

# 数据表示

# 13 元培數科 常欣海

Here we go! →

2024/9/25

# 基本概念

basic concepts

**位 (bit)** : 数据存储的最小单位

**字节 (Byte)** : 最小的可寻址的存储器单位, 1

Byte = 8 bits

## 十六进制

- 0x 开头
- 0~9, A~F , 不区分大小写
- 一个字节的值域是  $00_{16} \sim FF_{16}$

$0xA = 10 = 1010$

$0xC = 12 = 1100$

$0xE = 14 = 1110$

\*需要熟练掌握进制转换

# 基本概念

basic concepts

**字长 (word size)** : 决定虚拟地址空间的最大大小

- 对于一个字长为  $w$  位的机器而言, 虚拟地址的范围为  $0 \sim 2^{w-1}$ , 程序最多访问  $2^w$  个字节
- 为什么? 我们需要计算机去寻址, 从而需要用  $w$  位二进制数去表示地址, 所以地址的个数就是  $2^w$ , 所以程序最多访问  $2^w$  个字节

## 字节顺序

- 小端序 (little endian) : 最低有效字节在最前面
- 大端序 (big endian) : 最高有效字节在最前面

**一定要先想清楚字节是怎么排序的 (那边是小地址那边是高地址), 再去辨析大端/小端序**

字节是一个整体, 不会说字节内部是按照小端序还是大端序排列

# 字节顺序

byte order

同样是表示 `0x12345678`，在不同的机器上，内存中的存储可能是：

- 小端序： (低地址) `0x78 0x56 0x34 0x12` (高地址)
- 大端序： (低地址) `0x12 0x34 0x56 0x78` (高地址)

**看到了吗，字节内部顺序是固定的，和大端序小端序无关！**

一定要在脑子里明确知道，所谓内存，就是一系列连续的字节。无论你是现在初学的一维线性模型，还是后面做 lab 时会考虑到的二维模型，首先都要明确那边是低地址那边是高地址。

## 应用

- 现今，大多数计算机都采用 **小端序**。
- 网络通讯中，一般采用 **大端序**。

# 字节顺序

byte order

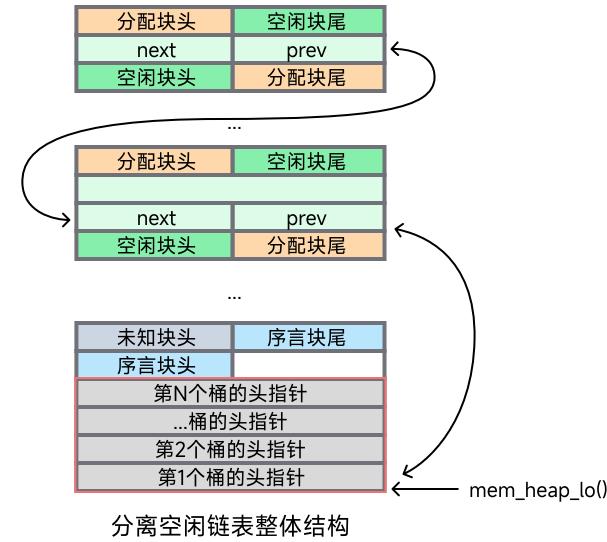
在这张图中，显示了一个内存的排布顺序。

- 同一行中，地址连续，越靠近右侧地址越小
- 同一列中，地址不连续，越靠近下方地址越小
- 也就是说，呈现出了一个 Z 字形

未来，你们可能会见识到各种内存排布方式，但无论如何，都要先搞清楚内存的排布方式，再去辨析大端序小端序。

全部内存 - 一块连续的内存空间（指令/数据） - 大端序/小端序 - 字节

当讨论的这个一块内存空间只有 1 个字节（例如：一个 `char` 类型）时，我们还需要考虑大端序小端序吗？  
NO!



# 布尔运算

boolean operation

`~ (not) 非`

0	1
---	---

1	0
---	---

`| (or) 或`

0	1
---	---

0	0	1
---	---	---

1	1	1
---	---	---

`& (and) 与`

0	0	0
---	---	---

1	0	1
---	---	---

`^ (eor) 异或`

0	1
---	---

0	0	1
---	---	---

1	1	0
---	---	---

# 逻辑运算

logical operation

- 逻辑运算符: `&&` (and, 与)、`||` (or, 或)、`!` (not, 非)
- 逻辑运算符的运算结果只有两种: `true` 和 `false`

**注意与位运算区分! 逻辑运算具有短路特性, 位运算没有!**

短路特性

- `&&` : 前一个表达式为假时, 后一个表达式不执行, 因为无论后一个表达式是什么, 结果都是假
- `||` : 前一个表达式为真时, 后一个表达式不执行, 因为无论后一个表达式是什么, 结果都是真

也即: 如果对第一个参数求值能确定表达式的结果, 那么逻辑运算符不会对第二个参数求值

Q: 为什么 `p && *p++` 不会引用空指针? (注意这里也有短路特性)

A: 因为 `&&` 运算符的短路特性, 当 `p` 为空指针时 (对应其数字表达为 `0x00000000`, 或者说是 `NULL`) , `*p` 不会被执行, 所以不会出现引用空指针的情况。

