Ch2: Data Representation

数据的机器级表示

第一讲 数值数据的表示

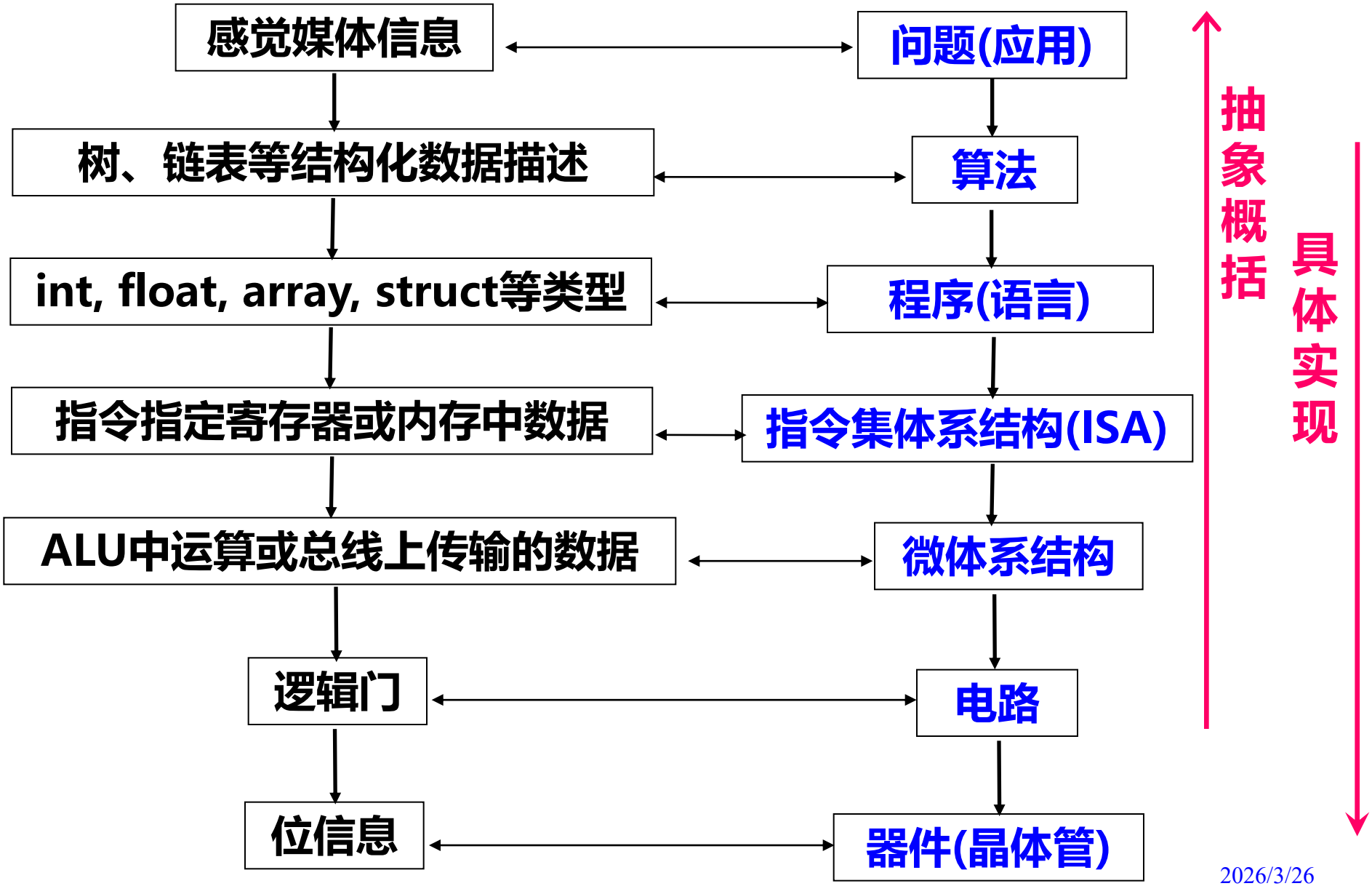
第二讲 非数值数据表示及
数据的宽度、存储排列

第一讲：数值数据的表示

主　要　内　容

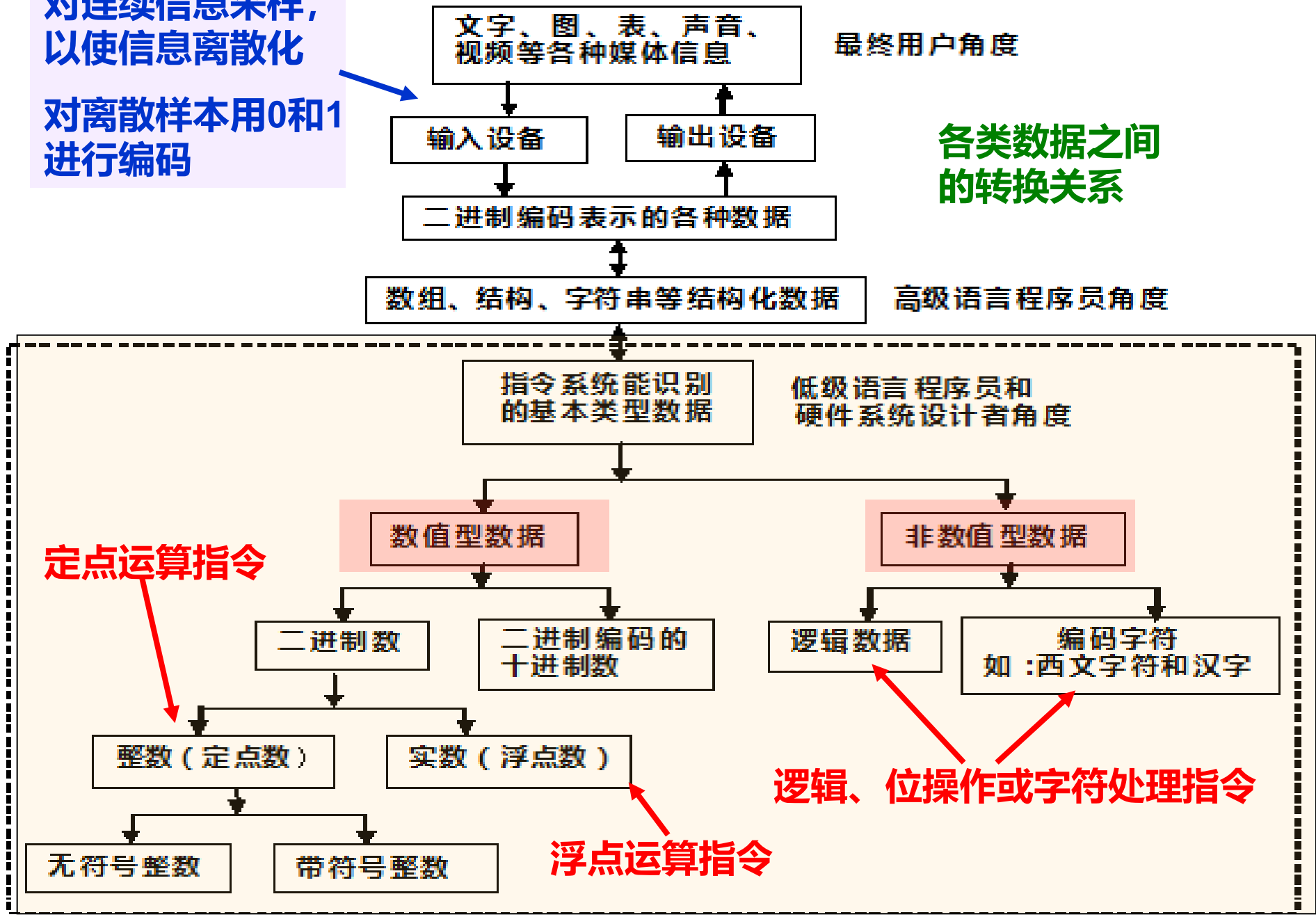
- 定点数的表示
  - 进位记数制
  - 定点数的二进制编码
    - 原码、补码、移码表示
  - 定点整数的表示
    - 无符号整数、带符号整数
- 浮点数格式和表示范围
- 浮点数的规格化
- IEEE754浮点数标准
  - 单精度浮点数、双精度浮点数
  - 特殊数的表示形式
- C语言程序中的整数类型、浮点数类型
- 十进制数表示

