
第二章 数据的机器级表示与处理

张宇

本章背景介绍

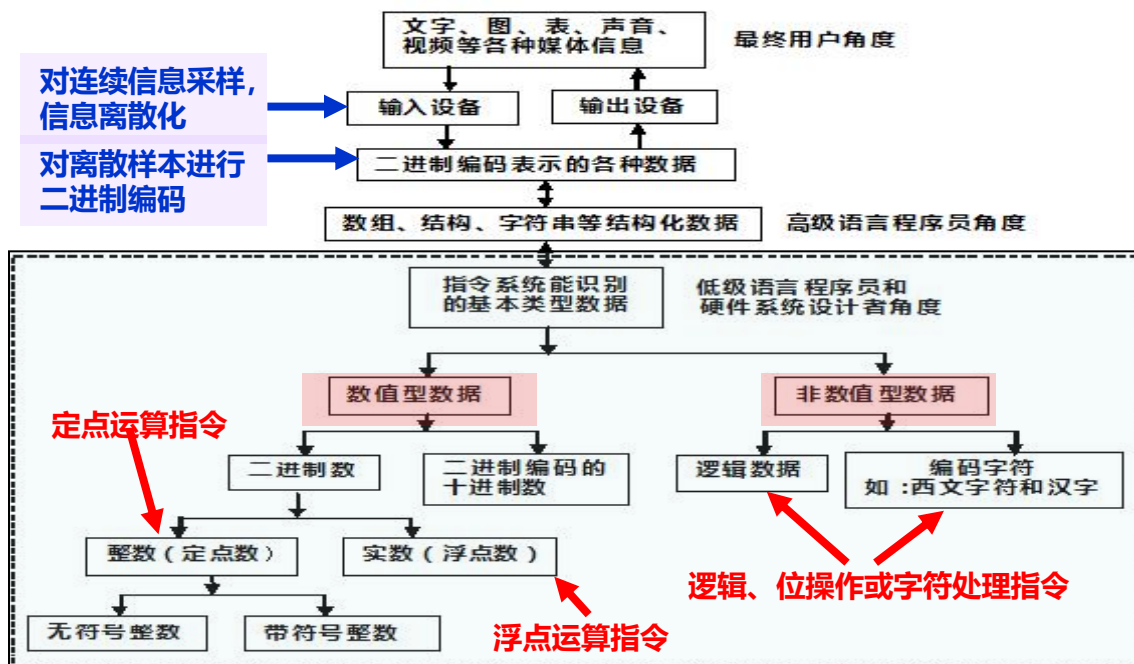
数据是计算机处理的对象，它可以以不同的表现形态呈现：

- **从计算机外部看**，计算机可处理数值、文字、图像、声音、视频及各种模拟信息，它们被称为**感觉媒体信息**；

由于计算机存储、加工和传输数据的部件位数有限，计算机只能表示和处理离散的信息。这需要通过输入设备对感觉媒体信息进行采样和编码，转换成二进制编码表示。也就是说，计算机内部存储和处理的数据都是“数字化编码”后的数据

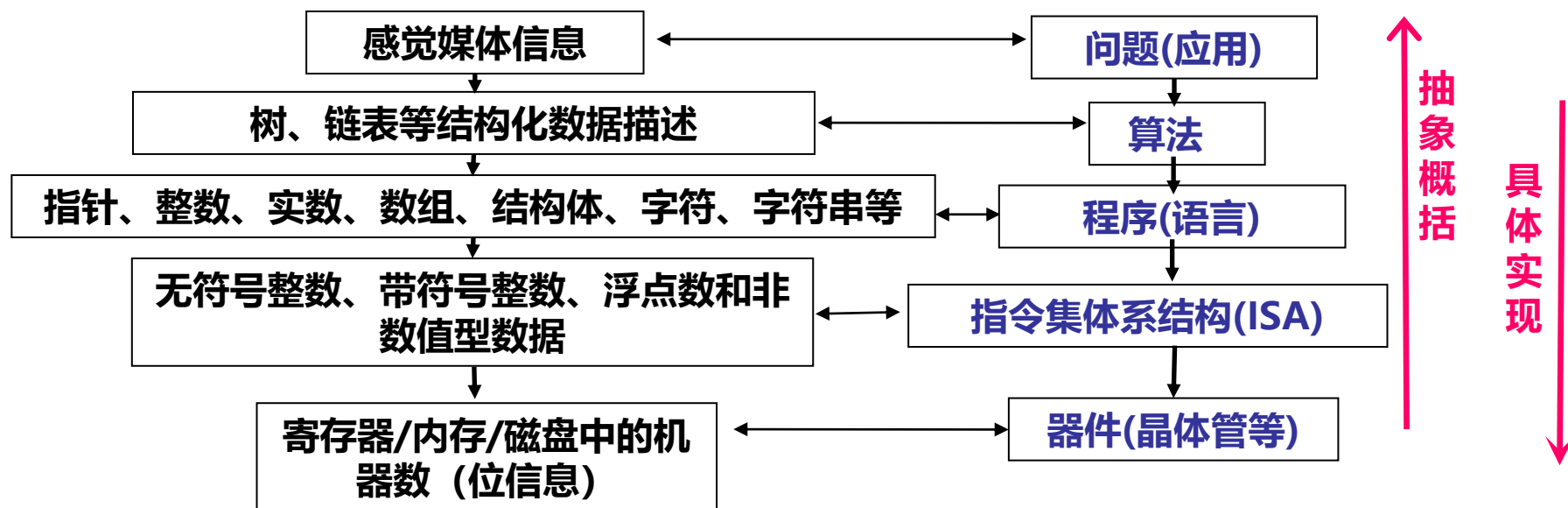
➤ 计算机内部数据呈现形态：

- ❑ 从算法描述的角度看，有图、树、表、队列等结构类型的数据
- ❑ 从高级语言程序员的角度看，有结构体、数组、指针、实数、整数、字符、字符串等类型的数据
- ❑ 从机器（指令）的角度看，数据只有**无符号整数、带符号整数、浮点数和非数值型数据**（位串）这几类简单的基本数据类型



本章主要教学目标

- 掌握计算机内部底层各种数据（主要是：**无符号整数、带符号整数、浮点数**）的**编码表示及其机器级运算机制**
- 了解高级语言程序中的**各种类型变量**对应的**表示形式**，并在高级语言程序中的**变量及其运算、机器数及其运算和底层硬件**（寄存器/内存/磁盘、ALU）**之间建立关联**



- 综合运用所学知识，分析高级语言程序设计中遇到的各种**与数据表示和运算**相关的问题，解释相应的执行结果

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

- **第二章：数据的机器级表示与处理**

- 2.6 数据的宽度和存储

- 数据的宽度和单位
 - 数据的存储和排列顺序（特别是：大端方式/小端方式）

• 第二章：数据的机器级表示与处理

– 2.7 数据的基本运算

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加、减、乘、除运算
- 浮点数的加、减、乘、除运算

从C程序的表达式出发，解释C语言中的运算在底层机器级的实现（通过介绍**机器数**在**电路**中的处理过程，解释表达式执行结果为什么是这样）

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2.1 数值数据的表示

- **基本概念**

- **数值数据的定义：**

- **数值数据**是表示数量多少的数据，即可以进行数值运算的数据，包括：无符号整数、带符号整数、浮点数，例如：1, -2, 1.0, 2.3, 1.0E2

- 数值数据的表示有三个要素

- 进位计数制
- 定点表示/浮点表示
- 二进制编码规则

◆ 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！

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

1、进位计数制

- 计算机系统中常用的进位计数制有：十进制、二进制、八进制、十六进制：

- **十进制数**：逢十进一（日常生活中用的）

- **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

各种进制的数之间的相互转换

— R进制数向十进制数的转换：按权展开

$$(xyz.ab)_R = (x \times R^2 + y \times R^1 + z \times R^0 + a \times R^{-1} + b \times R^{-2})_{10}$$

■ 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

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

– 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

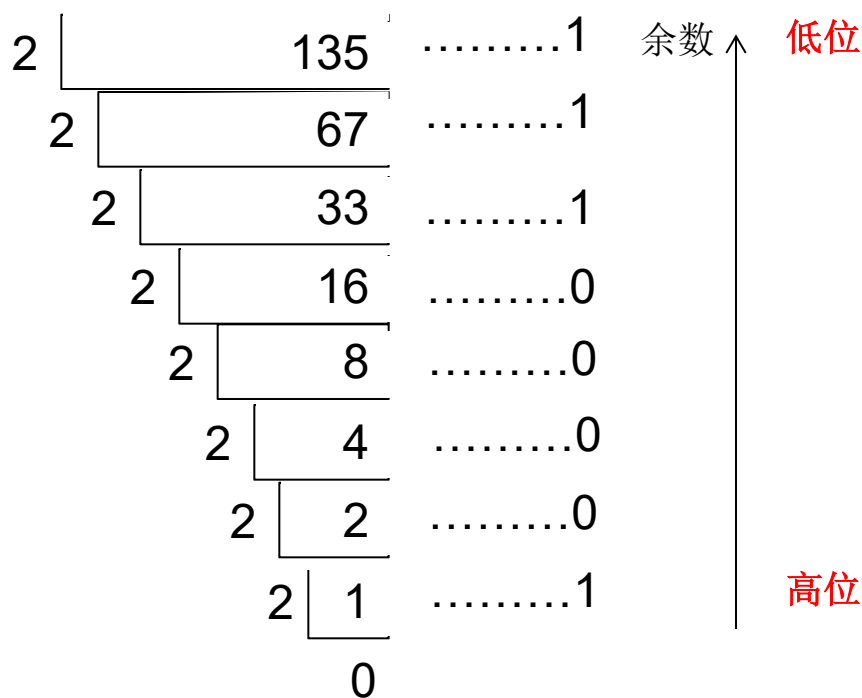
➤ 小数部分：乘基取整

◆ $(xyz.ab)_R$ 整数部分的转换 $M = x \times R^2 + y \times R^1 + z \times R^0$

口诀：除基取余、上右下左（上低下高）

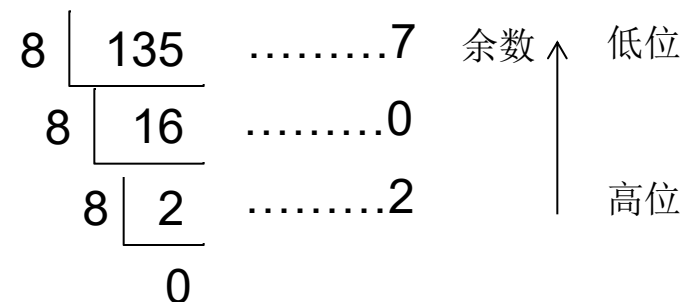
十进制数转二进制数

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



十进制数转八进制数

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



◆ $(xyz.ab)_R$ 小数部分的转换 $N = a \times R^{-1} + b \times R^{-2}$

口诀：乘基取整、上左下右（或者叫上高下低）

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
$0.375 \times 2 = 0.75$	0	
$0.75 \times 2 = 1.5$	1	
$0.5 \times 2 = 1$	1	低位

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
$0.5 \times 8 = 4.0$	4	低位

◆ 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

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

一、二、八、十六进制数的相互转换

□ 二进制转换成八进制： $(10\ 000\ 111)_2 = (207)_8$

- 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位的前面补0**。

□ 二进制转换成十六进制： $(1000\ 0111)_2 = (87)_{16}$

- 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位的前面补0**。

□ 八进制转换成二进制： $(207)_8 = (10\ 000\ 111)_2$

- 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： $(87)_{16} = (1000\ 0111)_2$

- 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。

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2、定点表示/浮点表示 —— 用于解决小数点的表示问题

— 在日常中，带小数点的数非常常见，小数点位置也不固定

如： 123456. .123456 123.456

那么，计算机内部怎么表示小数点？

■ **定点表示：** 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
- **定点小数：** 小数点约定在数的最左边

可以看到，采用这种默认的表示方式后，在不存储小数点信息时，机器也不会出错

■ 可以看到：定点整数可以用于表示整数

■ 这种定点表示方式有问题吗？ **有问题**

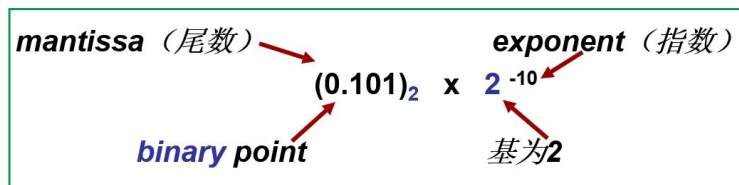
例如：假若小数点约定在最左边，这时候如果要存储123.456怎么办？

因为需要小数点左移3位使其转换成.123456的形式，这个数字3怎么存储

■ 因此，需要**浮点表示**，即允许小数点位置浮动。那么怎么做呢？

2、定点表示/浮点表示 —— 用于解决小数点的表示问题

■ 浮点表示：



采用这种表达方式，然后用一个**定点整数**存储**指数**。这样就允许我们通过调整**指数**进行等值的**小数点移位**，直到满足**提前约定的定点小数表示方式**

例如：**123.456**需要小数点左移**3**位使其转换成**.123456**的形式。我们只需要存储尾数**.123456**和指数**3**即可。而**基**可以**提前约定好**（比如**2**），**不用存储**。

可以看到：

- **浮点表示**可以基于**定点表示**来实现，用一个**定点小数**表示浮点数的**尾数**和一个**定点整数**表示浮点数的**指数**，**基可以提前约定好（比如2），不用存储**。
- 因此，任意一个**浮点数**都可以用一个**定点整数**和一个**定点小数**来表示。所以，在后面对数值数据进行编码时，我们只需要考虑**定点数**的编码方法（即：解决了定点数的编码就等于解决了定点数和浮点数的编码）

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

3、定点数的编码——主要需要解决**正负符号**表示问题

➤ 现实世界中的数通常带有正负号，或称作**真值**。

例如：-10（或-1010B）是真值

问题：怎么在机器上表示真值的正负号呢？

为了在计算机中表示真值的正负号，需要进行**符号数字化**
(例如用0表示正号，用1表示负号)

问题：如何使得数字化后的符号能和数值一起参与运算？

为了使得**数字化了的符号**能和**数值部分****一起参与运算**，产生了把**符号位**和**数值部分****一起**进行**编码**的各种方法。这种编码后的**数值数据**叫做**机器数**

3、定点数的编码——主要需要解决正负符号表示问题

➤ **机器数**：就是**数值数据**在**计算机内部**编码表示后的数。

假设**真值** X_T 的二进制形式如下：

注： $n-1$ 位数

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

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

则， X_T 的在二进制编码后的**机器数** X 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 $X_{n-2} \dots X_1 X_0$ 是其数值部分：

正数： $X_i = ? \quad X'_i$ 不同的映射对应着不同的
负数： $X_i = ? \quad X'_i$ 编码方法

➤ 现在主要有4种编码表示方法：**原码**、**补码**、**移码**、**反码**

1) 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, $X_{n-1}=1$, $X_i=X_i'$ ($0 \leq i \leq n-2$)

—由符号位+真值的数值位直接构成

Decimal	Binary	Decimal	Binary
0	0000	-0	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

• 原码编码有什么优点？

➤ 数值位与真值的数值位相同，符号位表示正负号，因此与真值的对应关系直观，转换方便，容易理解

$$\begin{array}{r} 0101 \\ \times 0101 \\ \hline 0101 \\ 0000 \\ + 0101 \\ \hline 00011001 \end{array}$$

➤ 用原码实现乘、除运算比较简便

• 原码编码又有什么缺点？

➤ 原码0有两种形式：正0和负0，表示不唯一，给程序员编程带来不便

➤ $[+0]_{\text{原}} = 0000\dots 0$

➤ $[-0]_{\text{原}} = 1000\dots 0$

➤ 原码表示后的带符号整数，加、减法运算规则复杂，不利于硬件设计：

✓ 需额外对符号位进行处理（即需要根据两个数的符号位，决定结果的符号位）

✓ 另外，需要对异号数相加或同号数相减做额外的判定：即在这种情况下，必须判断两个数的绝对值大小，以决定结果的符号

➤ 因此，后面有人设计了补码。现代计算机都是用补码来给带符号整数进行编码

2) 补码

学习补码首先需理解什么是模运算系统？

(1) 模运算系统： **模运算定义的代数系统**

如果两个数除以“模”后的余数相同，那么这两个数在此模运算系统中算是同一个数（或叫做模同余）

模同余：若A、B、M满足下列关系：

$$A = B + K \times M, \text{ K为整数,}$$

则，记为 $A \equiv B \pmod{M}$ ，并称为A和B为**模M同余**，即A和B各除以M后余数相同。

例如：时钟是一个典型的模运算系统（模是12）

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： $10 + 8 = 18 \equiv 6 \pmod{12}$

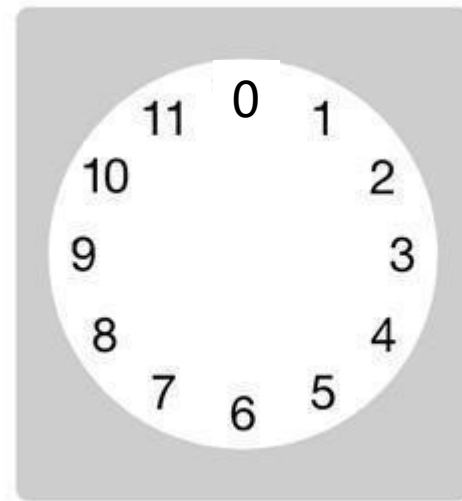
因为，在模12系统中：

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

即， $-4 \equiv 8 \pmod{12}$ （即向后4步和向前8步一样）

这里，称8是-4对模12的补码。

同样，有 $-3 \equiv 9$ 、 $-5 \equiv 7 \pmod{12}$ 等，所以-3的补码是9，-5的补码是7等



可以看到：

(1) 一个负数的补码等于模减该负数的绝对值。例如：M=12时，-4的补码 = $12 - 4 = 8$

(2) 在模运算系统中，某数A减去小于模的另一数B，可用A加上-B的补码来实现

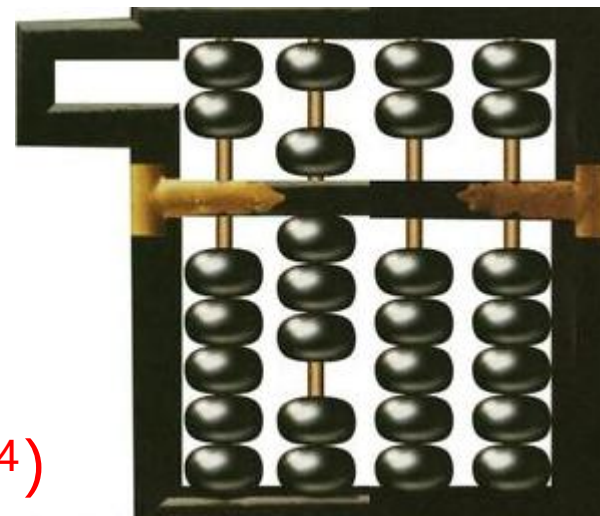
例如： $10 - 4 \equiv 10 + (12 - 4) \equiv 10 + 8 \pmod{12}$

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数的负数的补码(模M)。

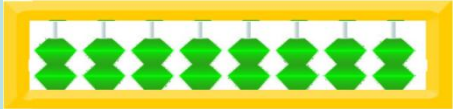
例2：假定算盘只有四档，且只能做加法，则在算盘上计算
9828-1928等于多少？

分析：“4位十进制数”，模10000运算系统

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



高位“1”被丢弃，只有低4位留在算盘上本质就是：取模，即只留余数

➤ 由于计算机中的存储、运算和传送部件都只有**有限位数**, 所以**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** , 运算的结果只能保留低 n 位, 高位将被舍弃

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

也可以看到:
 $-0100\ 0000$ 的补码等于模减去该负数的绝对值。因为, 这个模 2^8 最终也是作为最高位被丢弃了, 所以结果正确

(2) 补码的定义

- 正数的补码等于数的本身（原码）。
- 负数的补码等于模减去该负数的绝对值。

◆ 其实，数 X_T 的补码公式可表示为：

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

因为，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， $[X_T]_{\text{补}} = M - |X_T| = M + X_T \pmod{M}$

◆ 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 $M = 2^n$

为什么是这个区间，而且为什么还不对称

$[X]_{\text{补}} = 2^n + X_T \pmod{2^n}$ ，其中 $\underline{-2^{n-1} \leq X_T \leq 2^{n-1} - 1}$