# 位移

bit shift

- 左移: `<<`
- 右移: `>>`

## 逻辑右移 / 算术右移

- 逻辑右移: 左边补 0
- 算术右移: 左边补 最高位 (思考: 为什么? )
- 逻辑左移 / 算术左移: 右边补 0

C 语言: 有符号数算术右移, 无符号数逻辑右移

运算时, 需要注意运算符优先级 (如果记不住就全加上括号! )

优先级	运算符	结合律	助记
1	<code>::</code>	从左至右	作用域
2	<code>a++、a--、 type()、type{}、 a()、a[]、 . -&gt;</code>	从左至右	后缀自增减、 函数风格转型、 函数调用、下标、 成员访问
3	<code>!、~、 ++a、--a、+a、-a、 (type)、sizeof、&amp;a、 *a、 new、new[]、delete、delete[]</code>	从右至左	逻辑非、按位非、 前缀自增减、正负、 C 风格转型、取大小、取址、 指针访问、 动态内存分配
4	<code>.*、-&gt;*</code>	从左至右	指向成员指针
5	<code>a*b、a/b、a%b</code>	从左至右	乘除、取模
6	<code>a+b、a-b</code>	从左至右	加减
7	<code>&lt;&lt;、&gt;&gt;</code>	从左至右	按位左右移
8	<code>&lt;、&lt;=、&gt;、&gt;=</code>	从左至右	大小比较
9	<code>==、!=</code>	从左至右	等价比较
10	<code>a&amp;b</code>	从左至右	按位与
11	<code>^</code>	从左至右	按位异或
12	<code>,</code>	、	从左至右
13	<code>&amp;&amp;</code>	从左至右	逻辑与
14	<code>,</code>	、	、
15	<code>a?b:c、 =、+=、-=、*=、/=、%=、&amp;=、^=、 =、&lt;&lt;=、&gt;&gt;=</code>	从右至左	
16	<code>,</code>	从左至右	逗号

# 整数表示

Symbol	Type	Meaning	Page
$B2T_w$	Function	Binary to two's complement	100
$B2U_w$	Function	Binary to unsigned	98
$U2B_w$	Function	Unsigned to binary	100
$U2T_w$	Function	Unsigned to two's complement	107
$T2B_w$	Function	Two's complement to binary	101
$T2U_w$	Function	Two's complement to unsigned	107
$TMin_w$	Constant	Minimum two's-complement value	101
$TMax_w$	Constant	Maximum two's-complement value	101
$UMax_w$	Constant	Maximum unsigned value	99
$+^t_w$	Operation	Two's-complement addition	126
$+^u_w$	Operation	Unsigned addition	121
$*^t_w$	Operation	Two's-complement multiplication	133
$*^u_w$	Operation	Unsigned multiplication	132
$-^t_w$	Operation	Two's-complement negation	131
$-^u_w$	Operation	Unsigned negation	125

# 整数编码

integer encoding

## 无符号数

对向量  $x = [x_{w-1}, x_{w-2}, \dots, x_0]$ :

$$\text{B2U}_w(x) = \sum_{i=0}^{w-1} (x_i \times 2^i)$$

- $\text{B2U}_w$  表示把二进制编码转化为非负整数  
(Binary to Unsigned)

该函数是双射，即无符号数编码具有唯一性

- $\text{UMax}_w = 2^w - 1$

## 有符号数

对向量  $x = [x_{w-1}, x_{w-2}, \dots, x_0]$ :

$$\text{B2T}_w(x) = -x_{w-1} \times 2^{w-1} + \sum_{i=0}^{w-2} (x_i \times 2^i)$$

- $\text{B2T}_w$  表示把二进制编码转化为补码  
(Binary to Two's Complement)
- 当最高位为 1，整个数表示为一个负数
- 同样具有唯一性

# 整数编码

integer encoding

对于有符号数，取值范围是不对称的，典型例子 int：

- $TMax_{32} = 2147483647$
- $TMin_{32} = -2147483648$

为什么？

# 数据的其他表示

other representations

- **补码**: 上下文无关——同构环

$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

- **反码**: 0 的表示有两种: 000...0 和 111...1

$$B2O_w(\vec{x}) = -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

- **原码**: 最高位是符号位用来确定剩下的是正数还是负数 (e.g. 浮点数)

$$B2S_w(\vec{x}) = (-1)^{x_{w-1}} \cdot \sum_{i=0}^{w-2} x_i 2^i$$

# 有符号数与无符号数之间的转换

conversion between signed and unsigned

强制类型转换保持位值不变，只改变解释这些位的方式

## 补码→无符号数

对满足  $TMin_w \leq x \leq TMax_w$  的  $x$  来说：

- 若  $TMin_w \leq x < 0$ , 则  $T2U(x) = x + 2^w$
- 若  $0 \leq x \leq TMax_w$ , 则  $T2U(x) = x$

## 无符号数→补码

对满足  $0 \leq x \leq UMax_w$  的  $x$  来说：

- 若  $x > TMax_w$ , 则  $U2T(x) = x - 2^w$
- 若  $0 \leq x \leq TMax_w$  则  $U2T(x) = x$

对于负数，最高位“翻转了阵营”，从补码的代表的  $-2^{w-1}$  变成了  $+2^{w-1}$

当一个无符号数与一个有符号数进行计算时，有符号数在这个表达式中会被当做无符号数（即发生了隐式的强制类型转换）

例如：  $-1 > 0u$  此为无符号的比较

( $-1$  在计算机中表示为全 1, 即  $UMax_w$ , 而  $0u$  表示为全 0 的无符号数)

