

Lecture 4: Data Representation

数据的机器级表示

第4讲 数值数据的表示

数值数据的表示

主要内容

- ◆ 定点数的表示
 - 进位计数制
 - 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

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

◆ 真值和机器数

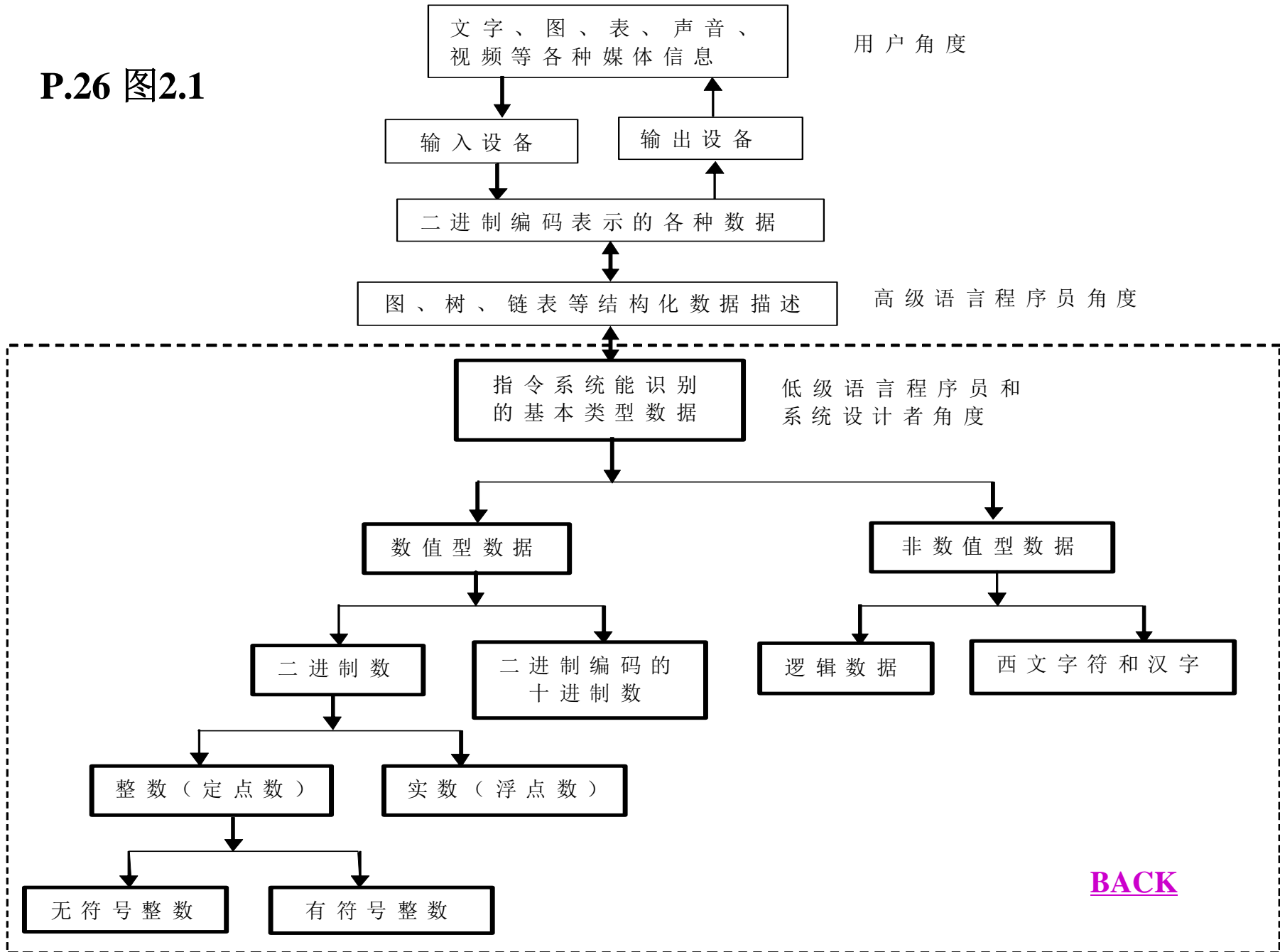
首先考虑数值数据的表示

- 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



C语言中哪些类型是数值数据？哪些是非...？

P.26 图2.1



BACK

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）

Sign and Magnitude (原码的表示)