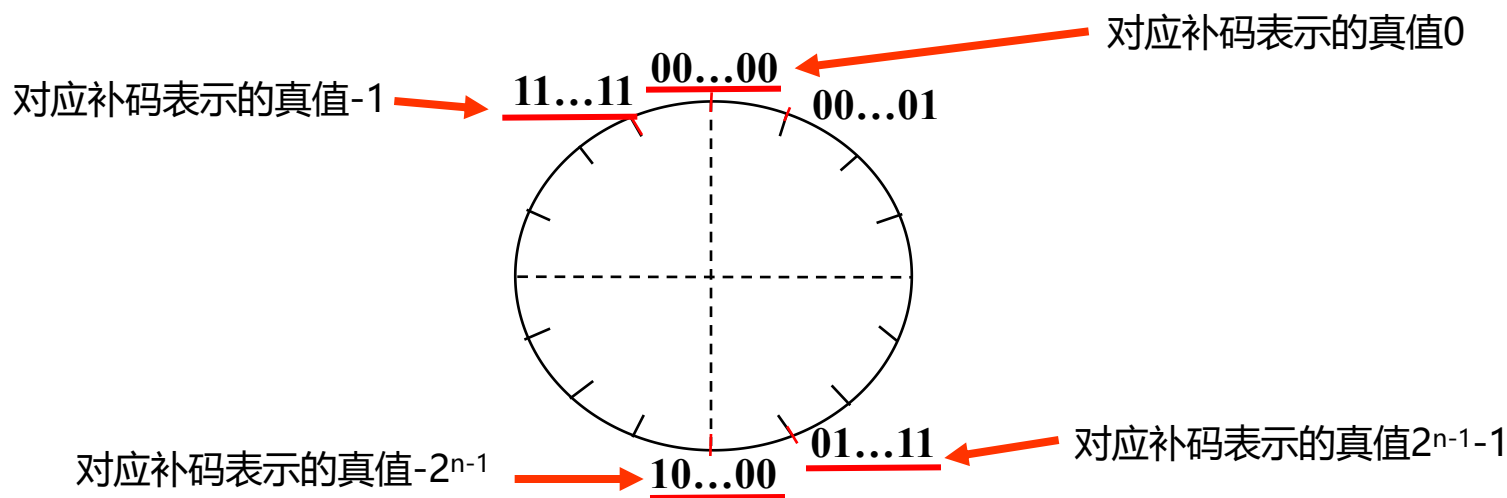
特殊数的补码

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**

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

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

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



思考：

① $[2^{n-1}]_{\text{补}} = ?$

②除了能表示 $[-2^{n-1}]_{\text{补}}$ ，**10...0** ($n-1$ 个0) 还能表示什么？

特殊数的补码

思考：

① $[2^{n-1}]_{\text{补}} = ?$

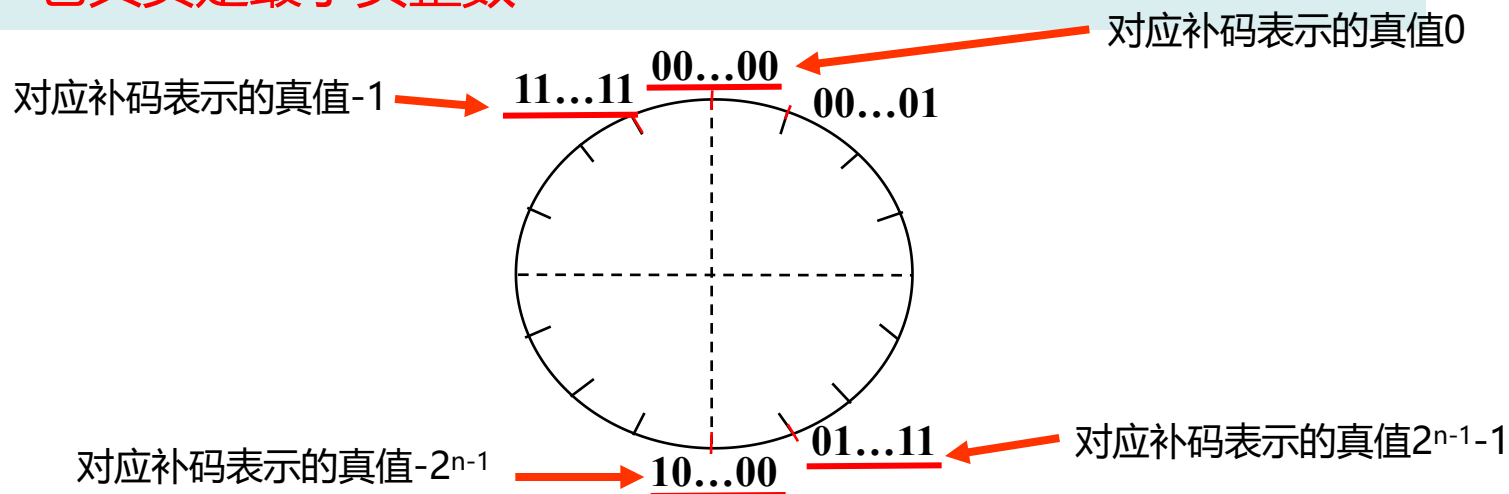
$$[2^{n-1}]_{\text{补}} = 2^n + 2^{n-1} \pmod{2^n} = (10\dots0)_2 \quad (n-1 \text{ 个 } 0) \quad \pmod{2^n}$$

$[2^{n-1}]_{\text{补}} = (10\dots0)_2 = [-2^{n-1}]_{\text{补}}$ 这么大的正数的补码怎么和负数补码一样？

所以，最大正整数为： $2^{n-1}-1$ （当然你也可以把 $(10\dots0)_2$ 定义为最大正数，但是，这使得没有明显的符号位，导致符号判断很难，特别是条件判断指令和带条件跳转指令的执行）

② $10\dots0$ ($n-1$ 个0) 表示多少？

-2^{n-1} 它其实是最小负整数



补码与真值(二进制表示)之间的转换

例: 设机器数有8位, 求真值123 (0111 1011B) 和 -123 (−0111 1011B) 的补码表示。

解: $[0111\ 1011]_{\text{补}} = 2^8 + 01111011$
 $= 1\ 0000\ 0000 + 0111\ 1011$
 $= 0111\ 1011 \pmod{2^8}$ 即 7BH

正数补码: 各位同真值中对应的各位;

$[-0111\ 1011]_{\text{补}} = 2^8 - 0111\ 1011 = 1\ 0000\ 0000 - 0111\ 1011$
 $= 1111\ 1111 - 0111\ 1011 + 1$
 $= 1000\ 0100 + 1$ 取反
 $= 1000\ 0101$ 即 85H

负数补码: 各位由真值 “各位取反, 末位加1” 得到

另外: 可以看到, $[-X_T]_{\text{补}}$ 可以通过对 $[X_T]_{\text{补}}$ 各位取反, 末位加1得到

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。

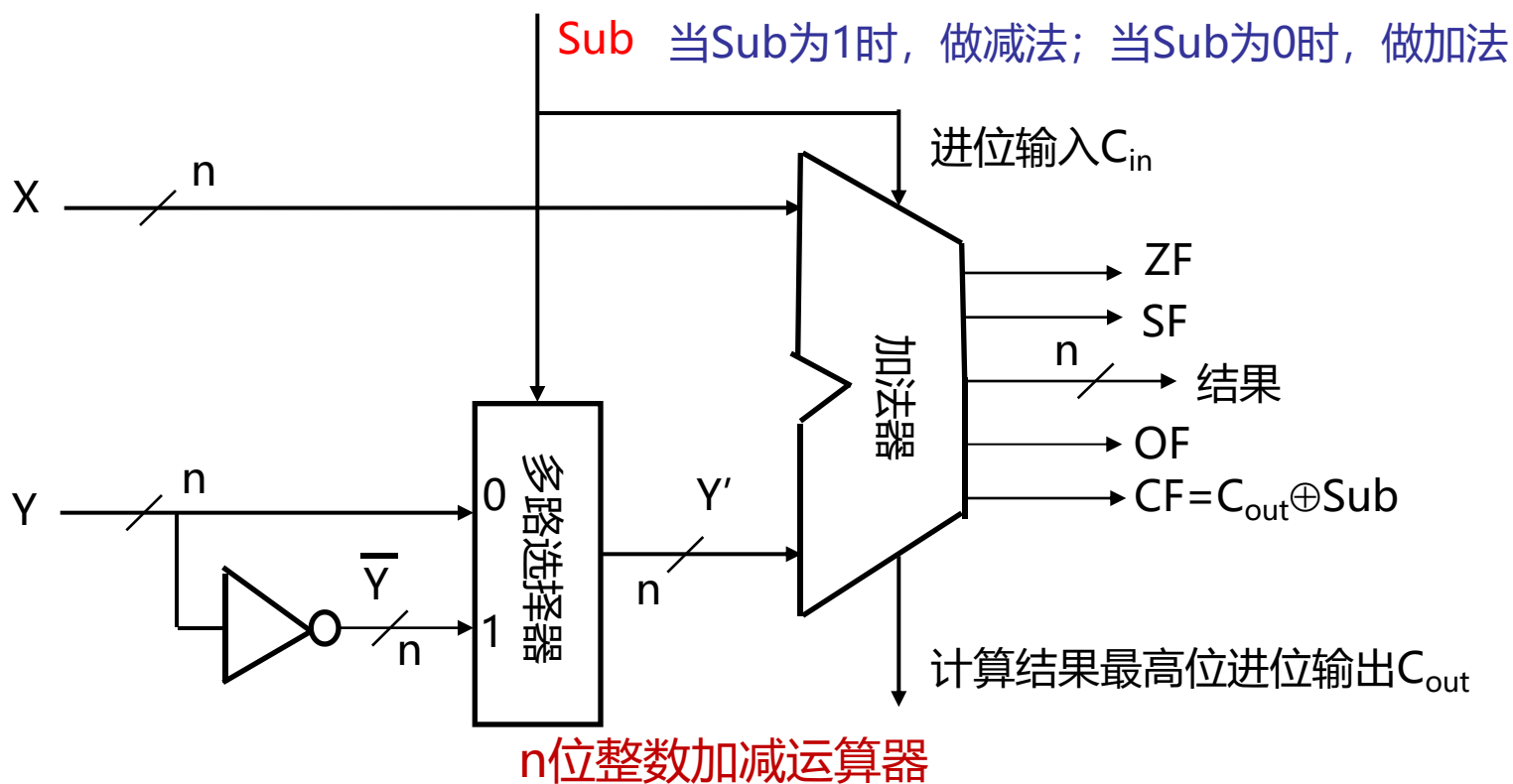
例：已知 $[X_T]_{\text{补}} = \underline{1000\ 0101}$ ，求其真值。

解： $X_T = -(\underline{0111\ 1010} + \underline{1}) = -0111\ 1011\text{B} = -123$

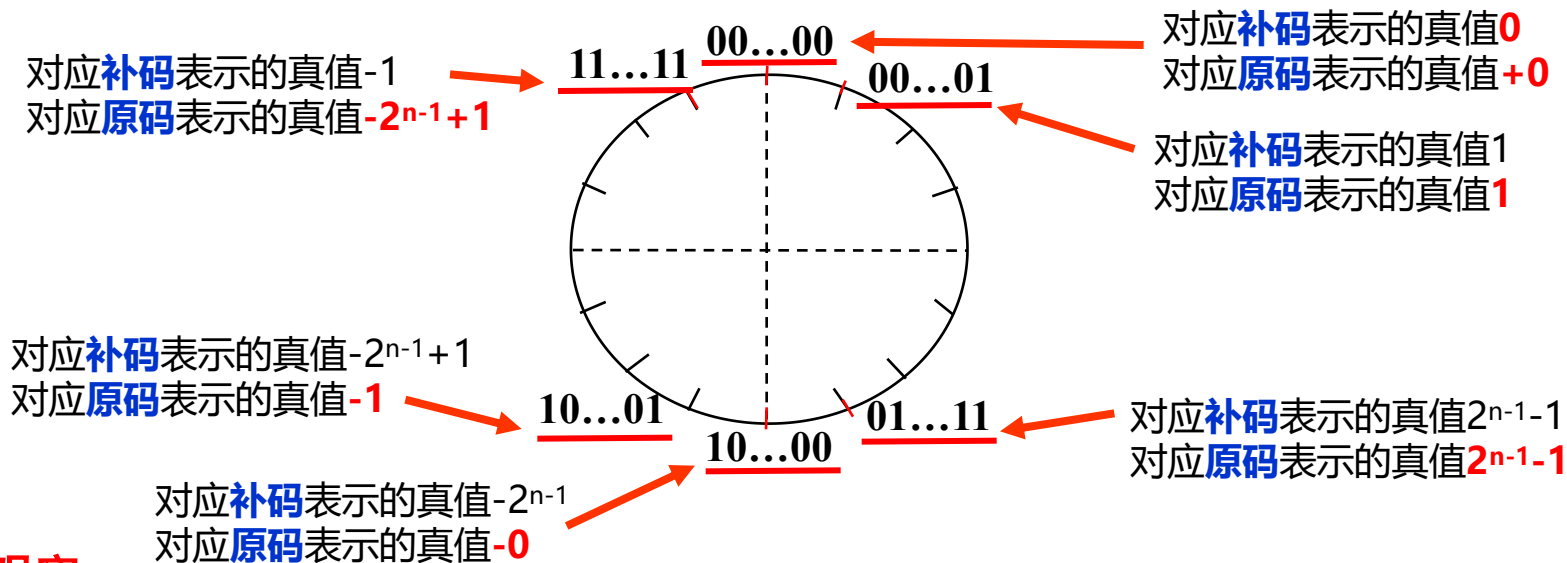
补码优点总结

由于以下优势，在现代计算机中，带符号整数采用补码表示

- 0的表示唯一。 $[+0]_{\text{补}} = [-0]_{\text{补}} = (00\dots0)_2$ ，方便条件判断指令和带条件跳转指令的执行（这些指令的整数的比较都是基于减法运算实现的）
- 减一个数的补码 $[Y]_{\text{补}}$ 等于加上这个数的负数的补码 $[-Y]_{\text{补}}$ ，从而允许用加法来实现减法运算，最终可以实现加减运算的统一。其中 $[-Y]_{\text{补}}$ 可以通过对 $[Y]_{\text{补}}$ 各位取反，末位加1得到



补码和原码编码规律对比

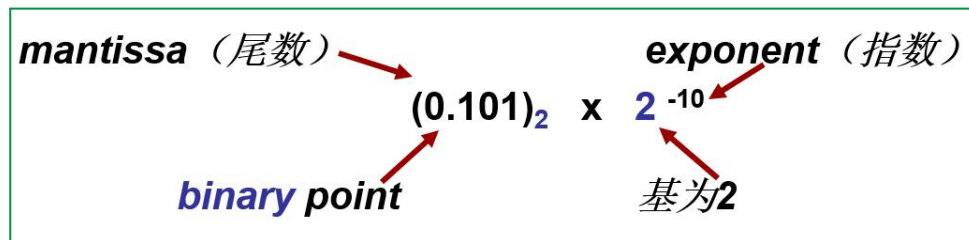


一些观察：

- 补码能表示的负数个数比正数个数多1，而原码负数个数和正数个数一样。为什么？
因为补码编码的最小负数 -2^{n-1} 没有与之对应的正数 2^{n-1} 。编程时不注意，程序容易出错（例如因为 $x=-2^{n-1}$ 没有与之对应的 2^{n-1} ，使得 $(x>y) == (-x<-y)$ 不一定成立）
- 对于补码表示， $-2^{n-1}-1=2^{n-1}-1$ （即 $2^{n-1}-1+1=-2^{n-1}$ ），编程时不注意容易导致程序出错
- 原码相对于补码能表示的数少一个，即 -2^{n-1} ，因为原码浪费了 $10\cdots00$ 来表示没用的-0
- 原码编码的连续机器数对应的真值不连续。这使得加减运算统一难以实现。而且，这使得原码编码的机器数加减运算复杂，例如 $x-y$ 结果的符号判断复杂：因为需根据 x 和 y 的符号及大小才能确定结果的符号（负数 x -正数 y =负数，正数 x -负数 y =正数，负数 x -负数 y 的符号需判断 x 和 y 的值的大小，正数 x -正数 y 的符号需判断 x 和 y 的值的大小）
- 补码编码的连续机器数对应的真值连续：从 $10\cdots00$ 到 $00\cdots00$ 再到 $01\cdots11$ 能连续表示真值 -2^{n-1} 到0再到 $2^{n-1}-1$ ，为加减运算统一提供了机会，如 $x+11\cdots11=x+\text{模}+0-1=x+0-1=x-1$

3) 移码

■ 浮点数的一般形式：



问题：浮点表示的指数可以是正数，也可以是负数，怎么表示指数的正负呢？

■ 移码就是用于给浮点表示中的指数（或叫做阶）编码。

思考：为什么需要移码？

因为在浮点数加减法运算时，首先要对阶（比较两个浮点数阶的大小并使之相等）。为了简化阶的比较操作，使操作过程不涉及阶的符号（当不涉及符号时，可以直观地将两个数按位从左到右进行比对），计算机对每个阶的实际值（即真值）都加上一个正常数，使所有的阶都转换为非负数

例如： $-1 + 128 = 127$

$1 + 128 = 129$

- 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**”： $[E]_{\text{移}} = E + \text{偏置常数}$

例: 设移码有8位，偏置常数为 $2^7 - 1 = 127$ ，则-123的移码为:

$$[-123]_{\text{移}} = -123 + 127 = 4 = 0000\ 0100$$

- **移码**还原得到真值的方式： $E = [E]_{\text{移}} - \text{偏置常数}$

4) 反码

- 正数的反码与其真值相同；
- 负数的反码是对其真值各位逐位取反而得。

例如：-0001 0001 的反码表示为：1110 1110

计算机很少使用反码。反码只是数码变换的**中间表示形式**（例如，在整数加减运算器中，对于某数Y，-Y的补码可以**基于Y的反码+1**得到）的**称呼**或用于数据校验等

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2.2 整数的表示

■ 无符号整数 (Unsigned integer)

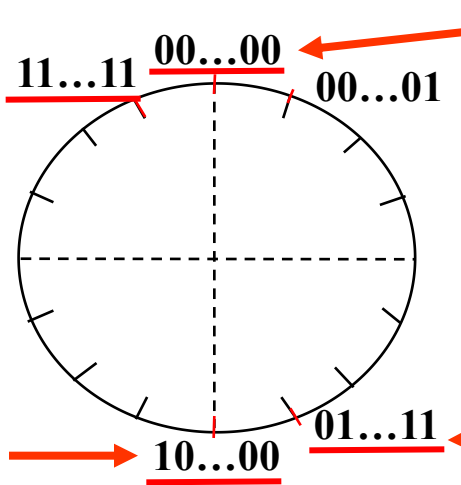
- **无符号整数** (简称无符号数) 是**所有二进制位**都用来**表示数值**，**没有符号位**。
- C语言中的unsigned int / unsigned short / unsigned long等，一般在一个常数的后面加一个 **"u"** 或 **"U"** 表示无符号数，例如，12345U, 0x2B3Cu
- 一般应用场合：在**全部是正数运算且不出现负值结果**的情况下，例如：数组下标、指针，地址运算等。

■ 带符号整数 (Signed integer)

- **带符号整数**必须用一个**二进制位表示符号**，其余位为**数值位**。
- C语言中的int/short/long等
- 虽然前面介绍的原码等都可以用于表示带符号整数，但是由于在采用**补码**表示后，**能实现加减运算统一**，因此**带符号整数**在现有计算机上都用**补码**表示，**最高位**是符号位 (0表示正数，1表示负数)

■无符号整数和带符号整数关系

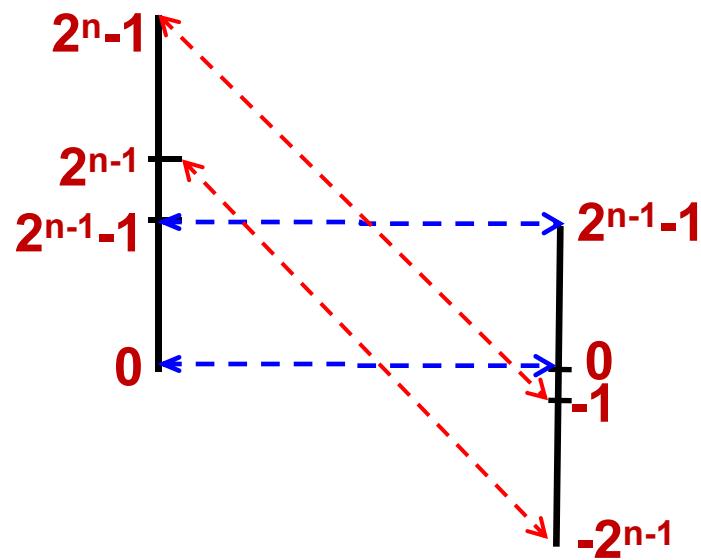
对应的无符号数的真值 2^n-1
对应的带符号数的真值 -1



对应的无符号数和带符号数的真值都是0

对应的无符号数的真值是 2^n-1
对应的带符号数的真值是 -2^n-1

对应的无符号数和带符号数的真值都是 2^n-1-1



观察:

- 无符号数的最大值是带符号数的最大值的两倍加一。
例如，8位无符号整数能表示的最大正数是255 (1111 1111)；8位带符号整数能表示的最大正数是127 (0111 1111)。这是因为：虽然它们能表示的总数值个数是相同的，都是 2^n 个，但是被补码用于表示负数的那些编码被无符号数用于表示正数。
- 当真值在 $[0, 2^n-1-1]$ 的区间内时，它们对应的无符号数和带符号数的编码也相同
- 对于10...00到11...11的编码，它们对应的无符号数和带符号数的真值不同，无符号数对应真值转换成带符号数对应真值需要减去模数 2^n
例如: 最大无符号数 2^n-1 的编码和带符号整数 -1 的编码相同
无符号整数 2^n-1 的编码和带符号整数 -2^n-1 的编码相同

C语言中允许无符号整数和带符号整数之间的转换

无符号整数  带符号整数

- (1) **内容不变**：类型**转换前后的机器数不变**，即**各位0/1值不变**
- (2) **含义新译**：只是**转换前后对机器数的解释发生了变化**，即**转换后数的真值将按照转换后的类型对机器数进行解释得到**

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，它的输出结果是什么？为什么？

输出结果：x = 4294967295 = -1

u = 2147483648 = -2147483648

可以看到，-2147483648和2147483648在机器数表示上一样：都是100...0，只是作为不同数据类型进行解释，所以对应的真值不同

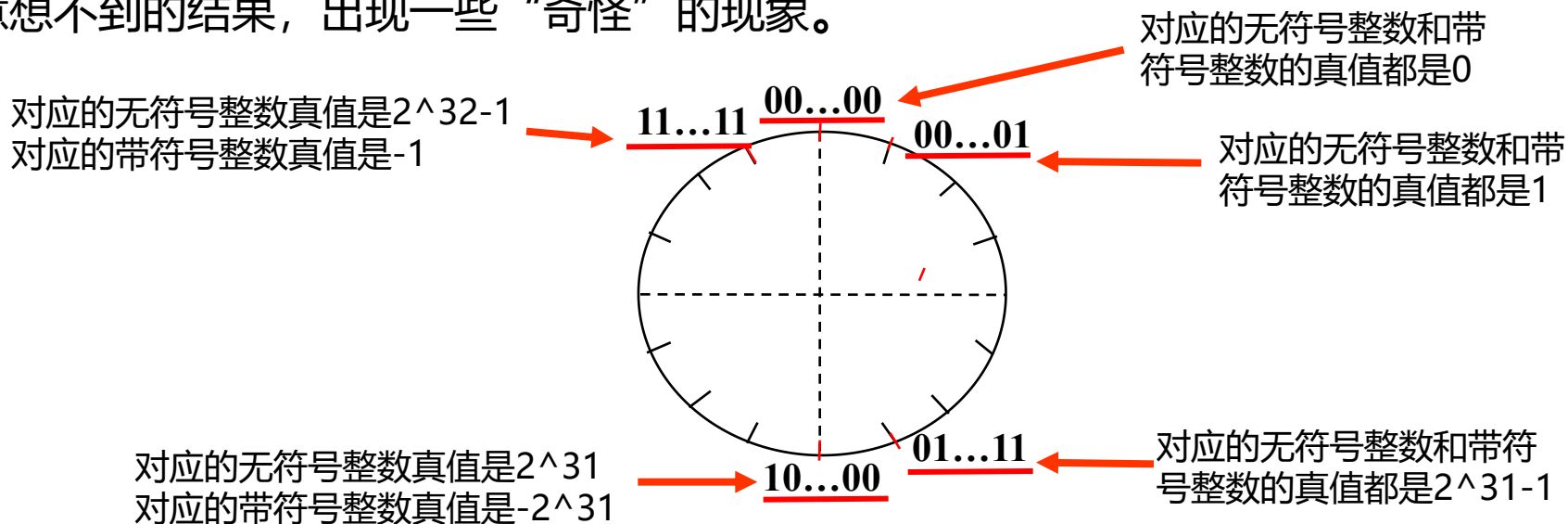
② $2147483648 = 2^{31}$ ，无符号数表示为“100...0”，被解释为32位

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

C语言中允许无符号整数和带符号整数之间的转换

思考： C语言中无符号数和带符号数在转换时只是转换后数的真值将按照转换后的类型对机器数进行解释得到，而类型转换前后的机器数不变。这样会有什么问题的呢？

问题： 在混合表达式中，若同时有无符号整数和带符号整数，则C编译器可能会将带符号整数强制转换为无符号整数（类型提升）。因此会造成程序在某些情况下产生意想不到的结果，出现一些“奇怪”的现象。



◆ 假定以下关系表达式在32位机器上执行，**U**表示无符号数，否则表示带符号数(补码表示)，试分析以下结果

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	$11...1B (2^{32}-1) > 11...10B (2^{32}-2)$

思考：带*的结果是不是与“常规预想”的结果相反？

为什么对同样的机器数进行比较，而结果不一样？

例如，在有些32位系统上，

1) 在ISO C90标准下，C表达式 $-2147483648 < 2147483647$ 的执行结果为false。Why?