# 有符号数与无符号数之间的转换

conversion between signed and unsigned

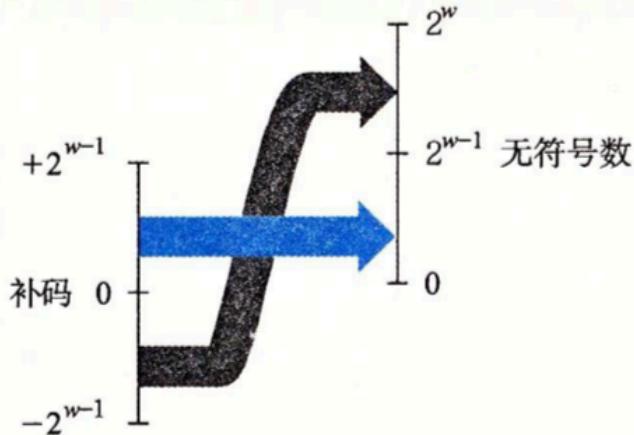


图 2-17 从补码到无符号数的转换。函数  $T2U$  将负数转换为大的正数

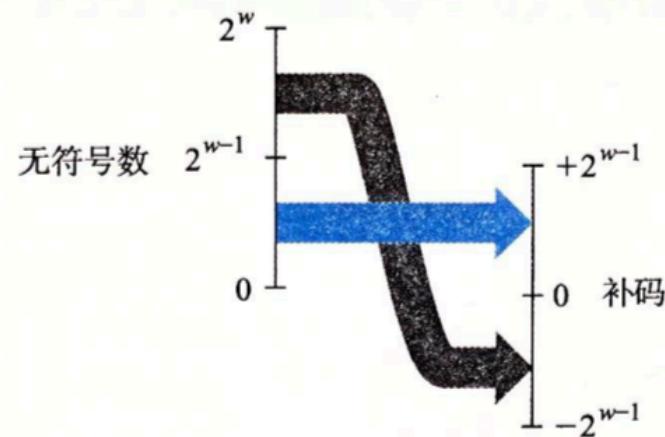


图 2-18 从无符号数到补码的转换。函数  $U2T$  把大于  $2^{w-1}-1$  的数字转换为负值

# 扩展和截断

extension and truncation

## 扩展一个数

- 对于无符号数，高位补 0
- 对于有符号数（补码），高位补符号位

当把 short（有符号数，2 字节）强制转换为 unsigned（无符号数，4 字节）时？

### 先进行数位扩展，再转换为无符号数

为什么对于有符号数，高位补符号位？

Hint: 消消乐！

- 若最高位为 0，显然；
- 若最高位为 1，则你补的一堆 1 和原先的 1 等价

设现在位数为  $m$ ，补到  $n$  位，且  $n > m$ ，则：

$$-2^{n-1} + \sum_{i=m-1}^{n-2} 2^i = -2^{m-1}$$

# 扩展和截断

extension and truncation

## 截断一个数

- 对于无符号数  $x = [x_{w-1}, \dots, x_0]$ , 截断为  $w'$  位, 则  $x' = [x_{w'-1}, \dots, x_0]$   
即  $x' = x \bmod 2^k$
- 补码截断, 原理上与无符号数类似, 但对于数位的解释方式不同

# 整数加减法

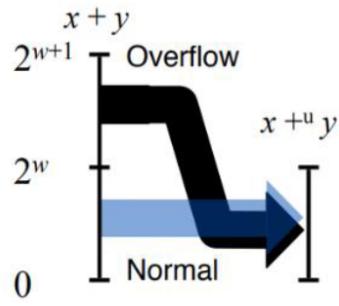
integer addition and subtraction

## 无符号数加法

由于两个  $w$  位的无符号数相加，结果的范围是  $[0, 2^{w+1} - 2]$ ，而这需要  $w + 1$  位来表示，所以实际上：

结果表示  $x + y$  的低  $w$  位，也即  $x + y \bmod 2^w$

- 当  $x + y < 2^w$  时，结果正确；
- 当  $x + y \geq 2^w$  时， $x + y$  的结果需要减去  $2^w$ ，才会得到实际结果，即  $x + y - 2^w$ ，这就是 **溢出**



无符号数加法

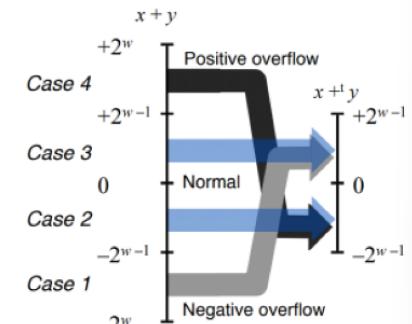
# 整数加减法

integer addition and subtraction

## 有符号数加法

由于两个  $w$  位的有符号数相加，结果的范围是  $[-2^w, 2^w - 2]$ ，所以实际上：

- 若  $-2^{w-1} \leq x + y < 2^{w-1}$ ，结果正确；
- 若  $x + y \geq 2^{w-1}$ ，结果需要减去  $2^w$ ，才会得到实际结果，即  $x + y - 2^w$ ，此时称为 **正溢出/上溢出**
- 若  $x + y < -2^{w-1}$ ，结果需要加上  $2^w$ ，才会得到实际结果，即  $x + y + 2^w$ ，此时称为 **负溢出/下溢出**



