实　验　报　告

一.实验名称：第一次实验

二、实验学时：40

三、实验内容和目的：

实验 1 要求使用有限的位操作符完成指定功能的函数，实验前请认真阅读附件中的实验指导，然后完成代码编写调试，测试平台为 linux。

四、实验原理：

bitXor：用~和&运算完成^（异或）运算，例如 bitXor(4, 5) = 1。

思路：核心是两个转换式 x^y=((~x)&y)|(x&(~y))，以及 x|y=~((~x)&(~y))。首先利用异或的运算法则，即式子 x^y=((~x)&y)|(x&(~y))，初步的将异或转化为与或表达式，即代码中的 var1|var2。然后利用式子 x|y=~((~x)&(~y))将或式转化为与非式即可。

```
int bitXor(int x, int y) {  //var1 | var2=~((~var1) & (~var2))
    int var1 = (~x) & y;
    int var2 = x & (~y);    //x^y=((~x) & y) | (x & (~y))=var1 | var2
    int var3 = ~((~var1) & (~var2));
    return var3;
}
```

Tmin：返回最小的二进制补码整数，Tmin 是 0x80000000。

思路：核心是弄清楚二进制补码的表示。
Int 类型 4 个字节，32 位，Tmin 的补码构成为：符号位为 1 其余全 0。想要返回 0x80000000，那么肯定要用到移位运算符，将常数 1 左移 31 位即可。

```
int tmin(void) {    //即 0x80000000
    return 1<<31;
}
```

isTmax：如果是补码最大值就返回 1，否则返回 0，Tmax 是 0x7FFFFFFF。

思路：核心是对两个对象是否相同的判断。
想要返回 0、1，肯定要用到 !运算符，!0=1，其他的返回结果都为 0。根据上题可知 Tmin，我们将 Tmin 按位取反即为 Tmax。之后，我们只需判断 x 和 Tmax 是否相等即可。这里给出我判断两二进制数是否相等的方法：将两个二进制数 a、b 分别按位相与以及取反相与，即 a&b 和(~a)&(~b)，再做异或若结果全为 1，两数即相等。例如 1010 和 1110，a&b=1010&1110=1010，(~a)&(~b)= 0101 &0001=0001，异或结果为 1011；1010 和 1010，a&b=1010&1010=1010，(~a)&(~b)=0101&0101=0101，异或结果为 1111。这样子将异或结果按位取反，再使用 !运算符即可得到结果。

```
int isTmax(int x) {
    int var1=1<<31; //按位取反即 Tmax=~var1
    return !(~(((~var1) & x) ^ (var1 & (~x))));
}
```

allOddBits：如果所有奇数位为 1 就返回 1，否则返回 0，例如 allOddBits(0xFFFFFFFD)=0,allOddBits(0xAAAAAAAA)=1。

思路：核心是拆而和的思想。
由于不能使用超过 8 位的常数，所以考虑将输入的 x 拆成四个字节分别处理。先分别与 0xAA 相与得到奇数位上的数字（偶数位上全 0），再分别与 0xAA 进行异或，如果所有奇数位为 1 的话，异或的结果将为全 0。之后将所有的字节结果加和，再取非即为所求返回值。

```
int allOddBits(int x) {
    int var1=0xAA & x;
    int var2=0xAA & (x>>8);
    int var3=0xAA & (x>>16);
    int var4=0xAA & (x>>24);
    int var5=(var1^0xAA) + (var2^0xAA) + (var3^0xAA) + (var4^0xAA);
    return !var5;
}
```

negate：返回-x。

思路：根据二进制补码整数运算，取负将 x 按位取反加 1 即可。

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

isAsciiDigit：当 0x30≤x≤0x39（对应 ASCII 为'0'到'9'）时返回 1。

思路：将比较大小转化为构造有效位，右移 n 位判断末位是否为 1 实现。
0x30≤x≤0x39 即 48~57，对应 00110000~00111001，有效位为后 6 位，则可通过验证 x>>6 是否全 0 来保证高位都是 0。48~57 均大于整型位数 32，为了构造有效位位移，将 x>>1，即 24~28，共 5 个数（24、25、26、27、28），对应 0x1F。则 0x1F<<24 即为右移基准，通过将 0x1F<<24 右移 24~28 位后判断末位是否为 1 得到结果，结果为 1 说明是 AsciiDigit。

```
int isAsciiDigit(int x) { //48~57
    return (0x1F<<24>>(x>>1)) &!(x>>6); //右移 24~28 & 保证高位都是 0
}
```

