

第二章 数据的机器级表示与处理

2019.4

王多强

群名称：ICS2019

群 号：1015148119



群名称：ICS2019

群 号：1015148119

数据是计算机处理的对象

- 现实世界中，有数值、文字、图、声音、视频及各种模拟信息，它们被称为**感觉媒体信息**；
- 计算机系统中，从机器指令的角度看，数据只有**整数**、**浮点数**和**位串**几类简单的基本数据类型，而且所有信息都是用**二进制**表示的。
- 对现实世界中的感觉媒体信息进行**采样**、**编码**，最终转换成计算机内部以二进制编码形式表示的数据的过程称为**数字化编码**。

编码：就是用少量简单的基本符号，对大量复杂多样的信息进行一定规律的组合。

信息编码的两大要素： **基本符号**和**组合规则**。

如，电报码中用4位十进制数字的组合表示汉字

中：0022、国：0948

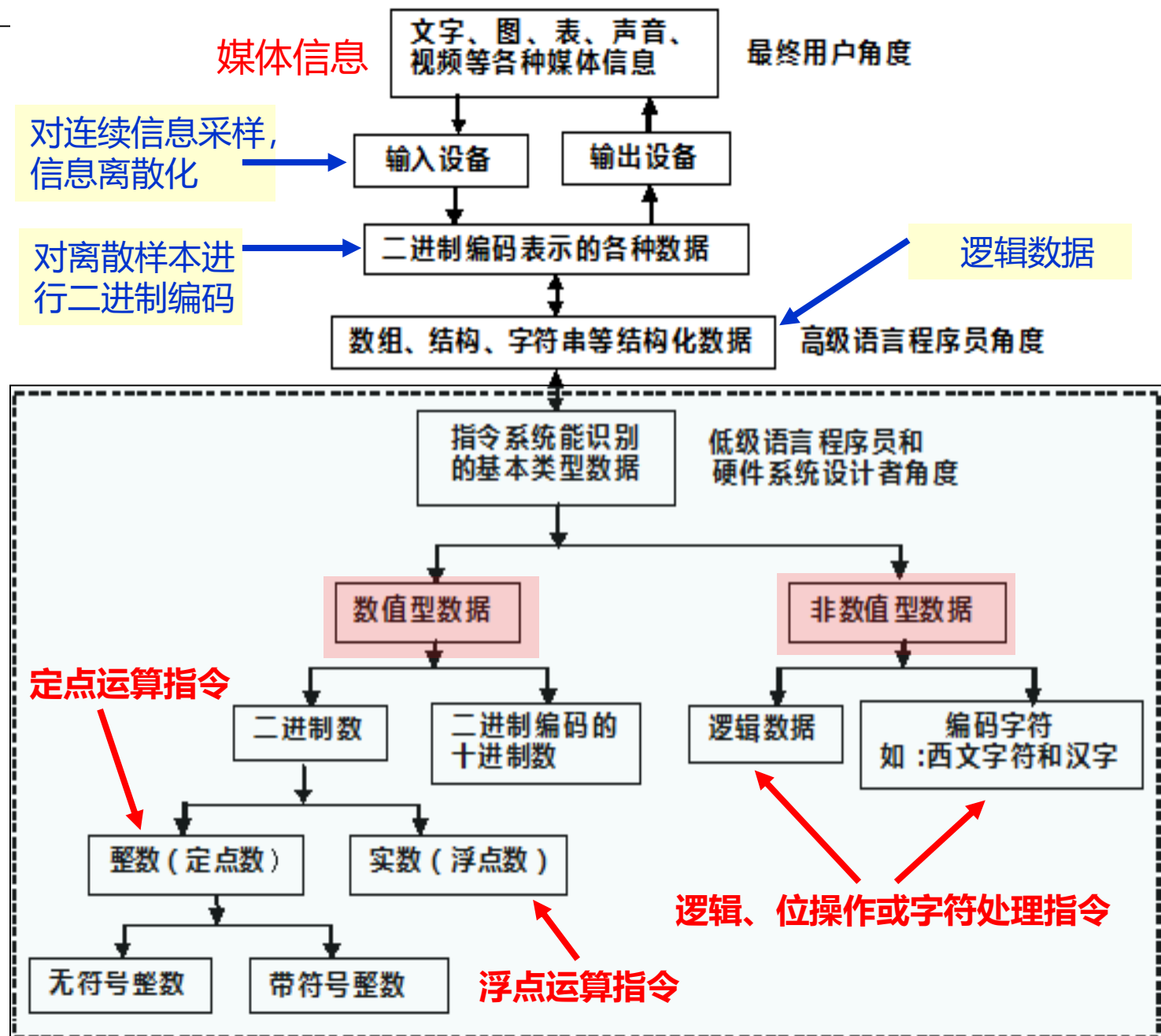
汉语拼音是用26个拼音字母的组合表示汉字：

中：zhong、国：guo

数字化编码过程：就是对感觉媒体信息进行采样，将现实世界中的连续信息转换为计算机中的离散的“样本”信息，然后对样本信息用“0”和“1”进行数字化编码。

- ◆ 现实世界中的各种**媒体信息**被转换成二进制编码的**数字化信息**后，在计算机内部进行存储、运算和传送。
- ◆ 计算机内部，所有的信息都是**二进制编码数据**。

计算机外部信息与内部数据的转换



◆ 指令所处理的数据分为两大类型：数值数据和非数值数据

数值数据：指用来表示数量多少的数据。

- 可比较大小
- 分为整数和实数；

整数又分为无符号整数和带符号整数。

非数值数据：一般指字符数据或逻辑数据。

- 表示为经过编码的位串
- 位串不表示数量，没大小之分。

高级语言程序转换为机器语言程序后，每条机器指令的操作数就只有**4种基本数据类型**：**无符号定点整数、带符号定点整数、浮点数、非数值型数据**（位串）。

本章介绍计算机内部各种数据的编码表示及其运算方法，了解高级语言程序中的各种类型变量对应的机器表示形式及其机器级运算相关的问题。

- **分以下三个部分介绍**

- 第一讲：数值数据的表示
- 第二讲：非数值数据的表示、数据的存储
- 第三讲：数据的运算

- 解释C语言中的运算在底层机器级的实现

2.1 数值数据的表示

本部分介绍：

- **定点数的编码表示：**无符号整数、带符号整数
- **浮点数的编码表示：**IEEE754标准
- **C语言程序的整数类型和浮点数类型的机器级表示**

• 数值数据表示的三要素：

- 进位计数制
- 定/浮点表示
- 编码规则

◆ 要确定一个数值数据的值必须先确定这三个要素

例如，问01011001的值是多少？

89？

一百零一万又一千零一？

答案是：不知道！

1、进位计数制

■ 常用的进位计数制有：

- 十进制数：逢十进一
- 二进制数：逢二进一
- 八进制数：逢八进一
- 十六进制数：逢十六进一

■ R进制数：

- 逢R进一
- R称为“基数”

各种进制的数之间的相互转换

(1) R进制数向十进制数的转换：**按权展开**

$$(xyz.ab)_R = (x \times R^2 + y \times R^1 + z \times R^0 + a \times R^{-1} + b \times R^{-2})_R$$

如：

■ 二进制数转换成十进制数：基数为2

$$\begin{aligned}(10101.01)_2 &= (1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2})_{10} \\ &= (21.25)_{10}\end{aligned}$$

■ 十六进制数转换成十进制数：基数为16

$$\begin{aligned}(3A.C)_{16} &= (3 \times 16^1 + 10 \times 16^0 + 12 \times 16^{-1})_{10} \\ &= (58.75)_{10}\end{aligned}$$

(2) 十进制数向R进制数的转换:

分**整数部分**和**小数部分**分别转换，然后合并

➤ **整数部分**: 除基取余

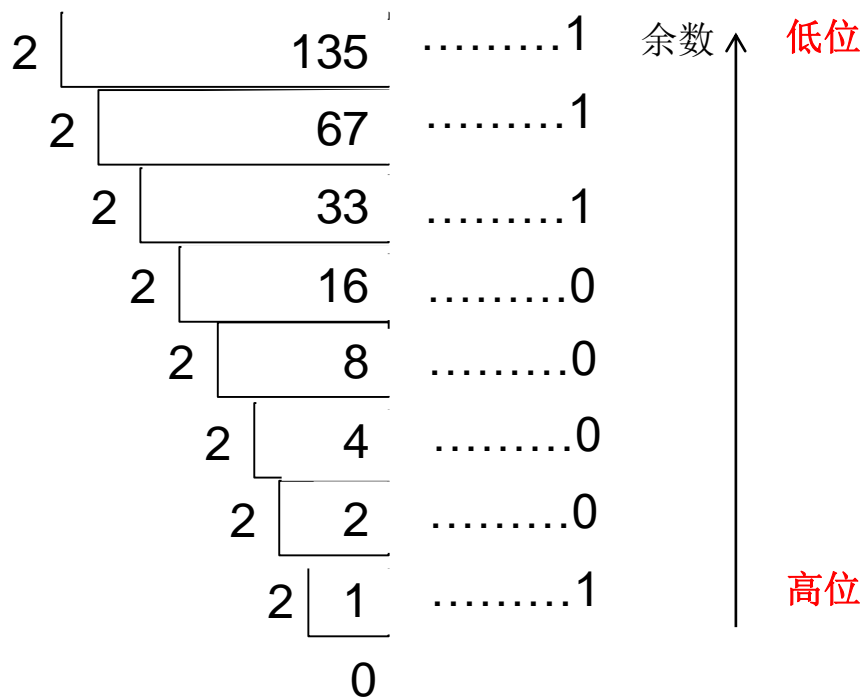
➤ **小数部分**: 乘基取整

◆ 整数部分的转换

口诀：除基取余、上右下左（上低下高）

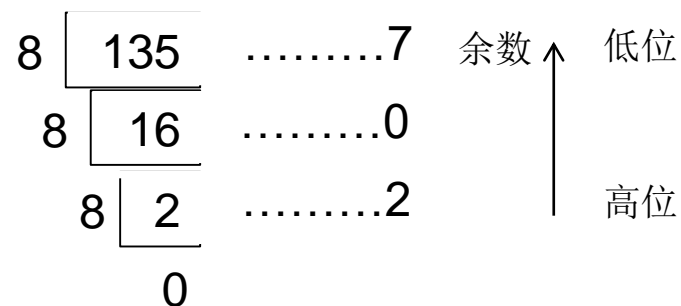
十进制数转二进制数

$$(135)_{10} = (10000111)_2$$



十进制数转八进制数

$$(135)_{10} = (207)_8$$



◆ 小数部分的转换

口诀：乘基取整、上左下右（上高下低）

十进制小数转换二进制小数

$$(0.6875)_{10} = (0.1011)_2$$

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

十进制小数转换八进制小数

$$(0.6875)_{10} = (0.54)_8$$

	整数部分	
$0.6875 \times 8 = 5.5$	5	↓ 高位 低位
$0.5 \times 8 = 4.0$	4	

◆ 合并结果

十进制数转换二进制数

$$(135.6875)_{10} = (10000111.1011)_2$$

十进制小数转换八进制小数

$$(135.6875)_{10} = (207.54)_8$$

(3) 二、八、十六进制数的相互转换

□ 二进制转换成八进制: $(10000111)_2 = (207)_8$

- 从低位到高位，每3个二进制位组成一组，翻译成等值的八进制数码表示即可，高位不足3位的前补0。

□ 二进制转换成十六进制: $(10000111)_2 = (87)_{16}$

- 从低位到高位，每4个二进制位组成一组，翻译成等值的十六进制数码表示即可，高位不足4位的前补0。

□ 八进制转换成二进制: $(207)_8 = (10000111)_2$

- 把八进制数的每一位用3位的等值二进制数替换即可，高低位次序不变。

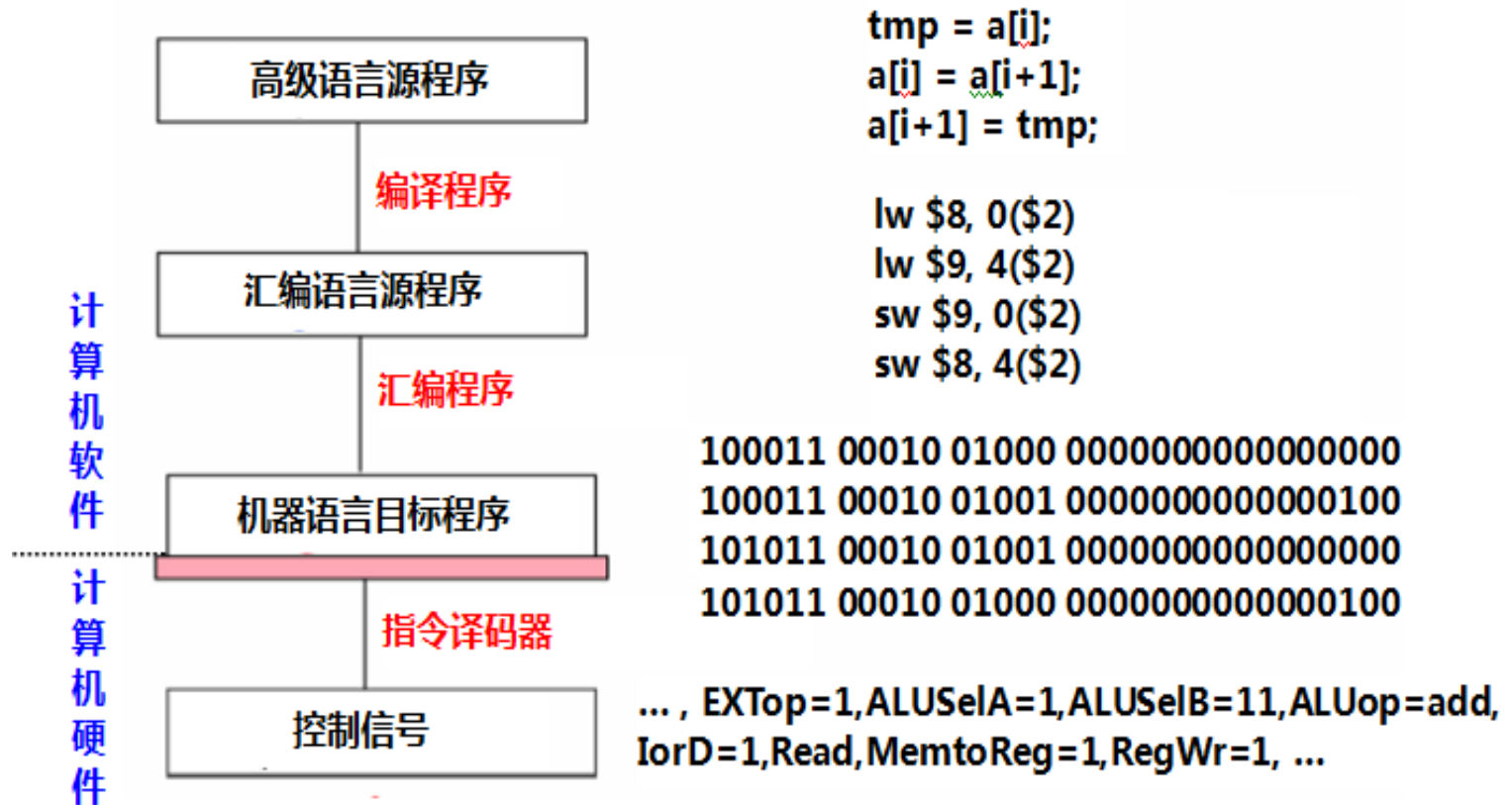
□ 十六进制转换成二进制: $(87)_{16} = (10000111)_2$