“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



信息的二进制编码

- 计算机的外部信息与内部机器级数据

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

首先考虑数值数据的表示

- 真值和机器数
  - 机器数：用0和1编码的计算机内部的0/1序列
  - 真值：机器数真正的值，即：现实中带正负号的数

数值数据的表示

- 数值数据表示的三要素
  - 进位计数制
  - 定、浮点表示
  - 如何用二进制编码

  即：要确定一个数值数据的值必须先确定这三个要素。

  例如，机器数 01011001的值是多少？ 答案是：不知道！
- 进位计数制
  - 十进制、二进制、十六进制、八进制数及其相互转换
- 定/浮点表示（解决小数点问题）
  - 定点整数、定点小数
  - 浮点数（可用一个定点小数和一个定点整数来表示）
- 定点数的编码（解决正负号问题）
  - 原码、补码、反码、移码 （反码很少用）

Decimal / Binary（十 / 二进制数）

- The decimal number 5836.47 in powers of 10:

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

- The binary number 11001 in powers of 2 :

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

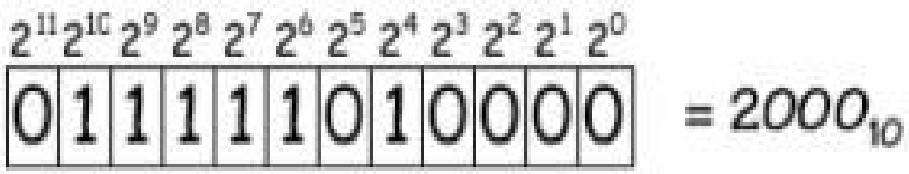
$$= 16 + 8 + 0 + 0 + 1 = 25$$

- 用一个下标表示数的基（ radix / base)

  或用后缀B-二进制（H-十六进制（前缀0x-）、O-八进制）

  $11001_2 = 25_{10}$、 11001B= 25

Octal / Hexadecimal (八 / 十六进制数)



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

$2^3=8$

03720

Octal - base 8

000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7

$2^4=16$

0x7d0

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - a
0011 - 3	1011 - b
0100 - 4	1100 - c
0101 - 5	1101 - d
0110 - 6	1110 - e
0111 - 7	1111 - f

一个8进制数字用3位二进制数字表示

一个16进制数字用4位二进制数字表示

计算机用二进制表示所有信息!

为什么要引入 8 / 16进制?

8 / 16进制是二进制的简便表示。便于阅读和书写!

它们之间对应简单，转换容易。

在机器内部用二进制，在屏幕或其他外部设备上表示时，转换为10进制或8/16进制数，可缩短长度

早期有用8进制数简便表示2进制数

现在基本上都用16进制数表示机器数

Conversions of numbers

(1) 二、八、十六进制数的相互转换

① 八进制数转换成二进制数

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

② 十六进制数转换成二进制数

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

③ 二进制数转换成八进制数

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

④ 二进制数转换成十六进制数

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

Conversions of numbers

(3) 十进制数 => R进制数

整数部分和小数部分分别转换

① 整数(integral part)----“除基取余，上右下左”

② 小数(fractional part)----“乘基取整，上左下右”

} 理论上的做法

实际按简便方法先转换为二进制数，再按需转换为8/16进制数

整数：2、4、8、16、…、512、1024、2048、4096、…、65536

小数：0.5、0.25、0.125、0.0625、0.03125、……

例：4123.25=4096+16+8+2+1+0.25=1 0000 0001 1011.01B

$=(101B.4)_{16}$

4023=(4096-1)-64-8=1111 1111 1111B-100 0000B-1000B

=1111 1011 0111B = = FB7H=$(FB7)_{16}$

Sign and Magnitude （原码的表示）

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	Binary
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

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

从 50年代开始，整数都采用补码来表示
但浮点数的尾数用原码定点小数表示

补码特性 - 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　　**（13 mod 12 等于1，即13点钟等于1点钟）**

假定钟表时针指向10点，要将它拨向6点，

有两种拨法：

① 倒拨4格：10- 4 = 6

② 顺拨8格：10+8 = 18 ≡ 6 (mod 12)

模12系统中：10- 4 ≡ 10+8 (mod 12)

- 4 ≡ 8 (mod 12)

- 4的模12补码等于8。

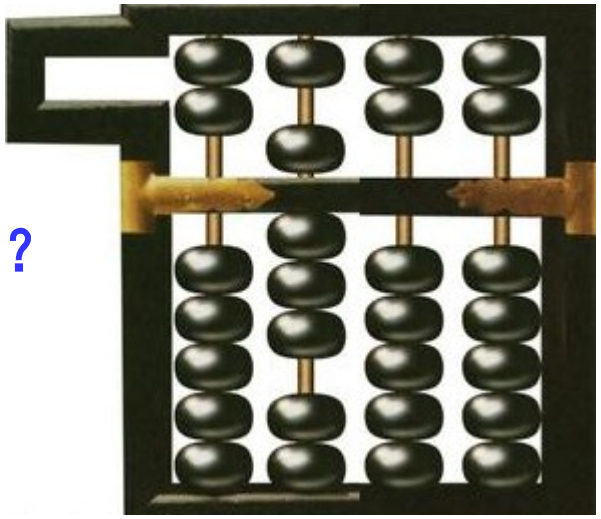
同样有 -3 ≡ 9 (mod 12)；-5 ≡ 7 (mod 12) 等

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，数x减去小于模的数y，总可以用数x加上-y的补码来代替。

补码（modular运算）：实现 + 和 – 的统一

补码的表示



现实世界的模运算系统举例

例1：“钟表”模运算系统

假定时针只能顺拨，从10点倒拨4格后是几点？

10- 4 = 10+(12- 4) = 10+8 = 6 （mod 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？

