



南京大學
NANJING UNIVERSITY



数制与编码

南京大学

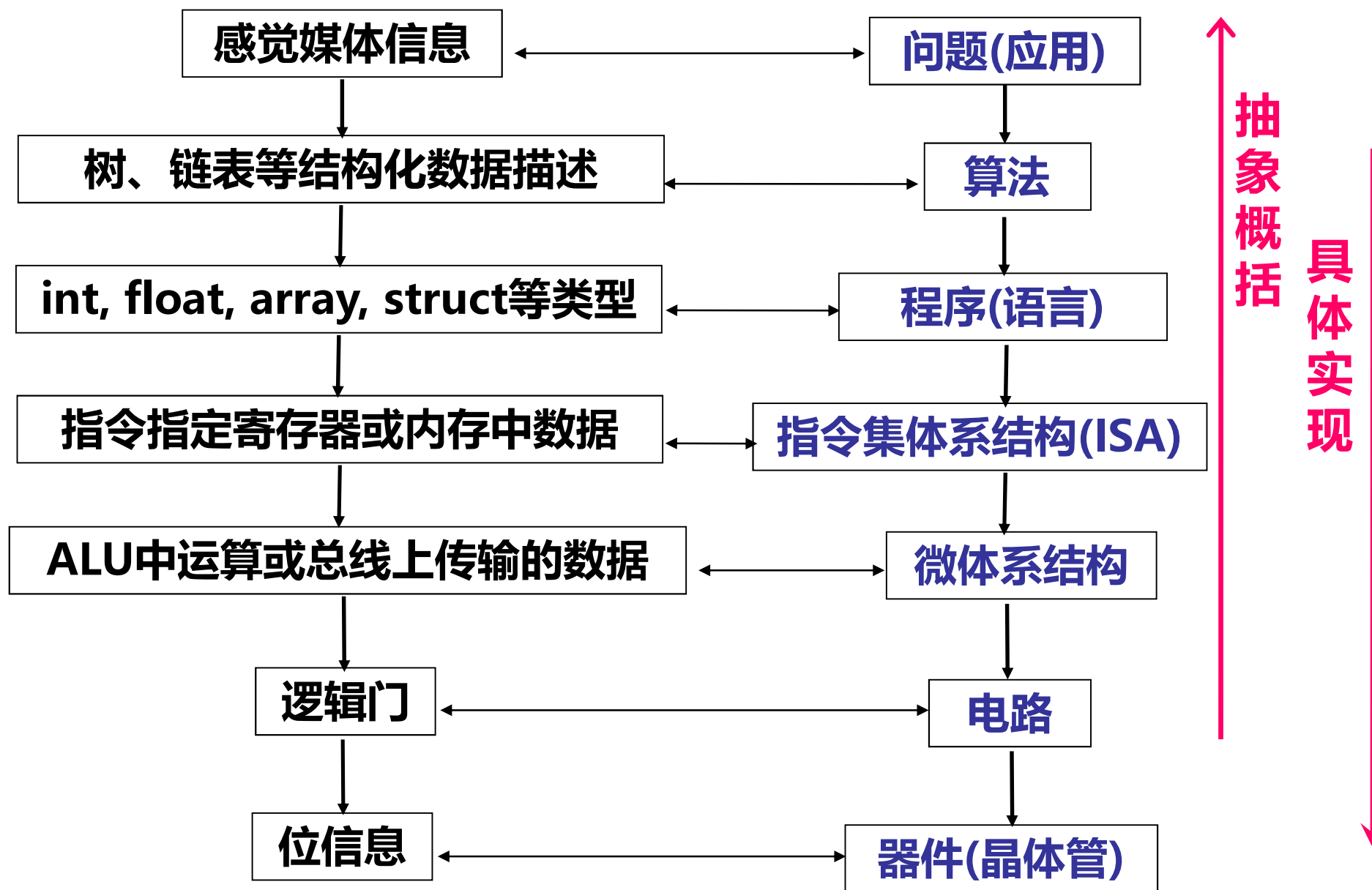
计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

“转换”的概念在数据表示中的反映



对连续信息采样，
以使信息离散化

对离散样本用0和1
进行编码

文字、图、表、声音、
视频等各种媒体信息

最终用户角度

输入设备

输出设备

二进制编码表示的各种数据

各类数据之间的
转换关系

数组、结构、字符串等结构化数据

高级语言程序员角度

指令系统能识别
的基本类型数据

低级语言程序员和
硬件系统设计者角度

数值型数据

非数值型数据

定点运算指令

二进制数

二进制编码的
十进制数

逻辑数据

编码字符
如：西文字符和汉字

整数（定点数）

实数（浮点数）

逻辑、位操作或字符处理指令

无符号整数

带符号整数

浮点运算指令

信息的二进制编码

- 机器级数据分两大类
 - 数值数据：无符号整数、带符号整数、浮点数（实数）
 - 非数值数据：逻辑数（包括位串）、西文字符和汉字
 - 计算机内部所有信息都用二进制（即：0和1）进行编码
 - 用二进制编码的原因
 - 制造二个稳定态的物理器件容易(电位高/低，脉冲有/无，正/负极)
 - 二进制编码、计数、运算规则简单
 - 正好与逻辑命题真/假对应，便于逻辑运算
 - 可方便地用逻辑电路实现算术运算
 - 真值和机器数（非常重要的概念！）
 - 机器数：用0和1编码的计算机内部的0/1序列
 - 真值：真正的值，即：现实中带正负号的数
- 例：unsigned short型变量x的真值是127，其机器数是多少？

127 = $2^7 - 1$ ，其机器数为0000 0000 0111 1111

数值数据的表示

- 数值数据表示的三要素

- 进位计数制

- 定、浮点表示

- 如何用二进制编码

即：要确定一个数值数据的值必须先确定这三个要素。

例如，20137564的值是多少？ 答案是：不知道！

- 进位计数制

- 十进制、二进制、十六进制、八进制数及其相互转换

- 定/浮点表示（解决小数点问题）

- 定点整数、定点小数

- 浮点数（可用一个定点小数和一个定点整数来表示）

- 定点数的编码（解决正负号问题）

- 原码、补码、反码、移码（反码很少用）

十进制 (Decimal) 计数制

- **十进制数**，每个数位可用**十个不同符号**0,1,2,...,9来表示，每个符号处在十进制数中不同位置时，所代表的数值不一样。

例如，2585.62代表的值是：

$$2585.62 = 2 \times 10^3 + 5 \times 10^2 + 8 \times 10^1 + 5 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2}$$

- 一般地，任意一个十进制数

$$D = d_n d_{n-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-m} \quad (m, n \text{ 为正整数})$$

- 其值可表示为如下形式：

$$V(D) = d_n \times 10^n + d_{n-1} \times 10^{n-1} + \dots + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2} + \dots + d_{-m} \times 10^{-m}$$

其中， d_i ($i = n, n-1, \dots, 1, 0, -1, -2, \dots, -m$) 可以是0,1,2,3,4,5,6,7,8,9这10个数字符号中的任何一个；