conditional：完成 x?y:z 运算，例如 conditional(2,4,5)=4。

思路：核心是怎么用位运算判断是否为 0。
当 x 不全 0 时，x 与-x 中总有一个为负数，右移 31 位后得到全 1（对应结果 y）；而 x 为 0 时，x 与-x 都全 0，右移 31 位后仍然全 0（对应结果 z）。通过位移结果分情况与对应 y/z 相与再加和即可得到目标结果。

```
int conditional(int x, int y, int z) {
    int var=((x | (~x+1))>>31); //当 x 全 0 时，var 全 0；当 x 不全零时，
x 与-x 中总有一个为负数，完成移位后得到全 1
    return (y & var) + (z & (~var));
}
```

isLessOrEqual：x≤y 时返回 1，否则返回 0。

思路：核心是要考虑 x 为 0x80000000，y 为 0x7fffffff 这种异常情况，不能简单通过相减得结果。
判断 x≤y 是否成立，一种简单思路是计算 x-y（即 x+(~y+1)），结果为 0 或结果符号位为 1 时返回 1，否则返回 0。但是，要特别注意 x 为 0x80000000，y 为 0x7fffffff 这种异常情况。此时 x+(~y+1)产生溢出，结果为 1，符号位为正，返回 0，但此时 x≤y 是成立的。

```
int isLessOrEqual(int x, int y) {
    //ERROR: Test isLessOrEqual(-2147483648[0x80000000],2147483647[0x7fffffff])
failed...
    //要考虑最高位符号位，上边 ERROR 符号位参与计算做差结果为 1，正数，产生错误
    int x_sign=(x>>31) & 1;
    int y_sign=(y>>31) & 1;
    int var1=x+(~y+1);
    int var1_sign=(var1>>31)&1;
```

```
    //return var1_sign | !((x & x)+(y & y));
    return (x_sign & !y_sign) | ((!(x_sign^y_sign)) & var1_sign) | (!var1); //异号 | 同号 | 相等
}
```

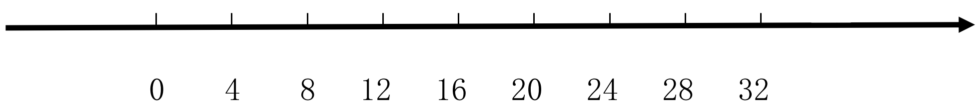
logicalNeg：实现！运算符，例如 logicalNeg(3)=0,logicalNeg(0)=1。

思路：核心仍然是怎么用位运算判断是否为 0。
当 x 不为 0 时，x 与-x 中总有一个为负数，即符号总有一个为 1。则通过 x 与-x 符号位取非再相与的结果即可判断 x 是否为 0，实现！运算。当 x 为零时，x 与-x 符号位取非均为 1，相与结果为 1，返回 1，同理 x 不为零时返回 0。

```
int logicalNeg(int x) { //x 不为 0 时，x 和-x 符号总有一个为 1
    int val1=(~((~x+1)>>31))&1;
    int val2=(~(x>>31))&1;
    return val1 & val2;
}
```

howManyBits：返回表示 x 的最少二进制补码位数，例如 howManyBits(12)=5，howManyBits(298)=10，howManyBits(-5)=4，howManyBits(0)=1，howManyBits(-1)=1，howManyBits(0x80000000)=32。

思路：核心是二进制补码的表示范围。
p.s.二进制补码的表示范围是-2^(n-1)~(2^n-1)-1。通过 x^(x<<1)去除符号位并进行异或，通过异或可以判断最高位，即首个出现 1（前边都是 0）的位的位置），然后依次判断高 16 位是否有 1；剩余位高 8 位是否有 1，剩余位高 4 位是否有 1，剩余位高 2 位是否有 1，最后的返回值记得加上符号位(+1)。




```
int howManyBits(int x) {
    int var1;
    x=x^(x<<1);
    var1=(!(x>>16))<<4; //16
    var1^=24; //有 1->var1=24,进而判断前 16 位的前 8 位;
              //全 0->var1=8，进而判断后 16 位的前 8 位
    var1^=(!(x>>var1))<<3;  //8
              //有 1->var1=24/8
              //全 0->var1=16/0
    var1^=4;  //同理，有 1->var1=(11000^00100)/(1000^0100)=28/12
              //全 0->var1=(10000^00100)/(000^100)=20/4
```

```
    var1^=(!(x>>var1))<<2;  //4
            //同理，有 1->var1=（28/12）/（20/4）
            //全 0->var1=（24/8）/（16/0）
            //即得到了上述数轴描述的基准点
    var1+=( (~0x5B)>>((x>>var1) & 30) ) &3;   //30=11110
            //0xFFFFFFA4(110100100) 00 01 10 11
            //对最后的4位(I，II，III，IV)分为最高位I为1/II为1/III
            //为1/IV为1四种情况分别考虑
    return var1+1;
}
```

