CSAPPLab1 DataLab

这是一个关于机器级的整数、浮点数表示和位运算的实验。要求用给定的操作符、尽可能少的操作数去实现对应的函数功能。

第一题 bitAnd

```c
/*
* bitAnd - x&y using only ~ and |
*   Example: bitAnd(6, 5) = 4
*   Legal ops: ~ |
*   Max ops: 8
*   Rating: 1
*/
int bitAnd(int x, int y) {
    return ~(~x | ~y);
}
```

该题目要求使用"|"和"~"操作实现与运算("&")。联想到逻辑运算中的德摩根定律：$A\&B = \overline{\overline{A\&B}} = \overline{\overline{A}|\overline{B}}$，由此得出答案。

第二题 getByte

```c
/*
* getByte - Extract byte n from word x
*   Bytes numbered from 0 (LSB) to 3 (MSB)
*   Examples: getByte(0x12345678,1) = 0x56
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 6
*   Rating: 2
*/
int getByte(int x, int n) {
    return (x >> (n << 3)) & 0xFF;
}
```

该题目要求从一个字中取出相应的字节。思路很简单，就是将给定数字右移，使待取出字节位于最低位，与上0xFF即可。

第三题 logicalShift

```c
/*
* logicalShift - shift x to the right by n, using a logical shift
*   Can assume that 0 <= n <= 31
*   Examples: logicalShift(0x87654321,4) = 0x08765432
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 20
*   Rating: 3
*/
int logicalShift(int x, int n) {
    // 避免左移32的运算
    int i = 32 + (~n);
    return ((x >> n) & ((1 << i) + (~0) + (1 << i)));

    // 错误。输入(-2147483648[0x80000000],0[0x0])时，输出0[0x0]，实际应输出-2147483648[0x80000000]。
    // 原因：此时i=32，1<<i的值为1（可能采用循环移位，但直接代入数字1<<32值为0，注：VS中测试）而非0.
    int i = 33 + (~n);      // 32 - n
    return ((x >> n) & ((1 << i) + (~1) + 1));      // (x >> n) & (1 << i - 1)
}
```

该题目要求用逻辑移位操作将 x 向右移动 n 位。因对于有符号数计算机默认执行算数移位，此题应考虑到当 x 为负数的情况，如示例，对其逻辑移位最左端不补符号位。 思路为首先将 x 右移 n 位，然后与上一个最高 n 位为 0 ,其余位为 1 的数。

> - **得到低 n 位为 1 的数:** $(1 << n) - 1$ **或** $((1 << (n-1)) - 1 + ((1 << (n-1)))$
>
> 如：0x3F，此时 $n = 6$ 。
>
> > $1 << n \to 01000000b \xrightarrow{-1} 00111111b$ 。
> >
> > $1 << (n-1) \to 00100000b \xrightarrow{-1} 00011111 \xrightarrow{+(1<<(n-1))} 00111111b$

第四题 bitCount

```c
/*
* bitCount - returns count of number of 1's in word
```

```
 *   Examples: bitCount(5) = 2, bitCount(7) = 3
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 40
 *   Rating: 4
 */
int bitCount(int x) {
    int _mask1 = 0x55 + (0x55 << 8);
    int _mask2 = 0x33 + (0x33 << 8);
    int _mask3 = 0x0f + (0x0f << 8);
    int mask1 = _mask1 + (_mask1 << 16);
    int mask2 = _mask2 + (_mask2 << 16);
    int mask3 = _mask3 + (_mask3 << 16);
    int mask4 = 0xff + (0xff << 16);
    int mask5 = 0xff + (0xff << 8);
    x = x - ((x >> 1) & mask1);
    x = (x & mask2) + ((x >> 2) & mask2);
    x = ((x >> 4) + x) & mask3;
    x = ((x >> 8) + x) & mask4;
    x = ((x >> 16) + x) & mask5;
    return x;

    // 简短版本，但是题目要求只能使用0x00-0xff范围内数字。
    //x = x - ((x >> 1) & 0x55555555);
    //x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    //x = ((x >> 4) + x) & 0x0f0f0f0f;
    //x = ((x >> 8) + x) & 0x00ff00ff;
    //x = ((x >> 16) + x) & 0x0000ffff;
    //return x;
}
```

