

# 评价标准

## ■ 平时20%+作业20%+期末60%

- 4次作业，各5%，晚于Date Due提交会失去成绩
- 平时作业与期末，王湘浩实验班与其他班会有一定区别

## ■ 五个宽限日

- 作业或者签到
- 用完后，我们不会接受任何延长时限的请求
- 例如：Date Due 5月13日，如使用两个宽限日，5月15日23:59前提交都可以

# **Part1: 系统中的数据表示和处理 (比特/位 Bits、字节 Bytes, 和整数 Integers)**

人工智能学院:计算机原理与系统结构

Aug. 29, 2025

# 主要内容：比特、字节 和 整型数

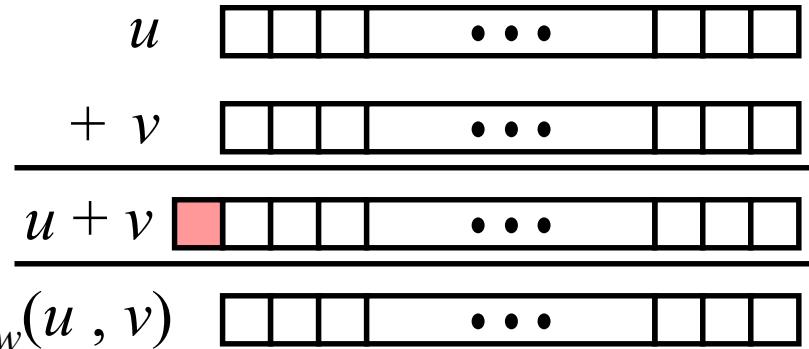
- 以比特表示信息
- 比特运算
- 整型数 (**Integers**)
  - 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算：加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 无符号数加法

操作数:  $w$  bits

相加后真实的位数:  $w+1$  bits

丢弃进位:  $w$  bits



- 标准加法

- 忽略进位

- 当运算溢出时，相当于进行以下模运算：

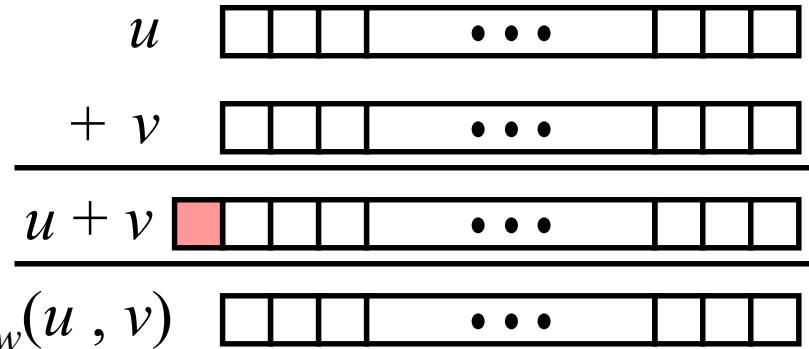
$$s = \text{UAdd}_w(u, v) = (u + v) \mod 2^w$$

# 无符号数加法

操作数:  $w$  bits

相加后真实的位数:  $w+1$  bits

丢弃进位:  $w$  bits



- 标准加法

- 忽略进位

- 当运算溢出时，相当于进行以下模运算：

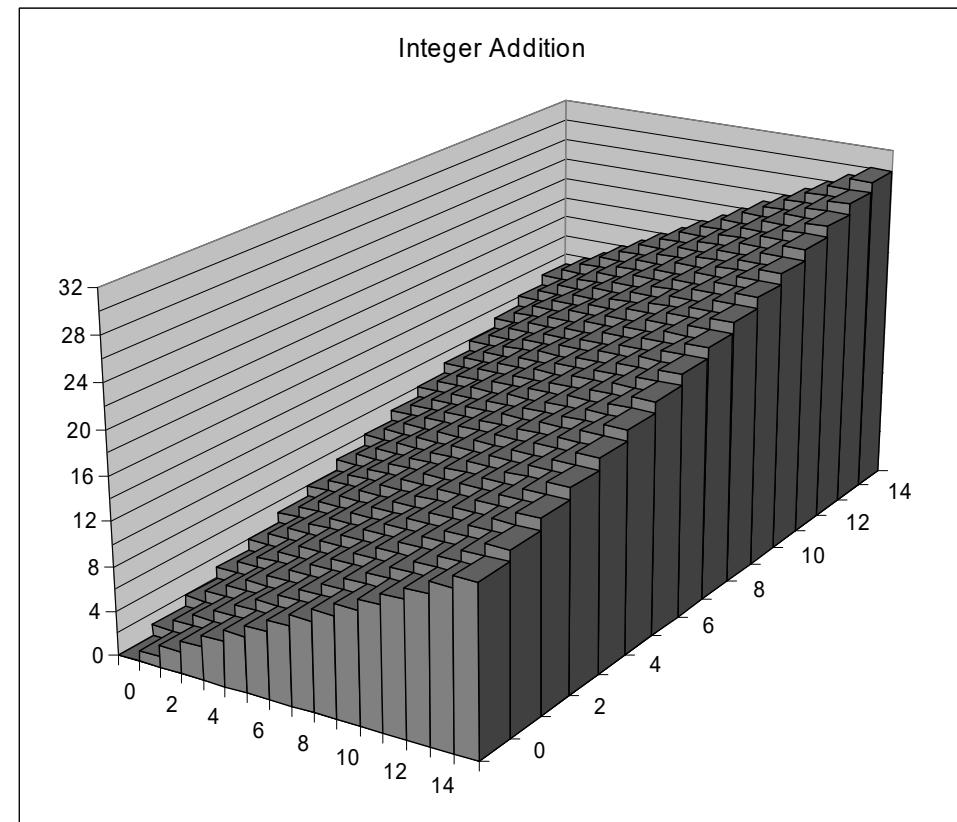
$$s = \text{UAdd}_w(u, v) = (u + v) \mod 2^w$$

# 整数加法可视化

$$\text{Add}_4(u, v)$$

## ■ 整数加法

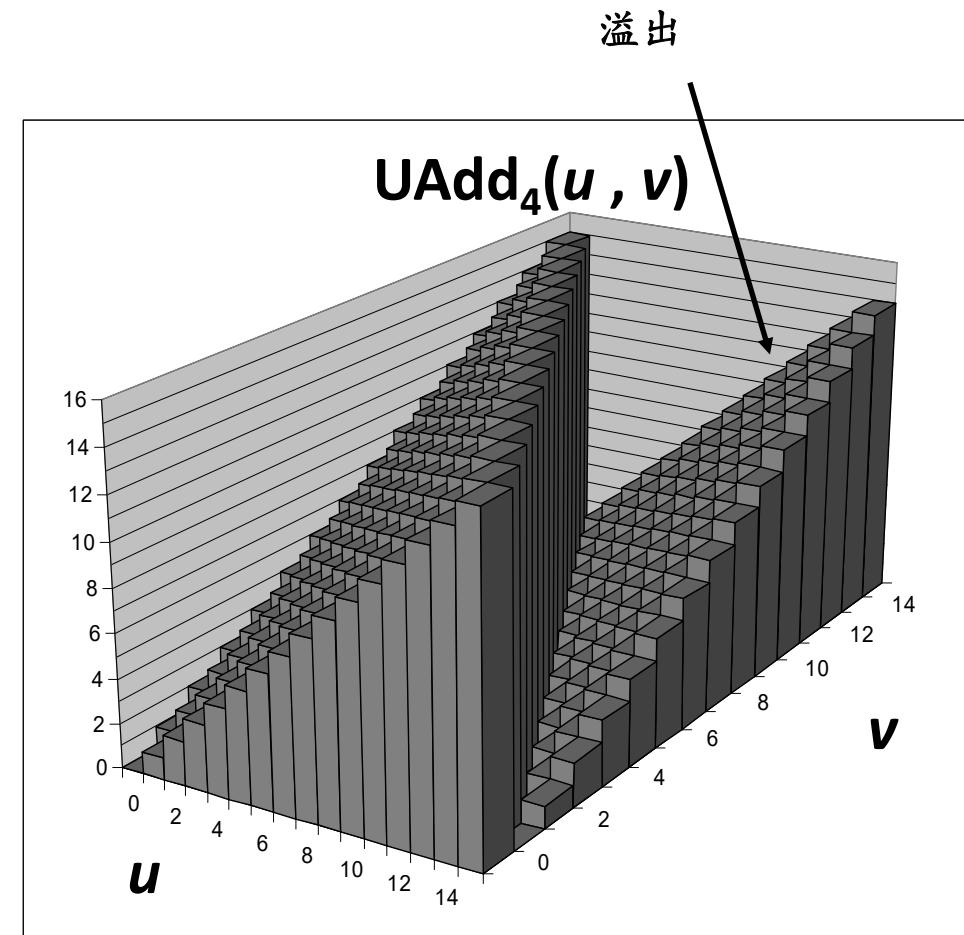
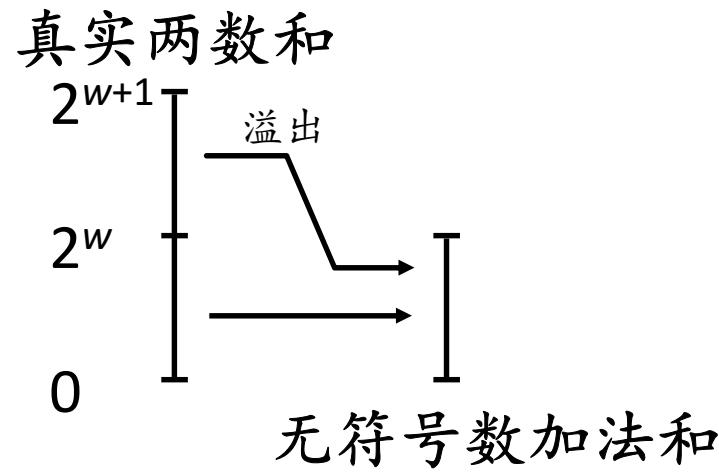
- 4-bit 整数 ( $u, v$ )
- 计算真实的两数和  $\text{Add}_4(u, v)$
- 两数和随着  $u$  和  $v$  的增加而线性增加
- 表面为斜面形