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

- ◆ 容易理解，但是：
 - ✓ 0 的表示不唯一，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，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

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。

$$\text{同样有 } -3 \equiv 9 \pmod{12}$$

$$-5 \equiv 7 \pmod{12} \text{ 等}$$

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)：+ and - 的统一

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？

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

$$= 9828 + 8072$$

$$= \boxed{1}7900$$

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

取模的含义就是只留余数，高位的“1”被丢弃！相当于只有低4位留在算盘上。

运算器是一个模运算系统，适合用补码表示和运算

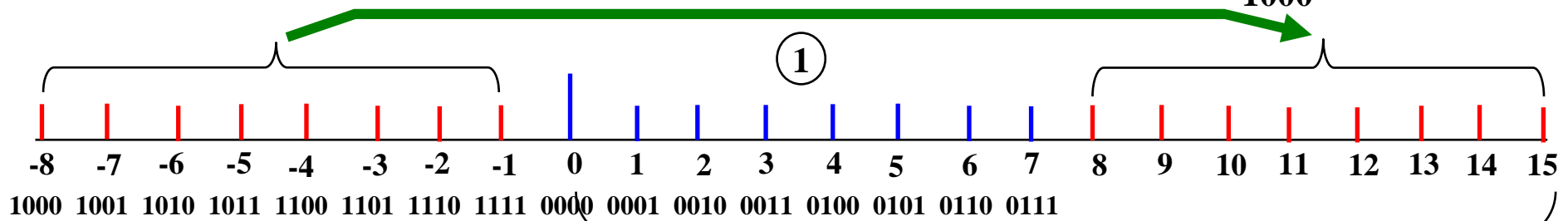
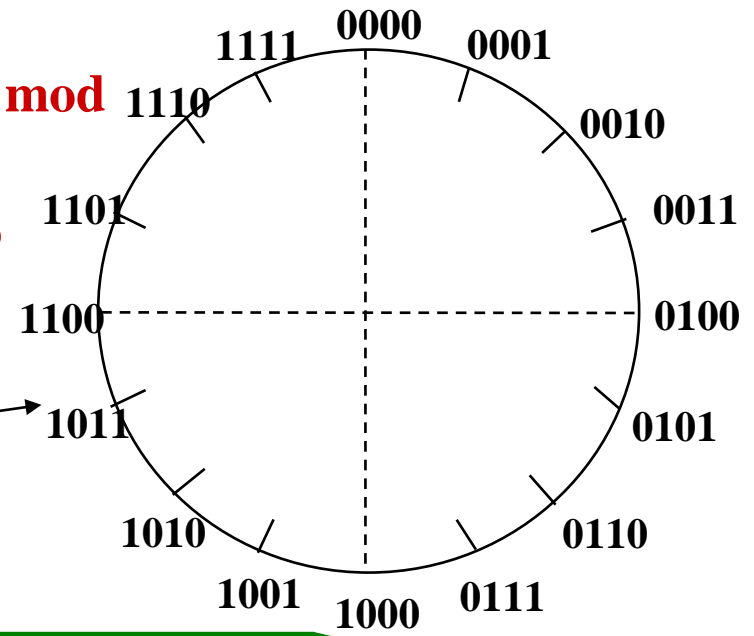
计算机中运算器只有有限位。假定为n位，则运算结果只能保留低n位，故可看成是个只有n档的二进制算盘。所以，其模为 2^n 。

补码的定义 假定补码有n位，则：

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

定点小数： $[X]_{\text{补}} = 2 + X \quad (-1 \leq X < 1, \text{ mod } 2)$

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。
真值范围为：-8 ~ +7



[-8,-1] is shifted to [8,15]. The modul here is 10000

2

求特殊数的补码

假定机器数有n位

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

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

$$\textcircled{3} [-1.0]_{\text{补}} = 2 - 1.0 = \mathbf{1.00\dots0} \quad (\mathbf{n-1} \text{个} \mathbf{0}) \quad (\mathbf{mod} \ 2)$$

$$\textcircled{4} [+0]_{\text{补}} = [-0]_{\text{补}} = \mathbf{00\dots0} \quad (\mathbf{n} \text{个} \mathbf{0})$$

补码与真值之间的简便转换

例: 设机器数有8位, 求123和-123的补码表示。

如何快速得到123的二进制表示?