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

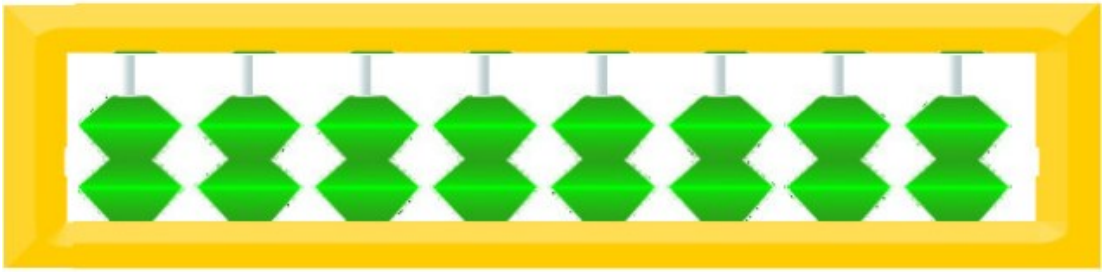
$=9828+8072$

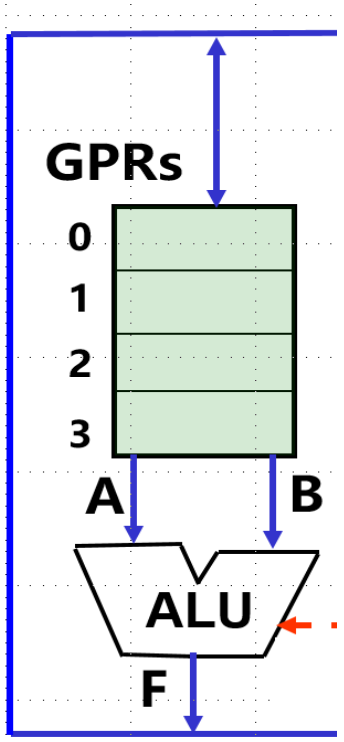
$=\boxed{1}\,7900$

$=7900 \pmod{10^4}$

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

计算0111 1111 - 0100 0000 = ?

0111 1111 - 0100 0000 = 0111 1111 + (2^8- 0100 0000)

=0111 1111 + 1100 0000 = 1 0011 1111 (mod 2^8)

= 0011 1111

只留余数，1被丢弃

结论1： 一个负数的补码等于对应正数补码的“各位取反、末位加1”

求特殊数的补码

假定机器数有n位

① $[-2^{n-1}]_{补} = 2^n - 2^{n-1} = 10\ldots0$（n-1个0） （mod 2^n）

② $[-1]_{补} = 2^n - 0\ldots01 = 11\ldots1$（n个1） （mod 2^n）

③ $[-1.0]_{补} = 2 - 1.0 = 1.00\ldots0$（n-1个0） （mod 2）

④ $[+0]_{补} = [-0]_{补} = 00\ldots0$（n个0）

注：计算机中并不会出现-1.0的补码，这里只是想说明同一个真值在机器中可能有不同的机器数！

Two's Complement （补码的表示）

- 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（模4）补码：双符号，用于存放可溢出的中间结果。

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

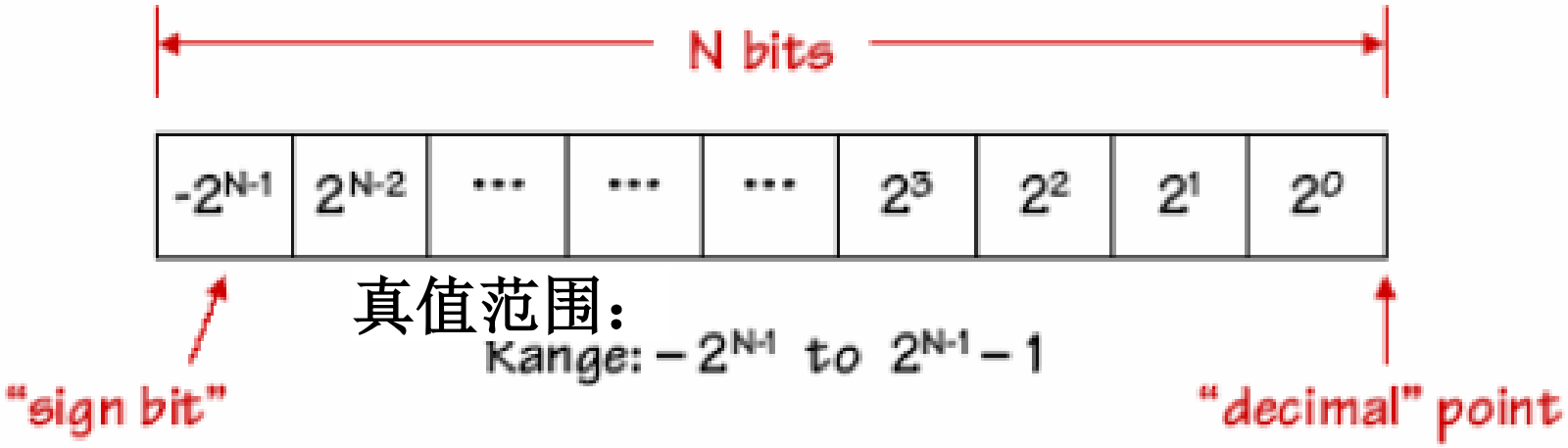
+0和-0表示唯一

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。

如何求补码的真值

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

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



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

符号为0，则为正数，数值部分同

符号为1，则为负数，数值各位取反，末位加1

例如：补码“11010110”的真值为：-0101010=-(32+8+2)=-42

Excess (biased) notion- 移码表示

- 什么是“excess (biased) notation-移码表示”？

  将每一个数值加上一个偏置常数（ Excess / bias）

- 一般来说，当编码位数为n时，bias取 2^{n-1}

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

  -8 (+8) ~ 0000B

  -7 (+8) ~ 0001B

  …

  0 (+8) ~ 1000B

  …

  +7 (+8) ~ 1111B

  0的移码表示唯一

  移码和补码仅第一位不同

- 为什么要用移码来表示指数（阶码）？

  移码主要用来表示浮点数阶码！

  便于浮点数加减运算时的对阶操作（比较大小）

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

补码： 111 < 011 ?
(-1) (3)

简化比较

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

移码：011 < 111
(3) (7)

Unsigned integer(无符号整数)

- 机器中字的位排列顺序有两种方式：（例：32位字: $0\ldots01011_2$）
  - 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场合下，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”

Signed integer（带符号整数，定点整数）

- 计算机必须能处理正数(positive) 和负数(negative)，MSB表示数符
- 有三种定点编码方式
  - Signed magnitude （原码）

    现用来表示浮点（实）数的尾数
  - One's complement （反码）

    现已不用于表示数值数据
  - Two's complement （补码）

    50年代以来，所有计算机都用补码来表示定点整数
- 为什么用补码表示带符号整数？
  - 补码运算系统是模运算系统，加、减运算统一
  - 数0的表示唯一，方便使用
  - 比原码和反码多表示一个最小负数
  - 与移码相比，其符号位和真值的符号对应关系清楚

带符号整数和无符号数的比较

- 扩充操作有差别
  - 例如，MIPS提供了两种加载指令（load byte unsigned / load byte）
    - 无符号数：lbu $t0, 0($s0) ；$t0高24位补0 （称为0扩展）
    - 带符号整数： lb $t0, 0($s0) ；$t0高24位补符 （称为符号扩展）
- 数的比较有差异
  - 无符号数：MSB为1的数比MSB为0的数大
  - 带符号整数： MSB为1的数比MSB为0的数小
  - 例如，MIPS中提供了不同的比较指令，如：
    - 无符号数：sltu $t0, $s0, $s1 （set less than unsigned）
    - 带符号整数： slt $t1, $s0, $s1 （set less than）

  假定： $s0=1111 1111 1111 1111 1111 1111 1111 1111

  $s1=0000 0000 0000 0000 0000 0000 0000 0001

  则：$t0和$t1分别为多少？ 答案：$t0和$t1分别为0和1。
- 溢出判断有差异（无符号数根据最高位是否有进位判断溢出，通常不判）
  - MIPS规定：无符号数运算溢出时，不产生“溢出异常”

