./dlc -e bits.c：

```
a1341543541@ubuntu:~/ICS/lab/lab1/datalab-handout$ ./dlc -e bits.c
dlc:bits.c:143:bitAnd: 4 operators
dlc:bits.c:154:getByte: 3 operators
dlc:bits.c:165:logicalShift: 6 operators
dlc:bits.c:186:bitCount: 33 operators
dlc:bits.c:196:bang: 6 operators
dlc:bits.c:205:tmin: 1 operators
dlc:bits.c:218:fitsBits: 14 operators
dlc:bits.c:230:divpwr2: 13 operators
dlc:bits.c:240:negate: 2 operators
dlc:bits.c:250:isPositive: 5 operators
dlc:bits.c:263:isLessOrEqual: 15 operators
dlc:bits.c:292:ilog2: 46 operators
dlc:bits.c:312:float_neg: 7 operators
dlc:bits.c:349:float_i2f: 25 operators
dlc:bits.c:379:float_twice: 19 operators
```

./btest：

```
a1341543541@ubuntu:~/ICS/lab/lab1/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
 2      2       0       fitsBits
 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: 41/41
```

bitAnd：

思想：A 且 B 的反就是非 A 或非 B，将非 A 或非 B 取反则就是 A 且 B，在这里单个位运算与逻辑运算等价。

getByte：

思想：可以将 x 向左移动（3-n）*8 位再向右移动 24 位，此时想要的 byte 位置会处于最低有效字节，但考虑到算术右移的问题，可以直接将 x 向右移动 n*8 位。即 n<<3 位，只有最低有效字节的数据是需要的，因此可以与一个最低有效字节全为 1 其余为全是 0 的数（0xff）做位与运算。

logicalShift：

如果一个二进制数最高位为 1，算术右移后都补 1，要使补的位变成 0，可以将右移 n

位后的数与[31..(32-n)]位都是 1 其余位是 0 的数取异或；即使原本编译器是逻辑右移，补的数是 0，也可以与 0x00000000 取异或。这个数可以通过单独提出参数的符号位后右移 n 位再左移 1 位获得。

bitCount：

统计一个二进制数里面的 1 的个数，基本思路就是将所有的位上的一个个数相加，每一位可以通过(x>>n)&0x01 来取到，n 是 1 到 32 的整数。但是由于操作符最大数量的限制，一个个相加是不允许的，可以计算每个 byte 上 1 的个数，再相加；可以通过(x>>n)&0x01010101 来不断移位相加获得一个 count，此时每个 byte 上储存的是这个 byte 上的 1 的个数，通过(count>>16)+count 来实现第 0 个字节与第 2 个字节、第 1 个字节与第 3 个字节上的数字相加，再((count>>8)+count)&0xff 实现较小两位字节的相加，并除去非最低有效字节上的数。

至于为什么不选择 0x00010001 是因为会超过操作符上限；不选 0x11111111 是因为按照上述方法处理后得到的 1 的最大个数为 15，而一个 4byte 的二进制数最多可以有 32 个 1。

bang：

0、0x80000000 与其他非 0 数的区别是取补码后其符号位是否与取之前相同，即 0、取反加 1 后符号位仍为 0；0x80000000 取反加 1 后符号位仍为 1；而其余非 0 取反加 1 后符号位不同了， 因此只需要将 x 与其补码的符号位满足以下运算（设*为操作符）：0*0=1；1*0=0；0*1=0；1*1=0；即或非运算，最后返回符号位的值。

tmin：

二进制补码数最小的是 0x80000000，即 1<<31。

fitsBits:

如果 x 是一个正数，则 x 能被 n 位 bit 表示的条件是 x 右移(n-1)位的结果是 0（第 n-1 位是符号位，需要为 0），如果不为 0，说明 x 需要更多的位数的 bit 才能表示；如果 x 是一个负数，则 x 能被 n 位表示的要求是 x 右移(n-1)位后的结果全为 1（第 n-1 位是符号位，需要为 1），或者是~x 右移(n-1)位后的结果为 0。只需要将两个综合起来考虑即可。

divpwr2：

直接的 x>>n 得到的结果 result 总是有 x>=(result<<n)。但这个函数的要求是：如果 x 为正，x>=(result<<n)；如果 x 为负：x<=(result<<n)。