- 把十六进制数的每一位用4位的等值二进制数替换即可，高低位次序不变。

4.25讲课内容

1.2 程序的开发和执行过程

- 使用机器语言、汇编语言、高级语言编程
- 不同层次语言之间的等价转换

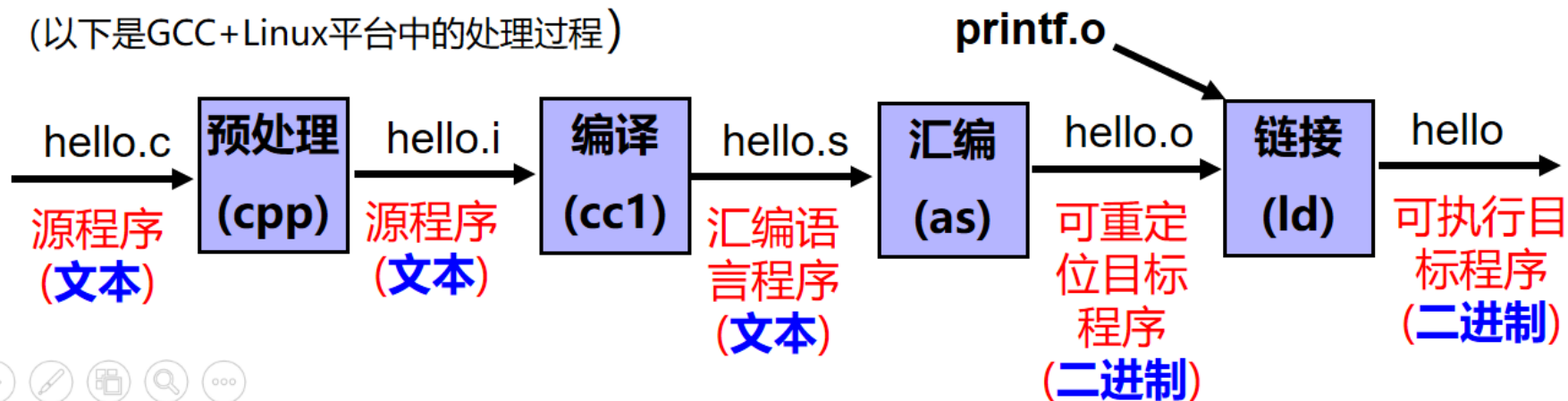


➤ 源程序编辑、编译、链接，得到可执行目标程序

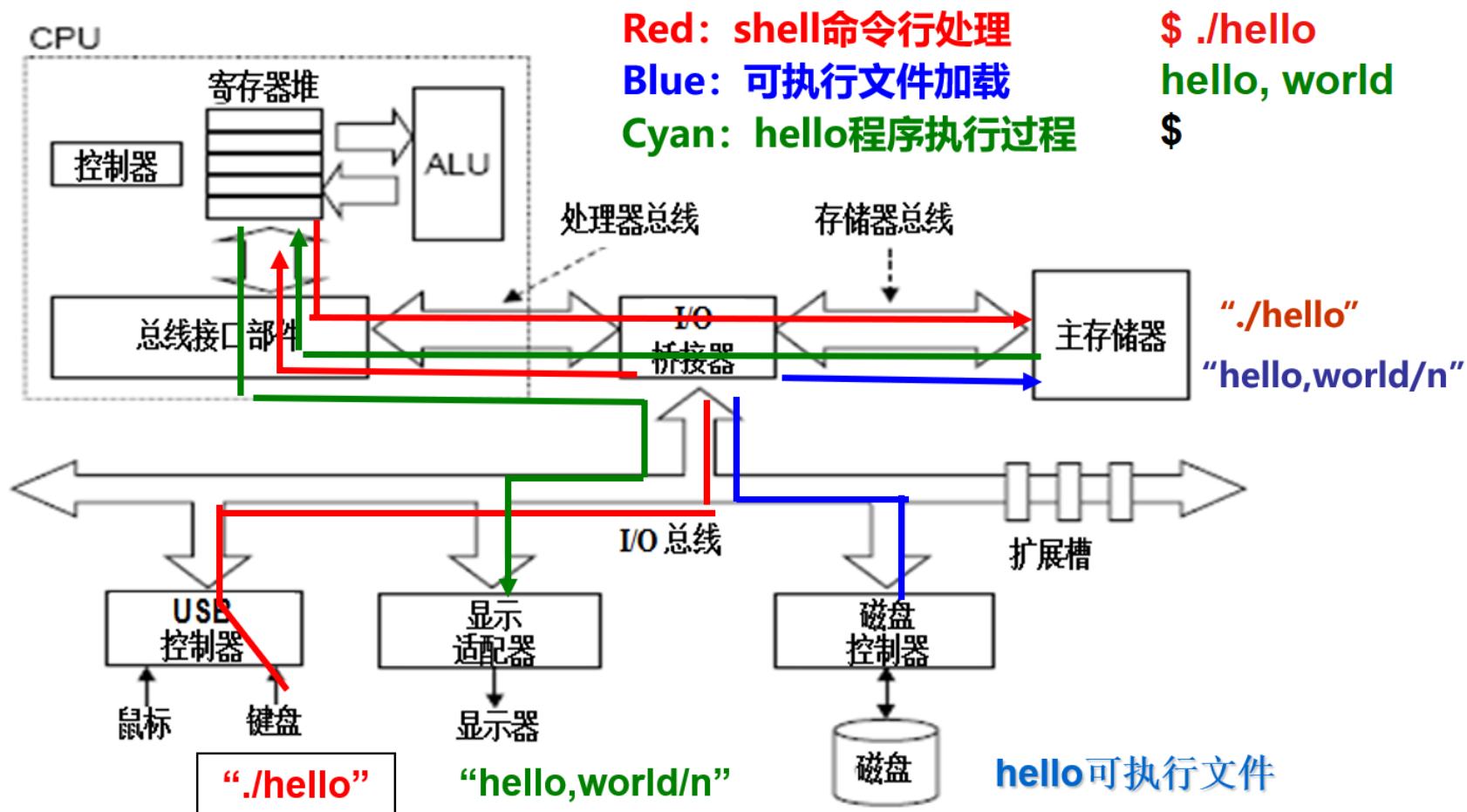
hello.c的ASCII文本文件内容

```
# i n c l u d e < s t d i o .  
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46  
h > \n \n i n t < s p > m a i n ( ) \n {  
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123  
\n < s p > < s p > < s p > < s p > p r i n t f ( " h e l  
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108  
l o , < s p > w o r l d \n " ) ; \n }  
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

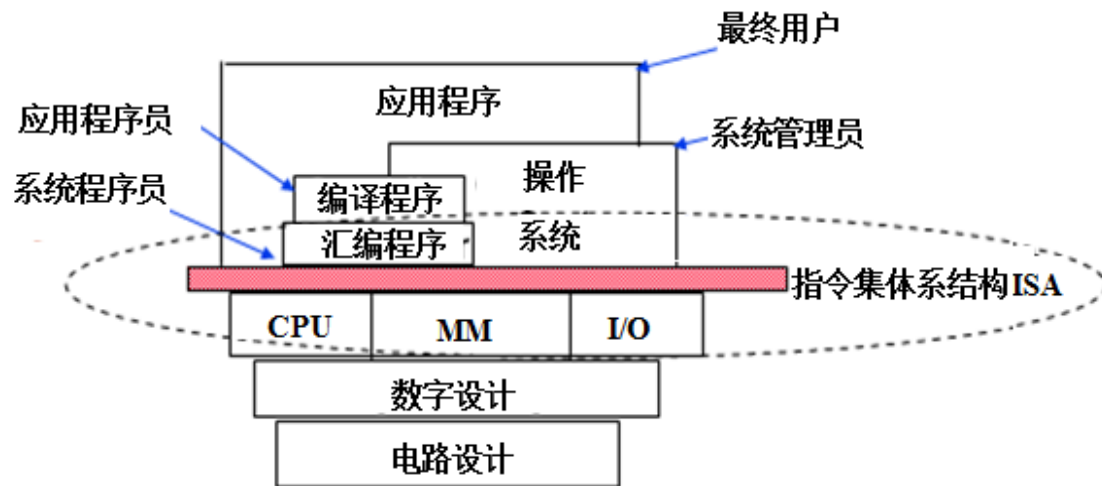
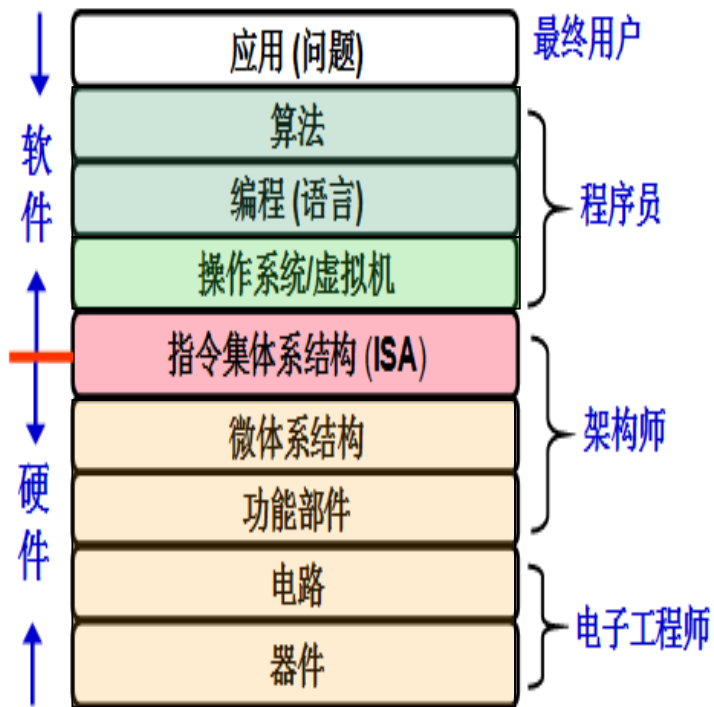
(以下是GCC+Linux平台中的处理过程)



➤ 程序的执行



1.3 计算机系统的层次结构

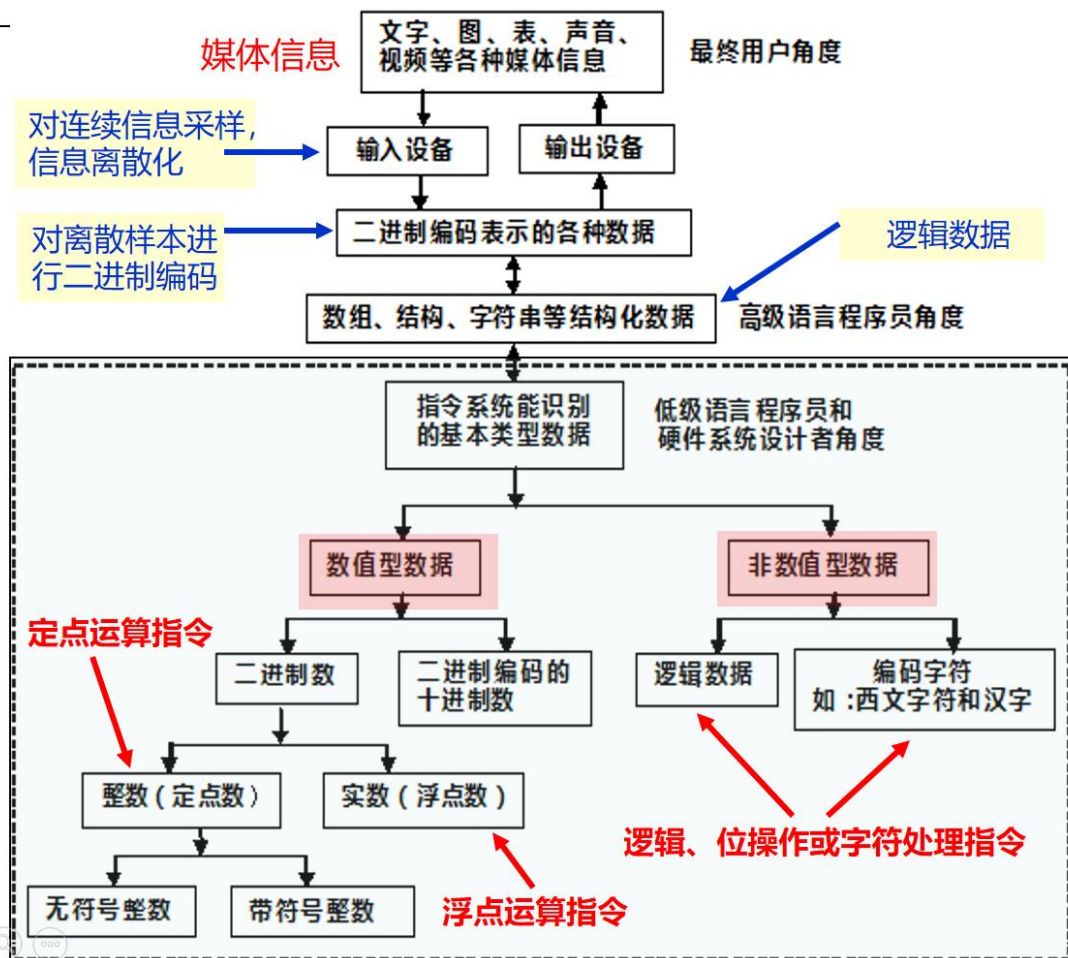


ISA是对计算机组成的抽象，定义了一台计算机可以执行的所有指令的集合和软件使用硬件的方式方法

第二章 数据的机器级表示与处理

◆ 计算机外部信息与内部数据的转换

数字化编码过程：对感觉媒体信息进行采样，将现实世界中的连续信息转换为计算机中的离散的“样本”信息，然后对样本信息用“0”和“1”进行数字化编码。



2.1 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、
编码规则

◆ 进位计数制

– R进制数向十进制数的转换：按权展开

$$(xyz.ab)_R = (x \times R^2 + y \times R^1 + z \times R^0 + a \times R^{-1} + b \times R^{-2})_R$$

– 十进制数向R进制数的转换

➤ 整数部分：除基取余

➤ 小数部分：乘基取整

2、定/浮点表示

—— 用于解决小数点的问题

注：这里说的小数点位置，是指小数点在数的真值里的位置，

如：123456. .123456 123.456

而不是指小数点在该数的机器码里的哪一位上。

■ **定点数**：小数点位置约定在固定位置

➤ **定点整数**：小数点约定在数的最右边

123456.

➤ **定点小数**：小数点约定在数的最左边

.123456

■ **浮点数**：浮点数被表示为： $m \cdot 2^n$ 的形式，小数点位置因n的不同而“浮动”

➤ 在机器级表示时需要明确阶码和尾数的大小。

3、定点数的编码

机器数：一个数值型数据在计算机内部的编码表示；

■ 机器数有多种表示形式：**原码**、**补码**、**反码**、**移码**

真 值：纯数学意义下机器数真正的值，一般直接由

正负号(+、-) + **绝对值**

表示。

如，真值 $(-00001010)_2$

➤ 其8位补码是：11110110B

➤ 机器数是：11110110B

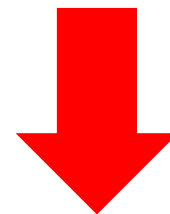
(在现代计算机里，负数在计算机里是用补码表示的)

设机器数X的**真值** X_T 的二进制形式如下：

$$X_T = \pm X'_{n-2} X'_{n-3} \dots X'_1 X'_0 \quad (\text{定点整数})$$

$$\text{或 } X_T = \pm 0.X'_{n-2} X'_{n-3} \dots X'_1 X'_0 \quad (\text{定点小数})$$

注：n-1位数



则， X_T 的**n位**二进制编码的**机器数** X 表示为：

$$X = X_{n-1} X_{n-2} \dots X_1 X_0$$

注：n位数

其中，最高位 X_{n-1} 是**符号位**；

后n-1位 $X_{n-2} \dots X_1 X_0$ 是**数值位**；

那么， $X_i = X'_i$ 吗？

1) 原码

由**符号位** + **真值的数值位**构成的数的表示形式称为原码。

◆ 原码编制的基本规则:

- 当 X_T 为正数时, $X_{n-1}=0$, $X_i=X_i'$ ($0 \leq i \leq n-2$)
- 当 X_T 为负数时, $X_{n-1}=1$, $X_i=X_i'$ ($0 \leq i \leq n-2$)

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

• 原码表示的优点：

- 符号位表示正负号，数值位与真值的数值位相同，因此与真值的对应关系直观，转换方便，容易理解
- 用原码实现乘、除运算比较简便

• 原码表示的缺点：

- 真值0有两种形式：正0和负0，表示不唯一，给使用带来不便
 - $[+0]_{\text{原}} = 0000\dots 0$
 - $[-0]_{\text{原}} = 1000\dots 0$
- 加、减法运算规则复杂：需要对异号数相加或同号数相减做额外的判定
- 而这些符号位的“额外”处理不利于硬件设计

➤ 现代计算机不用原码表示整数，而是都采用**补码**来表示。

➤ 但原码在计算机中有一处应用：

浮点数的尾数用原码定点小数表示。

2) 补码

(1) 模运算 (modular运算)

模同余：在模运算系统中，若A、B、M满足下列关系：

$$A = B + K \times M, \text{ K为整数,}$$

则称B和A为**模M同余**，即**A和B各除以M后余数相同**。

记为 $A \equiv B(\text{mod } M)$ 。

◆ 在一个模运算系统中，称**一个数与它除以"模"后的余数等价**。

现实世界中的模运算系统：时钟，是一种模12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： $10 + 8 = 18 \equiv 6 \pmod{12}$

在模12系统中：

$$-4 \equiv 8 \pmod{12} \quad \text{-4与8模12同余}$$

称8是 -4对模12的补码，即 $8 = 12 + (-4)$

则， $10 - 4 \equiv 10 + (-4) \equiv 10 + (12 - 4) \equiv 10 + 8 \equiv 6 \pmod{12}$

即，在模运算中，减去一个数等于加上这个数负数的补码。

规则1) 一个负数的补码等于**模减该负数的绝对值**。

如：M=12时，-4的补码 = $12 - 4 = 8$

规则2) 对于某一确定的模，某数A减去小于模的另一数B，可用**A加上-B的补码来代替**——在模同余的前提下等价。

如： $10 - 4 \equiv 10 + (12 - 4) \equiv 10 + 8 \pmod{12}$

补码的优点：

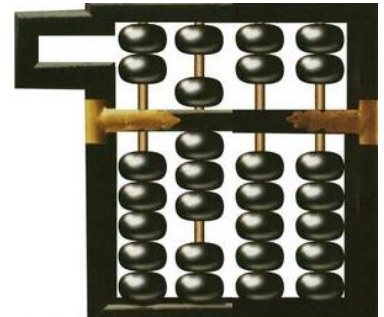
在模运算系统里，可以**实现加减运算的统一**，即用**加法来实现减法运算**——减一个数等于加上这个(负)数的补码(模M)。

在现代计算机中，**用补码表示带符号的整数**。

例2：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？

分析：“4位十进制数”是一个模10000的运算系统

$$\begin{aligned}\text{因此, } 9828-1928 &= 9828 + (10^4 - 1928) \\ &= 9828 + 8072 \\ &= \boxed{1}7900 \\ &= 7900 \quad (\text{mod } 10^4)\end{aligned}$$



只有低4位留在算盘上：取模，只留余数，高位“1”被丢弃！

计算机是一个模运算系统：计算机中的存储、运算和传送数据的部件都只有有限位数。假定运算器有n位，则运算的结果只能保留低n位，超过的高位将被舍弃，这就形成一个模运算系统。

例：设有8位二进制加法器模运算系统

计算 **0111 1111 - 0100 0000 = ?**

$$0111\ 1111 - \underline{0100\ 0000}$$

$$= 0111\ 1111 + (\underline{2^8 - 0100\ 0000})$$

$$= 0111\ 1111 + \underline{1100\ 0000}$$

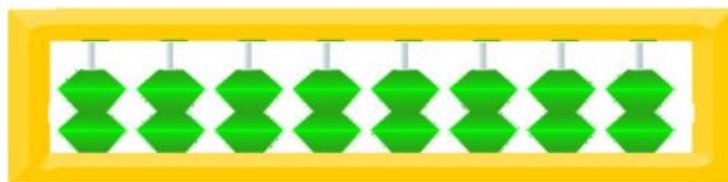
$$= \boxed{1}0011\ 1111 \pmod{2^8}$$

$$= 0011\ 1111$$

“1”被丢弃，
只留余数

计算机中的运算器是模运算系统

假定运算器有 n 位，则计算机中的运算器就是一个**模为 2^n** 的模运算系统（相当于有 n 档的二进制算盘）。



(2) 补码的定义

- 正数的补码等于数的原码。
- 负数的补码等于模与该负数的绝对值之差。

◆ 数 X_T 的补码公式可表示为：

- 当 X_T 为正数时, $[X_T]_{\text{补}} = X_T = M + X_T \pmod{M}$
- 当 X_T 为负数时, $[X_T]_{\text{补}} = M - |X_T| = M + X_T \pmod{M}$

即, 对于任意一个数 X_T , 都有 $[X_T]_{\text{补}} = M + X_T \pmod{M}$

◆ 对具有1位符号位和 $n-1$ 位数值位的 n 位二进制整数,

有: 模 $M = 2^n$

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

补码与真值之间的转换

例: 设机器数有8位, 求123和-123的补码表示。(二进制表示)


解: **123**: 真值: $+1111011\text{B}$, 原码: 01111011B

$$\begin{aligned}\text{补码: } [01111011]_{2\text{补}} &= 2^8 + 01111011 \\ &= 100000000 + 01111011 \\ &= \mathbf{01111011} \pmod{2^8} \quad \text{即 } 7\text{BH}\end{aligned}$$

正数补码: 等于原码

-123: 真值: -1111011B , 原码: 11111011B

$$\begin{aligned}\text{补码: } [-1111011]_{\text{补}} &= 2^8 - 01111011 = 10000\ 0000 - 01111011 \\ &= \mathbf{1111\ 1111} - \mathbf{0111\ 1011} + 1 \\ &= 1000\ 0100 + 1 \\ &= \mathbf{1000\ 0101} \quad \text{即 } 85\text{H}\end{aligned}$$



负数补码: (该数绝对值的) 原码取反加1

由补码求真值：

- 若符号位为0，则真值的**符号为+**，其数值部分不变；
- 若符号位为1，则真值的**符号为-**，其数值部分仍为：

各位取反，末位加1。

例：已知 $[X_T]_{\text{补}} = 1 \text{ } \underline{0110100}$ ，求其真值。

解： $X_T = -(\underline{1001011} + \underline{1}) = -1001100$

◆ 已知 $[X_T]_{\text{补}}$ 求 $[-X_T]_{\text{补}}$ ：对 $[X_T]_{\text{补}}$ 各位取反，末位加1。

◆ 对最小负数取负会发生溢出

◆ 变形补码：P40



特殊数的补码

假定机器数有 n 位，最高位为符号位，数值位有 $n-1$ 位，**模为 2^n**

$$\textcircled{1} [-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = (10\dots0)_2 \quad (n-1\text{个}0) \quad (\text{mod } 2^n)$$

$$\textcircled{2} [-1]_{\text{补}} = 2^n - (0\dots01)_2 = (11\dots1)_2 \quad (n\text{个}1) \quad (\text{mod } 2^n)$$

$$\textcircled{3} [+0]_{\text{补}} = [-0]_{\text{补}} = (00\dots0)_2 \quad (n\text{个}0) \quad \text{0的表示唯一}$$

思考：

$$\textcircled{1} [2^{n-1}]_{\text{补}} = ? \quad [2^{n-1}]_{\text{补}} = 2^n + 2^{n-1} (\text{mod } 2^n) = (10\dots0)_2 \quad (n-1\text{个}0) \quad (\text{mod } 2^n)$$

负数？ 所以：最大正整数为： **$2^{n-1}-1$**

$$\textcircled{2} -0、+0\text{都是}00\dots0 \text{ (} n\text{个}0 \text{)} , \text{那}10\dots0 \text{ (} n-1\text{个}0 \text{)} \text{表示多少？} \quad \textbf{-}2^{n-1}$$

所以：最小负整数为： **-2^{n-1}**

3) 反码

- 正数的反码与其原码相同;
- 负数的反码是符号位不变, 数值位为原码的数值位逐位取反。

如: 8位原码表示的符号整数 $(10010001)_{\text{原}}$, 其反码为:

$$((10010001)_{\text{原}})_{\text{反}} = (11101110)_{\text{反}}$$

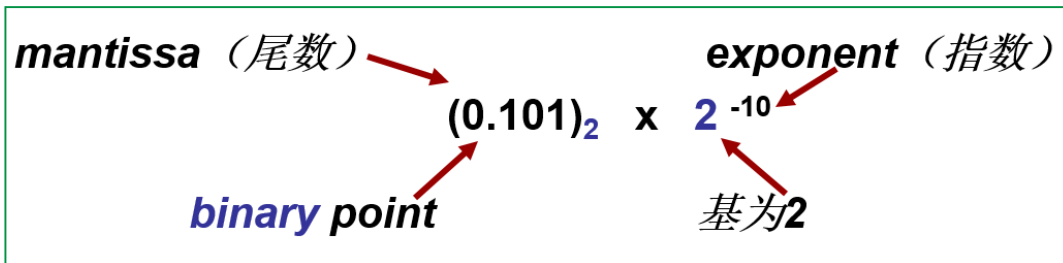
首位1为符号码, 表示为负数

负数的补码等于它的反码+1

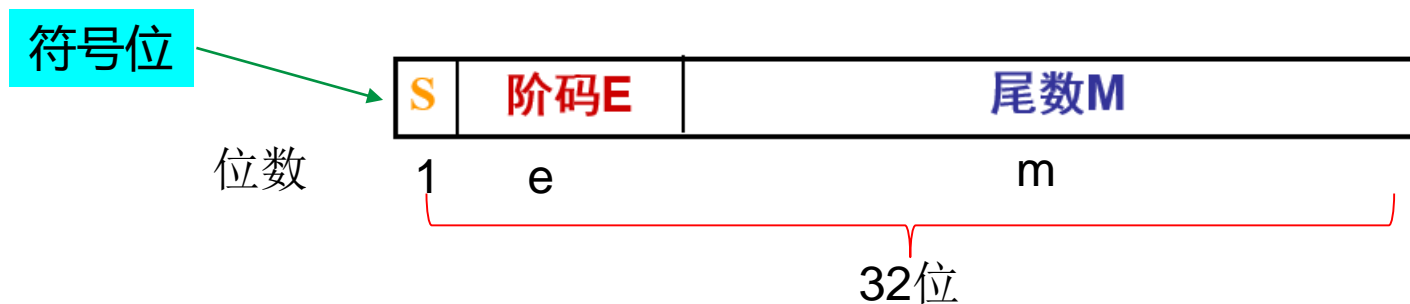
4) 移码

■ 浮点数的阶码用“移码”表示

浮点数的一般形式：



计算机中，浮点数表示成：



浮点数的阶可以是正数，也可以是负数，用e位表示阶码，

怎么表示阶码的正负呢？

◆ 为阶码设置自己的符号位吗？ NO！

■ e位的阶码可表示的数值范围：

◆ 带符号： $-2^{e-1} \sim 2^{e-1}-1$ 如8位： $-128 \sim 127$

◆ 无符号： $0 \sim 2^e-1$ 如8位： $0 \sim 255$

为了简化阶的比较操作，使操作过程**不涉及阶的符号**，对每个阶的实际值都**加上一个正常数**，**使所有的阶都转换为非负数**。

如： $-1 + 128 = 127$

$1 + 128 = 129$

■ 所加的正常数称为**偏置常数** (bias)。

假设用来表示阶E的移码的位数为e，则偏置常数通常取 2^{e-1} 或 $2^{e-1}-1$

■ 加了偏置常数的编码称为“**移码**”： $[E]_{\text{移}} = \text{偏置常数} + E$

■ **阶码还原**：**移码-偏置常数**

2.2 整数的表示

- 整数分为：**无符号整数**和**有符号整数**
均采用**定点整数表示**

- **带符号整数** (Signed integer)

- **最高位为符号位**：0表示正数，1表示负数
 - **其余位为数值位**，可表示的数值范围是 $-2^{n-1} \sim 2^{n-1}-1$ 。
- 例如，8位带符号整数的数据范围是 $-128 \sim 127$ 。

- **带符号整数用补码表示**

补码运算系统是模运算系统，实现加、减运算的统一。

■ 无符号整数 (Unsigned integer)

➤ 如果编码的所有二进制位都用来表示数值，没有符号位，则该编码表示的是**无符号整数**（简称无符号数）。

■ 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，表示指针等。

➤ 由于编码中没有符号位，表示数值的位数比位数相同的带符号整数多一位，所以**无符号整数表示的正整数范围大一倍**。

例如，8位无符号整数最大是255 (1111 1111)

8位带符号整数最大为127 (0111 1111)

(但需要注意：不管是无符号还是有符号， n 位二进制数表示的**数的个数是相同的，都是 2^n 个**)

C语言程序中的整数及其相互转换

1) C语言的整数类型

- **无符号整数**: unsigned int / short / long
- **带符号整数**: int (short / long)

C语言中，一般在一个数的后面加一个 “u” 或 “U” 表示无符号数；如，12345U, 0x2B3Cu

2) 数据类型的转换

如果表达式中有无符号整数和带符号整数之间的类型转换：

- (1) **机器数不变**：类型转换后，机器数各位的0/1值不变；
- (2) **真值需要将按照转换后的类型进行解释。**

影响：会造成程序在某些情况下发生意想不到的结果。

以下关系表达式在32位机器上执行，**整数用补码表示**，观察结果

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*	$011...1B (2^{31}-1) < 100...0B(2^{31})$
$2147483647 > (\text{int}) 2147483648U$	带	1*	$011...1B (2^{31}-1) > 100...0B (-2^{31})$
$-1 > -2$	带	1	$11...1B (-1) > 11...10B (-2)$
$(\text{unsigned}) -1 > -2$	无	1	$11...1B (2^{32}-1) > 11...10B (2^{32}-2)$

带*的结果：按照当前类型对“位值”进行解释

例，考虑以下C代码：