“10”称为基数 (base)，它代表每个数位上可以使用的不同数字符号个数。 **10^i 称为第*i*位上的权。**

运算时，**“逢十进一”**。

二进制 (Binary) 计数制

- **二进制数**，每个数位可用**两个不同符号**0和1来表示，每个符号处在不同位置时，所代表的数值不一样。

例如，100101.01代表的值是：

$$(100101.01)_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 37.25$$

- 一般地，任意一个二进制数

$$B = b_n b_{n-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m} \quad (m, n \text{ 为正整数})$$

- 其值可表示为如下形式：

$$V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \dots + b_{-m} \times 2^{-m}$$

其中， b_i ($i = n, n-1, \dots, 1, 0, -1, -2, \dots, -m$) 可以是0或1

“2” 称为**基数 (base)**，它代表每个数位上可以使用的不同数字符号个数。 **2^i** 称为第*i*位上的权。

运算时，**“逢二进一”**。

后缀 **“B”** 表示二进制数，如
01011010B

R进位计数制

- 在**R进制**数字系统中，应采用**R个基本符号**（0, 1, 2, . . . , R-1）表示各位上的数字，采用“**逢R进一**”的运算规则，对于每一个数位i，该位上的权为 R^i 。R被称为该数字系统的基。

二进制：R=2，基本符号为0和1

八进制：R=8，基本符号为0,1,2,3,4,5,6,7

十六进制：R=16，基本符号为0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

十进制：R=10，基本符号为0,1,2,3,4,5,6,7,8,9

$2^3=8$ ，对应3位二进制

$2^4=16$ ，对应4位二进制

二进制	八进制	十进制	十六进制	二进制	八进制	十进制	十六进制
0000	0	0	0	1000	10	8	8
0001	1	1	1	1001	11	9	9
0010	2	2	2	1010	12	10	A
0011	3	3	3	1011	13	11	B
0100	4	4	4	1100	14	12	C
0101	5	5	5	1101	15	13	D
0110	6	6	6	1110	16	14	E
0111	7	7	7	1111	17	15	F

八进制和十六进制

日常生活中用十进制表示数值，计算机中用二进制表示所有信息！
那为什么还要引入 八进制 / 十六进制呢？

八进制 / 十六进制是二进制的简便表示。便于阅读和书写！

它们之间对应简单，转换容易。

在机器内部用二进制表示，在屏幕或其他设备上表示时，转换为八进制/十六进制数，可缩短长度。

八进制：Octal（用后缀“O”表示）

十六进制：Hexadecimal（用后缀“H”，或前缀“0x”表示）

例：1010 1100 0100 0101 0001 0000 1000 1101B可写成

0xac45108d 0xAC45108D 或 ac45108dH AC45108DH

或 8进制：25421210215O

010 101 100 010 001 010 001 000 010 001 101

现代计算机系统多用十六进制表示机器数

十进制数与R进制数之间的转换

(1) R进制数 => 十进制数

按“权”展开

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

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

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

(2) 十进制数 => 二进制数，再将二进制转换为16或8进制

整数部分和小数部分分别转换

- ① 整数---- “除基取余，上右下左”
 - ② 小数---- “乘基取整，上左下右”
- } 理论上的做法

实际上，记住1、2、4、8、16、32、64、128、256、512、1024、2048、4096、8192、16384、32768、65536，.....就可简单进行整数部分的转换

记住0.5、0.25、0.125、0.0625、..... 就可简单进行小数部分的转换

十进制数与二进制数之间的转换

例1: $(835.6875)_{10} = (11\ 0100\ 0011.1011)_2$

整数---- “除基取余，上右下左”

小数---- “乘基取整，上左下右”

	余数	低位
2 835	1	↑
2 417	1	
2 208	0	
2 104	0	
2 52	0	
2 26	0	
2 13	1	

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

简便方法： $835 = 512 + 256 + 64 + 2 + 1$ ，故结果为 11 0100 0011

$0.6875 = 0.5 + 0.125 + 0.0625$ ，故结果为 0.1011

结果为 11 0100 0011.1011

这里有一个问题：小数点在计算机中如何表示？

十进制数与8进制数之间的转换

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

整数---- “除基取余，上右下左”

小数---- “乘基取整，上左下右”

8	835	余数	低位
8	104	3	
8	13	0	
8	1	5	
	0	1	高位

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

可能小数部分总得不到0，此时得到一个近似值

说明：现实中的精确值可能在机器内部无法用0和1精确表示！

定点数和浮点数

- 计算机中只有0和1，数值数据中的小数点怎么表示呢？

- 计算机中只能通过**约定小数点的位置**来表示

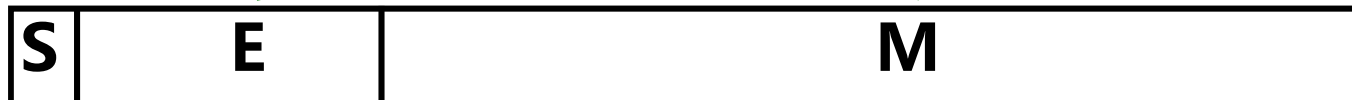
- 小数点位置约定在固定位置的数称为**定点数**

- 小数点位置约定为可浮动的数称为**浮点数**

结论：要解决数值数据的表示问题，只要解决定点数的编码问题！

- **定点小数**用来表示浮点数的尾数部分
- **定点整数**用来表示整数，分**带符号整数**和**无符号整数**
- 任何实数： $X = (-1)^S \times M \times R^E$

其中，S取值为0或1，用来决定数**X的符号**；**M是一个二进制定点小数**，称为数**X的尾数**（mantissa）；**E是一个二进制定点整数**，称为数**X的阶或指数**（exponent）；**R是基数**（radix、base），可以为2、4和16等。计算机中只要表示S、M和E三个信息，就能确定X的值，这称为**浮点数**





南京大學
NANJING UNIVERSITY



定点数的编码表示

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

数值数据的表示

- 数值数据表示的三要素
 - 进位计数制
 - 定、浮点表示
 - 如何用二进制编码
- 进位计数制
 - 十进制、二进制、十六进制、八进制数及其相互转换
- 定/浮点表示 (解决小数点问题)
 - 定点整数、定点小数
 - 浮点数 (可用一个定点小数和一个定点整数来表示)
- 定点数的编码 (解决正负号问题)
 - 原码、补码、移码、反码 (很少用)

原码 (Sign and Magnitude) 表示

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

“正” 号用0表示

“负” 号用1表示

数值部分不变!