SKIP

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			

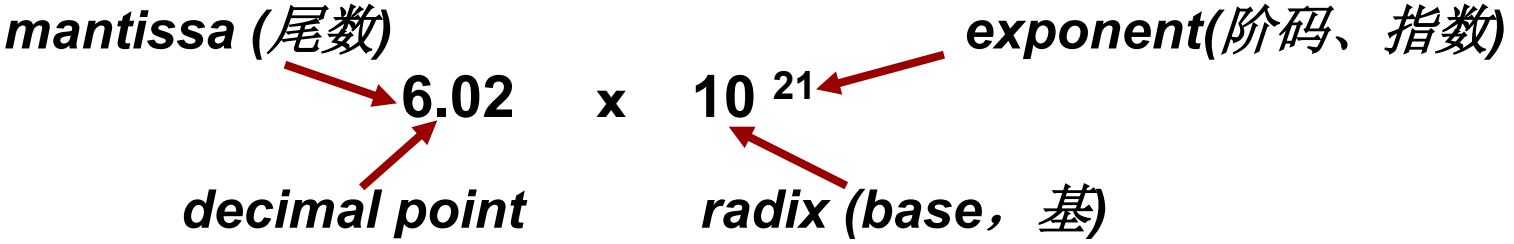
C语言程序中的整数

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！

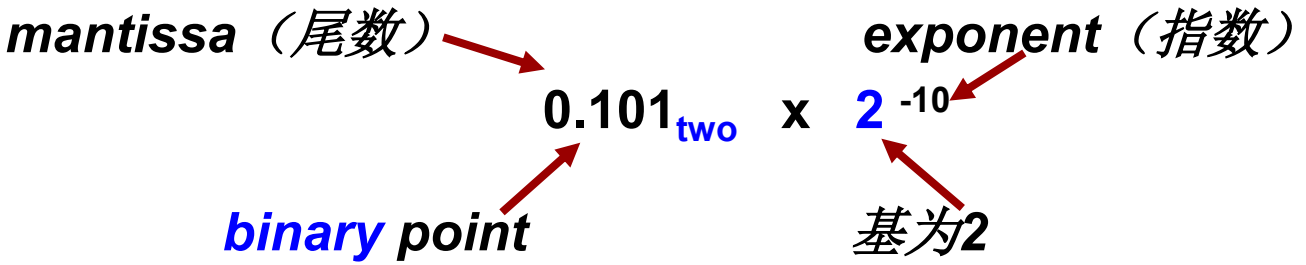
科学计数法(Scientific Notation)与浮点数

Example:



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

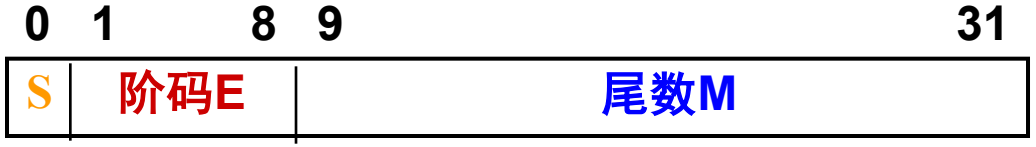
for Binary Numbers:



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。

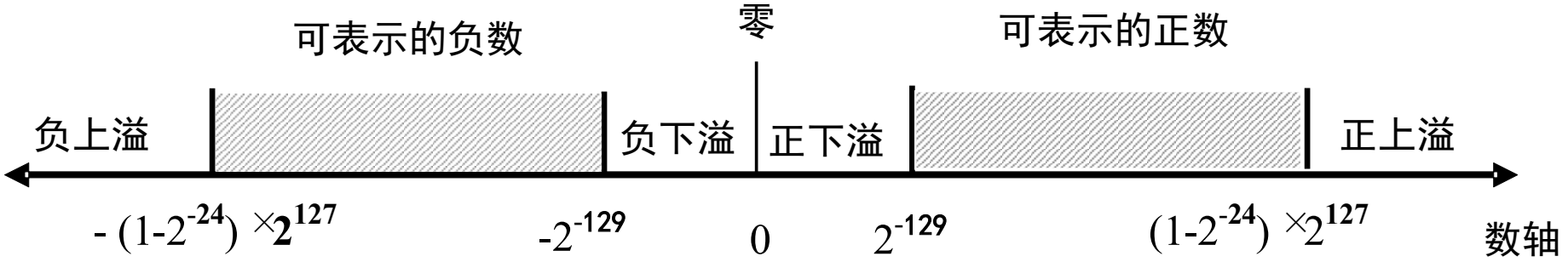


$+/-0.1xxxxx \times 2^{E}$

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

最大正数：$0.11\ldots1 \times 2^{11\ldots1} = (1-2^{-24}) \times 2^{127}$　最小正数：$0.10\ldots0 \times 2^{00\ldots0} = (1/2) \times 2^{-128}$

因为原码是对称的，所以其表示范围关于原点对称。



机器0：尾数为0 或 落在下溢区中的数

浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

浮点数的表示

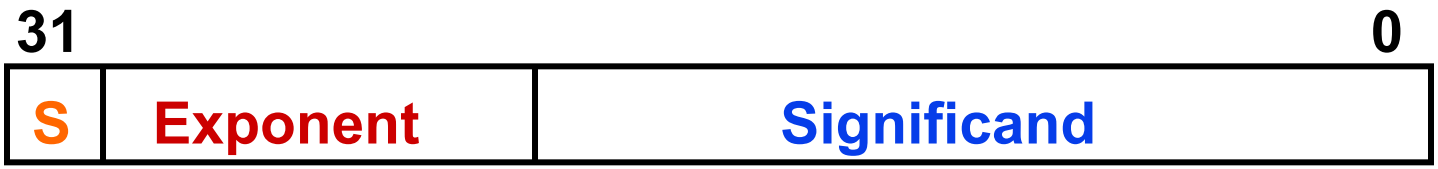
规定：小数点前总是“1”，故可隐含表示

注意：和前面例子的规定不太一样，显然这里更合理！

- Normal format（规格化数形式）：

  +/-1.xxxxxxxxxx × $2^{Exponent}$

- 32-bit 规格化数：



S 是符号位（Sign）

Exponent用移码（增码）来表示

Significand 表示 xxxxxxxxxxxxx，尾数部分

(基可以是 2/ 4 / 8 / 16，约定信息，无需显式表示)

- 早期的计算机，各自定义自己的浮点数格式

问题：浮点数表示不统一会带来什么问题？

“Father” of the IEEE 754 standard

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

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

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.