有符号数加法

# 判断溢出

determine overflow

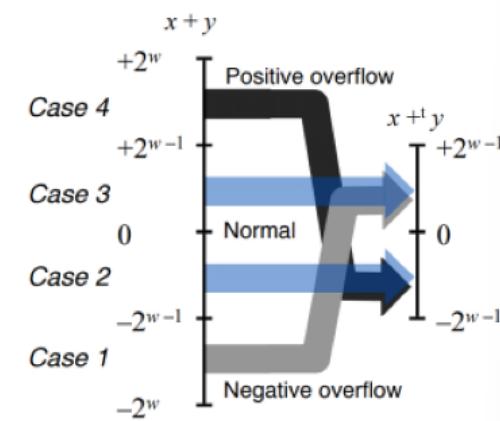
## 无符号数加法

- $s = x + y$ , 若  $s < x$  或  $s > y$  则溢出

## 有符号数加法

- $s = x + y$ , 若  $x > 0$  且  $y > 0$  且  $s \leq 0$ , 则正溢出;
- 若  $x < 0$  且  $y < 0$  且  $s \geq 0$ , 则负溢出

为什么只有这两种情况?



有符号数加法

# 判断溢出

determine overflow

关于整数加减法/溢出的一个有趣事实：这里的加法是个 **环**（阿贝尔群）！

无论你怎么改变次序，它总是可以轮换回来~

这是一个做题很方便的技巧！

- **补码的非**（加法逆元）：

- 数学公式

$$\neg_w^t x = \begin{cases} TMin_w & x = TMin_w \\ -x & x > TMin_w \end{cases}$$

- 计算机实现(位级表示)

$$\begin{aligned} \neg x &= \sim x + 1 \\ \text{because : } x + (-x) &= \underset{w}{=} 2^w \end{aligned}$$

# 补码的非运算

two's complement negation

对满足  $TMin_w \leq x \leq TMax_w$  的  $x$  来说：

- 若  $x = TMin_w$ , 则  $\sim x = TMin_w$
- 否则,  $\sim x = -x$  (算术上的)

注：这里式子左边的  $\sim x$  是 **算数逆元**

- 对于  $x \equiv TMin_w$  时, 本身  $\sim x$  也在表示范围内, 结论是平凡的
- 对于  $x = TMin_w$  时,  $\sim x$  不在表示范围内。

但是, 两个  $TMin_w$  相加, 溢出到了上一位, 结果是 0, 所以此时我们说  $TMin_w$  的算数逆元就是它自身

回忆：刚才说的  $\sim x + 1 = -x$

# 乘除法

multiplication and division

## 无符号数乘法

等于乘法计算得到的结果（一个  $2w$  位长的数），而后截断到  $w$  位

## 补码乘法

等于有符号数乘法后得到的结果（一个  $2w$  位长的数），无符号截断到  $w$  位，而后将最高位转化为符号位

**核心：粗暴的位级表示截断**

## 乘以二的幂次

此时，**相当于左移**

# 乘除法

multiplication and division

特别注意：除法这里以二的幂次作为除数，也只有此时可以采用右移来取巧。

## 无符号数除法

无符号数  $u$  除以  $2^k$ ：将  $u$  逻辑右移  $k$  位

(此时恰好就是舍入到 0 的)

## 有符号数除法

直接右移  $x \gg k$  得到的是 向下舍入 的结果，而不是正常来讲的向 0 取整。

例如： $-3 / 2 = -1$ ，而  $-3 \gg 1 = -2$

向 0 取整： $(x < 0) ? (x + (1 << k) - 1 : x) \gg k$

证明见书 P73

# 浮点数表示

floating point

IEEE 754 标准

本质是对实数的近似

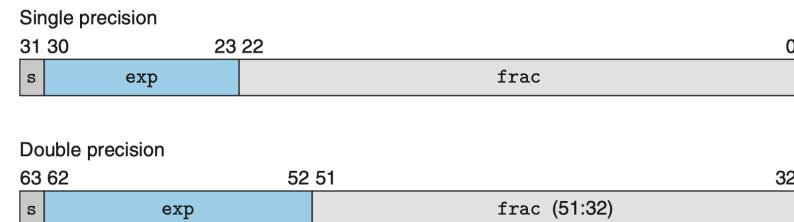
- 符号位: 1 位
- 指数位:  $e$  位
- 尾数位:  $m$  位

实际表示:

- 单精度浮点数 ( `float` ) :  $e = 8, m = 23$
- 双精度浮点数 ( `double` ) :  $e = 11, m = 52$

各个位长多少要牢牢记清楚! 考试会考!

注意, 0 的表示有两种, +0 和 -0



# 理解 IEEE 754

underlying IEEE 754

## 规格化值

$$V = (-1)^s \times (1.M) \times 2^E$$

- $s$  为符号位,  $M$  为尾数,  $E$  为阶码
- $E = [e] - Bias$ , 其中  $[e]$  为阶码处这  $e$  位的实际值,  $Bias = 2^{e-1} - 1$
- 尾数隐含的 1 (Implied leading 1), 可以获得 1 位额外精度