◆ 容易理解, 但是:

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

从 50年代开始, 整数都采用补码来表示

但浮点数的尾数用原码定点小数表示

补码 - 模运算 (modular 运算)

重要概念：在一个模运算系统中，一个数与它除以“模”后的余数等价。

时钟是一种模12系统 现实世界中的模运算系统

假定钟表时针指向10点，要将它拨向6点， 则有两种拨法：

① 倒拨4格： $10 - 4 = 6$

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

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

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

则，称8是-4对模12的补码 (即：-4的模12补码等于8)。

同样有 $-3 \equiv 9 \pmod{12}$

$$-5 \equiv 7 \pmod{12} \text{ 等}$$

结论1：一个负数的补码等于模减该负数的绝对值。

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

补码 (modular运算)：+ 和- 的统一

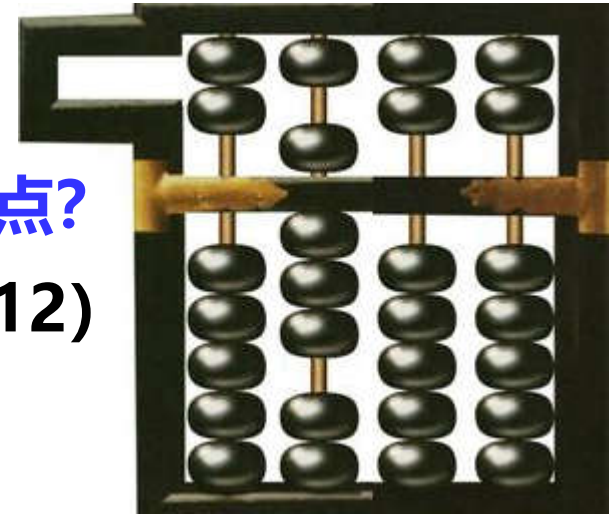
补码 (2's complement) 的表示

现实世界的模运算系统举例

例1: “钟表” 模运算系统

假定时针只能顺拨, 从10点倒拨4格后是几点?

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



例2: “4位十进制数” 模运算系统

假定算盘只有四档, 且只能做加法, 则在算盘上计算

9828-1928等于多少?

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

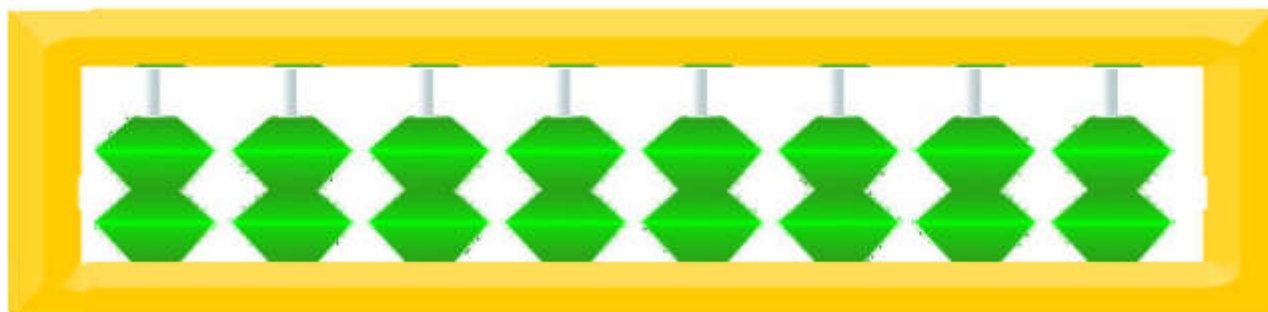
$$= 9828 + 8072$$

$$= \boxed{1}7900$$

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

取模即只留余数, 高位“1”被丢弃!
相当于只有低4位留在算盘上。

计算机中的运算器是模运算系统



8位二进制加法器模运算系统

$[-0100\ 0000]_{\text{补}} = ?$

计算 $0111\ 1111 - 0100\ 0000 = ?$

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

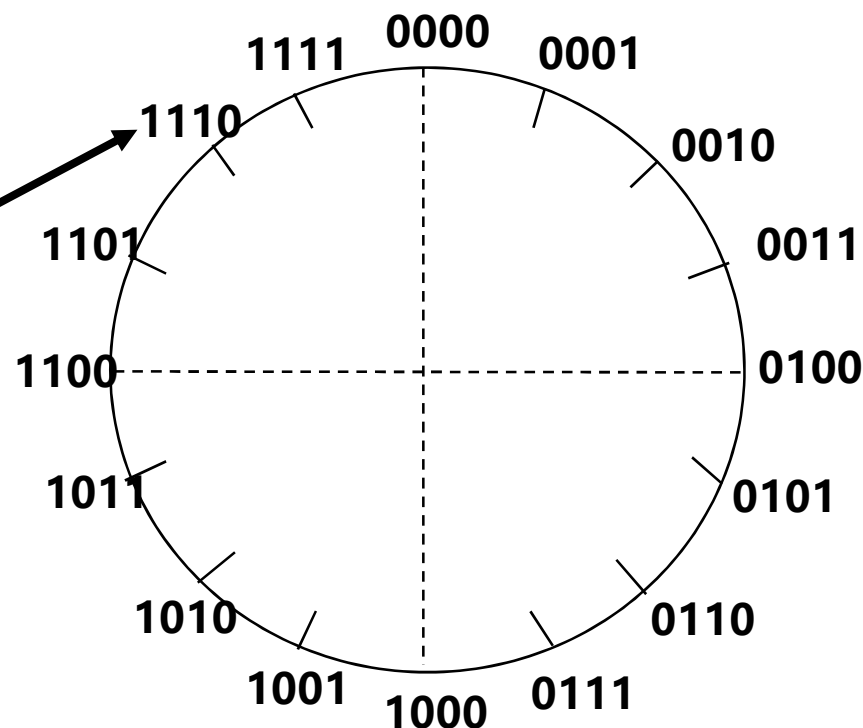
只留余数，“1”被丢弃

结论1： 一个负数的补码等于将对应正数补码
各位取反、末位加一

运算器适合用补码表示和运算

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

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



补码的定义 假定补码有 n 位，则：

$$[X]_{\text{补}} = 2^n + X \quad (-2^{n-1} \leq X < 2^{n-1}, \text{ mod } 2^n)$$

X 是真值， $[x]_{\text{补}}$ 是机器数

真值和机器数的含义是什么？

求特殊数的补码

假定机器数有n位

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

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

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

32位机器中，int、short、char型数据的机器数各占几位？

32位、16位、8位

变形补码（4's complement）的表示

补码定义: $[X]_{\text{补}} = 2^n + X \quad (-2^n \leq X < 2^n, \text{ mod } 2^n)$

- 正数: 符号位 (sign bit) 为0, 数值部分不变
- 负数: 符号位为1, 数值部分 “各位取反, 末位加1”

变形 (4's) 补码: 双符号, 用于存放可能溢出的中间结果。

Decimal	补码	变形补码	Decimal	Bitwise Inverse	补码	变形补码
0	0000	00000	-0	1111	0000	00000
1	0001	00001	-1	1110	1111	11111
2	0010	00010	-2	1101	1110	11110
3	0011	00011	-3	1100	1101	11101
4	0100	00100	-4	1011	1100	11100
5	0101	00101	-5	1010	1011	11011
6	0110	00110	-6	1001	1010	11010
7	0111	00111	-7	1000	1001	11001
8	1000	01000	-8	0111	1000	11000

值太大, 用4位补码无法表示, 故 “溢出”
但用变形补码可保留符号位和最高数值位

+0和-0表示唯一

求真值的补码


例: 设机器数有8位, 求123和-123的补码表示。

如何快速得到123的二进制表示?