1989
ACM Turing
Award Winner!

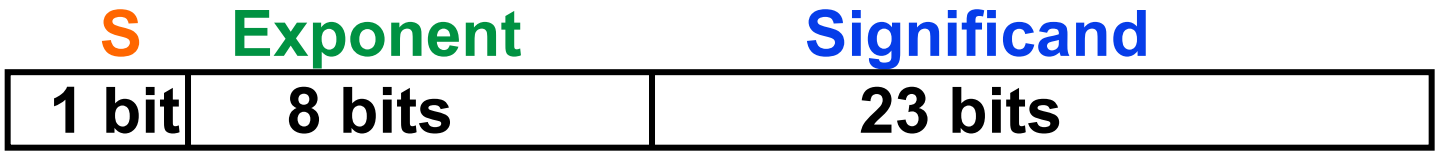
www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

IEEE 754 Floating Point Standard

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

Single Precision ： (Double Precision is similar)



- Sign bit: 1 表示negative ; 0表示 positive
- Exponent（阶码 / 指数）：

全0和全1用来表示特殊值！

  - SP规格化数阶码范围为0000 0001 (-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（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: $(-1)^S$ x (1 + Significand) x $2^{(Exponent-127)}$

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

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

Ex: Converting Binary FP to Decimal

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

1	0111 1101	110 0000 0000 0000 0000 0000

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

- **Sign**: 1 => negative
- **Exponent**:
  - $0111\ 1101_{two} = 125_{ten}$
  - Bias adjustment: 125 - 127 = -2
- **Significand**:

$$1 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 0\times2^{-4} + 0\times2^{-5} + \ldots$$

$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

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

Ex: Converting Decimal to FP

-12.75

1. Denormalize: -12.75
2. Convert integer part:

   $12 = 8 + 4 = 1100_2$
3. Convert fractional part:

   $.75 = .5 + .25 = .11_2$
4. Put parts together and normalize:

   $1100.11 = 1.10011 \times 2^3$
5. Convert exponent: $127 + 3 = 128 + 2 = 1000\ 0010_2$

1	1000 0010	100 1100 0000 0000 0000 0000

The Hex rep. is C14C0000H

Normalized numbers（规格化数）

前面的定义都是针对规格化数（normalized form）

How about other patterns?

Exponent	Significand	Object
1-254	anything implicit leading 1	Norms
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

Representation for 0

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 00000000000000000000000

-0: 1 00000000 00000000000000000000000

Representation for +∞/-∞

∞ : infinity

In FP, 除数为0的结果是 +/- ∞, 不是溢出异常.（整数除0为异常）

为什么要这样处理?

- 可以利用+∞/-∞作比较。 例如：X/0>Y可作为有效比较

How to represent +∞/-∞?

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

+∞ : 0 11111111 00000000000000000000000

-∞ : 1 11111111 00000000000000000000000

Operations

5.0 / 0 = +∞, -5.0 / 0 = -∞

5+(+∞) = +∞, (+∞)+(+∞) = +∞

5 - (+∞) = -∞, (-∞) - (+∞) = -∞ etc

Representation for “Not a Number”

Sqrt (- 4.0) = ? 0/0 = ?

- Called Not a Number (NaN) - “非数”

How to represent NaN

Exponent = 255

Significand: nonzero

NaNs can help with debugging

Operations

sqrt (-4.0) = NaN 0/0 = NaN

op (NaN,x) = NaN +∞+(-∞) = NaN

+∞- (+∞) = NaN ∞/∞ = NaN

etc.

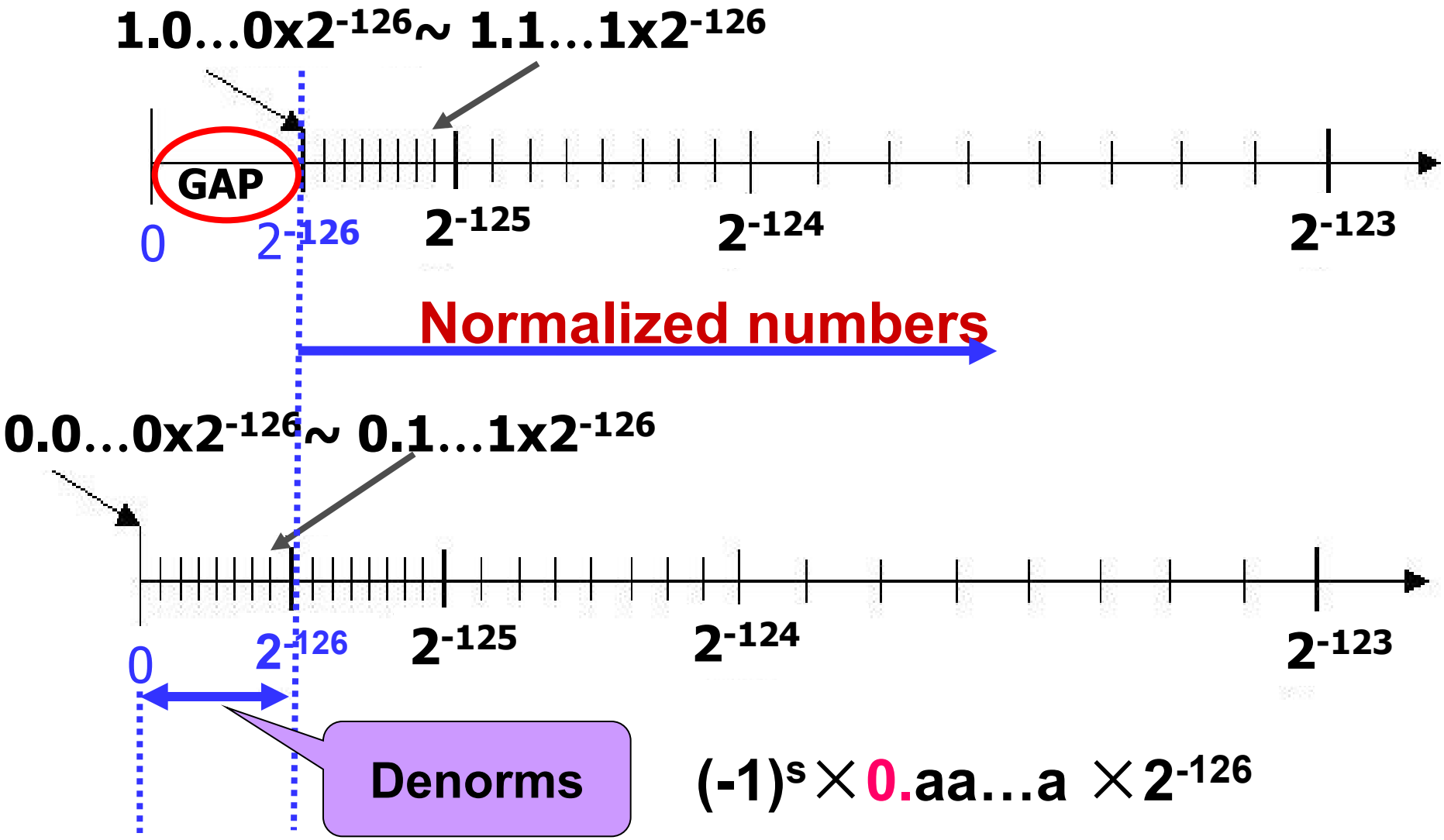
Representation for Denorms(非规格化数)

What have we defined so far? (for SP)

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

Used to represent Denormalized numbers

Representation for Denorms



Questions about IEEE 754

- What's the range of representable values?

  The largest number for single: +1.11…1x 2^{127} 约 +3.4 x 10^{38}

  How about double? 约 +1.8 x 10^{308}

- What about following type converting: not always true!

```
if ( i == (int) ((float) i) )  {
      printf ("true");
}
if ( f == (float) ((int) f) )  {
      printf ("true");
}
```

How about double? True!

How about double? Not always true!

