DataLab实验报告

- 姓名：房向南
- 学号：2022201560

函数实现思路

bitXor

```
int bitXor(int x, int y) {

    return ~(~x & ~y) & ~(x & y);

}
```

- 思路：题目要求用~和& 实现异或，(~x & ~y)找出了x,y中都是0的位，(x & y)找出了x,y中都是1的位，再对这两个结果进行取反操作，使得x,y中都是0或者都是1的位标记为0，其余不相同的位标记为1。也就是说分开来找到了两个数中的相同位，让对应位置为0，从而实现了异或操作。

thirdBits

```
int thirdBits(void) {
    int x = 0x49;
    x = x | (x << 9);
    x = x | (x << 18);
    return x;
}
```

- 思路：题目要求生成一个隔两个二进制位就有一个二进制位为1的数。因为能使用的常数大小有限，所以使用了0x49，即二进制的01001001。为了满足题目要求，应该让这个数向左移九位，使得原先在最低位的1和上一个1相隔两个0。第二次左移同理。通过向左移位并且和原数进行或操作，生成了题目所需要的数。

fitsShort

```
int fitsShort(int x) {
    int y = x>>15;
    return !((y>>16)^y);
}
```

- 思路：题目要求判断一个数是否可以改写成一个有符号十六位整数。如果一个数可以按要求改写，那他的第十五位到第31位都是1或者都是0。于是先将这个数右移十五位得到一个所谓的“符号位”，然后将y继续右移16位得到该数原本的符号位。如果这个数能按要求改写，那么它一定符合前面陈述的条件。即x左移十五位和x左移31位异或能得到0，如果不符合要求，则得到一个非零数。再用！将其映射到题目要求的返回值即可。

isTmax

```
int isTmax(int x) {
    volatile int y=x+1;
    return (!(y+y))^(!y);
}
```

- 思路：题目要求判断一个数是不是Tmax。本题利用了Tmax的一个性质，即加一之后变成Tmin，再将两个Tmin相加即可得到0。但是这样做需要排除-1，如果x是-1，那么y就是0，!y就是1。利用这一点排除了-1的影响。

fitsBits

```
int fitsBits(int x, int n) {
    int m=33+ ~n ;//这个是32减去n的意思 32+(~n+1);
    int mask = (x << m) >> m;//通过x左移m位再右移m位
    return !(x ^ mask);
}
```

- 思路：题目要求判断x是否能表示为n位整数。首先令m=32-n，并通过左移m位再右移m位使x除了低n位以外全部变为符号位，如果x符合要求，那他和这个mask的异或就是0，反之是1。

upperBits

```
int upperBits(int n) {
  return ((!!n)<<31)>>(n+(~0));
}
```

- 思路：题目要求生成一个高n位全部是1的数。基本思路是将1左移31位得到Tmin，再右移n-1位得到题目要求的数。但是n为0时这样做会得到-1，于是利用两次!将不为零的n和为零的n区分开。

anyOddBit

```
int anyOddBit(int x) {
    int n = 0xAA;
    int m=n|(n<<8);
    int p=m|(m<<16);
    return !!(x & p);
}
```

- 思路：为了检查x是否至少有一个奇数位，利用了常数0xAA，也就是10101010这个允许使用范围内奇数位全部为1的数。接着通过两次移位得到10101010101010101010101010101010。将这个数与x做按位与操作，如果x至少有一个奇数位不为0，那么按位与的结果就是一个非零数。最后通过两次!将这个值映射到题目要求的返回值。

byteSwap

```
int byteSwap(int x, int n, int m) {
    int   nn =  n << 3;
    int   mm =  m << 3;
    int p = ( ( x >>  mm ) ^ ( x >>  nn) ) & 0xFF;
    return x ^ ( ( p <<  mm) | ( p  <<  nn) );

}
```

- 思路：题目要求交换第n位和第m位的内容。首先把m和n转换为相应的位偏移，也就是乘以8。之后把x第n个byte和第m个byte右移至最低位然后异或后用0xFF取出。之后把这个值分别左移至相应的byte上，用按位或进行组合后再和x进行异或。本道题最关键的一个点就是利用了$a^b a=b$这一特性。

absVal

```
int absVal(int x) {
    int mask = x >> 31;
    return (x + mask) ^ mask;
}
```

- 思路：首先获取这个函数的符号位，记录在mask中。如果x是正数，mask就是0，x为负数时mask是-1。所以当x为正数时，x + mask就是x本身，再异或mask也是x本身。当x是负数时，x + mask是x-1，再异或mask就是x-1的按位取反。通过mask来区分正数与负数，使返回的总是绝对值的值。

divpwr2

```
int divpwr2(int x, int n) {
    int isNeg = x >> 31;
    int a = ((isNeg & 1) << n) + isNeg;
    return (x + a) >> n;
}
```