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

$-123 = -01111011B$

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

$[-01111011]_{\text{补}} = 2^8 - 01111011 = 10000\ 0000 - 01111011$

 $= 1111\ 1111 - 0111\ 1011 + 1$
 $= 1000\ 0100 + 1$ ← 各位取反, 末位加1
 $= 1000\ 0101$, 即 85H。

简便方法: 从右向左遇到第一个1的前面各位取反

当机器数为16位时, 结果怎样? $\text{mod} = 2^{16}$

求补码的真值

令： $[A]_{\text{补}} = a_{n-1}a_{n-2}\cdots a_1a_0$

则： $A = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$

例如：补码 “11010110” 的真值为

$$-2^7 + 2^6 + 2^4 + 2^2 + 2 = -128 + 64 + 16 + 4 + 2 = -42$$

补码 “01010110” 的真值为

$$-0 \cdot 2^7 + 2^6 + 2^4 + 2^2 + 2 = 64 + 16 + 4 + 2 = 86$$

理论上的求法

简便求法：


符号为0，则为正数，数值部分相同

符号为1，则为负数，数值各位取反，末位加1

例如：补码 “01010110” 的真值为

$$+1010110 = 64 + 16 + 4 + 2 = 86$$

例如：补码 “11010110” 的真值为



$$-0101010 = -(32 + 8 + 2) = -42$$

移码表示 Excess (biased) notion

- 什么是移码表示？

将每一个数值加上一个偏置常数 (Excess / bias)

- 通常，当编码位数为 n 时，bias取 2^{n-1} 或 $2^{n-1}-1$ (如 IEEE 754)

Ex. $n=4$: $E_{\text{biased}} = E + 2^3$ (bias = $2^3 = 1000\text{B}$)

-8 (+8) ~ 0000B

-7 (+8) ~ 0001B

...

0 (+8) ~ 1000B

...

+7 (+8) ~ 1111B

0的移码表示唯一

当bias为 2^{n-1} 时，移码和补码仅第一位不同

移码用来表示浮点数的阶

- 为什么要用移码来表示指数（阶码）？

便于浮点数加减运算时的对阶操作 (比较大小)

例: $1.01 \times 2^{-1} + 1.11 \times 2^3$

$1.01 \times 2^{-1+4} + 1.11 \times 2^{3+4}$

补码: 111 < 011 ?
(-1) (3)

简化比较

移码: 011 < 111
(3) (7)



南京大學
NANJING UNIVERSITY



C语言中的整数

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn



2015.6

C语言支持的基本数据类型

C语言声明	操作数类型	存储长度（位）
(unsigned) char	整数 / 字节	8
(unsigned) short	整数 / 字	16
(unsigned) int	整数 / 双字	32
(unsigned) long int	整数 / 双字	32
(unsigned) long long int	-	2×32
char *	整数 / 双字	32
float	单精度浮点数	32
double	双精度浮点数	64
long double	扩展精度浮点数	80 / 96

整数类型分：无符号整数和带符号整数

无符号整数 (Unsigned integer)

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

带符号整数 (Signed integer)

- 计算机必须能处理正数(positive) 和负数(negative) , 用MSB表示数符 (0--正数 , 1--负数)
- 有三种定点编码方式
 - Signed and magnitude (原码)
定点小数 , 用来表示浮点数的尾数
 - Excess (biased) notion (移码)
定点整数 , 用于表示浮点数的阶 (指数)
 - Two's complement (补码)
50年代以来 , 所有计算机都用补码来表示带符号整数
- 为什么用补码表示带符号整数 ?
 - 补码运算系统是模运算系统 , 加、减运算统一
 - 数0的表示唯一 , 方便使用
 - 比原码多表示一个最小负数

C语言程序中的整数

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

常在一个数的后面加一个“u”或“U”表示无符号数

若同时有无符号和带符号整数，则C编译器将带符号整数强制转换为无符号数

假定以下关系表达式在32位用补码表示的机器上执行，结果是什么？

关系表达式	运算类型	结果	说明
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -2147483647-1			
2147483647U > -2147483647-1			
2147483647 > (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

C语言程序中的整数

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

带*的结果与常规预想的相反！

C语言程序中的整数

例如，考虑以下C代码：

```
1 int x = -1;
2 unsigned u = 2147483648;
3
4 printf ( "x = %u = %d\n" , x, x);
5 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？


$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

- ◆ 因为-1的补码整数表示为“11...1”，作为32位无符号数解释时，其值为 $2^{32}-1 = 4\ 294\ 967\ 296-1 = 4\ 294\ 967\ 295$ 。
- ◆ 2^{31} 的无符号数表示为“100...0”，被解释为32位带符号整数时，其值为最小负数： $-2^{32-1} = -2^{31} = -2\ 147\ 483\ 648$ 。

编译器处理常量时默认的类型


- C90



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

$$2147483648 = 2^{31}$$

- C99



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

C语言程序中的整数

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

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

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

在ISO C99标准下，2147483648为long long型，因此

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

由C语言中的
“Integer
Promotion”
规则决定的。

2) `i < 2147483647` 按int型数比较，结果为true。

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

