数、定点小数、浮点数
- 如何用二进制编码: 定点数的编码, 浮点数的编码

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? 整数? 小数? 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 )

- 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 )

# Signed integer（带符号整数，定点整数）

---

- 计算机必须能处理正数(positive) 和负数(negative), MSb(最高位的)表示数的符号(简称数符)
- 有三种定点编码方式
  - Signed magnitude（原码） 现用来表示浮点（实）数的尾数
  - One's complement（反码） 现已不用于表示数值数据
  - Two's complement（补码）  
50年代以来，所有计算机都用补码来表示定点整数
- 为什么用补码表示带符号整数？
  - 补码运算系统是模运算系统，加、减运算统一
  - 数0的表示唯一，方便使用
  - 比原码和反码多表示一个最小负数



# Sign and Magnitude（原码的表示）

Decimal	Binary
---------	--------

0	0000
---	------

1	0001
---	------

2	0010
---	------

3	0011
---	------

4	0100
---	------

5	0101
---	------

6	0110
---	------

7	0111
---	------

Decimal	Binary
---------	--------

-0	1000
----	------

-1	1001
----	------

-2	1010
----	------

-3	1011
----	------

-4	1100
----	------

-5	1101
----	------

-6	1110
----	------

-7	1111
----	------

◆ 容易理解， 但是：

- ✓ 0 的表示不唯一，故不利于程序员编程
- ✓ 加、减运算方式不统一
- ✓ 需额外对符号位进行处理，故不利于硬件设计
- ✓ 特别当  $a < b$  时，实现  $a - b$  比较困难

从 50年代开始，整数都采用补码来表示，但浮点数的尾数用原码定点小数表示

# 补码 - 模运算 (modular 运算)

重要概念：在一个模运算系统中，一个数与它除以“模”后的余数等价。

时钟是一种模12系统 现实世界中的模运算系统

假定钟表时针指向10点，要将它拨向6点， 则有两种拨法：

① 倒拨4格：  $10 - 4 = 6$

② 顺拨8格：  $10 + 8 = 18 \equiv 6 \pmod{12}$

模12系统中：  $10 - 4 \equiv 10 + 8 \pmod{12}$

$-4 \equiv 8 \pmod{12}$

则，称8是-4对模12的补码（即：-4的模12补码等于8）。

同样有  $-3 \equiv 9 \pmod{12}$

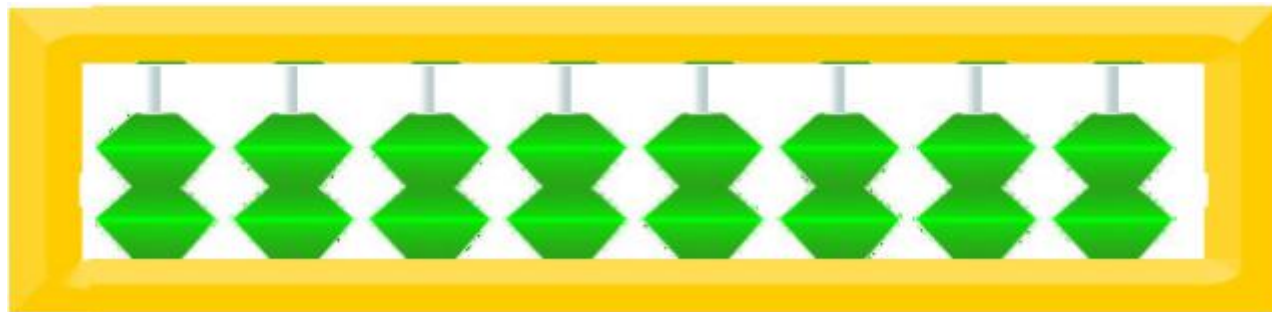
$-5 \equiv 7 \pmod{12}$  等

结论1： 一个负数的(2的)补码等于模减该负数的绝对值。

结论2： 对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一数负数的(2的)补码来代替。

补码 (modular 运算)：+ 和- 的统一

# 计算机中的运算器是模运算系统



## 8位二进制加法器模运算系统

计算  $0111\ 1111 - 0100\ 0000 = ?$

$$\begin{aligned} 0111\ 1111 - 0100\ 0000 &= 0111\ 1111 + (2^8 - 0100\ 0000) \\ &= 0111\ 1111 + 1100\ 0000 = \boxed{1}0011\ 1111 \pmod{2^8} \\ &= 0011\ 1111 \end{aligned}$$

只留余数，“1”被丢弃

结论1： 一个负数的补码等于对应正数补码的“各位取反、末位加一”

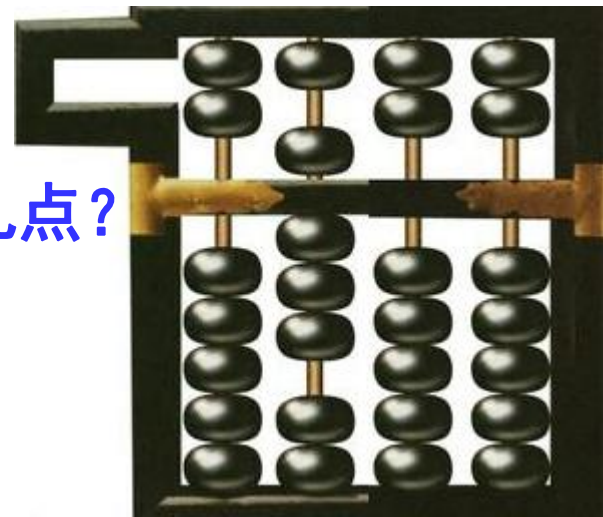
## (2的)补码的表示

### 现实世界的模运算系统举例

#### 例1：“钟表”模运算系统

假定时针只能顺拨，从10点倒拨4格后是几点？

$$10 - 4 = 10 + (12 - 4) = 10 + 8 = 6 \pmod{12}$$



#### 例2：“4位十进制数”模运算系统

假定算盘只有四档，且只能做加法，则在算盘上计算  
9828-1928等于多少？

$$9828 - 1928 = 9828 + (10^4 - 1928)$$

$$= 9828 + 8072$$

$$= \boxed{1}7900$$

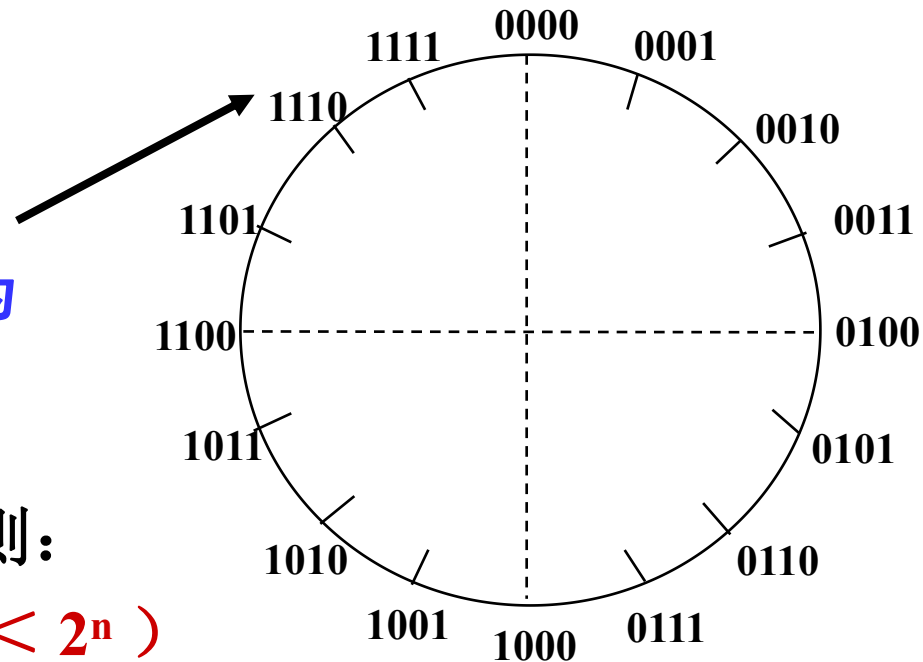
取模即只留余数，高位“1”被丢弃！  
相当于只有低4位留在算盘上。

$$= 7900 \pmod{10^4}$$

# 运算器适合用补码表示和运算

运算器只有有限位，假设为 $n$ 位，则运算结果只能保留低 $n$ 位，故可看成是个只有 $n$ 档的二进制算盘，因此，其模为 $2^n$ 。

当 $n=4$ 时，共有16个机器数：  
0000 ~ 1111，可看成是模为  
 $2^4$ 的钟表系统。真值的范围为  
 $-8 \sim +7$



补码的定义 假定补码有 $n$ 位，则：

$$[X]_{\text{补}} = (2^n + X) \bmod 2^n \quad (-2^n \leq X < 2^n)$$

$X$ 是真值， $[x]_{\text{补}}$ 是机器数

真值和机器数的含义是什么？

# 求特殊数的补码

假定机器数有n位

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

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

$$\textcircled{3} [+0]_{\text{补}} = [-0]_{\text{补}} = 00\dots0 \text{ (n个0)}$$

32位机器中 , int、 short、 char型数据的机器数各占几位 ?

# 补码与真值之间的简便转换

例：设机器数有8位，求123和-123的补码表示。

如何快速得到123的二进制表示？

解：123 = 127 - 4 = 01111111B - 100B = 01111011B



-123 = -01111011B

$[01111011]_{\text{补}} = 2^8 + 01111011 = 100000000 + 01111011$   
 $= 01111011 \pmod{2^8}$ ，即 7BH。

$[-01111011]_{\text{补}} = 2^8 - 01111011 = 10000\ 0000 - 01111011$   
 $= (1111\ 1111 + 1) - 0111\ 1011$   
 $= (1111\ 1111 - 0111\ 1011) + 1$   
 $= 1000\ 0100 + 1 \longleftarrow \text{各位取反，末位加1}$   
 $= 1000\ 0101$ ，即 85H。

当机器数为16位时，结果怎样？

# Unsigned integer(无符号整数)

- 机器中字的位排列顺序有两种方式：（例：32位字： $0\dots01011_2$ ）
  - 高到低位从左到右： $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011$  
  - 高到低位从右到左： $1101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$  
  - Leftmost和rightmost这两个词有歧义，故用LSb(Least Significant Bit)来表示最低有效位，用MSB来表示最高有效位
  - 高位到低位多采用从左往右排列
- 一般在全部是正数运算且不出现负值结果的情况下，可使用无符号数表示。例如，地址运算，编号表示，等等
- 无符号整数的编码中**没有符号位**
- 能表示的最大值大于位数相同的带符号整数的最大值（Why？）
  - 例如，8位无符号整数最大是255（1111 1111）  
而8位带符号整数最大为127（0111 1111）
- 总是正数，所以很多时候就**简称为“无符号数”**



# C语言程序中的整数

无符号数： unsigned int ( short / long)； 带符号整数： int ( short / long)

常在一个数的后面加一个 “u”或 “U”表示无符号数

若同时有无符号和带符号整数，则C编译器将带符号整数强制转换为无符号数，

以下关系表达式在32位用补码表示的机器上执行，带\*的结果与常规预想的相反！

关系表达式	类型	结果	说明
$0 == 0U$	无	1	$00...0B = 00...0B$
$-1 < 0$	带	1	$11...1B (-1) < 00...0B (0)$
$-1 < 0U$	无	0*	$11...1B (2^{32}-1) > 00...0B(0)$
$2147483647 > -2147483647 - 1$	带	1	$011...1B (2^{31}-1) > 100...0B (-2^{31})$
$2147483647U > -2147483647 - 1$	无	0*	$011...1B (2^{31}-1) < 100...0B(2^{31})$
$2147483647 > (int) 2147483648U$	带	1*	$011...1B (2^{31}-1) > 100...0B (-2^{31})$
$-1 > -2$	带	1	$11...1B (-1) > 11...10B (-2)$ <sub>17</sub>
$(unsigned) -1 > -2$	无	1	$11...1B (2^{32}-1) > 11...10B (2^{32}-2)$