```
1    int x = -1;
2    unsigned u = 2147483648;
3    printf ( "x = %u = %d\n" , x, x);
4    printf ( "u = %u = %d\n" , u, u);
```

在32位机器上运行上述代码时，输出结果为

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

① 因为 -1 的补码表示为 “ $11\cdots 1$ ”，所以，作为32位无符号数解释时，

其值为 $2^{32} - 1 = 4294967296 - 1 = 4294967295$ 。（ $0xFFFFFFFF$ ）

② $2147483648 = 2^{31}$ ，无符号数表示为 “ $100\cdots 0$ ”，被解释为32位

带符号整数时，其值为**最小负数**： $-2^{32-1} = -2^{31} = -2147483648$

例2.21 在有些32位系统上，

1) C表达式 **-2147483648 < 2147483647** 的执行结果为false。Why?

C90中，编译器对**数值型字面常量**的类型确定顺序：


	范围	类型
2147483647	$0 \sim 2^{31}-1$	int
2147483648	$2^{31} \sim 2^{32}-1$	unsigned int
	$2^{32} \sim 2^{63}-1$	long long
	$2^{63} \sim 2^{64}-1$	unsigned long long

编译器对表达式 “-2147483648 < 2147483647” 的处理是：首先处理 “-2147483648”，方法是将其分成负号和值2147483648两部分来处理(相当于0-2147483648)。由于2147483648的机器数是0x80000000，因此，在ISO C90标准下，将2147483648看成unsigned int；虽然对其取负并求补码，但最终结果仍为0x80000000，类型仍是无符号整数，值为2147483648。同时2147483647的类型也被提升，当做无符号数处理。

所以 “-2147483648” 与 “2147483647” 的比较实际上变成了无符号数2147483648与2147483647的比较，自然有 “-2147483648 < 2147483647” 的结果为false。

但在**ISO C99**标准下，结果为true。

C99中，编译器对**数值型字面常量**的类型确定顺序：



范围	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

在ISO C99标准下，字面常量“**2147483648**”被编译器解释成为64位的**long long型带符号整数**，将按照带符号整数的规则处理：

以补码表示机器数，但值被解释为负数，参与带符号数的运算。
所以“-2147483648 < 2147483647”的结果为true。

2) 若定义变量 `int i=-2147483648;`, 则 **`i < 2147483647`** 的执行结果为true。Why?

首先, 编译器在处理 `"int i=-2147483648;"` 时, 将机器数**`0x80000000`**按照i的类型进行转换 (**赋值转换**) 后赋给i, i为带符号整数, 机器数没变, 但i的值为-2147483648;

之后, **按照带符号整数的规则与计算** `i < 2147483647`, 所以结果为true。

3) 如果将表达式写成 **`"-2147483647-1 < 2147483647"`**, 则结果会怎样呢? Why?

◆ 编译器在处理**`-2147483647`**时, 首先将值2147483647 (`0x7FFFFFFF`, $2^{31}-1$) 看成带符号数, 对其取负, 得-2147483647 (补码表示), 然后再减1, 得到的是**带符号数结果**, 即-2147483648 (补码表示), 再然后作为带符号数参与和2147483647的比较, 所以**结果为true**。

同学们可以尝试运行一下程序（C90），看看结果是什么？

```
#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。

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

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

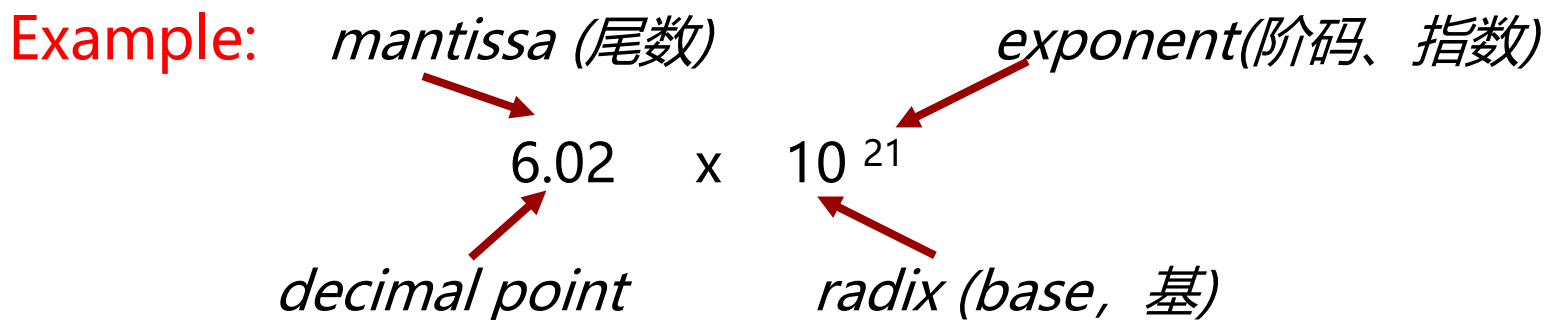
```
-2147483648 < 2147483647 is false
```

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

2.3 浮点数的表示

■ 浮点数的科学计数法(Scientific Notation)

Example: *mantissa (尾数)* 6.02×10^{21} *exponent (阶码、指数)*
decimal point *radix (base, 基)*



The diagram illustrates the components of the scientific notation 6.02×10^{21} . Red arrows point from the labels to the corresponding parts of the expression: 'mantissa (尾数)' points to '6.02', 'decimal point' points to the dot in '6.02', 'exponent (阶码、指数)' points to '21', and 'radix (base, 基)' points to '10'.

■ 尾数的规格化 (Normalized):

➤ 约定尾数小数点前有且只能有一位非0数

如：将0.0000000001表示成 1.0×10^{-9}

➤ 规格化的目的：将小数表现形式唯一化。

■ 也可以规定尾数整数部为0且小数部的首位不能为0。

如, for Binary Numbers:

The diagram illustrates the binary floating-point notation $(0.101)_2 \times 2^{-10}$. Red arrows point from labels to specific parts of the expression: 'mantissa (尾数)' points to '0.101', 'exponent (指数)' points to '-10', 'binary point' points to the dot in '0.101', and '基数为2' (base is 2) points to the '2' in '2^-10'.

$$\text{mantissa (尾数)} \rightarrow (0.101)_2 \times 2^{-10} \leftarrow \text{exponent (指数)}$$

$\text{binary point} \rightarrow$ $\text{基数为2} \rightarrow$

浮点数的编码: 只要对**尾数**、**指数** (阶码)、**符号位**三部分分别编码, 就可表示一个浮点数。而**基数**是约定好的。

浮点数表示：用**定点小数**表示尾数，用**定点整数**表示阶，再加上**符号位**合并而成。

即，对任意实数X，可以表示成为：

$$X = (-1)^s \times M \times R^E$$

其中，

- **M**是一个二进制定点小数，称为X的尾数（mantissa）；
- **E**是一个二进制定点整数，称为X的阶或指数(exponent)；
- **R**是基数（radix、base），可取2、4、16等，事先约定，不用另外表示。

所以，要表示浮点数，只要对**尾数**、**指数**、**符号位**分别编码即可

一种32位浮点数的规格化格式表示如下：



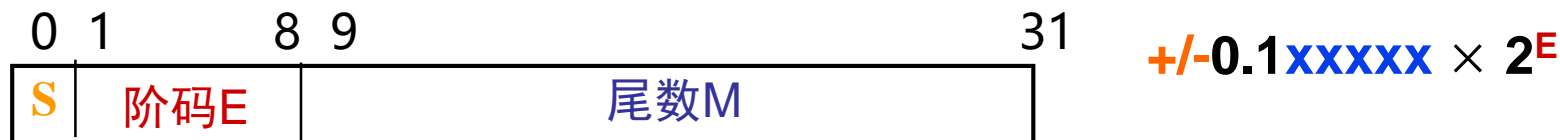
注：这里位的编号顺序从左往右依次为0~31

- 第0位：符号位S；
- 第1 ~ 8位：8位“移码”表示的阶码E。偏置常数取128；
- 第9 ~ 31位：23位二进制位，表示尾数M的24位原码小数。

规格化：尾数用定点小数表示，且规格化后整数部为0，而小数点后第一位必须是1。

- 这样，第一位默认的“1”可以不明显表示出来，故可用23个数位表示24位尾数。

32位规格化浮点数的表示范围。

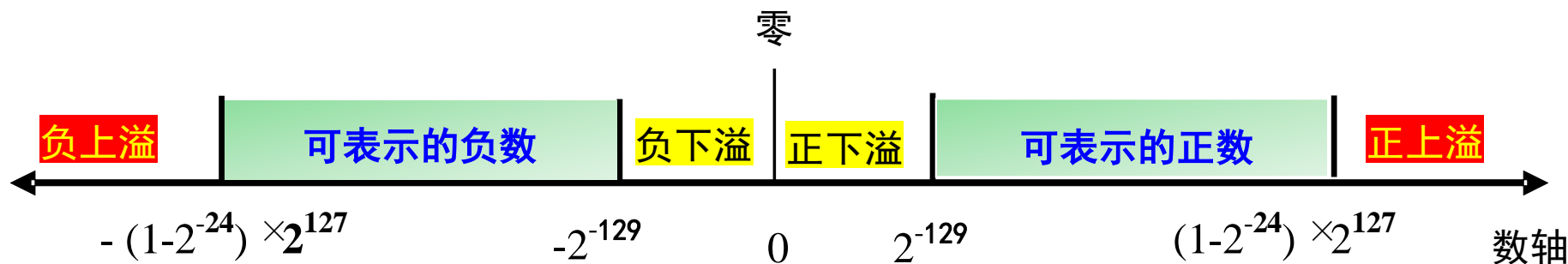


第0位数符S；第1~8位“移码”表示的阶码E（偏置常数为128）；第9~31位为24位二进制原码小数表示的尾数M。用23个数位表示24位尾数。

最大正数： $0.11\dots1 \times 2^{11111111} = (1-2^{-24}) \times 2^{255-128} = (1-2^{-24}) \times 2^{127}$

最小正数： $0.10\dots0 \times 2^{00000000} = (1/2) \times 2^{0-128} = (1/2) \times 2^{-128} = 2^{-129}$

负数：因为原码是对称的，所以其表示范围与正数关于原点对称。



- 浮点数表示的数值范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀：对每个固定的“阶”，有 2^{23} 个离散的数。

4.28 (第三次课)

1. 机器数、真值、原码、反码、补码、移码
2. C语言中类型不同的整数之间转换：机器数不变，真值不同

2. 浮点数的表示

$$\text{mantissa (尾数)} \rightarrow (0.101)_2 \times 2^{-10} \leftarrow \text{exponent (指数)}$$

$$\text{binary point} \rightarrow \text{decimal point in } (0.101)_2$$

$$\text{基为2} \rightarrow 2$$

浮点数的编码：对尾数、阶码、符号位三部分分别编码

尾数的规格化：尾数用定点小数表示，且规格化后整数部为0，
小数点后第一位必须是1。

- 第一位默认的“1”可以不明显表示出来,故可用23个数位表示24位尾数。

阶码编码：移码=阶码+偏置常数

“Father” of the IEEE 754 standard

- ◆ 早期，不同的计算机厂商有自己的浮点数格式。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，相互不兼容，对机器之间的数据交换带来麻烦。
- ◆ 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。现在所有计算机都采用IEEE 754来表示浮点数。

“Father” of the IEEE 754 standard



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



www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

设计了8087浮点处理器，对浮点运算有丰富的经验

IEEE 754标准:

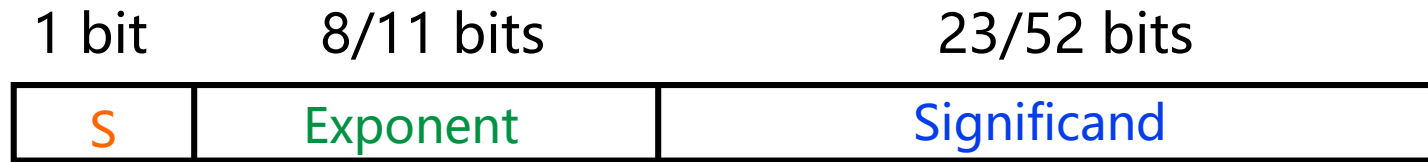
规格化数: $+/-1.\text{xxxxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$

规定: 小数点前总是“1”, 故可隐含表示。

➤ 注意: 和前面例子的规定不一样。

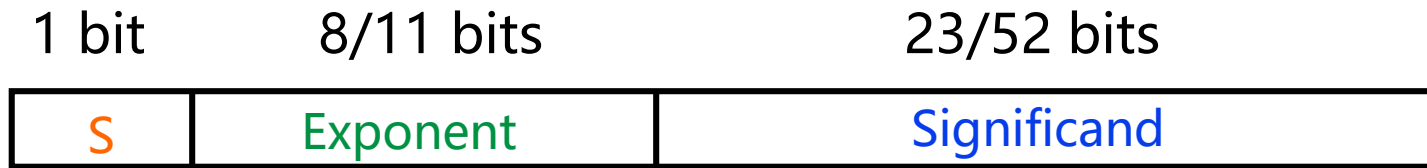
包括: 单精度、双精度浮点数

Single/Double Precision(SP/DP, 单/双精度浮点数, 32/64位)



- **符号位: 1位, 0/1**
- **阶 码: 8位/11位**
 - **阶码范围: $-126 \sim 127$ / $-1022 \sim 1023$**
 - **偏置常数: 127 / 1023**
 - **移码范围: $1 \sim 254$ / $1 \sim 2046$**
 - 移码视为无符号整数**
 - 全0和全1用来表示特殊值**
- **尾数: 23位/52位, 定点小数, 原码**
 - **规格化尾数最高位总是1, 隐含表示, 用23位表示24位尾数**
 - **$1 + 23$ bits (single) , $1 + 52$ bits (double)**

Single/Double Precision(SP/DP, 单/双精度浮点数, 32/64位)



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

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

Ex: Converting Binary FP to Decimal

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

1 011 11101 110 0000 0000 0000 0000 0000

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

Sign: 1 \Rightarrow negative

Bias, SP: 127

Exponent: 0111 1101_{two} = 125_{ten}

Bias adjustment: 125 - 127 = -2

Significand: (原码定点小数)

$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

Represents: $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$

Ex: Converting Decimal to FP

将-12.75转换为32位单精度规格化浮点数表示形式

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: $127 + 3 = 128 + 2 = 1000\ 0010_2$

11000000 10011000 00000000 00000000

The Hex rep. is **C14C0000H**

IEEE754对于**全0或全1的阶码**有特殊的指代。

Exponent	Significand	Object
1-254	规格化尾数	规格化模式
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

非规格化模式

非规格化模式的数代表什么呢？

1) 全0阶码+全0尾数表示数值0

■ IEEE 754的0有两种表示:

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

■ 一般情况下+0和-0等效。

2) 阶码为0, 尾数不为0表示非规格化数

◆ 非规格化数的作用：用于表示比最小规格化数还小的浮点数

➤ 最小规格化数： $1.0...0 \times 2^{-126}$

◆ 规则：

➤ 阶码默认为 -126（单精度，双精度默认为-1022）

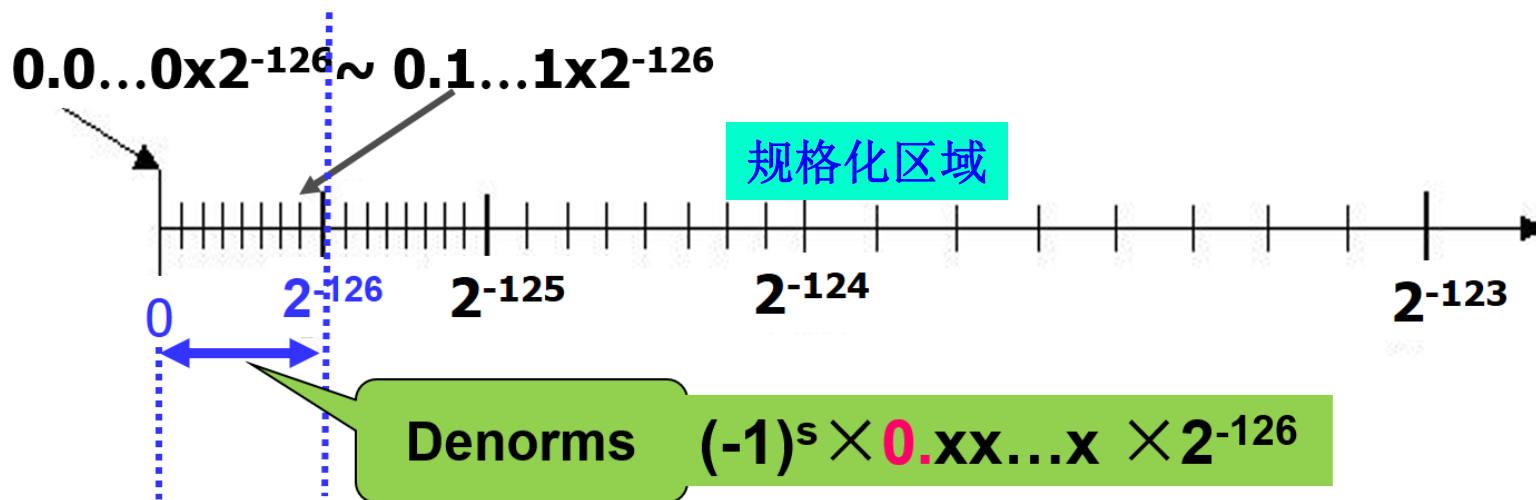
➤ 隐藏位为 0

◆ 非规格化数的数值为： $(-1)^s \times 0.f \times 2^{-126}$ （单精度）

或 $(-1)^s \times 0.f \times 2^{-1022}$ （双精度为）

表示区间 $[0.0...1 \times 2^{-126}, 0.1...1 \times 2^{-126}]$ 内的数

尾数不能为0



- ◆ 非规格化数共有 $2^{23}-1$ 个（不能为0）
- ◆ 弥补了0到 $1.0...0 \times 2^{-126}$ 之间的空缺，**处理阶码下溢**（比最小阶码还小的情况）。

