华东师范大学软件学院实验报告

实验课程：计算机系统　　**年级：**23 级本科　　**实验成绩：**

实验名称：Labs (1) Bits.c　　**姓名：**张梓卫

实验编号：（1）　　**学号：**10235101526　　**实验日期：**2024/03/09

指导教师：肖波　　**组号：**

一、实验目的

掌握 Linux 系统的基本操作，掌握位运算及基本的机器语言中的底层运算。

二、实验过程与分析

1、BitAnd：

根据布尔运算中的德摩根定律 **A&B = !(!A | !B)**，在限制条件下（仅可使用按位取反、按位取或，这两个运算符的情况下），无更优解。

```
1、int bitAnd(int x, int y) {
    return ~ ( ~x | ~y );
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f bitAnd
Score   Rating  Errors  Function
 1      1       0       bitAnd
Total points: 1/1
```

2、getByte：

根据题目说明：**Bytes numbered from 0 (LSB) to 3 (MSB)**，这是一个以大端法存储的方式（一个字节以十六进制的方式表示为 2 位，大端法是从左到右的地址顺序）

使用经典方法：获取最低字节：与想要得到的字节数量的十六进制数相与，即可将其他位置零。如：**101...0101 1010 1000 & 00...0000 1111 1111 (0xFF) = 00...00 1010 1000**

通过相似的思路，我们可以先将 **N << 3**（即 **N * 8**，因为每个字节占 8 位），**X** 进行逻辑右移 **N** 个字节，那么即可将想要的字节放到最右端，相与得到的即为结果。

```
int getByte(int x, int n) {
    return ( x >> ( n << 3 ) ) & 0xff ;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f getByte
Score   Rating  Errors  Function
 2      2       0       getByte
Total points: 2/2
```

3、logicalShift：

题目要求：进行逻辑右移。

注意区分：逻辑右移（只添加 0）和算术右移（与最高位保持一致）

看到题目时，信心满满地使用了：x >> n

```
int logicalShift(int x, int n) {
    return (x >> n);
}
```

突然意识到，计算机中默认是进行算术右移而不是逻辑右移的（非无符号整数）我们的目标应该使得右移 n 位，添加的全为 0，故进行右移后，将前面移动的 n 位全部置 0 即可（这一步操作构造一个前 n 位是 0，后面都是 1 的数相与即可）

问题来到了如何构造如同 00001111 的数：

首先我们有 **0001111 & 1100** 的例子：

```
   00000000000000000000000000001111
&  00000000000000000000000000001100
 ---------------------------------
   00000000000000000000000000001100
```

我们知道，当不是相同位数的二进制数相与时，系统会自动添加 0 补全数字，那么，我们考虑类似的操作：

先将 1 < < 31 得到 1 00..00 （31 个 0）；

然后右移 n 位（算术右移），得到 11.. 1 00.00 （n+1 个 1，31-n 个 0）；

再左移一位取反，有 00.. 00 11..11（n 个 0，32-n 个 1）；

代码如下：

```
int logicalShift(int x, int n) {
    int AlgoRightShift = x >> n;
    int step1 = (1 << 31) >> n;
    int step2 = step1 << 1;
    int step3 = ~step2;
    return AlgoRightShift & step3;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f logicalShift
Score   Rating  Errors  Function
 3      3       0       logicalShift
Total points: 3/3
```

4、bitCount：

题目要求：返回一个数的二进制中的 1 的个数。

首先想到的思路：定义 Count = 0；for 循环中嵌套 if ((x>> 1) & (0x01))判断，若成立，则 Count = Count + 1；但题目中不允许使用循环和判断条件。