# C语言程序中的整数

例如，考虑以下C代码：

```
1 int x = -1; // = 32位的 "11...1"
2 unsigned u = 2147483648; // = 32位的 "100...0"
3
4 printf ( "x = %u = %d\n" , x, x); // x = 4294967295 = -1
5 printf ( "u = %u = %d\n" , u, u);
   // u = 2147483648 = -2147483648
```

在32位机器上运行上述代码时，它的输出结果是什么？为什么？

$x = 4294967295 = -1$  // 因为 = 32位的 "11...1"

$u = 2147483648 = -2147483648$  // 因为 = 32位的 "100...0"

◆ 因为-1的补码整数表示为 "11...1"，作为32位无符号数解释时，其值为 $2^{32}-1 = 4\ 294\ 967\ 296-1 = 4\ 294\ 967\ 295$ 。

◆  $2^{31}$ 的无符号数表示为 "100...0" = 2147483648，被解释为32位带符号整数时，其值为最小负数： $-2^{32-1} = -2^{31} = -2\ 147\ 483\ 648$

# C语言程序中的整数

	常量范围	类型
ISO C90标准	$0 \sim 2^{31}-1$	int
	$2^{31} \sim 2^{32}-1$	unsigned int
	$2^{32} \sim 2^{63}-1$	long long
	$2^{63} \sim 2^{64}-1$	unsigned long long
ISO C99标准	$0 \sim 2^{31}-1$	int
	$2^{31} \sim 2^{63}-1$	long long
	$2^{63} \sim 2^{64}-1$	unsigned long long

2011年12月8日，国际标准化组织（ISO）和国际电工委员会（IEC）再次发布了C语言的新标准，名叫**ISO/IEC 9899:2011-Information technology--Programming languages -- C** <sup>[10]</sup>，简称**C11标准**，原名**C1X**。这是C语言的第三个官方标准，也是C语言的最新标准。

# C语言程序中的整数

- 1) 在有些32位系统上，C表达式 $-2147483648 < 2147483647$ 的执行结果为false。Why？
- 2) 若定义变量“`int i=-2147483648;`”，则“`i < 2147483647`”的执行结果为true。Why？
- 3) 如果将表达式写成“ $-2147483647-1 < 2147483647$ ”，则结果会怎样呢？Why？

1) 在ISO C90标准下，2147483648为unsigned int型，因此

“ $-2147483648 < 2147483647$ ”按无符号数比较，  
10.....0B比01.....1大，结果为false。

由C语言中的  
“Integer  
Promotion”  
规则决定的。

在ISO C99标准下，2147483648为long long型，因此

“ $-2147483648 < 2147483647$ ”按带符号整数比较，  
10.....0B比01.....1小，结果为true。

2) `i < 2147483647`按int型数比较，结果为true。

3)  $-2147483647-1 < 2147483647$ 按int型比较，结果为true。

# 科学计数法(Scientific Notation)与浮点数

## Example:

*mantissa* (尾数) → **6.02** × **10<sup>21</sup>** ← *exponent*(阶、指数)  
*decimal point* → . ← *radix* (base, 基)

- **Normalized form (规格化形式)** : 小数点前只有一位非0数
- 同一个数有多种表示形式。例：对于数 1/1,000,000,000
  - Normalized (唯一的规格化形式):  $1.0 \times 10^{-9}$
  - Unnormalized (非规格化形式不唯一) :  $0.1 \times 10^{-8}$ ,  $10.0 \times 10^{-10}$

## for Binary Numbers:

***mantissa*** (尾数) → **0.101**<sub>two</sub> x **2**<sup>-10</sup> ← ***exponent*** (指数)

← ***binary point*** ← **基为2**

**只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）**

# 浮点数(Floating Point)的表示范围

例：画出下述32位浮点数格式的规格化数的表示范围。

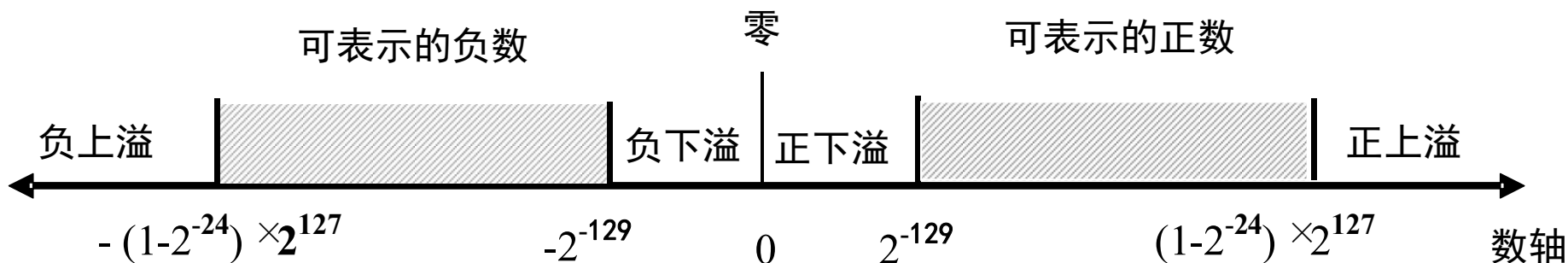


第0位数符S；第1~8位为8位移码表示阶码E（偏置常数为128）；  
第9~31位为24位二进制原码小数表示的尾数M。

规格化尾数的小数点后第一位总是1，故规定第一位默认的“1”不明显表示出来。这样可用23个数位表示24位尾数。

最大正数：0.11...1 × 2<sup>11...1</sup> = (1-2<sup>-24</sup>) × 2<sup>127</sup>    最小正数：0.10...0 × 2<sup>00...0</sup> = (1/2) × 2<sup>-128</sup>

因为原码是对称的，所以其表示范围关于原点对称。



机器0：尾数为0 或 落在下溢区中的数

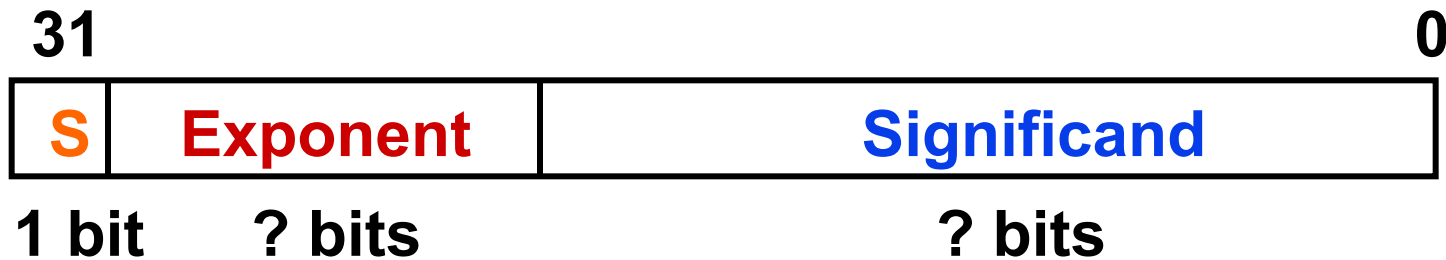
浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

# 浮点数的表示

- Normal format（规格化数形式）：

$\text{+/-1.xxxxxxxxxxxx} \times R^{\text{Exponent}}$

- 32-bit 规格化数：



**S** 是符号位（Sign）

**Exponent** 用移码（增码）来表示

**Significand** 表示 xxxxxxxxxxxxxx，尾数部分

（基可以是 2 / 4 / 8 / 16，约定信息，无需显式表示）

- 早期的计算机，各自定义自己的浮点数格式

问题：浮点数表示不统一会带来什么问题？

# “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IEEE 754的制定

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

规格化数： $\pm 1.\text{xxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$

规定：小数点前总是“1”，故可隐含表示。  
注意：和前面例子规定不一样，这里更合理！

Single Precision：

S	Exponent	Significand
1 bit	8 bits	23 bits

- 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(单精度)规格化数阶码范围为00000001(-126)~11111110(127)

- bias为127 (single单), 1023 (double双)

为什么用127？若用128，  
则阶码范围为多少？

- Significand（尾数）：

- 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:  $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$       0000 0001 (-127) ~ 1111 1110 (126)

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

# Ex: Converting Binary FP to Decimal

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

1011 11101 110 0000 0000 0000 0000 0000

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

- **Sign:** 1  $\Rightarrow$  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:**  $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$

# Ex: Converting Decimal to FP

---

**-12.75**

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

# Normalized numbers (规格化数)

---

前面的定义都是针对规格化数 (normalized form)

其它非规格化的数呢？

Exponent	Significand	Object
<b>1-254</b>	<b>anything implicit leading 1</b>	<b>Norms</b>
<b>0</b>	<b>0</b>	<b>?</b>
<b>0</b>	<b>nonzero</b>	<b>?</b>
<b>255</b>	<b>0</b>	<b>?</b>
<b>255</b>	<b>nonzero</b>	<b>?</b>

# Representation for 0

## How to represent 0? +0.0,和 -0.0都可以有效表示

**exponent:** all zeros

**significant: all zeros**

## What about sign? Both cases valid.

**+0: 0 00000000 000000000000000000000000**

```
-0: 1 00000000 000000000000000000000000
```

# Representation for $+\infty/-\infty$

In FP, 除数为0的结果是  $\pm\infty$ , 不是溢出异常. (整数除0为异常)

为什么要这样处理?

$\infty$  : infinity

- 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

$$5.0 / 0 = +\infty, \quad -5.0 / 0 = -\infty$$

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

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

# Representation for “Not a Number”

---

$\text{Sqrt}(-4.0) = ?$        $0/0 = ?$

- Called **Not a Number (NaN)** - “非数”, “非数值”

How to represent NaN

**Exponent** = 255

**Significand**: nonzero

**NaNs can help with debugging**

**Operations** 下面的操作生成一个 “非数值” 的数

$\text{sqrt}(-4.0) = \text{NaN}$

$0/0 = \text{NaN}$

$\text{op}(\text{NaN}, x) = \text{NaN}$

$+\infty + (-\infty) = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$

$\infty/\infty = \text{NaN}$

etc.

# Representation for Denorms(非规格化数)

What have we defined so far? (for SP)

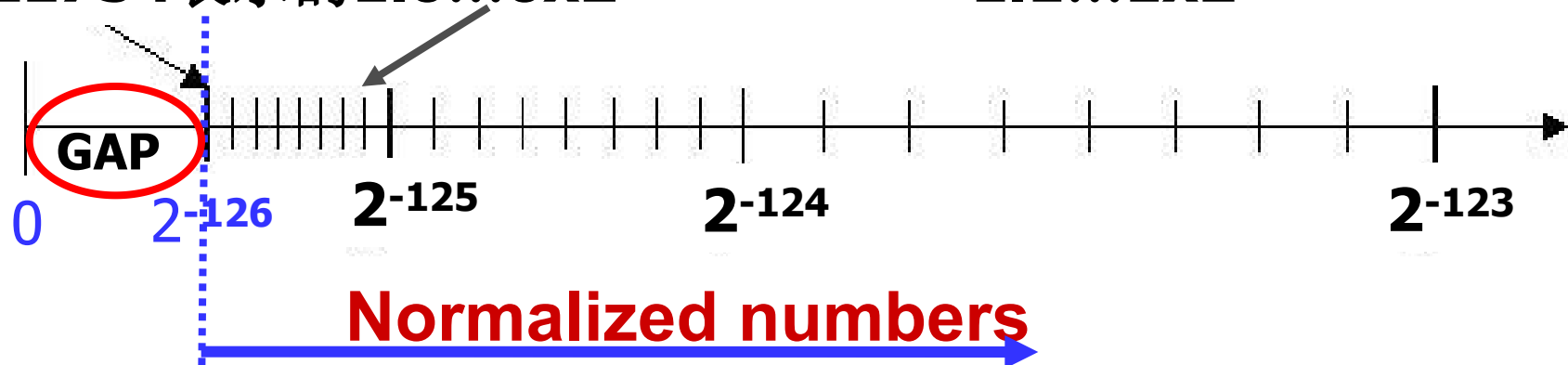
Exponent	Significand	Object
0	0	<b>+/-0</b> 代表非规格化数
0	<b>nonzero</b>	<b>Denorms</b>
1-254	anything implicit leading 1	<b>Norms</b>
255	0	<b>+/- infinity</b>
255	<b>nonzero</b>	<b>NaN</b>