可以考虑当 x 为负数时将取相反数变成正数再将结果取相反数，即~((~x+1)>>n)+1；但如果 x 为正数时使用这个方法会导致 x<=(result<<n)。但如果 x 为正数时仅仅将 x 按位取反，再移位再按位取反是不影响结果的，即：~(~x>>n)；这两个表达式最大的差别在于使 x 按位取反还是取相反数，而这两个操作的差别在于是否将按位取反后的 x 加 1；x 是负数则加 1，正数则加 0，因此可以将要加的数设置成 x 的符号位上的数。但需要考虑 0x800000 的特殊情况（按位取反加 1 后符号位不变），此时这个数可以当作正数来处理，因此处理其符号位时要单独考虑其特殊性，使其符号位与正数符号位是一样的。

negate：

有符号二进制数的负数储存形式就是按位取反加 1。

isPositive：

非负数与负数的差别在位符号位，则将 x 右移 31，如果 x 是非负数，右移后结果为 0，取非得 1；如果 x 是负数，右移后结果不为 0（算术右移后为 0xffffffff，逻辑右移后为 1），取非得 0。但此时当 x=0 时得到的结果不符合要求。将 x 取非再取非后将得到 0 或者 1，当且仅当 x 为 0 时!(!x)为 0，否则为 1，因此将前后两项做位与运算:

x 为正，!(x>>31)=1，!(!x)=1，位与运算结果为 1;

x 为 0，!(x>>31)=1，!(!x)=0，位与结果为 0;

x 为负，!(x>>31)=0，!(!x)=1，位与结果为 0;

isLessOrEqual：

如果 x 与 y 符号位相同，此时做相减运算不会产生溢出的问题，则由 x<=y 可得(y-x)>=0，此时只需要看 y-x 的符号位是否为 1 就可与得到结果（方式与 isPositive 中一样）。如果 x 与 y 符号位不同，则 x 符号位为 1 而 y 符号位为 0 时 x<y，否则 x>y。

ilog2：

求$\log_2 x$的向下取整的数（x>0），只有值为 1 的最高位 bit 影响结果，低于最高位的权重之和加起来也小于最高位的权重。而得到最高位的方法可以利用!(!(x>>1))+ !(!(x>>2))+……+!(!(x>>30))来判断，但使用的符号超出了限制，于是可以将 x 最左边 1 的后面全部变成 1，再数出 1 的个数就是 x 的最高位 1 的位置，再减 1 就得到结果。

float_neg：

浮点数的取负数只需要将符号位取反，可以将传入形参 uf 与 0x80000000 取异或操作，实现符号位的取反，其余位的不变。要判断 uf 是否是 NaN，要判断阶码字段是否全为 1 且小数字段是否不全为 0。可以将阶码位加 1 取阶码的八位判断是否全为 1，如果是，加 1 后取只取阶码位得全 0；否则非全 0。只需将得到的结果与 frac 进行逻辑与操作即可判断。

float_i2f：

要实现将整数强制转换为浮点数的功能，首先需要考虑整数是否为 0，是的话则返回 0，不是的话，则需要单独取出其符号位作为转换后浮点数的符号位。如果参数为负，需要将其取补码转化为正数 positive_x（0x80000000 要单独输出，因为没法转换）。还需要判断浮点数的幂的数值 n，可以将 positive_x 右移（n-1）位后查看其值是否为 0。

由于不为 0 的整数的绝对值一定大于等于 1，则其转换为浮点数后的阶码段不会全为 0 或者全为 1。将参数左移（31-n）位使最左边的 1 刚好处于最高有效位。此时用作小数位的位置区域是[30 … 8]位。

接下来要检查是否有进位：第 7 位为 1 且[6 … 0]位不全为 0 则需要进 1；或者第 7 位为 1，[6 … 0]位全为 0 并且第 8 位为 1 要进 1。最后将阶码、符号位、小数部分按顺序排列就可得到结果。

float_twice：

0 的两倍还是 0，如果参数是无穷大，两倍后还是无穷大，因此可以直接将阶码位全为 1 的参数直接返回，此时 NaN 也被包含在其中。如果阶码位全为 0，则将 frac 位左移 1，左移后的进位的 1 如果超过第 22 位，自动变成阶码位，不会影响结果。其余情况的乘以 2 只需将阶码位的数值加 1 即可，如果值有溢出（加 1 后阶码位变成全为 1），则输出无穷大。

reference：

思考 bang 的时候，室友问了一句：如何写一个函数将全为 1 的参数返回 1 其余返回 0，我直接脱口而出：加 1 取逻辑非。这促使我想到了 0 与非 0 的一个很大的区别，即其补码与原数的符号位是否相同，同时还想到了一个反例，将其排除后就得到了结果。

思考 ilog2 查阅的资料：

https://blog.csdn.net/fall221/article/details/12322439?utm_medium=distribute.wap_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.wap_blog_relevant_no_pic2&depth_1-utm_source=distribute.wap_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.wap_blog_relevant_no_pic2

https://bbs.csdn.net/topics/350160094?utm_medium=distribute.wap_relevant.none-task-discussion_topic-BlogCommendFromBaidu-6.wap_blog_relevant_no_pic2&depth_1-utm_source=distribute.wap_relevant.none-task-discussion_topic-BlogCommendFromBaidu-6.wap_blog_relevant_no_pic2

其余的参考：
https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/

https://blog.csdn.net/CAU_Ayao/article/details/83987120?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param
（降低 gcc 版本的）

教材《深入理解计算机系统》P78-84，P91

https://blog.csdn.net/aobi6343/article/details/101118023?utm_medium=distribute.pc_relevant.none-task-blog-OPENSEARCH-2.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-OPENSEARCH-2.channel_param