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0x8000 0000，因此，在ISO C90标准下，将2147483648看成unsigned int；虽然对其取负并求补码，但最终机器数仍为0x8000 0000，类型仍是**无符号整数**，值为2147483648。同时2147483647（对应的机器数是0x7FFF FFFF）的类型也被提升，当做无符号数处理。所以 “ -2147483648 （机器数0x8000 0000）” 与 “2147483647（机器数0x7FFF FFFF）” 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**。

C90中，编译器对**数值型字面常量**的**类型确定顺序**：

<div>2147483647 = $2^{31}-1$</div> <hr/> <div>2147483648 = 2^{31}</div> <div>↓</div>	范围	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

2) 在ISO C90标准下, 若定义变量 “int i=-2147483648;” , 则 “i < 2147483647” 的执行结果为true。Why?


编译器在处理 “int i=-2147483648;” 时, 会根据i的类型, 将机器数0x80000000在赋给i时进行类型转换 (赋值转换) , 解释成带符号整数, 即 -2^{31} , 然后赋给变量i (注: 机器数没变) , 所以, i的值最终被看作带符号的-2147483648 (机器数0x80000000) , 然后按照带符号整数的规则与2147483647 (机器数0x7FFF FFFF) 进行比较, 所以表达式 “i < 2147483647” 结果为true

3) 在ISO C90标准下, 如果将表达式写成 “-2147483647-1 < 2147483647” , 则结果会怎样呢? Why?

此时, 结果为true。因为, 编译器首先将2147483647 ($2^{31}-1$, 机器数0x7FFFFFFF) 看成带符号数, 对其取负, 得-2147483647 (机器数0x80000001) , 然后再将其减1, 得到带符号数-2147483648 (机器数0x80000000) , 再然后作为带符号数和 2147483647 (机器数0x7FFFFFFF) 比较, 所以结果为true

4) 在ISO C99标准下，C表达式 $-2147483648 < 2147483647$ 的执行结果为true。Why?

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（即 2^{31} ）”被编译器解释成为64位的long long型带符号数，从而“-2147483648（机器数0x80000000）”与“2147483647（机器数0x7FFFFFFF）”的比较是按照带符号整数类型进行比较，所以“-2147483648 < 2147483647”的结果为true。

- 从前面例子，我们可以看到，都是-2147483648和2147483647进行比较，结果确不一样？ Why？

编译器进行解释时，对**无符号数**比较运算，对应的指令是**无符号比较运算指令**，而**带符号数**的比较运算对应的指令是**带符号比较运算指令**。这**两个指令判定规则不同**，从而使得**相同的机器数**在进行比较运算后**结果不同**。因此，在编程时一定要注意**数据类型的定义**和**编译器默认的数据类型转换规则**。

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2.3 浮点数的表示

从前面可以看到，整数表示范围是有限的。为了**扩大数的表示范围**，同时又能表示**带小数点的实数**，计算机需要支持浮点数

• **对任意实数X，可以表示成为如下形式：**

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

其中，s是符号位，M是称为X的尾数（mantissa），E是称为X的阶或指数(exponent)，R是基数（radix、base）

但是，对于现实世界中的一个实数53/1000000000，有多种表示形式：**0.053**
 $\times 10^{-6}$, **0.00053×10^{-4}** 等形式。这些不唯一的**非规格化**（Unnormalized）形式会导致什么问题？

(1) 同一个数表示不唯一，导致浮点运算很多操作难以实现，比如：比较操作

(2) **浮点数尾数的位数决定浮点数的精度**，这种非规则化表示就有可能**浪费**大量的**尾数的位数**存储没啥用处的**0**，导致数据精度下降。例如：假定尾数只能存储小数点后的2位，那么 **0.053×10^{-6}** 将会四舍五入成 **0.05×10^{-6}** ，而 **0.00053×10^{-4}** 将会四舍五入成 **0.00×10^{-4}** ，导致数据精度下降

2.3 浮点数的表示

■ 规格化 (Normalized):

比如规定**小数点前只有一位非0数**（当然也可以规格化为其他形式），例如

1.0×10^{-9}

- 左归 尾数左移1位，阶减1
- 右归 尾数右移1位，阶加1
- 直到尾数满足规格化形式

例如： 0.053×10^{-6} 和 0.00053×10^{-4} 可以通过**左规**变成 5.3×10^{-8} ，

而 530.0×10^{-10} 可以通过**右规**变成 5.3×10^{-8}

可以看到，在规格化后，这可以使得浮点数**尾数的数位**尽量地保存更多的**尾数的有效数字（避免了无用的0的存储）**，**提高数据精度**。同时，这也使得浮点数的表现形式唯一

2.3 浮点数的表示

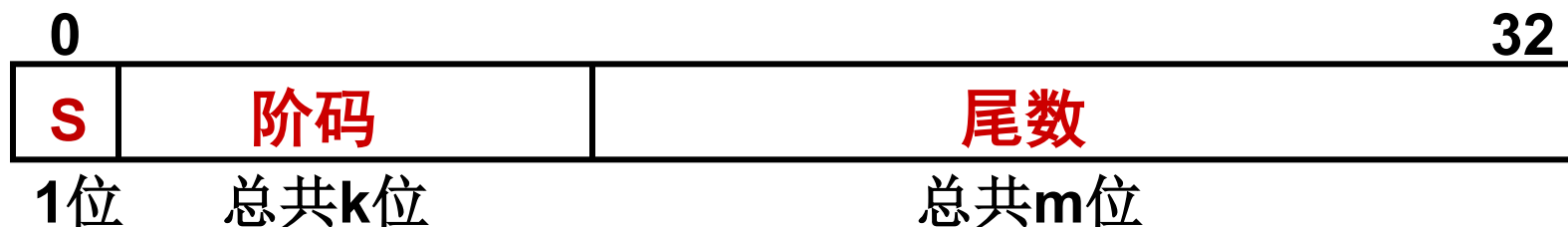
- 在计算机中，怎么表示浮点数？

- 基于前面介绍的浮点表示方法，我们在此基础上可以这样：

- 用一位数表示符号位（常用0表示正数，1表示负数）
- 用一个原码编码的二进制定点小数表示浮点数的尾数
- 用一个移码编码的二进制定点整数表示浮点数的阶
- 因为基数可以事先约定为2，所以计算机可以不用存储此信息，也就是说我们只需要存储符号位、尾数编码和阶编码就可以存储一个浮点数

2.3 浮点数的表示

- 那么对于任意二进制浮点数 $X = (-1)^s \times M \times R^E$ ，怎么存储这些编码后的数据（即**符号位**、**尾数编码**、**阶编码**）呢？
- 以32位浮点编码格式为例



S 符号位，决定浮点数是正数（ $S=0$ ）还是负数（ $S=1$ ）

阶码 k位移码编码的定点整数表示的**阶**，**阶码**=**阶E**+**偏置常数**

尾数 n位原码编码的定点小数表示的**尾数**

思考：1. 为什么先存符号位s，再存阶码，再存尾数？可以按其它顺序存储吗？有没有什么特别考虑使得这样做最好？

对阶使得尾数放最后最好。符号位放最前面方便判断数的符号和进行符号的操作（比如：乘除结果符号的计算）

2. k和m的取值会分别影响什么？

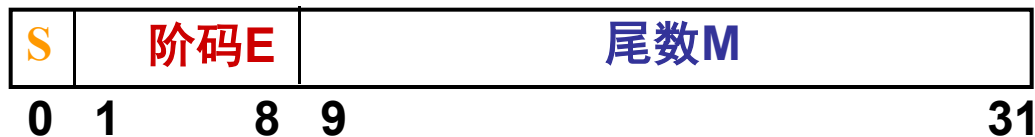
k影响浮点数表示范围。

m决定有效位数，影响精度

浮点数表示范围和精度分析的例子

这是一种32位浮点数字格式的规格化表示的例子：

假如：约定数X在规格化后，整数部分需要为0，而小数点后第一位是1，即： $X = +/-0.1xxxxx \times 2^E$

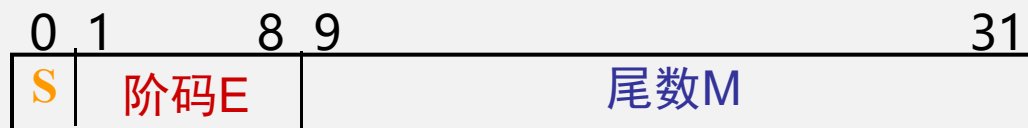


注：这里位的编号顺序从左往右依次为0~31

- 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 $2^7=128$ ；
- 第9~31位：23位二进制位

由于第一位默认的“1”可以不明显表示出来，故可用23个数位表示尾数M的24位原码小数

在这个例子，32位规格化浮点数表示范围是多少？



$$X = +/- 0.1xxxxx \times 2^E$$

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

我们也可以看到：

- 尾数M的位数反映了数X的有效位数，反映了数之间的稀疏度，决定了X的表示精度；有效位数越多，表示精度越高；
- 阶的位数决定了数X的表示范围



我们可以看到：

- 机器零：是尾数为0的数(不满足规格化形式)(包括落在下溢区中的数)，有+0和-0之分 (why?)
- 浮点数范围比定点数范围(最大32位无符号数也只是 $2^{32}-1$)大，但是浮点数能表示的数比定点数能表示的数更稀疏，且不均匀。思考：为什么更稀疏？为什么不均匀？

因为，32位编码能表示的数的个数没变多，故浮点数的数之间更稀疏。

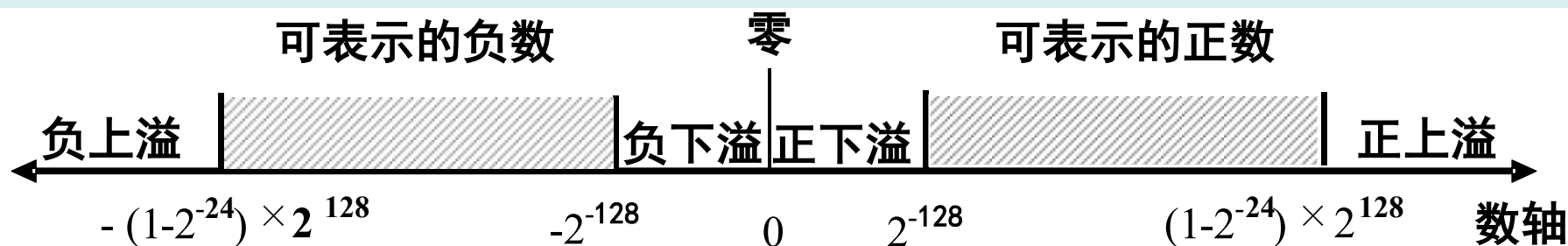
因为，对每个固定的“阶”，由于尾数位数只有23位，所以只能表示 2^{23} 个离散的数

• 思考：若将上例从 $X=+/- 0.1xxxxx \times 2^E$ 改为 $X=+/- 1.xxxxx \times 2^E$

即，如果使尾数形式为 $M=1.xx...x$ （第一位可以隐藏，即仍然用23位原码表示24位定点小数），阶码偏置常数不变，那么表示范围和精度又是多少？

最大正数： $1.11...1 \times 2^{1111\ 1111} = (2-2^{-23}) \times 2^{255-128} = (2-2^{-23}) \times 2^{127} = (1-2^{-24}) \times 2^{128}$

最小正数： $1.00...0 \times 2^{0000\ 0000} = (1+0) \times 2^{0-128} = 2^{-128}$



可以看到：浮点数的范围和精度随着规格化方法的不同而不同

总结：

- 浮点数尾数的位数决定了浮点数的精度，编码时要尽可能多的保留尾数的位数！浮点数阶的位数决定了浮点数的表示范围
- 浮点数范围和精度也与其规格化方式有关！
- 浮点数往往是规格化的，两个浮点数运算的结果也要进行“规格化”
 - 左归 当尾数出现 $\pm 0.0...0ffff$ ，需要左归。尾数左移1位，阶码减1
 - 右归 当尾数出现 $\pm ffff.ffff$ ，需要右归。尾数右移1位，阶码加1
 - 直到尾数为规格化形式

- 早期，出于**表示范围**和**精度**的原因，不同计算机产商各自定义自己的浮点数表示格式：**规格化方式**、**阶码位数**和**尾数位数**各异。

32-bit 规格化数：



- S 是符号位 (Sign)
- Exponent 用移码来表示
- Significand 表示尾数部分

计算机产商各自定义自己的浮点数表示格式。这会带来什么问题？

浮点数表示的不统一，使得不同产商生成的计算机之间在进行数据交换或程序移植时面临严重的问题

“Father” of the IEEE 754 standard

- ◆ 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

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

8087浮点处理器

IEEE 754浮点数（包括：32位单精度、64位双精度浮点数）标准：

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

S	Exponent	Significand
1 bit	8/11 bits	23/52 bits

- **Sign bit:** 1 表示负数；0表示正数
- **Exponent**（8位/11位，阶码）：**移码编码的定点整数**
偏置常数，SP: 127; DP: 1023
规格化数的阶码范围，SP: 1~254，DP: 1~2046 全0和全1用来表示特殊值！
规格化数的阶码真值范围，SP: -126 ~ 127; DP: -1022 ~ 1023
- **Significand**（23位/52位，尾数）：**原码编码的定点小数**
 - 规格化尾数最高位总是1，所以隐含表示，省1位

因此，可以有 1 + 23 bits（single，单精度），1 + 52 bits（double，双精度）表示尾数

- IEEE 754标准的浮点数的二进制机器数转换成其真值的方法：

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

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

例子: 将二进制单精度浮点表示转换成十进制真值

- 将IEEE 754标准的浮点数的二进制机器数0XBEE0 0000H转换成其真值:

10111 1101 110 0000 0000 0000 0000 0000

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

- 符号位**S**: 1 \Rightarrow 负号
- 阶码: (移码定点整数)
 - $0111\ 1101_2 = 125_{10}$
 - Bias adjustment: $125 - 127 = -2$
- 尾数: (原码定点小数)
 - $1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$
 - $= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$
- 真值: $-1.75_{10} \times 2^{-2} = -0.4375$

例子: 将十进制真值转换成二进制单精度浮点表示

- 将-12.75转换为IEEE 754标准的32位单精度规格化浮点数表示形式
 $\text{+/-}1.\text{XXXXXXXXXX} \times 2^{\text{Exponent}}$

1. 整数转换成原码定点整数:

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

2. 小数部分转换成原码定点小数:

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

3. 将两部分放在一起, 并且规格化:

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

4. 将阶表示为移码: $127 + 3 = 128 + 2 = 1000\ 0010_2$

1100 0010 100 1100 0000 0000 0000 0000

因此, -12.75对应的机器数是 **C14C0000H**

思考: 1.0和-1.0的IEEE 754标准的32位单精度规格化浮点数表示形式是什么?

+1: 0 01111111 000000000000000000000000

-1: 1 01111111 000000000000000000000000

IEEE 754浮点数表示中的特殊情况解释

阶码	尾数	值
1-254	任何值	规格化数
0	0	?
0	非零值	?
255	0	?
255	非零值	?

这些非格式化模式将代表什么呢？

1) 0的表示:

■ IEEE 754将阶码全0并且尾数全0的位序列定义为0

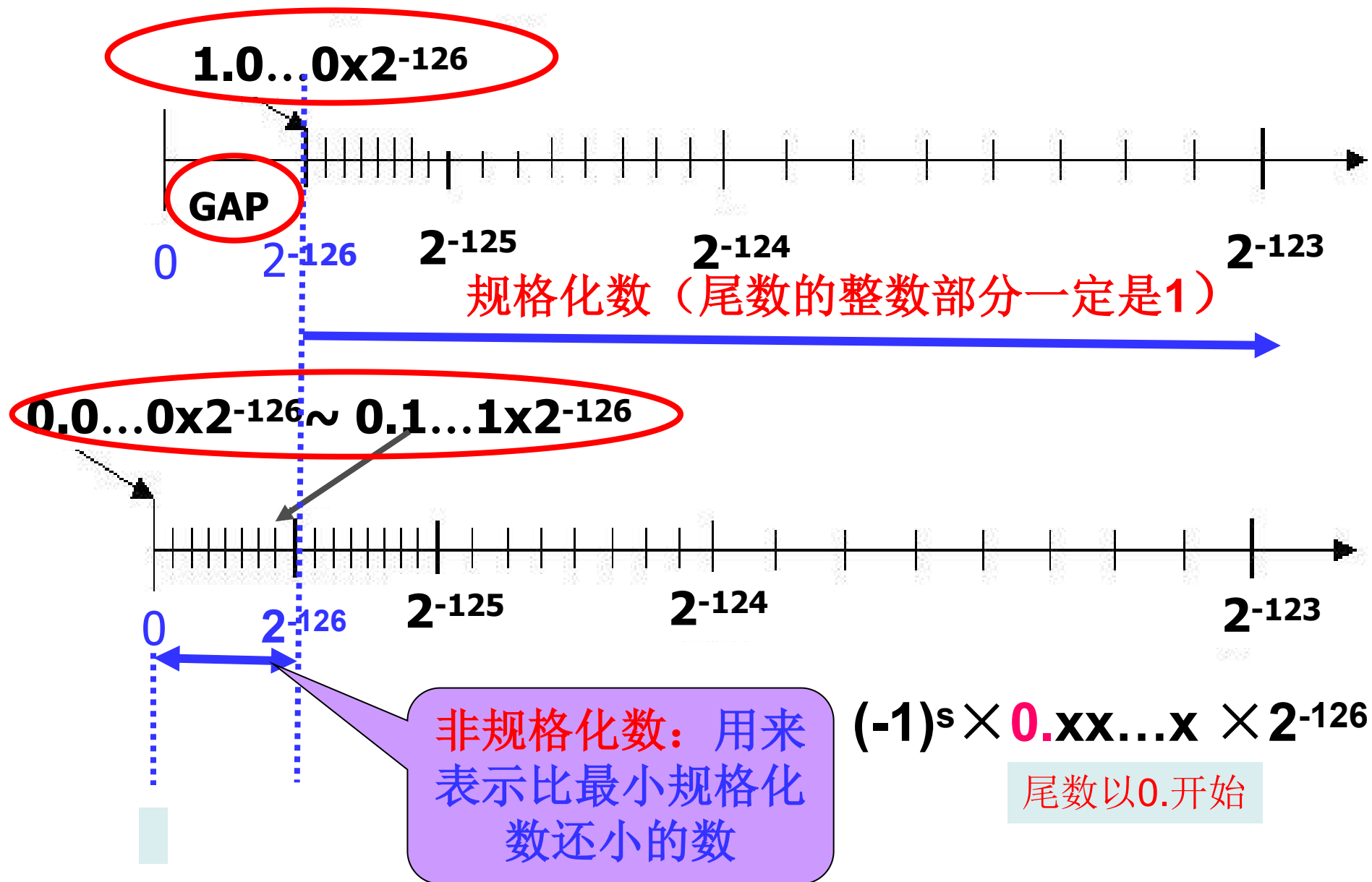
■ IEEE 754的0有两种表示:

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

• 一般情况下+0和-0等效。

2) 非规格化数



2) 非规格化数

■ IEEE 754将**阶码全0**并且**尾数不全为0**的位序列称作**非规格化数**

◆ 阶码：单精度默认为-126，双精度默认为-1022

◆ 尾数部分的数值为0.f，因此非规格化数表示的数值为：

$$\begin{aligned} & (-1)^s \times 0.f \times 2^{-126} \quad (\text{单精度}) \quad \text{或} \quad \text{注：非规格化数的} \\ & (-1)^s \times 0.f \times 2^{-1022} \quad (\text{双精度}). \quad \text{隐藏位为0} \end{aligned}$$

◆ 非规格化数区间：(0.0...0x2⁻¹²⁶, 0.1...1x2⁻¹²⁶],

最小能表达的数是0.0...01x2⁻¹²⁶ 总共有2²³-1个 注：固定阶码
= 2⁻²³ × 2⁻¹²⁶ = 2⁻¹⁴⁹

◆ 非规格化数的作用：**表示比最小规格化数(例如2⁻¹²⁶) 还小的浮点数**，弥补了0到1.0...0x2⁻¹²⁶之间的空缺，**解决规格化数计算过程阶码下溢问题**

3) $+\infty/-\infty$ 的表示

- $+\infty/-\infty$ 是两个特殊的数：**阶码全1**并且**尾数全0**

$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000

- 无穷大数既可作为操作数，也可是运算的结果。
- **作用：**
 - ◆ 引入**无穷大数**使得在计算过程出现异常时，如 $5.0 / 0$ ，也可以继续执行下去

例如，在进行如下操作时，也能产生明确的结果：

$$\begin{array}{ll} 5.0 / 0 = +\infty, & -5.0 / 0 = -\infty \\ 5 + (+\infty) = +\infty, & (+\infty) + (+\infty) = +\infty \\ 5 - (+\infty) = -\infty, & (-\infty) - (+\infty) = -\infty \end{array}$$

浮点数除0的例子 (这是网上的一个帖子)

为什么整数除0会发生异常?

为什么浮点数除0不会出现异常?

```
#include <conio.h>
#include <stdio.h>
int main()
```

```
{
    int a=1, b=0;
    printf( "Division by zero:%d\n ", a/b);
    getchar();
    return 0;
}
```

```
int main()
{
    double x=1.0, y=-1.0, z=0.0;
    printf( "division by zero:%f %f\n ", x/z, y/z);
    getchar();
    return 0;
}
```

在浮点运算中，一个数除以0，结果为正无穷大（负无穷大）

问题一：为什么整除int型会产生错误？是什么错误？

二：用double型的时候结果为1. #INF00和-1. #INF00，作何解释???

