Data lab

1. `int bitAnd(int x, int y)#And`
    a. 第一个很容易，使用`~((~x)|(~y))`即可求得and。

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

2. `int getByte(int x, int n)#获取最后几位`

将需要的字节移至最右侧并遮罩其他字节

```
int getByte(int x, int n) {
//总操作数: 3
   return (x>>(n<<3))&(0xff);
}
```

3. `logicalShift(int x, int n)#实现logic Shift`

```
int logicalShift(int x, int n) {
//总操作数: 9
    int mostSignificantBit = (x >> 31) & 1;     // 提取最高位
    x = x ^ (mostSignificantBit << 31);        // 删除最高位
    x = x >> n;                                // 执行算数右移
    x = x ^ (mostSignificantBit << (32 + (~n))); // 恢复最高位
    return x;
}
```

4. `bitCount(int x)#一共有多少个1？`
    a. **第一阶段：**
        - `mask1 = 0x55 | (0x55 << 8);`
            - 构建一个16位的掩码，其中每两位有一位是1，即`01010101 01010101`。
        - `mask2 = mask1 | (mask1 << 16);`
            - 扩展这个掩码到32位。

- `x = (x & mask2) + ((x >> 1) & mask2);`
    - 这步将每两位分成一组，并计算每组中的1的数量。右移操作 `x >> 1` 将原始的 `x` 中的每个位移到相邻的位置上，接着与掩码做AND操作来隔离每一组的第二个位。最后，我们将处理后的 `x` 与原始的 `x` 相加，从而得到每组中1的数量。

b. **第二阶段**：
- 使用 `mask1 = 0x33 | (0x33 << 8);` 构建了一个新的16位掩码，即 `00110011 00110011`。
- 这步与第一阶段相似，但这次处理的是每4位一组。我们计算每4位中的1的数量。

c. **第三阶段**：
- 使用 `mask1 = 0x0F | (0x0F << 8);` 构建掩码 `00001111 00001111`。
- 这步将每8位分成一组，并计算每组中的1的数量。

d. **第四阶段**：
- 使用 `mask2 = 0xFF | (0xFF << 16);` 构建掩码 `00000000 11111111 00000000 11111111`。
- 将每16位分成一组，并计算每组中的1的数量。

e. **第五阶段**：
- 使用 `mask2 = 0xFF | (0xFF << 8);` 构建掩码 `11111111 11111111 00000000 00000000`。
- 这步将整个32位整数中的左16位和右16位相加，从而得到整数中1的总数量。

最终，返回的 `x` 值将是整数 `x` 中1的数量。

```
int bitCount(int x) {
//总操作数: 36
  int o, m;

  o = 0x55 | (0x55 << 8);
  m = o | (o << 16);
  x = (x & m) + ((x >> 1) & m);

  o = 0x33 | (0x33 << 8);
  m = o | (o << 16);
  x = (x & m) + ((x >> 2) & m);

  o = 0x0F | (0x0F << 8);
  m = o | (o << 16);
  x = (x & m) + ((x >> 4) & m);

  m = 0xFF | (0xFF << 16);
  x = (x & m) + ((x >> 8) & m);

  m = 0xFF | (0xFF << 8);
  x = (x & m) + ((x >> 16) & m);

  return x;
}
```

5. `bang(int x) #如何实现！`

首先，将x取反（每个位上的0变成1，1变成0）。

然后，对x进行一系列右移操作，每次将x的一半位与原来的x进行按位与运算。

最后，返回x的最低位（最右边的位），这个值为1或0，表示x的二进制表示中1的个数的奇偶性。

```
int bang(int x) {
//总操作数: 12
    x = ~x;
    x = x & (x >> 16);
    x = x & (x >> 8);
    x = x & (x >> 4);
    x = x & (x >> 2);
    x = x & (x >> 1);
    return x & 1;
}
```

6. `tmin(void)#最小值`

```
int tmin(void){
//总操作数: 1
  return 1<<31;
}
```

7. `fitsBits(int x, int n)#` 判断x是否能用n位的补码表示

这段代码检查一个整数x是否能用n个二进制位表示，通过将x右移n位后与0进行比较来判断。

```
int fitsBits(int x, int n) {
//总操作数: 7
    x = x >> (n + (~0));
    return (!x) | (!(x + 1));
}
```

8. `divpwr2(int x, int n)#` 除法

这段代码将整数x除以2的n次幂，考虑了向零舍入的情况

```
int divpwr2(int x, int n) {
//总操作数: 10
    int roundBias = x & ((1 << n) + (~0));
    return (x >> n) + ((x >> 31) & (1 ^ (!roundBias)));
}
```

