datalab 报告

姓名：吴政坤

学号：2023202317

总分	bitXor	logtwo	byteSwap	reverse	...
36	1	4	4	3	…

test 截图：

解题报告

1.bitXor

`~(~x & ~y)`： `x` 和 `y` 中至少有一个为1的位置。

`~(x & y)`： `x` 和 `y` 中至少有一个为0的位置。

`~(~x & ~y) & ~(x & y)`： `x` 和 `y` 中一个为0，另一个为1的位置，这正是异或操作的定义。

2.samesign

0代表正，1代表负

先判断特殊情况：（0，0）返回1；(0,1)=0,(0,0)=1

一般情况下，同符号返回1，不同返回0；

(!x & !y) 判断x、y是否均为0

((x ^ 0) && (y ^ 0))判断x、y中是否有且只有一个0

!((x >> 31) ^ (y >> 31)) 同符号返回1，不同返回0

3.logtwo

检查最高位是否在16位以上,如果是，result+16，以此类推 ; 若最高位多于16位，v就右移16位，否则右移0位，以此类推

例如：

```
int shift16 = (v > 0xFFFF) << 4;
 result |= shift16;
 v >>= shift16;
```

同理，将result和shift8，shift4，shift2进行相同的运算

最后一步： result |= (v > 0x1);

这里result直接与真值或运算，可以减少步骤

4.byteSwap

题目要做第n,m个字节的交换，利用掩码mask1=0xff； mask2 = ~((0xff << n) | (0xff << m));

mask2的作用是是把x中第n,m的字节清为0。

先 n <<= 3； m <<= 3; 将m,n乘8，方便与掩码&

再int temp = (((mask1 & (origin_x >> m)) << n) | ((mask1 & (origin_x >> n)) << m));此步骤实现temp=2个字节到达互换后的位置

再 x &= mask2; 用mask2把x中n,m的字节清为0

最后x |= temp;完成互换

5.reverse

通过循环32次，逐位x 的每一位移动到 result 的相应位置。我觉得这个思路比较巧妙。
每次循环中，将 x 右移一位，将 result 左移一位，并将 x 的最低位添加到 result 的最低位。
当 x 变为0时，循环结束，result 即为反转后的结果。
为了减少操作数，使用分治法，逐步将32位分成更小的部分，然后逐级反转。

6.logicalShift

利用掩码 int mask=0xffffffff>>n;从左往右前n位是0，后32-n位是1

result=(x>>n)&mask

7.leftBitCount

运用并行的思想，先判断前16位是否全为1，若是，count+=16,然后x<<16;若否，则进入下一个判断，x<<count16

再判断前8位是否全为1，若是，count+=8,然后x<<8;

再判断前4位是否全为1，若是，count+=4,然后x<<4;

以此类推一直到判断前一位是否为1

例如：

```
int mask = 1 << 31;

count16 = (!(~(x & (mask >> 15)) >> 16)) << 4;

 count += count16;

 x <<= count16;
```

8.float_i2f (见亮点)

9.floatScale2

unsigned e = (uf >> 23) & 0xFF;

分三种情况：

如果是 NaN，直接返回uf

检查e是否为 0, 是则直接左移尾数位： return (uf << 1) | (uf & 0x80000000);

(uf & 0x80000000)保证了符号位不变

一般情况下： unsigned newE = e + 1;

unsigned result = (uf & 0x807FFFFF) | (newE << 23);

0x807fffff：s=1，阶码=00000000，M为23个1

uf& 0x807FFFFF,使s和M保持uf原本的不变，但阶码全置0

10.float64_f2i

思路比较常规，按照题目所说一步一步求即可

```
int float64_f2i(unsigned uf1, unsigned uf2)
{

    int  E;
    unsigned S = (uf2 >> 31);
    unsigned M,value;
    int mask1 = 0x000007ff; // 用于得到E
    int mask2 = 0x000fffff;
    E = ((uf2 >> 20) & mask1);
    int actual_E = E - 1023;
    M = ((uf2 & mask2) << 11) | (((uf1 >> 21) & mask1)) | (0x80000000); // uf2的低20位+uf1的高11位
    // 处理指数
    if (actual_E >= 31)
    {
        // 溢出
        return  0x80000000;
    }
    else if (actual_E < 0)
    {
        // 下溢
        return 0;
    }
    value = (M >> (31 - actual_E)) & ~(0x80000000 >> (31 - actual_E) << 1);
    if (S)
    {
        value = -value;
    }
    return value;
}
```

11.floatPower

思路比较简单

```
unsigned floatPower2(int x) {

    int newE = x + 127;


    if (newE >= 255) {
        return 0x7F800000; // 返回 +INF
    } else if (newE <= 0) {
        return 0x00000000; // 返回 0.0
    }

    // 构造新的浮点数表示
    unsigned result = (newE << 23);

    return result;
}
```