3) 指数全1、尾数全0 表示 $+\infty/-\infty$

∞ : infinity

$+\infty/-\infty$: 是两个特殊的数, 表示比**最大/最小有限数**还大/小的**抽象数**。

➤ 形式: 指数全1、尾数全0

$+\infty$: 0 1111111 000000000000000000000000

$-\infty$: 1 1111111 000000000000000000000000

➤ 作用: 引入无穷大数使得在浮点计算过程出现异常时,
如 $5.0 / 0$, 还可以继续执行下去。

观察表达式结果: $5/0$

如果整数除0, 程序异常终止。

$5.0/0$

浮点数除0, 输出: 1.#INF00, 程序不会终止。 ($-\infty$: -1.#INF00)

- ◆ **无穷大数还可作为操作数。** 用作操作数时，系统有两种处理方式：

(1) **产生明确结果。**

如： $5 + (+\infty) = +\infty$, $(+\infty) + (+\infty) = +\infty$

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

(2) **产生不发信号的非数NaN。**

如： $(+\infty) + (-\infty)$, $+\infty - (+\infty)$, $+\infty / +\infty$



输出： -1.#IND00

$+\infty * +\infty$

输出： 1.#IND00

- 自己可以编程测试上述表达式，看看会输出什么结果信息。
- $+\infty$ 和 $-\infty$ 怎么产生？

4) 阶码全1，尾数非零表示非数

- ◆ 非数：没有定义的数（NaN, Not a Number）
- ◆ 形式：阶码全1，尾数非零
- ◆ 非数的作用：

- 程序执行异常检测

如： $+\infty / +\infty = -1.\text{\#IND00}$ $0.0 / 0 = -1.\text{\#IND00}$

$+\infty + (-\infty) = -1.\text{\#IND00}$

- 编译器可以用非数作为浮点数变量的非初始化值

```
float f;  
printf("f=%f",f);
```

输出： 1.#IND00

非数分为两种：**静止的NaN**和**通知的NaN**

◆ 尾数最高有效位为1，为**不发信号** (quiet) 非数：

当结果产生这种非数时，**不发异常操作通知，不进行异常处理。**

如：**-1.#IND00**

◆ 尾数最高有效位为0，为**发信号**(signaling)非数：

当结果产生这种非数时，**发异常操作通知，通知系统进行异常处理。**

同时，由于NaN的尾数是非0的数，除了**第一位**有定义外，其余各位没有定义，所以**可用其余位来指定具体的异常事件。**

单精度/双精度浮点数格式:

阶码个数: 254 / 2046

格式化数:

- 阶的范围: $-126 \sim 127$ / $-1022 \sim 1023$
- 数值个数: $254 \times 2^{23} = 1.98 \times 2^{31}$ / $2046 \times 2^{52} = 1.99 \times 2^{63}$
- 数据范围: $10^{-38} \sim 10^{38}$ / $10^{-308} \sim 10^{308}$

非规格化数:

- 固定阶码: -126 / -1022
- 最小可表示数: $0.0...01 \times 2^{-126} = 2^{-149}$ / $2^{-52} \times 2^{-1022} = 2^{-1074}$

其他问题：

单精度/双精度扩展： 用更多的字节表示更大范围的浮点数。

P49, 自学

2.3.4 C语言中的浮点数类型

◆ 基本浮点数类型：float, double

分别与IEEE 754单精度浮点数格式和双精度浮点数格式相对应。

◆ C语言双精度扩展：long double

随编译器和处理器类型不同而不同（见P49）。

◆ 当程序中存在int、float、double等类型之间强制类型转换时，程序使用到的值是数值转换后的结果，可能引发异常。

① 从int转换为float:

- ◆ float表示范围大于int, **数据不会发生溢出, 但有效数字可能会被舍去;**

② 从int或float转换为double:

- ◆ 因为double的有效位多于int和float, **可以保留精确值;**

③ 从double转换为float:

- ◆ 因为float表示范围小, 所以**数据可能发生溢出;** 同时因为float的有效位少于double, 故**可能存在数据舍入;**

④ 从float或double转换为int:

- ◆ 因为int没有小数部分, 所以**数据可能会被向0方向截断:**

如: 1.9999被转换为1, -1.9999被转换为-1

- ◆ 同时, 由于int的表示范围小, 故**可能发生溢出。**

例2.25 假定变量i、f、d的类型分别是int、float、double，可以取除 $+\infty$ 、 $-\infty$ 和NaN之外的任意值。判断下列每个C语言关系表达式在32位机器上运行时是否永真。

① **`i == (int)(float)i`**

有效位丢失

② **`f == (float)(int)f`**

小数部分丢失

③ **`i == (int)(double)i`**

可以精确转换

④ **`f == (float)(double)f`**

可以精确转换

⑤ **`d == (float)d`**

丢失有效位或溢出

⑥ **`f == -(-f)`**

仅改变一下符号

⑦ **`(d+f)-d == f`**

大数吃小数

因为类型转换而引起的错误有的时候不易察觉，却会因此而造成重大的损失，所以编程的时候要格外小心。避免发生词类错误。

2.4 十进制数的表示（本节自学）

在计算机内部表示十进制数，可以采用两种方式：

1) 用ASCII码字符串表示

把十进制数看成字符串，如“1234”，直接用0~9数码的ASCII码（30H~39H）表示。

特点：

- 一个十进制数码对应8位二进制数（一个字节，数码符号）；
- 一个十进制数在计算机内部需**占用多个连续字节**；
- **方便输入输出，不方便运算**。运算前必须转换成数值（二进制数或BCD码），然后才能参与计算。

2) BCD码 (Binary-Coded Decimal, 二进制编码的十进制数)

- ◆ 每个十进制数位用4位二进制位串表示

 - 16选10, 多种方案, 不唯一

- ◆ 一个字节可以分成前后两组4位的位串, 从而可以表示两个十进制数码

- ◆ 计算机有专门的指令和逻辑电路可以直接进行BCD码的运算

- ◆ 分为有权BCD码和无权BCD码

(1) 有权BCD码

4个二进制数位都有一个确定的权，对应的十进制数可通过按权展开的方式进行换算。

➤ 8421码：

- 用四位二进制数的前10个代码 $(0000)_2 \sim (1001)_2$ 分别表示对应的10个十进制数字0~9。

如： $(1234)_{10} = (0001001000110100)_{\text{BCD}}$

- 每位的权从左到右分别为8、4、2、1，因此称为**8421码**，也称自然BCD码。

(2) 无权BCD码

4个二进制数位没有一个确定的权，不能简单地按权展开。

如：余3码、格雷码

在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为**格雷码**（Gray Code）。

格雷码有多种编码形式，如4位编码：

十进制数	4位自然二进制码	4位典型格雷码	十进制余三格雷码	十进制空六格雷码	十进制跳六格雷码	步进码
0	0000	0000	0010	0000	0000	00000
1	0001	0001	0110	0001	0001	00001
2	0010	0011	0111	0011	0011	00011
3	0011	0010	0101	0010	0010	00111
4	0100	0110	0100	0110	0110	01111
5	0101	0111	1100	1110	0111	11111
6	0110	0101	1101	1010	0101	11110
7	0111	0100	1111	1011	0100	11100
8	1000	1100	1110	1001	1100	11000
9	1001	1101	1010	1000	1000	10000
10	1010	1111	----	----	----	----
11	1011	1110	----	----	----	----
12	1100	1010	----	----	----	----
13	1101	1011	----	----	----	----
14	1110	1001	----	----	----	----
15	1111	1000	----	----	----	----

余3码是一种BCD码，它是**由8421码加3后形成的**，即在8421码基础上每个8421BCD码加上二进制数0011得到的。

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

- 8421码中无1010 ~ 1111这6个代码
- 余3码中无0000 ~ 0010、1101 ~ 1111这6个代码
- 余3码不具有有权性，但具有自补性，余3码是一种“对9的自补码”。
- 优点：执行十位数相加时可以产生正确的进位信号，而且给减法运算带来了方便。

2.5 非数值数据的编码表示（本节自学）

逻辑值、字符等数据都是非数值数据，在机器内部它们用一个二进制位串表示。

1. 逻辑值的编码表示

逻辑数据按位对待处理，以位串的形式存在

- 一个逻辑值用1个bit表示，取值1或0
- N位二进制数可表示N个逻辑数据

逻辑数据的运算：按位与、按位或、逻辑左移、逻辑右移等位运算

如何识别逻辑数据：

- 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。
- 靠机器指令来识别——专门进行逻辑运算的指令，其中的操作数当作逻辑数据、按位使用。

2.西文字符的编码表示:ASCII码

- 西文字符的特点

- 西文字符集：26个英文字母（大小写）及数学符号、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

- 西文字符表示（常用编码为7位ASCII码）

- 十进制数字(0/1/2.../9)：30H~39H

特点： 编码高三位均为011，低4位为0~9的8421码

- 英文字母(A/B/.../Z、a/b/.../z)：41H~5AH、61H~7AH

特点： b_5 位为0是大写字母， b_5 位为1是小写字母，大小写转换方便

- 专用符号：+/-/%/*/&/.....

小写字母=大写字母+20H

- 控制字符（不可打印或显示）

3.汉字及国际字符的编码表示

1) 汉字的特点

- 汉字是表意文字，一个字是一个方块图形，字和词不能靠简单的“组合”表示。
- 汉字数量巨大，总数超过6万字，这对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。

2) 汉字编码

- **输入码** (外码) : 对汉字用相应按键进行编码表示, 用于输入。
 - 如: 区位码、全拼、简拼、双拼、五笔字型、智能ABC等
- **内码**: 用于在系统对汉字进行存储、查找、传送
 - 如: **国标码** (GB2132-80, 6763个汉字)
 - 区位码**: 94行X94列, 7位区号+7位位号
 - 机内码**: 双字节, 由区位码的区、位号各加20H, 再将最高位置1得到。
- **ISO/IEC10646**: 国际标准, 包含全世界现在书面语言文字所使用的所有字符的标准编码, 每个字符用4字节 (称为UCS-4) 或2字节编码 (称为UCS-2) 。
- **Unicode**: Windows操作系统使用, 与UCS-2一致

3) 汉字内码

至少需2个字节才能表示一个汉字内码。为什么？

– 由汉字的总数决定！

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。

区位码： 0010100 1010011 1453H

国标码： 00110100 01110011

在区位码的基础上区、位号各加20H，即3473H。

问 题：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，如何区分？

构造机内码：将每个字节的最高位各设为“1”后，得到其机内码，如
B4F3H (1011 0100 1111 0011B)，这样就不会和ASCII码混淆。

3) 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 (字符) 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

■ 字形主要有两种描述方法：

- **字模点阵描述** (图像方式)：黑白点转换成1/0编码。
 - ◆ 描述汉字字模的**二进制点阵数据**称为汉字的**字模点阵码**。
 - ◆ 大小一致，每个汉字是16X16、24X24、36X36的方阵。
- **轮廓描述** (图形方式)：描述**汉字笔画的轮廓曲线**，然后画出来。
 - ◆ 可用直线勾画，也可用曲线勾画 (如True Type字形)。
 - ◆ 精度高，任意放缩。

■ 汉字库的使用

- 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？

用汉字内码对其在字库中的位置进行索引！

2.6 数据的宽度和存储

1. 数据的宽度和单位

- 位 (比特, bit)

bit是计算机中处理、存储、传输信息的最小单位。

- 字节 (Byte)

- 一个字节是由8个bit组成的 “位组”。
- 字节是计算机二进制信息的基本计量单位;
- 字节是最小可寻址单位

——现代计算机中, 存储器按字节编址 (*addressable unit*)。

- 字 (WORD) :

一个字等于2个字节, 16位

- 双字 (DWORD) :

一个双字为两个字宽度, 4个字节, 32位

- 四字 (QWORD) :

一个四字为四个字宽度, 8个字节, 64位

2. “字” 和 “字长”

通俗的说法：某台机器是32位机或64位机，其中的32、64指的就是字长

—— 字和字长是不同的概念

字长：指CPU内部用于**整数运算**的**数据通路**的宽度（**位数**）。

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**(ALU)、**存储**(寄存器)和**传送**(总线)的部件。

◆ 这些部件的宽度基本上要一致，才能相互匹配。

◆ **字长**等于CPU内部用于整数运算的运算器的位数和通用寄存器的宽度。

字和字长是不同的概念，字是数据长度的度量单位，一个字等于两个字节。字和字长表示的宽度（位数）可以一样，也可不同。

例如，Intel x86体系结构定义“字”的宽度为16位；但从386开始，“字”还是“字”，但字长是32位（**双字**宽度）。64位机字长是4字宽度。

3.数据量的计量单位

- 存储二进制信息时的计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), $1\text{KB}=2^{10}\text{字节}=1024\text{B}$ 千
- “兆字节” (MB), $1\text{MB}=2^{20}\text{字节}=1024\text{KB}$ 百万
- “千兆字节” (GB), $1\text{GB}=2^{30}\text{字节}=1024\text{MB}$ 10亿
- “兆兆字节” (TB), $1\text{TB}=2^{40}\text{字节}=1024\text{GB}$ 万亿

–PB

–EB

注：存储容量用字节的数量度量

硬盘容量被等于为 $10^m\text{B}<2^n\text{B}$



● 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), $1\text{ Tbps} = 10^{12} \text{ b/s} = 1000 \text{ Gbps}$

注：通信带宽可用bit率度量也可用字节速率度量

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

4. 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。计算机底层的**机器级数据表示**必须提供相应的支持。 (这里以C语言为例)
- ◆ 不同机器上表示的同一种类型的数据可能宽度不同。

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	Compaq Alpha 机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

- 数据的字节数随**机器字长**和**编译器**的不同而不同。
- 必须确定相应的机器级数据表示方式和相应的处理指令才能正确地表示和处理相应的数据

Compaq Alpha是一个针对高端应用的64位机器, 即字长为64位

5. 数据的存储和排列顺序

- 机器内部的存储单元用**字节编址**，**线性地从0开始向后编码**。
- 对一个**多字节的数据**，如4字节的整数，每个字节在机器内是如何存放的？

如，设 $\text{int } i = -65535$ ， $[-65535]_{\text{补}} = \text{FFFF0001H}$ 。若 i 存放在100号单元开始的4个字节里，问 4个字节的内容各是什么？



一个数据可能需要用若干字节表示

- 如整数用4字节表示，如：unsigned i = 0xC0A00001;

□ 最高位所在的字节称为**最高有效字节**，记为**MSB** (Most Significant Byte) ；

- 如 “C0” 所在的字节；

□ 最低位所在的字节成为**最低有效字节**，记为**LSB** (Least Significant Byte) ；

- 如 “01” 所在的字节；

(注：这个概念可以推广到“位”上，分别成为最高有效位和最低有效位，也记为MSB和LSB)

□ 数据的每个字节在机器内的存放顺序就是LSB/MSB的顺序。

字节地址:		100	101	102	103	大端方式 小端方式
有两种可能的方式	存储方式1:	FF	FF	00	01	
	存储方式2:	01	00	FF	FF	

大端方式 (Big Endian) : 将数据的LSB存放在最高地址单元中, MSB存放在最低地址单元中。如上面的存储方式1。

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

小端方式 (Little Endian) : 将数据的LSB存放在最低地址单元中, MSB存放在最高地址单元中。如上面的存储方式2。

e.g. Intel 80x86, DEC VAX

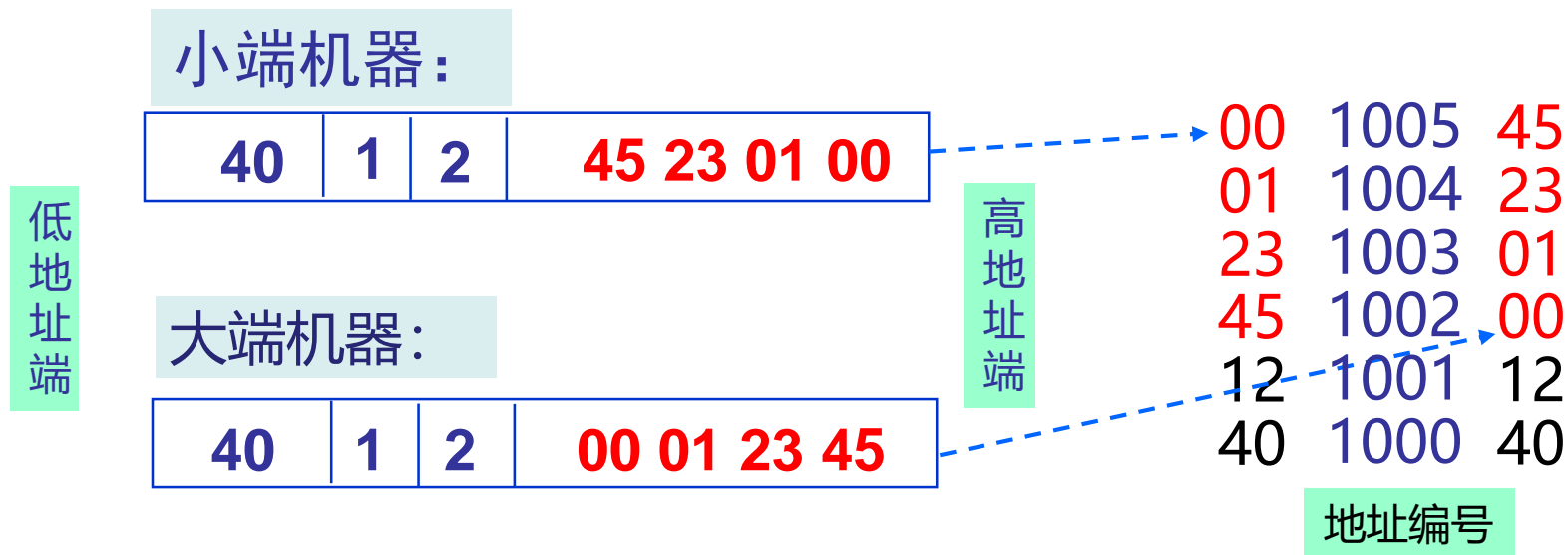
有些机器两种方式都支持, 可通过特定控制位来设定采用哪种方式。

一般, 数据在内存中的地址指**地址最小的字节的地址** (如**上例中的100**) , 所以**大端方式下数据的地址是MSB所在的地址**, **小端方式下数据的地址是LSB所在的地址**。

例: 设内存中下述指令的存放地址是1000

`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, 则内存中存放方式为:

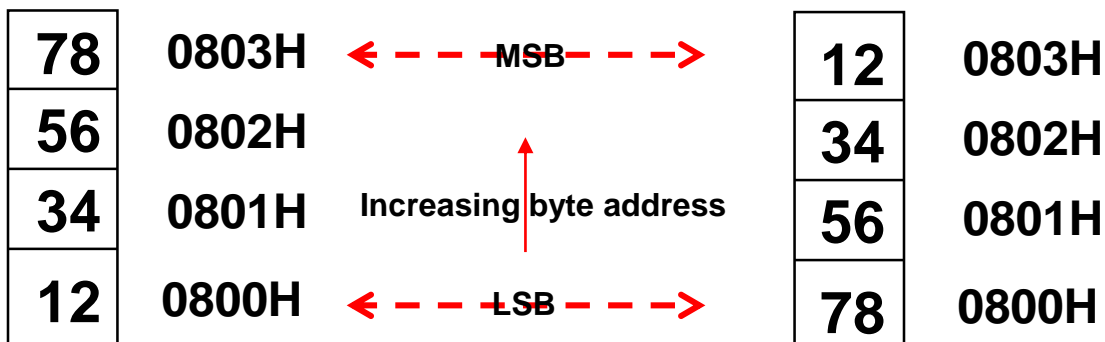


注: 只需要考虑指令中立即数的字节顺序!

字节交换问题

右侧单元中存放的数据是什么？

12345678H? 78563412H?