- How about FP add associative? FALSE!

  x = – 1.5 x 10^{38}, y = 1.5 x 10^{38}, z = 1.0

  (x+y)+z = (–1.5x10^{38}+1.5x10^{38}) +1.0 = 1.0

  x+(y+z) = –1.5x10^{38}+ (1.5x10^{38}+1.0) = 0.0

十进制数的表示

- 数值数据（numerical data）的两种表示

  Binary (二进制数)

  - 定点整数：Fixed-point number (integer)
    - Unsigned and signed int
  - 浮点数：Floating-point number (real number)

  Decimal (十进制数)

  - 用ASCII码表示
  - 用BCD（Binary coded Decimal）码表示

- 计算机中为什么要用十进制数表示数值？
  - 日常使用的都是十进制数，所以，计算机外部都使用十进制数。在一些有大量数据输入/出的系统中，为减少二进制数和十进制数之间的转换，在计算机内部直接用十进制数表示数值。

用ASCII码表示十进制数

- 前分隔数字串
  - 符号位单独用一个字节表示，位于数字串之前。
  - 正号用“+”的ASCII码(2BH)表示；负号用“-”的ASCII码(2DH)表示
  - 例：十进制数+236表示为: 2B 32 33 36H

    0010 1011 0011 0010 0011 0011 0011 0110B

    十进制数-2369表示为: 2D 32 33 36 39H

    0010 1101 0011 0010 0011 0011 0011 0110 0011 1001B
- 后嵌入数字串
  - 符号位嵌入最低位数字的ASCII码高4位中。比前分隔方式省一个字节。
  - 正数不变；负数高4位变为0111.
  - 例：十进制数+236表示为: 32 33 36H

    0011 0010 0011 0011 0011 0110B

    十进制数-2369表示为: 32 33 36 79H

    0011 0010 0011 0011 0011 0110 0111 1001B

缺点：占空间大，且需转换成二进制数或BCD码才能计算。

用BCD码表示十进制数

- 编码思想： 每个十进数位至少有4位二进制表示。而4位二进制位可组合成16种状态，去掉10种状态后还有6种冗余状态。
- 编码方案
  1. 十进制有权码
     - 每个十进制数位的4个二进制位（称为基2码）都有一个确定的权。8421码是最常用的十进制有权码。也称自然BCD（NBCD）码。
  2. 十进制无权码
     - 每个十进制数位的4个基2码没有确定的权。在无权码方案中，用的较多的是余3码和格雷码。
  3. 其他编码方案 （5中取2码、独热码等）
- 符号位的表示：
  - “+”：1100 ； “-”：1101
  - 例：+236=$(1100\ 0010\ 0011\ 0110)_{8421}$ (占2个字节)

    -2369=$(1101\ 0000\ 0010\ 0011\ 0110\ 1001)_{8421}$ (占3个字节)

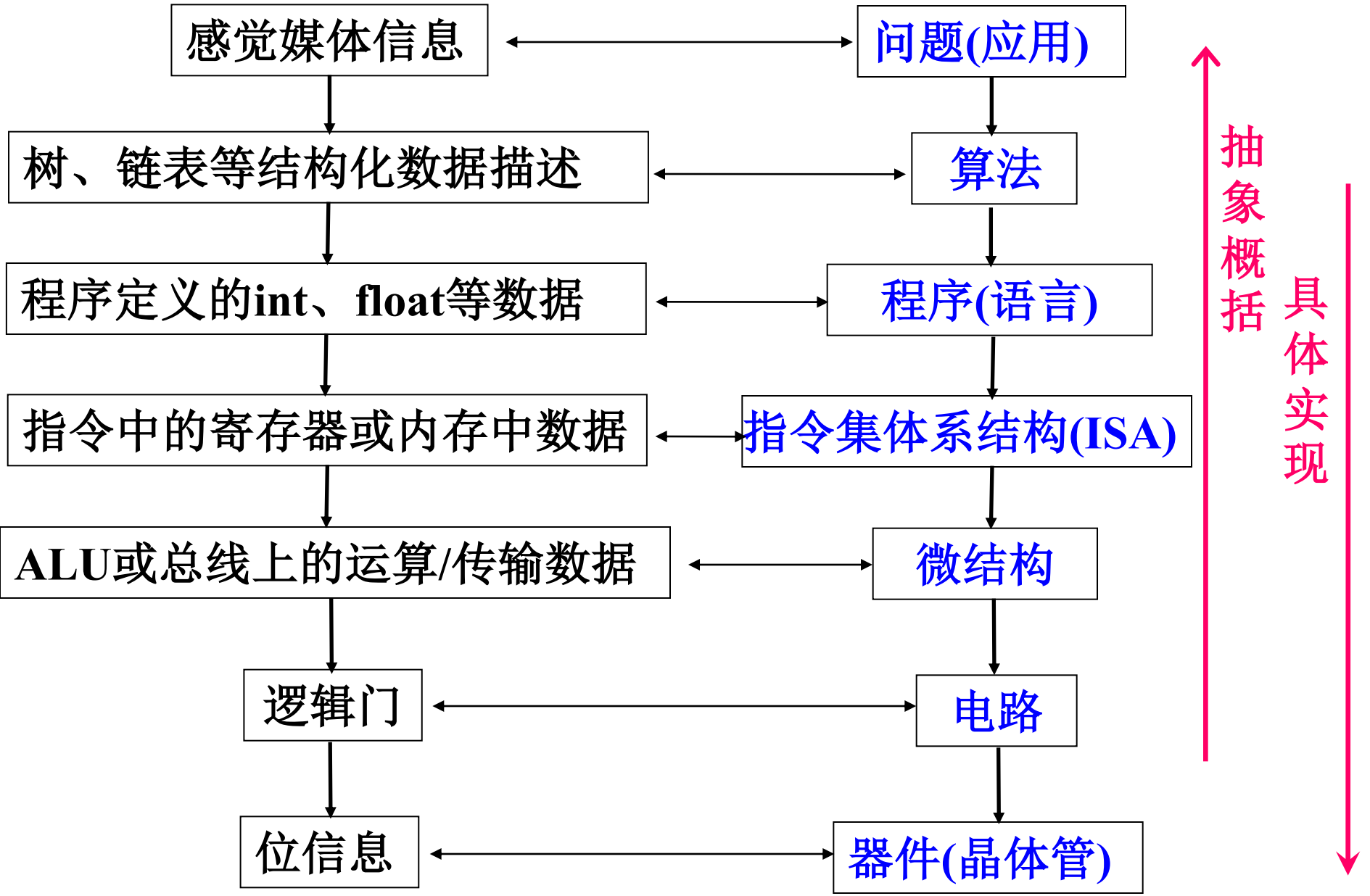
    补0以使数占满一个字节

第一讲小结

10在计算机中有几种可能的表示？
-10呢？

- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进制、定点/浮点、编码
- 整数的表示
  - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- C语言中的整数
  - 无符号数：unsigned int (short / long)；带符号数： int (short / long)
- 浮点数的表示
  - 符号；尾数：定点小数；指数（阶）：定点整数（基不用表示）
- 浮点数的范围
  - 正上溢、正下溢、负上溢、负下溢；与阶码的位数和基的大小有关
- 浮点数的精度：与尾数的位数和是否规格化有关
- 浮点数的表示（IEEE 754标准）：单精度SP（float）和双精度DP（double）
  - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
  - “零” (阶为全0，尾为全0)
  - ∞ (阶为全1，尾为全0)
  - NaN (阶为全1，尾为非0)
  - 非规格化数 (阶为全0，尾为非0，隐藏位为0)（P.42倒数第9行说明）
- 十进制数的表示：用ASCII码或BCD码表示