```
#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");
```

```
}
```

请大家试试在C99上的运行结果。

C90上的运行结果是什么？

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

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

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

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



南京大學
NANJING UNIVERSITY



浮点数的编码表示

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

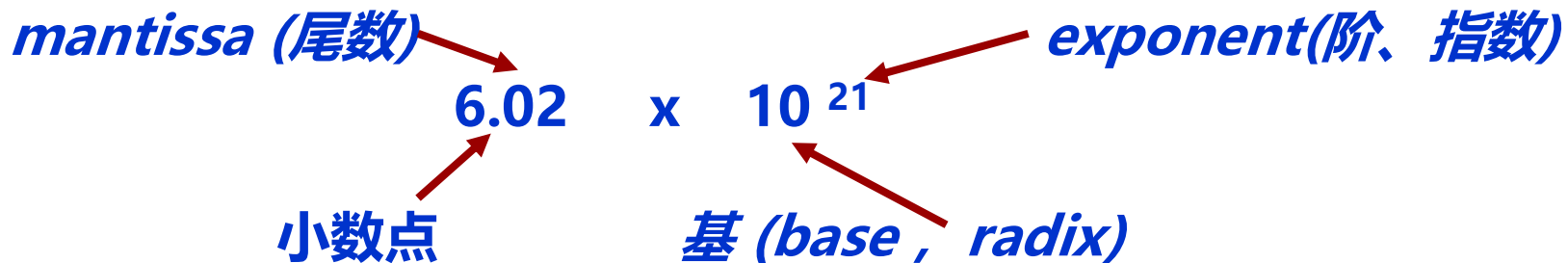
C语言支持的基本数据类型

C语言声明	操作数类型	存储长度（位）
(unsigned) char	整数 / 字节	8
(unsigned) short	整数 / 字	16
(unsigned) int	整数 / 双字	32
(unsigned) long int	整数 / 双字	32
(unsigned) long long int	-	2×32
char *	整数 / 双字	32
float	单精度浮点数	32
double	双精度浮点数	64
long double	扩展精度浮点数	80 / 96

实数类型分：单精度浮点、浮点双精度和扩展精度浮点

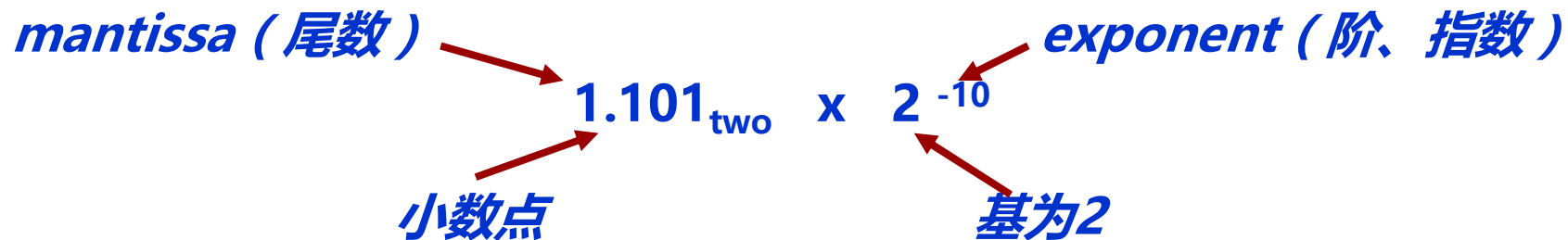
科学计数法(Scientific Notation)与浮点数

对于科学计数法（十进制数）：



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

对于二进制数实数



只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）

浮点数(Floating Point)的表示范围

例：画出下述32位浮点数格式的规格化数的表示范围。

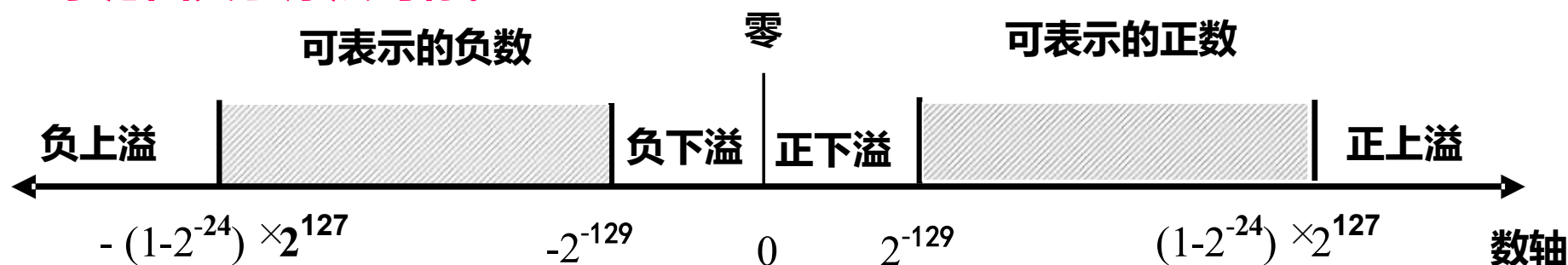


第0位数符S；第1~8位为8位移码表示阶码E（偏置常数为128）；第9~31位为24位二进制原码小数表示的尾数M。规格化尾数的小数点后第一位总是1，故规定第一位默认的“1”不明显表示出来。这样可用23个数位表示24位尾数。

因为原码对称，故其表示范围关于原点对称。

最大正数： $0.11...1 \times 2^{11...1} = (1-2^{-24}) \times 2^{127}$

最小正数： $0.10...0 \times 2^{00...0} = (1/2) \times 2^{-128}$



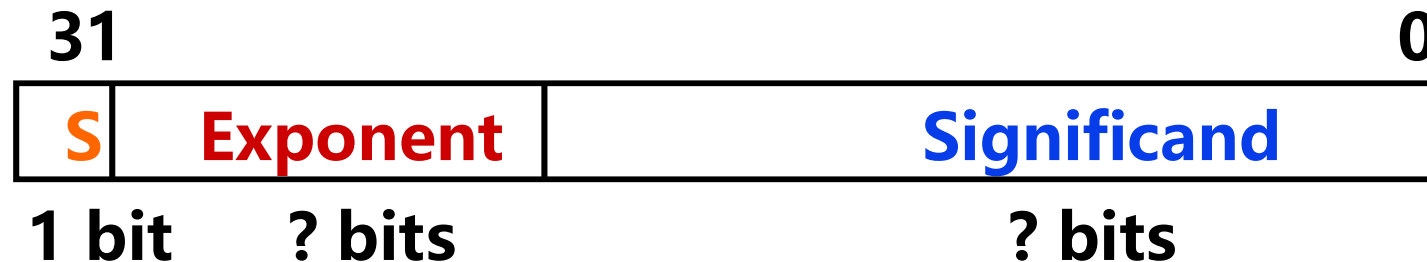
机器0：尾数为0 或 落在下溢区中的数

浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

浮点数的表示

- ° Normal format (规格化数形式) : 为了能表示更多有效数字, 通常规定规格化数的小数点前为1!
 $+/-1.\text{xxxxxxxxxxxx} \times R^{\text{Exponent}}$

- ° 32-bit 规格化数 :



S 是符号位 (Sign)

Exponent 用移码 (增码) 来表示

Significand 表示 xxxxxxxxxxxxxx (部分尾数)

(基可以是 2 / 4 / 8 / 16 , 约定信息 , 无需显式表示)

- ° 早期的计算机 , 各自定义自己的浮点数格式

问题 : 浮点数表示不统一会带来什么问题 ?

“Father” of the IEEE 754 standard

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

1970年代后期, IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

现在所有通用计算机都采用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

IEEE 754 标准

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

规定：小数点前总是“1”，故可隐含表示。

Single Precision (单精度)：

S	Exponent	Significand
1 bit	8 bits	23 bits

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

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

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

举例：机器数转换为真值

已知float型变量x的机器数为BEE00000H，求x的值是多少？

1 011 1101 110 0000 0000 0000 0000 0000