考虑手拆循环，要写 31 位.....这道题实在是太难了，参考了别人的代码，加入自己的理解，使用分治的思想（分组处理），对以下代码进行了解析：

汉明权重算法，基本思想是不断地将最低位 1 去掉，直到数字变为 0 为止，去掉一次，Count + 1.

```
int bitCount(int x) {
    int Mask1 = 0x11111111; // 使用 32 位全 1 的掩码常数
    int temp = x & Mask1; // 取 x 的低 32 位
    temp += x>>1 & Mask1; // 取 x 右移 1 位后的低 32 位，然后与 Mask 进行按位与，再加到
temp 上
    temp += x>>2 & Mask1; // 同上，右移 2 位
    temp += x>>3 & Mask1; // 同上，右移 3 位
    temp = temp + (temp >> 16); // 将 temp 高 16 位与低 16 位相加，结果保存在低 16 位

    int Mask2 = 0x0F0F;
    temp = (temp & Mask2) + ((temp >> 4) & Mask2); // 取 temp 的低 16 位和高 16 位的四
位和
    return (temp + (temp>>8)) & 0x3F; // 返回 temp 和 temp 右移 8 位的和，并且只保留低
6 位
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f bitCount
Score   Rating  Errors  Function
 4      4       0       bitCount
Total points: 4/4
```

5、bang：

题目要求：只有全零时，返回 1，其他情况都返回 0.

考虑到只有输入为 0 的情况才返回 1，那么考虑 0 的性质，作为最特别的判断条件，就是“0 的相反数是它本身”，再根据相反数 = 补码 + 1，

若一个数取反后，有一个二进制位不为 0，那么就会返回 0，即可写出下面的程序：

```
int bang(int x) {
    int Step1 = (x | (~x) + 1);
    int Step2 = Step1 >> 31;
    int Step3 = ~Step2;
    return Step3 & 1;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f bang
Score   Rating  Errors  Function
 4      4       0       bang
Total points: 4/4
```

6、tmin：

题目要求：返回 Tmin，送分题。

根据前段时间的学习，Tmin = 100..000 (31 位 0)，因为第一位是负数加权的，除第一位外，后面的都是正数加权，所以补码的最小值不是 Tmin = 1111...111（32 位），这个值实

际上是 -1 .

```
int tmin(void) {
    return 1 << 31;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f tmin
Score   Rating  Errors  Function
 1      1       0       tmin
Total points: 1/1
```

7、fitsBits：

题目要求：判断一个数是否能用 N 位的补码表示：

参考资料：二进制负数左侧有无数个 1，正数左侧有无数个 0；对于一个简单的数据：例如 0011，显然不能用 2 位的补码表示，因为有一位是符号位，但可以用 3 位的补码表示，所以左移 2 位，再算术右移回来，就会变成 1111，但如果左移 1 位，回来和之前是一样的，利用这个特性，可以有以下代码：

```
int fitsBits(int x, int n) {
    int Operation = 33 + ~n; // 32 - n;
    int Move = ((x << Operation) >> Operation);
    int Equal = Move ^ x; //如果和原来不一样，则返回 1
    return !Equal ;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f fitsBits
Score   Rating  Errors  Function
ERROR: Test fitsBits(-2147483648[0x80000000],32[0x20]) failed...
...Gives 1[0x1]. Should be 0[0x0]
Total points: 0/2
```

这题无论如何都不对，查找了相关资料编译运行也是报错的，考虑可能是因为 32 位系统或 64 位系统的差异导致的。

8、divPwr2：

题目要求：计算 x/(2^n)，Round toward zero

分析过程：注意到 2^n 实际就是右移操作，但是移位的结果由于是 int 类型定义的，故一定都是向下取整，题目要求向零取整，正数范围内已经实现了，所以现在要考虑负数范围内如何实现向零取整。

var = (1 << n) + (~0);：首先将 1 左移 n 位（即 2 的 n 次方），然后将结果与全 1 取反后相加，得到一个二进制表示的 n 位全 1 的数，存储到 var 变量中。

return (x + (sign & var)) >> n;：将输入的整数 x 与 (sign & var) 进行按位与运算，得到的结果再右移 n 位，即完成了对 x 除以 2 的^n 次幂的操作。具体步骤如下：

如果 x 是正数，(sign & var) 的结果为 0，相当于 x 不变，直接右移 n 位；

如果 x 是负数，(sign & var) 的结果为 var，即一个二进制表示的 n 位全 1 的数，相当于给 x 加上一个值（偏置值）然后再右移 n 位，实现了对负数进行向零舍入的操作。

```
int divpwr2(int x, int n) {
    int sign = 0, var = 0;
    sign = x >> 31;
    var = (1 << n) + (~0);
    return (x + (sign & var)) >> n;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f divpwr2
Score   Rating  Errors  Function
 2      2       0       divpwr2
Total points: 2/2
```

9、negate：

题目要求：取相反数。

取补码 +1 即可，应该无更优解。

```
int negate(int x) {
    return ( ~ x ) + 1;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f negate
Score   Rating  Errors  Function
 2      2       0       negate
Total points: 2/2
```

10、isPositive：

题目要求：判断 X 是否大于 0.