“转换”的概念在数据表示中的反映



第二讲　非数值数据、数据排列

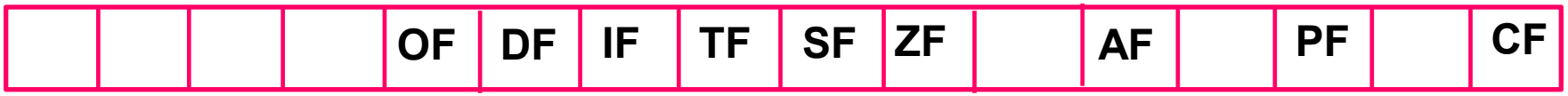
主　要　内　容

- ◆非数值数据的表示
  - 逻辑数据、西文字符、汉字
- ◆数据的宽度
- ◆数据的存储排列
  - 大端方式、小端方式

逻辑数据的编码表示

- 表示
  - 用一位表示 。例如，真：1 / 假：0
  - N位二进制数可表示N个逻辑数据，或一个位串
- 运算
  - 按位进行
  - 如:按位与 / 按位或 / 逻辑左移 / 逻辑右移 等
- 识别
  - 逻辑数据和数值数据在形式上并无差别，也是一串0/1序列，机器靠指令来识别。
- 位串
  - 用来表示若干个状态位或控制位（OS中使用较多）

例如，x86的标志寄存器含义如下：



西文字符的编码表示

- 特点
  - 是一种拼音文字，用有限几个字母可拼写出所有单词
  - 只对有限个字母和数学符号、标点符号等辅助字符编码
  - 所有字符总数不超过256个，使用7或8个二进位可表示
- 表示（常用编码为7位ASCII码）
  - 十进制数字：0/1/2…/9
  - 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

汉字及国际字符的编码表示

- 特点
  - 汉字是表意文字，一个字就是一个方块图形。
  - 汉字数量巨大，总数超过6万字，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。
- 编码形式
  - 有以下几种汉字代码：
  - 输入码：对汉字用相应按键进行编码表示，用于输入
  - 内码：用于在系统中进行存储、查找、传送等处理
  - 字模点阵或轮廓描述: 描述汉字字模点阵或轮廓，用于显示/打印

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？

字符集与汉字的内码

问题：西文字符常用的内码是什么？ **其内码就是ASCII码。**

对于汉字内码的选择，必须考虑以下几个因素：

① 不能有二义性，即不能和ASCII码有相同的编码。

② 尽量与汉字在字库中的位置有关，便于汉字查找和处理。

③ 编码应尽量短。

国标码（国标交换码）

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

可在汉字国标码的基础上产生汉字机内码

国际字符集

国际字符集的必要性

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

国际多字符集

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

数据的基本宽度

- 比特（bit）是计算机中处理、存储、传输信息的最小单位
- 二进制信息的计量单位是“字节”(Byte)，也称“位组”
  - 现代计算机中，存储器按字节编址
  - 字节是最小可寻址单位 *(addressable* unit *)*
  - 如果以字节为一个排列单位，则LSB表示最低有效字节，MSB表示最高有效字节
- 除比特和字节外，还经常使用“字”(word)作为单位
- “字”和 “字长”的概念不同

IA-32中的“字”有多少位？字长多少位呢？

16位　　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”指定点运算数据通路的宽度。

    （数据通路指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的宽度基本上要一致，才能相互匹配。因此，”字长”等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。）
  - “字”表示被处理信息的单位，用来度量数据类型的宽度。
  - 字和字长的宽度可以一样，也可不同。

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

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)，1KB=2^{10}字节=1024B
  - “兆字节”(MB)，1MB=2^{20}字节=1024KB
  - “千兆字节”(GB)，1GB=2^{30}字节=1024MB
  - “兆兆字节”(TB)，1TB=2^{40}字节=1024GB
- 通信中的带宽使用的单位有：
  - “千比特/秒”(kb/s)，1kbps=10^3 b/s=1000 bps
  - “兆比特/秒”(Mb/s)，1Mbps=10^6 b/s =1000 kbps
  - “千兆比特/秒”(Gb/s)，1Gbps=10^9 b/s =1000 Mbps
  - “兆兆比特/秒”(Tb/s)，1Tbps=10^{12} b/s =1000 Gbps

如果把b换成B，则表示字节而不是比特（位）

例如，10MBps表示 10兆字节/秒

程序中数据类型的宽度

- 高级语言支持多种类型、多种长度的数据
  - 例如，C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）
  - 不同机器上表示的同一种类型的数据可能宽度不同
- 程序中的数据有相应的机器级表示方式和相应的处理指令

(在第五章指令系统介绍具体指令)

C语言中数值数据类型的宽度 (单位：字节)

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

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，分配的字节数随机器字长和编译器的不同而不同。

Compaq Alpha是一个针对高端应用的64位机器，即字长为64位

数据的存储和排列顺序

$65535=2^{16}-1$

$[-65535]_{补}=FFFF0001H$

- 80年代开始，几乎所有通用机器都用字节编址
- ISA设计时要考虑的两个问题：
  - 如何根据一个地址取到一个32位的字？- 字的存放问题
  - 一个字能否存放在任何地址边界？- 字的边界对齐问题

例如，若 int i = -65535，存放在内存100号单元（即占100#～103#），则用“取数”指令访问100号单元取出 i 时，必须清楚 i 的4个字节是如何存放的。

Word:

FF	FF	00	01
103	102	101	100
msb			lsb
100	101	102	103

little endian word 100#

big endian word 100#

大端方式（Big Endian）：MSB所在的地址是数的地址

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

小端方式（Little Endian）：LSB所在的地址是数的地址

e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持，可通过特定控制位来设定采用哪种方式。

BIG Endian versus Little Endian

Ex1: Memory layout of a number ABCDH located in 1000

In Big Endian: →

CD	1001
AB	1000

In Little Endian: →

AB	1001
CD	1000

Ex2: Memory layout of a number 00ABCDEFH located in 1000

In Big Endian: →

00	1000
AB	1001
CD	1002
EF	1003

In Little Endian: →

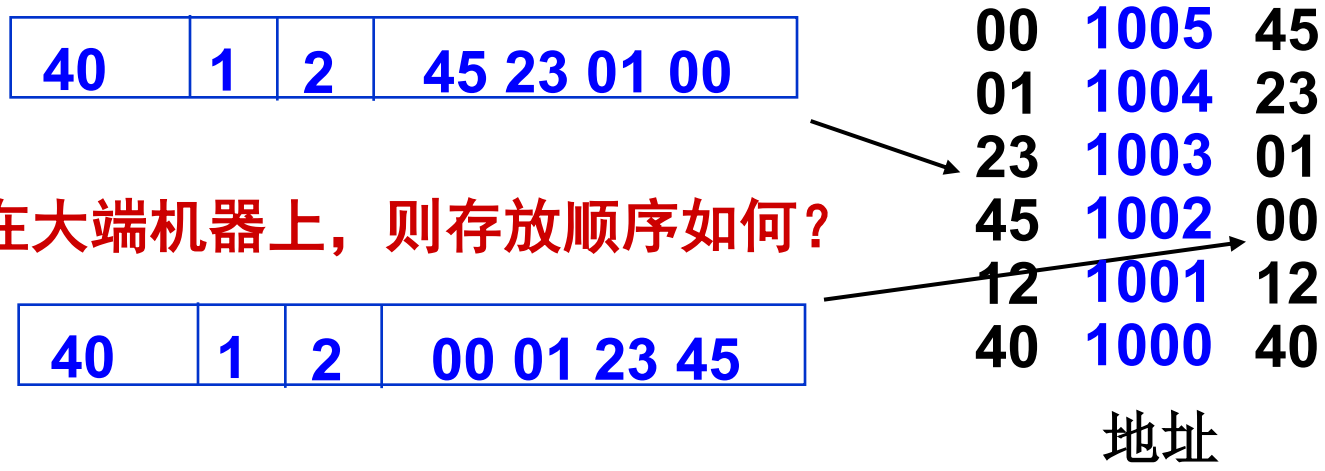
00	1003
AB	1002
CD	1001
EF	1000

BIG Endian versus Little Endian

Ex3: Memory layout of a instruction located in 1000

假定小端机器中指令：mov AX, 0x12345(BX)