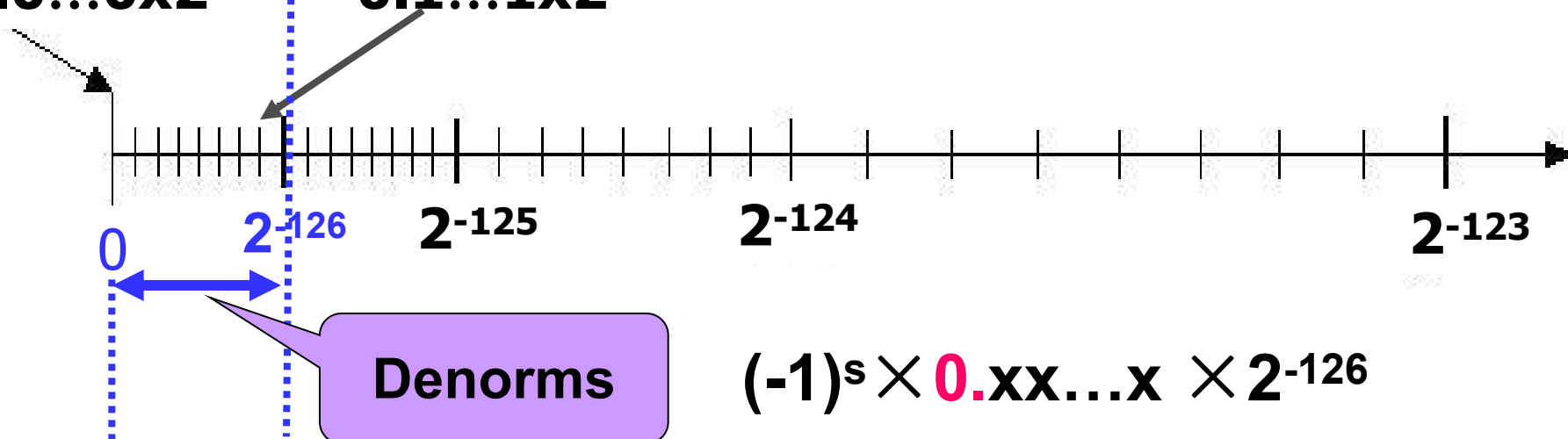
# Representation for Denorms(非规格化数)

$1.0...0 \times 2^{-126} \sim 1.1...1 \times 2^{-126}$

=IEEE754表示的  $1.0...0 \times 2^{(-126+127=1)} \sim 1.1...1 \times 2^{(-126+127=1)}$



$0.0...0 \times 2^{-126} \sim 0.1...1 \times 2^{-126}$



$$(-1)^s \times 0.xx...x \times 2^{-126}$$

# 关于浮点数精度的一个例子

```
#include <iostream>
using namespace std;
int main()
{
    float heads;
    cout.setf(ios::fixed, ios::floatfield);
    while(1)
    {
        cout << "Please enter a number: ";
        cin >> heads;
        cout << heads << endl;
    }
}
```

运行结果:

```
Please enter a number: 61.419997
61.419998
Please enter a number: 61.419998
61.419998
Please enter a number: 61.419999
61.419998
Please enter a number: 61.42
61.419998
Please enter a number: 61.420001
61.420002
Please enter a number:
```

**61.419998和61.420002是两个可表示数，两者之间相差0.000004。当输入数据是一个不可表示数时，机器将其转换为最邻近的可表示数。**

# 第一讲小结

10在计算机中有几种可能的表示？  
-10呢？

- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进制、定点/浮点、编码
- 整数的表示
  - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- C语言中的整数
  - 无符号数：unsigned int ( short / long)；带符号数：int ( short / long)
- 浮点数的表示
  - 符号；尾数：定点小数；指数（阶）：定点整数（基不用表示）
- 浮点数的范围
  - 正上溢、正下溢、负上溢、负下溢；与阶码的位数和基的大小有关
- 浮点数的精度：与尾数的位数和是否规格化有关
- 浮点数的表示（IEEE 754标准）：单精度SP（float）和双精度DP（double）
  - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
  - “零”（阶为全0，尾为全0）
  - $\infty$ （阶为全1，尾为全0）
  - NaN（阶为全1，尾为非0）
  - 非规格化数（阶为全0，尾为非0，隐藏位为0）
- 十进制数的表示：用ASCII码或BCD码表示

# 数据的表示

---

- 分以下三个部分介绍(红字部分为今天学习内容)
  - 第一讲：数值数据的表示
    - 定点数的编码表示、整数的表示、无符号整数、带符号整数、浮点数的表示
    - C语言程序的整数类型和浮点数类型
  - 第二讲：非数值数据的表示、数据的存储(今天)
    - 逻辑值、西文字符、汉字字符
    - 数据宽度单位、大端/小端、对齐存放
  - 第三讲：数据的校验码
    - 奇偶校验

# 逻辑数据的编码表示

- 表示

- 用一位表示。例如，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。

- 位串

- 用来表示若干个状态位或控制位（OS中使用较多）

例如，x86的标志寄存器含义如下：

				OF	DF	IF	TF	SF	ZF		AF		PF		CF <sub>37</sub>
--	--	--	--	----	----	----	----	----	----	--	----	--	----	--	------------------

# 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）

- 十进制数字：0/1/2.../9      //注:对应ASCII码 48-57<sub>10</sub>
- 英文字母：A/B/.../Z/a/b/.../z      //注:对应ASCII码 65-122<sub>10</sub>

- 必须熟悉上面对应的ASCII码！

- 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

# 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

- 有以下几种汉字码：

- 输入码：对汉字用相应按键进行编码表示，用于输入,如字音编码(微软拼音,全拼),字形编码(如五笔字型),形音编码
- 内码：用于在计算机系统中进行存储、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

# 汉字内码

因为汉字的总数多,至少需2个字节才能表示一个汉字内码。

可在GB2312国标码的基础上产生汉字内码

—为与ASCII码区别, 区位码(由区号和位号拼接成)加上32得到国标码, 将国标码的两个字节的第一位(bit)置“1”后得到一种汉字内码

—如,汉字“大”在(区位码)码表中位于第20行( $001-0100_2$ )、第83列( $101-0011_2$ )。因此区位码为 $001-0100_2(=20_{10})$   $101-0011_2(=83_2)$ 。

区位码各自加上 $32_{10}(=010-0000_2)$ 后, 国标码为( $001-0100+010-0000_2=$ ) $0011-0100_2(=34H)$ -( $101-0011+010-0000_2=$ ) $111-0011_2(=73H)$ , 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 ( $=1011-0100$   $1111-0011_2$ ), 因而不会和ASCII码混淆。



# 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

问题：如何知道到哪里找相应的字形信息？

汉字内码与其在字库中的位置有关！！

- 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）

# 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*)
  - 如果以字节为一个排列单位, 则LSB表示最低有效字节, MSB表示最高有效字节
- “字”(word)表示被处理信息的单位, 用来度量数据类型的宽度
  - x86体系结构定义1个“字” (WORD)的宽度为16位, 双字 (double WORD, DWORD) : 32位, 四字 (quad WORD, QWORD) : 64位

# “字长”的概念

- “字长”指CPU内部用于整数运算的数据通路的宽度(位数)

“字长”是指在同一时间CPU进行定点整数运算时能够处理的二进制数的位数，通常称处理字长为32位数据的CPU叫32位CPU。数据通路（第五章介绍）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”和“字长”的宽度可以一样，也可不同。

例如，x86体系结构，“字”的位数只有16位了。IA-32体系结构中“字”的位数也都是16位。

# 数据量的度量单位

- 存储二进制信息时的度量单位要比字节或字大得多
- 容量经常使用的单位有：
  - “千字节” (KB),  $1\text{KB}=2^{10}\text{字节}=1024\text{B}$
  - “兆字节” (MB),  $1\text{MB}=2^{20}\text{字节}=1024\text{KB}$
  - “千兆字节” (GB),  $1\text{GB}=2^{30}\text{字节}=1024\text{MB}$
  - “兆兆字节” (TB),  $1\text{TB}=2^{40}\text{字节}=1024\text{GB}$
- 通信中的带宽使用的单位有：
  - “千比特/秒” (kb/s),  $1\text{kbps}=10^3\text{ b/s}=1000\text{ bps}$
  - “兆比特/秒” (Mb/s),  $1\text{Mbps}=10^6\text{ b/s}=1000\text{ kbps}$
  - “千兆比特/秒” (Gb/s),  $1\text{Gbps}=10^9\text{ b/s}=1000\text{ Mbps}$
  - “兆兆比特/秒” (Tb/s),  $1\text{Tbps}=10^{12}\text{ b/s}=1000\text{ Gbps}$

如果把b换成B，则表示字节而不是比特（位）

例如，10MBps表示 10兆字节/秒

# 程序中数据类型的宽度

- 高级语言支持多种类型、多种长度的数据

- 例如，C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）

- 不同机器上表示的同一种类型的数据可能宽度不同

- 必须确定相应的机器级数据表示方式和相应的处理指令

C语言中数值数据类型的宽度 (单位: 字节)

C声明	典型32位机器 (单位:字节)	Compaq Alpha 机器 (单位:字节)
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

Compaq Alpha是一个针对高端应用的64位机器，即字长为64位

可见：同类型数据并不是所有机器都采用相同的宽度，分配的字节数随机器字长和编译器的不同而不同。

# 数据的存储和排列顺序

- 80年代开始, 几乎所有机器都用**字节编址**  $65535 = 2^{16} - 1$
- ISA设计时要考虑的两个问题:  $[-65535]_{\text{补}} = \text{FFFF0001H}$ 
  - 如何根据一个字节地址取到一个32位的字? - **字的存放问题**
  - 一个字能否存放在任何字节边界? - **字的边界对齐问题**

若  $\text{int } i = -65535$ , 存放在100号单元 (占100~103), 则用“取数”指令访问100号单元取出  $i$  时, 必须清楚  $i$  的4个字节是如何存放的。

Word:	FF	FF	00	01	little endian word 100#
	103	102	101	100	
	MSB			LSB	
	100	101	102	103	big endian word 100#

**大端方式 ( Big Endian )**: MSB所在的地址是数的地址

**e.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA**

**小端方式 ( Little Endian )**: LSB所在的地址是数的地址

**e.g. Intel 80x86, DEC VAX**

有些机器两种方式都支持, 可通过特定控制位来设定采用哪种方式。

# BIG Endian versus Little Endian

Ex3: Memory layout of a instruction located in 1000

即指令地址为1000

假定小端机器中指令: `mov AX, 0x12345(BX)`

其中操作码mov为40H, 寄存器AX和BX的编号分别为0001B和0010B, 立即数占32位, 则存放顺序为:



若在大端机器上, 则存放顺序如何?



00	1005	45
01	1004	23
23	1003	01
45	1002	00
12	1001	12
40	1000	40

地址

只需要考虑指令中立即数的顺序!