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

◦ 数符: 1 (负数)

◦ 阶 (指数):

• 阶码: 0111 1101B = 125

• 阶码的值: $125 - 127 = -2$

为避免混淆，用**阶码**表示**阶**的编码，用**阶**或**指数**表示阶码的值

◦ 尾数数值部分:

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

◦ 真值: $-1.75 \times 2^{-2} = -0.4375$

举例：真值转换为机器数

已知float型变量x的值为-12.75，求x的机器数是多少？

$$-12.75 = -1100.11\text{B}$$

$$= -1.10011\text{B} \times 2^3 \quad \text{阶（指数）为3}$$

因此，符号 $S=1$

$$\text{阶码 } E = 127 + 3 = 128 + 2 = 1000\ 0010$$

显式表示的部分尾数 Significant

$$= 100\ 1100\ 0000\ 0000\ 0000\ 0000$$

x 的机器数表示为：

1	1000 0010	100 1100 0000 0000 0000 0000
---	-----------	------------------------------

转换为十六进制表示为：C14C0000H

规格化数 (Normalized numbers)

前面的定义是针对规格化形式 (normalized form) 的数

那么，其他形式的机器数表示什么样的信息呢？

Exponent	Significand	
1-254	任意 小数点前隐含1	规格化形式
0 (全0)	0	?
0 (全0)	nonzero	?
255 (全1)	0	?
255 (全1)	nonzero	?

0的机器数表示

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

Single Precision (单精度) :

S	Exponent	Significand
1 bit	8 bits	23 bits

$+\infty/-\infty$ 的机器数表示

浮点数除0的结果是 $+/-\infty$, 而不是溢出异常. (整数除0为异常)

为什么要这样处理?

∞ : infinity

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

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

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

$+\infty$: 0 11111111 00000000000000000000000000000000

$-\infty$: 1 11111111 00000000000000000000000000000000

相关操作 :

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

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

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

“非数” 的表示

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

– 称为 **Not a Number (NaN)** - “非数”

How to represent NaN

Exponent = 255

Significand: nonzero

NaNs 可以帮助调试程序

相关操作：

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

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

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

etc.

$0/0 = \text{NaN}$

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

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

非规格化数(Denorms)的表示

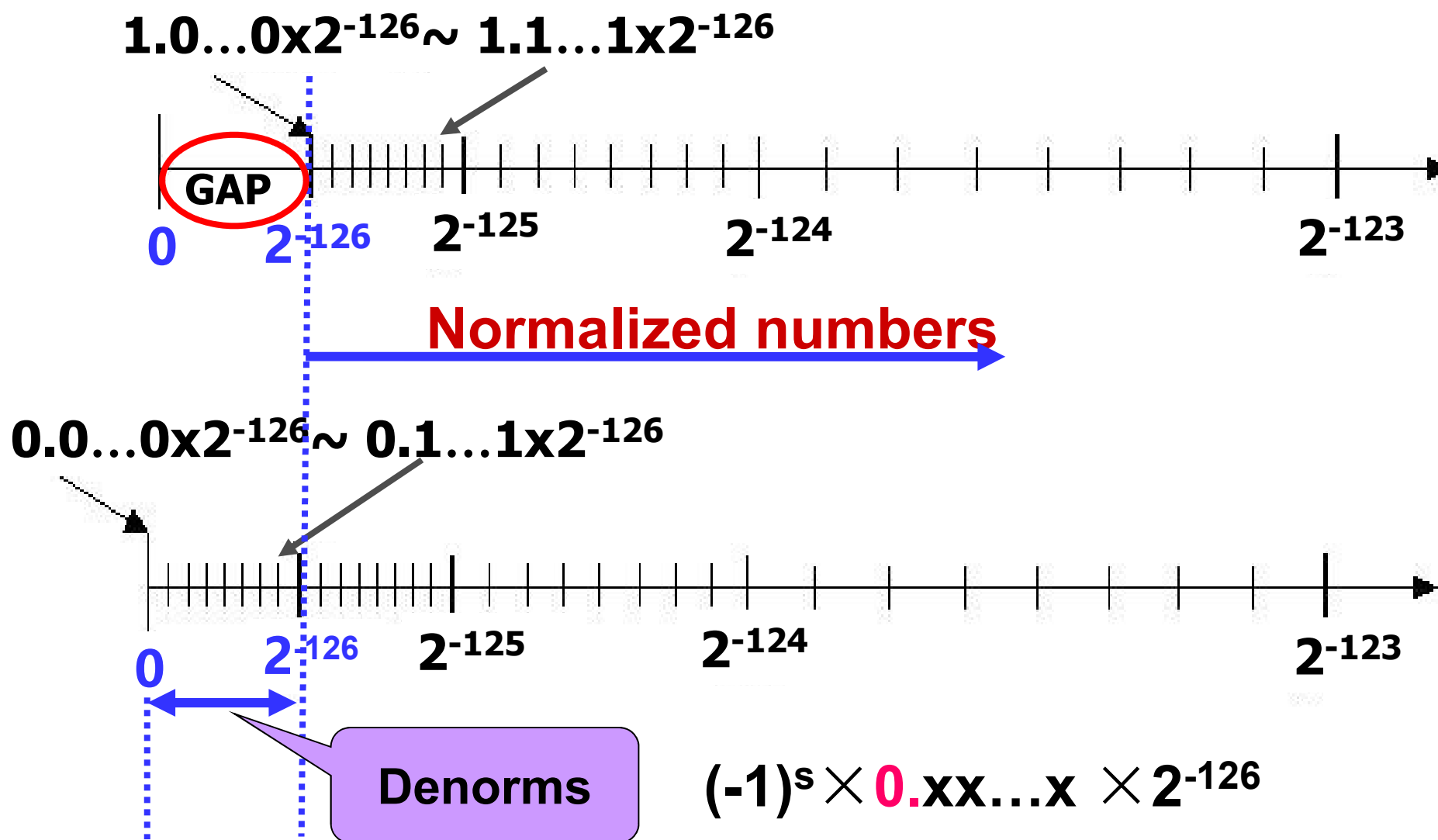
对于单精度FP，还有一种情况没有定义

Exponent	Significand	
0	0	+/-0
0	nonzero	Denorms
1-254	任意 小数点前隐含1	Norms
255	0	+/- infinity
255	nonzero	NaN



用来表示
非规格化数

非规格化数(Denorms)的表示



关于浮点数精度的一个例子

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

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

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:
```