9. `negate(int x)#取负`

```
int negate(int x) {
//总操作数: 2
```

```
  return (~x+1);
}
```

10. `isPositive(int x)`

    判断整数x是否为正数，通过将x的符号位右移并与1异或以实现。

```
int isPositive(int x) {
//总操作数: 4
    return ((x >> 31) + 1) ^ (!x);
}
```

11. `isLessOrEqual(int x, int y)#小于等于`

    首先，通过位操作获取x和y的符号位（最高位，0表示正数，1表示负数），将其存储在xNegSign和yNegSign变量中。

    然后，计算两个符号的相同性，将结果存储在sameSign变量中。如果x和y的符号相同，sameSign将为1，否则为0。

    最后，使用位操作将xNegSign、sameSign和yNegSign进行组合，以确定是否x <= y成立。返回值为1表示成立，0表示不成立。这个操作考虑了x和y的符号以及它们的相对大小。

```
int isLessOrEqual(int x, int y) {
//总操作数: 17
  int x_neg_y_posi = (x>>31&(!(y>>31)))&1;
  int x_posi_y_neg = !((y>>31&(!(x>>31)))&1);
  int sub = x + (~y);
  int sign = (sub>>31)&1;
  return (sign|x_neg_y_posi)&x_posi_y_neg;
}
```

12. `ilog2(int x)#求对数`

    这段代码用于计算一个整数x的以2为底的对数（即log2(x)）。它的逻辑如下：

    1. 初始化两个变量，count用于记录结果，remainder用于保存x右移后的余数。
    2. 通过将x右移16位并将结果保存在remainder中，然后检查remainder是否不等于0。如果不等于0，说明x的高16位有1，因此将count增加16，然后通过将x减去remainder左移16位的结果（即将高16位清零）来更新x。
    3. 重复步骤2，但这次是将x右移8位，并根据remainder是否不等于0来更新count和x。
    4. 继续重复这个过程，每次右移的位数减半，直到最后一次右移2位，然后将剩余的x加到count中。
    5. 最后，返回count减1，因为log2(x)的整数部分要减去1才能得到正确的结果。

    总的来说，这段代码通过逐步右移x的位数并检查余数是否不为0来计算log2(x)，最后返回结果减1。

```
int ilog2(int x) {
//总操作数: 25
    int count = 0;
```

```
    int remainder = 0;
    remainder = x >> 16;
    count = count + (remainder != 0) * 16;
    x = x - (remainder << 16) * (remainder != 0);
    remainder = x >> 8;
    count = count + (remainder != 0) * 8;
    x = x - (remainder << 8) * (remainder != 0);
    remainder = x >> 4;
    count = count + (remainder != 0) * 4;
    x = x - (remainder << 4) * (remainder != 0);
    remainder = x >> 2;
    count = count + (remainder != 0) * 2;
    x = x - (remainder << 2) * (remainder != 0);
    count = count + x;
    return count - 1;
}
```

13. `float_neg(unsigned uf)#取负`

```
unsigned float_neg(unsigned f) {
//总操作数: 2
    return f ^ (1 << 31);
}
```

14. `float_i2f(int x)#int to float`

这段代码实现了将整数x转换为单精度浮点数的操作。

1. 首先，处理了特殊情况：如果x为0，则直接返回0；如果x为0x80000000（负的最小整数），则返回一个特定的浮点数表示。
2. 接下来，确定浮点数的符号位（sign），如果x为负数，则将sign置为1，并将x取反变成正数。
3. 然后，设置浮点数的指数部分（exponent）为158，并将x左移，直到x的最高位为1，同时相应地减小指数。
4. 如果x的最低8位中有大于等于0x80的值，表示需要进行四舍五入操作，将x加1，并在加1后如果最高位进位了，还需要将指数加1。
5. 最后，提取x的小数部分（fraction），将指数左移23位，最后通过按位或操作将符号位、指数部分和小数部分合并，得到最终的单精度浮点数表示，并返回该值。

```
unsigned float_i2f(int x) {
  //
  // 执行整数到浮点数转换。
  // 使用无符号整数变量 t 和 f 以避免算术位移问题。
  //
  int exponent = 0, shift_amount = 0, carry = 0; // 修改变量名：b -> exponent, s -> shift_amount, c -> carry
  unsigned int result = 0x00000000;
  unsigned int t_copy = x, f_copy = x; // 修改变量名：t -> t_copy, f -> f_copy

  // 如果 x 为负数，则使用绝对值。
  if(x >> 31) {
    result = 0x80000000;
    t_copy = f_copy = (~x) + 1;
```

```
  }

  // 计算二进制表示中的位数。
  while(t_copy) {
    exponent = exponent + 1;
    t_copy = t_copy >> 1;
  }
  shift_amount = 24 - exponent;

  // 检查位数损失（精度损失）。
  if(shift_amount < 0) {
    shift_amount = 0;
    while(f_copy >> 25) {
      carry = carry | (f_copy & 1);
      f_copy = f_copy >> 1;
    }
    f_copy = (f_copy >> 1) + (f_copy & (carry | (f_copy >> 1)) & 1);
    if(f_copy >> 24) {
      exponent = exponent + 1;
    }
  }

  // 如果 x 为 0，则使指数位为 0。
  if(!exponent) {
    exponent = -126;
  }

  // 构建浮点数的各个部分并返回结果。
  result = result | ((f_copy << shift_amount) & 0x007FFFFF);
  result = result | ((exponent + 126) << 23);
  return result;
}
```

