第2章 信息的表示和处理II：浮点数

教师：吴锐
计算机科学与技术学院
哈尔滨工业大学

主要内容

- 二进制小数
- IEEE 浮点数标准: IEEE 754
- 舍入模式
- 浮点数运算
- C语言的浮点数

推荐阅读:Ch2.4

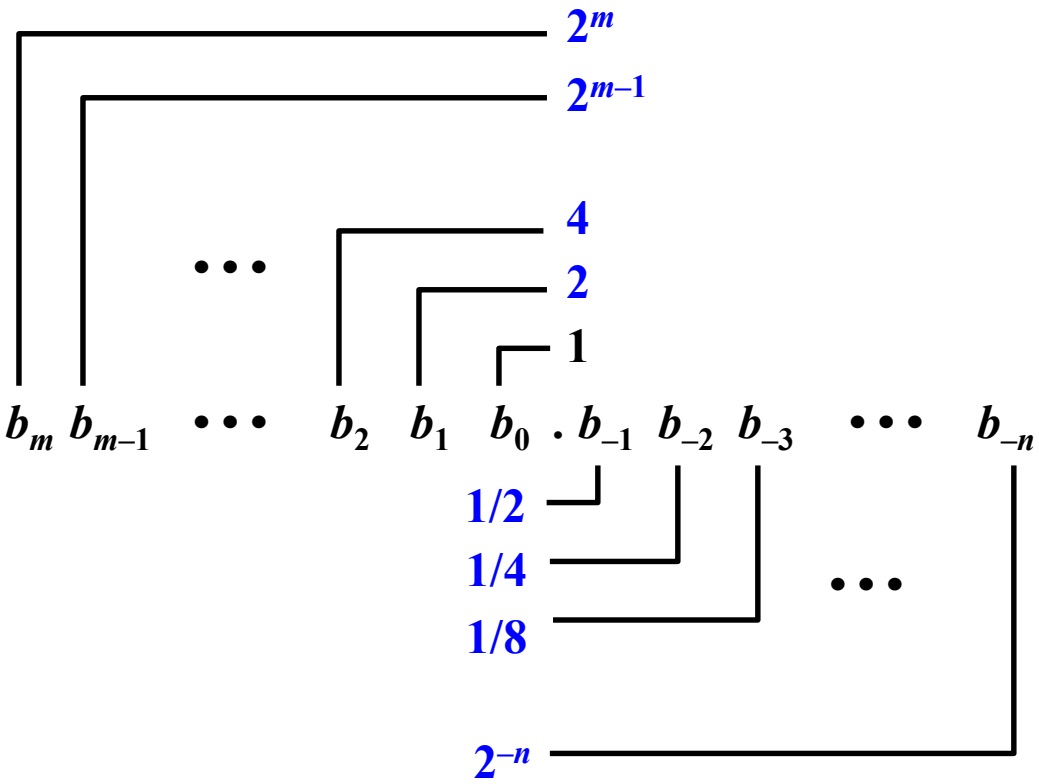
有理数编码

- **浮点表示很有用**
  - 对形如 $V = x \times 2^{y}$ 的有理数进行编码
  - 非常大的数($|V| \gg 0$)或非常接近0的数($|V| \ll 1$)
  - 实数的近似值

- **从程序员角度看**
  - 无趣
  - 晦涩难懂

二进制小数

- “小数点”右边的位代表小数部分



- 表示的有理数： $\sum_{i=-n}^{m} b_i \times 2^i$

二进制小数: 例子

数值	二进制小数
5 ¾	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

- **观察**
  - 除以2 ➔右移 (无符号数)
  - 乘以2 ➔左移
  - $0.111111\ldots_2$
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - 是最接近1.0的小数
    - 表示为$1.0 - \varepsilon$

二进制数的问题

- **局限性 1——近似表示**
  - 只能精确表示形如 $x/2^k$的数值
  - 其他有理数的二进制表示存在重复段
    - 数值　　二进制表示
      - 1/3　　0.0101010101[01]…2
      - 1/5　　0.001100110011[0011]…2
      - 1/10　　0.0001100110011[0011]…2

二进制数的问题

- 在计算机内的实现问题
  - 长度有限的 w位
  - 只能在*w*位内设置一个二进制小数点
  - 限制了数的范围(非常小？非常大?)