解: $123 = 127 - 4 = 01111111\text{B} - 100\text{B} = 01111011\text{B}$

$-123 = -01111011\text{B}$

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

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**”

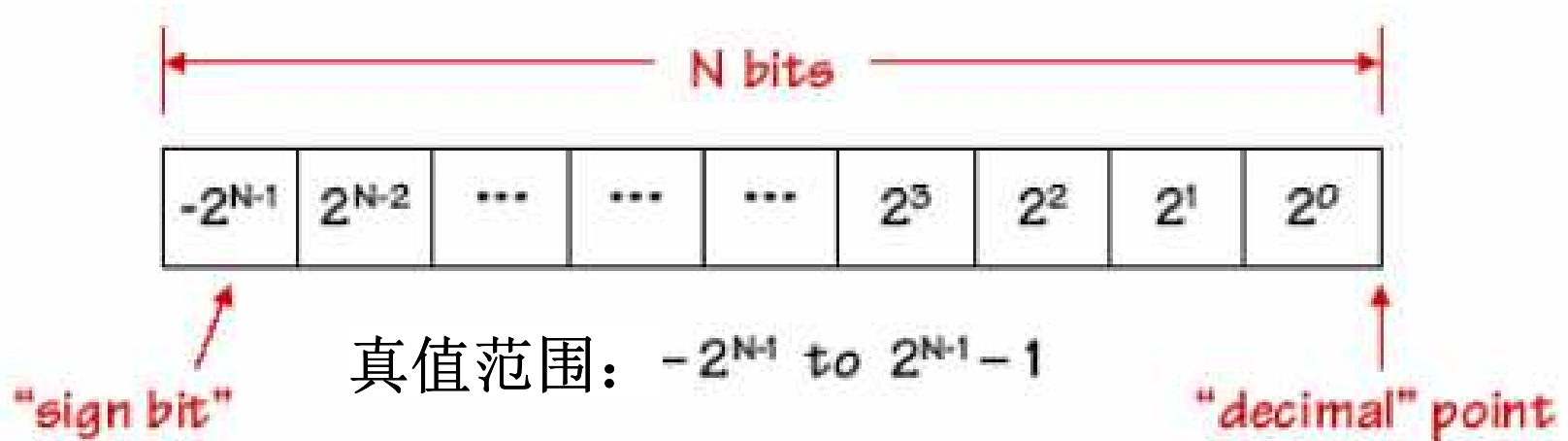
变形 (模4) 补码：双符号，用于存放可溢出的中间结果。

	Decimal	补码	变形补码	Decimal	Bitwise Inverse	补码	变形补码
+0和-0表示唯一	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。

如何求补码的真值

根据补码各位上的“权”，可以求出一个补码的值



8-bit 2's complement example:

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

当 $N=4$ 时，范围为: $-2^3 \sim 2^3 - 1$ (即: $-8 \sim +7$)

当 $N=32$ 时，范围为: $-2^{31} \sim 2^{31} - 1$

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

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

Excess (biased) notation- 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}

$$\text{Ex. } n=4: E_{\text{biased}} = E + 2^3 \quad (\text{bias} = 2^3 = 1000_2)$$

$$-8 (+8) \sim 0000_2$$

$$-7 (+8) \sim 0001_2$$

...

$$0 (+8) \sim 1000_2$$

...

$$+7 (+8) \sim 1111_2$$

0的移码表示惟一

移码和补码仅第一位不同

移码主要用来表示
浮点数阶码！

◦ 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

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

补码： $1111 < 0011$?
(-1) (3)

简化比较

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

移码： $0011 < 0111$
(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： $0\dots01011_2$ ）

- 高到低位从左到右：0000 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 0000

← MSB

- MIPS采用高到低从左往右排列

- 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

- ◆ 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
- ◆ 无符号数的编码中没有符号位
- ◆ 在字长相同的情况下，它的表示范围大于有符号数
- ◆ 无符号数总是整数，所以很多时候就简称为“无符号数”
- ◆ 最大8位无符号整数是11111111B，其值为255

Signed integer（带符号整数）

- ◆ 计算机必须能处理正数(positive) 和负数(negative), MSB表示数符
- ◆ 有三种表示方式
 - Signed magnitude（原码）
用来表示浮点（实）数的尾数
 - One's complement（反码）
现已不用
 - 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的表示惟一，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

- 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”