取符号位，是 1 就是负数，最初选择了 return ! ((x >> 31) & 1);

Unix > ./btest -f isPositive 后得分为 0，发现只判断了符号，但没有考虑可能有刚好为 0 的情况，故将代码修改为 return !((x >> 31) & 1) & x;

```
int isPositive(int x) {
    int SignX = (x >> 31) & 1;
    return !((SignX) | !x);
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f isPositive
Score   Rating  Errors  Function
 3      3       0       isPositive
Total points: 3/3
```

注意：上述过程，不能再使用布尔代数进行化简：(A + B')' = A' • B，因为这里的与符号不是逻辑运算符（&&），而是位运算符（&）。

11、isLessOrEqual：

题目要求：判断是否 x == y 或 x < y.

分析过程：比较大小，优先作差法和作商法，我们知道在计算机的二进制计算中，减法其实是由加法取代的。首先要知道经典操作 x == y 等价于 ！(x ^ y), 另外允许的符号中只有加号，那么做差法比较大小可以取（补码+1）。

根据 x 和 y 的对称性，我们不妨就判断 x - y < 0 是否成立，即判断 x + (-y) < 0 是否成立，根据上述函数 isPositive(int x)，我们可以知道当 x >> 31 取符号位时，便能判断结果的正负，则

令 Flag1 = !(x ^ y) ; Res = x + ((~y) + 1), 若 Res >> 31 为 1，则说明结果为负数，即 x - y < 0 成立，此时应该返回 1，由此有以下的判断式子 Flags1 | (Res >> 31)，化简得如下表达式：

```
int isLessOrEqual(int x, int y) {
    return ( !(x ^ y)) | ((x + ((~ y) + 1)) >> 31);
}
```

```
...
...Gives 0[0x0]. Should be 1[0x1]
Total points: 0/3
```

结果出错，检查，认为前一个判断没问题，问题一定出在后面的 x + (-y) 里，可能存在溢出的情况。考虑极端情况：当 x → INT_MAX ，Y → INT_MIN 时，显然会发生溢出，这种情况是由于 X 和 Y 异号引起的，显然，当 X 和 Y 同号时的相减判断才有必要，异号时，负数显然为最小的数，由此我们考虑如下代码：

```
int SignX = (x >> 31) & 1 ;
int SignY = (y >> 31) & 1 ;
X < Y 时，异号的情况：return ( SignX & (!SignY) ) ;
X < Y 时，同号的情况：return ( (SignX & SignY) & (((x + ((~ y) + 1)) >> 31))) ;
```

注意到符号的判断实际上是重复的操作，为增强可读性，将代码修改如下所示：

解释：先判断是否相等，然后判断是否异号，若 X 为负数，则返回 1，若同号，则返回 Res.运行结果如下图所示：

```
int isLessOrEqual(int x, int y) {
    int SignX = (x >> 31) & 1 ;
    int SignY = (y >> 31) & 1 ;
    int isSameSign = ! (SignX ^ SignY);
    int isEqual = !(x ^ y);
    int Res = (x + ((~y) + 1)) >> 31;
    return ( isEqual ) | (  (!( isSameSign ) & SignX ) | ( isSameSign & Res )  );
    }
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f isLessOrEqual
Score   Rating  Errors  Function
 3      3       0       isLessOrEqual
Total points: 3/3
```

12、ilog2：

题目要求：求 log(2)(x) = ?