# 无符号数加法可视化

## ■ 无符号数加法溢出

- 如果和  $\geq 2^w$
- 至多溢出一次

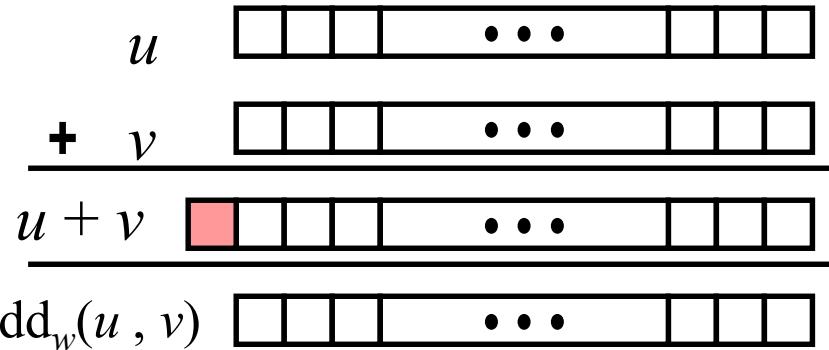


# 补码加法

操作数:  $w$  bits

相加后真实的位数:  $w+1$  bits

丢弃进位:  $w$  bits



## ■ TAdd 和 UAdd 具有完全相同的比特层面表现

- C语言中有符号数(补码)与无符号数加法:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

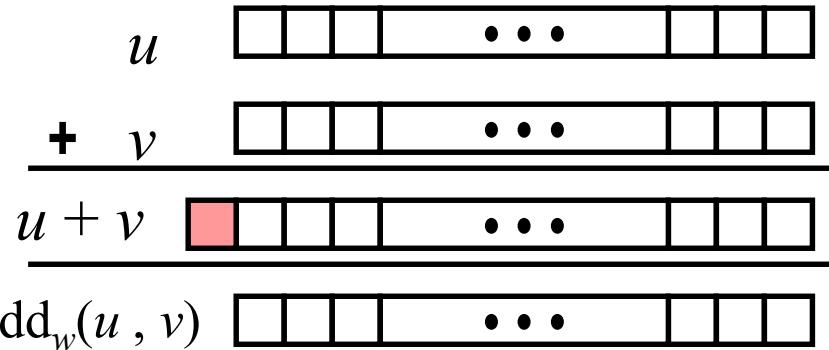
- 将会有:  $s == t$

# 补码加法

操作数:  $w$  bits

相加后真实的位数:  $w+1$  bits

丢弃进位:  $w$  bits



## ■ TAdd 和 UAdd 具有完全相同的比特层面表现

- C语言中有符号数(补码)与无符号数加法:

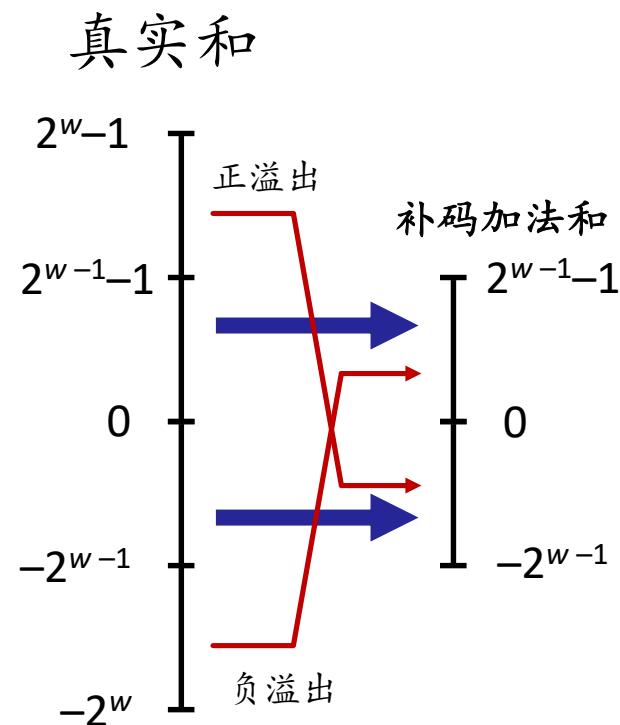
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