Big Endian

Little Endian

12345678H

◆ 存放方式不同的机器间程序移植或数据通信时，又会发生什么问题？

- 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

如： **Little endian**: GIF, PC Paintbrush, Microsoft RTF, etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

第三讲：数据的运算

介绍C语言中的运算在底层机器级的实现方法

- ✓ 按位运算、逻辑运算、移位运算、位扩展、位截断运算
- ✓ 无符号和带符号整数的加减乘除运算
- ✓ 变量与常数之间的乘除运算
- ✓ 浮点数的加减乘除运算

2.7 数据的运算

- 高级语言程序中用到的各种运算通过编译成为底层的算术运算指令和逻辑运算指令，然后在机器硬件上被执行。
- **C语言中的各种运算最终被解释成了什么样的机器级指令呢（汇编）？**
- 同时了解基本运算部件ALU的设计和执行。

1. 位运算

- ◆ 位运算是**数值型运算**。
- ◆ C语言的位运算与**汇编位运算指令**一一对应。

C语言 位运算符	含义	对应的汇编指令 位逻辑运算	备注
	按位或	OR	直接对应汇编级的 (位) 逻辑运算
&	按位与	AND	
~	按位取反	NOT	
^	按位异或	XOR	

典型应用：构造掩码

- ◆ 用于屏蔽或提取一个位串中特定的一个子串。

mask = 0x00ff, x = 0x0101

y = x&mask = 0x0001

2. 逻辑运算

- ◆ 逻辑运算是**非数值运算**。
- ◆ 逻辑运算的结果值是true/false，C语言中用1/0表示。
- ◆ 汇编中没有与高级语言逻辑运算直接对应的指令。高级语言程序中的逻辑运算最终被解释成为对左右两个操作数进行非零测试和条件转移指令实现。

C语言 逻辑运算符	含义	对应的汇编指令
	逻辑或	汇编中没有直接对应的指令
&&	逻辑与	
!	逻辑非”	

3. 移位运算

C语言的移位运算（<<、>>）分别与机器级的**左移**或**右移**指令对应。

◆ 移位运算又分为**逻辑移位**：SHL、SHR

算术移位：SAL、SAR

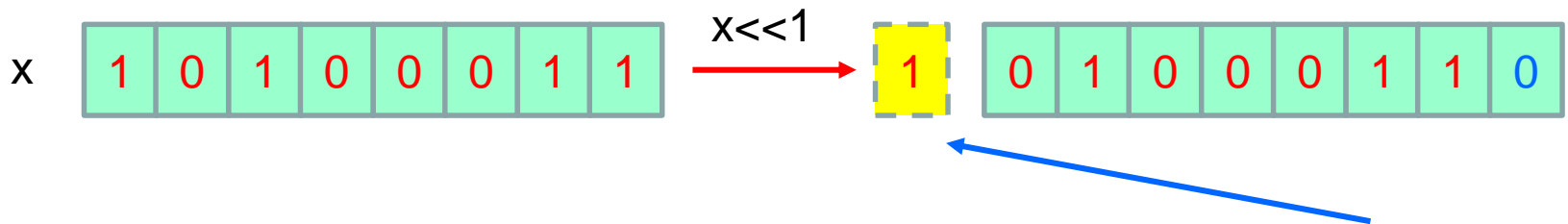
◆ C语言源程序在编译的时候，**编译器根据数据的类型确定是逻辑移位还是算术移位**，并翻译成相应的机器级指令：

➤ **无符号数对应逻辑移位**

➤ **带符号整数对应算术移位**

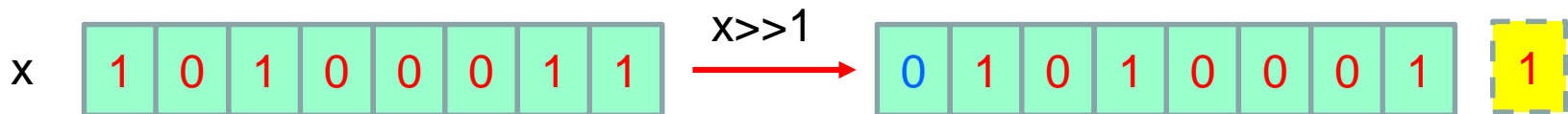
➤ **逻辑移位**：对应**无符号数**的移位运算

➤ **逻辑左移 (SHL)**：高位移出，低位补0；



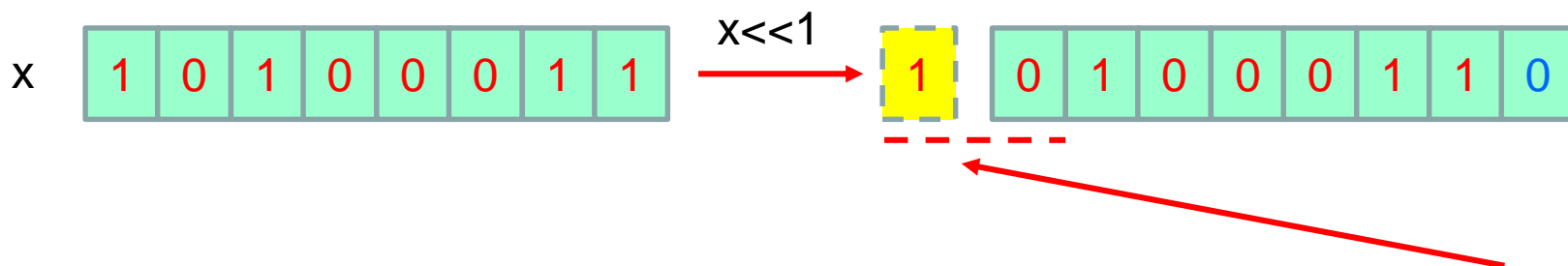
溢出判定：若逻辑左移**移出的高位是1**，则称发生“**溢出**”

➤ **逻辑右移 (SHR)**：低位移出，高位补0；



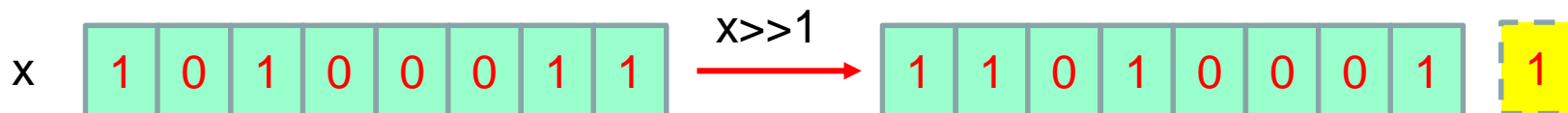
➤ **算术移位**：对应**带符号整数**的移位运算

➤ **算术左移 (SAL)**：高位移出，低位补0；



溢出判定：算术左移后若**符号位不同于移位前**，则发生“**溢出**”

➤ **算术右移 (SAR)**：低位移出，高位补符；



移位操作可以实现整数的乘2或除2操作

◆ **左移**：相当于乘2

- 如果乘2后的结果大于整数的表示范围，则会发生**溢出**。

◆ **右移**：相当于除2（整除）

例：已知32位寄存器IR中存放的变量x的机器数是80 00 00 04H。

问题 1)：当x是unsigned int类型时，x的真值是多少？x/2的值是多少？

x/2的机器数是什么？2x的值是多少？2x的机器数是什么？

解：(1) **x的真值**：x 是unsigned int类型**无符号整数**，所以真值是：

$$+ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100B = 2^{31} + 2^2$$

(2) **x/2的真值**： $(2^{31} + 2^2) / 2 = 2^{30} + 2^1$

另，由**x逻辑右移1位**可以得到机器数：

$$0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010B$$

真值为： $2^{30} + 2^1$

所以，根据x的真值求出的x/2的值与对x的机器数逻辑右移得到的x/2的值是一样的：**逻辑右移等价于除2运算**

(3) **2x的真值：** $(2^{31}+2^2) \times 2 = 2^{32}+2^3$

另，由**x逻辑左移1位**可以得到机器数：

0000 0000 0000 0000 0000 0000 0000 1000B

真值为： **$2^3=8$**

$2^{32}+2^3 \neq 8$ ，说明**结果溢出**，即根据x的真值求出的2x的值与对x的机器数直接**逻辑左移**得到的2x的值不一样，说明2x的值超出最大可表示值 $2^{32}-1$ ，用32位无法表示。（注：根据模运算规则，实际结果等于模 2^{32} 的结果）

非溢出时，逻辑左移等价于乘2运算。

逻辑左移溢出判断：移除的最高位是否为1，为1则表示溢出

例：已知32位寄存器IR中存放的变量x的**机器数是80 00 00 04H**。

问题 2)：当x是int类型时，x的值是多少？x/2的值是多少？x/2的机器数是什么？2x的值是多少？2x的机器数是什么？

解：(1) **x的真值**：x是int类型时，是**带符号数**，机器数80 00 00 04H最高位为1，所以是负数的补码，真值是：

$$\text{- 0111 1111 1111 1111 1111 1111 1111 1100B} = -(2^{31}-2^2)$$

$$(2) \text{ x/2的真值: } -(2^{31}-2^2)/2 = - (2^{30}-2^1)$$

而由**x算术右移1**位可以得到机器数：

$$\text{1100 0000 0000 0000 0000 0000 0000 0010B}$$

$$\text{真值为 (补码还原) : } -(2^{30}+2^1)$$

所以，根据x的真值求出的x/2的值与对x的机器数算术右移得到的x/2的值是一样的：**算术右移等价于除2运算**。

(3) **2x的真值**: $-(2^{31}-2^2) \times 2 = -(2^{32}-2^3)$

而由**x算术左移1位**可以得到机器数:

0000 0000 0000 0000 0000 0000 0000 1000B

真值为: $2^3=8$

$-(2^{32}-2^3) \neq 8$, 说明**结果溢出**。即根据x的真值求出的2x的值与对x的机器数直接算术左移得到的2x的值不一样, 说明2x的真值 $-(2^{32}-2^3)$ 比最小可表示值 -2^{31} 还要小, 用32位无法表示。

(注: 根据模运算规则, 实际结果等于模 2^{32} 的结果)

非溢出时, 算术左移等价于乘2运算。

算术左移溢出判断: 若移位前、后符号位不同, 则表示溢出

4. 位扩展和位截断运算

根据C语言赋值操作的规则，若将**位数（字节数）少**的数据赋给**位数（字节数）多**的变量时，发生**数据扩展**，反之进行**数据截断**。

如：short si = -32768;

int i;

i = si; //将两字节数据扩展成4字节数据

si = i; //将四字节数据截断成2字节数据

机器级使用**位扩展传送指令**和**mov指令**实现位扩展和位截断运算。

◆ 位扩展传送汇编指令：将位数少的数扩展成为位数多的数

➤ 0扩展传送指令：高位补0

movzbw、movzbl、movzwl、movzbq、movzwq

➤ 符号扩展传送指令：高位补符号位

movsbw、movsbl、movswl、movsbq

◆ AT&T格式

◆ 机器级的截断操作直接用传送指令mov实现。

注：

◆ 位数相同的数据传送时不存在扩展问题；

直接用mov、movb、movw、movl、movq完成

◆ C语言中没有专门的位扩展/截断运算，数据扩展/截断发生在数据类型转换过程中。

AT&T: 美国电话电报公司
贝尔实验室 \in AT&T

◆ 这里使用的汇编是**AT&T格式**

- 不同于MASM, MASM采用的是**Intel格式**。
- AT&T格式是linux下gcc和objdump使用默认格式。

◆ 典型的特点

- **指令右面带后缀, 表示操作数的长度:**

- b: 1字节/8位; w: 2字节/16位; l: 双字/32位; q: 四字/64位

如: movzbw、movzbl、movzwl、movzbq、movzwq

- **寄存器操作数的形式: % + 寄存器名**

如%eax, 表示寄存器EAX的内容, 即R[`eax`]

- **存储器操作数的形式: 偏移量(基址寄存器, 变址寄存器, 比例因子)**

如: 100(%ebx,%esi,4), 表示: $M[R[ebx] + 4R[esi] + 100]$

- **汇编指令的基本形式: op src, dst, 含义为 “dst \leftarrow dst op src”**

如, addl(, %ebx,2), %eax 含义为: $R[`eax`] \leftarrow R[`eax`] + M[2 * R[`ebx`]]$

扩展操作的例子：

设有定义（大端表示）：

```
short si = -32768;
```

```
unsigned short usi = si;    //类型转换，等长，不扩展或截断
```

```
int i = si;                //2字节到4字节扩展，符号扩展
```

```
unsigned ui = usi;         // 2字节到4字节扩展，0扩展
```

变量名称	类型	十进制表示	十六进制表示	扩展类型	汇编指令	备注
si	带符号短整型	-32768	80 00	/		2字节
usi	无符号短整型	32768	80 00	/	mov	2字节
i	带符号整型	-32768	FF FF 80 00	符号扩展	movswl	4字节
ui	无符号整型	32768	00 00 80 00	0扩展	movzwl	4字节

movswl: 将做了符号扩展的字传送到双字

movzwl: 将做了零扩展的字传送到双字

截断操作的例子：

设有定义如下（大端表示）

```
int i = 32768;
short si = (short)i; //截断
int j = si;           //扩展
```

机器级，截断操作直接用
传送指令（mov）实现

变量名称	类型	十进制表示	十六进制表示	扩展类型	汇编指令	备注
i	带符号整型	32768	00 00 80 00	/	/	2字节
si	带符号短整型	-32768	80 00	/	mov	2字节
j	带符号整型	-32768	FF FF 80 00	符号扩展	movswl	4字节

说明：原来的i（值为32768）经过截断、扩展后，其值变为了-32768。

截断带来的问题：

1) 截断一个数可能会改变初值。

截断溢出：当长数的值超出了短数的表示范围，截断时就会发生溢出（有效字节丢失），称为**截断溢出**。

2) 因为截断而发生错误，引起**截断错误**。

- ◆ 截断错误会导致程序出现意外结果，但一般又不会导致系统异常和错误报告，因此，错误的隐蔽性很强，编程的时候需要特别注意。

C语言还有哪些基本运算？

关系运算：对应**cmp**、**jmp**指令

赋值运算：对应**mov**或**位扩展**指令

条件运算：复合运算，由**关系运算**+**条件转移**合成

第四次课 (4.29)

浮点数编码：IEEE 754标准

规格化数： $+/-1.\text{XXXXXXXXXX}_{\text{two}} \times 2^{\text{Exponent}}$

Single/Double Precision(SP/DP, 单/双精度浮点数, 32/64位)



非规格化数的定义：

- ± 0 : 全0阶码+全0尾数, 00000000H或80000000H
- 非规格化数: 阶码全0、尾数非零, $(-1)^s \times 0.f \times 2^{-126}$
- $\pm \infty$: 指数全1、尾数全0, 7F800000H或FF800000H
- NaN: 阶码全1, 尾数非零, 尾数最高有效位为1(quiet), 尾数最高有效位为0(signaling)

十进制数的编码表示

- 字符串的表示方法
- BCD码：有权BCD码（8421码）、无权BCD码（余3码、格雷码等）

非数值数据的编码表示

- 逻辑值的编码表示：用1位表示个逻辑值，
True (1) 或False (0)
- 西文字符的编码表示：ASCII码
- 汉字的编码表示：GB2132-80, 6763个汉字
区位码+2020H=国标码+8080H=机内码
- ISO/IEC10646/Unicode：大字符集的国际编码

数据的宽度

- 位、字节、字、双字、四字等
- 字长：CPU内部用于整数运算的数据通路的宽度
- 数据量的度量单位：K、M、G、T、P
- 存储容量的基本单位：字节；通信带宽的一般单位：位

数据存储的字节排列

- **大端方式**：最低有效字节存放在高地址单元，MSB所在的地址是数的地址（字面看起来是顺的）
- **小端方式**：最低有效字节存放在低地址单元，LSB所在的地址是数的地址（字面看起来是反的）

int i = -65535, [-65535]补=FFFF0001H。若i存放在100号单元开始的4个字节里

	lsB			msB	
字节地址:	100	101	102	103	
存储方式1:	FF	FF	00	01	big endian word 100#
存储方式2:	01	00	FF	FF	little endian word 100#

还有2.8运算的部分内容

2.7 数据的运算

C语言中的各种运算与汇编指令之间的对应关系

1-1对应的:

- C语言位运算 (&、|、~、^) : 对应汇编的逻辑运算
(AND、OR、NOT、XOR)
- C语言移位运算 (<<、>>) : 对应汇编的移位运算
 - 无符号数: 对应逻辑移位, SHL、SHR;
 - 带符号数: 对应算术移位, SAL、SAR
- 移位操作可以实现乘2 (左移) 或除2 (右移) 操作:
 - 有高位溢出和低位丢失现象

- 位扩展和位截断运算：直接用汇编传送指令mov实现

- 位扩展：将位数少的数扩展为位数多的数

发生在C语言变量类型提升的时候

- 0扩展：movzbw、movzbl、movzwl、movzbq、movzwq

- 符号扩展：movsbw、movsbl、movswl、movsbq

- 截断操作：mov，把位数多的数截断为位数少的数。

发生在“长数”向“短数”赋值的时候。

截断有截断溢出和截断错误的现象发生。

1: 多的“复合”运算

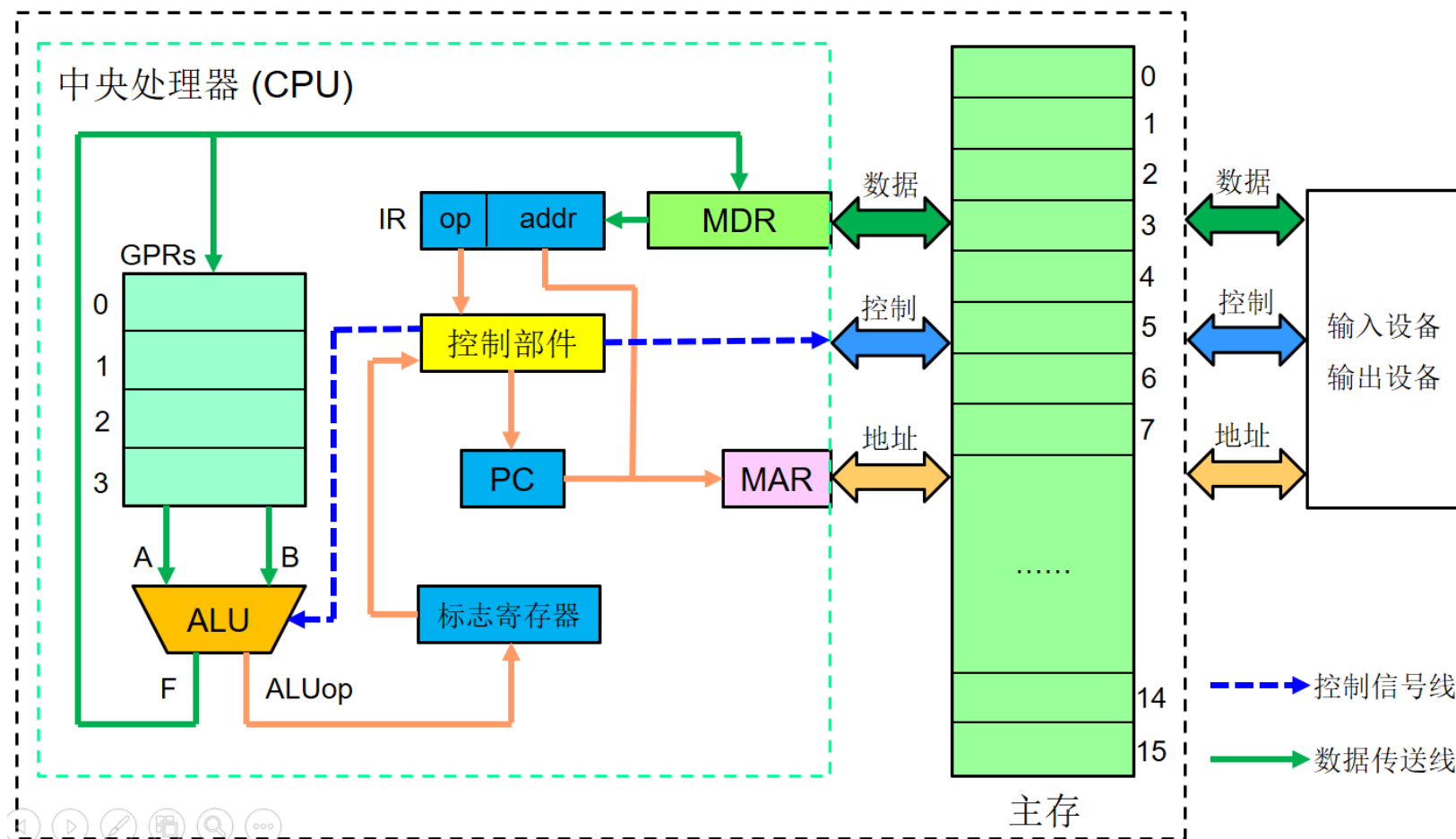
- C语言逻辑运算（&&、||、!）：用汇编级的比较指令和条件转移（CMP + JMP）的混合语句来实现
- C语言关系运算（<、<=、>、>=、==、!=）：用汇编级的比较指令和条件转移（CMP+JMP）的混合语句来实现

需要注意的问题：计算内部数据都是用有限的位数表示的(16/32/64位等)，因此而发生数位的上溢出或下丢失问题。

5. 整数加减运算

- ◆ 整数包括无符号整数和带符号整数。
- ◆ 在现代计算机系统中，**无符号整数和带符号整数的加减运算实现方式完全一样——在同一个加减运算电路（整数加减运算器）上实现。**

回顾：冯.诺依曼计算机结构模型



CPU: 中央处理器; PC: 程序计数器; MAR: 存储器地址寄存器
ALU: 算术逻辑部件; IR: 指令寄存器; MDR: 存储器数据寄存器
GPRs: 通用寄存器组 (由若干通用寄存器组成, 早期就是累加器)

预习：IA-32的定点寄存器组

	31	16	15	8	7	0	
EAX				AH	(AX)	AL	累加器
EBX				BH	(BX)	BL	基址寄存器
ECX				CH	(CX)	CL	计数寄存器
EDX				DH	(DX)	DL	数据寄存器
ESP				SP			堆栈指针
EBP				BP			基址指针
ESI				SI			源变址寄存器
EDI				DI			目标变址寄存器
EIP				IP			指令指针
EFLAGS				FLAGS			标志寄存器