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

若同时有无符号数和带符号整数，则C编译器隐含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
<code>0 == 0U</code>			
<code>-1 < 0</code>			
<code>-1 < 0U</code>			
<code>2147483647 > -2147483647-1</code>			
<code>2147483647U > -2147483647-1</code>			
<code>2147483647 > (int) 2147483648U</code>			
<code>-1 > -2</code>			
<code>(unsigned) -1 > -2</code>			

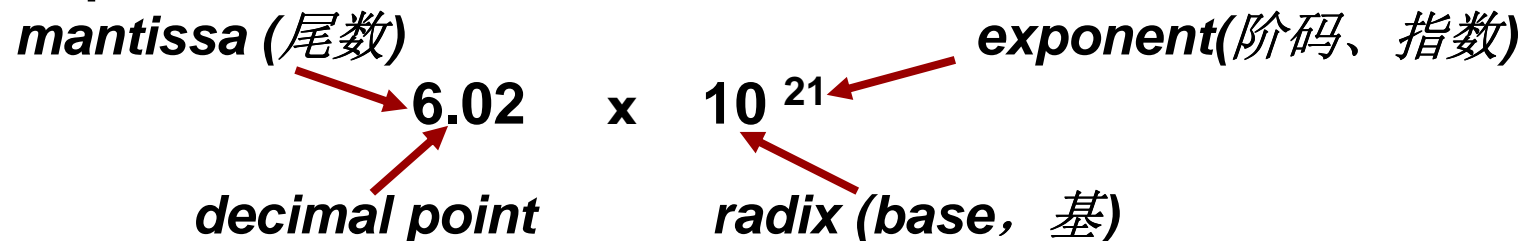
C语言程序中的整数

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！

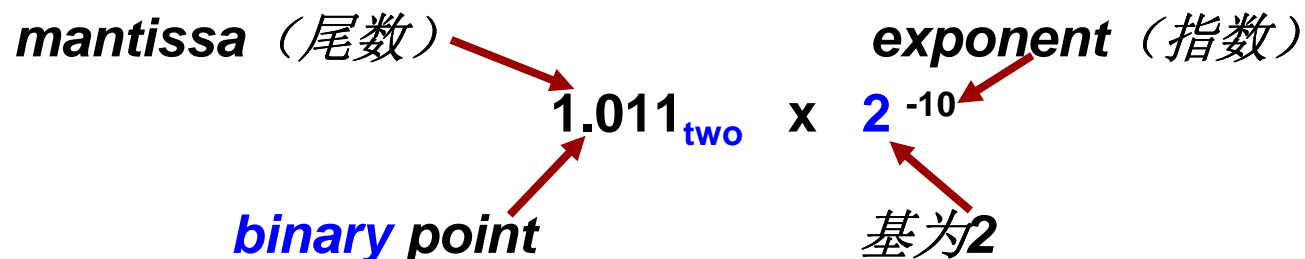
科学计数法(Scientific Notation)与浮点数

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): 1.0×10^{-9}
 - Unnormalized (非规格化形式不唯一) : 0.1×10^{-8} , 10.0×10^{-10}

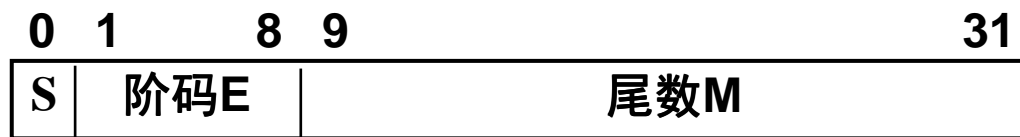
for Binary Numbers:



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

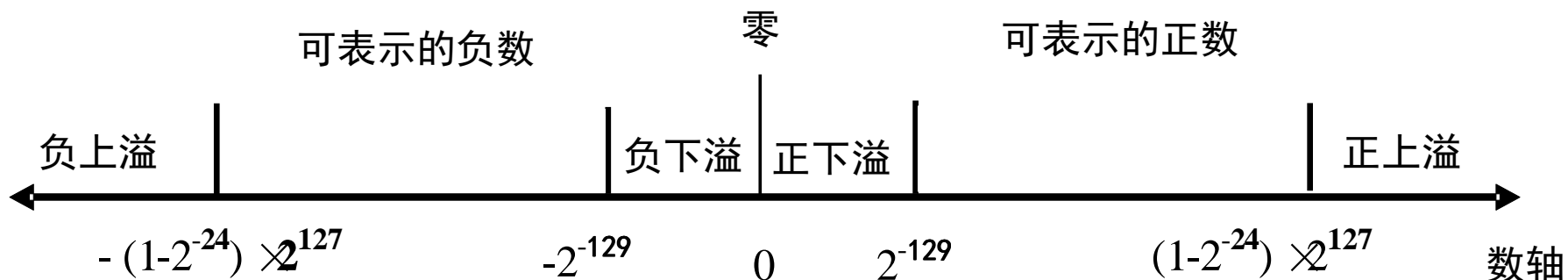
例：画出下述32位浮点数字格式的表数范围。



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： $0.11\dots1 \times 2^{11\dots1} = (1-2^{-24}) \times 2^{127}$ 最小正数： $0.10\dots0 \times 2^{00\dots0} = (1/2) \times 2^{-128}$

因为原码是对称的，所以其表示范围是关于原点对称的。



机器0：阶码为0 或 落在下溢区中的数

浮点数范围比定点数大，但数的个数没变多，故数之间更稀疏，且不均匀

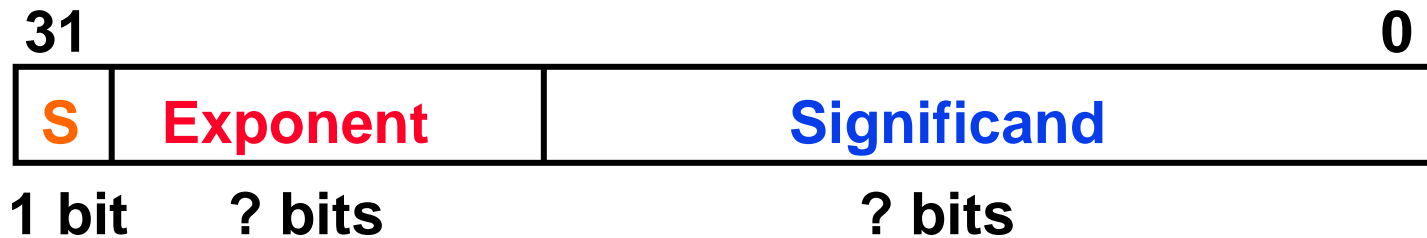
浮点数的表示

- **Normal format**（规格化数形式）：

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

小数点前面总是“1”，故可隐含表示

- **32-bit** 规格化数：



S 是符号位（**Sign**）

Exponent用 excess (or biased) notation (移码/增码)来表示

Significand 表示 **XXXXXXXXXXXX**，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IEEE754的制定

现在所有计算机都采用IEEE754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 Floating Point Standard

Single Precision : (Double Precision is similar)

S	Exponent	Significand
1 bit	8 bits	23 bits

- **Sign bit:** 1 表示negative ; 0表示 positive
- **Exponent (阶码 / 指数) :** 全0和全1用来表示特殊值!
 - SP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: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

0000 0001 (-127)

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

~ 1111 1110 (126)

Ex: Converting Binary FP to Decimal

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

10111 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:**
$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$
- **Represents:** $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$ (= -4.375×10^{-1})

Ex: Converting Decimal to FP

$$-1.275 \times 10^1$$

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 000000000000000000000000

-0: 1 00000000 000000000000000000000000

Representation for $+\infty/-\infty$

∞ : infinity

In FP, 除数为0的结果是 $\pm\infty$, 不是溢出异常.

为什么要这样处理?

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

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

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

$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000

Operations

$$5 / 0 = +\infty, \quad -5 / 0 = -\infty$$

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

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

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

op (NaN,x) = NaN

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

etc.

$0/0 = \text{NaN}$


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

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

Representation for Denorms(非规格化数)

What have we defined so far? (for SP)