- 将会有:  $s == t$

# 补码加法溢出

## ■ 性质

- 真实和需要 $w+1$ 位
- 丢弃最高有效位 (MSB)
- 将剩余的比特/位视作补码



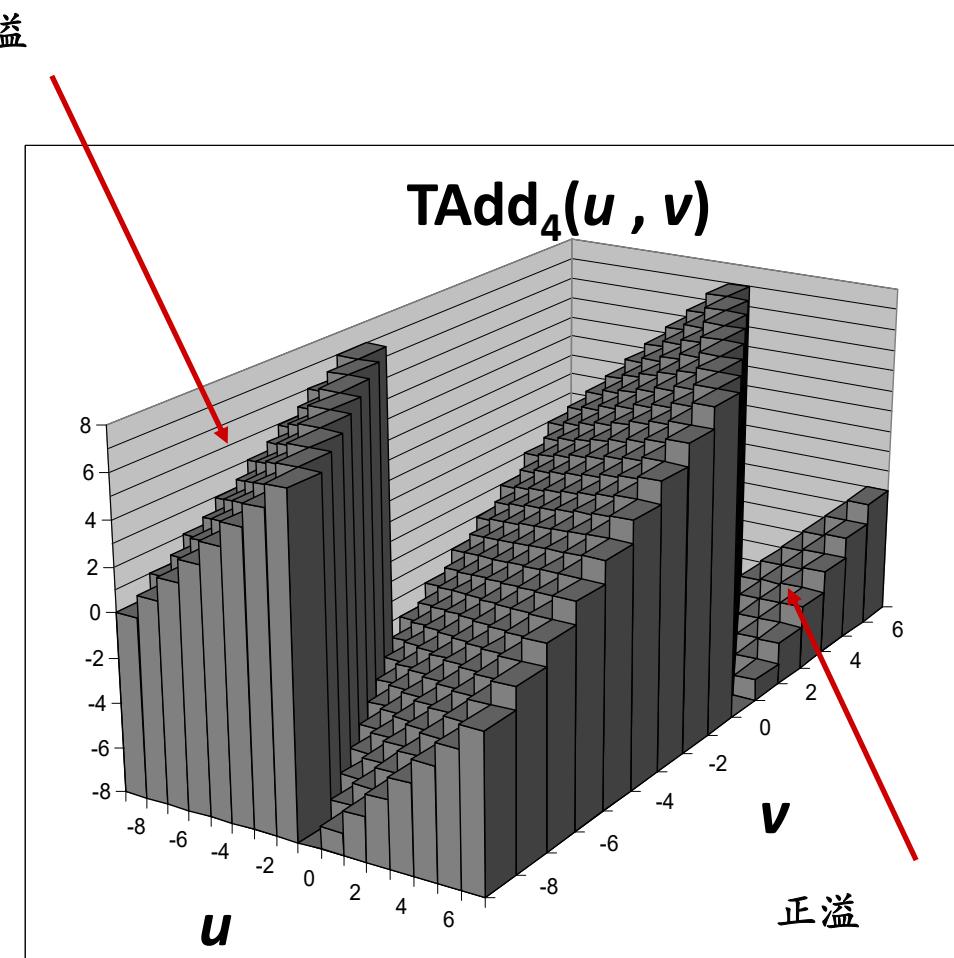
# 补码加法可视化

## ■ Values

- 4位补码
- 数值范围-8到+7

## ■ Wraps Around

- 如果和  $\geq 2^{w-1}$ 
  - 变成负数
  - 最多一次
- 如果和  $< -2^{w-1}$ 
  - 变成负数
  - 最多一次



# 乘法

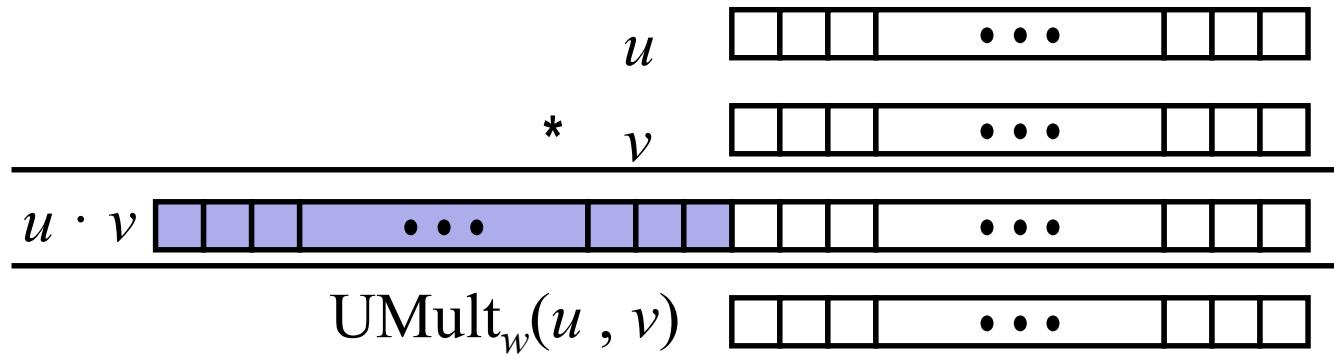
- 目标: 计算  $w$  位的两个数  $x$  和  $y$  的乘积
  - 有符号数或无符号数
- 但是, 乘积的精确结果可能超过  $w$  位
  - 无符号数: 最多  $2w$  位
    - 结果范围:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - 二进制补码最小值 (负数) : 最多  $2w-1$  位
    - 结果范围:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - 二进制补码最大值 (正数) : 最多  $2w$  位, 但仅适用于  $(TMin_w)^2$ 
    - 结果范围:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- 因此, 为了保持精确的结果.....
  - 需要随着每次乘积的计算不断扩展字长。
  - 如果需要, 可以用软件完成, 例如, 使用“arbitrary precision”算法包。

# C语言无符号乘法

操作数:  $w$  bits

真实乘积:  $2 \cdot w$  bits

丢弃  $w$  位:  $w$  bits



## ■ 标准乘法

- 忽略高  $w$  位

## ■ 相当于对乘积执行了模运算

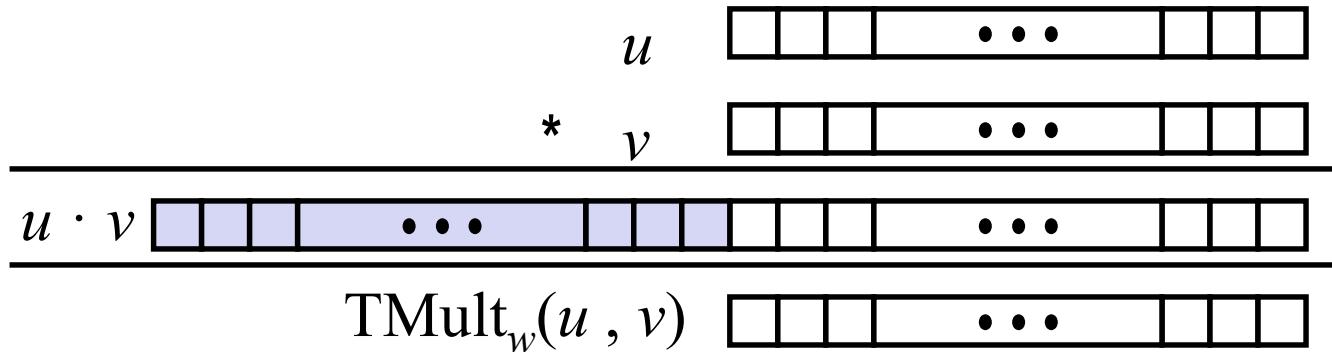
- $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

# C语言有符号(补码)乘法

操作数:  $w$  bits

真实乘积:  $2 \cdot w$  bits

丢弃  $w$  位:  $w$  bits



## ■ 标准乘法功能

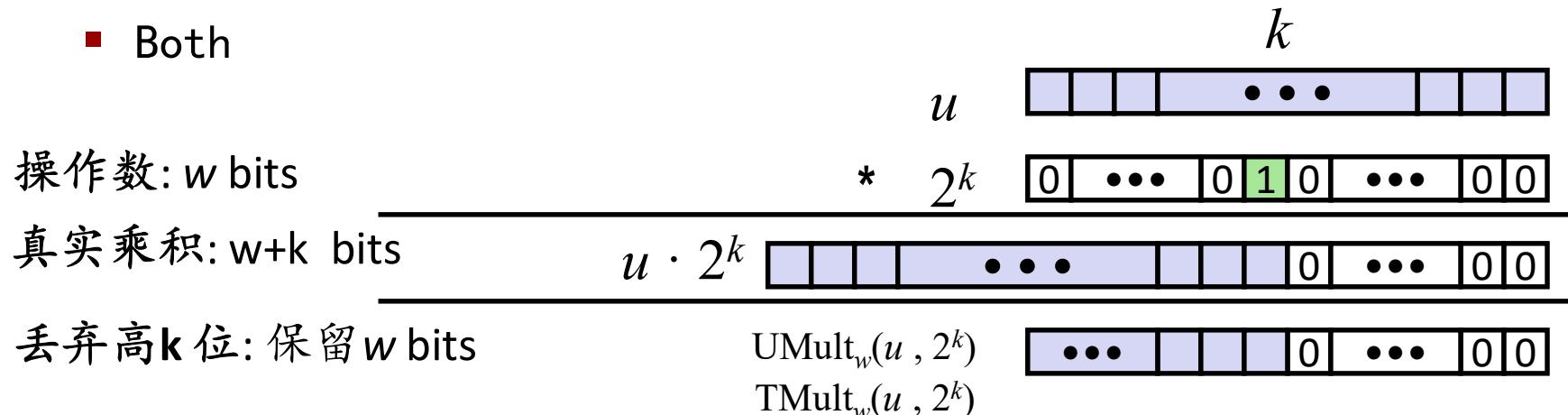
- 忽略高  $w$  位
- 有符号数乘与无符号数乘的不同之处：乘积的符号扩展
- 乘积的低位相同

$$\begin{array}{r}
 & 1110 \ 1001 & -23 \\
 * & 1101 \ 0101 & * \ -43 \\
 \hline
 0000 \ 0011 \ 1101 \ 1101 & \hline 989 \\
 \hline
 & 1101 \ 1101 & -35
 \end{array}$$