8个通用寄存器

32位，可以根据操作数长度分别存取寄存器的最低8位、最低16位或全部32位

两个专用寄存器

IP：用来存放即将执行的指令的地址（作用同PC）

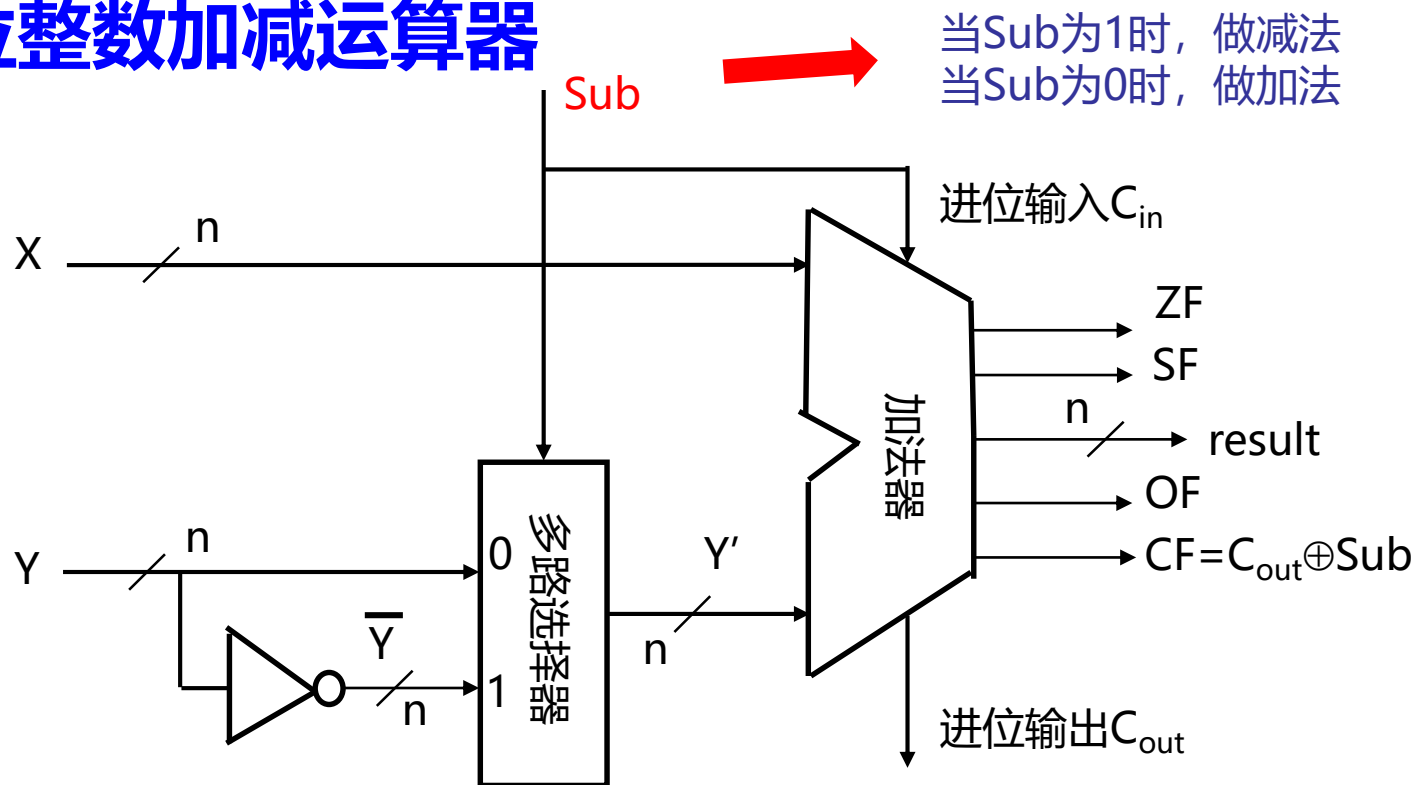
6个段寄存器

CS
SS
DS
ES
FS
GS

8个通用寄存器：可以存放各类定点操作数的。

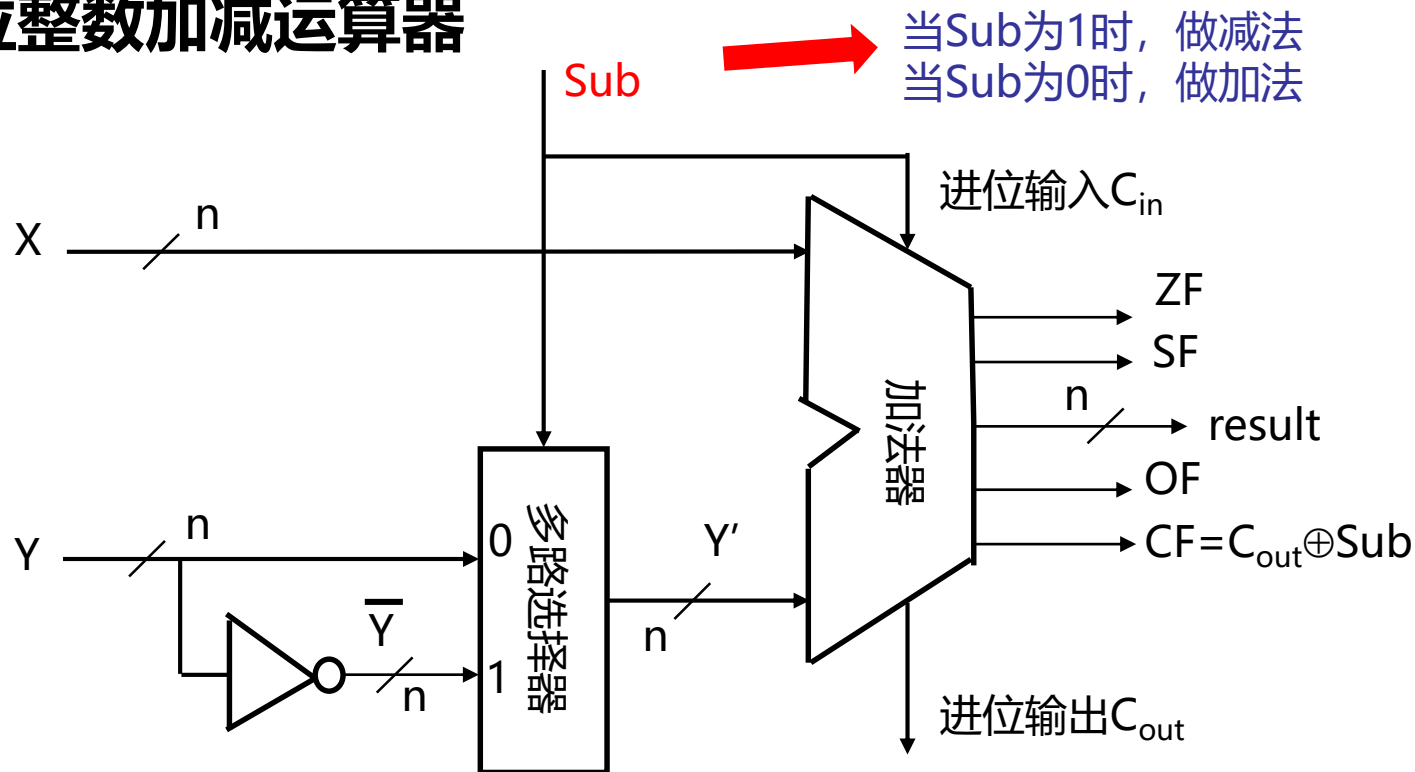
6个段寄存器：CPU根据段寄存器的内容，与寻址方式确定的有效地址一起，并结合其他用户不可见的内部寄存器，生成操作数所在的存储地址。

n位整数加减运算器



- ◆ X、Y：两个n位的0/1序列，为输入x、y的补码（正数的原码 或 模-|负数|）。
- ◆ **Sub**：操作控制信号，Sub=0做加法，Sub=1做减法
- ◆ 不管是带符号减法还是无符号数减法，减法运算都是用被减数X加上减数Y的负数的补码来实现。
 - 减数的负数的补码：Y取反+1。
 - **反向器**对Y的各位取反，并将**Sub**作为加法器进位送到加法器，当Sub=1时，实现 $x-y = X + \overline{Y} + 1$ 。当Sub=0时，实现 $x+y = X + Y$ 。

n位整数加减运算器



标志位

- ◆ **ZF**: 零标志, 如果结果为0, ZF置1, 否则ZF=0。
- ◆ **OF**: 溢出标志, **当X和Y' 的最高位相同但不同于结果的最高位时**, OF置1。
OF标志只对带符号数有意义, OF=1表示发生了溢出; 否则表示没有溢出。
- ◆ **SF**: 符号标志, **SF=result的最高位**。
SF只对带符号数有意义, SF=1表示结果为负数; 否则为非负数。
- ◆ **CF**: 进/借位标志, 加法时, CF=1表示有进位;
减法时, CF=1表示有借位, $CF = Sub \oplus C_{out}$ 。 (\oplus 异或)

用“系统思维”分析问题

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

当参数len为0时，似乎返回值应该是0，但是执行时却发生访存异常。

但当len为int型时则正常。Why?

分析：根本原因在于其中包含的**两次减法运算**在上述电路中的计算结果“不如预期”所致。

访问违例地址为何是0xC0000005?



Unhand

哪里**有两次减法运算**?

Access Violation.

确定

① 由于len为unsigned类型，所以第一个减法运算：“len-1”按无符号数进行。

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

翻译成汇编指令

```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jbe .L3
    ...
```

取i

取len

i 在%eax中， len在%edx中

%eax: 0000 0000

%edx: 0000 0000

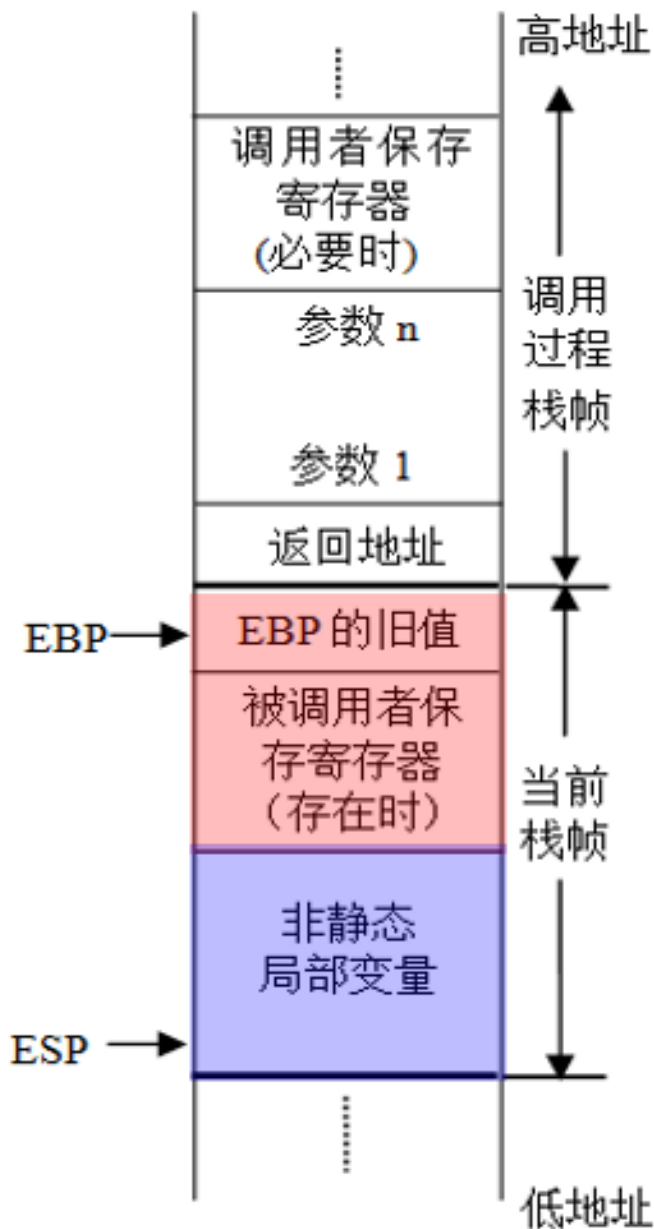
subl 指令的执行结果是什么？

① E

第一个减法运算: "**len-1**" 按无符号数进行。

sum
{
if
r
}

i
%
%



指令

sum:

...
.L3:

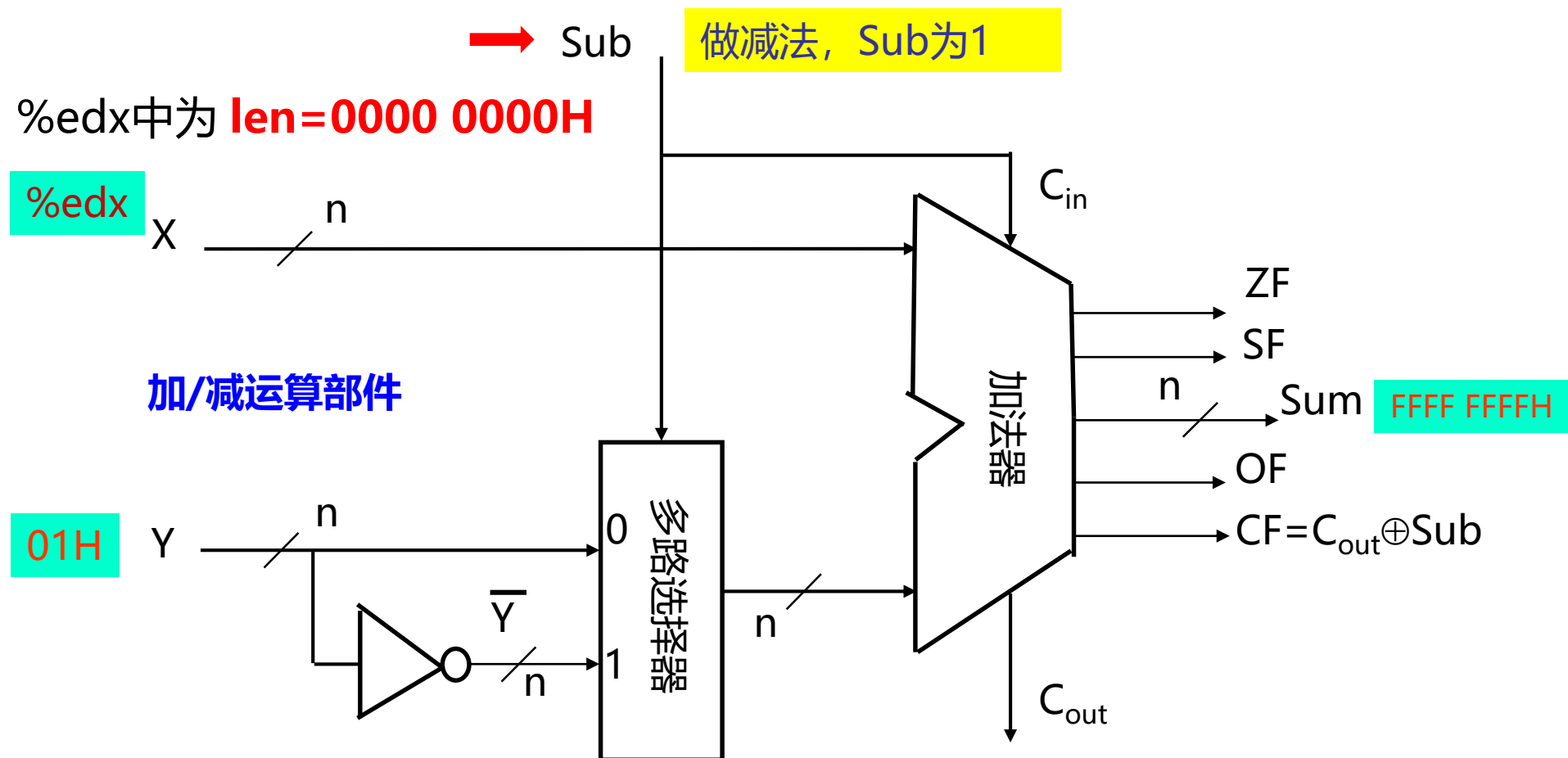
movl -4(%ebp), %eax
movl 12(%ebp), %edx
subl \$1, %edx
cmpl %edx, %eax
jbe .L3

取i

取len

栈帧: 每个函数都有自己的一个栈空间称为“栈帧”, 用于存放局部变量等, ebp是栈帧的底, 并且从高地址向低地址生长。

计算len-1: subl \$1, %edx指令的执行结果



执行 “**subl \$1, %edx**” : $X=0000\ 0000H$, $Y=0000\ 0001H$, $Sub=1$,
 $[-Y]_{补} = FFFF\ FFFFH$, 因此Sum是32个1, 即**%edx中为FFFF FFFFH**。

② $i \leq \text{len}-1$: 对应的是比较和转移指令 (cmp+jmp), 先在运算器上执行一个**减法运算**, 然后根据标志 (CF) 来判断大小——第二个减法运算。

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

```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jbe .L3
    ...
```

i 在eax中, len-1的结果在edx中:

%eax: 0000 0000

%edx: FFFF FFFF

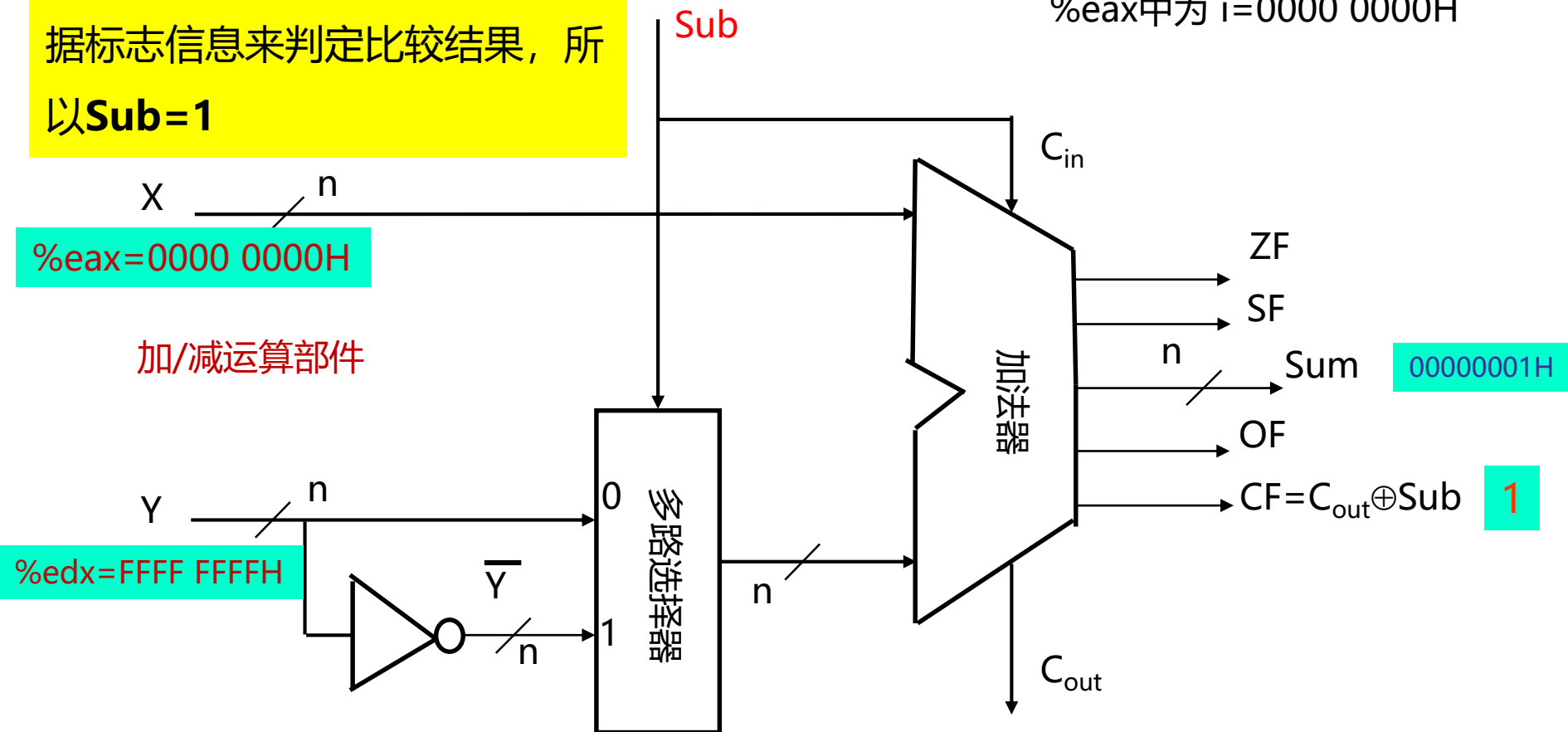
无符号跳转指令

同时, 由于len-1当作无符号整数处理, 所以对应的条件转移指令是jbe

$i < len-1$: `cmpl %edx,%eax`指令的执行结果

比较指令是通过做减法，然后根据标志信息来判定比较结果，所以**Sub=1**

已知%edx中为 $len-1 = \text{FFFF FFFFH}$
%eax中为 $i = \text{0000 0000H}$



“`cmpl %edx,%eax`” 执行时，隐含一个**减法运算**：X=00000000H，Y为FFFFFFFFH，Sub=1，减运算的结果是：**Sum=0...01**，**C_{out}=0**，而**CF=Sub⊕C_{out}=1**，ZF=0，OF=0，SF=0