4) 非数 (NaN, Not a Number)

◆ IEEE 754将**阶码全1**并且**尾数非零**的情形定义为**NaN**, 表示一个没有定义的数

◆ 例如: 如下异常操作会产生非数

$$\text{sqrt}(-4.0) = \text{NaN} \quad 0/0 = \text{NaN} \quad 0 * \infty = \text{NaN}$$

$$\text{op}(\text{NaN}, x) = \text{NaN} \quad +\infty + (-\infty) = \text{NaN}$$

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

◆ 非数的作用:

- 编译器可以用非数作为浮点数变量的非初始化值
- 指导程序执行异常处理

4) 非数 (NaN, Not a Number)

非数分为两种： 静止的NaN和通知的NaN

◆ 尾数最高有效位为1，定义为不发信号（静止的）NaN：

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。

◆ 尾数最高有效位为0，定义为发信号(通知的) NaN：

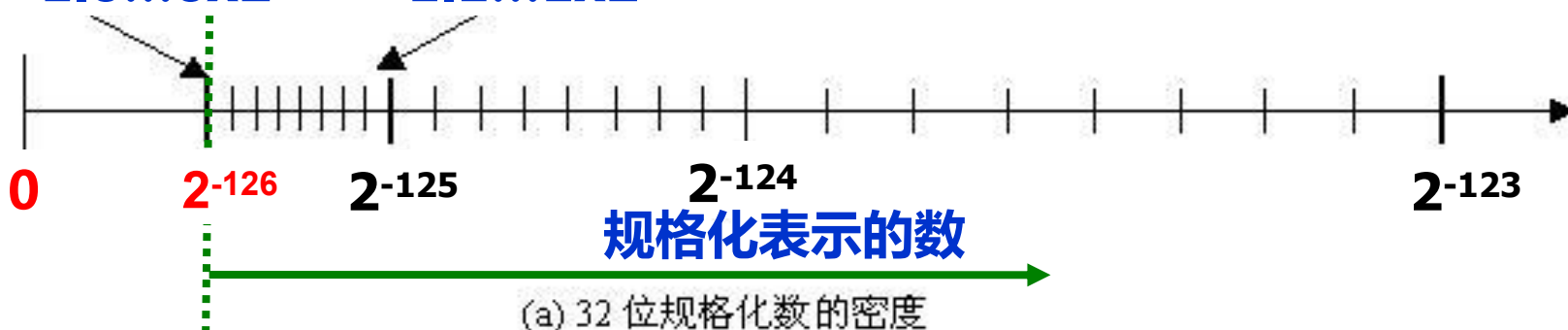
➤ 当结果产生这种非数时，发一个异常操作通知，通知系统进行异常处理

➤ 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。

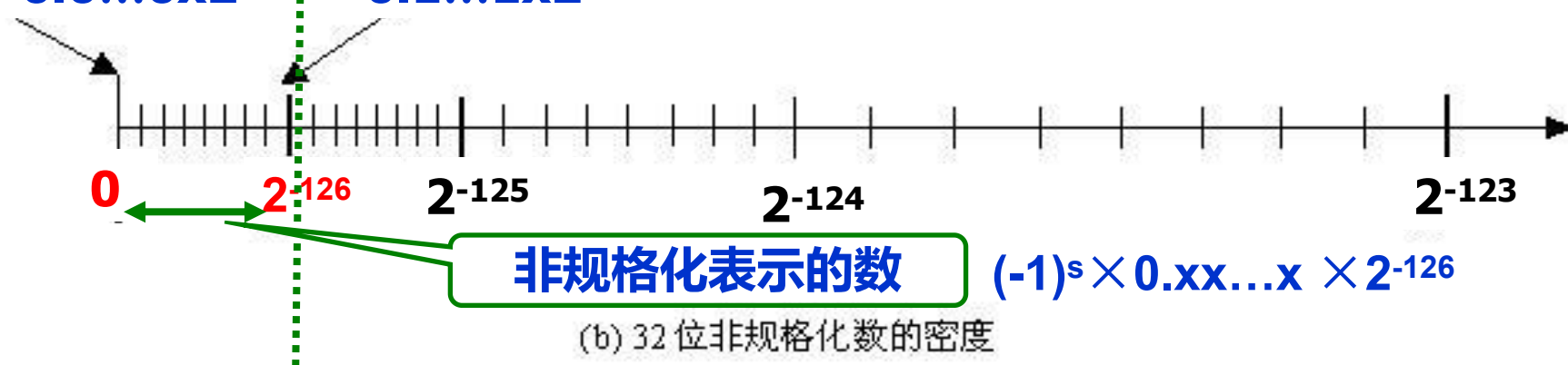
规格化数和非规格化数的范围和密度

以32位单精度IEEE 754编码表示的正数为例

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



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



- 单精度浮点数的表示范围多大? $\pm 1.11...1 \times 2^{127}$ 约 $\pm 3.4 \times 10^{38}$
- 双精度浮点数的表示范围多大? $\pm 1.11...1 \times 2^{1023}$ 约 $\pm 1.8 \times 10^{308}$
- 每个区间有 2^{23} 个不同实数, 区间段长递增 2^{-126} 、 2^{-125} 、 2^{-124} 、... 可表示实数越来越稀疏

浮点数精度的例子

当采用IEEE 754对浮点数进行表示，在其可表示浮点数的范围内，每个区间 $[2^n, 2^{n+1})$, $n \in [-126, 127]$ ，以及区间 $[0, 2^{-126})$ 中只有 2^{23} 个实数可以编码，其它浮点数不可编码，机器通过舍入将其转换成邻近的可表示数，因此浮点数表示会存在精度误差问题

```
#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;
    }

    return 0;
}
```

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，其余为不可表示的数。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

数据编码小结

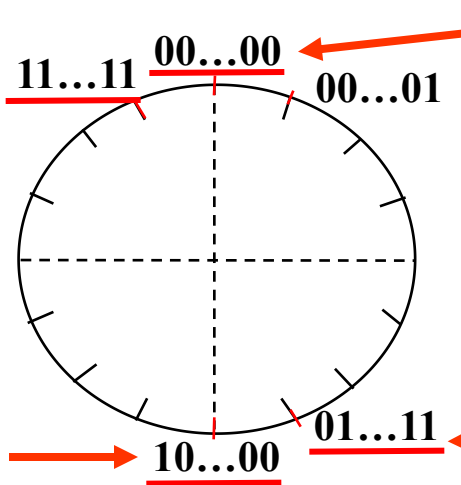
- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进位计数制、定点/浮点表示（用于解决**小数点**问题）、编码方式（用于解决**符号**问题）（常用的典型编码方式：原码、补码、移码等）

补码：在一个模运算系统中， $[X]_{\text{补}} = 2^n + X \quad (-2^{n-1} \leq X < 2^{n-1}, \text{ mod } 2^n)$

由于某数A减去小于模的另一数B，总可以用该数A加上-B的补码来实现，因此计算机系统作为一个模运算系统，在按补码编码后，可以实现+和-的统一

- C语言中的整数及其相互转换问题（类型转换前后**机器数不变**）
 - 无符号数**（所有二进制位都用来表示数值，没有符号位）；**带符号数**（补码）

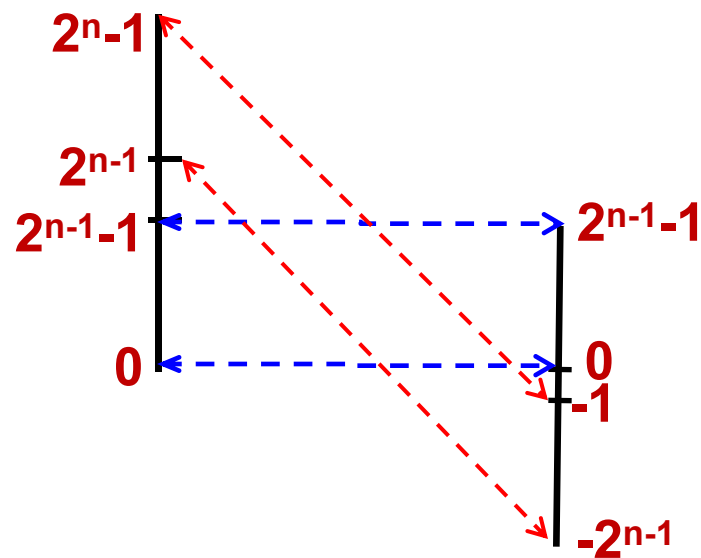
对应的**无符号数**的真值 2^{n-1}
对应的**带符号数**的真值 -1



对应的**无符号数**和**带符号数**的真值都是 0

对应的**无符号数**的真值是 2^{n-1}
对应的**带符号数**的真值是 -2^{n-1}

对应的**无符号数**和**带符号数**的真值都是 $2^{n-1}-1$



无符号数

带符号数

数据编码小结

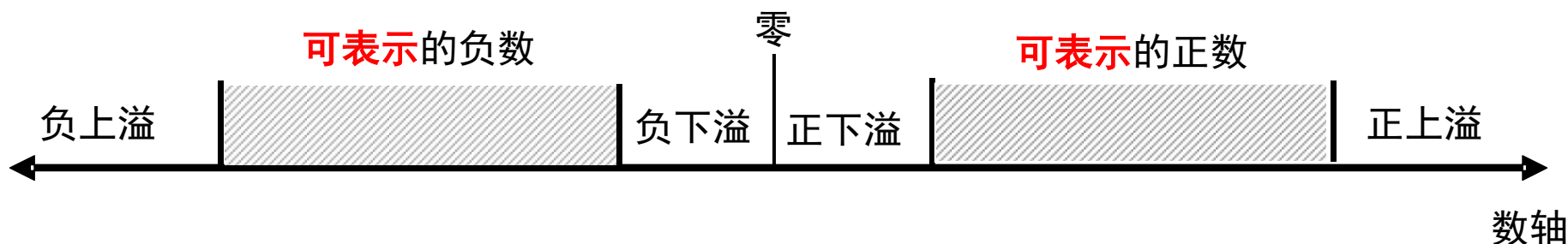
- 浮点数的表示

- 符号；尾数：定点小数表示（原码）；指数（阶）：定点整数表示（移码）



- 浮点数的范围

- 与阶码的位数有关，正上溢、正下溢、负上溢、负下溢；



- 浮点数的精度：与尾数的位数和是否规格化有关

浮点数只能表示部分数据，其它浮点数不可表示，机器通过舍入将这些不可表示的浮点数转换成邻近的可表示数，因此浮点数表示会存在精度误差问题

数据编码小结

- 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 $(-1)^s \times 1.xx...x \times 2^{\text{Exponent}}$
 - “零” (阶为全0, 尾为全0)
 - 非规格化数 (阶为全0, 尾为非0, 尾数最高位隐含为0)
 - ∞ (阶为全1, 尾为全0) 单精度: $(-1)^s \times 0.xx...x \times 2^{-126}$
 - NaN (阶为全1, 尾为非0) 双精度: $(-1)^s \times 0.xx...x \times 2^{-1022}$
 - 规范化数

1 bit	不是全0和全1	f ≠ 0或f=0都行
-------	---------	-------------

- 零

1 bit	全为0	f = 0
-------	-----	-------

- 非规范化数

1 bit	全为0	f ≠ 0
-------	-----	-------

- 无穷大

1 bit	全为1	f = 0
-------	-----	-------

- NaN

1 bit	全为1	f ≠ 0
-------	-----	-------

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2.4 十进制数的表示

- 人们日常使用的是十进制，从而在计算机外部（如：键盘输入、屏幕显示或打印输出），看到的数据基本上都是十进制形式，因此，需要计算机内部也能表示和处理十进制数据，以方便直接进行十进制数的输入输出或直接用十进制数进行计算。
- 在计算机内部表示十进制数，可以采用两种方式：
 - ASCII码字符表示
 - BCD（Binary-Coded Decimal，二进制编码的十进制数）码表示

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对应的ASCII码（30H~39H）表示。

特点：

- 一个十进制数对应8位二进制数（一个字节）；
- 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，但不方便运算**。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

由于4位二进制位串能表示16个状态, 因此需要从这16个状态中选10个表示十进制数的0~9。从而存在多种BCD码方案

◆ 特点:

- 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

表示每个十进制数位的4个二进制数位都有一个确定的权

➤ 最常用的一种是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

表示每个十进制数位的4个二进制数位没有确定的权

➤ 常用的有：格雷码、余3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码是由**8421码加3后形成的**，即在8421码基础上每位十进制数BCD码再加上二进制数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。

- **第二章：数据的机器级表示与处理**

- 2.1 数值数据的表示

- 进位计数制
 - 定点表示/浮点表示
 - 定点数的编码表示（原码、补码、移码、反码）

- 2.2 整数的表示

- 无符号整数和带符号整数的表示
 - C语言中整数及其相互转换

- 2.3 浮点数的表示

- 主要介绍IEEE 754浮点数标准

- 2.4 日常中十进制数的表示（例如：ASCII码0~9）（自学）

- 2.5 非数值数据的编码表示（自学）

2.5 非数值数据的编码表示

非数值数据（例如**逻辑值**、**字符**等）是没有大小之分的位串（在机器内部它们用一个**二进制位串**表示），不表示数量的多少

1. 逻辑值的编码表示

逻辑数据**按位对待的数据**，以位串的形式存在

- 一个逻辑值用1个位表示，取值1或0
- N位二进制数可表示N个逻辑数据
- 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列。

那么计算机如何识别逻辑数据？

计算机靠指令（专门进行逻辑运算的指令）来识别，其中的操作数当作逻辑数据、按位使用。

逻辑数据的运算包括：**按位与**、**按位或**、**逻辑左移**、**逻辑右移**等位运算

2.西文字的编码表示

- 西文字符的特点

- 西文字符集：26个英文字母（大小写）、数字及数学符号、标点符号等辅助字符
- 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（最广泛常用编码为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： b₅位为0是大写字母，b₅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

a) 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。

b) 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
- **机内码**（内码）：用于在系统对汉字进行存储、查找、传送

c) 汉字内码

- 由于汉字总数多，一个汉字内码需用2个字节才能表示
- 汉字内码是在区位码（**区位码**对汉字进行编码方式：汉字被对应到94行X94列的码表中，而每个汉字由7位区号+7位位号区分）的基础上产生的

注：出于信息传输的原因，GB2132标准在每个汉字的区号和位号各自加上20H，得到**国标码**，以实现汉字信息在不同计算机系统之间进行交换使用。为与ASCII码区别，通常将国标码的两个字节的第一位都设置为“1”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： 001 0100 101 0011B 即1453H

国标码： 0011 0100 0111 0011B 即3473H

即：在**区位码**的基础上，**区号和位号各加20H**。

问题：国标码前面的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：我们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。

d) 汉字的字模点阵码和轮廓描述

■ 为便于打印和显示汉字，**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。
 - ◆ 汉字笔画的轮廓用一组直线和曲线勾画, 它记下每一直线和曲线的数学描述公式。
 - ◆ 精度高, 可任意放缩。

■ 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!

- **第二章：数据的机器级表示与处理**

- **2.6 数据的宽度和存储**

- 数据的宽度和单位
 - 数据的存储和排列顺序（特别是：**大端方式/小端方式**）

2.6 数据的宽度和存储

a) 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)

- 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*)

- 字节是计算机二进制信息的基本计量单位，一个字节是由8个bit组成的“位组”

- 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

-
- 由于二进制信息的**存储容量**通常需要很多的字节，因此**存储容量**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

● 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

b) “字长”

—— 字长和字是不同的概念

■ **字长**：指CPU内部用于整数运算的数据通路的宽度。

数据通路：指CPU内部数据流经的路径，以及路径上的部件，包括
数据运算ALU、寄存器和总线等部件

也就是说，**字长**等于CPU内部运算器位数和通用寄存器宽度

通俗的说法：某台机器是32位机或64位机，其中的32或64指的就是字长

■ **字**：表示被处理的信息的单位，用来度量各种数据类型的宽度。

例如，单字宽度的无符号数或带符号数（就是说这个数是16位的）

双字宽度的无符号数或带符号数（就是说这个数是32位的）

■ 从上面定义，我们可以看到，**字和字长是不同的概念。字和字长可以有**
一样的大小、也可以有不同大小，如IA-32中字为16位，字长为32位

例如：C语言中数值数据类型的宽度（单位：字节）

— 同一种类型的数据的宽度可能会随**机器字长**和**编译器**的不同而不同

— 因此，我们在相应的机器上进行**数据处理**时，必须确定**此数据相应的机器级表示方式**

C声明	典型32位 机器（即字 长是32位）	典型64位 机器（即字长是 64位）
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

c) 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 $\text{int } i = -65535$ ， $[-65535]_{\text{补}} = \text{FF FF } 00 \text{ } 01\text{H}$ 。若 i 存放在100号单元开始的4个字节里，问100号地址开始的 4个字节存储单元的内容各是什么？

字节地址: **100** 101 102 **103**

存储方式1:

FF	FF	00	01
----	----	----	----

存储方式2:

01	00	FF	FF
----	----	----	----

MSB LSB
设 int i = -65535, [-65535]补=FF FF 00 01H

LSB表示最低有效字节

MSB表示最高有效字节

字节地址: 100 101 102 103

存储方式1:

FF	FF	00	01
----	----	----	----

 MSB LSB

大端方式 (Big Endian) : 将最低有效字节存放在大地址单元中, 而将数据的最高有效字节存放在小地址单元中。
此时, MSB所在的地址就是这个数的地址。

例如: IBM 360/370机器

字节地址: 100 101 102 103

存储方式2:

01	00	FF	FF
----	----	----	----

 LSB MSB

小端方式 (Little Endian) : 将最低有效字节存放在小地址单元中, 而将数据的最高有效字节存放在大地址单元中。
此时, LSB所在的地址就是这个数的地址。

例如: Intel 80x86机器

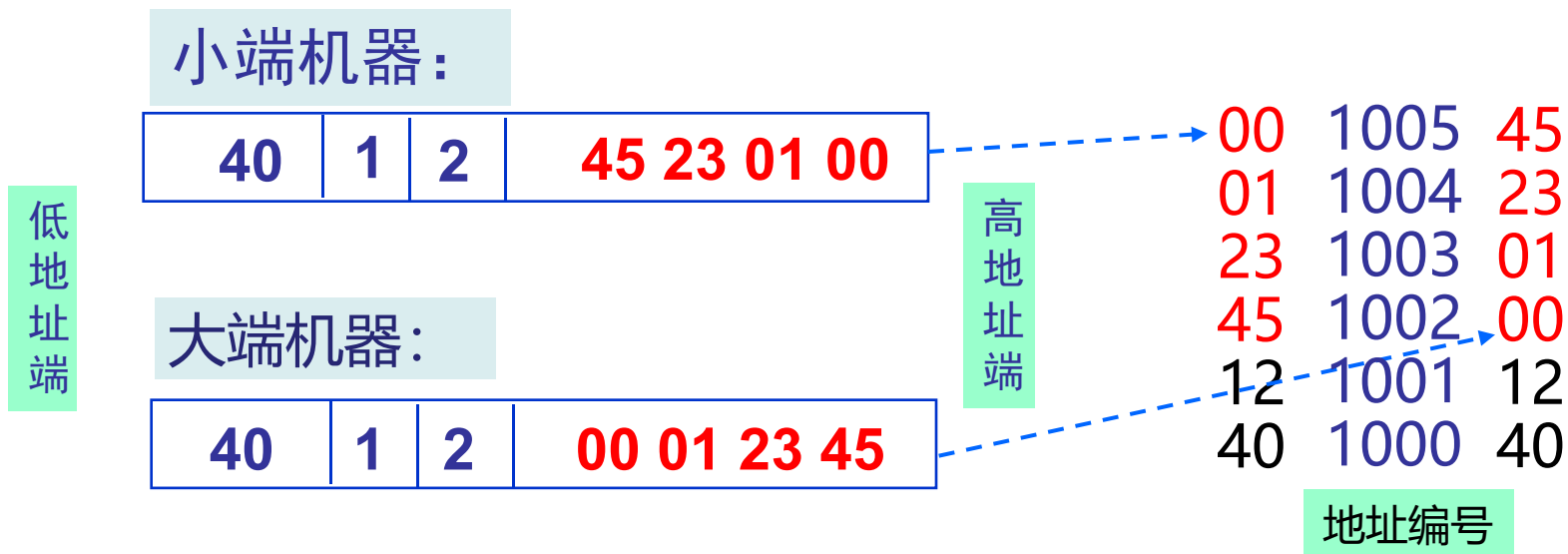
机器之间的字节交换问题

例如: 设内存中下述指令的存放地址是1000

`mov AX, 0x12345(BX)`

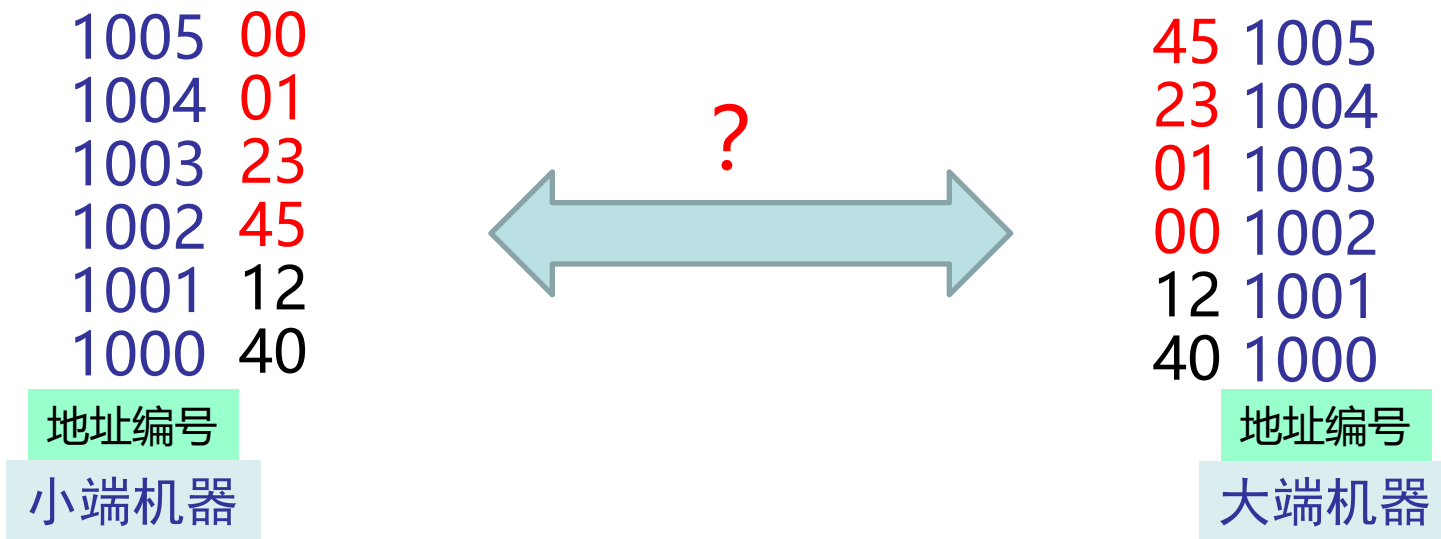
其中, 操作码mov为**40H**, 寄存器AX和BX的编号分别为**0001B**和**0010B**, 立即数**0x12345**占**32位**, 则这条指令在内存中存放方式是什么?