## 非规格化值

$$V = (-1)^s \times (0.M) \times 2^{1-Bias}$$

- $E = 1 - Bias$ , 虽然此时  $[e] = 0$ , 但如此规定可以实现规格化值和非规格化值的平滑过渡
- 尾数没有隐含的 1

非规格化值的看似反直觉的定义实现了**规格化值和非规格化值的平滑过渡**

# 理解 IEEE 754

underlying IEEE 754

$$V = (-1)^s \times 2^E \times M$$

偏置 :  $Bias = 2^{k-1} - 1$

指数:  $E = e - Bias$

## 特殊值

- 最小的非规格化数  $not_{min} = 2^{1-Bias} * 2^{-n} = 2^{2-2^{k-1}} * 2^{-n} = 2^{2-2^{k-1}-n}$
- 最大的规格化数  $not_{max} = 2^{1-Bias} * (1 - 2^{-n}) = 2^{2-2^{k-1}} * (1 - 2^{-n})$
- 最小的规格化数  $yes_{min} = 2^{1-Bias} * (1) = 2^{2-2^{k-1}}$
- 最大的规格化数  $yes_{max} = 2^{(2^{k-1})-Bias-1} * (2 - 2^{-n}) = 2^{2^{k-1}-1} * (2 - 2^{-n})$

# 理解 IEEE 754

underlying IEEE 754

## 无穷 ( $\pm\infty$ )

- 符号位  $s$  区分正无穷/负无穷
- 阶码  $[e] = 1\dots1$  (全1) , 尾数  $M = 0$
- 表示无穷大, 在计算中可以用于表示溢出或未定义的结果

## 零

- 符号位  $s$  任意, 阶码  $[e] = 0\dots0$  (全0) , 尾数  $M = 0$

## 非数值 (NaN, Not a Number)

- 符号位  $s$  任意, 阶码  $[e] = 1\dots1$  (全1) , 尾数  $M \not\equiv 0$
- NaN 用于表示未定义或无法表示的值, 例如  $0/0$  或  $\sqrt{-1}$
- 通常在比较中被认为不等于任何值, 包括自身

## 特殊值的应用

- 处理异常情况: 如除零错误、溢出、下溢等
- 提高计算的鲁棒性, 避免程序崩溃
- 在数值算法中用于标记和处理特殊情况

# 理解 IEEE 754

underlying IEEE 754

## 平滑过渡

考虑某一进制,  $e = 2, m = 3$  (考试的时候可能就会考改变进制的题哦! )

0 01 000

0 00 111

$$Bias = 2^{e-1} - 1 = 2^{2-1} - 1 = 1$$

$$1.000_2 \times 2^{01_2-1} = 1.000_2$$

$$0.111_2 \times 2^{1-1} = 0.111_2$$

# 浮点舍入

rounding

由于浮点数是对实数的近似，浮点数可以精确表示的实数是有限的，所以舍入是不可避免的。

## 实数舍入到浮点数

- 向偶数舍入 (round-to-even) (round-to-nearest)
- 类比“四舍六入五成双”，**避免统计偏差**

$$1.234 \Rightarrow 1.0$$

$$1.678 \Rightarrow 2.0$$

$$1.500 \Rightarrow 2.0$$

\* 实际上，这一过程发生在浮点数精度最末位

## 浮点数转整型

- 如果舍入，向零舍入
- 如果溢出，C 语言未规定 (undefined behavior)，各自处理 (Intel:  $T_{\min}^w$ ，即舍入到限定最接近的数)

# 浮点类型转换

- 舍入规则
  - 一般情况看舍入位，0 舍1入
  - 类似“四舍六入五成双”，若舍入位为1且后续位全0，向偶舍入
- int/float,double 互转
  - int 转 float
    - 可能发生舍入
  - double 转 float
    - 可能溢出，可能舍入
  - float,double 转 int
    - 如果需要舍入，向零舍入
    - 如果发生溢出，未定义行为，一般得到 T<sub>min</sub>
  - 其余情况，得到精确值
- 和 long 之间的转换类似，但需另作讨论

# 浮点运算

floating point arithmetic

- 加法可交换, 加法不可结合, `NaN` 没有加法逆元
- **浮点加法单调性**, 如果  $a \geq b$ , 那么对于任何  $a, b$  以及  $x$  的值, 除了 `NaN` 都有  $x + a \geq x + b$
- 乘法可交换, 乘法不可结合, 乘法在加法上不可分配
- 小心特殊值: `+inf`, `-inf`, `NaN` (`NaN` 不能通过任何运算变为其它值)

\* 这些东西说了没用, 得你自己做题踩坑才会知道。

- Invalid operation: 零乘无穷、零除零、无穷除无穷、无穷绝对值相减...
- Division by zero: 有穷数除零结果为无穷
- Overflow: 无穷
- Underflow: 舍入结果
- Inexact: 舍入结果

更多详情请见 [IEEE754 异常处理](#)

# 浮点运算

floating point arithmetic

## 01 加法注意事项

### Mathematical Properties of FP Add

#### ■ Compare to those of Abelian Group

- Closed under addition? **Yes**
  - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
  - Overflow and inexactness of rounding
  - $(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14$
- 0 is additive identity? **Yes**
- Every element has additive inverse? **Almost**
  - Yes, except for infinities & NaNs

#### ■ Monotonicity

- $a \geq b \Rightarrow a+c \geq b+c?$  **Almost**
  - Except for infinities & NaNs