该题目要求给定数字二进制表示中的 1 的个数。通常的想法为通过循环操作判断每一位是否为 1 ，但题目要求不能采用循环操作。该问题的思路为首先计算相邻每 2 个 bit 位 1 的个数，然后计算相邻每 4 个 bit 位 1 的个数，接着是每 8 个、每 16 个，最后计算得到总的 32 个 bit 位的 1 个数。

> - **计算相邻每 2 个 bit 位 1 的个数**
>
> | v | x = (v>>1) & 0b01 | v - x （v中1的个数） |
> | --- | --- | --- |
> | 0b00 | 0b00 | 0b00 |
> | 0b01 | 0b00 | 0b01 |
> | 0b10 | 0b01 | 0b01 |
> | 0b11 | 0b01 | 0b10 |

例：x = 0x12345678 = 0b0001 0010 0011 0100 0101 0110 0111 1000

- x = x - ((x >> 1) & 0x55555555) ，计算相邻每 2 个 bit 位 1 的个数;

  x = 0x11245564 = 0b0001 0001 0010 0100 0101 0101 0110 0100
- x = (x & 0x33333333) + ((x >> 2) & 0x33333333)，求和计算每 4 个 bit 位 1 的个数;

  x = 0x11212231 = 0b0001 0001 0010 0001 0010 0010 0011 0001
- x = ((x >> 4) + x) & 0x0f0f0f0f，求和计算每 8 个 bit 位 1 的个数;

  x = 0x02030404 = 0b0000 0010 0000 0011 0000 0100 0000 0100
- x = ((x >> 8) + x) & 0x00ff00ff，求和计算每 16 个 bit 位 1 的个数;

x = 0x00050008 = 0b0000 0000 0000 0101 0000 0000 0000 1000

- x = ((x >> 16) + x) & 0x0000ffff，求和计算 32 个 bit 位 1 的个数；

  x = 0x0000000d = 0b0000 0000 0000 0000 0000 0000 0000 1101

参考：https://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bits-in-a-32-bit-integer

---

第五题 bang

```
/*
* bang - Compute !x without using !
*   Examples: bang(3) = 0, bang(0) = 1
*   Legal ops: ~ & ^ | + << >>
*   Max ops: 12
*   Rating: 4
*/
int bang(int x) {
    // 思路一
    int sign1 = x >> 31;
    int sign2 = ((~x) + 1) >> 31;
    return ((~sign1) & (~sign2)) & 0x01;

    // 思路二
    //int tmp = ~x + 1;
    //tmp = tmp | x;
    //tmp = tmp >> 31;
    //return tmp + 1;
}
```

该题目要求当 $x=0$ 时返回 1 ，其他情况返回 0 ，相当于判断输入是否为 0 。思路一为判断 x 和 $-x$ 的符号位。若两者符号位均为 0 ，则 $x=0$ ；若两者符号相异，则 $x \neq 0$ ；若两者符号位均为 1 ，则 $x=$ 0x80000000 。思路二为 x 和 $-x$ 相或，只要两者之一符号位为 1 ，右移 31 位后为$x=$ 0x $FFFFFFFF$ ，加1得 0 ，对于边界值 $x=$ 0x80000000 仍是如此。

第六题 tmin

```
/*
* tmin - return minimum two's complement integer
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 4
*   Rating: 1
*/
int tmin(void) {
    return 0x01 << 31;
}
```

该题目较为简单，返回最小的整数二进制补码值，即最高位为 1 ，其余位为 0 。

第七题 fitsBits

```c
/*
* fitsBits - return 1 if x can be represented as an
*  n-bit, two's complement integer.
*   1 <= n <= 32
*   Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 15
*   Rating: 2
*/
int fitsBits(int x, int n) {
    int tmp = (x << (32 - n)) >> (32 - n);
    return !(~tmp + 1 + x);
}
```