南京大學
NANJING UNIVERSITY



非数值数据的编码表示

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

逻辑数据的编码表示

- 计算机中何时会用到逻辑数据？
 - 表示逻辑（关系）表达式中的逻辑值：真 / 假
 - 例如，对于关系表达式： $(x > 0)$ 并且 $(y \leq 0)$
 - “ $x > 0$ ”、“ $y \leq 0$ ”、“ $(x > 0)$ 并且 $(y \leq 0)$ ” 都是逻辑值
- 表示
 - 用一位表示。N位二进制数（位串）可表示N个逻辑数据
- 运算
 - 按位进行。如，按位与 / 按位或 / 逻辑左移 / 逻辑右移 等
- 识别
 - 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，计算机靠指令来识别。

西文字符的编码表示

- 特点

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

- 表示（常用编码为7位ASCII码）

- 十进制数字：0/1/2.../9
 - 英文字母：A/B/.../Z/a/b/.../z
 - 专用符号：+/-/%/*/&/.....
 - 控制字符（不可打印或显示）
- } 必须熟悉对应的ASCII码！

- 操作

- 字符串操作，如：传送/比较 等

$b_6b_5b_4b_3b_2b_1b_0$

ASCII码表

	$b_6b_5b_4$ =000	$b_6b_5b_4$ =001	$b_6b_5b_4$ =010	$b_6b_5b_4$ =011	$b_6b_5b_4$ =100	$b_6b_5b_4$ =101	$b_6b_5b_4$ =110	$b_6b_5b_4$ =111
$b_3b_2b_1b_0=0000$	NUL	DLE	SP	0	@	P	`	p
$b_3b_2b_1b_0=0001$	SOH	DC1	!	1	A	Q	a	q
$b_3b_2b_1b_0=0010$	STX	DC2	“	2	B	R	b	r
$b_3b_2b_1b_0=0011$	ETX	DC3	#	3	C	S	c	s
$b_3b_2b_1b_0=0100$	EOT	DC4	\$	4	D	T	d	t
$b_3b_2b_1b_0=0101$	ENQ	NAK	%	5	E	U	e	u
$b_3b_2b_1b_0=0110$	ACK	SYN	&	6	F	V	f	v
$b_3b_2b_1b_0=0111$	BEL	ETB	‘	7	G	W	g	w
$b_3b_2b_1b_0=1000$	BS	CAN	(8	H	X	h	x
$b_3b_2b_1b_0=1001$	HT	EM)	9	I	Y	i	y
$b_3b_2b_1b_0=1010$	LF	SUB	*	:	J	Z	j	z
$b_3b_2b_1b_0=1011$	VT	ESC	+	;	K	[k	{
$b_3b_2b_1b_0=1100$	FF	FS	,	<	L	\	l	
$b_3b_2b_1b_0=1101$	CR	GS	-	=	M]	m	}
$b_3b_2b_1b_0=1110$	SO	RS	.	>	N	^	n	~
$b_3b_2b_1b_0=1111$	SI	US	/	?	O	_	o	DEL

汉字及国际字符的编码表示

- 汉字特点

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

- 编码形式

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

问题：西文字符有没有输入码？有没有内码？

有没有字模点阵或轮廓描述？

GB2312-80字符集

- 由三部分组成

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

- 汉字的区位码

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

- 汉字的国标码

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

汉字内码

- 至少需2个字节才能表示一个汉字内码。为什么？

—由汉字的总数（超过6万字）决定！ $2^{16}=65536$

- 可在GB2312国标码的基础上产生汉字内码

—为与ASCII码区别，将国标码的两个字节的第一位置“1”后得到一种汉字内码（可以有不同的编码方案）

例：汉字“大”在码表中位于第20行、第83列。因此区位码为0010100 1010011，在区、位码上各加32得到两个字节编码，即00110100 01110011B=3473H。前面的34H和字符“4”的ASCII码相同，后面的73H和字符“s”的ASCII码相同，但是，将每个字节的最高位各设为“1”后，就得到其内码：B4F3H (10110100 11110011B)，因而不会和ASCII码混淆。

多媒体信息的表示

- 图形、图像、音频、视频等信息在机器内部也用0和1表示
 - 图形用构建图形的直线或曲线的坐标点及控制点来描述，而这些坐标点或控制点则用数值数据描述
 - 图像用构成图像的点（像素）的亮度、颜色或灰度等信息来描述，这些亮度或颜色等值则用数值数据描述
 - 音频信息通过对模拟声音进行采样、量化（用二进制编码）来获得，因此量化后得到的是一个数值数据序列（随时间变化）
 - 视频信息描述的是随时间变化的图像（每一幅图像称为一帧）
 - 音乐信息（MIDI）通过对演奏的乐器、乐谱等相关的各类信息用0和1进行编码来描述
 -

多媒体信息用一个复杂的数据结构来描述，其中的基本数据或者是数值数据，或者是用0/1编码的非数值数据



南京大學
NANJING UNIVERSITY



数据宽度和存储容量的单位

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

数据的基本宽度

- 比特 (bit , 位) 是计算机中处理、存储、传输信息的最小单位
 - 二进制信息最基本的计量单位是 “字节” (Byte)
 - 现代计算机中 , 存储器按字节编址
 - 字节是最小可寻址单位 (*addressable unit*)
 - 如果以字节为一个排列单位 , 则LSB表示最低有效字节 , MSB表示最高有效字节
 - 除比特 (位) 和字节外 , 还经常使用 “字” (word) 作为单位
 - “字” 和 “字长” 的概念不同
- IA-32中的 “字” 有多少位 ? 16位 字长多少位呢 ? 32位
- DWORD : 32位
- QWORD : 64位

数据的基本宽度

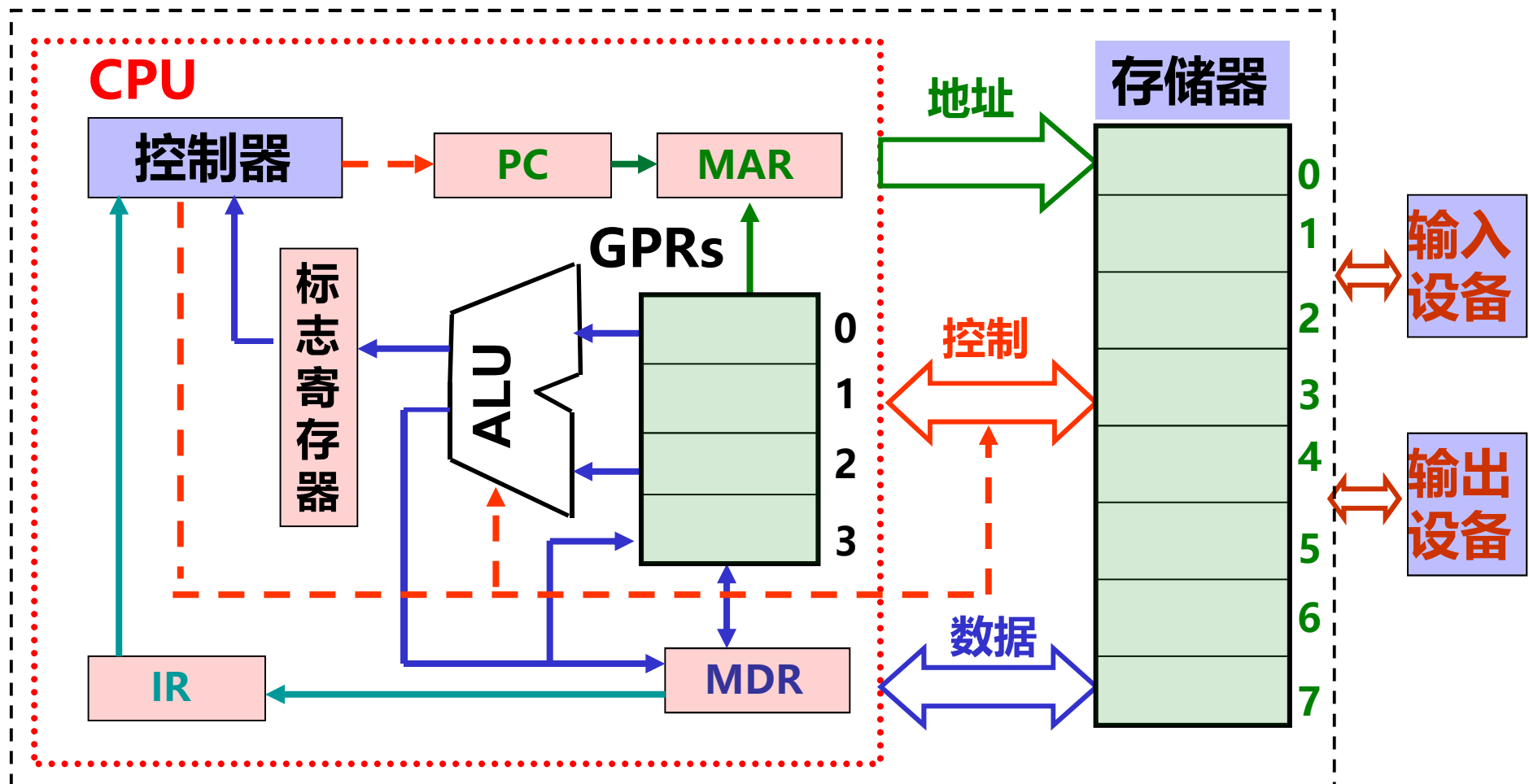
- “字” 和 “字长” 的概念不同
 - “字长” 指数据通路的宽度。
 - “字长” 等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度（这些部件的宽度都是一样的）
 - “字” 表示被处理信息的单位，用来度量数据类型的宽度
 - 字和字长的宽度可以一样，也可不同
 - 例1：对于x86体系结构，不管字长多少，定义“字”的宽度都为16位，而从386开始字长就是32位了。
 - 例2：对于MIPS 32体系结构，其字和字长都是32位。