## 02 乘法注意事项

### Mathematical Properties of FP Mult

#### ■ Compare to Commutative Ring

- Closed under multiplication? **Yes**
  - But may generate infinity or NaN
- Multiplication Commutative? **Yes**
- Multiplication is Associative? **No**
  - Possibility of overflow, inexactness of rounding
  - Ex:  $(1e20*1e20)*1e-20 = \text{inf}, 1e20*(1e20*1e-20) = 1e20$
- 1 is multiplicative identity? **Yes**
- Multiplication distributes over addition? **No**
  - Possibility of overflow, inexactness of rounding
  - $1e20*(1e20-1e20) = 0.0, 1e20*1e20 - 1e20*1e20 = \text{NaN}$

#### ■ Monotonicity

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c?$  **Almost**
  - Except for infinities & NaNs

# 学习方法

## 使用 AI

## 手敲代码

### Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

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

Assume neither
d nor f is NaN
```

•  $x == (int)(float) x$  ✓  
 •  $x == (int)(double) x$  ✓  
 •  $f == (float)(double) f$  ✗  
 •  $d == (double)(float) d$  ✗  
 •  $f == -(x);$  ✓  
 •  $2/3 == 2/3.0$  ✗  
 •  $d < 0.0 \Rightarrow ((d*2) < 0.0)$  ✓  
 •  $d > f \Rightarrow -f > -d$  ✓  
 •  $d * d \geq 0.0$  ✓  
 •  $(d+f)-d == f$  ✗

请解释每一个案例，用中文



这张图片展示了一些与浮点数有关的C语言表达式，我们逐一解释每个表达式的正确性。

#### 1. $x == (int)(float) x$ (✗ 错误)

这个表达式不是对所有整数  $x$  都成立。原因是浮点数  $float$  只有有限的精度。当  $x$  足够大时， $(float) x$  会因为精度损失而不精确，从而  $int$  转换回去后可能不等于原始的  $x$ 。

#### 2. $x == (int)(double) x$ (✓ 正确)

$double$  的精度比  $int$  和  $float$  高，因此将  $int$  转换为  $double$  不会丢失精度，并且再转换回  $int$ ，结果还是  $x$ 。

#### 3. $f == (float)(double) f$ (✓ 正确)

$float$  转换为  $double$  会保持值不变，因为  $double$  有更高的精度。当  $double$  转换回  $float$  时，

给“ChatGPT”发送消息

ChatGPT 也可能犯错。请核查重要信息。

```
ICS [SSH: 10.129.81.70]
class-in > 01-data > c show_bytes.c
#include <stdio.h>
typedef unsigned char *pointer;
void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t%02x\n", start + i, start[i]);
    printf("\n");
}
int main()
{
    int a = 15213;
    printf("int a = 15213;\n");
    show_bytes((pointer)&a, sizeof(int));
    return 0;
}

[Running] cd "/home/ubuntu/ICS/class-in/01-data/" && gcc show_bytes.c -o show_bytes && "/home/ubuntu/ICS/class-in/01-data/"show_bytes
int a = 15213;
0xffffde6d7f984 0x6d
0xffffde6d7f985 0x3b
0xffffde6d7f986 0x00
0xffffde6d7f987 0x00

[Done] exited with code=0 in 0.068 seconds
```

The screenshot shows a terminal window titled "ICS [SSH: 10.129.81.70]" displaying C code for printing memory bytes. The code defines a `show\_bytes` function that prints memory starting from a pointer `start` for a length of `len`. It uses `printf` with format specifiers "%p" and "%02x". The `main` function creates an integer `a` with value 15213 and calls `show\_bytes` with it. The output shows the memory bytes at address 0xffffde6d7f984 followed by their hex values 0x6d and 0x3b, then 0x00 and 0x00 respectively. The terminal also shows the command to compile and run the program.

# 作业讲评

P88 2.59, 2.60

\*\* 2.59 编写一个 C 表达式，它生成一个字，由  $x$  的最低有效字节和  $y$  中剩下的字节组成。对于运算数  $x = 0x89ABCDEF$  和  $y=0x76543210$ ，就得到  $0x765432EF$ 。

\*\* 2.60 假设我们将一个  $w$  位的字中的字节从 0(最低位)到  $w/8-1$ (最高位)编号。写出下面 C 函数的代码，它会返回一个无符号值，其中参数  $x$  的字节  $i$  被替换成字节  $b$ :

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

以下示例，说明了这个函数该如何工作：

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678  
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

根据题目给出的示例，这里认为使用32位机器，一个字大小为4字节。

```
unsigned concat_byte(unsigned x, unsigned y){  
    return (x & 0xFF) | (y & 0xFFFFF00);  
}
```

# 作业讲评

P88 2.59, 2.60

\*\* 2.59 编写一个 C 表达式，它生成一个字，由  $x$  的最低有效字节和  $y$  中剩下的字节组成。对于运算数  $x = 0x89ABCDEF$  和  $y=0x76543210$ ，就得到  $0x765432EF$ 。

\*\* 2.60 假设我们将一个  $w$  位的字中的字节从 0(最低位)到  $w/8-1$ (最高位)编号。写出下面 C 函数的代码，它会返回一个无符号值，其中参数  $x$  的字节  $i$  被替换成字节  $b$ :

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

以下示例，说明了这个函数该如何工作：

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678  
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