③ 由于len为unsigned类型，所以对“ $i \leq \text{len}-1$ ”的判断按无符号数进行比较，对应的转移指令是无符号转移指令。

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

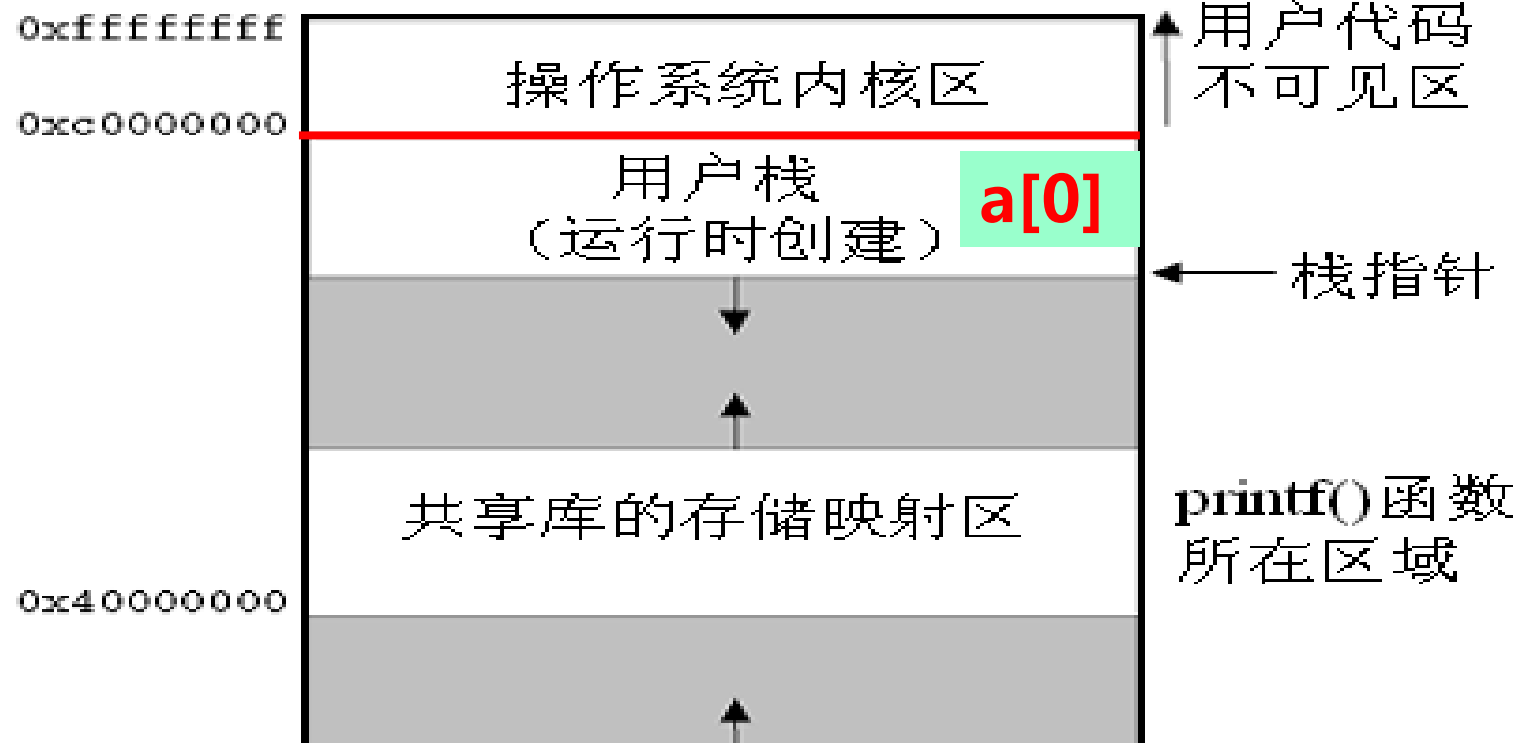
```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jbe .L3
    ...
```

无符号跳转指令

jbe .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0 OR ZF=1	无符号数A ≥ B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A ≤ B
JG/JNLE label	SF=OF AND ZF=0	有符号数A > B
JGE/JNL label	SF=OF OR ZF=1	有符号数A ≥ B
JL/JNGE label	SF≠OF AND ZF=0	有符号数A < B
JLE/JNG label	SF≠OF OR ZF=1	有符号数A ≤ B

“cmpl %edx,%eax” 执行结果是 **CF=1**, **ZF=0**, OF=0, SF=0, 说明满足条件（小于等于，为真），应转移到.L3执行！ 显然，对于每个i都满足条件，因为期间任何i的值都比32个1小，因此循环体被不断执行，最终导致**数组访问越界而发生存储器访问异常（访问到了内存保护区）**。



Microsoft Visual C++

X



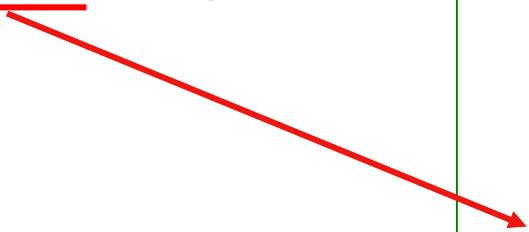
Unhandled exception in Test.exe: 0xC0000005: Access Violation.

确定

但若len为int类型，对应的转移指令是带符号转移指令**jle**

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

```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jle .L3
    ...
```



JLE/JNG label

SF≠OF OR ZF=1

有符号数A≤B

◆ **cmpl %edx,%eax** 执行结果是 **CF=1, ZF=0, OF=0, SF=0**。

所以，**jle**为false，就会退出循环。

整数加减运算的规则：

◆ 无符号数：

n位无符号整数的表示范围是 $0 \sim 2^n - 1$ ，不管是加法还是减法，对超过范围的结果，都要按照模运算或补码规则进行处理。

➤ 加法：
$$result = \begin{cases} x + y & (x + y < 2^n) \\ x + y - 2^n & (2^n \leq x + y < 2^{n+1}) \end{cases} \quad \text{公式(2.1)}$$

结果大于 2^n 时，取模 2^n 的余数

➤ 两个n位二进制数相加，结果可能有n+1位，但加运算的结果只取**低n位**，相当于模 2^n 运算。如果结果超过n位，则有溢出。

➤ 减法：
$$result = \begin{cases} x - y & (x - y > 0) \\ x - y + 2^n & (x - y < 0) \end{cases} \quad \text{公式(2.2)}$$

结果小于0时，取负数的补

➤ 减法运算用 $x + [-y]_{\text{补}}$ 来实现，所以有 $result = x + (2^n - y) = x - y + 2^n$

例2.29 假设8位无符号整数变量x和y的机器数分别是X和Y，在上面的电路中执行。

1) 若X=A6H, Y=3FH, 问x, y, x+y, x-y的值分别是多少?

解: x+y的机器数: $X+Y=1010\ 0110B + 0011\ 1111B=1110\ 0101=E5H$

x-y的机器数: $X-Y=1010\ 0110B + 1100\ 0001B=0110\ 0111B=67H$

因为是无符号整数运算，所以有：

$$x=A6H=166_{10}, \quad y=3FH=63_{10}$$

$$x+y=229, \quad x-y=103.$$

符合公式(2.1)和(2.2)

$$result = \begin{cases} x+y & (x+y < 2^n) \\ x+y-2^n & (2^n \leq x+y < 2^{n+1}) \end{cases} \quad result = \begin{cases} x-y & (x-y > 0) \\ x-y+2^n & (x-y < 0) \end{cases}$$

2) 若 $X=A6H$, $Y=FFH$, 问 x , y , $x+y$, $x-y$ 的值分别是多少?

解: $x+y$ 的机器数: $X+Y=1010\ 0110+1111\ 1111=1\ 1010\ 0101=A5H$

丢弃

$x-y$ 的机器数: $X-Y=1010\ 0110+0000\ 0001=1010\ 0111=A7H$

同样, 因为是无符号整数运算, 所以有:

$$x=A6H=166_{10}, \quad y=FFH=255_{10}$$

$$x+y=165, \quad x-y=167.$$

符合公式(2.1)和(2.2)

注: 因为 $166-255<0$,
所以按公式2.2亦有:

$$\begin{aligned} & x-y+2^8 \\ &=166-255+256 \\ &=167 \end{aligned}$$

$$result = \begin{cases} x+y & (x+y < 2^n) \\ x+y-2^n & (2^n \leq x+y < 2^{n+1}) \end{cases}$$

$$result = \begin{cases} x-y & (x-y > 0) \\ x-y+2^n & (x-y < 0) \end{cases}$$

◆带符号数：带符号整数加法运算同样在上述电路中执行

- n位带符号整数的表示范围是： $-2^{n-1} \sim 2^{n-1}-1$ 。超出范围的结果为溢出。

正溢出（大于 $2^{n-1}-1$ ）减 2^n ，**负溢出**（小于 -2^{n-1} ）加 2^n 。

➤ 加法：

$$result = \begin{cases} x + y - 2^n & (2^{n-1} \leq x + y) & \text{正溢出} \\ x + y & (-2^{n-1} \leq x + y < 2^{n-1}) & \text{正常} \\ x + y + 2^n & (x + y < -2^{n-1}) & \text{负溢出} \end{cases}$$

公式(2.3)

溢出问题：

- ◆ 如果两个数的符号相反，则一定不会溢出（相当于做减法，只会变小）；
- ◆ 如果两个数的符号相同，则和可能会溢出。结果为正时发生的溢出称为**正溢出**，结果为负时发生的溢出称为**负溢出**。

➤ 减法：

$$result = \begin{cases} x - y - 2^n & (2^{n-1} \leq x - y) & \text{正溢出} \\ x - y & (-2^{n-1} \leq x - y < 2^{n-1}) & \text{正常} \\ x - y + 2^n & (x - y < -2^{n-1}) & \text{负溢出} \end{cases}$$

公式(2.4)

➤ 减法运算用 $x + [-y]_{\text{补}}$ 来实现。

例2.30 假设8位带符号整数变量x和y的机器数分别是X和Y，在上面的电路中执行。

1) 若X=A6H, Y=3FH, 问x, y, x+y, x-y的值分别是多少?

解: **x+y的机器数**: $X+Y=1010\ 0110+0011\ 1111=1110\ 0101=E5H$

x-y的机器数: $X-Y=10100110+1100\ 0001=1\ 01100111=67H$

由于是带符号数运算，所以有：

$$x = -90, \quad y = 63,$$

$$x+y = -27, \quad x-y = 103 \quad (-90-63+256)。$$

符合公式(2.3)和(2.4)

$$result = \begin{cases} x + y - 2^n & (2^{n-1} \leq x + y) & \text{正溢出} \\ x + y & (-2^{n-1} \leq x + y < 2^{n-1}) & \text{正常} \\ x + y + 2^n & (x + y < -2^{n-1}) & \text{负溢出} \end{cases}$$

$$result = \begin{cases} x - y - 2^n & (2^{n-1} \leq x - y) & \text{正溢出} \\ x - y & (-2^{n-1} \leq x - y < 2^{n-1}) & \text{正常} \\ x - y + 2^n & (x - y < -2^{n-1}) & \text{负溢出} \end{cases}$$

真值: 103

负溢出: $-90-63 < -128$

机器码是补码，换算回来后

(1) x是负数，等于-90；

(2) $x+y = -90+63 = -27$

(3) $x-y = -90-63 = -153 < -128$

2) 若 $X=A6H$, $Y=FFH$, 问 x , y , $x+y$, $x-y$ 的值分别是多少?

解: $x+y$ 的机器数: $X+Y=1010\ 0110+1111\ 1111=1\ 1010\ 0101=A5H$

$x-y$ 的机器数: $X-Y=1010\ 0110+0000\ 0001$
 $=1010\ 0111=A7H$

真值: -91

$-90 + -1 = -91$

无溢出: 符号未变

真值: $-90 - (-1) = -89$

由于是带符号数运算, 所以有:

$x=A6H=-90$, $y=FFH=-1$,

$x+y=-91$, $x-y=-89$

符合公式(2.3)和(2.4)

$$result = \begin{cases} x + y - 2^n & (2^{n-1} \leq x + y) & \text{正溢出} \\ x + y & (-2^{n-1} \leq x + y < 2^{n-1}) & \text{正常} \\ x + y + 2^n & (x + y < -2^{n-1}) & \text{负溢出} \end{cases}$$

$$result = \begin{cases} x - y - 2^n & (2^{n-1} \leq x - y) & \text{正溢出} \\ x - y & (-2^{n-1} \leq x - y < 2^{n-1}) & \text{正常} \\ x - y + 2^n & (x - y < -2^{n-1}) & \text{负溢出} \end{cases}$$

前面的例子中，不管是无符号整数还是带符号整数，X和Y的机器数完全相同并在同样的电路中计算，得到的和（差）的机器数也完全相同。然后按照数据类型对结果进行解释而得到不同的值。**所以在电路中执行运算时，所有的数都只是一个0/1序列，无所谓类型。**

因此，在一些ISA中，整数的加法和减法指令不区分符号，如Intel X86指令集。但有的ISA区分无符号整数加（减）指令和带符号整数的加（减）指令，

- 如MIPS架构的指令系统中，提供专门的无符号整数加（减）指令（如addu、subu）和专门的带符号整数的加（减）指令（add、sub）



-
- 同时，由于在机器级底层对无符号数和带符号数的运算不加区分，而在高级语言程序执行中，**带符号数隐式地转换为无符号数，会发生一些意想不到的错误或存在漏洞。**因此，为避免此类问题，有些语言不支持无符号整数。

➤ 如Java。



6. 整数的乘运算

- 按照数学运算规则，两个 n 位整数相乘得到的结果可能是 $2n$ 位的整数。
- 但在计算机中，两个“ n 位”整数相乘得到的实际结果还是一个 n 位整数，即结果只取 $2n$ 位乘积中的低 n 位。（ n =字长）

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

若 x 、 y 都是32位整数， $x*y$ 被转换为乘法指令，在乘法运算电路得到的乘积是64位，但是，只取其低32位赋给 z 。

- 存在的问题：数据溢出！

二进制乘法

数学规则



乘法电路实现

$$\begin{array}{r} 101010 \quad \text{被乘数} \\ \times 1011 \quad \text{乘数} \\ \hline 101010 \\ 101010 \\ 000000 \\ + 101010 \\ \hline 1110011110 \quad \text{结果} \end{array}$$

部分积

图 1-1 二进制乘法计算

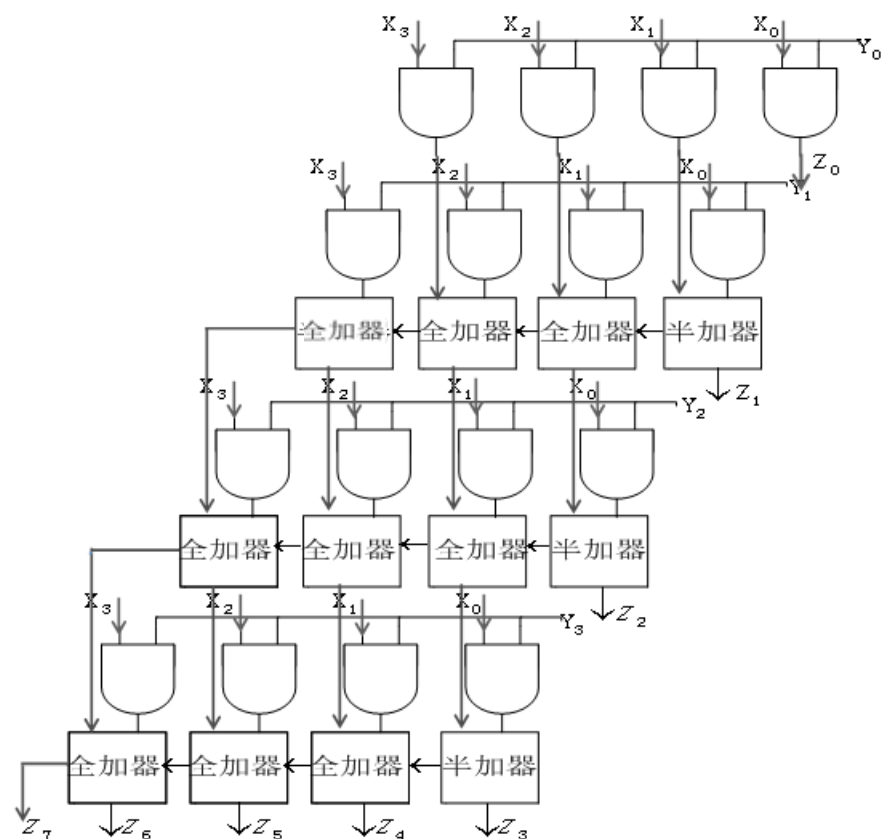


图 1-3 阵列乘法器

机器级乘法指令的操作数长度为 n ，乘积长度为 $2n$ ：

- ◆ IA-32中，若指令有一个操作数SRC（如mulb %bl或imulb %bl），另一个源操作数在累加器AL（8位）/AX（16位）/EAX（32位）中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。（注：整数乘法指令MUL / IMUL的操作数可以为出一、两或三个）
- ◆ 在MIPS处理器中，mult会将两个32位带符号整数相乘，得到的64位乘积置于两个32位内部寄存器Hi和Lo中。

存在的问题：

- 乘法指令本身不判溢出，仅保留 $2n$ 位乘积，供软件使用（如截断为 n 位）。
- 如果程序不采用防止溢出的措施，且编译器也不生成用于溢出处理的代码，就会发生一些由于整数溢出而带来的问题，如： $x^2 \leq 0$ 。

■ 但乘法指令可生成**溢出特征标识**，编译器可使用**2n位乘积的高n位**来判断是否溢出！

- **符号数乘溢出判断**：高位寄存器中的每一位是否等于低位寄存器中的第一位，相同则为无溢出，否则为溢出。
- **无符号数乘溢出判断**：高位寄存器中的所有位是否都等于0，都为0为无溢出，否则而溢出。

■ 也或者通过**反向计算**进行判定：

- 当 **!x 且 $z/x==y$** 时无溢出，否则溢出。

如：在计算机内部，一定有 $x^2 \geq 0$ 吗？

若 x 是带符号整数，则不一定！
如 x 是浮点数，则一定！

例如，当 $n=4$ 时， $5^2 = -7 < 0$ ！

0101	
× 0101	
0101	
+ 0101	
00011001	结果溢出

只取低4位，值为-111B=-7

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

若 x 、 y 和 z 都改成`unsigned`类型，
则判断方式为：乘积的高 n 位为全0，
则不溢出

如何判断返回的 z 是正确的值？

当 $!x$ 且 $z/x == y$ 为真时正确

什么情况下，乘积是正确的呢？

有符号：当 $-2^{n-1} \leq x*y < 2^{n-1}$ （不

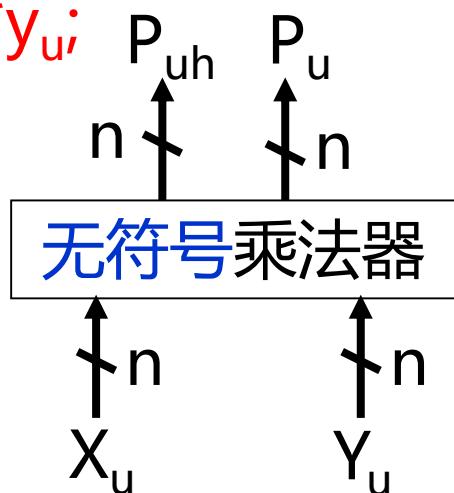
溢出）时，即：乘积的高 n 位为全0或全1，并等于低 n 位的最高位！

即：乘积的高 $n+1$ 位为全0或全1

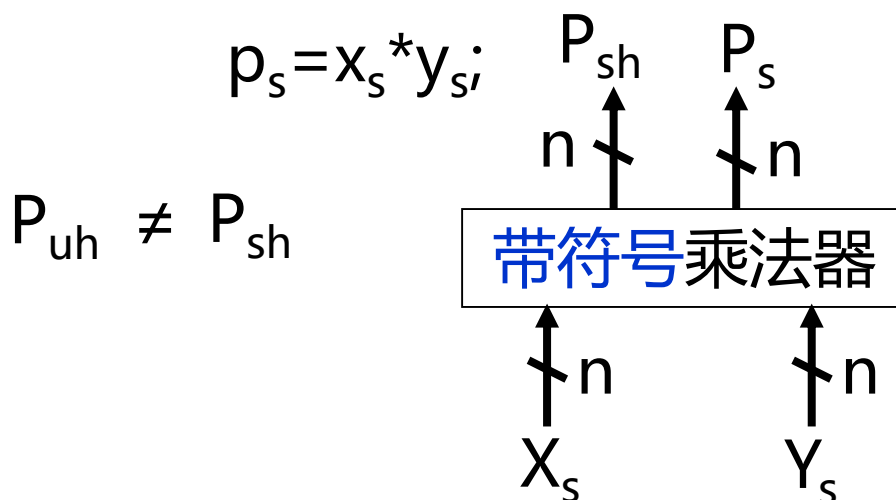
二进制乘法的性质(P65~66, 自学)