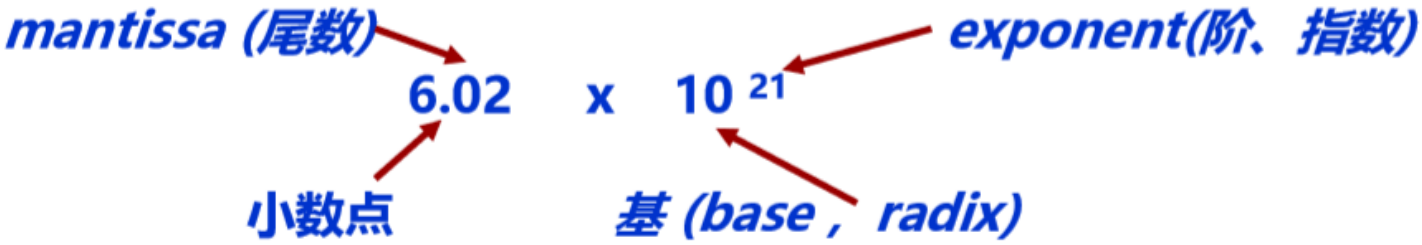
- **定点数**
  - 小数点隐含在*w*位编码的某一个固定位置上
    - 例如MSB做符号位，隐含后面是小数点，表示小于1.0的纯小数
    - 123.456怎么办？？？

浮点数

- 二进制小数
- **IEEE 浮点数标准: IEEE 754**
- 浮点数示例
- 舍入、加法与乘法
- C语言的浮点数
- 小结

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}

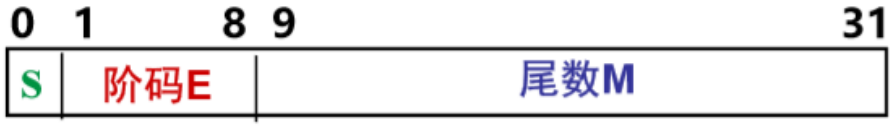
不唯一

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。



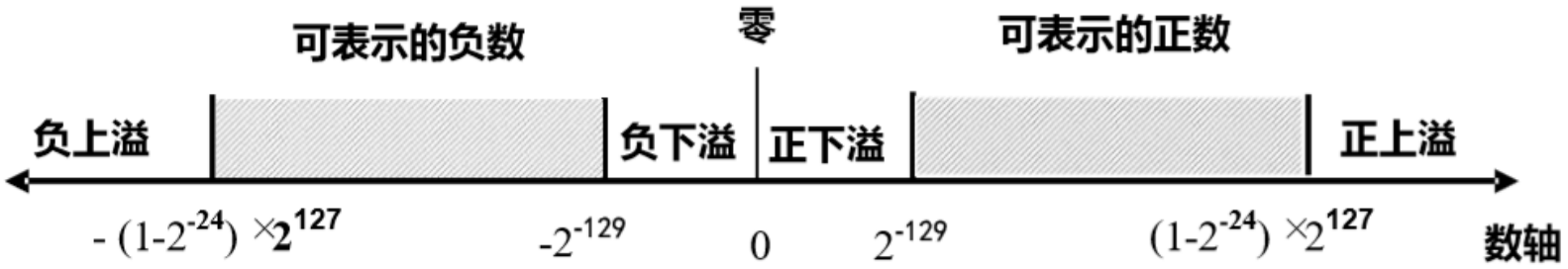
$+/-0.1xxxxx * 2^{E}$

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

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

因为原码对称，故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

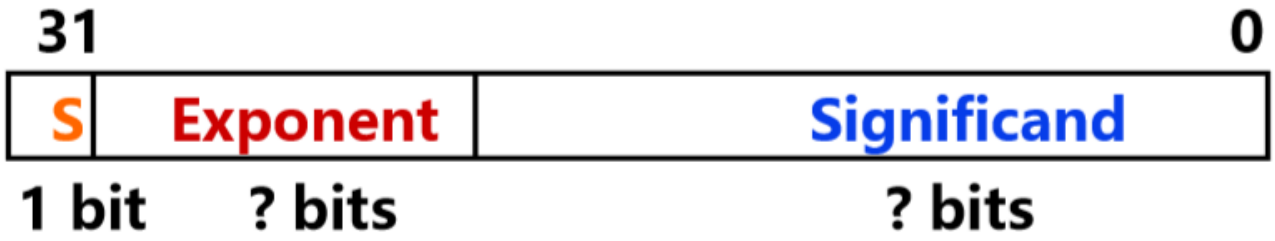
浮点数的表示

- Normal format（规格化数形式）：

  +/-1.xxxxxxxxxx × $R^{Exponent}$

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！

- 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（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？