该题目要求判断给定值 x 是否可以用 n 位整数二进制补码表示。思路为将 x 左移 $32-n$ 位，即只保留 n 位对其表示，再右移回原来的位置，若两者相同，即差值为 0 ，则可以表示；否则原数值已经改变，说明不能用 n 位表示。

第八题 divpwr2

```c
/*
* divpwr2 - Compute x/(2^n), for 0 <= n <= 30
*  Round toward zero
*   Examples: divpwr2(15,1) = 7, divpwr2(-33,4) = -2
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 15
*   Rating: 2
*/
int divpwr2(int x, int n) {
    // 思路二
    int signmask = x >> 31;
    int mask = (1 << n) + (~1) + 1; // 取高位为0，低n位为1   (1<<n)-1
    return (x >> n) + (signmask & !!(x & mask));

    // 思路一 （错误。因存在特殊情况最小负数 x = 0x80000000，其绝对值仍为自身）
    //int mask = x >> 31;
    //x = (x + mask) ^ mask;    // 取绝对值
    //x = x >> n;
    //return (x + mask) ^ mask; // 代入原符号位
}
```

该题目要求采用“向零舍入”的方式计算 $x/2^n$ 的整数值。对于正数，直接将 x 右移 n 位即可；对于负数，若该负数不能被 2^n 整除，则右移 n 位后的值需加 1 。思路一为将 x 先转换为正值，右移 n 位后将结果代入相应符号位输出。思路二为直接对原数 x 右移 n 位，并判断 x 的低 n 位是否为 0 ，若不为 0 说明相除后有余数，且 x 为负数时，右移后的结果加 1 返回；其余情况下直接返回右移后的值即可。

> - **取整数x的绝对值**
>
> ```c
> int mask  = x >> 31;
> x = (x + mask) ^ mask;
> ```
>
> 如：对于 8 bit 位数据，若 x = 0b10011000 = -104
>
> mask = x >> 7 = 0b11111111

> (x + mask) ^ mask = (0b10011000 + 0b11111111) ^ 0b11111111 = 0b01101000 = 104
>
> - **取整数x的负数**
>
> ```
> int mask  = 0xffffffff;     // 注：当 mask=0 时，x值保持不变
> x = (x + mask) ^ mask;
> 或
> x = (~x) + 1;
> ```

第九题 negate

```
/*
* negate - return -x
*   Example: negate(1) = -1.
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 5
*   Rating: 2
*/
int negate(int x) {
    return ~x + 1;
}
```

该题目要求返回 x 的负数，较为简单。

第十题 isPositive

```
/*
* isPositive - return 1 if x > 0, return 0 otherwise
*   Example: isPositive(-1) = 0.
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 8
*   Rating: 3
*/
int isPositive(int x) {
    // 思路三
    return (!!x) & (!(x & (1 << 31)));

    // 思路二（错误）
    //x = ~x + 1;         // 数值为0x80000000时错误
    //return (x >> 31) & 0x01;

    // 思路一（错误）
    //return !(x & (1 << 31)); // 数值为0时错误
}
```

该题目要求判断输入是否为正值（$x > 0, return\ 1;\ x \leq 0, return\ 0$），较为简单。思路一为直接返回符号位的相反值，但存在特殊值 $x = 0$ 。思路二为直接返回 $-x$ 的符号位，但存在特殊值 $x = 0x80000000$ 。思路三为将特殊值和一般指分开处理。