- 假定两个n位无符号整数 x_u 和 y_u 对应的机器数为 X_u 和 Y_u ,
 $p_u = x_u \times y_u$, p_u 为n位无符号整数且对应的机器数为 P_u ;
- 而另, 两个n位带符号整数 x_s 和 y_s 对应的机器数为 X_s 和 Y_s ,
 $p_s = x_s \times y_s$, p_s 为n位带符号整数且对应的机器数为 P_s 。
- 若 $X_u = X_s$ 且 $Y_u = Y_s$, 则 $P_u = P_s$ 。

$$p_u = x_u * y_u;$$



$$p_s = x_s * y_s;$$



自学

运算	x	X	y	Y	$x \times y$	$X \times Y$	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	<u>0000</u> 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	<u>1111</u> 1000	-8	1000	不溢出

- $X \times Y$ 的高n位可以用来判断溢出（结果超出n位表示范围），规则如下：
 - 无符号：若高n位全0，则不溢出，否则溢出
 - 带符号：若高n位全0或全1且等于低n位的最高位，则不溢出。

7. 整数的除运算

- 对于带符号整数来说， n 位整数除以 n 位整数不会发生溢出，除非是： $-2^{n-1}/-1 = 2^{n-1}$ 会发生溢出外（Why?）。
 - 因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞。
- 因为整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即
 - 正数商取比自身小的最接近整数（Floor，地板）
如： $7/2=3$
 - 负数商取比自身大的最接近整数（Ceiling，天板）。
如： $-7/2=-3$

-
- 整数除0的结果可以用什么机器数表示？

- ◆ 整数除0的结果无法用一个机器数表示！

- ◆ 整数除法时除数不能为0，否则会发生“异常”，此时需要调出操作系统中的异常处理程序来处理。

- ◆ 回顾：对比浮点数除0和整数除0

变量与常数之间的乘、除运算

- **变量与常数之间的乘**：使用乘法电路，一次乘法运算需要多个时钟周期（ ≥ 10 ）才能完成。
 - 相比较而言，一次移位、加法和减法等运算只要一个或更少的时钟周期，**整数乘法运算比移位和加法等运算所用时间长的多。**

为了优化乘法，编译器在处理**变量与常数**相乘时，往往**以移位、加法和减法的组合运算**来代替乘法运算。

例如，表达式 $x*20$ ，编译器可以利用 $20=16+4=2^4+2^2$ ，

将 $x*20$ 转换为 $(x \ll 4) + (x \ll 2)$ 。

即：一次乘法由了两次移位和一次加法实现。

- 注：不管是无符号整数还是带符号整数的乘法，利用**移位和加减运算**组合的方式得到的结果都和采用直接相乘得到的结果一样，即使乘积有溢出。

变量与常数之间的除运算

由于计算机中除法运算比较复杂，而且不能用流水线方式实现，所以一次除法运算大致需要**30个或更多个时钟周期**，比乘法指令的时间还要长！

同样，为了缩短除法运算的时间，**编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。**

- 无符号数：逻辑右移
- 带符号数：算术右移
- 整数除法规则：结果一定取整数
 - **能整除时**，直接右移，且移出的全为0，得到的结果正确。
例如， $12/4=3$: 0000 1100 >> 2 = 0000 0011
 $-12/4=-3$: 1111 0100 >> 2 = 1111 1101
 - **不能整除时**，直接右移，移出的位中有**非0值**，此时的结果可能存在问题（带符号负数异常）

注：要实现朝零舍入的目标

- 不能整除时的舍入处理：低位截断、朝零舍入

- 无符号数、带符号正整数：移出的低位直接丢弃，结果正确

如： $14/4=3$ ： $0000\ 1110 \gg 2 = 0000\ 0011$

- 带符号负整数：需加偏移量(2^k-1)，然后再右移 k 位，
低位截断（这里 k 是右移位数）

如： $-14/4=-3$ （朝0方向舍入）

➤ 若直接截断，则 $1111\ 0010 \gg 2 = 1111\ 1100 = -4 \neq -3$

➤ 应先纠偏，再右移： $k=2$ ，故 $(-14 + 2^2 - 1)/4 = -3$

即： $1111\ 0010 + 0000\ 0011 = 1111\ 0101$

$1111\ 0101 \gg 2 = 1111\ 1101 = -3$

例：假设 x 为一个int型变量，请给出一个用来计算 $x/32$ 的值的函数div32。要求只能用右移、加法以及任何按位运算，但不能使用除法、乘法、模运算、比较运算、循环语句和条件语句（包括条件运算?:）。

分析：

因为是除32，所以可采用右移5位的方式实现，取 $k=5$ ($32=2^5$)

- 若 x 为正数，则将 x 右移 k 位得到商；
- 若 x 为负数，则 x 需要加一个偏移量(2^k-1)后再右移。

所以，结果的计算规则是： $(x \geq 0 ? x : (x+31)) >> 5$

- 但题目要求不能用比较和条件语句（包括?:），怎么办？
- 主要是要找一个计算偏移量 b 的方式。

找一个计算偏移量b的方式:

- x为正时，直接有b=0;
- x为负时，应有b=31，怎么推导出这个值？

方法：可以通过操纵x的符号位得到b

不管是正数还是负数：

首先 $x \gg 31$ 得到的是32位符号（全0或全1）；

然后取出其中的最低5位，就是偏移量b。

```
int div32(int x)
{ /* 根据x的符号得到偏移量b */
  int b=(x>>31) & 0x1F;
  return (x+b)>>5;
}
```

注：本题不能写成如下的函数

```
int div32_Error(int x) /*错误*/
{
  return ( x>=0 ? x : (x+31))>>5;
}
```


8. 浮点数运算及结果（本部分自学）

设两个**规格化浮点数**分别为 $A = M_a \cdot 2^{E_a}$ $B = M_b \cdot 2^{E_b}$,则浮点

运算有: $A \pm B = (M_a \pm M_b \cdot 2^{-(E_a - E_b)}) \cdot 2^{E_a}$ (假设 $E_a \geq E_b$)

$$A * B = (M_a * M_b) \cdot 2^{E_a + E_b}$$

$$A / B = (M_a / M_b) \cdot 2^{E_a - E_b}$$

上述运算结果可能出现以下几种情况:

- **阶码上溢**: 一个正指数超过了阶码最大允许值, $>127 \Rightarrow +\infty/-\infty/\text{溢出}$
- **阶码下溢**: 一个负指数超过了阶码最小允许值, $<-126 \Rightarrow +0/-0$
- **尾数溢出**: 最高有效位有进位, $10.xxx \Rightarrow$ 右规
- **非规格化尾数**: 数值部分高位为0, $0.0..0xx \Rightarrow$ 左规
- **右规或对阶时, 右段有效位丢失** \Rightarrow 尾数舍入

尾数溢出, 结果不一定溢出

运算过程中添加保护位

涉及操作: **对阶, 左规、右规, 精度处理** (保护位、舍入位、粘位的设计)

1) 浮点数加/减运算

- 十进制科学计数法的加法例子

$$1.123 \times 10^5 + 2.560 \times 10^2$$

其计算过程为：

$$1.123 \times 10^5 + 2.560 \times 10^2$$

$$= 1.123 \times 10^5 + 0.002560 \times 10^5$$

$$= (1.123 + 0.00256) \times 10^5$$

$$= 1.12556 \times 10^5$$

移出的低位不舍弃

$$= 1.126 \times 10^5$$

“对阶”：使两数的阶码相等

➤ 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值；

➤ 尾数加减运算前，必须**对阶**！
最后还要考虑舍入

➤ 计算机内部的二进制运算也一样！

➤ IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0。
同时，移出的低位保留到特定的“附加位”上。

浮点数加减法基本要点

假定： X_m 、 Y_m 分别是浮点数 X 和 Y 的尾数， X_e 和 Y_e 分别是 X 和 Y 的阶码，并设 $Y_e > X_e$ 。步骤如下：

(1) **求阶差**： $\Delta e = Y_e - X_e$ (设 $Y_e > X_e$ ，结果的阶码为 Y_e)

(2) **对阶**：小阶向大阶看齐，将 X_m 右移 Δe 位，尾数变为

$$X_m * 2^{X_e - Y_e}$$

◆ 通常，为**保证精度**，尾数右移时，低位移出的位不立即丢掉，适当保留（有位置保存的话）并参加尾数部分运算，称为**附加位**。

(3) **尾数加减**： $X_m * 2^{X_e - Y_e} \pm Y_m$

◆ IEEE 754标准下，尾数加减实际上是**定点原码小数**的加减，并且**附加位**参与运算。

二进制下，两个尾数相加减后的结果有如下形式：

➤ $\pm 1.bb\dots b$

正常

➤ $\pm 1b.bb\dots b$



需要右规：

小数点左移、阶码增大

➤ $\pm 0.00001bb\dots b$



需要左规

小数点右移、阶码变小

IEEE 754规格化尾数形式为： $\pm 1.bb\dots b$ ，所以怎么办？

——进行规格化处理！

(4) 规格化:

◆ 当尾数最高位有进位: $\pm 1b.bb\dots b$, 需**右规**:

➤ 尾数右移一次(小数点左移), 阶码加1, msb为1 (隐藏位)

➤ 阶码加1后要判断阶码是否上溢 (比最大可表示的阶码还要大)

浮点数溢出判定规则: 阶码上溢, 则结果溢出

➤ IEEE 754 加减运算时, 最多只需一次右规: $\pm 1b.bb\dots b$

◆ 当尾数高位为0, $0.00001bb\dots b$, 需**左规**:

➤ 尾数左移一次(小数点右移), 阶码减1, 直到msb为1 (隐藏位)

➤ 每次阶码减1后要判断阶码是否下溢 (比最小可表示的阶码还要小)

➤ 阶码下溢到无法用非规格化数表示时, 则结果为0

(5) 尾数舍入：

在有附加位参加运算时，尾数可能比规定位数长，则需考虑舍入（有多种舍入方式，见后面的专门讨论）。

(6) 若运算结果尾数是0，则需要将阶码也置0。

为什么？

尾数为0说明结果应该为0（阶码和尾数为全0）。

例：用二进制浮点数形式计算 $0.5 + (-0.4375)$

解： $0.5 = 1.000 \times 2^{-1}$

$-0.4375 = -(0.25 + 0.125 + 0.0625) = -0.0111\text{B} = -1.110 \times 2^{-2}$

对阶： $-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$ 右规，阶码加1

尾数加减： $\underline{1.000} \times 2^{-1} + (\underline{-0.111} \times 2^{-1}) = 0.001 \times 2^{-1}$

左规： $0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$ 左规，阶码减3

判断溢出：无

尾数用原码表示

结果为： $1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

Extra Bits(附加位)

◆ 增加附加位的**目的**：保证计算精度

◆ 附加位**作用**：保存对阶时向右移出的位

◆ **加多少附加位才合适？**

➤ 无法给出准确的答案！但有总比没有好

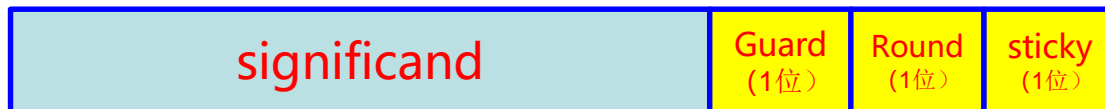
➤ IEEE754规定：**中间结果须在右边至少加2个附加位** (guard & round)

Guard (保护位)：在尾数右边的1位

Round (舍入位)：在保护位右边的1位

sticky (粘位)：在舍入位右边的1位。

只要舍入位右边有任何非0数字，粘位就被置1，否则为0.



附加位的作用：

用以保护对阶时右移的位或运算的中间结果。

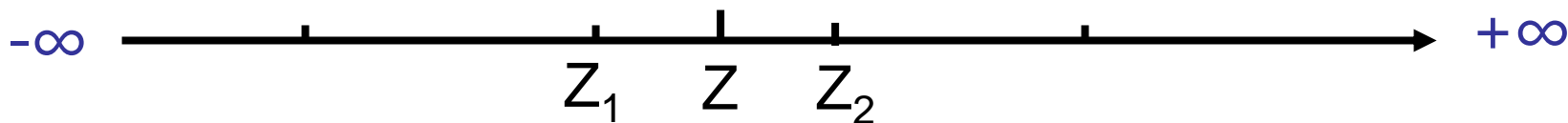
附加位的使用：

- 左规时被移到significand中;
- 舍入时，做为舍入的依据。

如何对保留的附加位进行舍入？

IEEE 754提供了四种舍入方式：

Z_1 和 Z_2 分别是 Z 的最近的可表示的左、右两个数

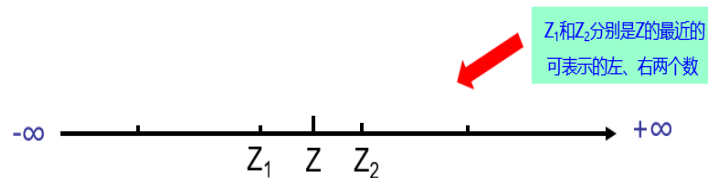


- (1) 就近舍入：舍入为最近可表示的数 Z_1 或 Z_2 ，或舍入为偶数；
- (2) 朝 $+\infty$ 方向舍入：舍入为 Z_2 ，即取比 Z 大的最近可表示数
- (3) 朝 $-\infty$ 方向舍入：舍入为 Z_1 ，即取比 Z 小的最近可表示数
- (4) 朝0方向舍入：直接丢弃所需位后面的所有位。

- 正数：取比其小的最近可表示数 (Z_1)
- 负数：取比其大的最近可表示数 (Z_2)

向0趋近

就近舍入：舍入为最近可表示的数



➤ Z 恰好不是 Z_1 和 Z_2 的中间值：根据Guard 位，0舍1入

注：此时，附加位(Guard 和Round)的值为：01 (舍) 或11 (入)

significand	0/1	1
-------------	-----	---

例：1.1101**11** → 1.1110(入) 1.1101**01** → 1.1101(舍)

➤ Z 恰好是 Z_1 和 Z_2 的中间值：强迫结果为偶数，即，保护位前面的位为1，则入，否则舍

注：此时，附加位 (Guard 和Round) 的值恰好为：10 (0.5)

significand	1	0
-------------	---	---

例：1.110**110** → 1.1110(入) 1.111**010** → 1.1110(舍)

注：“强制转换为偶数”不同于简单的四舍五入，这种舍入使左、右两个数 Z_1 和 Z_2 都有被舍入到的机会。

粘位的使用：

见P70，自学。

阶码溢出问题

在进行尾数规格化或对阶时，可能会对结果的阶码执行加、减运算，可能会造成阶码的溢出。

(1) 若阶码变全1，则发生“**阶码上溢**”。

此时，系统报“**阶码上溢**”异常或把结果置为 $+\infty$ 或 $-\infty$ （视机器而定）

(2) 若阶码变全0，则发生“**阶码下溢**”。

此时，系统报“**阶码下溢**”异常或把结果置为 $+0$ 或 -0 （视机器而定）

注：浮点数的溢出不以尾数的溢出判断，因为尾数“溢出”可以通过右规操作得到纠正，因此**浮点数是否溢出要通过判断阶是否溢出而定**。

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出

```
#include <stdio.h>
main()
{
    float a;
    double b;
    a = 123456.789e4;
    b = 123456.789e4;
    printf( "%f/n%f/n" ,a,b);
}
```

运行结果如下：

1234567936.000000

1234567890.000000

问题1：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

问题2：为什么float情况下输出的结果会比原来的大？

原因：float的24位二进制尾数只能精确表示7个十进制有效数位，多于7位时后面的数位是舍入后的结果，舍入后的值可能会更大，也可能更小。

double尾数是53位，所以不存在上述问题（但也是有限的尾数）。

2) 浮点数乘除运算

● 如何进行浮点数乘除运算？

- 在进行运算前，首先应对参加运算的操作数进行判0处理、规格化操作、溢出判断，并要确定参加运算的两个操作数是正常的规格化浮点数。
- 浮点数乘除不需要“对阶”操作，但同样需要对结果规格化、舍入、阶码溢出等处理。

● 乘除运算规则：

设两个浮点数 $x = M_x \times 2^{E_x}$, $y = M_y \times 2^{E_y}$, 则乘除运算的结果

$$\text{为: } x \times y = (M_x \times 2^{E_x}) \times (M_y \times 2^{E_y}) = (M_x \times M_y) \times 2^{E_x + E_y}$$

$$x / y = (M_x \times 2^{E_x}) / (M_y \times 2^{E_y}) = (M_x / M_y) \times 2^{E_x - E_y}$$

3) 浮点运算的异常问题: “大数吃小数”

- 由于浮点加减运算中需要对阶并最终进行舍入，所以可能导致“大数吃小数”，而使得浮点数运算不能满足加法和乘法结合律。

例： 设 $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$, 则

$$(x+y)+z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1.0$$

$$x+(y+z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + \underline{1.0}) = 0.0$$

对阶造成“小数”尾数中的有效尾数全部被右移后丢弃，从而使得小数变为0

举例：爱国者导弹定位错误

故事：1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹集中美军军营，28名士兵阵亡。

原因：由爱国者导弹系统时钟内的一个软件错误造成，根本原因是浮点精度问题：

- 爱国者导弹系统中有一个**内置时钟**，用**计数器**实现，每隔0.1秒计数一次。
- **0.1的二进制**表示是一个无限循环序列：**0.00011[0011]...**
- 程序中用**24位定点二进制小数** x 来表示0.1，并用 x 乘以计数值作为以秒为单位的时间，**0.1的机器数**是**24位近似表示**：

0.000 1100 1100 1100 1100 1100B

分析： x与0.1的误差是多少？

$$\begin{aligned} & 0.1 - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100B \\ &= 0.00011[0011]... - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100B \\ &= 2^{-20} \times 0.1 \approx 9.54 \times 10^{-8} \end{aligned}$$

而在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，相当于计数了 $100 \times 60 \times 60 \times 10 = 36 \times 10^5$ 次。

故，导致的时钟偏差为： $9.54 \times 10^{-8} \times 36 \times 10^5 \approx 0.343$ 秒。

➤ 当时飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少？

距离误差是 2000×0.343 秒 ≈ 687 米

所以，正是由于这个时钟误差，尽管雷达系统已经侦测到飞毛腿导弹，但爱国者导弹却找不到实际上正在来袭的飞毛腿导弹。致使起初的目标发现被视为一次假警报，而导致拦截失败。

实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案。可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。

1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

继续分析

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
 - $0.1 = 0.0\ 0011[0011]B = (+1.1\ 0011\ 0011\ 0011\ 0011\ 0011\ 00B \times 2^{-4})$ 24位规格化小数，
 - 故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
 - 由于float型仅24位有效位数（尾数），后面的有效位全被截断，故x与0.1之间的误差为：
$$|x-0.1| = 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ [1100]...B$$
$$= 0.1 \times 2^{-24} \approx 5.96 \times 10^{-9}。$$
 - 则，100小时后时钟偏差为 $5.96 \times 10^{-9} \times 36 \times 10^5 \approx 0.0215$ 秒。
故距离偏差 $0.0215 \times 2000 \approx 43$ 米。
 - 比爱国者导弹系统精确约16倍。（比24位定点小数提高了4位精度）

- 若用32位二进制定点小数：

$$x = 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 110\underline{1}\text{ B}$$

表示0.1，则误差又会是多少？

- 此时，x与0.1之间的误差约为：

这里有一个进位，提高了.5精度

$$\begin{aligned} |x - 0.1| &= 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 00\ 1100\ [1100]...B \\ &= 0.1 \times 2^{-30} \\ &\approx 9.31 \times 10^{-11} \end{aligned}$$

则，100小时后时钟偏差 $9.31 \times 10^{-11} \times 36 \times 10^5 \approx 0.000335$ 秒。

预测的距离偏差仅为 $0.000335 \times 2000 \approx 0.67$ 米。

比爱国者导弹系统精确约1024倍。

(相当于比24位定点小数提高了10位精度)

● 从上述结果可以看出：

- 用32位定点小数表示0.1 比采用float精度高64倍 (相当于精度提高8位)。
- 用float表示在计算速度上慢，因为，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比采用定点小数直接将两个二进制数相乘慢得多。

● 带来的启示：

- ✓ 程序员应对底层机器级数据的表示和运算有深刻理解；
- ✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确；
- ✓ 不能遇到小数就用浮点数表示，有些情况下，可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点（相当于先放大若干倍，再缩小若干倍，用整数运算提高运算的速度）。

数据的运算小结

- C语言中涉及的运算
 - 整数算术运算、浮点数算术运算
 - 按位、逻辑、移位、位扩展和位截断
- 整数的加、减运算
 - 计算机中的“算盘”：模运算系统（高位丢弃、用标志信息表示）
 - 带符号整数和无符号数的加、减都在同一个“算盘”中
 - 现实与计算机中的运算结果有差异（计算机是模运算系统）
- 整数的乘、除运算
 - 无符号整数：逻辑左移 k 位等于乘 2^k 、逻辑右移 k 位等于除 2^k
 - 带符号整数乘：算术左移 k 位等于乘 2^k
 - 带符号整数除：（ $x + 2^k - 1$ ）算术右移 k 位等于 x 除以 2^k
- 浮点数运算
 - 加减：对阶/尾数加减/规格化/舍入（就近舍入到偶数）（大数吃小数）
 - 乘除：尾数相乘除，阶码相加减

第二章作业

- 课后习题：

**9、10、17、21、24、28、29、31、
34（偶数小题）、35（奇数小题）、36、39**