IEEE 浮点数

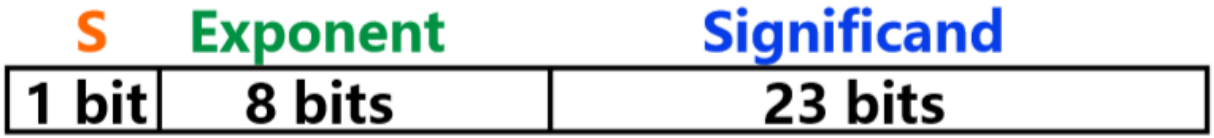
- **IEEE 标准 754**
  - William Kahan 从1976年开始为Intel 设计(1989获图灵奖)
  - 1985年成为浮点运算的统一标准，快速, 易于实现、精度损失小
  - 优雅、易理解
  - 所有主流的CPU都支持
  - 之前有很多不同格式、不太关注精确性

IEEE 754标准

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

规定：小数点前总是“1”，故可隐含表示。

Single Precision（单精度）：



- 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: $(-1)^S$ x (1 + Significand) x $2^{(Exponent-127)}$

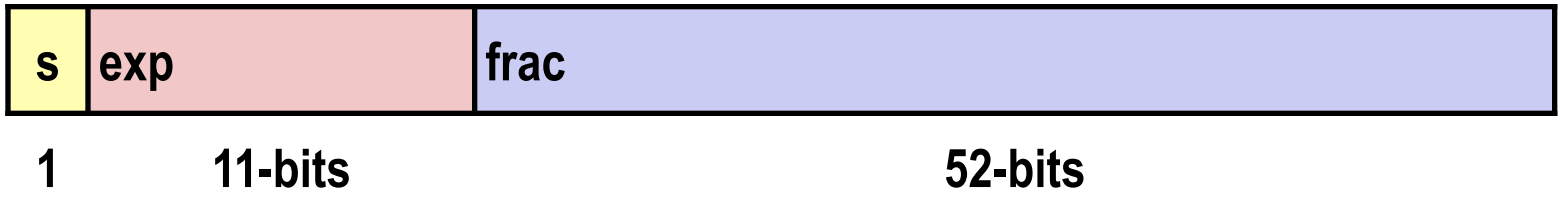
DP: $(-1)^S$ x (1 + Significand) x $2^{(Exponent-1023)}$

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

精度选项

- 单精度: 32 bits

- 双精度: 64 bits



- 扩展精度: 80 bits (Intel)

规格化数

$v = (-1)^s M 2^E$

- **条件: exp ≠ 000…0 且 exp ≠ 111…1**

- **阶码(Exponent) 采用偏置值编码: *E* = *Exp* – *Bias***
  - *Exp*: exp 字段的无符号数值
  - 偏置*Bias* = $2^{k-1} - 1$，*k* 为阶码的位数
    - 单精度: 127 (Exp: 1…254, E: -126…127)
    - 双精度: 1023 (Exp: 1…2046, E: -1022…1023)

- **尾数(Significand) 编码隐含先导数值1: *M* = 1.xxx…x_2**
  - xxx…x: 是 frac字段的数码
  - frac=000…0 (M = 1.0)时，为最小值
  - frac=111…1 (M = 2.0 – ε)时，为最大值
  - 额外增加了一位的精度（隐含值1）

规格化编码示例

$v = (-1)^s M 2^E$
$E = Exp - Bias$

- 数值: `float F = 15213.0`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- 尾数(Significand)
  - $M = 1.\underline{1101101101101}_2$
  - $frac = \underline{1101101101101}0000000000_2$
- 阶码(Exponent)
  - $E = 13$
  - $Bias = 127$
  - $Exp = 140 = 10001100_2$
- 编码结果:

0	10001100	11011011011010000000000
s	exp	frac

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 11101	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
- 尾数数值部分:

$$1 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 0\times2^{-4} + 0\times2^{-5} + \ldots$$
$$= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$$

- 真值: $-1.75\times2^{-2} = -0.4375$

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

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=-1100.11B

=-1.10011B x 2^3　　阶（指数）为3

因此，符号 S=1

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

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	?