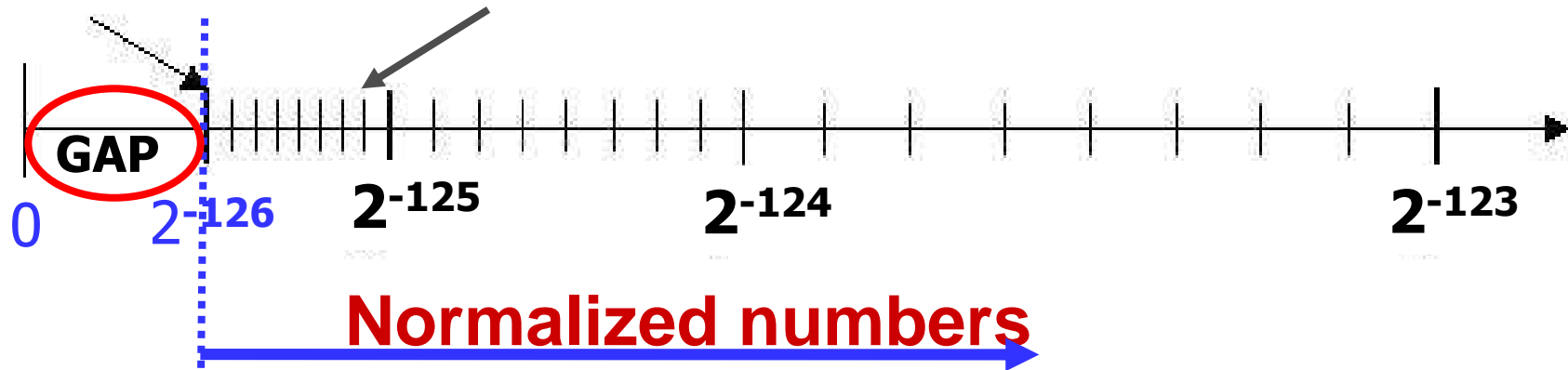
Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN



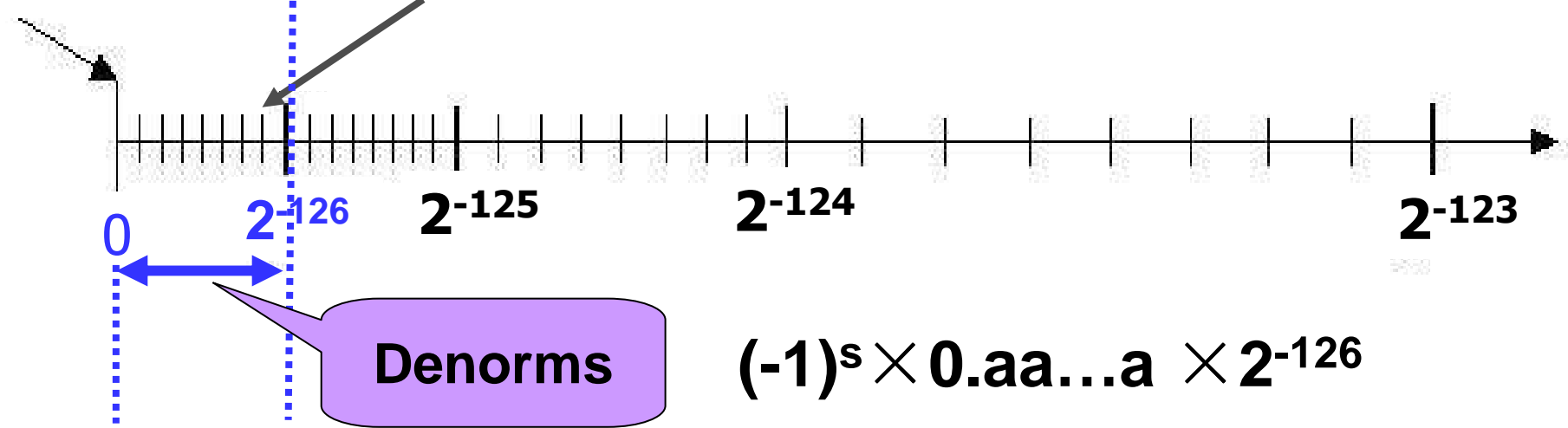
Used to represent
Denormalized
numbers

Representation for Denorms

$$1.0\dots0 \times 2^{-126} \sim 1.1\dots1 \times 2^{-126}$$



$$0.0\dots0 \times 2^{-126} \sim 0.1\dots1 \times 2^{-126}$$



$$(-1)^s \times 0.aa\dots a \times 2^{-126}$$

Questions about IEEE 754

- ◆ What's the range of representable values?

The largest number for single: $+1.11\dots1 \times 2^{127}$
约 $+3.4 \times 10^{38}$

How about double? 约 $+1.8 \times 10^{308}$

- ◆ What about following type converting: not always true!

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

How about
double?

True!

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

How about
double?

Not always
true!

- ◆ How about FP add associative? FALSE!

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

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

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