注: 只需要考虑指令中立即数的顺序!



机器之间的字节交换问题

- 虽然每个系统内部数据排列顺序是一致的，但存放方式不同的机器间程序移植或数据通信时，又会发生什么问题呢？



很明显，在排列顺序不同的系统之间进行程序移植或数据通信时，需要进行顺序转换

- 此外，音频、视频和图像等文件格式读写或处理程序都涉及到字节顺序问题
例如：Little endian: GIF、Microsoft RTF等
Big endian: JPEG、Adobe Photoshop等

- **第二章：数据的机器级表示与处理**

- 2.7 数据的基本运算

- 按位运算和逻辑运算
 - 左移运算和右移运算
 - 位扩展运算和位截断运算
 - 无符号和带符号整数的加减运算
 - 无符号和带符号整数的乘运算
 - 无符号和带符号整数的除运算
 - 浮点数的加减运算
 - 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

2.7 数据的运算

1. 按位运算

- **为什么需要按位运算**

- 计算机中所有的信息（包括非数值型数据）都可以看作是以**位向量**表示的，所以需要支持**按位运算**：

- **位向量及按位运算可以描述集合的子集及其运算**
 - **位向量与掩码的运算可以选择或屏蔽位向量中的一些信息**
 - **甚至，整数和浮点数的算术运算（例如：整数乘除运算）都可以转换为等价的位向量运算**

位向量

- 位向量

位向量指有固定长度 w ，由0、1组成的串

- 位向量运算

位向量 x 和 y 的运算定义为 x 和 y 的每个对应元素之间的运算

- 按位或: “|”
- 按位与: “&”
- 按位取反: “~”
- 按位异或: “^”

- 位向量运算示例

设 $w=4$, $a=[0110]$, $b=[1100]$

$a|b=1110$, $a\&b=0100$, $\sim b=0011$, $a^b=1010$

按位运算示例

- 示例1：表示有限集和实现集合运算

思考：如果没有按位运算操作，你会如何实现求两个集合的交集和并集？

假设 $A=\{0,1,2,\dots,7\}$, $x_i=1$ 当且仅当 $i\in A$

$a=[01101001]$, 代表子集 $A1=\{0,3,5,6\}$

$b=[01010101]$, 代表子集 $A2=\{0,2,4,6\}$

$a\&b=[01000001]$, 代表 $A1\cap A2=\{0,6\}$

$a|b=[01111101]$, 代表 $A1\cup A2=\{0,2,3,4,5,6\}$

- 示例2：选择信号

假设位向量 a 中存放外界信号，要选取 a 中第0、2、4、6位的4个信号，你会怎么做？

我们可以设置 $b=[01010101]$ ，然后用 $a\&b$ 即可达到目的

- 示例3：如何从16位采样数据 y 中提取高位字节？

我们可以通过构造掩码 $0xFF00$ 和用“&”实现

例如：当 $y=0x2C0B$ 时，有 $x=y \& 0xFF00$ 得到结果为： $x=0x2C00$

逻辑运算

• 逻辑运算 ||、&&、!

逻辑运算和按位运算有什么不同？

➤ 操作参数和结果的意义不同

逻辑运算认为参数非0为TRUE，参数为0代表FALSE

按位运算只有在参数被限定为0或1时，才与逻辑运算相同

!0x00=0x01（逻辑运算）， ~0x00=0xFF（位运算）

0x69&&0x0F=0x01（逻辑运算）， 0x69&0x0F=0x09（位运算）

!(a^b) 等价于 a==b（混合运算）

➤ 逻辑运算与按位运算在计算表达式时不同

若第一个参数可以确定表达式的值，则逻辑运算不对第二个参数求值。然而，按位运算会按表达式完成所有操作。

假设a=0x00，则a&&5/a=0x00，不会发生除以0

a&5/a=? 发生除0错误

假设p=NULL，则p&&*p++不会发生间接引用空指针

2. 移位运算

➤ 左移: $x \ll k$

➤ 右移: $x \gg k$

◆ 移位运算分为逻辑移位和算术移位

操作	值
参数x	[0110 0011]、[1001 0101]
$x \ll 4$ (逻辑或算术左移)	[0011 0000]、[0101 0000]
$x \gg 4$ (逻辑右移)	[0000 0110]、[0000 1001]
$x \gg 4$ (算术右移)	[0000 0110]、[1111 1001]

◆ C语言根据数据的类型确定是逻辑移位还是算术移位:

➤ x是无符号整数: 逻辑移位

➤ x是带符号整数: 算术移位

➤ 移位运算用途:

- 提取部分信息 (例如: 没用的信息移出, 留下有用的信息)
- 扩大或缩小数值的2、4、8...倍

例如: $x \ll k = x * 2^k$

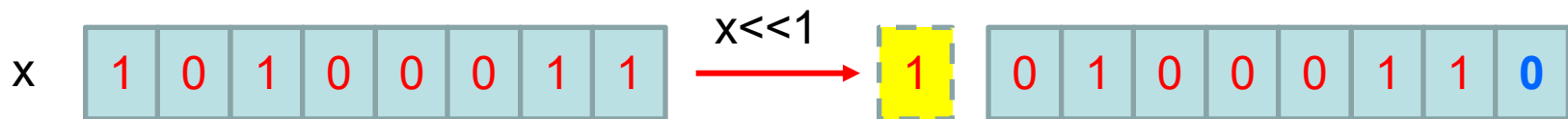
$x \gg k = x / 2^k$

在通过移位来扩大或缩小数值时，可能发生**溢出**：

思考：那么怎么判断是否**溢出**呢？

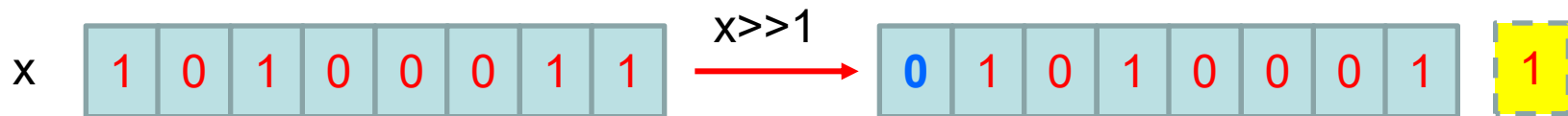
➤ **逻辑移位**：无符号整数 乘2（左移一位）或除2（右移一位）

➤ **逻辑左移**：高位移出，低位补0；汇编指令：SHL



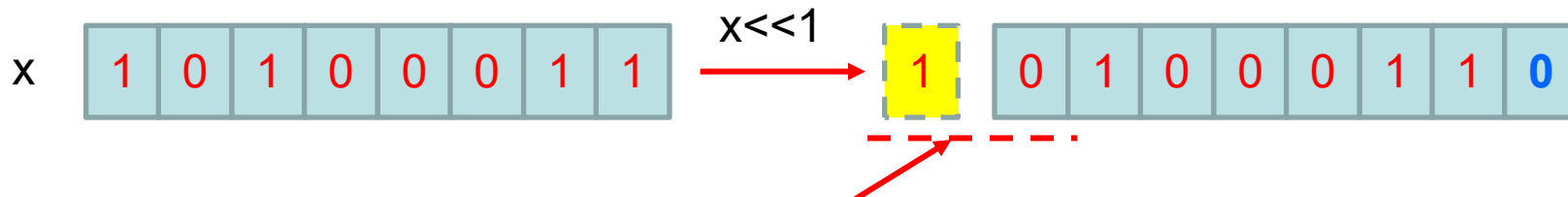
注：若逻辑左移**移出的位含1**（说明超出了无符号数表示范围），则称发生“**溢出**”

➤ **逻辑右移**：低位移出，高位补0；汇编指令：SHR



➤ 算术移位：带符号整数：乘2（左移一位）或除2（右移一位）

➤ 算术左移：高位移出，低位补0；汇编指令：SAL



注：若算术左移，**移位后的符号位不同于以前的符号位**，则发生**溢出**。

思考：这时候为什么说明发生溢出？

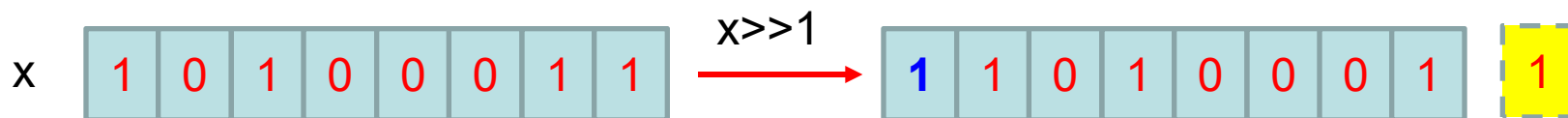
当 $-2^{n-1} \leq x \leq 2^{n-1}-1$ 时, $-2^{n-1} \leq x \cdot 2^k \leq 2^{n-1}-1$, 才说明不溢出(即没超出带符号数表示范围)
也就是说：

如果是 $0 \leq x \leq 2^{n-1}-1$ (机器码是 00...0 到 01...1 之一) 时, $0 \leq x \cdot 2^k \leq 2^{n-1}-1$ (机器码也是 00...0 到 01...1 之一), **不溢出**;

如果是 $-2^{n-1} \leq x < 0$ (机器码是 10...0 到 11...1 之一) 时, $-2^{n-1} \leq x \cdot 2^k < 0$ (机器码也是 10...0 到 11...1 之一), **不溢出**;

综上所述，要想不溢出，需要保证没有发生**符号位的相反值**移到**最高位**或**移出**的现象

➤ 算术右移：低位移出，高位补符号；汇编指令：SAR



移位运算应用示例

例如：已知32位寄存器中存放的变量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是**无符号整数**，所以x的真值是：

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

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

另外，因为x的机器数是：

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100B$$

所以，**由x逻辑右移1位可以得到机器数是**：

$$0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010B = 40\ 00\ 00\ 02H$$

按**无符号数**解释，该机器数的真值也是： $2^{30} + 2^1$

可以看到，对于无符号整数，**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。**

移位运算应用示例

(3) 因为 $x=2^{31}+2^2$,

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

另外, 因为 x 机器数是:

1000 0000 0000 0000 0000 0000 0000 0100B

所以, x 逻辑左移1位得到的机器数是:

0000 0000 0000 0000 0000 0000 0000 1000B = 00 00 00 08H

按无符号数解释, 该机器数的真值是: 8

可见, $2^{32}+2^3 \neq 8$, 这是因为结果溢出, 即 $2x$ 的值超出无符号数最大可表示值 $2^{32}-1$ 。

因此, 如果在逻辑左移时, 移出的位有1, 则表示溢出

移位运算应用示例

例如：已知32位寄存器中存放的变量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是带符号整数，由于最高位为1，因此机器数80 00 00 04H是负数的补码，所以x的真值是：

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

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

另外，因为x的机器数是：

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100B$$

所以，由x算术右移1位得到的机器数是：

$$1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010B = C0\ 00\ 00\ 02H$$

该机器数的真值为： $-(2^{30}-2^1)$

所以，对于带符号整数，根据x的真值求出的x/2的值与对x的机器数进行算术右移得到的x/2的值是一样的。

移位运算应用示例

(3) 因为 $x = -(2^{31} - 2^2)$,

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

另外, 因为 x 的机器数是:

1000 0000 0000 0000 0000 0000 0000 0100B

所以, x 算术左移1位得到的机器数是:

0000 0000 0000 0000 0000 0000 0000 1000B = 00 00 00 08H

该机器数的真值是: 8

可见, $-(2^{32} - 2^3) \neq 8$, 这是因为结果溢出, 即 $2x$ 的真值 $-(2^{32} - 2^3)$ 比最小可表示值 -2^{31} 还要小。

因此, 对于算术左移一位, 移位前、后符号位不同, 表示溢出

3. 位扩展和位截断运算

- C语言不同类型数据占用字节数不同，在占用字节数不同的各数据类型之间转换时，应该如何处理？
 - 例如：short类型转换成int/unsigned int类型
 - 例如：int/unsigned int类型转换成short类型
- C语言中虽然没有位扩展和截断运算符，但是编译器在编译时会根据类型转换前后数据长短确定是扩展还是截断
 - 扩展：若遇到一个占用字节数少的数向占用字节数多的数转换，则进行位扩展运算
 - 无符号数： 0扩展，前面补0
 - 带符号整数： 符号扩展，前面补符号
 - 截断：若占用字节数多的数转换成占用字节数少的数则进行位截断运算
强行将高位丢弃

截断会带来什么问题？

- 1) 截断一个数可能会因为溢出而改变初值。当位截断时，如果发生溢出，称为截断溢出。
- 2) 截断溢出只会导致程序出现意外结果，而不会导致任何异常或错误报告，因此，错误的隐蔽性很强，编程的时候需要特别注意。

位扩展操作的例子：

输出下面si, usi, i, ui的十进制和十六进制值

short si = -32768 (**机器数：80 00**) ;