- 思路：首先获取符号位，储存在isNeg中。如果是正数，isNeg是0；如果是负数，isNeg就是-1。将isNeg按位与1取出最后一位，左移n为后加上isNeg的值制造出偏移量。正数的偏移量是0，右移n位就是正常向下取整。负数通过偏移量进行向0取整的操作。

float_neg

```
unsigned float_neg(unsigned uf) {
unsigned ans = uf^0x80000000;
    unsigned nan = uf&0x7fffffff;
    if (nan>0x7f800000)
        return uf;
    return ans;
}
```

- 思路：先通过将这个浮点数异或0x80000000将符号位取反。然后取出除了符号位之外的阶码和尾数，如果这个数是nan，那取出的值就大于0x7f800000。如果不是nan，就可以直接返回符号位取反后的浮点数。

logicalNeg

```
int logicalNeg(int x) {
  return ((x | (~x + 1)) >> 31) + 1;
}
```

- 思路：~x+1是-x，x异或-x得到的数中最高位和x是否非零有关，如果x为0，最高位就是0，如果x不为0，最高位就是1。右移31位得到0或-1，再加上一得到题目要求的返回值。

bitMask

```
int bitMask(int highbit, int lowbit) {
    int a = ~0;
    int b = a << highbit;
    int x = (a << lowbit) & (~(b << 1));
    return x;
}
```

- 思路：让a等于-1，a左移highbit相当于生成一个低highbit为0，剩余位为1的数。a左移lowbit同理。将b左移一位后取反得到低highbit+1位是1，其余为0的数，与a左移lowbit按位与得到一个从lowbit位到highbit位的掩码。

isGreater

```
int isGreater(int x, int y) {
    int ny = ~y;
    int v1 = x & ny;
    int v2 = x ^ ny;
    int v3 = x + ny;
    int v4 = (v2 & v3) | v1;
    return !(v4 >> 31);
}
```

- 思路：x 和 ~y 的按位与得到一个整数，它的二进制表示中，y 中为1的位在 x 中都为0，其他位保持不变。x 和 ~y 的按位异或，得到一个整数，它的二进制表示中，y 和 x 中不同的位为1，相同的位为0。x 和 ~y 相加，得到一个数 v3。在 v3 中，y 中为0的位在 x 中保持不变，而 y 中为1的位在 x 中都变成了1。v2 和 v3 进行按位与运算，然后将结果与 v1 进行按位或运算。在 v4 中，v1 中为1的位被保持为1，v2 和 v3 中为1的位也被保持为1。将 v4 右移31位，得到符号位的值（0或1），然后取反操作。最终的结果是，如果 x 大于 y，v4 的符号位为0，函数返回1；否则返回0。

logicalShift

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

- 思路：((1<<31)>>n)<<1生成一个高n位为1的数。取反得到一个高n位为零的数m。将这个数与算术右移的结果按位与即可得到逻辑移位的结果。

satMul2

```
int satMul2(int x) {
    int nrml_rlt = x << 1;
    int juge = (x ^ nrml_rlt) >> 31;
    int tool = nrml_rlt >> 31;
    int Tmin = 1 << 31;
    return ((~juge) & nrml_rlt) | (juge & (Tmin ^ tool));
}
```

- 思路：nrml_rlt是x乘以二的结果，juge用于判断x的符号位是否发生改变。如果乘积溢出，juge 将等于-1；否则，juge 将等于0。tool是乘积的符号位，如果乘积为正，tool 将等于0；如果乘积为负，tool 将等于-1。(~juge) & nrml_rlt 这部分用于处理没有溢出的情况，即乘积没有超出范围。(~juge) 的结果是1，将符合条件的 nrml_rlt 返回。juge & (Tmin ^ tool) 这部分用于处理溢出的情况。(Tmin ^ tool) 的结果是32位整数的最大值，如果乘积为正，tool 为0，此时返回最大值；如果乘积为负，tool 为-1，此时返回Tmin。

subOK

```
int subOK(int x, int y) {
    int a = y + (~x);
    int b=y^a;//如果溢出，溢出的结果和y符号相同，最高位应该是0
    int c=x^y;//xy异号的话最高位是1.同号不会溢出最高位是0
    return ((c & b) >> 31) + 1;//同号直接返回1，表示不会溢出。异号如果没溢出就是1，溢出就是（-1+1
}
```

- 思路：x-y导致溢出只有两种情况，x正y负结果为负，x负y正结果为正。通过判断符号位可以判断x-y是否溢出。

trueThreeFourths

```
int trueThreeFourths(int x) {
    int x1 = x>>2;
    int x2 = x&3;
    int a = (x1+x1+x1);
    int s = (x>>31)&3;
    int b = (x2+x2+x2+s)>>2;
    return a+b;
}
```