第十一题 isLessOrEqual

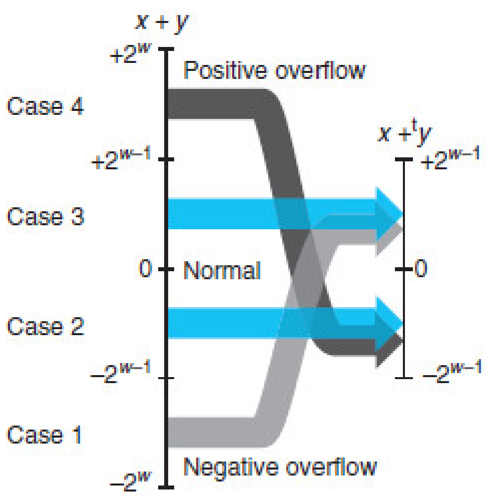
```
/*
```

```
 * isLessOrEqual - if x <= y  then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
int isLessOrEqual(int x, int y) {
    // 思路二
    int signx = (x >> 31) & 1;
    int signy = (y >> 31) & 1;
    int sign1 = signx & (!signy);
    int sign2 = (!(signx ^ signy)) & (((x + (~y)) >> 31) & 1);  // 此处 x<=y, 即 x-y<=0, return 1. 等价为 x-y-1<0, return 1. 进而可直接判断符号位返回。
    return sign1 | sign2;

    // 思路一（错误）
    // 输入-2147483648[0x80000000],2147483647[0x7fffffff]时错误  两个数直接加减会溢出，造成错误结果
    //int tmp = y + (~x + 1);
    //return (~(tmp >> 31)) & 0x00000001;
}
```

该题目要求判断输入两个值的大小关系。思路一为直接将两个值相减，然后判断符号位，但该种做法会出现值溢出的情况，不可取。因为同号相加、异号相减可能导致值溢出，如下图所示：

Figure 2.24
Relation between integer and two's-complement addition. When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than or equal to 2^{w-1}, there is a positive overflow.



思路二为应先判断符号位，若符号位不同，则直接可以判断大小关系；若符号位相同，则进行相减运算不会溢出，进而可以判断大小。如下表：

signx	signy	return
0	0	...
0	1	0
1	0	1
1	1	...

第十二题 ilog2

```
/*
```

```
* ilog2 - return floor(log base 2 of x), where x > 0
*   Example: ilog2(16) = 4
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 90
*   Rating: 4
*/
int ilog2(int x) {
    int bitNum = (!!(x >> 16)) << 4;
    bitNum = bitNum + ((!!(x >> (bitNum + 8))) << 3);
    bitNum = bitNum + ((!!(x >> (bitNum + 4))) << 2);
    bitNum = bitNum + ((!!(x >> (bitNum + 2))) << 1);
    bitNum = bitNum + (!!(x >> (bitNum + 1)));
    return bitNum;
}
```

该题目要求输入 x 的以 2 为底的整数次幂（向下取整），实则是求出输入 x 的二进制表示中 1 所处最高位的位置，如输入 $x = 0x00010111$，则输出应为 4。思路为二分法进行判断，首先判断高 16 位值是否为 0，若为 0，则最高位 1 存在于低 16 位，否则存在于高 16 位；然后继续再上次判断出的 16 位数据中判断最高位 1 存在于高 8 位或者是低 8 位；如此继续判断，最终返回最高位 1 的位置。

第十三题 float_neg

```
/*
* float_neg - Return bit-level equivalent of expression -f for
*   floating point argument f.
*   Both the argument and result are passed as unsigned int's, but
*   they are to be interpreted as the bit-level representations of
*   single-precision floating point values.
*   When argument is NaN, return argument.
*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
*   Max ops: 10
*   Rating: 2
*/
unsigned float_neg(unsigned uf) {
    unsigned result = uf ^ 0x80000000;
    unsigned tmp = uf & 0x7fffffff;
    if (tmp > 0x7f800000)
        return uf;
    return result;
}
```

该题目要求返回输入float类型值 uf 的负数，直接改变符号位即可，但应注意题目要求输入为 NaN 时，直接返回输入值，所以应加入输入值是否为 NaN 的判断。单精度浮点数数值的分类如下图所示：