# 用移位实现“乘以2的幂”

- 无论有符号数还是无符号数：

- $u \ll k$  得到  $u * 2^k$
- Both



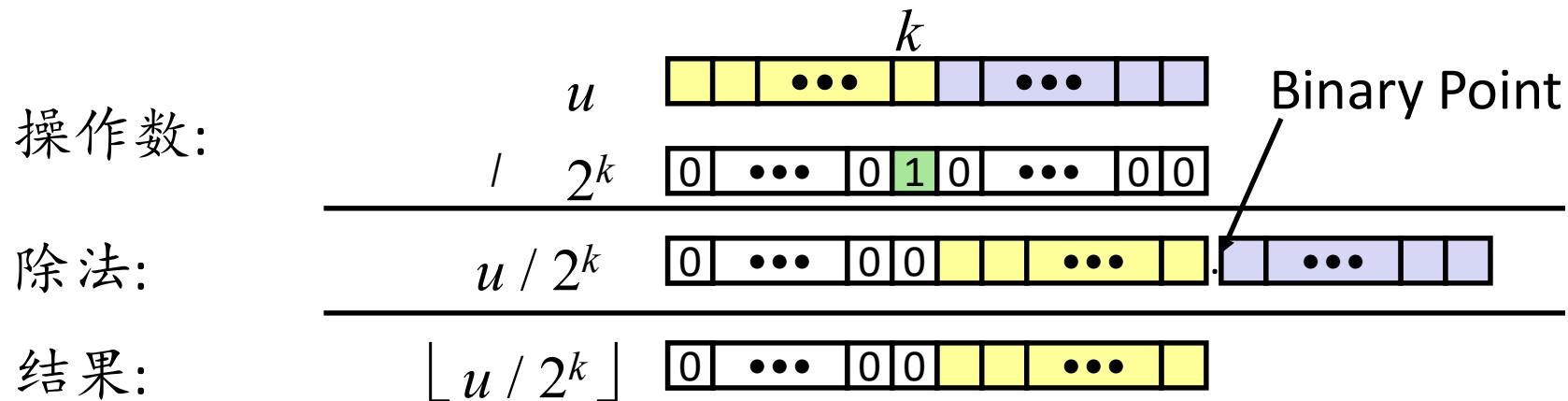
- 例子

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- 绝大多数机器，移位比乘法快
  - 编译器自动生成基于移位的乘法代码

# 用移位实现无符号数“除以2的幂”

- 无符号数“除以2的幂”的商

- $u \gg k$  得到  $\lfloor u / 2^k \rfloor$
- 使用逻辑右移



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

# 用移位实现有符号数“除以2的幂”

- 有符号数“除以2的幂”的商
  - $u \gg k$  得到  $\lfloor u / 2^k \rfloor$
  - 使用算术右移
- 算术移位与逻辑移位
  - 逻辑移位：无论符号位如何，都将所有的位向左或向右移动，空出的位用零填充。
  - 算术移位：用于保留符号位。在进行除法时，算术移位会确保符号位不变，空出的位由符号位填充

# 取负数

- 取反+1
- $\sim x + 1 = -x$
- 步骤一：对数字进行取反，即把二进制表示中的每一位都反转，0变1，1变0。
- 步骤二：对取反后的结果加1，这一步完成后，得到的就是原数字的负值。

# 主要内容：比特、字节 和 整型数

- 以比特表示信息
- 比特运算
- 整型数 (**Integers**)
  - 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算：加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 算术运算：基本规则

## ■ 加法：

无/有符号数的加法：正常加法后再截断，位级运算相同

- 无符号数：加后对  $2^w$  求模
  - 数学加法 + 可能减去  $2^w$
- 有符号数：修改的加后对  $2^w$  求模，使结果在合适范围
  - 数学加法 + 可能减去或加上  $2^w$

## ■ 乘法：

无/有符号数的乘法：正常乘法后加截断操作，位级运算相同

- 无符号数：乘后对  $2^w$  求模
- 有符号数：修改的乘后对  $2^w$  求模，使结果在合适范围内

# 为何用无符号数？

- 一定要知道隐含的转换规则，否则不要用

- 易犯的错误

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

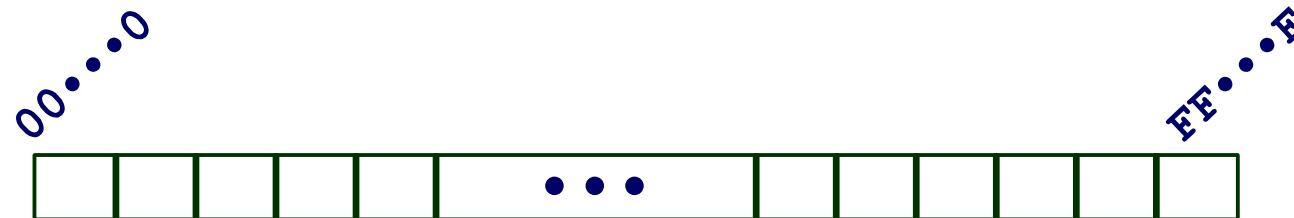
- 不易察觉的问题

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```

# 主要内容：比特、字节 和 整型数

- 以比特表示信息
- 比特运算
- 整型数 (Integers)
  - 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算：加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 面向字节的内存组织管理



- 程序用地址来引用内存中的数据
  - 内存可看做巨大的“字节数组”
    - 实际上不是这样，但不妨这样联想
- 地址就像这个“字节数组”的索引
  - 指针变量可保存地址数值
- 注意：
  - 操作系统为每个进程提供私有的地址空间
  - 每个进程可访问自己地址空间中的内存数据，彼此不干扰。

# 机器字 Machine Words

- 任何机器都有一个 “字长/位数” (Word Size)
  - 整型值数据的名义长度
  - 1985年intel 386 CPU开始, 大多数机器使用32位 (4字节)
    - 地址空间最大4GB (2<sup>32</sup> bytes)
  - 目前, 64位的机器是主流
    - 潜在地, 可以有18 PB 的可寻址内存
    - 约 $18.4 \times 10^{18}$ 字节, 18,400,000 TB
  - 机器依然支持多种数据格式
    - 位数的一部分或几倍长度
    - 始终是整数个字节

# C数据类型的典型大小(字节数)

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	8	8	8
<b>long double</b>	-	-	10/16
pointer	4	8	8

# 字节排序

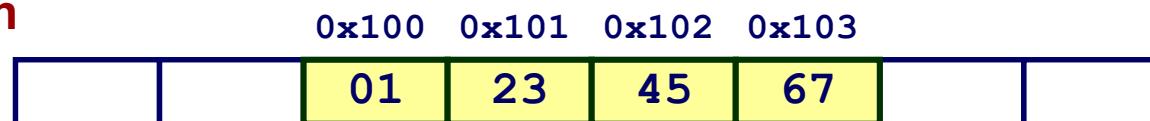
- “字” (word) 有多个字节，各字节在内存中如何排列？
- 惯例
  - 大端序、大尾序 (Big Endian) : Sun, PPC Mac, Internet
    - 最低有效位字节的地址最高
  - 小端序、小尾序 (Little Endian) : x86、运行Android 的 ARM处理器、iOS和Windows
    - 最低有效位字节的地址最低
- 双端序 (Bi-Endian)
  - 机器可以配置成大端序或小端序
  - 很多新近的处理器均支持双端序