15. `float_twice(unsigned uf)#`乘法

这段代码接受一个无符号整数参数uf，该参数表示一个单精度浮点数的二进制表示。

首先，通过位操作提取出浮点数的符号位、指数位和尾数位。

然后，根据特定的指数范围和位操作来将浮点数转换为整数。如果指数超过特定范围，函数返回一个特殊值0x80000000，表示溢出或不合法的输入。如果指数在有效范围内，它将尾数与指数一起重新组合成整数。

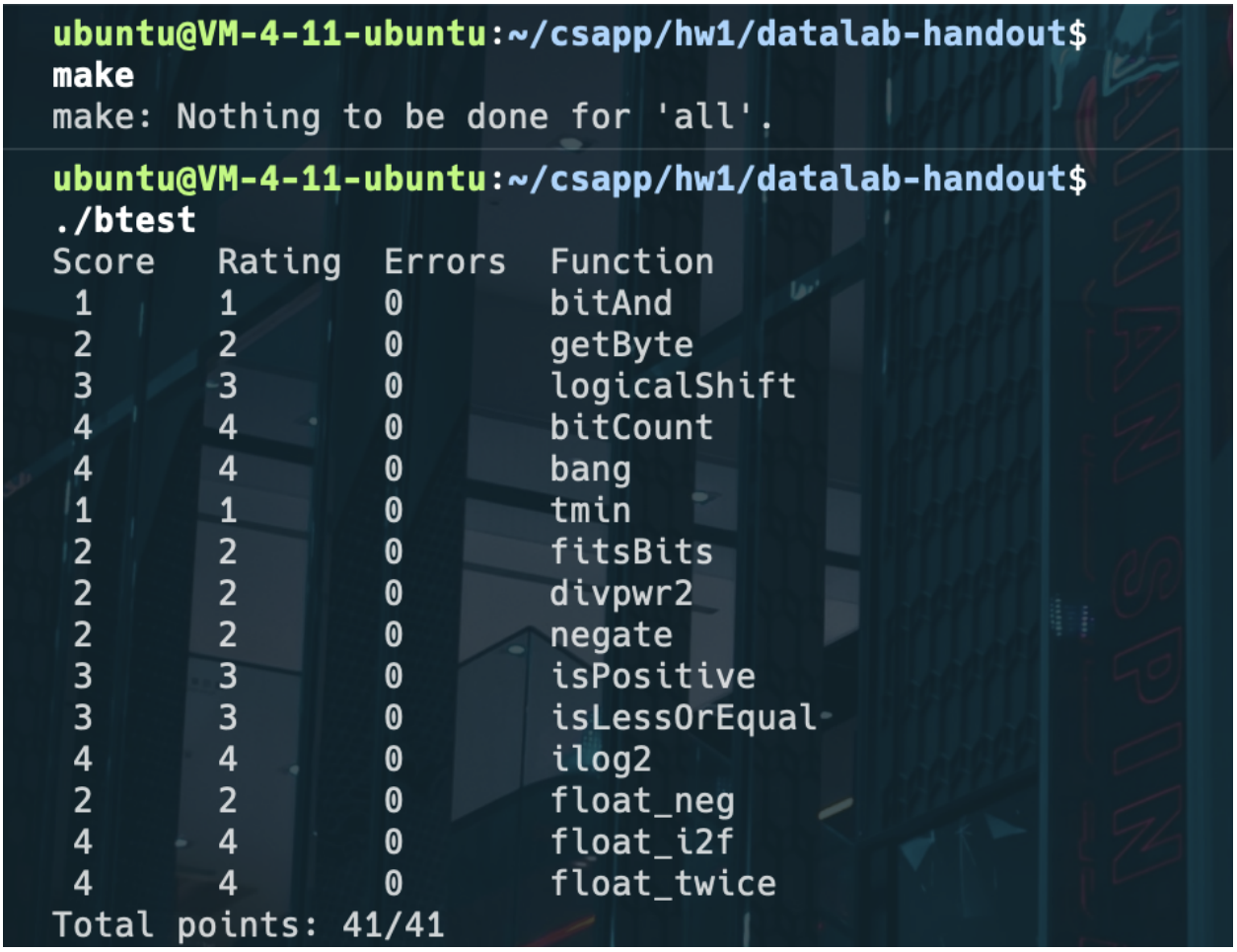
最后，如果符号位为1（表示负数），则将结果取负数，否则返回正整数结果。

总之，该函数的主要逻辑是将单精度浮点数表示转换为整数表示，考虑了符号、指数和尾数等因素。

```
unsigned float_twice(unsigned uf) {
  // 提取浮点数uf的指数部分
  unsigned exponent = (uf >> 23) & 0xFF;

  // 如果指数部分为0，表示uf为非规格化数（或者0）
  if (!exponent) {
    // 将指数部分设置为最小规格化数的指数值
    exponent = 0xFF;
    // 将uf的符号位保持不变，并将尾数部分左移1位
    uf = (uf & 0x80000000) | (uf << 1);
  }
```

```
  // 如果指数部分为254（0xFE），表示uf为无穷大或NaN
  if (!(exponent ^ 0xFE)) {
    // 将指数部分设置为最大规格化数的指数值
    exponent = 0xFF;
    // 将uf的符号位保持不变，并将指数部分左移23位
    uf = (uf & 0x80000000) | (exponent << 23);
  }

  // 如果指数部分不等于255（0xFF），表示uf为规格化数或者非规格化数
  if (exponent ^ 0xFF) {
    // 将uf的尾数部分加上1（尾数加1，相当于浮点数加上2^(-23)）
    uf = uf + (1 << 23);
  }

  return uf;
}
```

总结



```
ubuntu@VM-4-11-ubuntu:~/csapp/hw1/datalab-handout$
./dlc bits.c

ubuntu@VM-4-11-ubuntu:~/csapp/hw1/datalab-handout$
./dlc -e bits.c
dlc:bits.c:144:bitAnd: 4 operators
dlc:bits.c:155:getByte: 3 operators
dlc:bits.c:170:logicalShift: 9 operators
dlc:bits.c:200:bitCount: 36 operators
dlc:bits.c:216:bang: 12 operators
dlc:bits.c:225:tmin: 1 operators
dlc:bits.c:238:fitsBits: 7 operators