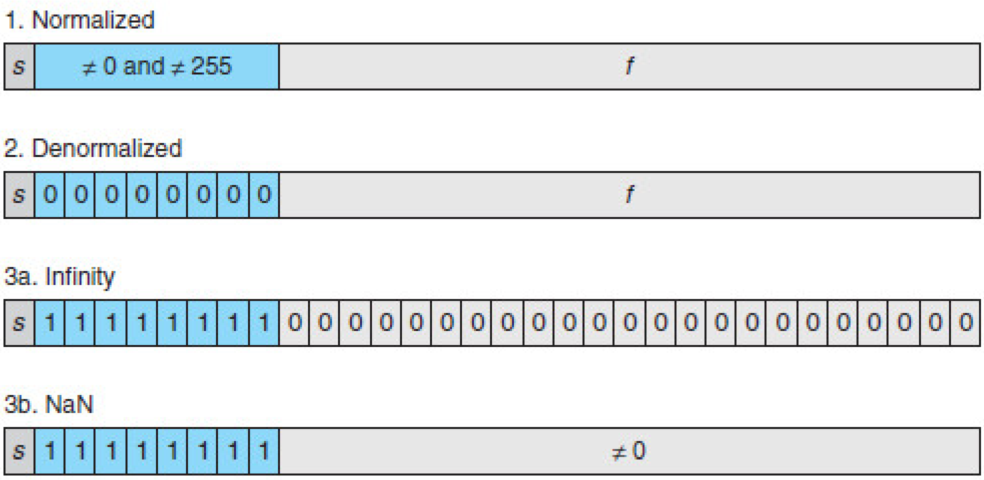


Figure 2.33 **Categories of single-precision floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

第十四题 float_i2f

```
/*
* float_i2f - Return bit-level equivalent of expression (float) x
*   Result is returned as unsigned int, but
*   it is to be interpreted as the bit-level representation of a
*   single-precision floating point values.
*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
*   Max ops: 30
*   Rating: 4
*/
unsigned float_i2f(int x) {
    int sign;
    int bitNum = 31;
    int bit;
    int tmp;
    if (x == 0)
        return 0;
    sign = x & 0x80000000;
    if (sign == 0x80000000)
        x = (~x) + 1;
    tmp = x;
    x = x << (31 - bitNum);
    bit = (x & 0x80) >> 7;  // 取进位位
    bitNum = bitNum + 127;
    if ((x & 0xff) == 0x80) {
        x = (x >> 8) & 0x007fffff;
        x = x + (x & 0x01);
    }
    else {
        x = (x >> 8) & 0x007fffff;
        x = x + bit;
    }
    return sign + (bitNum << 23) + x;
}
```

该题目要求将 int 类型数据转化为 $float$ 类型。标准浮点格式如下图所示：

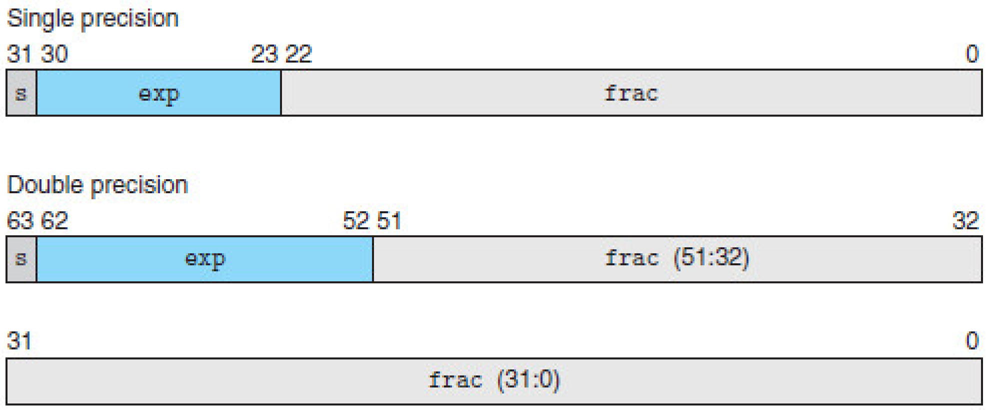


Figure 2.32 Standard floating-point formats. Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