# Byte Swap Problem (字节交换问题)

78	3
56	2
34	1
12	0

Big Endian

↑  
increasing  
byte  
address

12	3
34	2
56	1
78	0

Little Endian

上述存放在0号单元的数据（字）是什么？ **12345678H?** **78563412H?**

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- ◆ 每个系统内部是一致的，但在系统间通信时可能会发生问题！
- ◆ 因为顺序不同，需要进行顺序转换

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

**ex. Little endian: GIF, PC Paintbrush, Microsoft RTF, etc**

**Big endian: Adobe Photoshop, JPEG, MacPaint, etc**



# Alignment(对齐)

**Alignment:** 要求数据存放的起始地址是相应的边界地址

- 数据存放的起始地址必须为该数据的数据类型所占字节数的倍数
    - `int` 型变量（4字节）的存储起始地址必须为4的倍数
    - `short`型变量（2字节）的存储起始地址必须为2的倍数
  - 目前机器字长一般为32位或64位，而存储器地址按字节编址
  - 指令系统支持对字节、半字、字及双字的运算，也有位处理指令
  - 各种不同长度的数据存放时，有两种处理方式：
    - 按边界对齐（假定存储“字”的宽度为32位，按字节编址，最多每4个字节可同时读写）
      - 字地址：4的倍数（地址的二进制表示的最低两位为0）
      - 半字地址：2的倍数（低位为0）
      - 字节地址：任意（1的倍数）
    - 不按边界对齐
- 坏处： 可能会增加访存次数！（学了存储器组织后会更明白！）

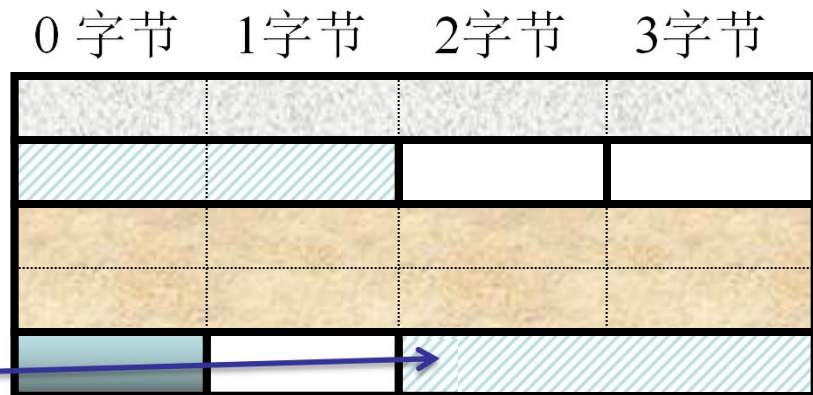
# Alignment(对齐)

**按边界对齐存储/访问:**存储器按字节编址,数据按其数据类型的字节大小的倍数作为基址开始连续的1、2、3或4个字节存储,每次访存从此基址开始读写。

**按边界对齐存放**

int i;	//&i=0;	00
short k;	//&k=4	04
double x;	//&x=8	08
char c;	//&c=16	12
short j;	//&j=18	16

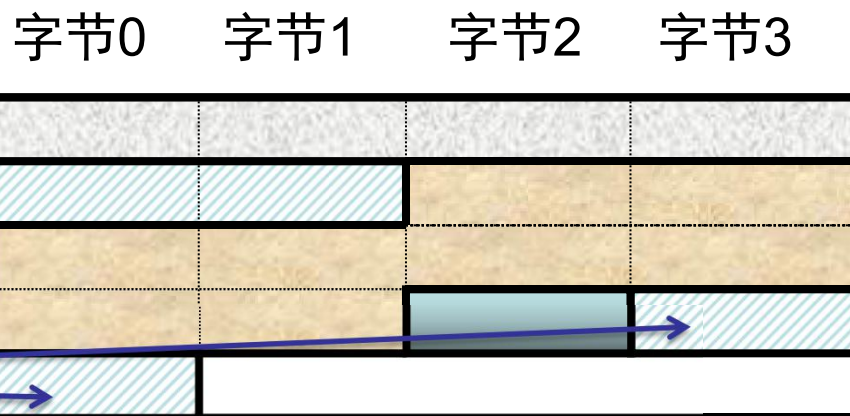
x: 2次访存  
j: 1次访存



**不按边界对齐存放**

int i;	//&i=0;	00
short k;	//&k=4	04
double x;	//&x=6	08
char c;	//&c=14	12
short j,...	//&j=15	16

x: 3次访存  
j: 2次访存



**不按边界对齐虽节省了空间,但增加了访存次数! 目前来看,按照边界对齐可能会浪费一点存储空间,但是没有关系!**

# Alignment(对齐) 举例

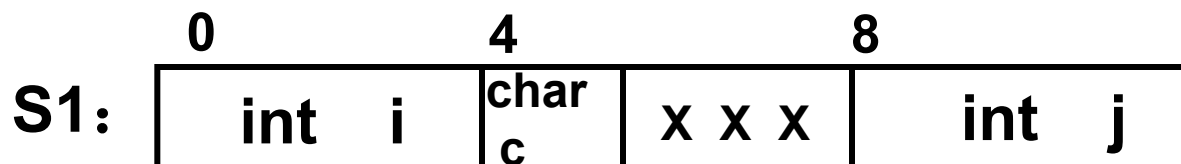
例如，考虑下列两个结构声明：

```
struct S1 {  
    int    i;  
    char   c;  
    int    j;  
};
```

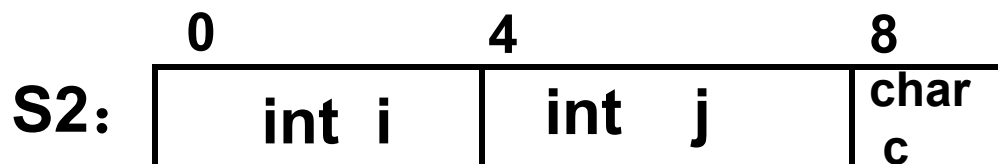
```
struct S2 {  
    int    i;  
    int    j;  
    char   c;  
};
```

在要求对齐的情况下，哪种结构声明更好？

**S2比S1好**

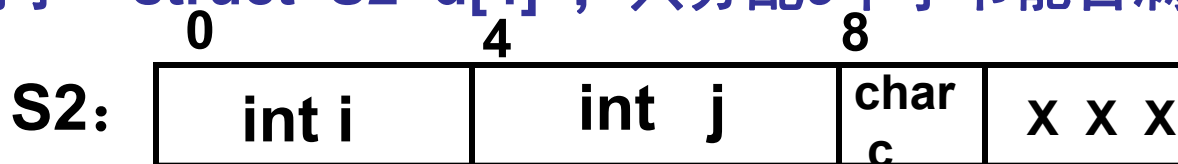


**需要12个字节**



**只需要9个字节**

对于 “struct S2 d[4]”，只分配9个字节能否满足对齐要求？ **不能！**



**也需要12个字节**

# 第二讲小结

---

- 非数值数据的表示

- 逻辑数据用来表示真/假或N位位串，按位运算
- 西文字符：用ASCII码表示
- 汉字：汉字输入码、汉字内码、汉字字模码

- 数据的宽度

- 位、字节、字（不一定等于字长）
- k / K / M / G / T / P / E / Z / Y 有不同的含义

- 数据的存储排列

- 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
  - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么？
- 大端方式：用MSB存放的地址表示数据的地址
- 小端方式：用LSB存放的地址表示数据的地址
- 按边界对齐可减少访存次数

# 编译器处理常量时默认的类型

- C90



如果常量范围 =》 默认的	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{32}-1$	unsigned int
$2^{32} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long

- C99



如果常量范围 =》 默认的	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long



```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int x=-1;
```

```
    unsigned u=2147483648;
```

```
    printf("x = %u = %d\n", x, x);
```

```
    printf("u = %u = %d\n", u, u);
```

```
    if(-2147483648 < 2147483647)
```

```
        printf("-2147483648 < 2147483647 is true\n");
```

```
    else
```

```
        printf("-2147483648 < 2147483647 is false\n");
```

```
    if(-2147483648-1 < 2147483647)
```

```
        printf("-2147483648-1 < 2147483647\n");
```

```
    else if(-2147483648-1 == 2147483647)
```

```
        printf("-2147483648-1 == 2147483647\n");
```

```
    else
```

```
        printf("-2147483648-1 > 2147483647\n");
```

```
}
```

C99的结果大家回去试试。

C90上的运行结果是什么？

```
x = 4294967295 = -1
```

```
u = 2147483648 = -2147483648
```

```
-2147483648 < 2147483647 is false
```

```
-2147483648-1 == 2147483647
```

# 检测系统的字节顺序

- union的存放顺序是所有成员从低地址开始，利用该特性可测试CPU的大/小端方式。

```
#include <stdio.h>
void main()
{
    union NUM
    {
        int a;
        char b;
    } num;
    num.a = 0x12345678;
    if(num.b == 0x12)
        printf("Big Endian\n");
    else
        printf("Little Endian\n");
    printf("num.b = 0x%X\n", num.b);
}
```

Little Endian num.b = 0x78
-------------------------------

请猜测在IA-32上的打印结果。

# C语言中对齐方式的设定

- 在C语言中，结构是一种复合数据类型，其构成元素既可以是基本数据类型（如**int**、**long**、**float**等）的变量，也可以是一些复合数据类型（如数组、结构、联合等）的数据单元。在结构中，编译器为结构的每个成员按其自然边界（**alignment**）分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。
- 为了使**CPU**能够对变量进行快速的访问,变量的起始地址应该具有某些特性,即所谓的”对齐”。比如4字节的**int**型,其起始地址应该位于4字节的边界上,即起始地址能够被4整除。
- 字节对齐的作用不仅是便于**cpu**快速访问，同时合理的利用字节对齐可以有效地节省存储空间。
- 对于**32**位机来说，4字节对齐能够使**cpu**访问速度提高，比如说一个**long**类型的变量，如果跨越了4字节边界存储，那么**cpu**要读取两次，这样效率就低了。在**32**位机中使用1字节或者2字节对齐，反而会使变量访问速度降低。所以要考虑处理器类型，及编译器的类型。在**Visual C**中默认是4字节对齐的，**GNU gcc**也是默认4字节对齐。



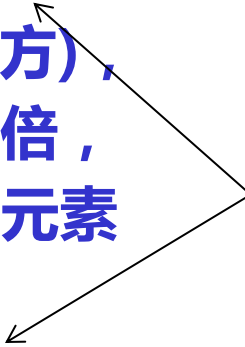
# 对齐方式的设定

## #pragma pack(n)

- 为编译器指定**结构体或类内部的成员变量**的对齐方式。
- 当自然边界（如int型按4字节、short型按2字节、float按4字节）比n大时，按n字节对齐。
- **缺省或#pragma pack()**，按自然边界对齐。

## \_\_attribute\_\_((aligned(m)))

- 按m字节对齐(m必须是2的幂次方)，且其占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐。



为编译器指定一个**结构体或类或联合体或一个单独的变量(对象)**的对齐方式。

## \_\_attribute\_\_((packed))

- 不按边界对齐，称为紧凑方式。

# #include<stdio.h> 对齐方式的设定

```
//typedef unsigned char uint8_t;
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出:

**Struct size is: 1024, aligned on 1024**

**Allocate f1 on address: 0x0**

**Allocate f2 on address: 0x4**

**Allocate f3 on address: 0x5**

**Allocate f4 on address: 0x8**

**Allocate f5 on address: 0xc**

if main

```
46 // typedef
47 struct {
48     uint32_t    f1;
49     uint8_t     f2;
50     uint8_t     f3;
51     uint32_t    f4;
52     uint64_t    f5;
53 } __attribute__((aligned(1024))) ts;
54 int main()
55 {
56     printf("Struct size is: %d, aligned on 1024\n", sizeof(ts));
57     printf("Allocate f1 on address: 0x%x\n", &ts.f1);
58     printf("Allocate f1 on address: 0x%x\n", &ts.f2);
59     printf("Allocate f1 on address: 0x%x\n", &ts.f3);
60     printf("Allocate f1 on address: 0x%x\n", &ts.f4);
61     printf("Allocate f1 on address: 0x%x\n", &ts.f5);
62     /* printf("...")
63     printf("Al
64     printf("Al
65     printf("Al
66     */
67     return 0;
```

输出:

Struct size is: 1024, aligned on 1024

Allocate f1 on address: 0x0

Allocate f2 on address: 0x4

Allocate f3 on address: 0x5

Allocate f4 on address: 0x8

Allocate f5 on address: 0xc

"D:\1010-Chn-organization-2018\hello.exe"

Struct size is: 1024, aligned on 1024

Allocate f1 on address: 0x404090

Allocate f1 on address: 0x404094

Allocate f1 on address: 0x404095

Allocate f1 on address: 0x404098

Allocate f1 on address: 0x40409c

请按任意键继续. . .