非规格化数

$$v = (-1)^s M 2^E$$
$$E = 1 - Bias$$

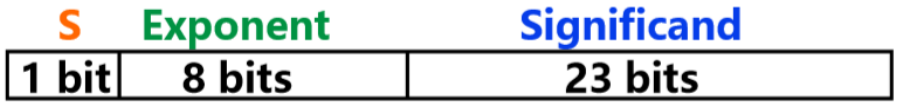
- 条件: exp = 000…0
  - 阶码(Exponent) 值: ***E*** = 1 – Bias (instead of ***E*** = 0 – ***Bias***)
  - 尾数(Significand)编码隐含先导数值0: ***M*** = 0.xxx…x_2
    - **xxx…x**:是 frac字段的数码

- 情况1: **exp** = 000…0, **frac** = 000…0
  - 表示值0
  - 注意有不同的数值 +0 和 –0 (why?)

+0: 0 00000000 00000000000000000000000

-0: 1 00000000 00000000000000000000000

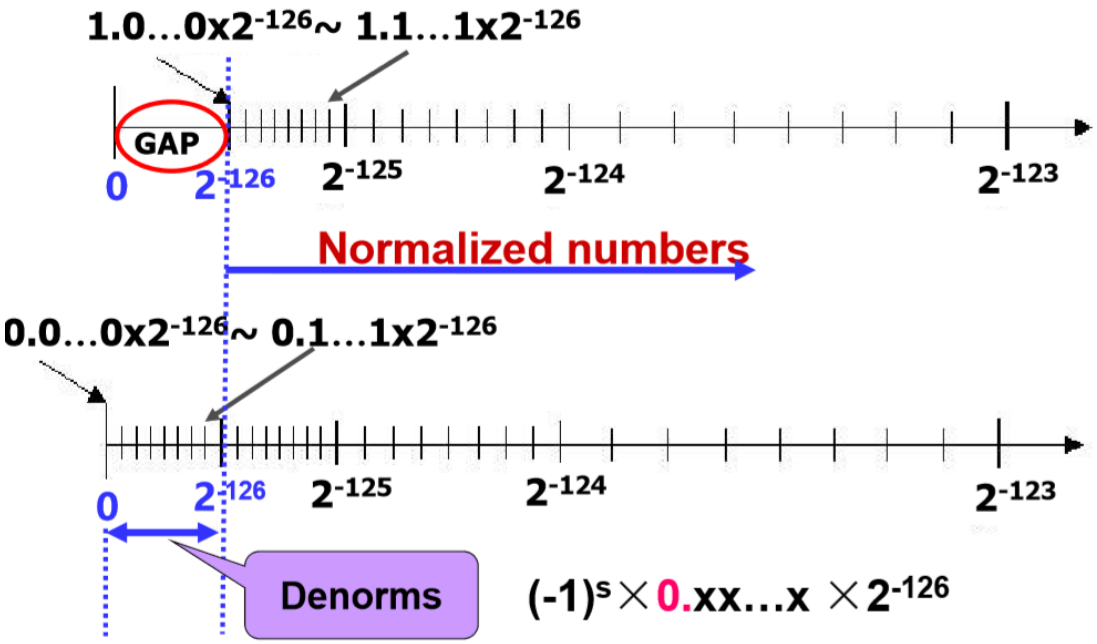
Single Precision（单精度）：



非规格化数

$$v = (-1)^s M 2^E$$
$$E = 1 - Bias$$

- 条件: exp = 000…0
  - 阶码(Exponent) 值: ***E*** = 1 – Bias (instead of ***E*** = 0 – ***Bias***)
  - 尾数(Significand)编码隐含先导数值0: ***M*** = 0.xxx…x_2
    - **xxx…x**:是 frac字段的数码
- 情况2: **exp** = 000…0, **frac** ≠ 000…0
  - 最接近0.0的那些数
  - 间隔均匀



特殊值

- 条件: exp = 111…1
- 情况3: exp = 111…1, frac = 000…0
  - 表示 无穷(infinity) ∞
  - 溢出的运算
  - 正无穷、负无穷

浮点数除0的结果是 +/- ∞, 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+∞/-∞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