floatScale2：返回位级的浮点数 2*f 运算。

思路：核心是分情况讨论浮点数运算。
当 uf 为 0 时，2*0=0，返回 0。当 uf 为-0 或正负无穷时，2*f 运算产生溢出，返回 uf。当 uf 为正负非规格化数时（非规格化数对应的阶码全 0），返回值为将非规格化数尾数部分右移一位，其余位不变。最后，当 uf 为规格化数时只需将阶码加 1 即为返回值。

```
unsigned floatScale2(unsigned uf) {
    //Test floatScale2(8388608[0x800000]) failed...
    //...Gives 4194304[0x400000]. Should be 16777216[0x1000000]
    //Floating point value -0
    //Bit Representation 0x80000000, sign=1, exponent = 0x00, fraction = 0x000000
    //Denormalized.  -0.0000000000 X 2^(-126)
    if(uf==0)   return uf;  //2*0=0
    if((uf==(1<<31)) || (((uf>>23) & 0xFF)==0xFF)) return uf;  //-0或正负无穷时
    if(((uf>>23) &0xFF)==0x00)  //处理正负非规格化数，非规格化数对应的阶码全0
        return ((uf & 0x007FFFFF)<<1) | (uf & 0xFF800000);
    return uf+0x00800000;  //规格化数，阶码加1
}
```

floatFloat2Int：完成位级的浮点数到整型的转化。

思路：核心是理解浮点数和整型的表示方法。
分别让 uf 与 0x80000000、0x7F800000、0x007FFFFF 相与然后相应右移得到浮点数的符号部分 S、指数部分 E 及尾数部分 M，注意指数部分要减 127，尾数部分应另外算上隐藏的一位共 24 位，即 M 记录的是小数部分(1.***)<<23。然后进行分条件的转化。当 E<0 时，说明 uf 为 0~1 间的小数，对应的整数为 0。当 E>=31，超出整型的表示范围，产生溢出，按题目要求返回 0x80000000。另外，由于 M 记录的是小数部分(1.***)<<23，则当 E>23 时，var1=M<<(E-23)，E≤23 时，var1=M>>(23-E)；此时当符号位为 1 时还需要对 var1 取负。
此处我有点困惑的是判断条件中如果不加(E+127)==0 会产生 0x800000 的测试错误，按理说此时 E=1-127=-126 也满足 E<0 的呀，但它就是不通过。

```
int floatFloat2Int(unsigned uf) {
    //ERROR: Test floatScale2(8388608[0x800000]) failed...
    //...Gives 4194304[0x400000]. Should be 16777216[0x1000000]
    //ERROR: Test floatFloat2Int(1065353216[0x3f800000]) failed...
    //...Gives 0[0x0]. Should be 1[0x1]
    int S=(uf & 0x80000000)>>31;    //符号部分
    int E=((uf & 0x7F800000)>>23)-127;  //指数部分
    int M=(uf & 0x007FFFFF)+0x00800000;  //尾数部分，算上隐藏的一位，尾数部分一共24 位
    //M 记录的是小数部分(1.***)<<23
    int var1=0;
    if(E<0 || (E+127)==0) return 0;   //对应的整数绝对值小于 1
    else if(E>=31)  return 0x80000000;  //溢出
        else if(E>23)  var1=M<<(E-23);
            else if(E<=23)   var1=M>>(23-E);
    if(S)   var1=~var1+1;
    return var1;
}
```

floatPower2：返回位级的 2.0^x 运算，输入为整型，如果结果太大返回+INF。

思路：核心是理解浮点数的表示方法。
按从小到大分情况讨论。当 x<-150 时，太小，无法转换为对应浮点数。当-150≤x≤-127 时，对应的是非规格化数，非规格化数对应的阶码全 0，尾数字段表示 2 的幂，则让 1 左移(-x-127)位即可（由于 2 的幂都为正）。当-126≤x≤127，可表示为正常规格化浮点数，通过(x+127)<<23 更改阶码即可。另外，x≥128，对应结果太大返回+INF，即 0xFF<<23。