```

#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n", sizeof(struct test));
    printf("size=%d\n", sizeof(struct test1));
    printf("size=%d\n", sizeof(struct test2));
}

```

输出结果是什么？

size=15

size=24

size=24

```

#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((aligned(8)))
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n", sizeof(struct test));
    printf("size=%d\n", sizeof(struct test1));
    printf("size=%d\n", sizeof(struct test2));
}

```

```

20 {
21     char x2;
22     int x1;
23     short x3;
24     long long x4;
25 };
26 struct test3
27 {
28     char x2;
29     int x1;
30     short x3;
31     long long x4;
32
33 }__attribute__((aligned(8)));
34
35 void main()
36 {
37     printf("\n s1=%d", sizeof(struct test1));
38     printf("\n s2=%d", sizeof(struct test2));
39     printf("\n s3=%d", sizeof(struct test3));
40 }
41

```

"D:\1010-Chn-organization-2018

s1=15  
s2=24  
s3=24请按任意键继续. . .

size=15  
size=24  
size=24



```
#include <stdio.h>
#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
```

如果设置了pragma pack(1),  
结果又是什么?

```
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
```

```
struct test2
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((aligned(8)));
```

```
size=15
size=15
size=16
```

```
void main()
{
    printf("size=%d\n", sizeof(struct test));
    printf("size=%d\n", sizeof(struct test1));
    printf("size=%d\n", sizeof(struct test2));
}
```

```

#include <stdio.h>
#pragma pack(2)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));

```

如果设置了pragma pack(2),  
结果又是什么？

```

struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};

```

```

struct test2
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((aligned(8)));

```

size=15

size=16

size=16

```

void main()
{
    printf("size=%d\n", sizeof(struct test));
    printf("size=%d\n", sizeof(struct test1));
    printf("size=%d\n", sizeof(struct test2));
}

```

# C表达式类型转换顺序

不同类型的“数据”进行操作时，先转换成相同的数据类型，然后进行操作，规则是由低运算优先级向高优先级转换。

unsigned long long(高优先级)

↑

long long

↑

unsigned

↑

int

↑

(unsigned) char, short  
(低优先级)

```
#include <stdio.h>
void main()
{
    unsigned int a = 1;
    unsigned short b = 1;
    char c = -1;
    int d;

    d = (a > c) ? 1:0;
    printf("%d\n", d);
    d = (b > c) ? 1:0;
    printf("%d\n", d);
}
```

混用不同类型的变量时，会发生类型转换。  
猜测执行结果是什么？

0 (a,c都转换成 unsigned int型，再相减)

1 (b,c都转换成 int型，再相减)

(int型同signed int型)