相关操作：

5.0 / 0 = +∞, -5.0 / 0 = -∞

5+(+∞) = +∞, (+∞)+(+∞) = +∞

5 - (+∞) = -∞, (-∞) - (+∞) = -∞ etc

特殊值

- 条件: **exp = 111…1**
- 情况4: **exp = 111…1, frac ≠ 000…0**
  - 表示：不是一个数Not-a-Number (NaN)
  - 表示没有数值结果（非实数或无穷），例如：
    sqrt(–1), ∞ − ∞, ∞ × 0

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

– 称为 Not a Number (NaN) - “非数”

How to represent NaN

Exponent = 255

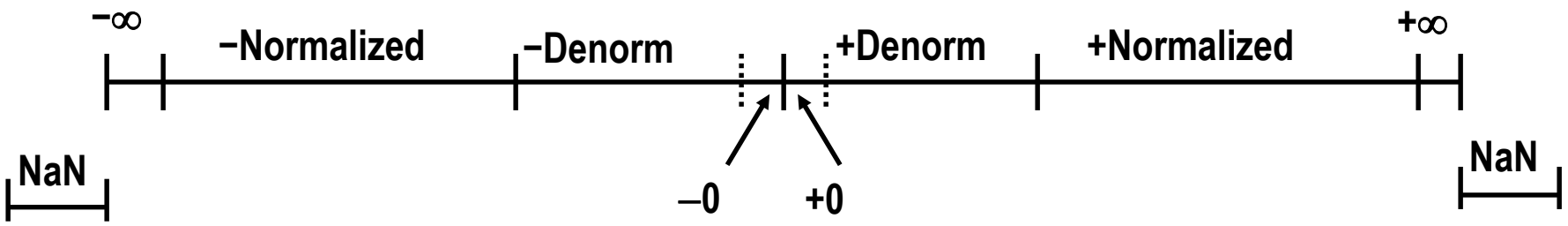
Significand: nonzero

NaNs 可以帮助调试程序

相关操作：

sqrt (-4.0) = NaN	0/0 = NaN
op (NaN,x) = NaN	+∞+(-∞) = NaN
+∞- (+∞) = NaN	∞/∞ = NaN
etc.	

浮点编码总结



浮点数的例子

```
int main()
{
    float f;

    for(;;)
    {
        printf("请输入一个浮点数:");
        scanf("%f",&f);
        printf("这个浮点数的值是:%f\n",f);
        if(f==0) break;
    }
    return 0;
}
```

运行结果:

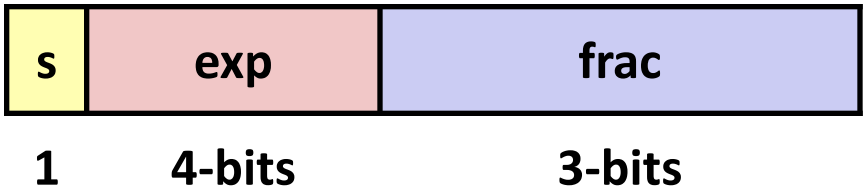
```
Please enter a number: 61.419997
61.419998
Please enter a number: 61.419998
61.419998
Please enter a number: 61.419999
61.419998
Please enter a number: 61.42
61.419998
Please enter a number: 61.420001
61.420002
Please enter a number:
```

61.419998和61.420002是两个可表示数，两者之间相差0.000004。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。

浮点数

- 二进制小数
- IEEE 浮点数标准: IEEE 754
- **浮点数示例**
- 舍入、加法与乘法
- C语言的浮点数
- 小结

小浮点数例子——1字节浮点数



- 8位浮点编码
  - 符号位：最高有效位
  - 阶码(Exponent)4位，偏置为7
  - 小数(`frac`) 3位

- 和IEEE 相同的格式
  - 规格化、非规格化
  - 0、NaN、无穷的表示

动态范围(仅正数)