unsigned short usi = si; //类型转换

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：

输出下面i, si, j的十进制和十六进制值

```
int i = 32768 (机器数: 00 00 80 00) ;
```

```
short si = (short)i; //截断
```

```
int j = si; //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
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

- **第二章：数据的机器级表示与处理**

- **2.7 数据的基本运算**

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加减运算
- 无符号和带符号整数的乘运算
- 无符号和带符号整数的除运算
- 浮点数的加减运算
- 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

4. 整数加减运算

- ◆ 整数包括无符号整数和带符号整数。
- ◆ 在计算机系统中，**无符号整数和带符号整数的加减运算电路是完全一样的**，都是采用**整数加减运算器**

思考：怎么样设计**整数加减运算器**才能达到此目的？

4. 整数加减运算

回顾：根据补码时学的知识有：

对于带符号数 x 和 y ， $x+y$ 可以用 x 的补码加上 y 的补码实现，即 $x+y$ 在计算机上可以这样计算： $[x+y]_{\text{补}} = [x]_{\text{补}} + [y]_{\text{补}}$

$x-y$ 可以用被减数 x 的补码加上减数 y 的负数的补码来实现，其中，已知 $[y]_{\text{补}}$ 求 $[-y]_{\text{补}}$ 可以通过对 $[y]_{\text{补}}$ 各位取反，末位加1得到。即， $x-y$ 在计算机上可以这样计算： $[x-y]_{\text{补}} = [x+(-y)]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} = [x]_{\text{补}} + \overline{[y]_{\text{补}}} + 1$ ，其中 $\overline{[y]_{\text{补}}}$ 就是对 $[y]_{\text{补}}$ 各位取反得到的结果

另外，其实，上面原理对于**无符号数**也成立。例如：我们可以将 n 位无符号数表示的数**看作是** $n+1$ 位补码表示的正数（只不过溢出判断不一样）。

因此，不管是带符号数还是无符号数，假设 X 和 Y 分别是 x 和 y 的机器数，即补码（带符号数）或无符号数表示（无符号数），我们都有：

$$x+y=X+Y$$

$$x-y=X+\overline{Y}+1, \text{ 其中 } \overline{Y} \text{ 就是对 } Y \text{ 各位取反得到的结果}$$

4. 整数加减运算

◆ 思考：怎么样才能基于如下知识实现**整数加减运算器**？

不管是**带符号数**还是**无符号数**，令 $[x]_{\text{机器数}} = X$ ； $[y]_{\text{机器数}} = Y$ ，

我们都有： $x+y = X+Y$ ， $x-y = X+\overline{Y}+1$

分析：可以看到，进行加减运算的时候，其实可以使用多路选择器使得它们的计算公式统一

具体来说，我们可以用**sub=0**表示进行**加法**运算，并且此时使得 $Y' = Y$

用**sub=1**表示进行**减法**运算，并且此时使得 $Y' = \overline{Y}$

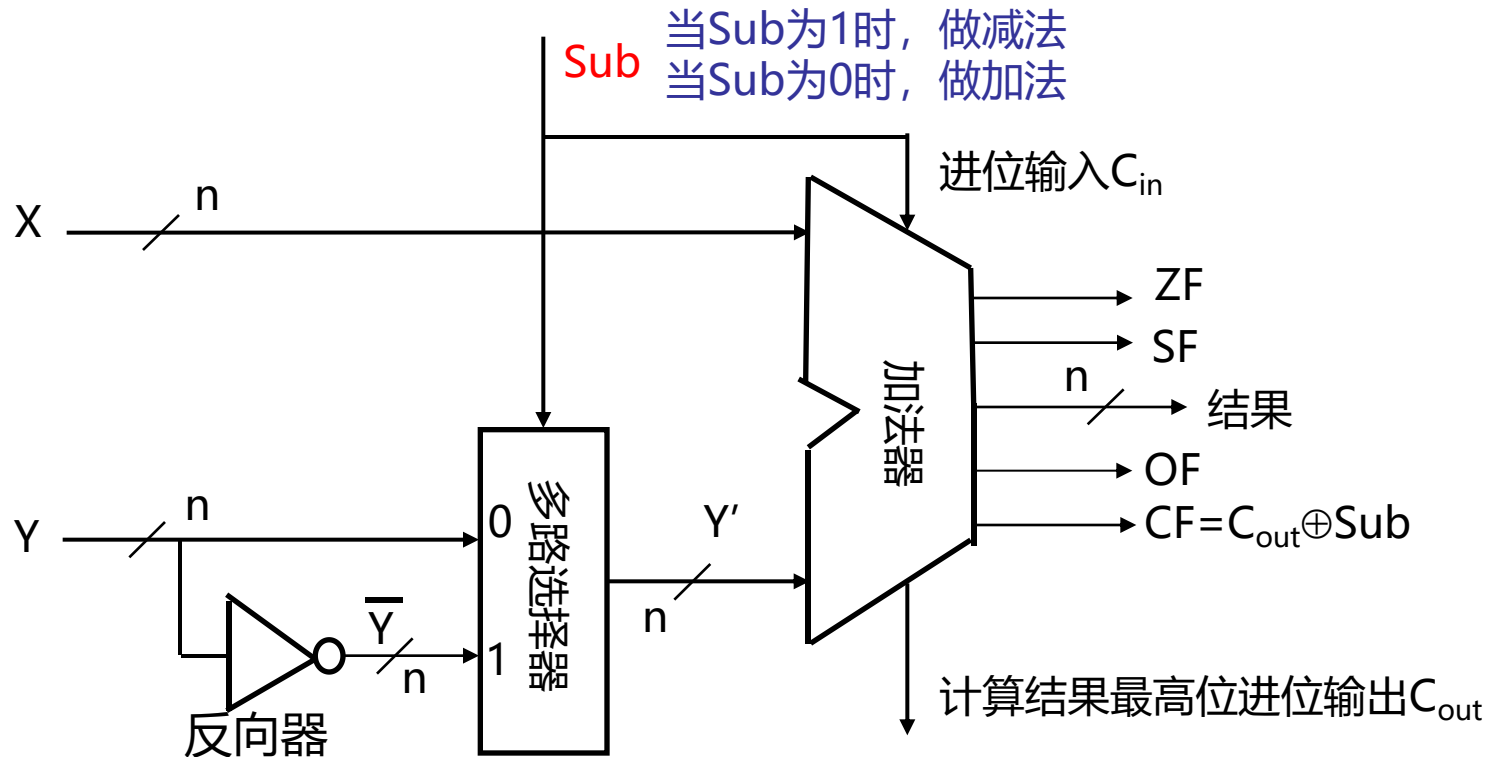
也就是说，可以通过**选择器**来**根据sub的值**，**选择**使得 $Y' = Y$ 或 \overline{Y}

这样就有：当**sub=1**时， $x-y = X+\overline{Y}+1 = X+ Y' +1 = X+ Y' +\text{sub}$

当**sub=0**时， $x+y = X+Y = X+Y+0 = X+ Y' +0 = X+ Y' +\text{sub}$

可以看到，**不管什么情形**，我们**只需要**计算 $X+ Y' +\text{sub}$ ，就能**根据sub的值**计算出需要的结果（即sub=1时，计算出x-y；sub=0时，计算出x+y）

n位整数加减运算器具体实现



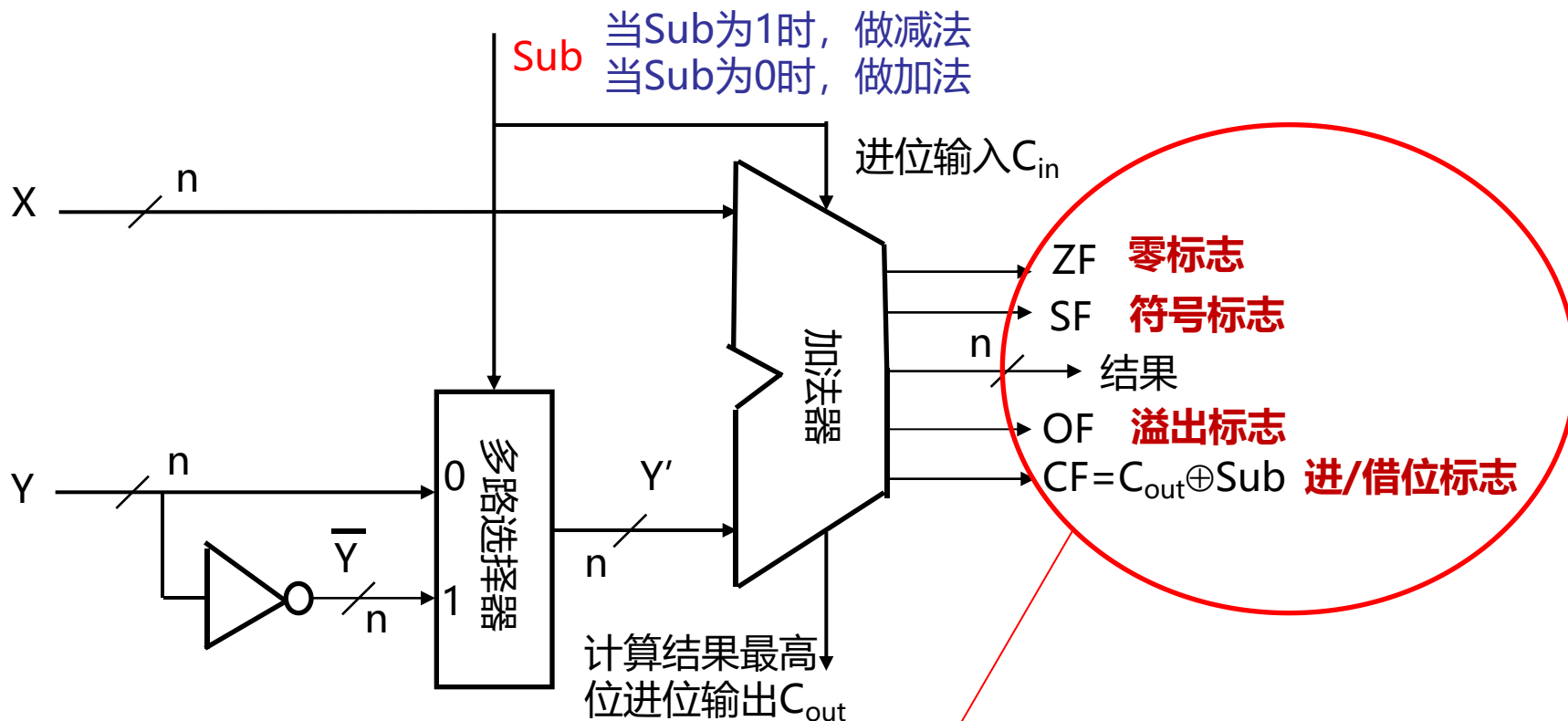
◆ X和Y：是n位的0/1序列，是需要计算的真值x、y对应的**机器数**，即补码（带符号数）或无符号数表示（无符号数）

◆ Sub是操作控制信号(**由控制器在指令译码后发出**)，Sub=0做加法，Sub=1做减法

□ 当Sub=0时， $Y'=Y$ ，从而有 $X+Y'+Sub = X+Y+Sub = X+Y+0 = x+y$

□ **反向器**负责对Y的各位取反获得 \bar{Y} 。当Sub=1时， $Y' = \bar{Y}$ ，并将Sub=1作为加法器的进位 C_{in} 送到加法器，从而实现 $X+Y'+Sub = X+\bar{Y}+Sub = X+\bar{Y}+1 = x-y$

n位整数加减运算器



问题：为什么在生成计算结果时，还需要生成标志位？

- 为了判断加减运算器计算结果在“溢出”时，是否与期望的结果一致 (因为有时候溢出是为了做模运算保证结果正确性，有时候是不期望的真溢出)。特别是，整数加减运算器不知道所运算的是带符号数还是无符号数，如果X和Y的机器数相同，那么在加法 (或减法) 运算后结果的机器数也会相同。然而，它们的溢出判断方法不同，因此需要多个标志位区分各种情况。
- 为了在条件转移指令执行时，用作判断是否转移执行的条件，例如：执行 if ($x < y$)

条件转移指令根据标志位进行转移

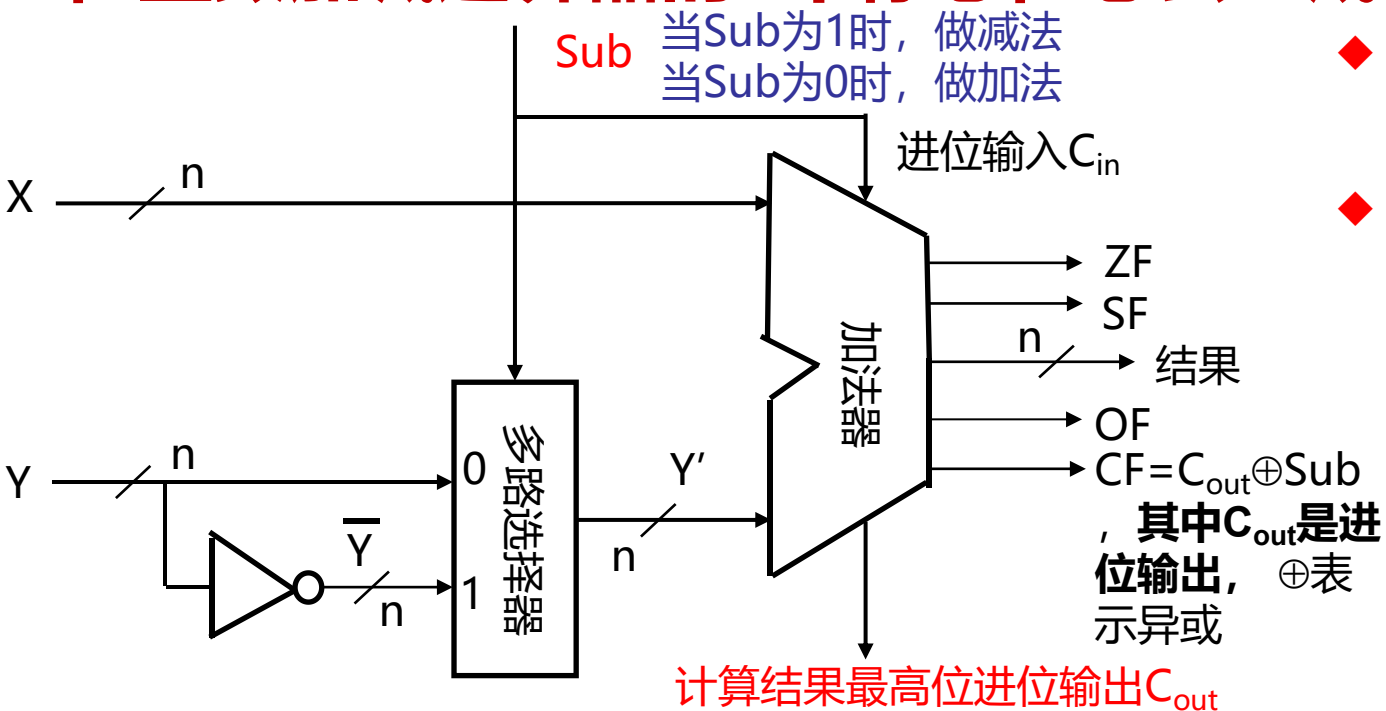
(1)根据单个标志的值转移

(2)按无符号整数比较转移

(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 $A > B$
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 $A \geq B$
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 $A < B$
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 $A \leq B$
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 $A > B$
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 $A \geq B$
15	jl/jnge label	SF \neq OF AND ZF=	带符号整数 $A < B$
16	jle/jng label	SF \neq OF OR ZF=1	带符号整数 $A \leq B$

n位整数加减运算器的4个标志位怎么生成？



- ◆ **ZF**：零标志，如果结果为0，ZF置1，否则ZF=0
- ◆ **OF**：溢出标志，只对带符号整数有效。当X和Y'+ C_{in} 最高位相同但不同于结果最高位时，OF置1，表示发生溢出（即只需要考虑“正+正”和“负+负”情况，其余情况不可能溢出）；否则OF=0

- ◆ **SF**：符号标志，只对带符号整数有效。表示带符号整数加减运算结果的符号位，即SF=结果的最高位（即结果的符号位）

- ◆ **CF**：加减运算时的进/借位标志，只对无符号整数有效。 $CF = C_{out} \oplus Sub$ ，其中 C_{out} 是进位输出， \oplus 表示异或。在加法时，CF=1（即 $C_{out}=1$ ）表示有进位，对于无符号数加法表示结果溢出；在减法时，CF=1（即 $C_{out}=0$ ）表示不够减（或称作“有借位”）

思考：为什么CF这么计算能表达这些意思？

对于无符号数加法运算， $C_{out}=1$ 说明 $X+Y'+Sub=X+Y+0=x+y \geq 2^n$ ，即 $x+y$ 的计算结果溢出；

对于无符号数减法运算， $C_{out}=0$ 说明 $X+Y'+Sub=X+\overline{Y}+1=x+(2^n-y)=x-y+2^n < 2^n$ ，

即 $x-y < 0$ （即不够减，有借位）。所以，可以结合ZF和CF判断无符号数x和y的大小。

思考：为什么CF不能用于带符号数判溢出？

因为CF=1对于带符号数而言可能是期望的溢出以实现模运算，例如：正数（2）+负数（-1）不溢出，但是CF为1

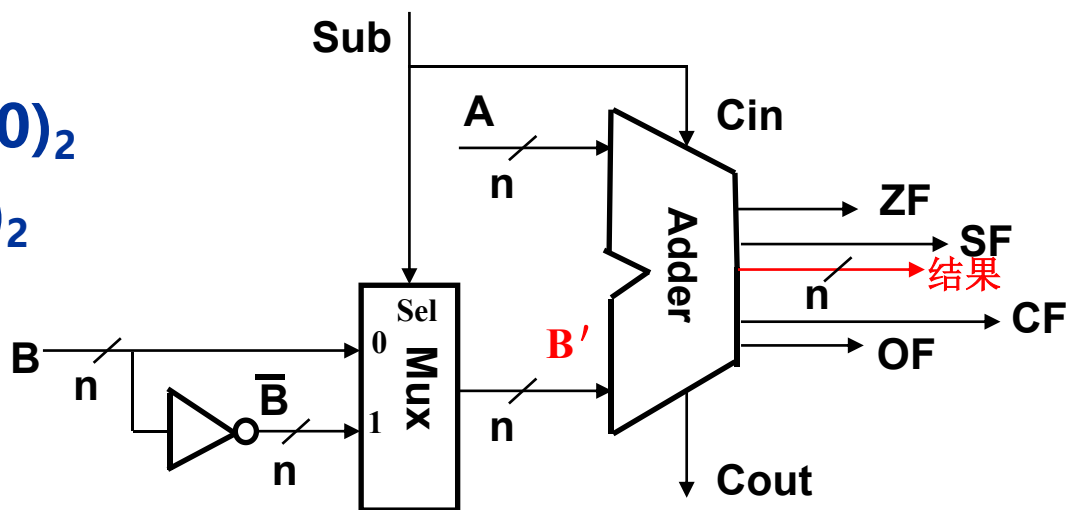
整数加减运算器中无/带符号数的加/减溢出举例

若 $n=8$, 无符号数

$$(166)_{10} = A6H = (1010\ 0110)_2$$

$$(63)_{10} = 3FH = (0011\ 1111)_2$$

$$166+63=?\quad 166-63=?$$



整数加减运算器

$$\begin{array}{r} 166_{10} = 1010\ 0110_2 \quad A \\ + \quad 63_{10} = 0011\ 1111_2 \quad B' + Cin \\ \hline \quad \boxed{0}1110\ 0101 \end{array}$$

进位 $C_{out}=0$ 结果=229 $CF=0$ $OF=0$

$$\begin{array}{r} 166_{10} = 1010\ 0110_2 \quad A \\ + \quad [-63_{10}]_{\text{补}} = 1100\ 0001_2 \quad B' + Cin \\ \hline \quad \boxed{1}0110\ 0111 \end{array}$$

进位 $C_{out}=1$ 结果=103 $CF=0$ $OF=1$

两个无符号数 x 和 y 相加/减

□ 如果 $x \pm y \in [0, Umax_n]$, 则无溢出, 此时有: 进/借位标志 $CF=0$, 而 OF 可为0或1, 结果 $=x \pm y$

整数加减运算器中无/带符号数的加/减溢出举例

若 $n=8$, 无符号数

$$(166)_{10} = A6H = (1010\ 0110)_2$$

$$(255)_{10} = FFH = (1111\ 1111)_2$$

$$166 + 255 = ? \quad 166 - 255 = ?$$

$$\begin{array}{r} 166_{10} = 1010\ 0110_2 \quad A \\ + \quad 255_{10} = 1111\ 1111_2 \quad B' + Cin \\ \hline \end{array}$$

$$\begin{array}{r} \quad \quad \quad \boxed{1} 1010\ 0101 \\ \hline \end{array}$$

进位 $C_{out}=1$ 结果 $=165=421-256$

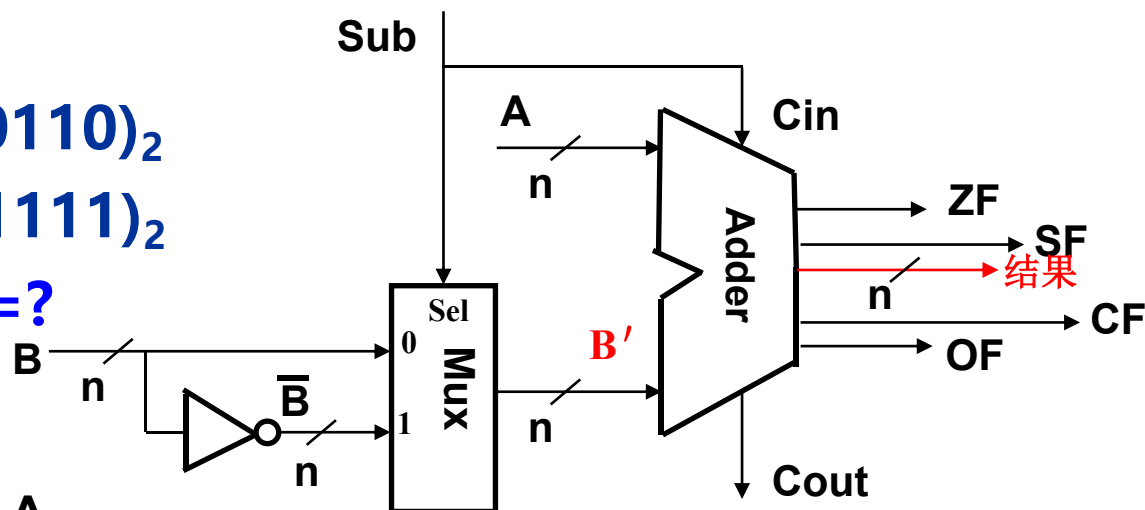
$CF=1$ $OF=0$

$$\begin{array}{r} 166_{10} = 1010\ 0110_2 \quad A \\ + [-255_{10}]_{补} = 0000\ 0001_2 \quad B' + Cin \\ \hline \end{array}$$

$$\begin{array}{r} \quad \quad \quad \boxed{0} 1010\ 0111 \\ \hline \end{array}$$

进位 $C_{out}=0$ 结果 $=167=-89+256$

$CF=1$ $OF=0$



整数加减运算器

两个无符号数 x 和 y 相加/减

□ 如果 $x+y \in [Umax_n+1, 2*Umax_n]$, 则溢出, 即 $CF=1$, 结果 $=x+y-2^n$

□ 如果 $x-y \in [-Umax_n, -1]$, 也溢出, 即 $CF=1$, 结果 $=x-y+2^n$

□ 也就是说, 对于无符号数, 如果进/借位标志 $CF=1$, 那么结果溢出, 而 OF 可为0或1

整数加减运算器中无/带符号数的加/减溢出举例

若 $n=8$,带符号数

$$(-90)_{10} = A6H = (1010\ 0110)_2$$

$$(63)_{10} = 3FH = (0011\ 1111)_2$$

$$-90 + 63 = ? \quad -90 - 63 = ?$$

$$\begin{array}{r} -90_{10} = 1010\ 0110_2 \quad A \\ + \quad 63_{10} = 0011\ 1111_2 \quad B' + Cin \\ \hline \end{array}$$

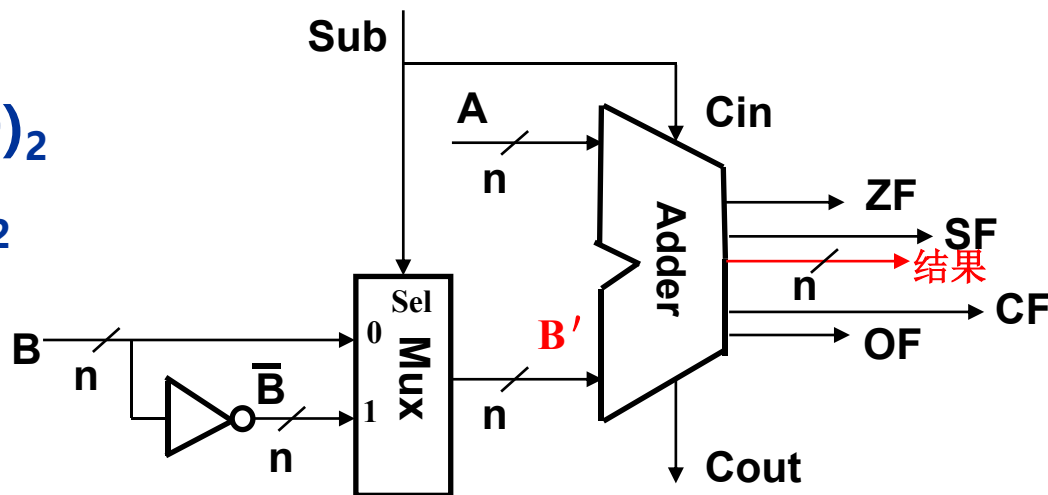
$$\quad \boxed{0}1110\ 0101$$

进位 $C_{out}=0$ 结果=-27 $CF=0$ $OF=0$

$$\begin{array}{r} -90_{10} = 1010\ 0110_2 \quad A \\ + \quad [-63_{10}]_{\text{补}} = 1100\ 0001_2 \quad B' + Cin \\ \hline \end{array}$$

$$\quad \boxed{1}0110\ 0111$$

进位 $C_{out}=1$ 结果=103=-153+256
 $CF=0$ $OF=1$



整数加减运算器

两个带符号数x和y相加/减

□ 如果 $x \pm y \in [Tmin_n, Tmax_n]$, 无溢出

此时, $OF=0$, 结果= $x \pm y$

□ 如果 $x \pm y \in [2 * Tmin_n, Tmin_n - 1]$, 负溢出

此时, $OF=1$, 结果= $x \pm y + 2^n$

□ 进/借位标志CF可以为0或1

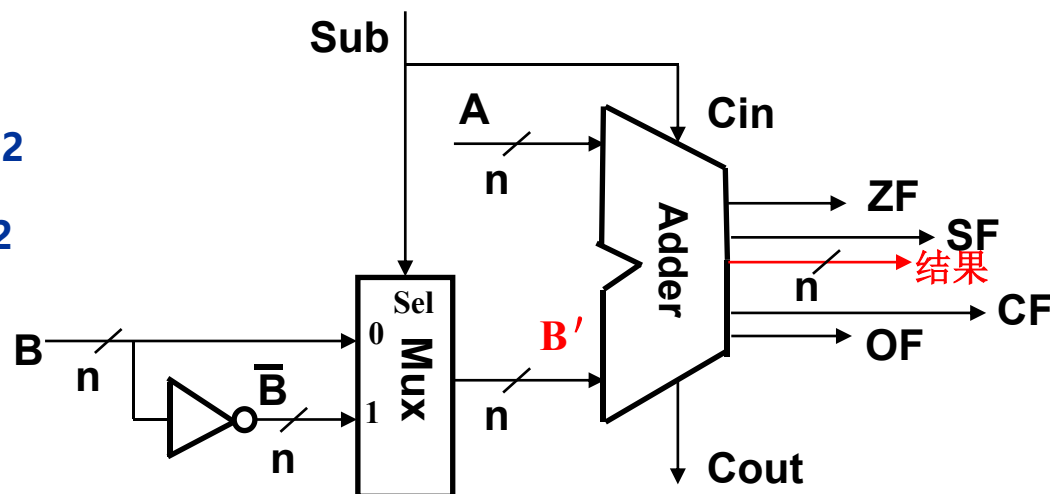
整数加减运算器中无/带符号数的加/减溢出举例

若 $n=8$,带符号数

$$(90)_{10} = 5AH = (0101\ 1010)_2$$

$$(63)_{10} = 3FH = (0011\ 1111)_2$$

$$90 + 63 = ? \quad 90 - 63 = ?$$



整数加减运算器

$$\begin{array}{r} 90_{10} = 0101\ 1010_2 \quad A \\ + \quad 63_{10} = 0011\ 1111_2 \quad B' + Cin \\ \hline \end{array}$$

进位 $C_{out}=0$ 结果 $=-103=153-256$
 $CF=0\ OF=1$

$$\begin{array}{r} 90_{10} = 0101\ 1010_2 \quad A \\ + \quad [-63_{10}]_{\text{补}} = 1100\ 0001_2 \quad B' + Cin \\ \hline \end{array}$$

进位 $C_{out}=1$ 结果 $=27\ CF=0\ OF=0$

两个有符号数 x 和 y 相加/减

□ 如果 $x \pm y \in [Tmax_n + 1, 2 * Tmax_n]$, 正溢出, 此时 $OF=1$, 结果 $=x \pm y - 2^n$

□ 如果 $x \pm y \in [Tmin_n, Tmax_n]$, 无溢出, 此时 $OF=0$, 结果 $=x \pm y$

□ 进/借位标记可以为0或1

- 在前面的例子中，我们也可以发现：不管是无符号数还是带符号数，如果 X 和 Y 的机器数相同，在同样的电路中计算，得到的和（或差）的机器数及标志位也完全相同
- 但是，相同的机器数会作为无符号数和带符号数解释来获得想要的结果，同时可以根据无符号数和带符号数的溢出判断条件，获得正确的溢出判断结果

例如，在前面例子中的：

无符号数

166+63=?

$$\begin{array}{r}
 166_{10} = 1010\ 0110_2 \\
 +\ 63_{10} = 0011\ 1111_2 \\
 \hline
 \boxed{0}1110\ 0101
 \end{array}$$

进位C_{out}=0 Sum=229 CF=0 OF=0
无溢出

166-63=?

$$\begin{array}{r}
 166_{10} = 1010\ 0110_2 \\
 +\ [-63_{10}]_{补} = 1100\ 0001_2 \\
 \hline
 \boxed{1}0110\ 0111
 \end{array}$$

进位C_{out}=1 Sum=103 CF=0 OF=1
无溢出

带符号数

-90+63=?

$$\begin{array}{r}
 -90_{10} = 1010\ 0110_2 \\
 +\ 63_{10} = 0011\ 1111_2 \\
 \hline
 \boxed{0}1110\ 0101
 \end{array}$$

进位C_{out}=0 Sum=-27 CF=0 OF=0
无溢出

-90-63=?

$$\begin{array}{r}
 -90_{10} = 1010\ 0110_2 \\
 +\ [-63_{10}]_{补} = 1100\ 0001_2 \\
 \hline
 \boxed{1}0110\ 0111
 \end{array}$$

进位C_{out}=1 Sum=103 CF=0 OF=1
溢出

无符号数A6H=(166)₁₀, 3FH=(63)₁₀
带符号数A6H=(-90)₁₀, 3FH=(63)₁₀

- 可以看到，在电路中执行运算时，所有的数都只是一个0/1序列，在底层机器级并不区分操作数是什么类型（例如，在Intel X86指令集中，加/减指令并不区分是无符号数的加/减指令还是带符号数的加/减指令），而只是由编译器根据高级语言中的类型定义对机器数进行不同的解释。

- 思考：这会带来什么问题？

由于在底层机器级对无符号数和带符号数的运算不加区分，然而编译器会根据其内定规则将带符号数隐式地转换为无符号数（类型提升），因此会发生一些意想不到的错误

例子：无符号数和带符号数混合运算带来的问题

例2.28 以下C语言程序计算数组a的元素和。当参数len为0时,返回结果应该是0。但在机器上执行时,却发生了存储器访问异常,原因是什么?

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

例子：无符号数和带符号数混合运算带来的问题

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

无符号整数转移指令

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

取len

取i

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

subl \$1, %edx指令的执行结果（计算len-1）

做减法，Sub为1

→ Sub=1

%edx

X

n

已知%edx中为 len=0000 0000H

0000 0001H

Y

n

n

Y'

多路选择器

加法器

C_{in}

C_{out}

ZF

SF

n

结果

OF

CF=C_{out}⊕Sub

执行“subl \$1, %edx”：X=0000 0000H，Y=0000 0001H，Sub=1，

Y'= FFFF FFFE H，因此运算结果是FFFF FFFFH，即%edx变为FFFF FFFFH。

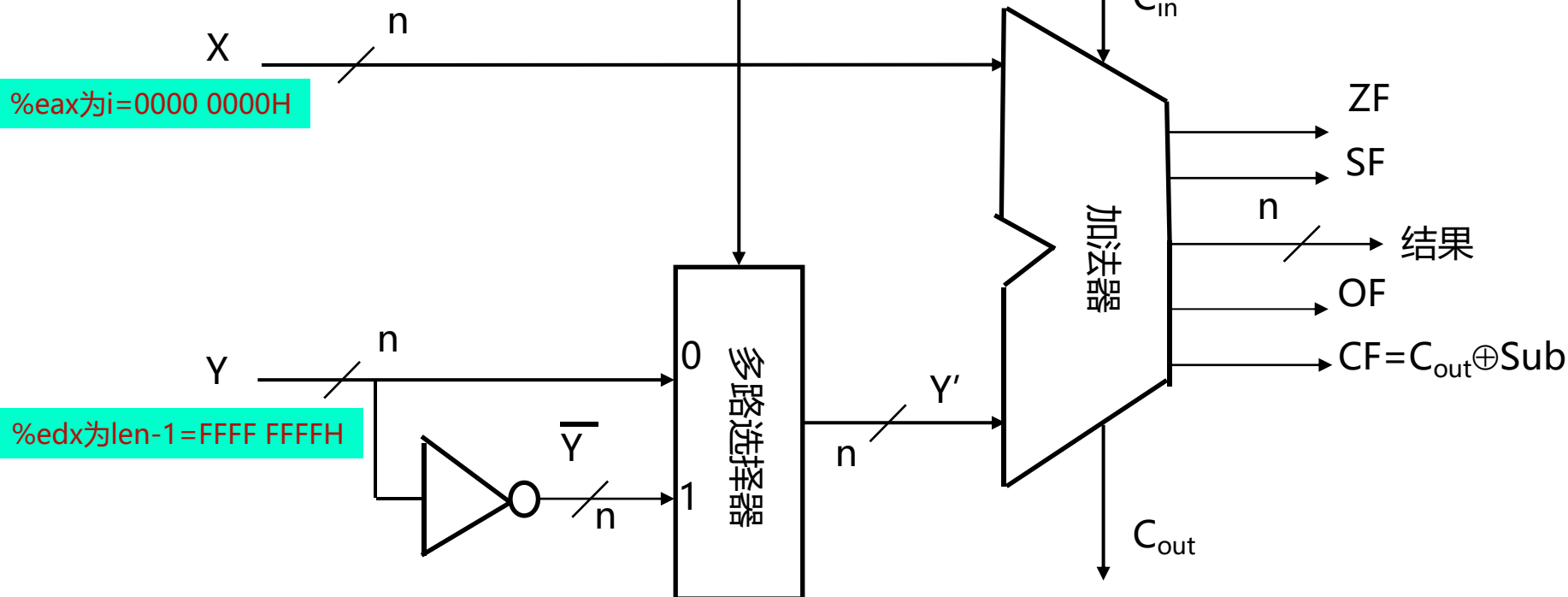
ZF=0，OF=0，SF=1，C_{out}=0，CF=C_{out}⊕Sub=0⊕1=1

cmpl %edx,%eax指令的执行结果 (执行 $i \leq \text{len}-1$)

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

已知%eax中为 $i=0000\ 0000\text{H}$

%edx中为 $\text{len}-1=\text{FFFF}\ \text{FFFFH}$



“`cmpl %edx,%eax`” 执行时, 隐含一个减法运算: $X=00000000\text{H}$, $Y=\text{FFFFFFFFH}$,
 $\text{Sub}=1$, $Y' = 00000000\text{H}$, 因此运算后结果是 00000001H ,

$\text{ZF}=0$, $\text{OF}=0$, $\text{SF}=0$, $C_{out}=0$, $CF = \text{Sub} \oplus C_{out} = 1 \oplus 0 = 1$

jbe .L3指令的执行结果 (执行 $i \leq len-1$)

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 $A \geq B$
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 $A \leq B$
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 $A \geq B$
JL/JNGE label	SF \neq OF AND ZF=0	有符号数 $A < B$
JLE/JNG label	SF \neq OF OR ZF=1	有符号数 $A \leq B$

“`cmpl %edx,%eax`” 执行后标志位是 ZF=0, OF=0, SF=0, CF=1, 说明满足条件（小于或等于），应转移到.L3继续执行！

因为任何32位整数都小于等于32个1，所以对于每个带符号整数 i 都满足条件。这意味着进入死循环，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。

如果将len改为int类型，则对“ $i \leq \text{len}-1$ ”的判断按带符号整数进行比较，对应的是带符号整数转移指令。尽管此时整数加减运算器运算结果及生成的所有标志位信息和len为unsigned时完全一样，也会正常跳出循环，而不出错

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

因为len是int,所以是带符号整数转移指令

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

“`cmpl %edx,%eax`” 执行后标志位仍然是 $\text{ZF}=0, \text{OF}=0, \text{SF}=0, \text{CF}=1$

指令	转移条件	说明
JLE/JNG label	$\text{SF} \neq \text{OF}$ OR $\text{ZF}=1$	有符号数 $A \leq B$

执行后标志位 $\text{SF}=\text{OF}=0$ 并且 $\text{ZF} \neq 1$ ，因此不满足跳转条件，从而退出循环

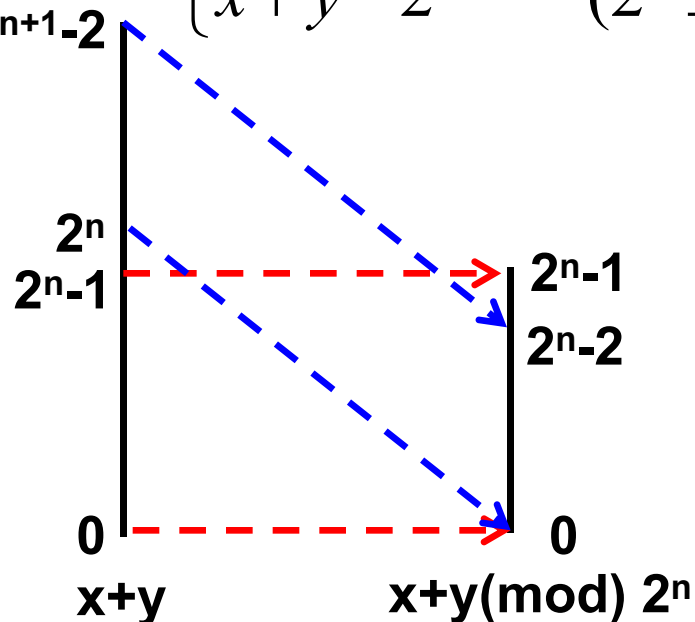
整数加减运算器的运算规则

◆ 无符号整数在加减运算器上的运算规则

- **加法**：对于无符号数 $x, y \in [0, 2^n - 1]$ 有 $x + y \in [0, 2^{n+1} - 2]$ 。因为其编码够成一个环（即在上述电路中执行**加法运算**结果是 $x + y \pmod{2^n} \in [0, 2^n - 1]$ ），所以当 $2^n \leq x + y < 2^{n+1}$ 时，电路计算出的结果相当于减掉了 2^n （即运算结果只保留低 n 位， $C_{out}=1$ ）

➤ 即，无符号整数在**加减运算器**上**加法运算**公式为：

$$result = \begin{cases} x + y & (x + y < 2^n) \quad \text{正常} \quad (C_{out}=0) \\ x + y - 2^n & (2^n \leq x + y < 2^{n+1}) \quad \text{溢出} \quad (C_{out}=1) \end{cases} \quad \text{公式(2.1)}$$



□ 蓝色区域说明结果出现“溢出”，结果需要减 2^n

□ 溢出状态可以用
 $CF = C_{out} \oplus Sub$ 判断

□ 红色区域说明结果正常

无符号整数加法溢出判断

- C程序运行时，**不会将溢出作为错误信号**，但有时需要判定是否发生溢出，那么**如何判断**无符号数加运算结果**是否溢出**呢？
- 条件码判断 **CF=1溢出；CF=0正常**
- 或者，通过数学等价式判断

例如：对于无符号数X、Y，如果**结果Sum < X**（或**结果Sum < Y**），那么X+Y溢出

思考：为什么 “结果Sum < X（或结果Sum < Y）” 等价于 “X+Y溢出” ？

思考：如何实现如下函数，用于判断无符号数加法溢出？无溢出返回true