```

0804841c <main>:
804841c: 55          push    %ebp
804841d: 89 e5       mov     %esp,%ebp
804841f: 83 e4 f0    and     $0xffffffff0,%esp
8048422: 83 c0 20    sub     $0x20,%esp
8048425: c7 01       unsigned int a=1; → movl    $0x1,0x1c(%esp)
804842c: 0f          unsigned short b=1; → movw    $0x1,0x1a(%esp)
804842d: 6b 01       char c=-1; → movb    $0xff,0x19(%esp)
8048434: c6 44 24 1  d=(a>c)?1:0 → movsbl  0x19(%esp),%eax
8048439: 0f be 44 2  cmp     0x1c(%esp),%eax
804843e: 3b 44 24    setb    %al
8048442: 0f 92 c0    movzbl  %al,%eax
8048445: 0f b6 c0    mov     %eax,0x14(%esp)
8048448: 89 44 24 14 mov     0x14(%esp),%eax
804844c: 8b 44 24 14 mov     %eax,0x4(%esp)
8048450: 89 44 24 04 movl    $0x8048520,(%esp)
8048454: c7 04 24 20 85 04 08 call    8048300 <printf@plt>
804845b: e8 a0 fe ff ff
8048460: 0f b7 54 24 1a
8048465: 0f be 44    d=(b>c)?1:0 → movzwl  0x1a(%esp),%edx
804846a: 39 c2       movsbl  0x19(%esp),%eax
804846c: 0f 9f c0    cmp     %eax,%edx
804846f: 0f b6 c0    setg    %al
8048472: 89 44 24 14 movzbl  %al,%eax
8048476: 8b 44 24 14 mov     %eax,0x14(%esp)
804847a: 89 44 24 14 mov     0x14(%esp),%eax
804847e: 89 44 24 04 mov     %eax,0x4(%esp)
8048485: c7 04 24 20 85 04 08 movl    $0x8048520,(%esp)
8048485: e8 76 fe ff ff call    8048300 <printf@plt>
804848a: c9         leave
804848b: c3         ret

```

# 数据的表示

- 分以下三个部分介绍 (红色标注的为今天下面要讲的)
  - 第一讲：数值数据的表示
    - 定点数的编码表示、整数的表示、无符号整数、带符号整数、浮点数的表示
    - C语言程序的整数类型和浮点数类型
  - 第二讲：非数值数据的表示、数据的存储(今天)
    - 逻辑值、西文字符、汉字字符
    - 数据宽度单位、大端/小端、对齐存放
  - 第三讲：数据的校验码(今天)
    - 奇偶校验

# 数据的检/纠错 (Error Detect/Correct)

- 为什么要进行数据的错误检测与校正？

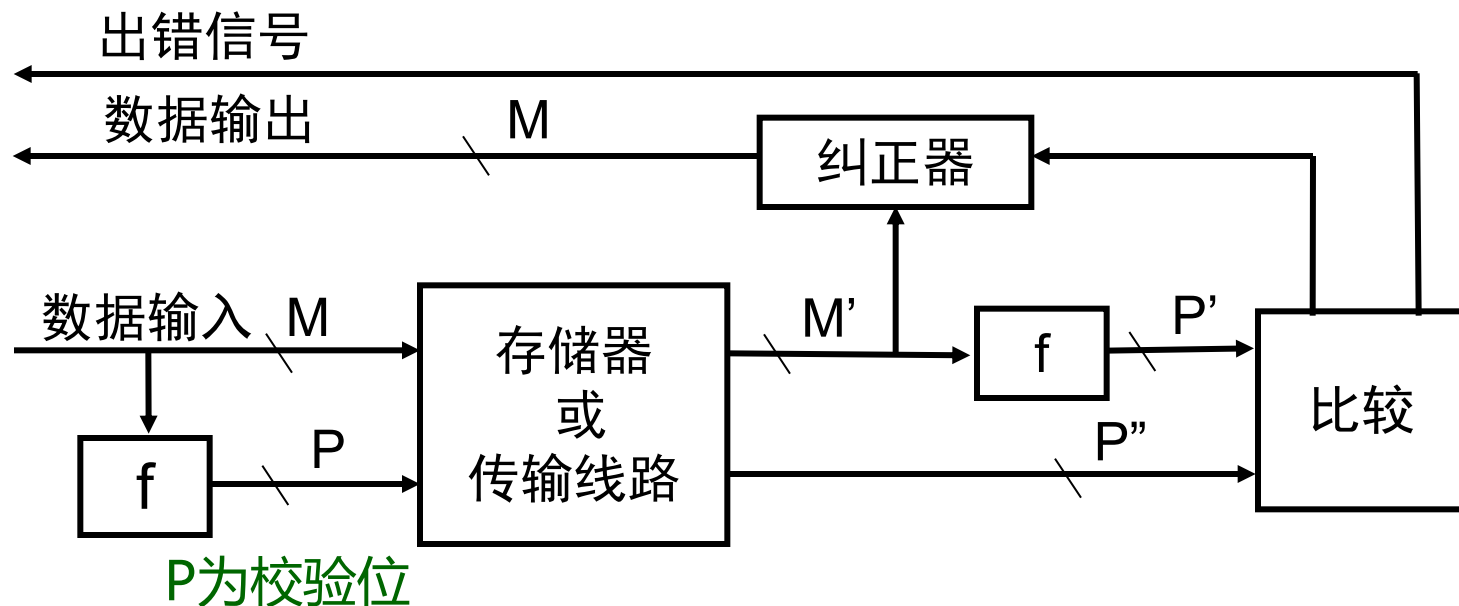
存取和传送时，由于元器件故障或噪音干扰等原因会出现差错。措施：

(1) 从计算机硬件本身的可靠性入手，在电路、电源、布线等各方面采取必要的措施，提高计算机的抗干扰能力；

(2) 采取相应的数据检错和校正措施，自动地发现并纠正错误。

- 如何进行错误检测与校正？

— 大多采用“冗余校验”思想，即除原数据信息外，还增加若干位编码，这些新增的代码被称为校验位。



# 数据的检/纠错 (Error Detect/Correct)

- 为什么要进行数据的错误检测与校正？

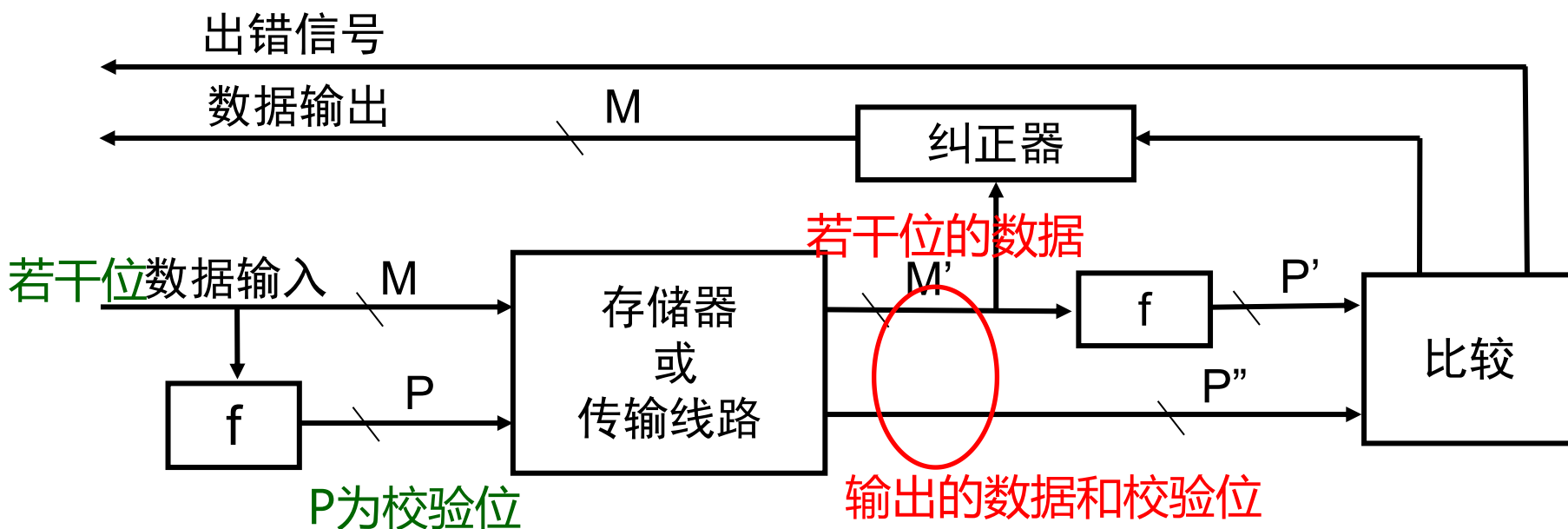
存取和传送时，由于元器件故障或噪音干扰等原因会出现差错。措施：

(1) 从计算机硬件本身的可靠性入手，在电路、电源、布线等各方面采取必要的措施，提高计算机的抗干扰能力；

(2) 采取相应的数据检错和校正措施，自动地发现并纠正错误。

- 如何进行错误检测与校正？

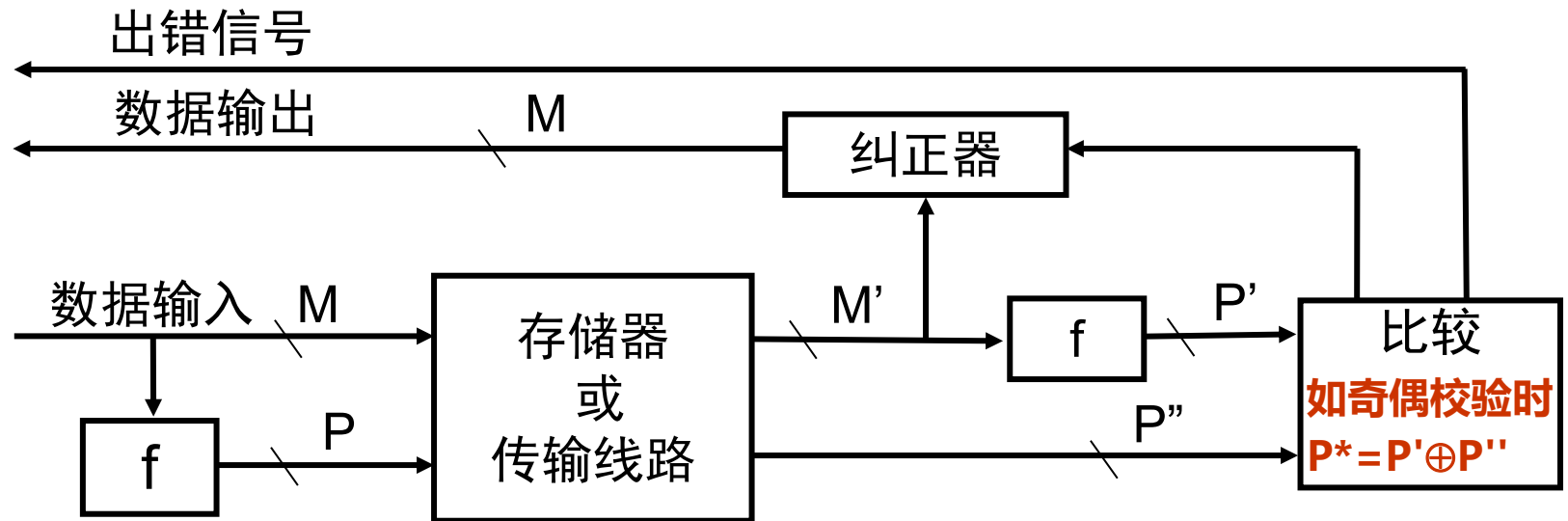
– 大多采用“冗余校验”思想，即除原数据信息外，还增加若干位编码，这些新增的代码被称为校验位。



# 数据的检/纠错

比较的结果为以下三种情况之一：

- ① 没有检测到错误，得到的数据位直接传送出去。
- ② 检测到差错，并可以纠错。数据位和比较结果一起送入纠错器，将正确数据位传送出去。
- ③ 检测到错误，但无法确认哪位出错，因而不能进行纠错处理，此时，报告出错情况。



BACK

# 码字和码距

- 什么叫码距？
  - “码字”指数据位和校验位按某种规律排列得到的代码
    - 或称为：由若干位代码组成的一个字叫“码字”
  - 两个码字中具有不同代码的位的个数叫这两个码字间的“距离”
  - 码制中任意码字间最小距离为“码距”，它就是这个码制的距离
- 问题：“8421”码的码距是几？  
2 (0010) 和 3 (0011) 间距离为1，“8421”码制的码距为1。
- 码距与检错、纠错能力的关系（当 $d \leq 4$ ）
  - ① 若码距 $d$ 为奇数，则能发现 $d-1$ 位错，或能纠正 $(d-1)/2$ 位错。
  - ② 若码距 $d$ 为偶数，则能发现 $d/2$ 位错，并能纠正 $(d/2-1)$ 位错。
- 常用的数据校验码有：  
奇偶校验码、海明校验码、循环冗余校验码。

# 奇偶校验码

**基本思想：**增加一位奇（偶）校验位并一起存储或传送，根据终部件得到的相应数据和校验位，再求出新校验位，最后根据新校验位确定是否发生了错误。

**实现原理：**假设数据 $B=b_{n-1}b_{n-2}\dots b_1b_0$ 从源部件传送至终部件。在终部件接收到的数据为 $B'=b_{n-1}'b_{n-2}'\dots b_1'b_0'$ 。

**第一步：**在源部件求出奇（偶）校验位 $P$ 。

若采用奇校验，则 $P=b_{n-1}\oplus b_{n-2}\oplus \dots\oplus b_1\oplus b_0\oplus 1$ （这里多 $\oplus$ 了1）

若采用偶校验，则 $P=b_{n-1}\oplus b_{n-2}\oplus \dots\oplus b_1\oplus b_0$ （这里没有多）

**第二步：**在终部件求出奇（偶）校验位 $P'$ 。

若采用奇校验，则 $P'=b_{n-1}'\oplus b_{n-2}'\oplus \dots\oplus b_1'\oplus b_0'\oplus 1$ （这里多 $\oplus$ 了1）

若采用偶校验，则 $P'=b_{n-1}'\oplus b_{n-2}'\oplus \dots\oplus b_1'\oplus b_0'$

**第三步：**在接收到数据后计算最终的校验位 $P^*$ ，根据其值判断有无奇偶错

假定 $P$ 在终部件接受到的值为 $P''$ ，则 $P^*=P'\oplus P''$

① 若 $P^*=1$ ，则表示终部件接受的数据有奇数位错。

② 若 $P^*=0$ ，则表示终部件接受的数据正确或有偶数个错。



# 奇校验法举例

- 传输各位为：0100 0101=45H，以奇校验法校验

## 第一步：在源部件求出奇校验位P

- 若采用奇校验，则 $P=0\oplus 1\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1\oplus 1=0$
- 发送数据“0100 0101”及校验位  $P=0$

## 第二步：在终部件求出奇（偶）校验位P'。

- 接收端收到 $P''=0$ ,收到数据如下，在终部件求出奇（偶）校验位P'。

1) 收到数据为:00000101,  $P'_1=0\oplus 0\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1\oplus 1=1$

2) 收到数据为:01000001,  $P'_2=0\oplus 1\oplus 0\oplus 0\oplus 0\oplus 0\oplus 0\oplus 1\oplus 1=1$

## 第三步：在接收到数据后计算最终的校验位P\*，来判断有无奇偶错

1)  $P^*_1 = P'_1 \oplus P'' = 1\oplus 0 = 0$ ,表示终部件接受的数据有奇数位错

2)  $P^*_2 = P'_2 \oplus P'' = 1\oplus 0 = 0$ ,表示终部件接受的数据有奇数位错

以上错误不知错在哪个位上



# 海明校验码(自学)

---

- 由Richard Hamming于1950年提出，目前还被广泛使用。
- 主要用于存储器中数据存取校验。
- 基本思想：奇偶校验码对整个数据编码生成一位校验位。因此这种校验码检错能力差，并且没有纠错能力。如果将整个数据按某种规律分成若干组，对每组进行相应的奇偶检测，就能提供多位检错信息，从而对错误位置进行定位，并将其纠正。
  - 海明校验码实质上就是一种多重奇偶校验码。
- 处理过程：
  - 最终比较时按位进行异或，以确定是否有差错。
  - 这种异或操作所得到的结果称为故障字（syndrome word）。显然，校验码和故障字的位数是相同。
    - 每一组一个校验位，校验码位数等于组数！
    - 每一组内采用一位奇偶校验！

# 校验码位数的确定

- 假定数据位数为 $n$ ，校验码为 $k$ 位，则故障字位数也为 $k$ 位。 $k$ 位故障字所能表示的状态最多是 $2^k$ ，每种状态可用来说明一种出错情况。
- 若只有一位错，则结果可能是：
  - 数据中某一位错 ( $n$ 种可能)
  - 校验码中有一位错 ( $k$ 种可能)
  - 无错 (1种可能)

1+n+k种情况