$v = (-1)^s M 2^E$

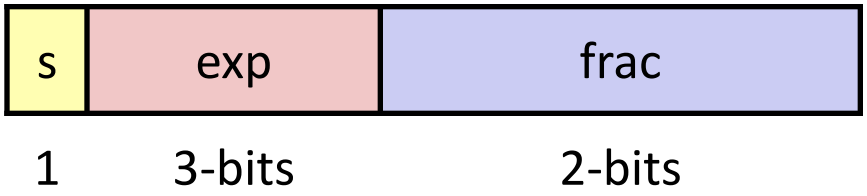
n: E = Exp – Bias

d: E = 1 – Bias

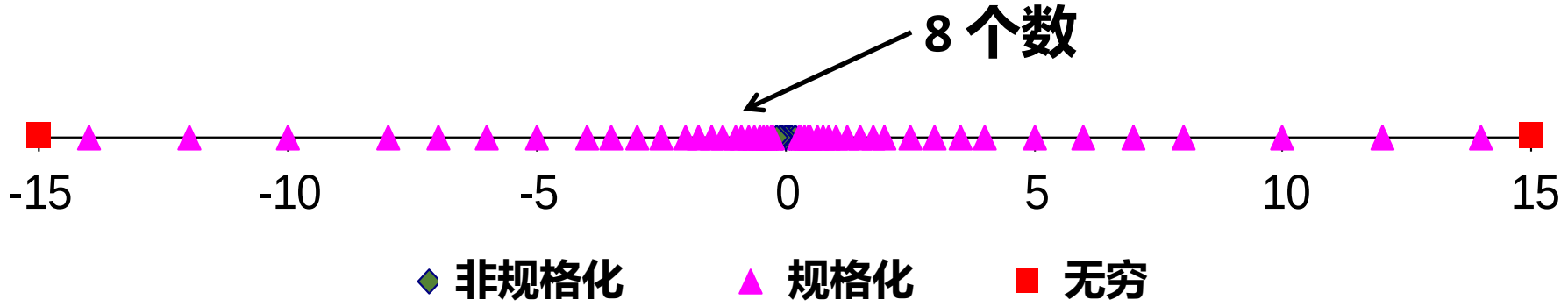
	s exp frac	E	Value	
非规格化数	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	最接近0
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	最大非规格化数
规格化数	0 0001 000	-6	8/8*1/64 = 8/512	最小规格化数
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	最大规格化数
	0 1111 000	n/a	inf	

数值分布

- **6-bit类 IEEE格式浮点数**
  - e： 阶码(Exponent) 位数3
  - f： 小数位数 2
  - 偏置bias= $2^{3-1}-1 = 3$

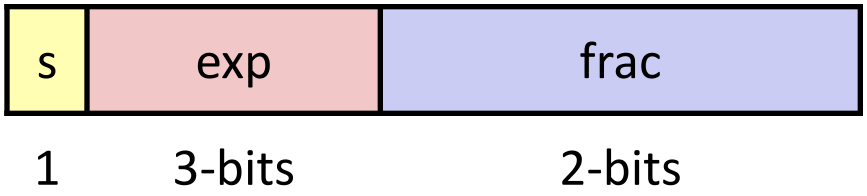


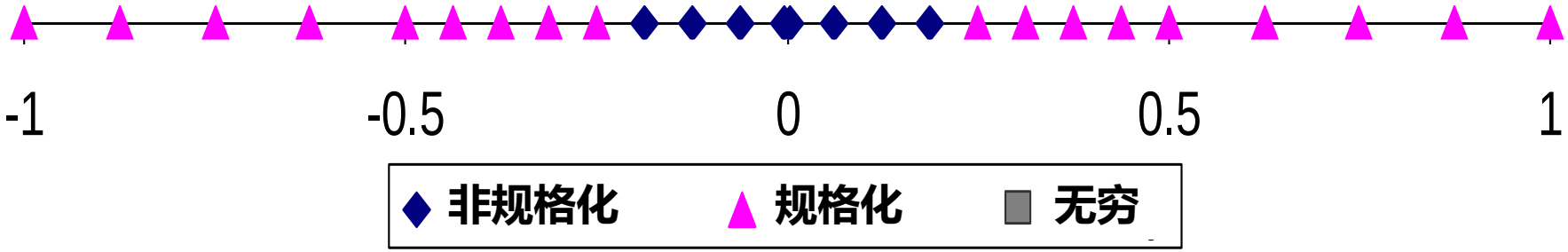
- **注意：数值在趋近于0时变密集**



数值分布(放大观察)

- **6-bit类 IEEE格式**
  - e： 阶码(Exponent) 位数3
  - f： 小数位数 2
  - 偏置bias= $2^{3-1}-1 = 3$