# 字节排序示例

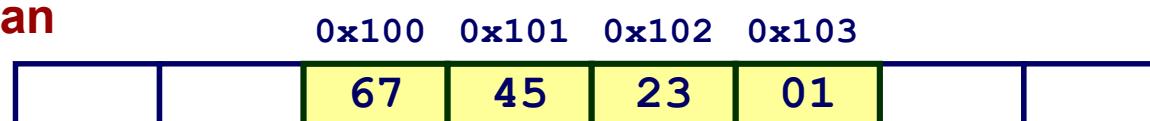
## ■ 示例

- 变量x有4字节数值0x01234567
- 假定x的地址为 0x100

**BigEndian**

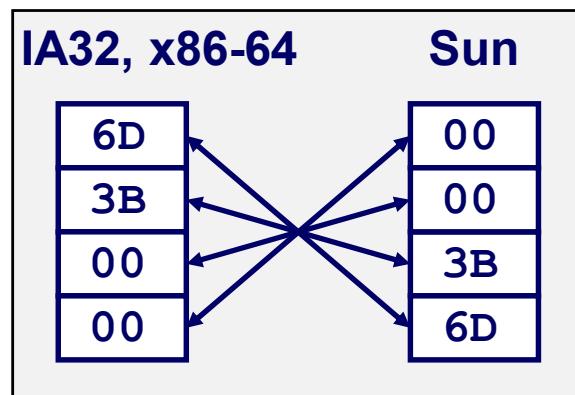


**LittleEndian**

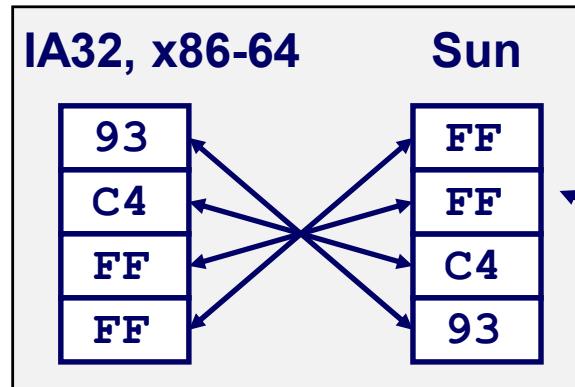


# 整型数的表示

```
int A = 15213;
```



```
int B = -15213;
```

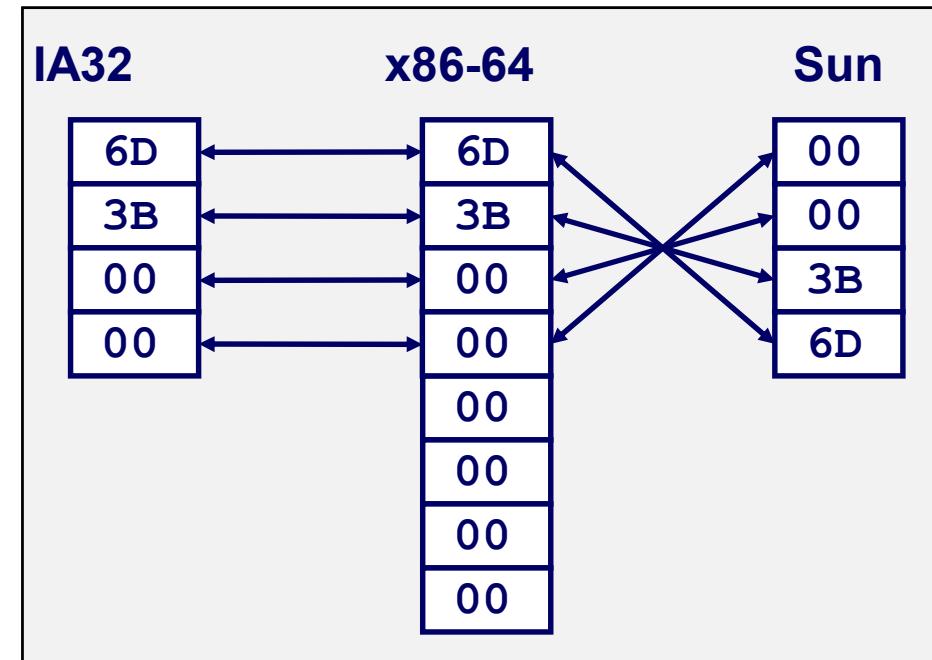


Decimal: 15213

Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

```
long int C = 15213;
```



Two's complement representation

# 验证数的表示

- 打印数据字节表示的程序代码
  - 将指针转换成unsigned char \* 类型，从而按字节数组处理

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

printf 指令：  
%p：打印指针  
%x：16进制格式打印

# show\_bytes 的执行实例

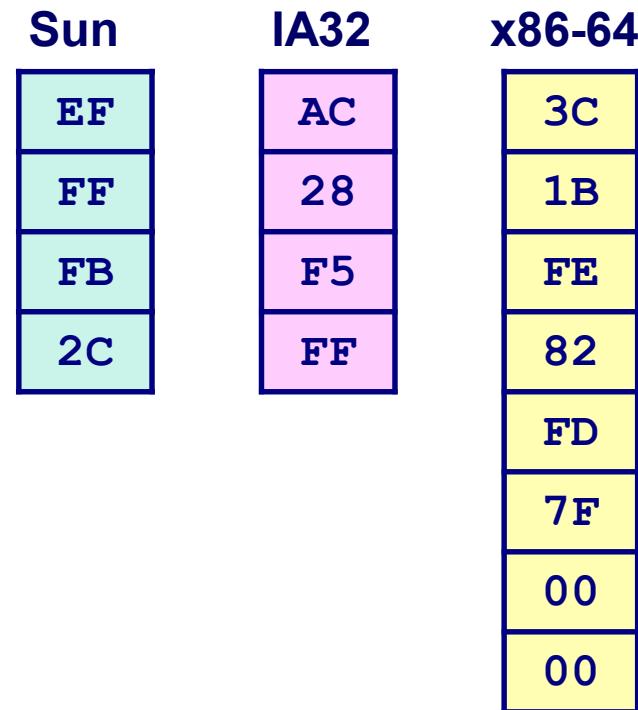
```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

## Result (Linux x86-64):

```
int a = 15213;
0x7ffffb7f71dbc      6d
0x7ffffb7f71dbd      3b
0x7ffffb7f71dbe      00
0x7ffffb7f71dbf      00
```

# 指针的表示

```
int B = -15213;  
int *P = &B;
```



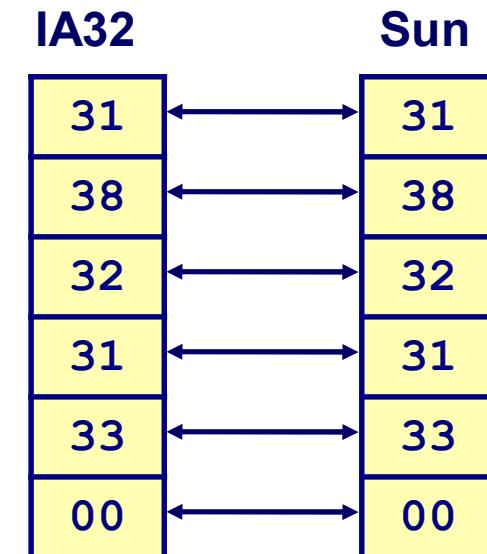
不同的编译器、机器会有不同的运行结果。甚至程序的每次运行结果都不同

# 字符串的表示

## ■ C字符串

- 用字符数组表示
- 每个字符都是ASCII格式编码
  - 字符集合的标准7位编码
  - 字符'0'的编码是 0x30
- 字符串以null结尾
  - 最后的字符 = 0