- 思路：先将x右移两位，相当于乘以4，用3取出x的最后两位，相当于x除以4的余数。计算 x1 的三倍，由于右移两位相当于除以4，所以 x1 的三倍相当于除以4后再乘以3，即 x1 * 3 / 4。s为符号位右移31位后得到的结果的后两位。利用符号位对正负数进行区分。 计算 x2 的三倍，并且加上s，这样就相当于将 x2 乘以3。然后右移2位，相当于除以4。确保了向0取整。

isPower2

```
int isPower2(int x) {
    int s=x>>31;
    return  (!(x&(x+~0))) &(s+1) & !!x;
}
```

- 思路：首先获取符号位。对于x&（x-1）来说，如果x是2的n次方，x-1就是低n位为1，其余位为0的数，和x按位与可以得到0。由于题目说了没有负数符合题意，所以(s+1)用来排除负数的影响。!!x用来排除0的影响。

float_i2f

```
unsigned float_i2f(int x) {
    unsigned sign=0,shiftleft=0,flag=0,tmp;
    unsigned absx=x;
    if( x==0 ) return 0;
    if( x<0 ){
    sign=0x80000000;
    absx=-x;
    }
    while(1){
        tmp=absx;
        absx=absx<<1;
        shiftleft=shiftleft+1;
        if( tmp&0x80000000 ) break;
    }
    if( (absx & 0x01ff) > 0x0100 ) flag=1;//因为左移了直到最高位是1，这是判断它第九位以后是不是近
    if( (absx & 0x03ff) == 0x0300 ) flag=1;//0000 0011 1111 1111最后十位如果结果是00000011000
    return sign+(absx>>9)+((159-shiftleft)<<23)+flag;
}
```

- 思路：先判断x是否等于0，如果是直接返回0。然后将x<0时的符号位设置为1，值变为相反数。之后的while循环将这个整型数不损失值的情况下移到最高位。之后两个判断是为了之后给符号位和阶码移位时的近似。最后按位组装起来就好。

howManyBits

```
int howManyBits(int x) {
    int b16,b8,b4,b2,b1,b0;
    int mask = x >> 31;
    x = (mask & ~x) | (~mask & x);
    b16 = !!(x >> 16) << 4;
    x >>= b16;
    b8 = !!(x >> 8) << 3;
    x >>= b8;
    b4 = !!(x >> 4) << 2;
    x >>= b4;
    b2 = !!(x >> 2) << 1;
    x >>= b2;
    b1 = !!(x >> 1);
    x >>= b1;
    b0 = x;
    return b16 + b8 + b4 + b2 + b1 + b0 + 1;
}
```

- 思路：mask = x >> 31; 用于获取 x 的符号位，即最高位。如果 x 是负数，mask 将等于-1（即所有位都是1）；如果 x 是非负数，mask 将等于0。(mask & ~x) | (~mask & x) 将 x 的符号位翻转。如果 x 是负数，将所有位取反得到其绝对值；如果 x 是非负数，不做任何操作。之后的几行分别判断 x 的剩余的高16，8，4，2位是否有非零的位，并对应的记录下来。此时剩下的就是 x 的最低位，将其赋值给 b0。最后计算所有非零位的总数，加上1，即为整数 x 的二进制表示中包含的位数。

float_half

```
unsigned float_half(unsigned uf) {
unsigned expo, expo_7, expo_2, uf_new;
    expo = uf & 0x7F800000;
    expo_7 = uf & 0x7F000000;
    expo_2 = uf & 3;
    if (expo == 0x7F800000){
        return uf;
    }
    else if (expo_7){
        uf_new = uf + 0xFF800000;
    }
    else{
        uf_new = uf >> 1;
        if (expo_2 == 3){
            uf_new += 1;
        }
        if (uf & 0x80000000){
            uf_new += (0x40000000);
        }
    }
    return uf_new;
}
```

- 思路：提取浮点数的指数部分 expo，以及额外提取7位指数部分 expo_7，和尾数的后两位 expo_2。如果输入浮点数是无穷大或NaN，则直接返回输入浮点数，因为它们在除以2后仍然保持不变。对于非特殊情况，将浮点数的指数部分右移1位，相当于除以2。然后，如果尾数的后两位 expo_2 是11（即3），说明原浮点数的小数部分大于等于0.5，所以需要在右移后的结果上加1，实现四舍五入。如果输入浮点数为负数，需要将右移后的结果加上0x40000000，这是因为符号位会变为1，需要在结果的最高位加上1，以保持负数的值正确。返回新的浮点数的表示，实现了将输入浮点数除以2的操作。