相关内容见书籍《深入理解计算机系统》第78页。同时应注意表示成浮点数的舍入问题（书籍83页），默认采用向偶数舍入的方式。例如，假设将十进制数舍入到最接近的百分位，则 1.2349999 舍入到 1.23， 1.2350001 舍入到 1.24， 而 1.2350000 和 1.2450000 都舍入到 1.24， 因为它们分别位于 1.23 和 1.24 及 1.24 和 1.25 的正中间，而 4 是偶数。解题思路为先记下符号位，然后将负值转换为正值进行相应的变换，参照书籍82页。需要注意的是舍入操作时的几种情况（对于单精度浮点数）：a、输入值得绝对值 $|x|$ 转换为二进制后的位数 $n \leq 23$ （不包含首位 1， 下同），无舍入情况，直接转换；b、若 $n > 23$， 舍去的位数最高位为 0， 直接舍去不进位；c、若 $n > 23$， 舍去的位数最高位为 1， 其余位全为 0， 即形如 $0b100000$ 的形式，需判断保留的最低位是否为 0， 若为 0， 说明为偶数，直接舍去不进位，若为 1， 说明为奇数，需进位（即加 1）； d、若 $n > 23$， 舍去的位数最高位为 1， 其余位不全为 0， 即形如 $0b100110$ 的形式，直接进位。

第十五题 float_twice

```
/*
* float_twice - Return bit-level equivalent of expression 2*f for
*   floating point argument f.
*   Both the argument and result are passed as unsigned int's, but
*   they are to be interpreted as the bit-level representation of
*   single-precision floating point values.
*   When argument is NaN, return argument
*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
*   Max ops: 30
*   Rating: 4
*/
unsigned float_twice(unsigned uf) {
    int flag = uf & 0x7f800000;
    if ((flag == 0x7f800000))
        return uf;
    unsigned tmpf = (uf & 0x007fffff) << (!flag);
    unsigned tmpe = (((uf & 0x7f800000) >> 23) + (!!flag)) & 0xff;
    return (uf & 0x80000000) + (tmpe << 23) + tmpf;
}
```

该题目要求输入 $float$ 值 uf 的 2 倍，需要考虑不同的情况，再次贴上单精度浮点数数值的分类图：

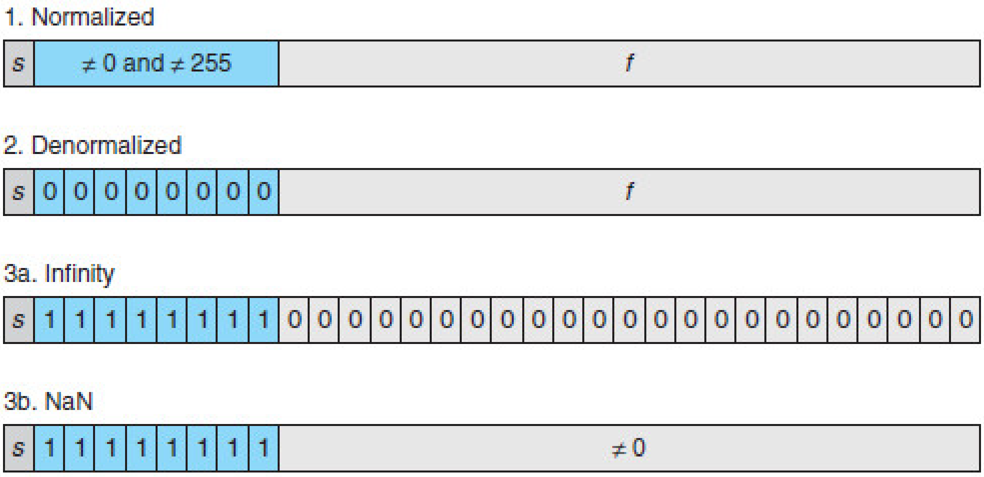


Figure 2.33 Categories of single-precision floating-point values. The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

思路如下：a、对于 $Normalized$ 的情况，乘以 2 相当于阶码位(exp)加 1 ，尾数位($frac$)不变；

b、对于 $Denormalized$ 的情况，乘以 2 相当于尾数位($frac$)左移 1 位，若有溢出阶码位(exp)加 1 ，否则阶码位(exp)不变；并且该种情况中包含除符号位外全为 0 的特殊情况（即输入为 0），对此上述操作同样适用。

c、对于 $Infinity$ 和 NaN 的情况，直接返回输入值。