```
int uadd_ok(unsigned x, unsigned y)
```

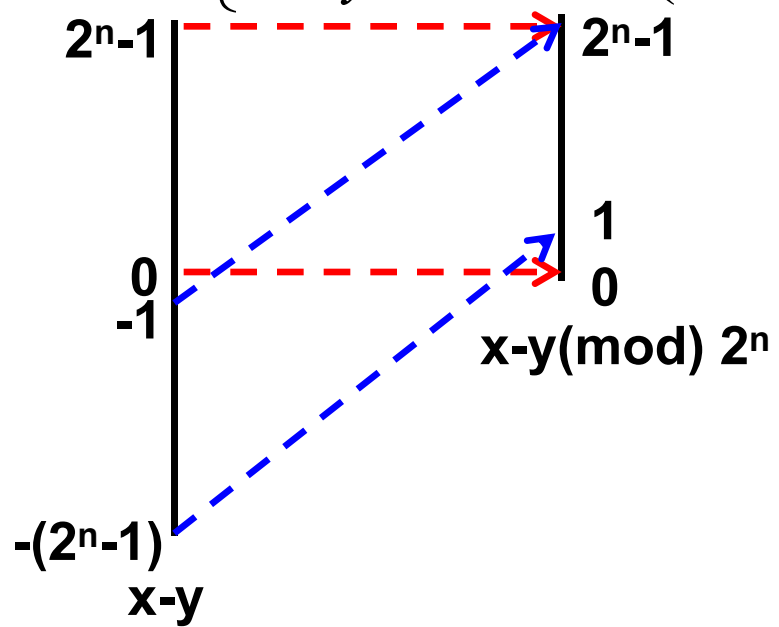
```
int uadd_ok(unsigned x, unsigned y)
{
    unsigned sum = x+y;
    return sum >= x;
}
```

◆ 无符号整数在加减运算器上的运算规则

- **减法**：对于无符号数 $x, y \in [0, 2^n-1]$ 有 $x-y \in [-(2^n-1), 2^n-1]$ 。因为其编码够成一个环（即在上述电路中执行减法运算结果是 $x-y(\text{mod } 2^n) \in [0, 2^n-1]$ ），所以当 $x-y < 0$ 时，电路计算出的结果相当于加上了 2^n 。
- 具体来说，因为运算电路实现 $x-y$ 时，采用对 y 的机器数取反加1实现，所以

$x-y = x + (2^n - y) = x - y + 2^n$ 。因此，当 $x-y > 0$ 时， $C_{out}=1$ ；当 $x-y < 0$ 时， $C_{out}=0$
➤即，无符号整数在加减运算器上**减法运算**公式为：

$$result = \begin{cases} x - y & (x - y > 0) \quad \text{正常} \quad (C_{out}=1) \\ x - y + 2^n & (x - y < 0) \quad \text{溢出} \quad (C_{out}=0) \end{cases} \quad \text{公式(2.2)}$$



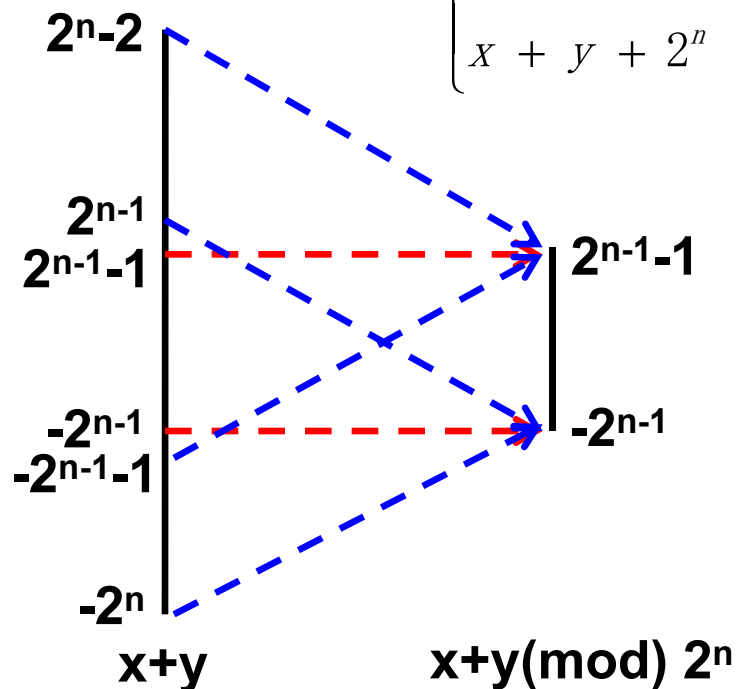
- 蓝色区域说明结果出现“溢出”，结果需要加 2^n
- 溢出状态可以用 $CF = C_{out} \oplus Sub$ 判断
- 红色区域说明结果正常

◆ 带符号整数在加减运算器上的运算规则：

- **加法：** 对于带符号整数 $x, y \in [-2^{n-1}, 2^{n-1}-1]$ ，则有 $x+y \in [-2^n, 2^n-2]$ 。因为其编码够成一个环（即在上述电路中执行**加法运算**的结果是 $x+y \pmod{2^n} \in [-2^{n-1}, 2^{n-1}-1]$ ），所以当 $2^{n-1} \leq x+y$ 时，电路计算出的结果相当于减去了 2^n （即运算结果只保留低 n 位）；当 $x+y < -2^{n-1}$ 时，相当于加上 2^n
➤即，带符号整数在**加减运算器**上**加法运算**公式为：

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

公式(2.3)



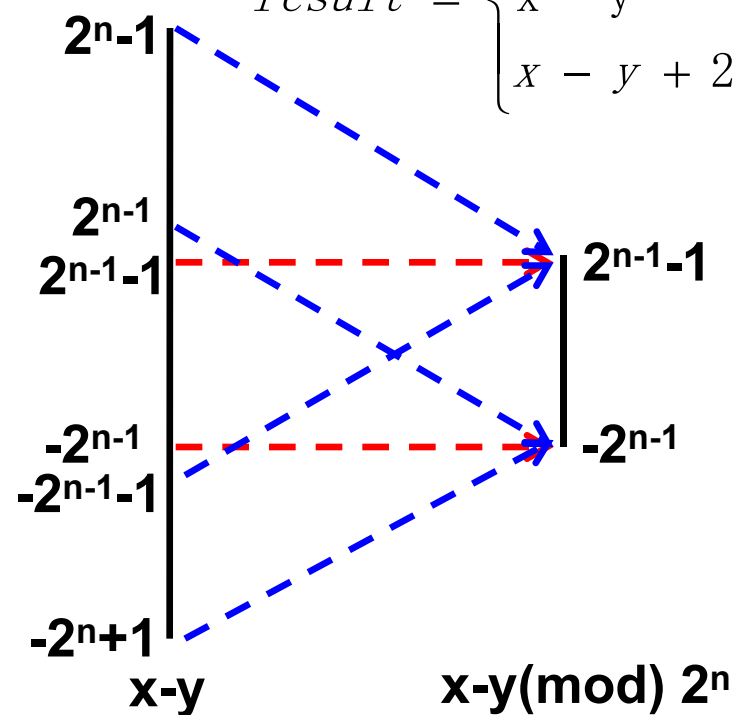
- 蓝色区域说明结果出现“溢出”，结果需要加/减 2^n
- 溢出状态可以用OF判断
- 红色区域说明结果正常

◆ 带符号整数在加减运算器上的运算规则：

- **减法**：对于带符号数 $x, y \in [-2^{n-1}, 2^{n-1}-1]$ ，有 $x-y \in [-2^n+1, 2^n-1]$ 。因为其编码够成一个环（即在上述电路中执行**加法运算**的结果是 $x-y(\bmod 2^n) \in [-2^{n-1}, 2^{n-1}-1]$ ），所以当 $2^{n-1} \leq x-y$ 时，电路计算出的结果相当于减去了 2^n （即运算结果只保留低 n 位），当 $x-y < -2^{n-1}$ 时，相当于加上了 2^n 。
➤即，带符号整数在**加减运算器**上**减法运算**公式为：

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

公式(2.4)



- 蓝色区域说明结果出现“溢出”，结果需要加/减 2^n
- 溢出状态可以用OF判断
- 红色区域说明结果正常

带符号整数加法溢出判断程序

- 如何用程序判断一个带符号整数相加没有发生溢出

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

公式(2.3)

带符号整数相加时溢出问题分析:

- ◆如果两个n位数x和y的符号**相反**，则一定不会溢出；
- ◆如果两个n位数x和y的符号**相同**，则可能会溢出。两个加数都是正数时发生的溢出称为正溢出，两个加数都是负数时发生的溢出称为负溢出。

/* 判断参数的加运算是否不溢出（即不溢出时返回true） */

int tadd_ok(int x, int y)

```
{  
    int sum = x+y;  
    int neg_over = x < 0 && y < 0 && sum >= 0;  
    int pos_over = x >= 0 && y >= 0 && sum < 0;  
    return !neg_over && !pos_over;  
}
```

带符号整数减法溢出判断程序

●如何用程序判断带符号整数相减是否溢出？

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

公式(2.4)

●思考：用以下程序检查带符号整数相减是否溢出，可以吗？

/* 判断参数的减运算是否不溢出（即不溢出时返回true） */

```
int tsub_ok(int x, int y)
```

```
{
    return tadd_ok(x, -y);
}
```

```
int tadd_ok(int x, int y)
```

```
{
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

不行。Why?

当x取小于0的数，y=0x8000 0000时，该函数neg_over=1，判断为负溢出。但实际x-y不会溢出
相反，当x取非负数，y=0x8000 0000时，该函数返回不溢出。但实际x-y会正溢出

课后试试：带符号减的溢出判断函数如何实现呢？

提示：带符号整数相减时溢出问题分析

◆如果两个n位数x和y的符号相同，则一定不会溢出；

◆如果两个n位数x和y的符号相反，则可能会溢出。正数减负数发生的溢出称为正溢出，负数减正数发生的溢出称为负溢出。

- **第二章：数据的机器级表示与处理**

- **2.7 数据的基本运算**

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加减运算
- 无符号和带符号整数的乘运算
- 无符号和带符号整数的除运算
- 浮点数的加减运算
- 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

5. 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会是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却还是一个n位整数，也就是说，其结果只取2n位乘积中的低n位。

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

x*y 被转换为乘法指令，在乘法运算电路中得到的乘积是64位，但是，只取其低32位赋给z。

因此这里可能会存在数据溢出的问题

例如：在计算机内部，一定有 $x^2 \geq 0$ 吗？

如x是浮点数，则一定！

若x是带符号整数，则不一定！

例如：当 $n=4$ 时， $5^2 = -7 < 0$ ！

	0101	
×	0101	
<hr/>		
0001	1001	结果溢出

只取低4位，值为-111B=-7

溢出判断：

- 乘法指令本身不判溢出，在机器级乘法指令的操作数长度为 n 时，虽然获得了 $2n$ 位的乘积结果，但是只有低 n 位供软件使用（截断为 n 位），另外高 n 位通常作为溢出标志用于溢出判断。
 - ◆例如：在IA-32中，若指令有一个8位/ 16位/ 32位的操作数SRC，另一个源操作数在累加器AL（8位）/AX（16位）/EAX（32位）中，将SRC和累加器内的源操作数内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。
- 因为硬件不判溢出，所以如果编译器不生成用于溢出处理的代码（例如C语言编译器就不生成），程序也不采用防止溢出的措施，就会发生一些由于整数溢出而带来的奇怪问题。

整数乘法溢出漏洞示例

以下程序存在什么漏洞，引起该漏洞的原因是什么？

/* 复制数组到堆中，count为数组元素个数 */

```
int copy_array(int *array, int count) {
```

```
    int i;
```

```
    /* 在堆区申请一块内存 */
```

```
    int *myarray = (int *) malloc(count*sizeof(int));
```

```
    if (myarray == NULL)
```

```
        return -1;
```

```
    for (i = 0; i < count; i++)
```

```
        myarray[i] = array[i];
```

```
    return count;
```

```
}
```

2002年，Sun Microsystems 公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

攻击者可构造特殊参数来触发整数溢出，以一段预设信息覆盖一个已分配的堆缓冲区，造成远程服务器崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。
如count= $2^{30}+1$ 时，
count*sizeof(int)=4。



堆 (heap) 中大量数据被破坏！

如何判断整数乘法是否溢出

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

若x、y和z都改成
unsigned类型，如何判断？

判断方式为:乘积的高n位为全0，则不溢出，因为此时
 $0 \leq x*y \leq 2^n - 1$

➤ 如何判断返回的乘积z是正确值（即不溢出）？

当 $!x \parallel z/x == y$ 为真时，计算结果z是正确值

➤ 还有其它方法判断乘积z是正确的吗？

带符号整数：当 $-2^{n-1} \leq x*y \leq 2^{n-1} - 1$ 时，不溢出。

也就是说: 如果是 $0 \leq x*y \leq 2^{n-1} - 1$ ，即2n位机器码是
0...0 00...0到0...0 01...1时，不溢出

如果是 $-2^{n-1} \leq x*y < 0$ ，即2n位机器码是
1...1 10...0到1...1 11...1时，不溢出

即：2n位乘积的高n位为全0或全1，并等于低n位的最高位时，不溢出！

即：乘积的高n+1位为全0或全1不溢出

- 总的来说，我们可以通过反向计算进行判定：当 $!x \parallel z/x == y$ 为真时为无溢出；否则溢出。
- 也可以使用 $2n$ 位乘积的高 n 位来判断是否溢出：
 - 根据高位寄存器中的每一位是否等于低位寄存器中的第一位来进行带符号整数乘溢出判断。若都相同，则无溢出；否则溢出。
 - 根据高位寄存器中的每一位是否都等于0进行无符号整数乘溢出判断。若都为0，则无溢出；否则溢出。

变量与常数之间的乘运算

- 通常使用乘法电路，进行一次乘法运算需要多个（通常需要10个左右）时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期，因此，**整数乘法运算**比移位和加法等运算所用时间长。
- 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、加减运算的组合**来代替乘法运算。（注意：不管是**无符号整数**还是**带符号整数**的乘法，即使在**乘积溢出**时，利用**移位**和**加减运算**组合的方式得到的结果都是和采用**直接相乘的结果是一样的**）