```
char S[6] = "18213";
```



# 浮点数

Aug. 29, 2025

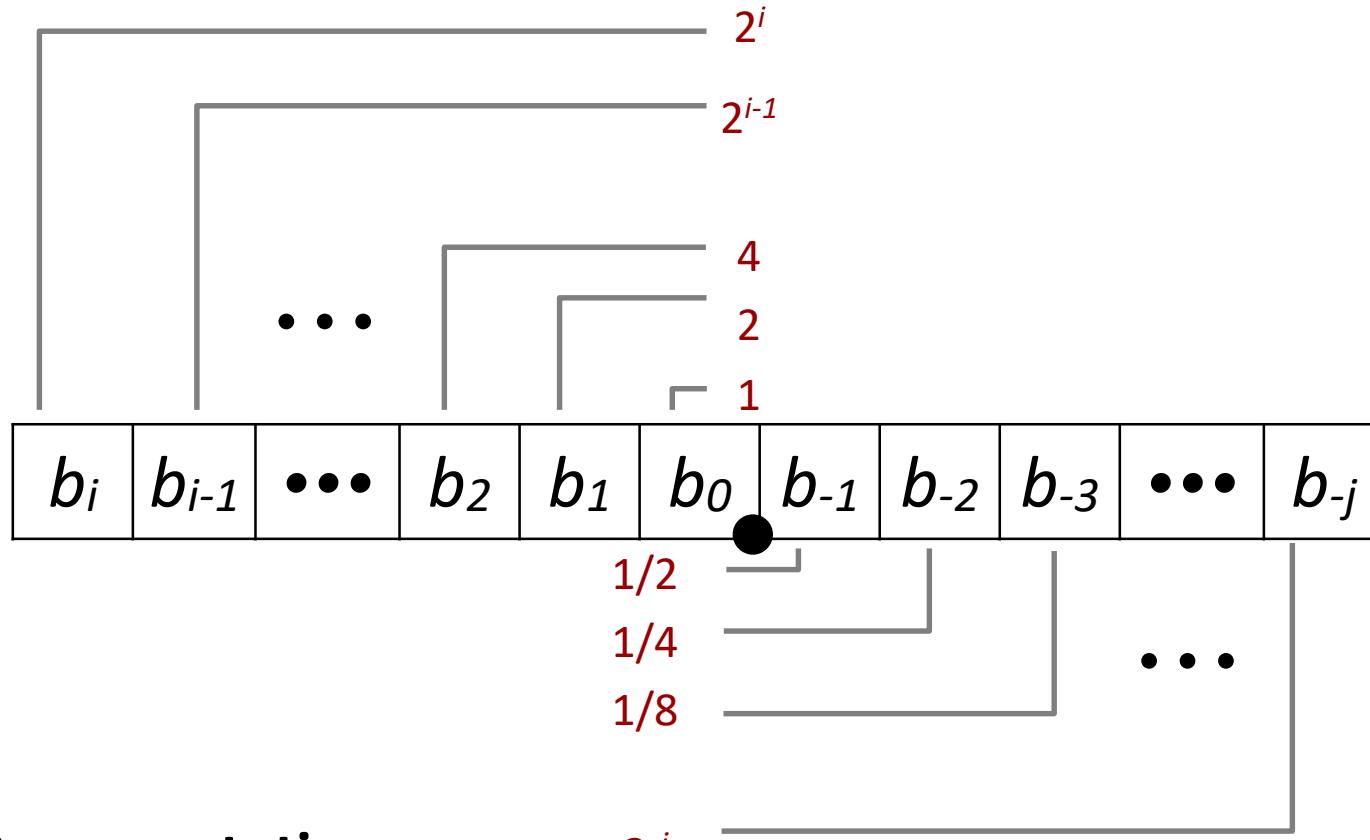
# 主要内容

- 背景：二进制小数
- IEEE 浮点标准：定义
- 示例和属性
- 舍入、加法、乘法
- C 语言浮点数
- 总结

# 二进制小数

■  $1011.101_2$ ?

# 二进制小数



## ■ Representation

- “小数点”右边的位代表小数部分
- 表示的有理数:

$$\sum_{k=-j}^i b_k \times 2^k$$

# 二进制小数: 例子

■ 数值	二进制小数
5 3/4	101.11 <sub>2</sub>
2 7/8	10.111 <sub>2</sub>
1 7/16	1.0111 <sub>2</sub>

## ■ 特点

- 除以2 → 右移 (无符号数)
- 乘以2 → 左移
- 0.111111...<sub>2</sub>
  - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
  - Use notation  $1.0 - \varepsilon$
  - 是最接近1.0的小数

# 二进制数的问题

## ■ 局限性 1——近似表示

- 只能精确表示形如  $x/2^k$ 
  - 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] ...<sub>2</sub>
  - $1/5$         0.001100110011 [0011] ...<sub>2</sub>
  - $1/10$       0.0001100110011 [0011] ...<sub>2</sub>

## ■ 局限性2：在计算机内的实现问题

- 长度有限的 w位
- 在w位内，二进制小数点只能有一种设定方式
- 限制了数的范围(非常小? 非常大?)

# 主要内容

- 背景：二进制小数
- IEEE 浮点标准：定义
- 示例和属性
- 舍入、加法、乘法
- C 语言浮点数
- 总结

# 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

## ■ 数值问题驱动

- 好的标准：舍入、上溢、下溢
- 硬件实现很难做得快
  - 在定义标准方面，数字分析师在硬件设计师中占主导地位。



# 浮点表示

## ■ 数的表示形式:

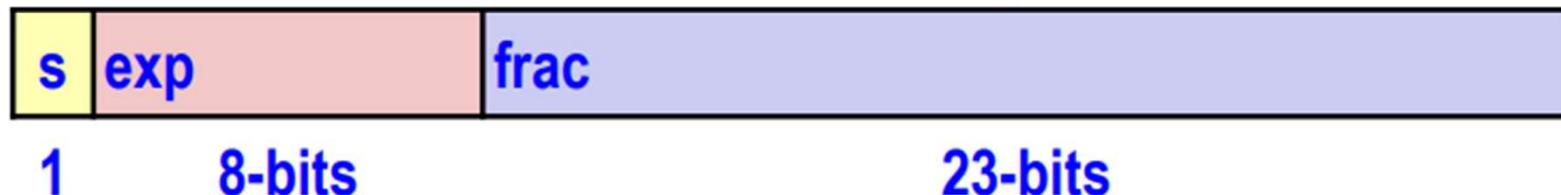
$$(-1)^s M \ 2^E$$

- 符号位 (Sign)  $s$  决定数的符号，是正数( $s=0$ )或负数( $s=1$ )
- 有效数/尾数 (Significand)  $M$  normally a fractional value in range [1.0,2.0).
- 阶码 (Exponent)  $E$  weights value by power of two
- Example:  $15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$
- 编码
  - 最高有效位(MSB) $s$ 作为符号位  $s$
  - exp 字段编码  $E$  (和 $E$ 不一定相等)
  - frac 字段编码尾数  $M$  (和 $M$ 不一定相等)

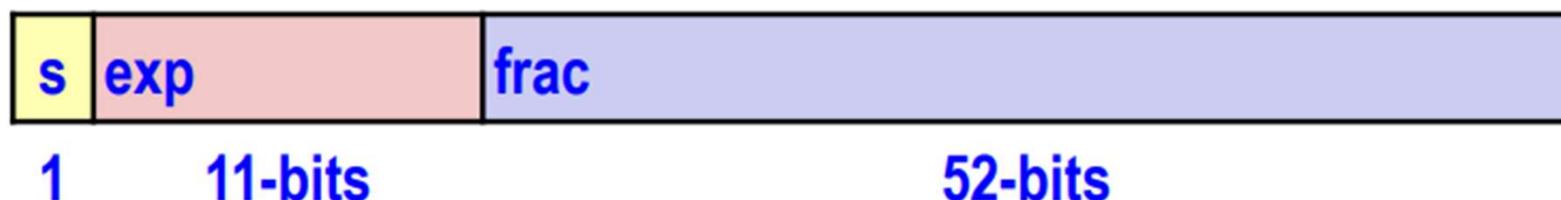


# 精度选项

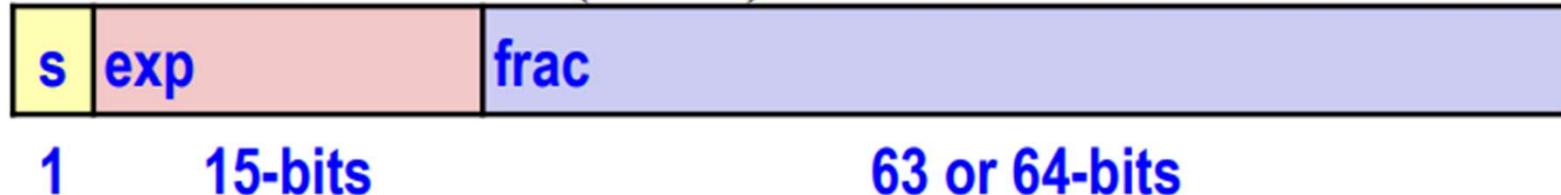
- 单精度: 32 bits  $\approx 7$  decimal digits,  $10^{\pm 38}$



- 双精度: 64 bits  $\approx 16$  decimal digits,  $10^{\pm 308}$



- 扩展精度: 80 bits (Intel )



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

# 规格化数

- 条件:  $\text{exp} \neq 000\dots0$  且  $\text{exp} \neq 111\dots1$
- 阶码(Exponent) 采用偏置值编码:  $E = \text{Exp} - \text{Bias}$ 
  - $\text{Exp}$ : exp 字段的无符号数值  $\text{Exp} = E + \text{Bias}$
  - 偏置  $\text{Bias} = 2^{k-1} - 1$ ,  $k$  为阶码的位数
    - 单精度: 127 ( $\text{Exp}: 1\dots254$ ,  $E: -126\dots127$ )
    - 双精度: 1023 ( $\text{Exp}: 1\dots2046$ ,  $E: -1022\dots1023$ )
- 尾数(Significand) 编码隐含先导数值1:  $M = 1.\text{xxx}\dots\text{x}_2$ 
  - $\text{xxx}\dots\text{x}$ : 是 frac 字段的数码
  - $\text{frac}=000\dots0$  ( $M=1.0$ )时, 为最小值
  - $\text{frac}=111\dots1$  ( $M=2.0 - \varepsilon$ )时, 为最大值
  - 额外增加了一位的精度 (隐含值1)

frac: 尾数域

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

# 规格化数

- 条件:  $\text{exp} \neq 000\dots0$  且  $\text{exp} \neq 111\dots1$
- 阶码(Exponent) 采用偏置值编码:  $E = \text{Exp} - \text{Bias}$ 
  - $\text{Exp}$ : exp 字段的无符号数值  $\text{Exp} = E + \text{Bias}$
  - 偏置  $\text{Bias} = 2^{k-1} - 1$ ,  $k$  为阶码的位数
    - 单精度: 127 ( $\text{Exp}: 1\dots254$ ,  $E: -126\dots127$ )
    - 双精度: 1023 ( $\text{Exp}: 1\dots2046$ ,  $E: -1022\dots1023$ )