假定最多有一位错，则 $n$ 和 $k$ 必须满足下列关系：

$$2^k \geq 1+n+k, \quad \text{即: } 2^k - 1 \geq n+k$$

- 有效数据位数和校验码位数间的关系
- 当数据有8位时，校验码和故障字都应有4位。

说明：4位故障字最多可表示16种状态，而单个位出错情况最多只有12种可能（8个数据位和4个校验位），再加上无错的情况，一共有13种。所以，用16种状态表示13种情况应是足够了。

# 有效数据位数和校验码位数间的关系

n和k的关系： $2^K - 1 \geq n + k$ （n和k分别为数据位数和校验位数）

	单纠错		单纠错/双检错	
数据位数	校验位数	增加率	校验位数	增加率
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

# 海明码的分组(自学)

- **基本思想:**  $n$ 位数据位和 $k$ 位校验位按某种方式排列为一个  $(n+k)$  位的码字, 将该码字中每个出错位的位置与故障字的数值建立关系, 通过故障字的值确定该码字中哪一位发生了错误, 并将其取反来纠正。

根据上述基本思想, 按以下规则来解释各故障字的值。

**规则1:** 若故障字每位全部是0, 则表示没有发生错误。

**规则2:** 若故障字中有且仅有一位为1, 则表示校验位中有一位出错, 因而不需纠正。

**规则3:** 若故障字中多位为1, 则表示有一个数据位出错, 其在码字中的出错位置由故障字的数值来确定。纠正时只要将出错位取反即可。

# 海明码的分组(自学)

以8位数据进行单个位检错和纠错为例说明。

假定8位数据 $M = M_8M_7M_6M_5M_4M_3M_2M_1$ ，4位校验位 $P = P_4P_3P_2P_1$ 。根据规则将 $M$ 和 $P$ 按一定的规律排到一个12位码字中。

据规则1，故障字为0000时，表示无错。

据规则2，故障字中有且仅有一位为1时，表示校验位中有一位出错。此时，故障字只可能是0001、0010、0100、1000，将这四种状态分别代表校验位中 $P_1$ 、 $P_2$ 、 $P_3$ 、 $P_4$ 位发生错误，因此，它们分别位于码字的第1、2、4、8位。

据规则3，将其他多位为1的故障字依次表示，是逻辑顺序，物理上  
况。因此，数据位 $M_1 \sim M_8$ 分别位于码字的第0 M和P是分开的！

0110(6)、0111(7)、1001(9)、1010(10)、1011(11)、1100(12)位。即  
码字的排列为： $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$

这样，得到故障字 $S = S_4S_3S_2S_1$ 的各个状态和出错情况的对应关系表，可根据这种对应关系对整个数据进行分组。

# 海明校验码分组情况(自学)

码字:  $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$  故障字 $S_4S_3S_2S_1$ 每一位的值反映所在组的奇偶性

序号 含义 分组	1	2	3	4	5	6	7	8	9	10	11	12	故障字	正 确	出错位											
	$P_1$	$P_2$	$M_1$	$P_3$	$M_2$	$M_3$	$M_4$	$P_4$	$M_5$	$M_6$	$M_7$	$M_8$														
															1	2	3	4	5	6	7	8	9	10	11	12
第4组								✓	✓	✓	✓	✓	$S_4$	0	0	0	0	0	0	0	1	1	1	1	1	1
第3组				✓	✓	✓	✓					✓	$S_3$	0	0	0	1	1	1	1	0	0	0	0	1	
第2组		✓	✓			✓	✓			✓	✓		$S_2$	0	0	1	1	0	0	1	1	0	0	1	1	0
第1组	✓		✓		✓		✓	✓		✓			$S_1$	0	1	0	1	0	1	0	1	0	1	0	1	0

数据位或校验位出错一定会影响所在组的奇偶性。

例: 若 $M_2$ 出错, 则故障字为0101, 因而会改变 $S_3$ 和 $S_1$ 所在分组的奇偶性。  
故 $M_2$ 同时被分到第3( $S_3$ )组和第1( $S_1$ )组。

问题: 若 $P_1$ 出错, 则如何? 若 $M_8$ 出错, 则如何?

SKIP

$P_1 \sim 0001$ , 分在第1组;  $M_8 \sim 1100$ , 分在第4组和第3组

# 校验位的生成和检错、纠错(自学)

- 分组完成后，就可对每组采用相应的奇（偶）校验，以得到相应的一个校验位。
- 假定采用偶校验（取校验位 $P_i$ ，使对应组中有偶数个1），则得到校验位与数据位之间存在如下关系：

$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8$$

$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8$$

$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7$$

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7$$

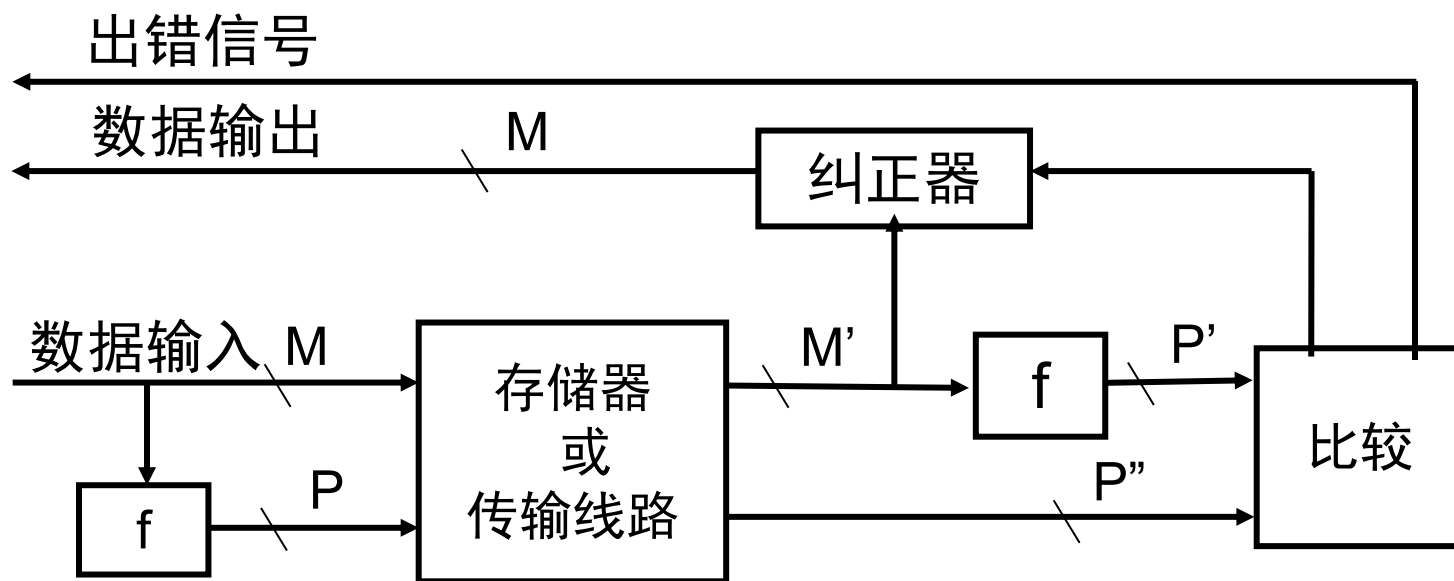
码字： $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$

# 海明校验过程(自学)

- 根据公式求出每一组对应的校验位 $P_i$  ( $i=1,2,3,4$ )
- 数据 $M$ 和校验位 $P$ 一起被存储，根据读出数据 $M'$ 得新校验位 $P'$
- 读出校验位 $P''$ 与新校验位 $P'$  按位进行异或操作，得故障字

$$S = S_4 S_3 S_2 S_1$$

- 根据 $S$ 的值确定：无错、仅校验位错、某个数据位错





# 海明码举例(自学)

假定一个8位数据M为： $M_8M_7M_6M_5M_4M_3M_2M_1=01101010$ ，根据上述公式求出相应的校验位为：

$$P_4 = M_5 \oplus M_6 \oplus M_7 \oplus M_8 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

$$P_3 = M_2 \oplus M_3 \oplus M_4 \oplus M_8 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_2 = M_1 \oplus M_3 \oplus M_4 \oplus M_6 \oplus M_7 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5 \oplus M_7 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

假定12位码字 ( $M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$ ) 读出后为：

(1) 数据位 $M'=M=01101010$ ，校验位 $P''=P=0011$

(2) 数据位 $M'=01111010$ ，校验位 $P''=P=0011$

(3) 数据位 $M'=M=01101010$ ，校验位 $P''=1011$

要求分别考察每种情况的故障字。

(1) 数据位 $M'=M=01101010$ ，校验位 $P''=P=0011$ ，即无错。

因为 $M'=M$ ，所以 $P'=P$ ，因此  $S = P'' \oplus P' = P \oplus P = 0000$ 。

## 海明码举例(自学)

(2) 数据位M'= 011**1**1010, 校验位P''=P=0011, 即M<sub>5</sub>错。

对M'生成新的校验位P'为:

$$P_4' = M_5' \oplus M_6' \oplus M_7' \oplus M_8' = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$P_3' = M_2' \oplus M_3' \oplus M_4' \oplus M_8' = 1 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_2' = M_1' \oplus M_3' \oplus M_4' \oplus M_6' \oplus M_7' = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$P_1' = M_1' \oplus M_2' \oplus M_4' \oplus M_5' \oplus M_7' = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 = 0$$

故障字S为:

$$S_4 = P_4' \oplus P_4'' = 1 \oplus 0 = 1$$

$$S_3 = P_3' \oplus P_3'' = 0 \oplus 0 = 0$$

$$S_2 = P_2' \oplus P_2'' = 1 \oplus 1 = 0$$

$$S_1 = P_1' \oplus P_1'' = 0 \oplus 1 = 1$$

因此, 错误位是第9位, 排列的是数据位M<sub>5</sub>, 所以检错正确, 纠错时, 只要将码字的第9位(M<sub>5</sub>)取反即可。

## 海明码举例(自学)

---

(3) 数据位 $M'=M=01101010$ , 校验位 $P''=1011$ ,

即: 校验码第4位( $P_4$ )错。

因为 $M'=M$ , 所以 $P'=P$ , 因此故障位 $S$ 为:

$$S_4 = P_4' \oplus P_4'' = 0 \oplus 1 = 1$$

$$S_3 = P_3' \oplus P_3'' = 0 \oplus 0 = 0$$

$$S_2 = P_2' \oplus P_2'' = 1 \oplus 1 = 0$$

$$S_1 = P_1' \oplus P_1'' = 1 \oplus 1 = 0$$

错误位是第1000位(即第8位), 这位上排列的是校验位 $P_4$ , 所以检错时发现数据正确, 不需纠错。