分析过程：找到最高位的 1 所在的位置即可，优先取高位。线性查找可以简化为二分查找。

引入一个新操作：缩位或（转成 Bool 类型）————！运算的缩位特性

!x = 1 当且仅当 x=0；否则 x = 1。，那么 x 不为 0 时，!!x = 1

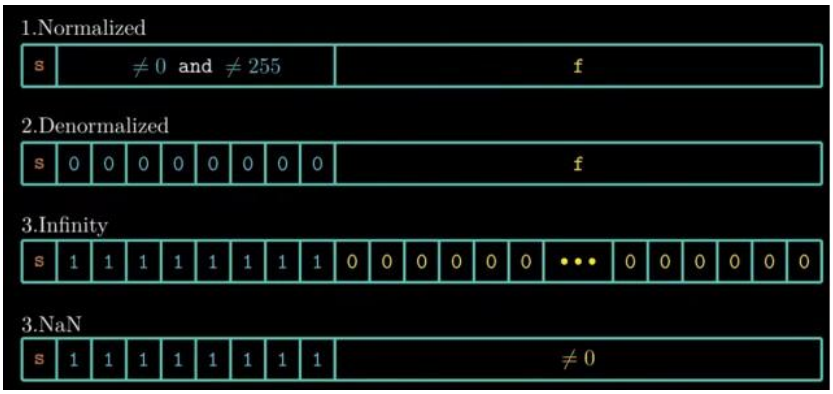
```
int ilog2(int x) { //二分查找
    int Tmp = 0;
    Tmp = (!!(x >> 16)) << 4; // 先查找左边 16 位中是否全为 0，如果为 0，就不动，如果有 1，就回到左边 16 位中继续找
    Tmp = Tmp + ((!!(x >> (Tmp + 8))) << 3);    // 再判断 8 位（即 17 - 24 位），同理
    Tmp = Tmp + ((!!(x >> (Tmp + 4))) << 2);    // 再判断 4 位（即 25 - 28 位），同理
    Tmp = Tmp + ((!!(x >> (Tmp + 2))) << 1);    // 再判断 2 位（即 29 - 31 位），同理
    Tmp = Tmp + (!!(x >> (Tmp + 1)));           // 再判断最后 1 位（即第 32 位），最后返回 Tmp 的值
    return Tmp; // 该值则是最先出现的 1 的位置
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f ilog2
Score   Rating  Errors  Function
 4      4       0       ilog2
Total points: 4/4
```

13、float_neg：

题目要求：返回 Float 类型的浮点数 -f 的等效位级别，若为 NaN，则返回传入的值

分析过程：首先要理解 Float 类型：1（符号）+8（指数）+23（尾数），当属于 NaN 时，低 23 位尾数不为 0，且从 24 位到 31 位均为 1，此时要作判断返回传入的数。



代码实现如下：

这里使用 1000..000（31 个 0）和原来的数进行异或操作，

```
unsigned float_neg(unsigned uf) {
    unsigned M = (1 << 23) - 1; // 得到 11..111(23 个 1)，用来判断小数字段是否不为 0
    unsigned Exp = 0xFF << 23; // 得到阶码全为 1 的数，左移是逻辑左移填充 0，得到 1111
1111 0000...000（23 位 0）
    unsigned Checkpoint1 = (uf & M);  // 将尾数置零，是否小数字段不为 0
    unsigned Checkpoint2 = (uf & Exp) == Exp; // 检查是否阶码部分都是 1
    if ( Checkpoint2 && Checkpoint1 ) { // 若成立，返回 Unsigned Float Value
        return uf;
    }
    // 否则直接取反
    return (1 << 31)^uf;
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f float_neg
Score   Rating  Errors  Function
 2      2       0       float_neg
Total points: 2/2
```

14、float_i2f：

题目要求：把 int 类型的数转换为 float 表示。

分析过程：首先要明白 Float 类型是如何表示的，在上面的题目中有。

```
unsigned float_i2f(int x) {
    unsigned Answer;
    int Frac = 0;     // 尾数的小数部分
    int Delta = 0; // 用于进行舍入运算
    int Tail = 0;  // 存储尾数
    int FloatExp = 0;     // 存储指数

    // 优先进行特殊情况处理
    if (x == 0) return 0;  // 如果输入为 0，则直接返回 0
    if (x == 0x80000000) return 0xcf000000;  // 如果输入为 0x80000000，则返回特定的值

    Answer = x & 0x80000000; // 提取符号位,此时 Answer 变量仍只表示符号位
    if (Answer) x = -x; //若 Answer 不为 0，则返回负数（取绝对值）

    // 题目不给用 for 循环，那就用 while，使得 x 一直移动到 0 时结束循环也可以。
    while ((x >> FloatExp)) FloatExp++;  // 计算出指数 E
    FloatExp = FloatExp - 1;  // 修正，因为循环条件导致的会多一个。

    x = x << (31 - FloatExp);  // 将整数左移，得到尾数和舍入位
    Tail = (x >> 8) & 0x007FFFFF; // 获取尾数的 23 位
    Frac = x & 0xFF; // 获取被舍入的部分

    // 判断是否需要向上舍入
    Delta = (Frac > 128) || ((Frac == 128) && (Tail & 1));
    Tail += Delta;
    FloatExp = FloatExp + 127; // 按照浮点数的表示规则计算阶码的值

    // 检查尾数是否溢出
    if (Tail >> 23) {
```

```
        Tail = Tail & 0x007FFFFF;
        FloatExp += 1;
    }

    // 将符号位、阶码和尾数合并得到最终结果
    Answer = Answer | FloatExp << 23 | Tail;
    return Answer;  // 返回转换后的单精度浮点数表示
}
```

```
Score = 61/71 [35/41 Corr + 26/30 Perf] (127 total operators)
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f float_i2f
Score   Rating  Errors  Function
 4      4       0       float_i2f
Total points: 4/4
```