- 尾数(Significand) 编码隐含先导数值1:  $M = 1.\text{xxx}\dots\text{x}_2$ 
  - $\text{xxx}\dots\text{x}$ : 是 frac 字段的数码
  - $\text{frac}=000\dots0$  ( $M=1.0$ )时, 为最小值
  - $\text{frac}=111\dots1$  ( $M=2.0 - \varepsilon$ )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

$$M = 1.\underline{1101101101101}_2$$

$$frac = \underline{1101101101101}0000000000_2$$

## ■ 阶码

$$E = 13$$

$$Bias = 127$$

$$Exp = 140 = 10001100_2$$

## ■ 结果:

0	10001100	110110110110100000000000
S	Exp (阶码/指数)	frac (尾数/有效数)

$$v = (-1)^s M 2^E$$
$$E = 1 - \text{Bias}$$

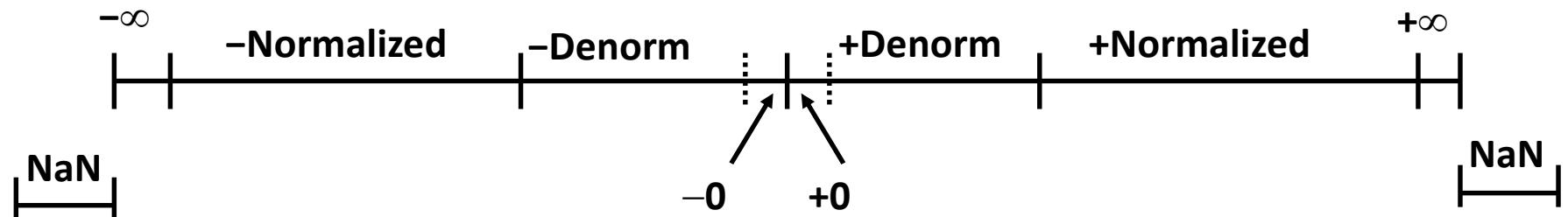
# 非规格化数

- 条件:  $\text{exp} = 000\dots0$ 
  - 阶码(Exponent) 值:  $E = 1 - \text{Bias}$  (**不是  $E = 0 - \text{Bias}$ !**)
  - 尾数(Significand)编码隐含先导数值0:  $M = 0.\text{xxx}\dots\text{x}_2$ 
    - $\text{xxx}\dots\text{x}$ :是 **frac**字段的数码
- 情况1:  $\text{exp} = 000\dots0$ ,  $\text{frac} = 000\dots0$ 
  - 表示值0
  - 注意有不同的数值 +0 和 -0 (why?)
- 情况2:  $\text{exp} = 000\dots0$ ,  $\text{frac} \neq 000\dots0$ 
  - 最接近0.0的那些数
  - 间隔均匀

# 特殊值

- 条件:  $\text{exp} = 111\dots1$
- 情况1:  $\text{exp} = 111\dots1$ ,  $\text{frac} = 000\dots0$ 
  - 表示无穷(infinity)  $\infty$
  - 溢出的运算
  - 正无穷、负无穷
  - E.g.,  $1.0/0.0 = -1.0/-0.0 = +\infty$ ,  $1.0/-0.0 = -\infty$
- 情况2:  $\text{exp} = 111\dots1$ ,  $\text{frac} \neq 000\dots0$ 
  - 表示: 不是一个数Not-a-Number (NaN)
  - 表示没有数值结果（实数或无穷）, 例如:  
 $\sqrt{-1}$ ,  $\infty - \infty$ ,  $\infty \times 0$

# 浮点编码可视化总结



# 主要内容

- 背景：二进制小数
- IEEE 浮点标准：定义
- 示例和属性
- 舍入、加法、乘法
- C 语言浮点数
- 总结

# 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)

	s	exp	frac	E	Value	
非 规 格 化 数	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	$(-1)^0 (0+1/4) * 2^{-6}$
	...	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
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	最小规格化数
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	$(-1)^0 (1+1/8) * 2^{-6}$
	...	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	$8/8 * 1 = 1$	
规 格 化 数	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	

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

$$n: E = Exp - Bias$$

$$d: E = 1 - Bias$$

最接近0

$$(-1)^0 (0+1/4) * 2^{-6}$$

最大非规格化数

最小规格化数

$$(-1)^0 (1+1/8) * 2^{-6}$$

closest to 1 below

closest to 1 above

# 阶码计算bias的优点

## ■ 二进制大小关系失效

假设我们想比较两个数：

数值	指数 e	阶码 (二进制)
0.5	-1	111 (假设补码表示)
1.0	0	000
2.0	1	001

问题：如果没有偏置值且直接使用补码，阶码的二进制大小顺序与实际数值大小不一致。

例如：

- 0.5 (二进制指数 111) 会被认为比 2.0 (001) 大，因为  $111 > 001$ 。
- 这样就无法直接用二进制比较浮点数大小，需要额外逻辑。

# 阶码计算bias的优点

- 二进制大小关系失效

数值	实际指数 $e$	存储阶码 = $e + bias$	阶码 (二进制)
0.5	-1	2	010
1.0	0	3	011
2.0	1	4	100

好处：

- 阶码从 0 到 7 分别对应  $e = -3$  到  $+4$ 。
- 存储的阶码是 **无符号整数**。
- **二进制大小顺序与实际数值顺序一致**，例如：

# 如果阶码不计算bias

- 二进制大小关系失效
- 需要额外用“带符号整数”存储指数
- 引入偏置值的核心目的是让存储的阶码是无符号整数，这样浮点数可以直接按二进制比较大小。

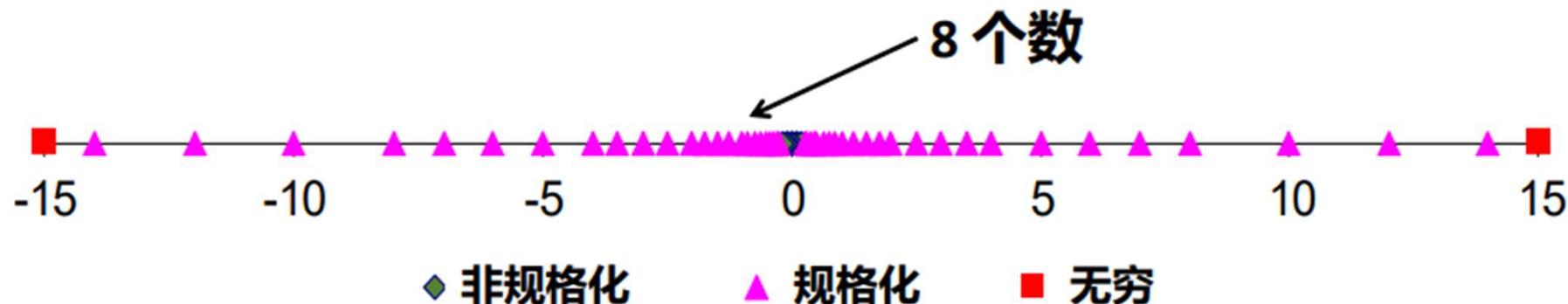
# 数值分布

## ■ 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=  $2^{3-1}-1 = 3$

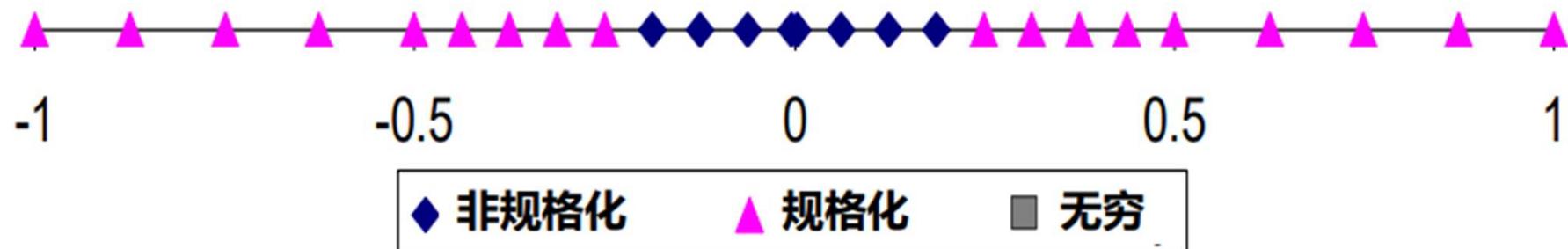