浮点数

- 二进制小数
- IEEE 浮点数标准: IEEE 754
- 浮点数示例
- **舍入、加法与乘法**
- C语言的浮点数
- 小结

浮点数运算: 基本思想

- $x +_f y = Round(x + y)$
- $x \times_f y = Round(x \times y)$
- **基本思想**
  - 首先，计算精确结果
  - 然后，变换到指定格式
    - 可能溢出：阶码(Exponent) 太大
    - 小数部分可能需要舍入

舍入

- 舍入模式(以美元舍入说明)

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
向0舍入	$1	$1	$1	$2	–$1
向下舍入 (−∞)	$1	$1	$1	$2	–$2
向上 (+∞)	$2	$2	$2	$3	–$1
向偶数舍入(默认)	$1	$2	$2	$2	–$2

细究“向偶数舍入”

- **默认的舍入模式**
  - 很难找到更好的方法
  - 其他方法都有统计偏差
    - 对正整数集合求和时，和将始终被低估或高估（负偏差、正偏差）
- **向偶数舍入**
  - 当恰好在两个可能的数值正中间时（中间值）：
    舍入后，最低有效位的数码为偶数
  - 其他时候：向最近的数值舍入
    - 比中间值小向下舍入，比中间值大向上舍入
- **以10进制数向最近的百分位舍入为例：**

7.8949999	7.89	(比中间值小：向下舍入)
7.8950001	7.90	(比中间值大：向上舍入)
7.8950000	7.90	(中间值—向上舍入)
7.8850000	7.88	(中间值—向下舍入)

二进制数的舍入

- **二进制小数的舍入**
  - “偶数”：最低有效位值为0
  - “中间值”：舍入位置右侧的位都是0，即形如: XXX $\underline{1}00\ldots_2$

- **例子**
  - 舍入到最近的1/4 (小数点右边第2位)

数值	二进制	舍入后	舍入动作	舍入后的值
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

浮点乘法

- $(-1)^{s1}\ M1\ \ 2^{E1}$ x $(-1)^{s2}\ M2\ \ 2^{E2}$
- **精确结果:** $(-1)^{s}\ M\ \ 2^{E}$
  - 符号(Sign) *s*: *s1* ^ *s2*
  - 尾数(Significand) *M*: *M1* x *M2*
  - 阶码(Exponent) *E*: *E1* + *E2*

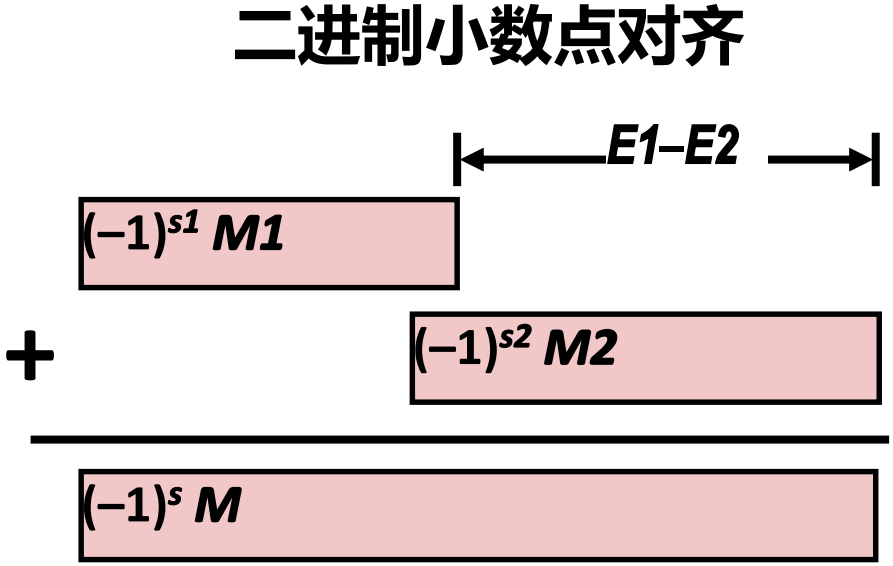
- **修正**
  - 如*M* ≥ 2, 将*M*右移(1位), *E*加*1*
  - 如 *E* 超出范围，则溢出
  - 将*M*舍入，以符合小数部分的精度要求
- **实现**
  - 主要问题：实现尾数(Significand)的乘

浮点数加法

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - 假设 *E1* > *E2*

- **准确结果:** $(-1)^{s}\ M\ \ 2^{E}$
  - 符号 *s*, 尾数*M*:
    - 有符号数对齐、相加的结果
  - 阶码(Exponent) *E*: *E1*