```
unsigned replace_byte(unsigned x, int i, unsigned char b)  
{  
    unsigned mask = 0xFF << (i * 8);  
    unsigned shifted_byte = b << (i * 8);  
    return (x & ~mask) | shifted_byte;  
}
```

# 作业讲评

## P91 2.71

- \* 2.71 你刚刚开始在一家公司工作，他们要实现一组过程来操作一个数据结构，要将 4 个有符号字节封装成一个 32 位 unsigned。一个字节中的字节从 0(最低有效字节)编号到 3(最高有效字节)。分配给你的任务是：为一个使用补码运算和算术右移的机器编写一个具有如下原型的函数：

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

也就是说，函数会抽取出指定的字节，再把它符号扩展为一个 32 位 int。

你的前任(因为水平不够高而被解雇了)编写了下面的代码：

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
    return (word >> (bytenum << 3)) & 0xFF;
}
```

- A. 这段代码错在哪里？
- B. 给出函数的正确实现，只能使用左右移位和一个减法。

# 作业讲评

P91 2.71

A : 这段代码错在，当获取的 `unsigned int` 字节的最高位上为1的时候，转换后的 `int` 应为负数，但这一算法得到的结果却是正数。

B :

```
int xbyte(packed_t word, int bytenum) {
    // 先将word左移，使word最高位上是所要提取byte的最高位
    int shifted = (int)(word << ((3 - bytenum) << 3));
    // 再执行右移运算，高位自动填入1或0，实现正负
    return shifted >> 24;
}
```

# 作业讲评

P93 2.86

\*2.86 与 Intel 兼容的处理器也支持“扩展精度”浮点形式，这种格式具有 80 位字长，被分成 1 个符号

94 第一部分 程序结构和执行

位、 $k=15$  个阶码位、1 个单独的整数位和  $n=63$  个小数位。整数位是 IEEE 浮点表示中隐含位的显式副本。也就是说，对于规格化的值它等于 1，对于非规格化的值它等于 0。填写下表，给出用这种格式表示的一些“有趣的”数字的近似值。

描述	扩展精度	
	值	十进制
最小的正非规格化数		
最小的正规格化数		
最大的规格化数		

将数据类型声明为 `long double`，就可以把这种格式用于与 Intel 兼容的机器编译 C 程序。但是，它会强制编译器以传统的 8087 浮点指令为基础生成代码。由此产生的程序很可能会比数据类型为 `float` 或 `double` 的情况慢上许多。

# 作业讲评

P93 2.86

Description	Value	Decimal
Smallest positive denormalized	0x000000000000000000000001	$2^{-63+2-2^{14}}$
Smallest positive normalized	0x000180000000000000000000	$2^{2-2^{14}}$
Largest normalized	0x7FFEEEEEEEEEFFFFFFFFFF	$(2 - 2^{-63}) \times 2^{2^{14}-1}$

# 作业讲评

## P94 2.87

\* 2.87 2008 版 IEEE 浮点标准，即 IEEE 754-2008，包含了一种 16 位的“半精度”浮点格式。它最初是由计算机图形公司设计的，其存储的数据所需动态范围要高于 16 位整数可获得的范围。这种格式具有 1 个符号位、5 个阶码位( $k = 5$ )和 10 个小数位( $n = 10$ )。阶码偏置量是  $2^{5-1} - 1 = 15$ 。

对于每个给定的数，填写下表，其中，每一列具有如下指示说明：

Hex：描述编码形式的 4 个十六进制数字。

$M$ ：尾数的值。这应该是一个形如  $x$  或  $\frac{x}{y}$  的数，其中  $x$  是一个整数，而  $y$  是 2 的整数幂。例

如： $0$ 、 $\frac{67}{64}$  和  $\frac{1}{256}$ 。

$E$ ：阶码的整数值。

$V$ ：所表示的数字值。使用  $x$  或者  $x \times 2^z$  表示，其中  $x$  和  $z$  都是整数。

$D$ ：(可能近似的)数值，用 printf 的格式规范 %f 打印。

举一个例子，为了表示数  $\frac{7}{8}$ ，我们有  $s = 0$ ， $M = \frac{7}{4}$  和  $E = -1$ 。因此这个数的阶码字段为

$01110_2$ (十进制值  $15 - 1 = 14$ )，尾数字段为  $1100000000_2$ ，得到一个十六进制的表示  $3B00$ 。其数值为 0.875。

标记为“—”的条目不用填写。

描述	Hex	M	E	V	D
-0				-0	-0.0
最小的>2 的值					
512				512	512.0
最大的非规格化数					
$-\infty$		—	—	$-\infty$	$-\infty$
十六进制表示为 3B00 的数	3B00				

# 作业讲评

P94 2.87

	Hex.	M	E	V	D
-0	8000	0	-14	-0	-0.0
最大幅值 > 2 的 10 次方	4001	$\frac{1025}{1024}$	1	$\frac{1025}{512}$	2.001953125
512	6000	1	9	512	$512 \cdot 0$
最小非负数	03FF	$\frac{1023}{1024}$	-14	$\frac{1023}{2^{24}}$	0.000060975551605224
-∞	FC00	-	-	-∞	-∞
+∞	3B30	$\frac{123}{64}$	-1	$\frac{123}{128}$	0.9609375

# 习题试炼 1

2. 在采用小端法存储机器上运行下面的代码，输出的结果将会是？

(int,unsigned 为 32 位长,short 为 16 位长,0~9 的 ASCII 码分别是 0x30~0x39)

```
char *s = "2018";
int *p1 = (int *)s;
short s1 = (*p1) >> 12;
unsigned u1 = (unsigned) s1;
printf("0x%x\n", u1);
```

