datalab 报告

总分	bitXor	samesign	logtwo	byteSwap	reverse	logicalShift	leftBitCount	float_i2f	floatScale2	float64_f2i	floatPower2
36	1	2	4	4	3	3	4	4	4	3	4

解题报告

亮点

整体上来讲，代码尽可能减少了声明变量和对变量的赋值，来减少没必要的空间占用和时间占用。

1. samesign 这个函数能够不创建新变量，直接返回对函数输入变量的处理表达式。
2. logtwo 这个函数用到了二分查找来快速定位 并且用或操作来避免了加法。
3. logicalShift 这个函数能够不创建新变量，直接返回对函数输入变量的处理表达式。
4. leftBitCount 这个函数用到了二分查找来快速定位。
5. floatScale2 这个函数通过分类，能够尽可能地减少计算次数。

bitXor

```
int bitXor(int x, int y) {
    return ~(x & y) & ~ (~x & ~y);
}
```

这个题比较简单，需要使用&和~来模拟异或操作，只需要先用(x & y)|(~x & ~y)来模拟异或，然后用德摩根定律来替换掉|操作就可以。

samesign

```
int samesign(int x, int y) {
    return !( (x ^ y) >> 31 ) & !( !!x ^ !!y );
}
```

这道题需要先排除x或者y是0的情况，在x和y只有一个是0的情况下，可以先用两次非操作将x和y处理成0或1的格式，然后使用!(!!x ^ !!y)排除掉x和y只有一个是0的情况，x和y同时为0的情况下x和y的最高位相同，可以和x和y都不是0的情况下同时处理。在这种情况下，只需要x和y的最高位相等就可以保障符号相同。

logtwo

```
int logtwo(int v) {
    int x=0,t=0;
    t = ((v >> 16) > 0) << 4;
    x |= t;
    v >>= t;
    t = ((v >> 8) > 0) << 3;
    x |= t;
    v >>= t;
    t = ((v >> 4) > 0) << 2;
    x |= t;
    v >>= t;
```

```
    t = ((v >> 2) > 0) << 1;
    x |= t;
    v >>= t;
    t = ((v >> 1) > 0);
    x |= t;
    v >>= t;
    return x;
}
```

这道题给定的参数都是只有一个1的，所以这道题是一道查找问题。由于不能使用循环操作且位数有限，于是可以使用二分查找来找到这个1，如果在某次二分操作中没找到1，那就在总位数加上这次二分查找排除的位数，并且将数据右移位数，反之就不加且不右移。同时这道题不让使用加法，且二分查找排除的位数都是二的整数次幂，只需要用位或操作代替加法即可。

byteSwap

```
int byteSwap(int x, int n, int m) {
    n <<= 3;
    m <<= 3;
    int mask1 = (x >> n) & 0xff;
    int mask2 = (x >> m) & 0xff;
    x &= ~((0xff << m) | (0xff << n));
    x = (mask1 << m) | x | (mask2 << n);
    return x;
}
```

这道题是将字节调换的操作，总共可以分成三步：提取并存储需要调换的两个字节；将原数据的这两个字节擦除；将字节粘贴到对应的位置。musk1和2就是提取的这两个字节，然后通过将这两个字节设为0之后&的操作就可以将这两个字节的位置变为0，之后使用|操作就可以粘贴字节数据。

reverse

```
unsigned reverse(unsigned v) {
    unsigned t = 0;
    int i = 0;
    while (i != 32)
    {
        t += (v & 1) << (31 - i);
        i += 1;
        v >>= 1;
    }
    return t;
}
```

这道题比较简单，t是输出的调换过的数据，只需要遍历v的三十二位，将v的第i位数据加到t的第31-i位就能得到v调换之后的结果。

logicalShift

```
int logicalShift(int x, int n) {
    return (x >> n) & ~(((x >> 31) << 31) >> n << 1);
}
```

这道题主要需要解决的问题解决是x在向右移位过程中自动补充的1，x >> 31能得到x的符号位信息，只需要将符号位再左移位到符号位再右移n-1位，就可以模拟出x右移n位之后最左侧的n-1位的状态。由于右移再左移能够将右侧的32-n位都变成0，利用掩码操作就可以保留右侧部分并且将左侧强制置为0。

leftBitCount

```
int leftBitCount(int x) {
    int p = 0, t = 0;
    p += !(x ^ 0xffffffff);
    t = !(((x >> 16) & 0xffff) ^ 0xffff);
    p += t << 4;
    x >>= (!t) << 4;
    t = !(((x >> 8) & 0xff) ^ 0xff);
    p += t << 3;
    x >>= (!t) << 3;
    t = !(((x >> 4) & 0xf) ^ 0xf);
    p += t << 2;
    x >>= (!t) << 2;
    t = !(((x >> 2) & 0x3) ^ 0x3);
    p += t << 1;
    x >>= (!t) << 1;
    t = !(((x >> 1) & 0x1) ^ 0x1);
    p += t;
    x >>= !t;
    return p;
}
```