dlc:bits.c:251:divpwr2: 10 operators
dlc:bits.c:261:negate: 2 operators
dlc:bits.c:271:isPositive: 4 operators
dlc:bits.c:284:isLessOrEqual: 18 operators
dlc:bits.c:326:ilog2: 35 operators
dlc:bits.c:344:float_neg: 7 operators
dlc:bits.c:398:float_i2f: 27 operators
dlc:bits.c:439:float_twice: 14 operators
```

```
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       9       logicalShift
4       4       0       2       36      bitCount
4       4       0       2       12      bang
1       1       0       2       1       tmin
2       2       0       2       7       fitsBits
2       2       0       2       10      divpwr2
2       2       0       2       2       negate
3       3       0       2       4       isPositive
3       3       0       2       18      isLessOrEqual
4       4       0       2       35      ilog2
2       2       0       2       7       float_neg
4       4       0       2       27      float_i2f
4       4       0       2       14      float_twice

Score = 71/71 [41/41 Corr + 30/30 Perf] (189 total operators)
```

注意边界

各种边界情况很重要，

如最大最小，TMax绝对值比Tmin绝对值小1等。

以及注意溢出，注意是多少位的，需要小心是否会存在溢出情况，如果可能的话，最好在一开始就考虑溢出的情况。

善用位操作

之前知道>><<左移右移可以进行乘除法，但是第一次大量使用移位来进行乘法还是有趣的，以及通过or操作来进行求和。

浮点数

浮点数的运算比int麻烦太多，不管是浮点数的表示还是浮点数的计算，浮点数都需要考虑很多，如是否会溢出，是否是NAN等判断逻辑

通过本次实验的收获也当然是关于的位级表示以及运算掌握比以前好很多，熟练很多，希望在以后的coding中会有实际用到的时候。