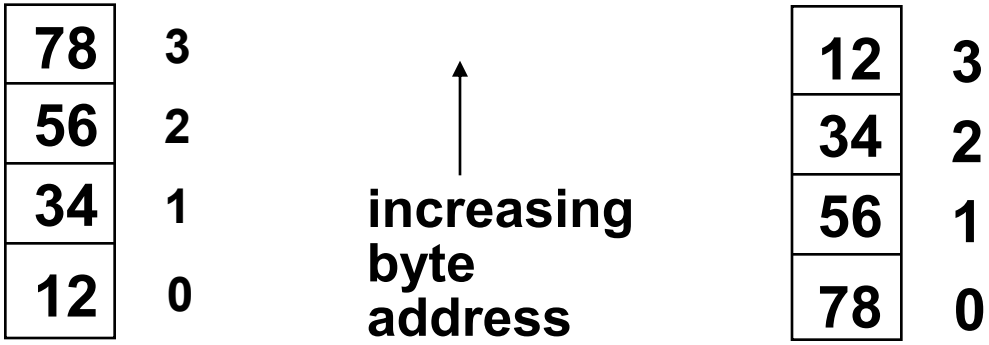
其中操作码mov为40H，寄存器AX和BX的编号分别为0001B和0010B，立即数占32位，则存放顺序为：



若在大端机器上，则存放顺序如何？

只需要考虑指令中立即数的顺序！

Byte Swap Problem（字节交换问题）



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

存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

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

音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

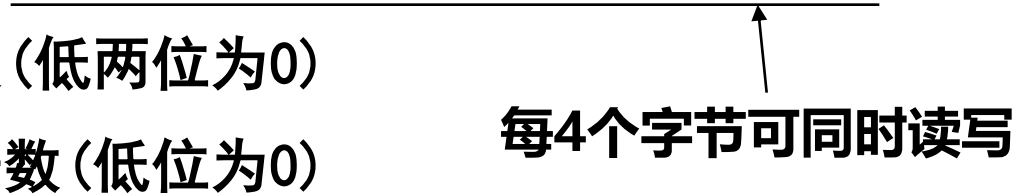
ex. Little endian: GIF, PC Paintbrush, Microsoft RTF,etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长一般为32位或64位，而存储器地址按字节编址
- 指令系统支持对字节、半字、字及双字的运算，也有位处理指令
- 各种不同长度的数据存放时，有两种处理方式：
  - 按边界对齐 （假定存储字的宽度为32位，按字节编址）
    - 字地址：4的倍数（低两位为0）
    - 半字地址：2的倍数（低位为0）
    - 字节地址：任意
  - 不按边界对齐



坏处：可能会增加访存次数！

（学了第四章存储器组织后会更明白！）

Alignment(对齐)

如：int i, short k, double x, char c, short j,……

存储器按字节编址

每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节

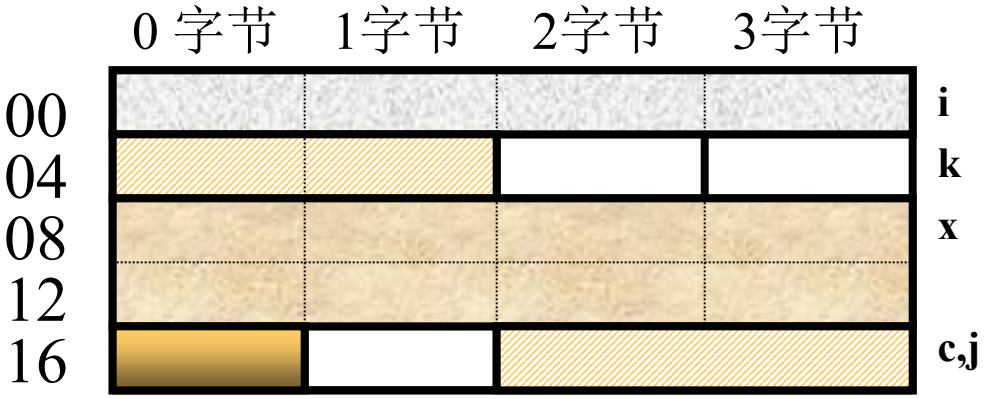
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

按边界对齐

x：2个周期

j：1个周期

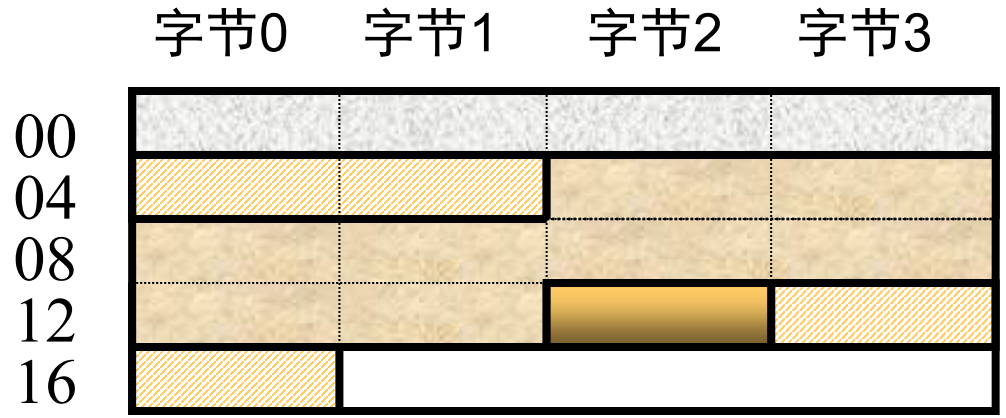


则：&i=0; &k=4; &x=8; &c=16; &j=18;……

边界不对齐

x：3个周期

j：2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

Alignment(对齐) 举例

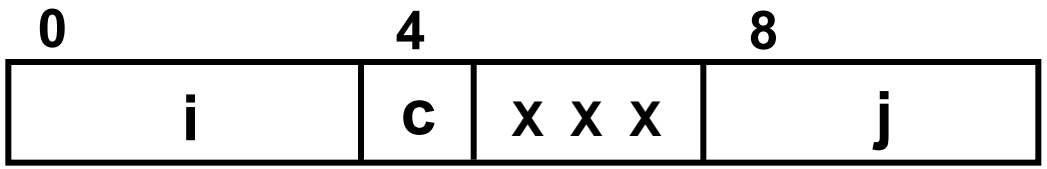
例如，考虑下列两个结构声明：

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

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

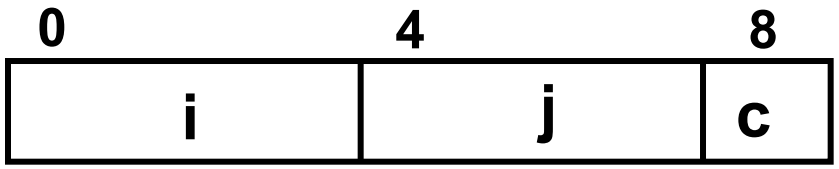
在要求对齐的情况下，哪种结构声明更好？ S2比S1好

S1：



需要12个字节

S2：



只需要9个字节

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

S2：

也需要12个字节

第二讲小结

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