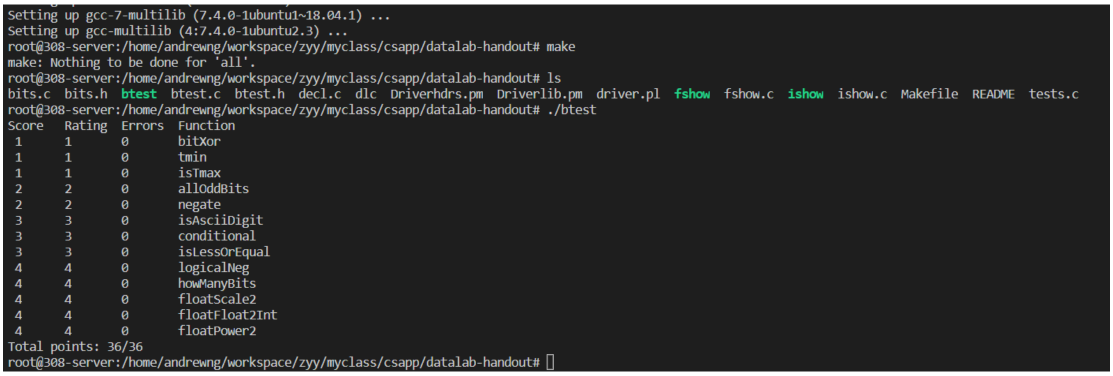
```
unsigned floatPower2(int x) {
    if(x<-150)  return 0;
    if(x>=-150 && x<=-127)  //非规格化数
        return 1<<(-x-127);   //由于 2 的幂都为正
    if(x>=-126 && x<=127)   //规格化数
        return (x+127)<<23;
    if(x>=128)  //结果太大返回+INF
        return 0xFF<<23;
    //Floating point value inf
    //Bit Representation 0x7f800000, sign = 0, exponent = 0xff, fraction = 0x000000
    //+Infinity
    return 0;
}
```

五. 实验步骤及结果：

1. tar xvf datalab-handout.tar 解压代码，在 bits.c 中解题
2. 配置 gcc 环境

sudo apt-get purge libc6-dev
sudo apt-get install libc6-dev
sudo apt-get install libc6-dev-i386
sudo apt-get install gcc-multilib

3. make and ./btest，结果如下图：



```
Setting up gcc-7-multilib (7.4.0-1ubuntu1~18.04.1) ...
Setting up gcc-multilib (4:7.4.0-1ubuntu2.3) ...
root@308-server:/home/andrewng/workspace/zyy/myclass/csapp/datalab-handout# make
make: Nothing to be done for 'all'.
root@308-server:/home/andrewng/workspace/zyy/myclass/csapp/datalab-handout# ls
bits.c  bits.h  btest  btest.c  btest.h  decl.c  dlc  Driverhdrs.pm  Driverlib.pm  driver.pl  fshow  fshow.c  ishow  ishow.c  Makefile  README  tests.c
root@308-server:/home/andrewng/workspace/zyy/myclass/csapp/datalab-handout# ./btest
Score   Rating  Errors  Function
 1      1       0       bitXor
 1      1       0       tmin
 1      1       0       isTmax
 2      2       0       allOddBits
 2      2       0       negate
 3      3       0       isAsciiDigit
 3      3       0       conditional
 3      3       0       isLessOrEqual
 4      4       0       logicalNeg
 4      4       0       howManyBits
 4      4       0       floatScale2
 4      4       0       floatFloat2Int
 4      4       0       floatPower2
Total points: 36/36
root@308-server:/home/andrewng/workspace/zyy/myclass/csapp/datalab-handout#
```

4. 每次修改 bits.c 文件后，重新 make 时总会出错，具体 error 如下：

gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
/usr/bin/x86_64-linux-gnu-ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/6/libgcc.a when searching for -lgcc
/usr/bin/x86_64-linux-gnu-ld: cannot find -lgcc
/usr/bin/x86_64-linux-gnu-ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/6/libgcc.a when searching for -lgcc
/usr/bin/x86_64-linux-gnu-ld: cannot find -lgcc

解决方案：重新安装 libc6-dev-i386 即可，指令如下：

sudo apt remove libc6-dev-i386 --purge
sudo apt install libc6-dev-i386
#具体参考网址 https://www.jianshu.com/p/35da802ed737