15、float_twice：

题目要求：将传入的 float 类型乘以 2，特殊情况要特殊处理。

题目分析：根据 13 中的 Float 浮点数的知识，判断指数部分较为重要：

如果指数部分不为 0，uf 是规格化数，但此时要判断是否为 NaN 或者无穷，如果不是，则将 uf 中的指数部分加 1，即乘以 2。

如果指数部分为 0，说明 uf 为非规格化数或者 0，此时将 uf 左移一位，然后加上之前提取的符号位，即乘以 2。

```
unsigned float_twice(unsigned uf) {

    unsigned SignFloat = uf & 0x80000000; //提取符号位
    unsigned ExpFloat = uf & 0x7F800000; //提取阶码位，掩码 0x7F800000 ：表示阶码全为 1
    unsigned FracFloat = uf & 0x007FFFFF; //提取尾数位
    if (ExpFloat == 0) { //如果阶码位全为 0，说明是非规格化数
        uf = ( FracFloat << 1 ) | SignFloat; // 尾数不变左移一位，再加上符号位。
    } else if ( ExpFloat != 0x7F800000) { // 如果阶码不全为 1，则在阶码部分加上 1，即可
        uf = uf + 0x800000;
    }
    return uf;
}
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest -f float_twice
Score   Rating  Errors  Function
 4      4       0       float_twice
Total points: 4/4
```

三、实验结果总结

在实验过程中，增进了对位运算、int 类型整数、float 类型的浮点数在计算机中的表示方法，不足的是，有部分题目较难，例如 bitCount 的分治解法是第一次见，增长了见识，同时学会了 Vim 编辑模式的简单使用，初步了解了 Linux 系统编程。

最难的是 bitCount，ilog2，float_i2f，这些都是在刚学习时自我考虑不出来的。浮点数的边界情况考虑，运算的溢出情况.....

以下是跑分截图：

```
5. Running './dlc -e' to get operator count of each function.

Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       4       bitAnd
2       2       0       2       3       getByte
3       3       0       2       6       logicalShift
4       4       0       2       25      bitCount
4       4       0       2       6       bang
1       1       0       2       1       tmin
0       2       1       0       6       fitsBits
2       2       0       2       7       divpwr2
2       2       0       2       2       negate
3       3       0       2       5       isPositive
3       3       0       2       17      isLessOrEqual
4       4       0       2       27      ilog2
2       2       0       2       9       float_neg
4       4       0       2       25      float_i2f
4       4       0       2       8       float_twice

Score = 67/71 [39/41 Corr + 28/30 Perf] (151 total operators)
deralive@10235101526:~/Test1/datalab-handout$
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitAnd
 2      2       0       getByte
 3      3       0       logicalShift
 4      4       0       bitCount
 4      4       0       bang
 1      1       0       tmin
ERROR: Test fitsBits(-2147483648[0x80000000],32[0x20]) failed...
...Gives 1[0x1]. Should be 0[0x0]
 2      2       0       divpwr2
 2      2       0       negate
 3      3       0       isPositive
 3      3       0       isLessOrEqual
 4      4       0       ilog2
 2      2       0       float_neg
 4      4       0       float_i2f
 4      4       0       float_twice
Total points: 39/41
```

```
deralive@10235101526:~/Test1/datalab-handout$ ./dlc -e bits.c
dlc:bits.c:11:bitAnd: 4 operators
dlc:bits.c:25:getByte: 3 operators
dlc:bits.c:41:logicalShift: 6 operators
dlc:bits.c:64:bitCount: 25 operators
bits.c:75: Warning: suggest parentheses around arithmetic in operand of |
dlc:bits.c:79:bang: 6 operators
dlc:bits.c:90:tmin: 1 operators
dlc:bits.c:106:fitsBits: 6 operators
dlc:bits.c:121:divpwr2: 7 operators
dlc:bits.c:133:negate: 2 operators
dlc:bits.c:146:isPositive: 5 operators
dlc:bits.c:162:isLessOrEqual: 17 operators
dlc:bits.c:179:ilog2: 27 operators
dlc:bits.c:202:float_neg: 9 operators
dlc:bits.c:250:float_i2f: 25 operators
dlc:bits.c:275:float_twice: 8 operators

Compilation Successful (1 warning)
```