- **修正**
  - *M* ≥ 2：将*M*右移(1位)，*E*加*1*
  - *M* < 1：将*M*左移*k* 位, *E* 减 *k*
  - *E*超范围：溢出
  - 将*M*舍入，以符合小数部分的精度要求



浮点数

- 二进制小数
- IEEE 浮点数标准: IEEE 754
- 浮点数示例
- 舍入、加法与乘法
- **C语言的浮点数**
- 小结

C语言中的浮点数

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 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
- 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

浮点数习题

- 针对下列C表达式:
  - 证明对所有参数值都成立
  - 或什么条件下不成立

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

假定d 和 f都不是NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

浮点数习题答案

- x == (int)(float) x　　No: 24 位尾数
- x == (int)(double) x　　Yes: 53位尾数
- f == (float)(double) f　　Yes: 增加精度
- d == (float) d　　No: 损失精度
- f == -(-f);　　Yes: 仅仅改变符号位
- 2/3 == 2/3.0　　No: 2/3 == 0
- d < 0.0⇒ ((d*2) < 0.0)　　Yes!
- d > f ⇒ -f < -d　　Yes
- d *d >= 0.0　　Yes!
- (d+f)-d == f　　No: 不具备结合性

浮点的悲剧

- 1991年2月25日
- 美国爱国者导弹拦截伊拉克飞毛腿导弹失败
- 后果：飞毛腿导弹炸死28名士兵
- 爱国者导弹的内置时钟计数器N每0.1秒记一次数。
- 时间计算

  T = N×0.1

  程序用24位数来近似表示0.1：

  x=0.0001 1001 1001 1001 1001 100

浮点的悲剧

- 0.1-x = 0.0000000000000000000000[1100][1100]...2
- 0.1-x = 2^{-20} ×0.1 = 9.54×10^{-8} （约等于）
- 程序运行100 小时后，累计的误差：
  100×3600× 10×9.54×10^{-8} =0.34344秒
- 飞毛腿速度：2000 m/s
- 飞毛腿位置的估计误差：686 m

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

爱国者导弹定位误差改进

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于$2^{-24}\times0.1 \approx 5.96\times10^{-9}$。100小时后时钟偏差$5.96\times10^{-9}\times36\times10^{5} \approx$ 0.0215秒。距离偏差$0.0215\times2000\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，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于$2^{-30}\times0.1 \approx 9.31\times10^{-11}$。100小时后时钟偏差 $9.31\times10^{-11}\times36\times10^{5} \approx 0.000335$秒。预测的距离偏差仅为 $0.000335\times2000 \approx 0.67$米。

天价“溢出”

- 代价5亿美元的溢出

天价“溢出”

- 主角：阿丽亚娜5(Ariane5)型火箭的首次发射
- 时间：1996.6.4
- 剧情：发射后仅37秒，偏离路径，解体爆炸
- 代价：5亿美元
- 原因：溢出
  - 溢出——将64位浮点数转换成16位有符号整型数时，发生溢出。这个溢出的整型数，用于描述火箭的水平速度
  - Ariane4的水平速度绝对不会超过16位数的范围，因此用了16位整数
  - Ariane5简单复用了这部分代码
  - 问题：Ariane 5 的水平速度是Ariane 4的5倍！！！

启示

- Ariana 5火箭和爱国者导弹的例子带来的启示

✓ 程序员应对底层机器级数据的表示和运算有深刻理解

✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确

✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数常量与整数变量相乘，然后再通过移位运算来确定小数点

小结

- **IEEE 浮点数 具有清晰的数学性质**
- **表示形如 M x 2^E 的数字**
- **对运算进行推理，而不用考虑其实现**
  - 就像有完美的精度，然后在进行舍入
- **和实数运算不同**
  - 结合性、分配性有冲突

有趣的数字

{single,double}

Description	*exp*	*frac*	*Numeric Value*
0	00…00	00…00	0.0
最小值的后非规格化数	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.4×10^{-45}			
▪ Double ≈ 4.9×10^{-324}			
最大的非规格化数	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
最小的后规格化数	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
▪ 刚刚比最大的非规格化数大			
1	01…11	00…00	1.0
最大的规格化数	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
▪ Single ≈ 3.4×10^{38}			
▪ Double ≈ 1.8×10^{308}			