SKIP

数据通路的宽度

- **数据通路**指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。

[BACK](#)



数据量的度量单位

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

如果把b换成B，则表示字节而不是比特（位）
例如，10MBps表示 10兆字节/秒

程序中数据类型的宽度

- 高级语言支持多种不同类型和不同长度的数据

- 例如，C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）
- 不同机器上表示的同一种类型的数据可能宽度不同

- 必须确定相应的机器级数据表示方式和相应的处理指令

C语言中数据类型的宽度 (单位：字节)

C声明	典型32位机器	典型64位机器
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

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

例如，ANSI C标准未规定long double的确切精度，所以对于不同平台有不同的实现。有的是8字节，有的是10字节，有的是12字节或16字节。



南京大學
NANJING UNIVERSITY



数据存储时的字节排列

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

数据的存储和排列顺序

- 80年代开始，几乎所有通用计算机都采用**字节编址**
- 在高级语言中声明的基本数据类型有char、short、int、long、long long、float、double、long double等各种**不同长度**数据
- 一个基本数据可能会占用多个存储单元

– 例如，若int型变量 $x = -10$ ， x 的存放地址为100，其机器数为FFFFFFF6H，占4个单元

$$-10 = -1010B$$

$$[-10]_{\text{补}} = \text{FFFFFFF6H}$$

- 需要考虑以下问题：

– 变量的地址是其最大地址还是最小地址？

最小地址，即 x 存放在100#~103#！

– 多个字节在存储单元中存放的顺序如何？

大端方式/小端方式

103#	FF	100#
102#	FF	101#
101#	FF	102#
100#	F6	103#

数据的存储和排列顺序

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

$$65535 = 2^{16} - 1$$

$$[-65535]_{\text{补}} = \text{FFFF0001H}$$

变量 i

FF	FF	00	01
103	102	101	100
MSB			LSB
100	101	102	103

小端 (little endian)

大端 (big endian)

大端方式 (Big Endian) : MSB所在的地址是数的地址

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

小端方式 (Little Endian) : LSB所在的地址是数的地址

e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持 , 可通过特定控制位来设定采用哪种方式。

检测系统的字节顺序

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

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

a 和 b 共用同一个空间

100 101 102 103

12	34	56	78	大端 小端
78	56	34	12	

猜测在IA-32
上的打印结果

Little Endian num.b = 0x78

大端/小端方式举例

假定小端方式机器中某条指令的地址为1000

该指令的汇编形式为：`mov AX, 0x12345(BX)`

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



若在大端机器上，则存放顺序如何？



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

只需要考虑指令中立即数的顺序！

大端/小端方式举例

- 以下是一个由反汇编器生成的一行针对IA-32处理器的机器级代码表示文本：

80483d2: 89 85 a0 fe ff ff mov %eax, 0xfffffea0(%ebp)

其中，80483d2是十六进制表示的指令地址

89 85 a0 fe ff ff 是机器指令

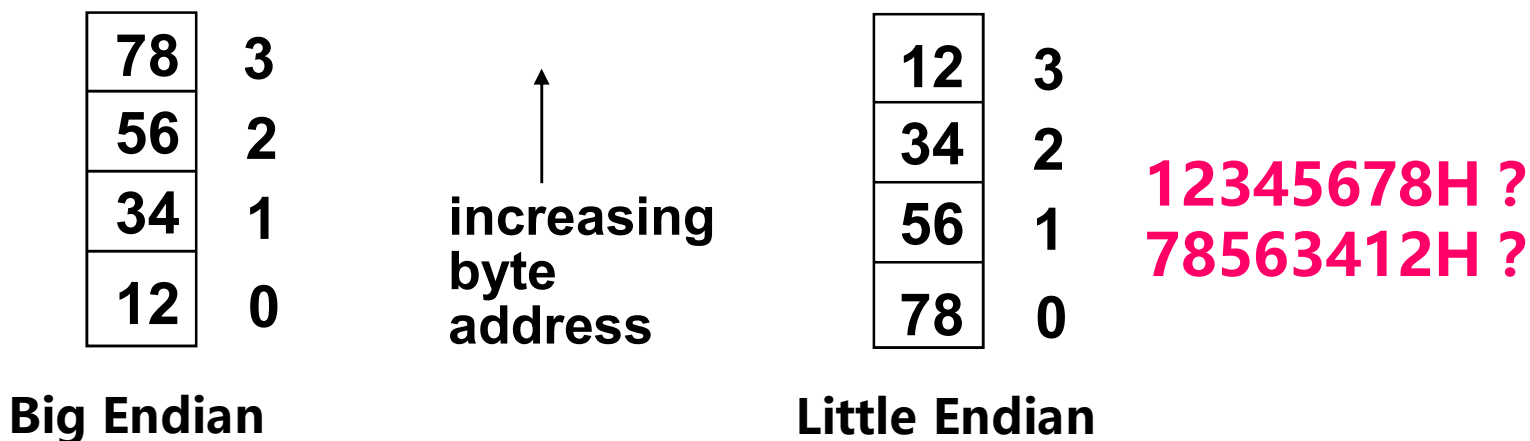
mov %eax, 0xfffffea0(%ebp) 是对应的汇编指令

0xfffffea0是立即数

请问：立即数0xfffffea0的值和所存放地址分别是多少？
IA-32是大端还是小端方式？

- 立即数0xfffffea0所存放的地址为0x80483d4；
- 立即数0xfffffea0的值为-10110000B=-176；
- IA-32采用的是小端方式！

字节交换问题



上述存放在0号单元的数据（字）是什么？

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

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

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

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

Big endian: Adobe Photoshop, JPEG, MacPaint, etc