道阻且长。

四、附录：

Bits.c 源代码：

```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) {
    //根据布尔运算中的德摩根定律 A&B = !(!A | !B), 在限制条件下，无更优解
    return ~ ( ~x | ~y );
}

/*
 * 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) {
    // LSB 指的是最右端
    // 按照大端法，由于每个字节占 8 位，将 X 右移使得目标位到达最低位，与 0xff = 1111 1111 相与，即可将其他位置零
    return ( x >> ( n << 3 ) ) & 0xff ;
}

/*
 * logicalShift - shift x to the right by n, using a logical shift
 *   Can astempe that 0 <= n <= 31
 *   Examples: logicalShift(0x87654321,4) = 0x08765432
```

```
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int logicalShift(int x, int n) {
    int AlgoRightShift = x >> n;
    int step1 = (1 << 31) >> n;
    int step2 = step1 << 1;
    int step3 = ~step2;
    return AlgoRightShift & step3;
}

/*
 * 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 m1 = 0x11 | (0x11 << 8);
    int Mask = m1 | (m1 << 16);
    int temp = x & Mask;
    temp += (x >> 1) & Mask;
    temp += (x >> 2) & Mask;
    temp += (x >> 3) & Mask;

    temp += (temp >> 16);

    Mask = (0xF << 8) | 0xF;
    temp = (temp & Mask) + ((temp >> 4) & Mask);
    temp = ((temp >> 8) + temp) & 0x3F;
    return temp;
}

/*
 * bang - Compute !x without using !
 *   Examples: bang(3) = 0, bang(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
int bang(int x) {
    // //只有输入为 0 的情况会进行特别处理，那么考虑 0 的相反数 是否是它本身
    int Step1 = (x | (~x) + 1);
    int Step2 = Step1 >> 31;
    int Step3 = ~Step2; //右移得到符号位，与 1 相与，若符号位为 0，说明原码即为 0，否则返回 1.
    return Step3 & 1;
}

/*
 * tmin - return minimum two'temp complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
    //  补码的最小值：10000.....
    return 1 << 31;
}

/*
 * fitsBits - return 1 if x can be represented as an
 *  n-bit, two'temp 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 Operation = 33 + ~n; // 32 - n;
    int Move = ((x << Operation) >> Operation);
    int Equal = Move ^ x; //如果和原来不一样，则返回 1
    return !Equal ;
}

/*
 * 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) {
    int sign = 0, var = 0;
    sign = x >> 31;
    var = (1 << n) + (~0);
    return (x + (sign & var)) >> n; // 这一步是判断正数 Or 负数，若是正数，就直接右移 n 位，若是负数则加
上了偏置值。
}

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

/*
 * isPositive - return 1 if x > 0, return 0 otherwise
 *   Example: isPositive(-1) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 3
 */
int isPositive(int x) {
    // 取符号位，是 1 就是负数
    int SignX = (x >> 31) & 1;
    return !((SignX) | !x);
}

/*
 * 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) {
    int SignX = (x >> 31) & 1 ;
    int SignY = (y >> 31) & 1 ;
    int isSameSign = ! (SignX ^ SignY);
    int isEqual = !(x ^ y);
    int Res = (x + ((~y) + 1)) >> 31;
```

```
    return ( isEqual ) | (  (!( isSameSign ) & SignX ) | ( isSameSign & Res )  );