这道题和之前的logtwo题目差不多，这道题需要找到从左数的第一个0，于是分别取t的左侧16位，8位……1位来找是否包含0，如果包含则右移且不加上本次检测的个数，检测剩余左半部分的结果，反之则不右移加上检测个数。特殊情况只有一种，即x为-1的情况下，如果加上先检测32位步骤将加上过多的检测次数，因此只需要当x=-1的时候，给检测数提前加上1，就可以减少计算步骤。

float_i2f

```
unsigned float_i2f(int x) {
    unsigned s = x & 0x80000000, m = 0, all = 0;
    if (x == -2147483648)
    {
        return 0xcf000000;
    }

    if (s != 0)
    {
        x = -x;
    }
    m = x << 1;
    int e = 0;
    while (!(m & 0x80000000))
    {
        if (e == 157)
        {
            break;
        }
        e += 1;
```

```
        m <<= 1;
    }
    m <<= 1;

    int temp = 0;
    if ((m & 0x1FF) > 0x100)
    {
        if (m + 0x100 < m)
        {
            temp = 1;
        }
        m += 0x100;
    }
    else if ((m & 0x1FF) == 0x100)
    {
        if ((m >> 9) & 1)
        {
            if (m + 0x100 < m)
            {
                temp = 1;
            }
            m += 0x100;
        }
    }
    m >>= 9;
    unsigned exp = 157 - e + temp;
    all = s + (exp << 23) + m;
    return all;
}
```

这道题是模拟将int转换为float的步骤。先检测x的符号位，如果是负数则不得不将x转换位正数来进行处理，但是这一步有一个问题，就是当x是-2147483648，也就是补码的最小位的时候，x在补码范围中没有负数，同时又无法同时处理正数和负数，只需要把这个唯一的特殊的数据排除，即求出这个数据的结果并且return。之后需要不断地左移x来找到最左侧是1的那一位来计算exp部分的值，这种情况下只有一种特殊情况，就是x为0的时候将导致无法跳出循环，同时只有x为0的情况下exp为0，只需要将x为0的时候设置e为157（之后会讲为什么是157）就能保证exp的值为0。此时m是x向右移到1在最左侧的情况，由于所有的非0整数都是规格数，只需要将m向右移一位去掉规格数的整数部分1。之后的if和else if是为了符合舍入规则：当舍入大于一半或等于一半且倒数第一位是1则进位，反之则舍去。同时在m + 0x100 < m即为m全是1的时候产生进位导致整个进位的时候m溢出的时候，exp需要加一位，只需要给这种情况下的exp加上temp的1。处理后将m右移到最右侧23位，计算exp（157=127+31-1，计算方式是bias的127加上31-e即为int的次数，最后减去1是因为在进入循环前已经右移一次来去除符号位）并且将符号位，exp和尾数m相加即可。

floatScale2

```
unsigned floatScale2(unsigned uf) {
    unsigned s = uf & 0x80000000, exp = uf & 0x7F800000, m = uf & 0x007FFFFF;
    if (exp == 0x7F800000)
    {
        return uf;
    }
    else if (exp == 0)
    {
        m = m << 1;
```

```
        return s | m;
    }
    else
    {
        exp += 0x00800000;
        return s | exp | m;
    }
}
```

这道题是为了得到f乘2的结果，如果f是Nan或者inf，只需要返回原值即可；如果是非规格数，因为非规格和规格数之间相差一位，即使是非规格数的真值右移，也将恰好变成规格数的exp等于1的情况，因此非规格数只需要将真值右移一位。如果是规格数，只需要将幂指数加一就可以。

float64_f2i

```
int float64_f2i(unsigned uf1, unsigned uf2) {
    int s = (uf2 >> 31) & 1,exp = (uf2 >> 20) & 0x7FF;
    int m_hi = uf2 & 0xFFFFF;
    unsigned m_lo = uf1;
    int E = exp - 1023;
    if (!(exp - 0x7FF)) {
        return 0x80000000;
    }

    if (E < 0) {
        return 0;
    }
    if (E > 31) {
        return 0x80000000;
    }

    int m1_hi = m_hi | 0x100000;
    int in;
    if (E >= 20) {
        in = (m1_hi << (E - 20)) | (m_lo >> (32 - (E - 20)));
    }
    else {
        in = m1_hi >> (20 - E);
    }
    if (s) {
        in = -in;
    }
    return in;
}
```

和正常的处理方式一样，先获得s，exp和frac(m)的值，m高位的值是m_hi，低位是m_lo。当原值是无限大的时候，返回int的最大值。如果e（指数）小于0，即int小于1的时候返回0。如果e大于31的时候也返回int最大值。由于e大于等于0的数都是规格数，只需要给m_hi加上缺失的整数1，在e大于20的时候需要左移m_hi，剩下的低位的0用m_lo补上；在e小于20的时候需要右移，不再考虑m_lo。最后如果是负数就求int的负数即可。

floatPower2

```
unsigned floatPower2(int x) {
    int exp = x + 127;
    unsigned result;
    if (exp >= 255) {
        return 0x7F800000;
    }
    if (exp <= 0) {
        return 0;
    }
    result = exp << 23;
    return result;
}
```

这道题需要返回2的x次方的exp，如果exp过大上溢就返回exp可能的最大值，下溢返回0，否则则返回移动到指定位置的exp本身。

反馈/收获/感悟/总结

在lab上大约花费了一天多的时间，处理的时间最长的题目是float_i2f题目。由于题目全是自己做的，也不知道这道题是不是考虑错误导致的题目最终导致逻辑链过长，花费了很长时间。感觉像是第二道题等能够用单纯的计算解决的这些题目是最有意思的，当然难度也不小，反而能够使用if和while的题目不需要太多的思考，如果是需要排除多种情况的题目（如float_i2f）需要不断地进行检验，感觉就没有很考验学生的思维（也有可能是我没想到好的解法）。

参考的重要资料

引用到了二分查找的思想，主要的参考资料是上课的PPT。