# 单纠错和双检错码(自学)

- 单纠错码 (SEC)

- 问题：上述( $n=8/k=4$ )海明码的码距是几？
- 码距 $d=3$ 。因为，若有一位出错，则因该位至少要参与两组校验位的生成，因而至少引起两个校验位的不同。两个校验位加一个数据位等于3。

例如，若 $M_1$ 出错，则故障字为0011，即 $P_2$ 和 $P_1$ 两个校验位发生改变，12位码字中有三位（ $M_1$ 、 $P_2$ 和 $P_1$ ）不同。

- 根据码距与检错、纠错能力的关系，知：这种码制能发现两位错，或对单个位出错进行定位和纠错。这种码称为单纠错码 (SEC)。

# 单纠错和双检错码(自学)

- 单纠错和双检错码 (SEC-DED)

- 具有发现两位错和纠正一位错的能力，称为单纠错和双检错码 (SEC-DED)。

- 若要成为SEC-DED，则码距需扩大到 $d=4$ 。为此，还需增加一位校验位 $P_5$ ，将 $P_5$ 排列在码字的最前面，即：

$P_5M_8M_7M_6M_5P_4M_4M_3M_2P_3M_1P_2P_1$ ，并使得数据中的每一位都参与三个校验位的生成。从表中可看出除了 $M_4$ 和 $M_7$ 外，其余位都只参与了两个校验位的生成。因此 $P_5$ 按下式求值：

$$P_5 = M_1 \oplus M_2 \oplus M_3 \oplus M_5 \oplus M_6 \oplus M_8$$

当任意一个数据位发生错误时，必将引起三个校验位发生变化，所以码距为4。

# 循环冗余码(自学)

---

循环冗余校验码 (Cyclic Redundancy Check)，简称CRC码

- 具有很强的检错、纠错能力。
- 用于大批量数据存储和传送(如：外存和通信)中的数据校验。

为什么大批量数据不用奇偶校验？

在每个字符后增加一位校验位会增加大量的额外开销；尤其在网络通信中，对传输的二进制比特流没有必要再分解成一个个字符，因而无法采用奇偶校验码。

- 通过某种数学运算来建立数据和校验位之间的约定关系。

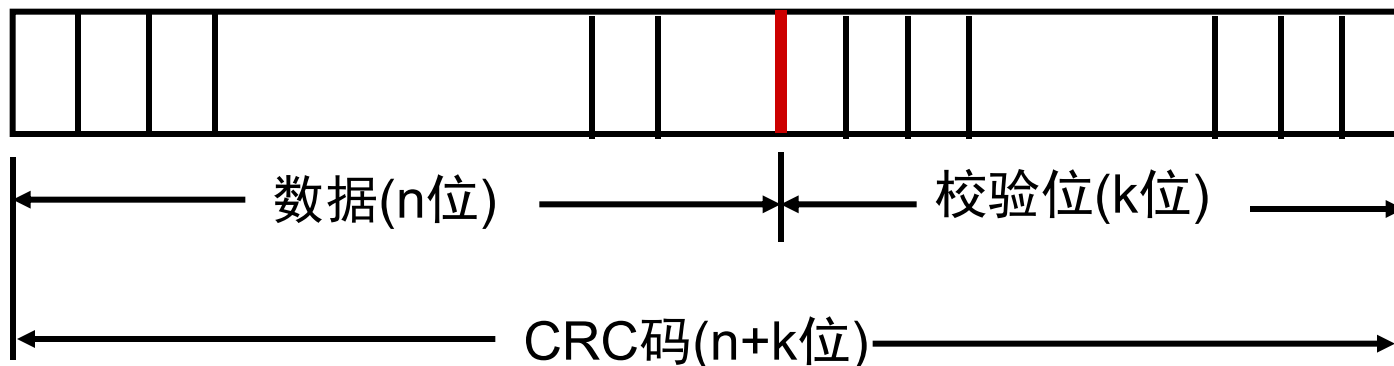
奇偶校验码和海明校验码都是以奇偶检测为手段的。

网络或通信课程中会学到。

# CRC码的检错方法(自学)

基本思想:

- 数据信息 $M(x)$ 为一个 $n$ 位的二进制数据, 将 $M(x)$ 左移 $k$ 位后, 用一个约定的“生成多项式” $G(x)$ 相除,  $G(x)$ 是一个 $k+1$ 位的二进制数, 相除后得到的 $k$ 位余数就是校验位。校验位拼接到 $M(x)$ 后, 形成一个 $n+k$ 位的代码, 称该代码为循环冗余校验 (CRC) 码, 也称  $(n+k, n)$  码。
- 一个CRC码一定能被生成多项式整除, 当数据和校验位一起送到接受端后, 只要将接受到的数据和校验位用同样的生成多项式相除, 如果正好除尽, 表明没有发生错误; 若除不尽, 则表明某些数据位发生了错误。通常要求重传一次。



# 循环冗余码举例(自学)

---

校验位的生成：用一个例子来说明校验位的生成过程。

– 假设要传送的数据信息为：100011，即报文多项式为：

$$M(x) = x^5 + x + 1。 \text{数据信息位数} n=6。$$

– 若约定的生成多项式为： $G(x) = x^3 + 1$ ，则生成多项式位数为4位，所以校验位位数 $k=3$ ，除数为1001。

– 生成校验位时，用 $x^3 \cdot M(x)$ 去除以 $G(x)$ ，即：  
 $100011000 \div 1001。$

– 相除时采用“模2运算”的多项式除法。



# 循环冗余码举例(自学)

$$X^3 \cdot M(x) \div G(x) = (x^8 + x^4 + x^3) \div (x^3 + 1)$$

$$\begin{array}{r} 100111 \\ 1001 \overline{) 100011000} \\ \underline{1001} \phantom{000} \\ 0011 \phantom{000} \\ \underline{0000} \phantom{000} \\ 0111 \phantom{000} \\ \underline{0000} \phantom{000} \\ 1110 \phantom{000} \\ \underline{1001} \phantom{000} \\ 1110 \phantom{000} \\ \underline{1001} \phantom{000} \\ 1110 \phantom{000} \\ \underline{1001} \phantom{000} \\ 111 \end{array}$$

余数

(模2运算不考虑加法进位和减法借位，上商的原则是当部分余数首位是1时商取1，反之商取0。然后按模2相减原则求得最高位后面几位的余数。这样当被除数逐步除完时，最后的余数位数比除数少一位。这样得到的余数就是校验位，此例中最终的余数有3位。)

校验位为111，CRC码为100011 111。如果要校验CRC码，可将CRC码用同一个多项式相除，若余数为0，则说明无错；否则说明有错。例如，若在接收方的CRC码也为100011 111时，用同一个多项式相除后余数为0。若接收方CRC码不为100011 111时，余数则不为0。

# 第二章小结

---

- 非数值数据的表示
  - 逻辑数据用来表示真/假或N位位串，按位运算
  - 西文字符：用ASCII码表示
  - 汉字：汉字输入码、汉字内码、汉字字模码
- 数据的宽度
  - 位、字节、字（不一定等于字长），k/K/M/G/...有不同的含义
- 数据的存储排列
  - 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
    - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么？
  - 大端方式：用MSB存放的地址表示数据的地址
  - 小端方式：用LSB存放的地址表示数据的地址
  - 按边界对齐可减少访存次数
- 数据的纠错和检错
  - 奇偶校验：适应于一字节长数据的校验，如内存

## 附录： Decimal / Binary (十 / 二进制数)

---

- ◆ The decimal number 5836.47 in powers of 10:

$$5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 \\ + 4 \times 10^{-1} + 7 \times 10^{-2}$$

- ◆ The binary number 11001 in powers of 2 :

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = 16 + 8 + 0 + 0 + 1 = 25$$

- ◆ 用一个下标表示数的基 (radix / base)

$$11001_2 = 25_{10}$$

# 附录： Octal / Hexadecimal ( 八 / 十六进制数)

$$\begin{array}{cccccccccccc} 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} = 2000_{10}$$

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

$$2^3=8$$

03720

Octal - base 8

000 - 0

001 - 1

010 - 2

011 - 3

100 - 4

101 - 5

110 - 6

111 - 7

$$2^4=16$$

0x7d0

Hexadecimal - base 16

0000 - 0    1000 - 8

0001 - 1    1001 - 9

0010 - 2    1010 - a

0011 - 3    1011 - b

0100 - 4    1100 - c

0101 - 5    1101 - d

0110 - 6    1110 - e

0111 - 7    1111 - f

计算机用二进制表示所有信息！

为什么要引入 8 / 16进制？

8 / 16进制是二进制的简便表示。便于阅读和书写！

它们之间对应简单，转换容易。

在机器内部用二进制，在屏幕或其他外部设备上表示时，转换为8/16进制数，可缩短长度

# 附录： Conversions of numbers

---

## (1) R进制数 => 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 => 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”

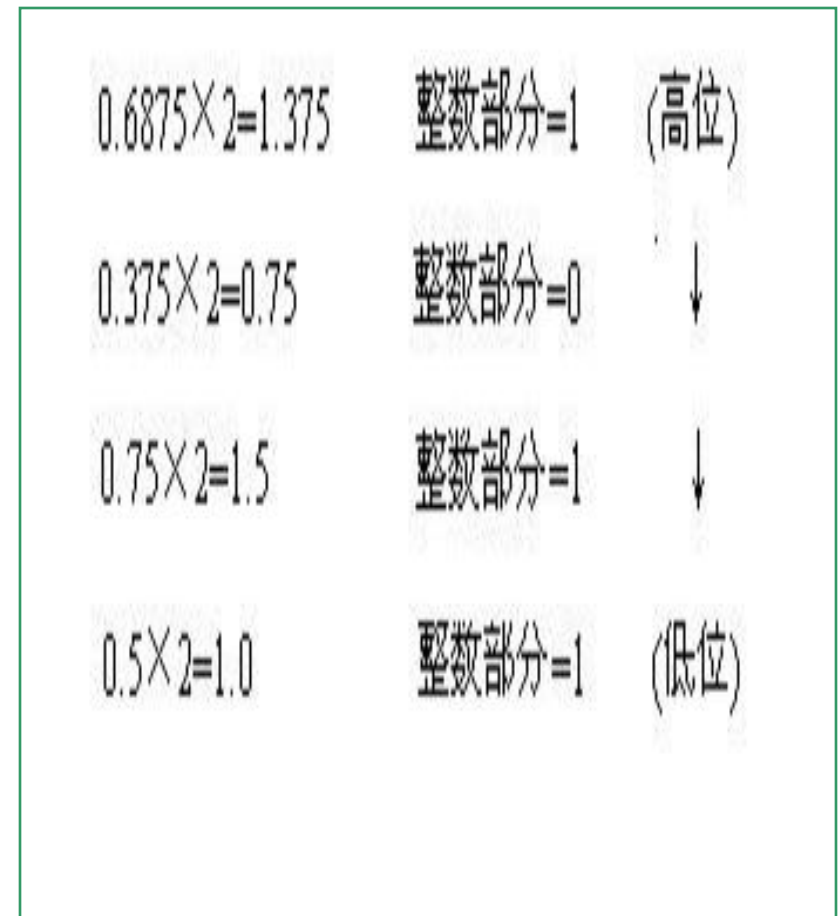
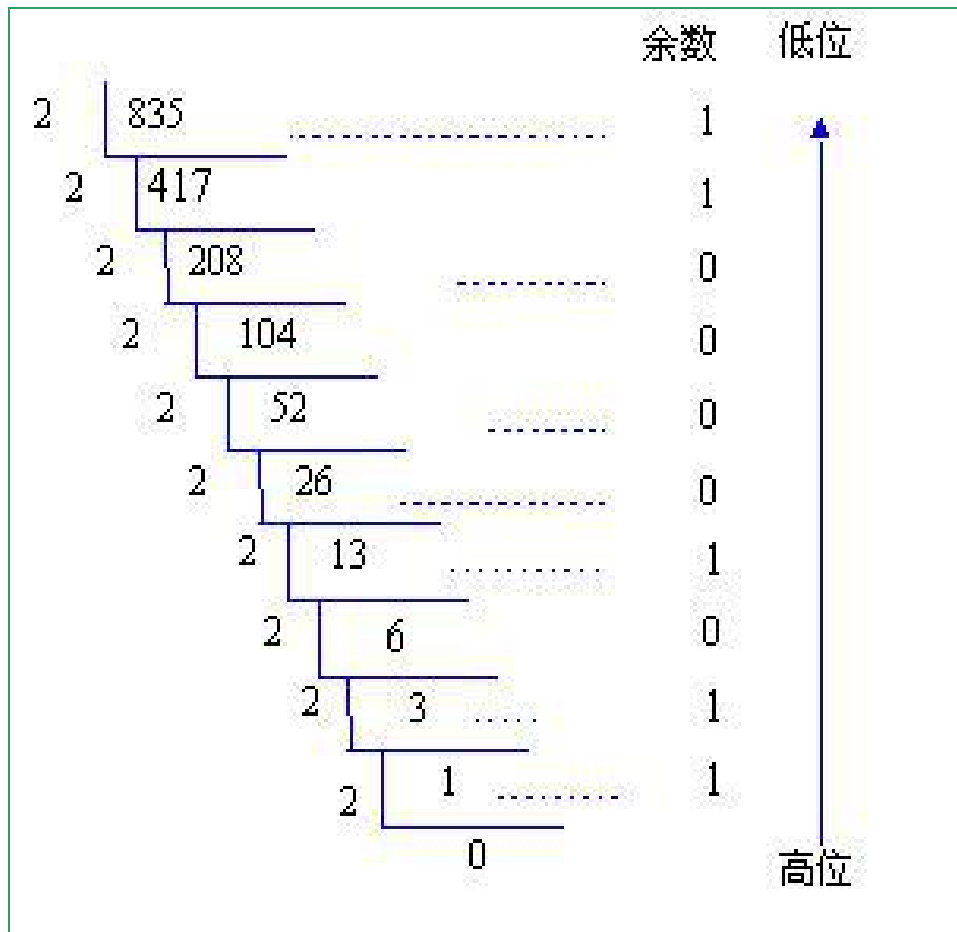
② 小数(fractional part)----“乘基取整，上左下右”

# 附录： Decimal to Binary Conversions

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”

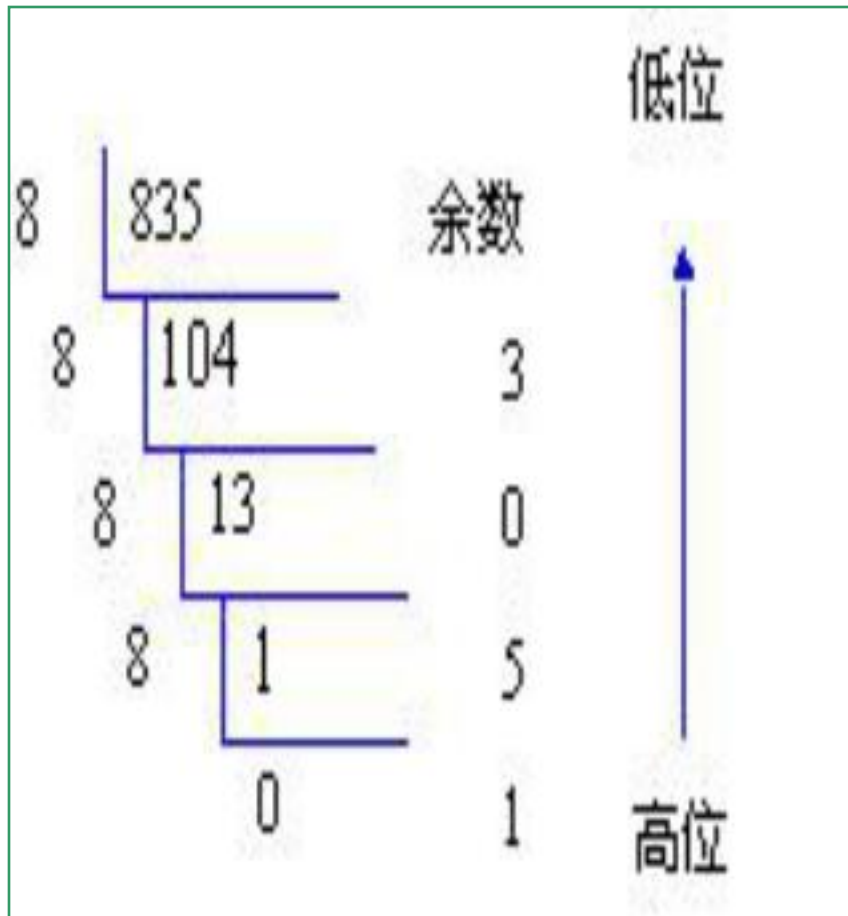


# 附录： Decimal to Binary Conversions

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)

# 附录： Conversions of numbers

---

## (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

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



# 本章作业

---

- **2 (1) 、 2 (7) 、**
- **7: (R1),(R2)的值改为 (R1)=0000 017AH,  
(R2)=FFFF F895H**
- **9**
- **10**
- **13: 变量的值改为6144**
- **14: 变量的值改为-6144**
- **17: x, y, i的值改为: x= -10.125, y=12, i= -125**
- **18: 采用偶校验, 接收方收到的校验位改为1010**