- A) 0x00002303
- B) 0x00032303
- C) 0xffff8313
- D) 0x00008313

# 习题试炼 1

答案：C

本题考查大小端存储，以及整数类型转化。字符串在存储时不区分大小端，前面的字符存储在低地址，而在被转化成整型的时候低地址被视为低位，因此\*p1 为 0x38313032，s1 为 0x8313。在由 short 转化为 unsigned 的时候，我们要先改变大小，之后完成有符号到无符号的转换，因此 u1 为 0xffff8313。

## 习题试炼 2

3. 运行下面的代码，输出结果是（其中 float 类型表示 IEEE-754 规定的浮点数，包括 1 位符号、8 位阶码和 23 位尾数）：

```
for(float f = 1;; f = f + 1)
{
    if(f + 1 - f != (float)1)
    {
        printf("%.0f\n", f);
        break;
    }
}
```

- A. 8388608 ( $=2^{23}$ )
- B. 16777216 ( $=2^{24}$ )
- C. 2147483647 ( $=2^{31}-1$ )
- D. 程序为死循环，没有输出

## 习题试炼 2

答案:B

解答: 这种 float 类型的 23 位尾数意味着其与 23 位及以内的整数可以建立一一对应的联系, 自然  $f + 1 - f == 1$  成立; 当  $f$  超过这一范围时,  $f + 1$  的运算结果会发生舍入, 因而无法恢复到之前的结果。随着  $f$  自增, 最小的超过精度范围的整数是  $2^{24}$ , 故可以得出选项。

# 习题试炼 3

4. 在 x86-64 机器上, 有如下的定义:

```
int x = _____;
int y = _____;
unsigned int ux = x;
unsigned int uy = y;
```

判断下列表达式是否等价:

(提示: 减法的运算优先级比按位异或高。布尔运算的结果都是有符号数。)

	表达式 A	表达式 B	等价吗?
(1)	$x > y$	$ux > uy$	Y N
(2)	$(x > 0) \    \ (x < ux)$	1	Y N
(3)	$x \wedge y \wedge x \wedge y \wedge x$	$x$	Y N
(4)	$((x >> 1) << 1) \leq x$	1	Y N
(5)	$((x / 2) * 2) \leq x$	1	Y N
(6)	$x \wedge y \wedge (\sim x) - y$	$y \wedge x \wedge (\sim y) - x$	Y N
(7)	$(x == 1) \ \&\ (ux - 2 < 2)$	$(x==1) \ \&\ ((\sim ux) - 2 < 2)$	Y N

# 习题试炼 3

提示：减法的运算优先级比按位异或高。布尔运算的结果都是有符号数。

	表达式 A	表达式 B	等价吗？
(1)	$x > y$	$ux > uy$	Y N
(2)	$(x > 0) \mid\mid (x < ux)$	1	Y N
(3)	$x \wedge y \wedge x \wedge y \wedge x$	$x$	Y N
(4)	$((x >> 1) << 1) \leq x$	1	Y N
(5)	$((x / 2) * 2) \leq x$	1	Y N
(6)	$x \wedge y \wedge (\sim x) - y$	$y \wedge x \wedge (\sim y) - x$	Y N
(7)	$(x == 1) \&\& (ux - 2 < 2)$	$(x==1) \&\& ((\sim ux) - 2 < 2)$	Y N

【答】(1) 取  $x=1, y=-1$  即不正确；(2) 取  $x=-1$  即不正确；(3) 正确，利用交换律、结合律，以及  $x \wedge x == 0$ ；(4) 正确，即使是对负数；(5) 不正确，负奇数该运算向 0 舍入；(6) 正确， $(\sim x) - y$  也就是  $(\sim x) + (\sim y) + 1$ ，注意运算优先级；(7) 不正确， $\sim ux$  是有符号数。

## 习题试炼 4

18. 若我们采用基于 IEEE 浮点格式的浮点数表示方法, 阶码字段  $\text{exp}$  占据  $k$  位, 小数字段  $\text{frac}$  占据  $n$  位, 则最大的非规格的正数是 \_\_\_\_\_ (结果用含有  $n, k$  的表达式表示)

## 习题试炼 4

答案:  $(1 - 2^{-n}) * 2^{-2^{k-1} + 2}$

解析: 阶码字段 exp: 00...00

小数字段 frac: 11...11

偏置值 bias:  $2^{k-1} - 1$

## 习题试炼 5

4. 关于浮点数，以下说法正确的是
- A. 给定任意浮点数  $a, b$  和  $x$ ，如果  $a > b$  成立（求值为 1），则一定  $a+x > b+x$  成立
  - B. 不考虑结果为 `NaN`、`Inf` 或运算过程发生溢出的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果
  - C. 不考虑输入为 `NaN`、`Inf` 的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果
  - D. 给定任意浮点数  $a, b$  和  $x$ ，如果  $a > b$  不成立（求值为 0），则一定  $a+x > b+x$  不成立。

## 习题试炼 5

答案: d。如果不出现 NaN 和 Inf, 浮点数是满足单调性的, 这时 a 和 d 都成立。如果出现 NaN 和 Inf, 那么任何时候比较运算的操作数有 NaN 和 Inf 就为 0, 就只有 d 成立。关于 b 和 c, 由于运算的偶然性, 高精度不一定比低精度精确。

# THANKS

Made by WalkerCH

[changxinhai@stu.pku.edu.cn](mailto:changxinhai@stu.pku.edu.cn)

Reference: [Weicheng Lin]'s presentation.