}

/*
 * 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) { //二分查找
    int Tmp = 0;
    Tmp = (!!(x >> 16)) << 4; // 先查找左边 16 位中是否全为 0，如果为 0，就不动，如果有 1，就回到左边 16 位中继续找
    Tmp = Tmp + ((!!(x >> (Tmp + 8))) << 3);    // 再判断 8 位（即 17 - 24 位），同理
    Tmp = Tmp + ((!!(x >> (Tmp + 4))) << 2);    // 再判断 4 位（即 25 - 28 位），同理
    Tmp = Tmp + ((!!(x >> (Tmp + 2))) << 1);    // 再判断 2 位（即 29 - 31 位），同理
    Tmp = Tmp + (!!(x >> (Tmp + 1)));           // 再判断最后 1 位（即第 32 位），最后返回 Tmp 的值
    return Tmp; // 该值则是最先出现的 1 的位置
}

/*
 * float_neg - Return bit-level equivalent of expression -f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned int'temp, 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 M = (1 << 23) - 1; // 得到 11..111(23 个 1)，用来判断小数字段是否不为 0
    unsigned Exp = 0xFF << 23; // 得到阶码全为 1 的数，左移是逻辑左移填充 0，得到 1111 1111 0000...000（23 位 0）
    unsigned Checkpoint1 = (uf & M);  // 将尾数置零，是否小数字段不为 0
    unsigned Checkpoint2 = (uf & Exp) == Exp; // 检查是否阶码部分都是 1
    if ( Checkpoint2 && Checkpoint1 ) { // 若成立，返回 Unsigned Float Value
        return uf;
    }
    // 否则直接取反
    return (1 << 31)^uf;
}
```

```
/*
 * 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) {
    unsigned Answer;
    int Frac = 0;     // 尾数的小数部分
    int Delta = 0; // 用于进行舍入运算
    int Tail = 0;  // 存储尾数
    int FloatExp = 0;     // 存储指数

    // 优先进行特殊情况处理
    if (x == 0) return 0;  // 如果输入为 0，则直接返回 0
```

```
    if (x == 0x80000000) return 0xcf000000;  // 如果输入为 0x80000000，则返回特定的值

    Answer = x & 0x80000000; // 提取符号位,此时 Answer 变量仍只表示符号位
    if (Answer) x = -x; //若 Answer 不为 0，则返回负数（取绝对值）

    // 题目不给用 for 循环，那就用 while，使得 x 一直移动到 0 时结束循环也可以。
    while ((x >> FloatExp)) FloatExp++;  // 计算出指数 E
    FloatExp = FloatExp - 1;  // 修正，因为循环条件导致的会多一个。

    x = x << (31 - FloatExp);  // 将整数左移，得到尾数和舍入位
    Tail = (x >> 8) & 0x007FFFFF; // 获取尾数的 23 位
    Frac = x & 0xFF; // 获取被舍入的部分

    // 判断是否需要向上舍入
    Delta = (Frac > 128) || ((Frac == 128) && (Tail & 1));
    Tail += Delta;

    FloatExp = FloatExp + 127; // 按照浮点数的表示规则计算阶码的值

    // 检查尾数是否溢出
    if (Tail >> 23) {
        Tail = Tail & 0x007FFFFF;
        FloatExp += 1;
    }

    // 将符号位、阶码和尾数合并得到最终结果
    Answer = Answer | FloatExp << 23 | Tail;
    return Answer;  // 返回转换后的单精度浮点数表示
}

/*
 * 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'temp, 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) {

    unsigned SignFloat = uf & 0x80000000; //提取符号位
    unsigned ExpFloat = uf & 0x7F800000; //提取阶码位，掩码 0x7F800000 ：表示阶码全为 1
    unsigned FracFloat = uf & 0x007FFFFF; //提取尾数位

    if (ExpFloat == 0) { //如果阶码位全为 0，说明是非规格化数
        uf = ( FracFloat << 1 ) | SignFloat; // 尾数不变左移一位，再加上符号位。
    } else if ( ExpFloat != 0x7F800000) { // 如果阶码不全为 1，则在阶码部分加上 1，即可
        uf = uf + 0x800000;
    }
    return uf;
}
```