亮点

1. reverse
2. float_i2f
3. logtwo

reverse

```
// 附上题目解题代码
unsigned reverse(unsigned x)
{
    x = (x & 0x55555555) << 1 | (x & 0xAAAAAAAA) >> 1;   // 交换相邻的1位
    x = (x & 0x33333333) << 2 | (x & 0xCCCCCCCC) >> 2;   // 交换相邻的2位
    x = (x & 0x0F0F0F0F) << 4 | (x & 0xF0F0F0F0) >> 4;   // 交换相邻的4位
    x = (x & 0x00FF00FF) << 8 | (x & 0xFF00FF00) >> 8;   // 交换相邻的8位
    x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16; // 交换相邻的16位
    return x;
}
```

通过循环32次，逐位x 的每一位移动到 result 的相应位置。我觉得这个思路比较巧妙。
每次循环中，将 x 右移一位，将 result 左移一位，并将 x 的最低位添加到 result 的最低位。
当 x 变为0时，循环结束，result 即为反转后的结果。
为了减少操作数，使用分治法，逐步将32位分成更小的部分，然后逐级反转。

float_i2f

```
unsigned float_i2f(int x)
{
    if (x == 0)
        return 0;           //  x = 0 的情况
    if (x == 0x80000000)    // x = TMin，-2147483648
        return 0xCF000000;
    unsigned int e = 0, E, M, temp_x, round;
    unsigned int result;

    unsigned s = x >> 31 & 1;

    if (x < 0)
    {
        x = -x;
    }

    temp_x = x;
    while ((temp_x >> 1) >= 1)
    {
        temp_x >>= 1;
        e++;
    }
    E = e + 127;
    x <<= (31 - e);//小数部分
    M = (x >> 8) & 0x7fffff;//使尾数对齐
    //一个 int 有 32 位，其中有 31 位可以用以表示精度，而 float 的尾数位有 23 位，所以我们需要将 int 的精度位右移 8 位（即损失 8 位精度），从而得到 float 的尾数位。
    round = x & 0xff;//要舍入的小数部分，取x的最低八位。根据 “四舍六入五成双” 的原则，对于最低位是否要舍入进行判断。
    if (round > 0x80)//10000000，即128，如果尾数部分的最低8位大于128，说明我们需要对尾数进行向上舍入
       M += 1;//阶码++
    else if (round == 0x80)
    {
        if (M & 1)
        {
            M += 1;
        }
    }//如果最低8位等于128，根据“四舍六入五成双”舍入规则，
     //如果当前尾数 M 的最低位为1，则我们加1。
     //如果最低位为0，则不做变化。
    if (M >> 23)
    {
        M &= 0x7fffff;
        E += 1;
    }//检查尾数 M 是否超出了23位。如果最高位为1（即 M >> 23 为真），这意味着尾数溢出，
//因此保留尾数的低23位，阶码 E 加1

    result = (s << 31) | M | (E << 23);

    return result;
}
```

这道题80%比较好写，按照IEEE754的定义求出S,E,M。但是我一开始忽略了8位精度的损失，通过网上查阅资料才发现到这个点，我认为这个点设计的很巧妙（主要是当时写的时候还没上到精度损失的内容）

logtwo

```
int logtwo(int v)
{
    int result = 0;
    int shift16 = (v > 0xFFFF) << 4;//值为0或16
    result |= shift16;
    v >>= shift16;

    int shift8 = (v > 0xFF) << 3;
    result |= shift8;
    v >>= shift8;

    int shift4 = (v > 0xF) << 2;
    result |= shift4;
    v >>= shift4;


    int shift2 = (v > 0x3) << 1;
    result |= shift2;
    v >>= shift2;

    result |= (v > 0x1);//这里result直接与真值或运算，可以减少步骤

    return result;
}
```

思路比较简单，检查最高位是否在16位以上,如果是result+16，以此类推；若最高位多于16位，v就右移16位，否则右移0位，以此类推
比较困难的是把步骤压到25以下，因为很容易写出26步的版本
我通过或运算和左移，优化掉了+ (例如result |= shift16;v >>= shift16;)
在最后一次并行运算时result直接与真值或运算，可以减少步骤

反馈/收获/感悟/总结

前前后后大概花了15个小时，我认为难度比较合适，我遇到困难点主要在：前期的环境配置、熟悉操作、个别题目的分析
我认为最卡我的地方是，float_i2f这个题，我在没有题目信息提示的情况下，根本没有想到精度损失的问题，应该是我对IEEE 754 还不够熟悉
通过datalab，我熟悉各种位运算，了解了部分计算的底层逻辑，了解了int、float类型的数的构成方式
总体而言，还是很有意思的））

参考的重要资料

更适合北大宝宝体质的 Data Lab 踩坑记
https://arthals.ink/blog/data-lab