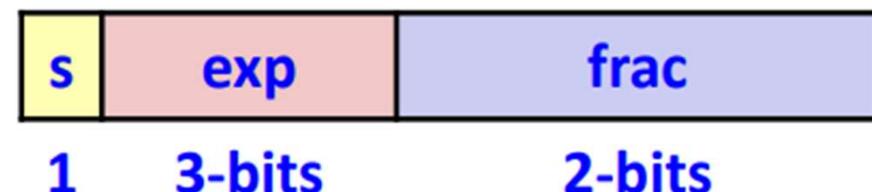


## ■ 注意：数值在趋近于0时变密集



# 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=  $2^{3-1}-1 = 3$



# IEEE编码的特殊性质

- 浮点0与整数0编码相同：所有bit均为0
- 几乎可以用与无符号整数相同的方式进行浮点数的比较
  - 先比较符号位
  - 必须考虑  $-0 = 0$
  - NaN的不确定性
    - 将比其他任何值都大
    - 比较将产生什么结果？
  - 其他方面均OK
    - 规格化值 vs. 非规格化值
    - 规格化值 vs. 无穷

# 主要内容

- 背景：二进制小数
- IEEE 浮点标准：定义
- 示例和属性
- 加法、乘法
- 总结

# 浮点数运算: 基本思想

■  $x +_f y = \text{Round}(x + y)$

■  $x \times_f y = \text{Round}(x \times y)$

## ■ 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 ( $-\infty$ )	\$1	\$1	\$1	\$2	-\$2
■ 向上 ( $+\infty$ )	\$2	\$2	\$2	\$3	-\$1
■ 向偶数舍入(默认)	\$1	\$2	\$2	<b>\$2</b>	-\$2

# 细究“向偶数舍入”

## ■ 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
- E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

# 二进制数的舍入

## ■ 二进制小数的舍入

偶数：最低有效位（LSB）是 0

奇数：最低有效位（LSB）是 1

(1) 找到“舍入位”

假设我们保留到第  $n$  位，舍入位就是第  $n+1$  位。

(2) 判断情况

- 情况 A：舍入位为 0 → 直接截断，不用进位。

- 情况 B：舍入位为 1 → 看舍入位右边是否还有 1：

- 如果有 → 直接进位。

- 如果没有 → 表示正好在“中间”，这时看保留位的最后一位：

- 如果是 0（偶数） → 不进位。

- 如果是 1（奇数） → 进位

# 二进制数的舍入

偶数：最低有效位 (LSB) 是 0  
奇数：最低有效位 (LSB) 是 1

## ■ 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:  $xxx \underline{1}00\dots_2$

## ■ 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
$23/32$	$10.00\underline{0}11_2$	$10.00_2$	( $<1/2$ —down)	2
$23/16$	$10.00\underline{1}10_2$	$10.01_2$	( $>1/2$ —up)	$2\frac{1}{4}$
$27/8$	$10.11\underline{1}00_2$	$11.00_2$	( $1/2$ —up)	3
$25/8$	$10.10\underline{1}00_2$	$10.10_2$	( $1/2$ —down)	$2\frac{1}{2}$

# 浮点乘法

- $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$
- 精确结果:  $(-1)^s M 2^E$ 
  - 符号(Sign)  $s$ :  $s_1 \wedge s_2$
  - 尾数(Significand)  $M$ :  $M_1 \times M_2$
  - 阶码(Exponent)  $E$ :  $E_1 + E_2$
- 修正
  - 如  $M \geq 2$ , 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

# 浮点数加法

■  $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

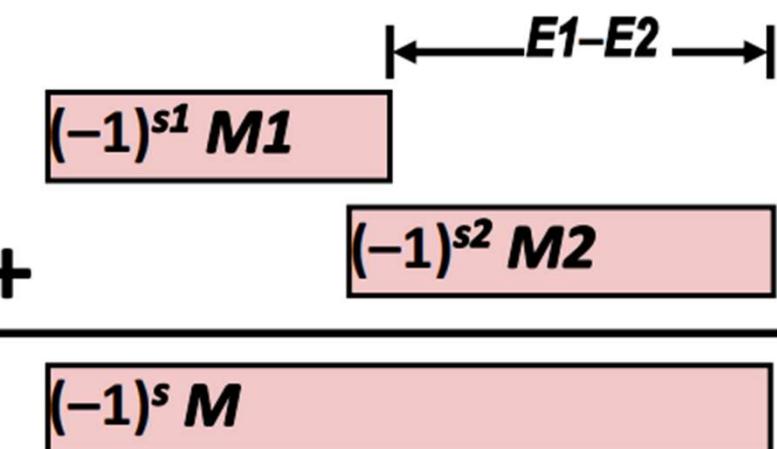
二进制小数点对齐

- 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

- $M \geq 2$ : 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

# 浮点数加法的数学性质

## ■ 与阿贝尔群比较

加法运算下：

- 是否封闭
  - 但可能产生无穷大或 NaN **Yes**
- 交换性(Commutative)? **Yes**
- 分配性(Associative)?
  - 溢出和舍入的不确定性  
 $(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14$**No**
- 0 是加法的单位元? **Yes**
- 每个元素都有逆元?
  - 除了无穷和NaN **Almost**

## ■ 单调性(Monotonicity)

- $a \geq b \Rightarrow a+c \geq b+c?$ 
  - 除了无穷和NaN **Almost**

# 浮点数乘法的数学性质

## ■ 与交换环相比

- 乘法下封闭性?
  - 但可能产生无穷或NaN
- 乘法的交换性?
  - Yes
- 乘法的结合性?
  - 可能溢出、舍入不精确
  - 例:  $(1e20 * 1e20) * 1e-20 = \inf$ ,  $1e20 * (1e20 * 1e-20) = 1e20$
- 1 是乘法的单位元?
  - Yes
- 乘法对加法的分配性?
  - 可能溢出、舍入不精确
  - $1e20 * (1e20 - 1e20) = 0.0$ ,  $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

## ■ 单调性

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c?$ 
  - 除了无穷和 NaN

*Almost*