例如，对于表达式 $x * 20$ ，编译器可以利用 $20 = 16 + 4 = 2^4 + 2^2$ ，将 $x * 20$ 转换为 $(x \ll 4) + (x \ll 2)$ ，这样，**一次乘法**转换成了**两次移位**和**一次加法**。

- **第二章：数据的机器级表示与处理**

- **2.7 数据的基本运算**

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加减运算
- 无符号和带符号整数的乘运算
- 无符号和带符号整数的除运算
- 浮点数的加减运算
- 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

6. 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，一般都不会发生溢出，除非 $-2^{n-1}/-1$ 。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**那样发生溢出的问题。而 $-2^{n-1}/-1=2^{n-1}$ ，超出带符号整数最大表示范围，从而溢出
- 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，如： **$7/2=3$**
 - **负数商**取比自身大的最接近整数，如： **$-7/2=-3$**

注意：整数除0的结果可以用什么机器数表示？机器会怎么处理？

- ◆ 整数除0的结果无法用一个机器数表示（注：浮点数除0的结果可以用无穷大数表示）
- ◆ 从而，此时会发生“异常”，需要调出操作系统中的异常处理程序来处理。

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！
- 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(即移出的为全0)，直接右移得到结果
例如， $12/4=3$ ：0000 1100 >> 2 = 0000 0011
-12/4=-3：1111 0100 >> 2 = 1111 1101
 - 不能整除时(即右移移出的位中有非0)，怎么办？
需要进行舍入处理

- 舍入处理方式：采用**朝零舍入**，即**截断**方式

- 对于**无符号数**或**带符号正整数**：移出的低位直接丢弃，结果正确

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

- 对于**带符号负整数**，这样做就不对了：

例如：对于-14/4，若直接截断（**对于负整数来说，此操作结果就是朝负无穷大方向舍入**），则有

$1111\ 0010 \gg 2 = 1111\ 1100 = -4$ ，而其实 $-14/4 = -3$

- 因此，对于**带符号负整数**，如果需要除以 2^k ，需**先加偏移量** (2^k-1) ，**然后再右移 k 位**（表示除以 2^k ），**进行低位截断**

例如：对于-14/4，因为 $k=2$ ，所以需加偏移量 $2^2-1=3$

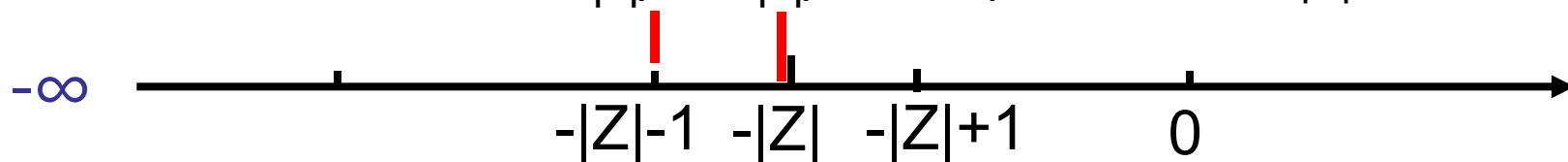
即： $1111\ 0010 + 0000\ 0011 = 1111\ 0101$

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

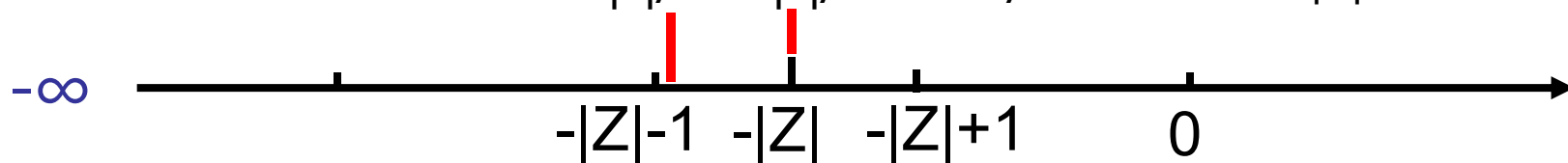
- 原理分析:

– 对于带符号负数 $-|x|$ ，先加偏移量 (2^k-1) ，然后再除以 2^k ，那么计算结果是： $(-|x| + 2^k - 1)/2^k = -|x|/2^k + 1 - 1/2^k$ ，在四舍五入时有3类情况：

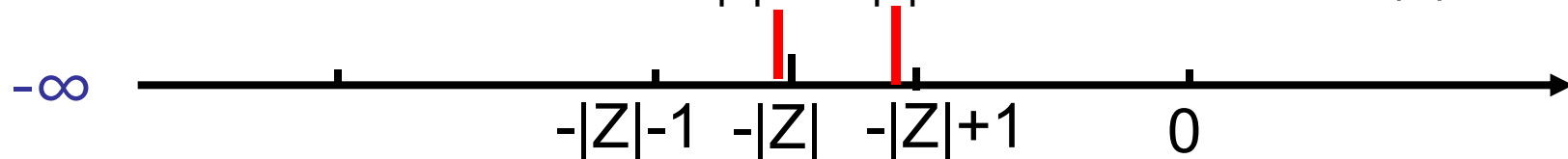
$$-|x|/2^k \quad -|x|/2^k + 1 - 1/2^k \quad \text{截断后为 } -|Z|-1$$



$$-|x|/2^k \quad -|x|/2^k + 1 - 1/2^k \quad \text{截断后为 } -|Z|$$



$$-|x|/2^k \quad -|x|/2^k + 1 - 1/2^k \quad \text{截断后为 } -|Z|$$



因此，对于带符号负整数，如果需要除以 2^k ，为了实现朝零舍入，可以先加偏移量 (2^k-1) ，然后再右移 k 位（表示除以 2^k ），进行低位截断

变量与常数之间的除运算例子

例如：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。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

分析：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 k 位得到商。因为 $32=2^5$ ，所以 $k=5$ 。

即： $(x \geq 0 ? x : (x+31)) \gg 5$

- 但上面要求不能用比较和条件语句，因此要找一个计算偏移量 b 的方式
- 由于 x 为正时 $b=0$ ， x 为负时 $b=31$ 。因此，可以试图从 x 的符号得到 b ：

不管是正数还是负数，由于 x 是带符号整数，所以右移是算术右移，因此 $x \gg 31$ 得到的是32位符号（全0或全1），取出最低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;
}
```

- **第二章：数据的机器级表示与处理**

- **2.7 数据的基本运算**

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加减运算
- 无符号和带符号整数的乘运算
- 无符号和带符号整数的除运算
- 浮点数的加减运算
- 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

7. 浮点数运算

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：

$$\begin{aligned} & 1.123 \times 10^5 + 2.560 \times 10^2 \\ = & 1.123 \times 10^5 + 0.002560 \times 10^5 \\ = & (1.123 + 0.00256) \times 10^5 \\ = & 1.12556 \times 10^5 \\ & \quad \text{舍入} \\ = & 1.126 \times 10^5 \end{aligned}$$

“对阶”：它就是使两数的阶码相等，目的是使得两个数的阶相同，使尾数可以直接相加减

- 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；
- 由于位数有限，最后还要舍入

计算机内部的二进制运算也一样

计算机上浮点数加减运算基本流程

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：

(a) **求阶差**： $\Delta e = Y_e - X_e$

阶是用移码表示的，如何计算浮点数 x 、 y 的阶差呢？

—两个规格化浮点数的**阶差**可用这两个数阶的**移码值相减**，并将**减后的二进制值视作补码来解析**即可

(b) **对阶**（目的是使得两个数的阶相同，使尾数可以直接相加减）：小阶向大阶看齐，将 X_m 小数点右移 Δe 位，尾数变为 $X_m * 2^{X_e - Y_e}$

◆ 注意：IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0

◆ 另外，为**保证精度**，尾数右移时，低位移出的位不丢掉，适当保留（有位置保存的话）到特定的“**附加位**”上，并参加尾数部分运算

浮点数加/减法对阶例子

- 实数 $x=1.5$, $y=-125.25$ 对阶过程及结果?

$[1.5]_{10} = [1.1]_2$, x 的二进制浮点数形式 $= 1.1 * 2^0$,

x 的阶0的移码为 $[0]_{\text{移码}} = 0111\ 1111$

x 的机器数为 $0\ 0111\ 1111\ 100\ 0000\ 0000\ 0000\ 0000\ 0000$

$[-125.25]_{10} = [-111\ 1101.01]_2$, $y = -1.1111\ 0101 * 2^6$,

y 的阶6的移码为 $[6]_{\text{移码}} = 1000\ 0101$

y 的机器数为 $1\ 1000\ 0101\ 111\ 1010\ 1000\ 0000\ 0000\ 0000$

$[0]_{\text{移码}} - [6]_{\text{移码}} = 0111\ 1111 - 1000\ 0101 = 1111\ 1010$,

补码解析为-6, 所以需要将 x 的尾数右移6位!

x 的尾数编码 $100\ 0000\ 0000\ 0000\ 0000\ 0000$

表示定点小数 $1.100\ 0000\ 0000\ 0000\ 0000\ 0000$

右移时需要将隐藏的“1”一起右移!

右移6位后为 $0.00\ 0001\ 100\ 0000\ 0000\ 0000\ 0000\ 0000$

移出的后六位如何处理? 尽量保留在附加位中

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

注意: 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**也参与运算。

- 浮点数x、y加/减运算时, 结果可能和IEEE 754规格化尾数形式 $\pm 1.bb...b$ 不相符, 怎么办?

此时, 则需要对结果进行规格化处理, 以保证结果仍然是IEEE

754规格化尾数形式: **$\pm 1.bb...b$**

- 具体来说, 如果**两个尾数相加减后的结果**有如下形式:

➤ $\pm 1b.bb...b$ \longrightarrow **需要右规**

➤ $\pm 0.00001bb...b$ \longrightarrow **需要左规**

问题: 为何IEEE 754 加减运算右规时最多只需一次?

因为即使是两个最大的尾数相加, 得到的和的尾数也不会达到4, 故尾数的整数部分最多有两位, 保留一个隐含的“1”后, 最多只有一位被右移到小数部分。

(d) 结果规格化:

◆ 对于尾数是 $\pm 1b.bb\dots b$ 形式, 需**右规**:

- 尾数右移一次(小数点左移), 阶码加1, 以符合 $\pm 1.bb\dots b$
- 阶码加1后**要判断阶码是否上溢** (即是否比最大可表示阶码大)

如果**阶码上溢**, 则算**结果溢出** (注意: 浮点数的溢出**不以尾数溢出来**判断, 因为尾数“溢出”可以通过**右规操作**得到纠正, 因此是否溢出要通过判断**阶是否溢出**而定)。此时, 有的机器产生阶码上溢异常, 而有的机器设为 $+\infty/-\infty$, 而不产生溢出异常

◆ 对于尾数是 $0.00001bb\dots b$ 形式, 需**左规**:

- 尾数左移一次(小数点右移), 阶码减1, 直到**符合要求或阶码为全0**
- 每次阶码减1后**要判断阶码是否为全0**。如果变成**全0**, 则说明结果是**非规格化数**, 此时应**停止左规**, 使**结果的尾数不变**, **阶码为全0**

(e) 尾数舍入处理:

在有附加位参加运算时，计算结果的尾数可能比规定位数长，因此需要舍入（有多种舍入方式，后面我们再专门讨论）

(f) 若运算结果尾数是0，则需要将阶码也置0。

为什么？

因为IEEE 754标准规定：只有尾数和阶都为0时才表示浮点数0（例如尾数为0，阶为255则表示单精度的无穷大）。而这里尾数为0，则说明结果应该为0，因此需要这样做。

浮点数加减运算舍入方法

Extra Bits(附加位)

◆ 增加附加位的**目的**：**保存对阶时向右移出的位**，并**允许它们参加计算和保存运算时的中间结果**作为舍入时依据，**提高计算精度**

◆ 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

• IEEE 754规定: 中间结果须在右边至少加2个附加位 (guard & round)，尾数加/减时附加位也要参加运算，运算结果的附加位最后舍入

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

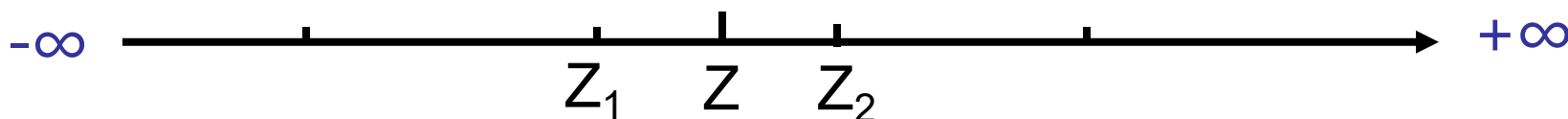
sticky (粘位) : 在舍入位右边的1位，只要舍入位右边有任何非0数字，粘位就被置1，否则为0.



如何根据保留的附加位进行舍入？

IEEE 754提供了四种舍入方式：

假定： Z_1 和 Z_2 分别是 Z 的最近的可表示的左、右两个数



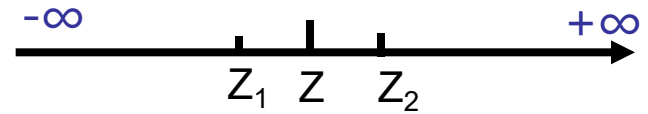
- (1) **就近舍入**：舍入为最近可表示的数 Z_1 或 Z_2 （中间值舍入为偶数）
- (2) 朝 $+\infty$ 方向舍入：舍入为 Z_2 ，即取比 Z 大的最近可表示数
- (3) 朝 $-\infty$ 方向舍入：舍入为 Z_1 ，即取比 Z 小的最近可表示数
- (4) 朝0方向舍入：直接丢弃所需位后面的所有位（为什么负整数需要加偏移量才能朝0方向舍入，而浮点数可以直接丢弃达到效果？）

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

向0趋近

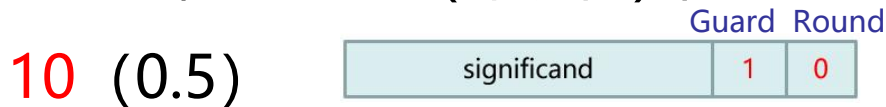
就近舍入：舍入为最近可表示的数

假定： Z_1 和 Z_2 分别是 Z 的最近的可表示的左、右两个数



- 当 Z 恰好是 Z_1 和 Z_2 的中间值时：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，Guard (保护位) 和Round (舍入位)附加位情形是：



例：1.110110 → 1.1110; (入) 1.111010 → 1.1110; (舍)

- 当 Z 不是 Z_1 和 Z_2 的非中间值时：保护位进行0舍1入；

注：此时，Guard (保护位) 和Round (舍入位)附加位情形是：



例：1.110111 → 1.1110; (入) 1.110101 → 1.1101; (舍)

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float情况下输出的结果会比原来的大？

原因：float可精确表示7个十进制有效数位，后面的数位是舍入后的结果。

2) 浮点运算的异常和精度问题

- “大数吃小数” 现象

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”的问题，这使得浮点数运算不能满足加减运算结合律。

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

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

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

因为对阶造成“小数”的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

浮点数加法运算举例

例如：用二进制浮点数形式计算 $0.5 + (-0.4375) = ?$

$$(0.5)_{10} = (0.1)_2,$$

二进制浮点数形式为 1.000×2^{-1}

$$(-0.4375)_{10} = -(0.25 + 0.125 + 0.0625) = (-0.0111)_2$$

二进制浮点数形式为 -1.110×2^{-2}

对 阶: $-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

加 减: $(1.000 + (-0.111)) \times 2^{-1} = 0.001 \times 2^{-1}$

左 规: $0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

判溢出舍入: 无

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

- **第二章：数据的机器级表示与处理**

- **2.7 数据的基本运算**

- 按位运算和逻辑运算
- 左移运算和右移运算
- 位扩展运算和位截断运算
- 无符号和带符号整数的加减运算
- 无符号和带符号整数的乘运算
- 无符号和带符号整数的除运算
- 浮点数的加减运算
- 浮点数的乘除运算

围绕C语言
中的运算，
解释这些运
算在底层机
器级的实现
方法

3) 浮点数乘除运算

- 如何进行浮点数乘除运算？

浮点数乘除运算类似于加减运算，其主要区别在于不需要“对阶”操作，但同样需要对结果规格化、舍入等处理

- 乘除运算规则：

设两个浮点数 $X = M_x \times 2^{E_x}$, $Y = M_y \times 2^{E_y}$, 则乘除运算结果如下

$$X \times Y = (M_x \times 2^{E_x}) \times (M_y \times 2^{E_y}) = (M_x \times M_y) \times 2^{E_x + E_y}$$

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

浮点运算的异常和精度问题

- 浮点数乘/除运算可能导致结果超出可表示范围，使得浮点数运算不能满足乘除运算结合律。

例如：对于 $x=y=1.0 \times 10^{30}$, $z=1.0 \times 10^{-30}$, 则有 $(x*y)*z \neq x*(y*z)$

$$(x*y)*z = (1.0 \times 10^{30} \times 1.0 \times 10^{30}) \times 1.0 \times 10^{-30}$$

$$= (1.0 \times 1.0) \times 10^{30+30} \times 1.0 \times 10^{-30}$$

$$= +\infty \times 1.0 \times 10^{-30}$$

$$= +\infty$$

大数乘积结果超出能表示范围，使得特殊数参加运算

$$x*(y*z) = 1.0 \times 10^{30} \times (1.0 \times 10^{30} \times 1.0 \times 10^{-30})$$

$$= 1.0 \times 10^{30} \times (1.0 \times 1.0 \times 10^{30-30})$$

$$= 1.0 \times 10^{30}$$

浮点运算精度和溢出举例

- 对于以下给定的关系表达式，判断是否永真。

```
int x ;  
float f ;  
double d ;
```

假定d和f都不是NaN和无穷大

int有效位数比float多，从int转换为float时，不会发生溢出，但可能有数据(作为尾数)被舍入

从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

从int或float转换为double时，因double有效位数更多(52位)，故能保留精确值，并且表示范围也大

从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入

- $x == (int)(float) x$ 否
- $f == (float)((int) f)$ 否
- $x == (int)(double) x$ 是
- $f == (float)(double) f$ 是
- $d == (float) d$ 否
- $f == -(-f)$ 是，因为浮点数取反就是简单的将符号位取反
- $2/3 == 2/3.0$ 否，因为左边是整数除法，商仍为整数，而右边是浮点数除法，有小数
- $d < 0.0 \Rightarrow (2*d) < 0.0$ 是，因为IEEE 754浮点数尾数采用原码表示，符号和数值部分分开运算，不管结果是否溢出，都不会影响符号位
- $d > f \Rightarrow -f > -d$ 是，同上
- $d*d \geq 0.0$ 是，同上
- $(d+f)-d == f$ 否，因为可能存在大数吃小数现象，比如 $d=1.79*10^{308}$ ， $f=1.0$

浮点运算举例：浮点数溢出导致 Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是**在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常**。因为，他们直接重用了原来的面向Ariana 4火箭的一部分代码。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数。然而，Ariana 5火箭比原来的Ariana 4火箭所能达到的速率高出了5倍，导致存放火箭水平速率的变量溢出。
- **在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误**，这种错误有时会带来重大损失，因此，编程时要非常小心。

浮点数运算举例：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，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。

原因：由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题：

- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用存有0.1的24位定点二进制小数 x 乘以计数值作为以秒为单位的时间
- 0.1的二进制表示是一个无限循环序列：0.000 1100 [1100]...B
- 很明显0.1的机器数（即 0.000 1100 1100 1100 1100 1100B）是0.1的真值的24位近似表示。因此，在程序用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时，会出现误差累积。

分析： x与0.1的误差是多少？

$$\begin{aligned} & 0.1 - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100B \\ &= 0.000\ 1100\ [1100]...B - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100B \\ &= 0.000\ \underline{0000\ 0000\ 0000\ 0000\ 0000\ 0000}\ 1100\ [1100]...B \\ &= 0.000\ 1100\ [1100]...B \times 2^{-20} = 0.1 \times 2^{-20} \approx 9.54 \times 10^{-8} \end{aligned}$$

20个0

而当时，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 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。

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？

$$0.1 = 0.000\ 1100\ [1100]...B = (+1.1\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 00B \times 2^{-4})$$

因此，采用IEEE 754单精度浮点数表示后，x的机器数为

0 011 1101 1 100 1100 1100 1100 1100 1100

由于float型仅有24位有效位数（即尾数从小数点前的1开始一共24位）表示尾数，而尾数后面的有效位全被截断，故x与0.1之间的误差为：

$$\begin{aligned} |x-0.1| &= 0.\underbrace{000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000}_{24\text{个}0} 1100\ [1100]...B \\ &= 0.000\ 1100\ [1100]...B \times 2^{-24} \\ &= 0.1 \times 2^{-24} \approx 5.96 \times 10^{-9}. \end{aligned}$$

因此，100小时后时钟偏差为 $5.96 \times 10^{-9} \times 36 \times 10^5 \approx 0.0215$ 秒。

因此，距离偏差是 $0.0215 \times 2000 \approx 43$ 米。比爱国者导弹系统精确约16倍。

• 若用32位二进制定点小数:

$$x = 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ B$$

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:

$$|x - 0.1|$$

$$= 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101 - 0.000\ 1100\ [1100]...B$$

$$\approx 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ [1111]... - 0.000\ 1100\ [1100]...B$$

$$= 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ [0011]...B$$

$$= 0.000\ \underline{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 00} \ 1100\ [1100]...B$$

$$= 0.000\ 1100\ [1100]...B \times 2^{-30}$$

$$= 0.1 \times 2^{-30} \approx 9.31 \times 10^{-11}$$

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

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

比前面的爱国者导弹系统精确约1024倍 (高于float表示64倍)

● 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6位)。
- 而且采用float表示在计算速度上比采用32位定点小数计算慢得多。

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。这相比直接对两个定点数相乘慢得多。

● 带来的启示：

- ✓ 程序员在编写程序时，应对底层机器级数据的表示和运算有深刻理解；
- ✓ 在计算机世界里，经常是“差之毫厘，失之千里”，需要精确再精确；
- ✓ 同时，也不能遇到小数就用浮点数表示。例如，在有些情况下，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（这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
 - 按位运算、逻辑运算、移位运算、位扩展运算和位截断运算
 - 整数算术运算、浮点数算术运算
- 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）
 - 带符号数和无符号数的加、减都在同一个“算盘（整数加减运算器）”中
 - 现实与计算机中的运算结果有差异（因为计算机是模运算系统）
- 整数的乘、除运算
 - 无符号整数： x 乘 2^k 等于 x 逻辑左移 k 位、 x 除 2^k 等于 x 逻辑右移 k 位
 - 带符号整数乘： x 乘 2^k 等于 x 算术左移 k 位
 - 带符号整数除： x 除以 2^k 等于 $x + 2^k - 1$ 然后算术右移 k 位（朝零舍入）
- 浮点数运算
 - 加减：对阶/尾数加减/结果规格化/舍入(就近舍入)（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（P80）：

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