DataLab 实验报告

- **BitXor**

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

由定义可知：a^b = (~a & b) | (a & ~b)

= (~a | a）&（~a | ~b）&（b | a）&（b | ~b ）

= ~(a & b）& ~(~a & ~b)

- **ThirdBits**

```c
int thirdBits(void) {
  int x = 0b01001001 | 0b01001001<<9;
  int x1 = x;
  int y = x<<15;
  int z = y | x;
  return z;
}
```

0b01001001 为最小 8 位的情况

x 为后 16 位情况

z 为 32 位的情况

- **fitsShort**

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

对 x 左移 16 位再右移 16 位得到 y，

若 x 和 y 相等则 x 满足条件返回 1，不然则返回 0

- **IsTmax**

```c
int isTmax(int x) {

  int y = x + 1;
  int z = y + y;
  int a = !y;

  return !(z | a);
}
```

本题主要关注两个点，x=Tmax 时和 x=-1。当且仅当此时，z = 0。

又有 x=-1 时 a=1； x=Tmax 时 a=0；

故可由 z 和 a 共同确定 x 是否 Tmax

- **Fitsbits**

```c
int fitsBits(int x, int n) {

  int y =  x >> (n + 63);
  return !((y+1)>>1);
}
```

利用 ub(Undefined Behavior)获得 y，即 x 右移 n-1

此时若 x 满足条件则 y 为 0(x>=0 时)或 0xFFFFFFFF(x<0)

然后 y+1 再右移 1 位，仅当 x 满足条件时结果为 0。

- **UpperBits**

```c
int upperBits(int n) {
  int x = ~n + 1;
  int y = x;
  int z = x>>31;
  return z<<y;
}
```

x = -n;

z = 0xFFFFFFFF(x != 0 时) 或 z = 0(x == 0 时)

结果返回使用 ub，即 z<<(32-n)

- **anyOddBit**

```c
int anyOddBit(int x) {
  int x1 = 0b10101010;
  int y = x1 | x1<<8;
  int z = y | y<<16;

    return !!(x&z);
}
```

x1，y 都是为了得到 z，即得到 0bAAAAAAAA

若 x 的奇数位有 1，则 x&z 一定不为 0；反之一定为 0。由此再用!!即得到最终结果

- **ByteSwap**

```c
int byteSwap(int x, int n, int m) {
    int a = n<<3;
    int b = m<<3;
    int k = ((x>>a)^(x>>b))&0xFF;

    return x^(k<<a|k<<b);
}
```

记 x 为 0xA3A2A1A0（Ai 代表一个字节）

则 k = 0xAn ^ 0xAm

由 0^x = x; x^x^y = y;可得 x^(k<<a) 为仅将 n 处的字节换成 m 处的，x^(k<<b)同理将 m 的字节换成 n 的

- **AbsVal**

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

当 x>=0 时，y = 0，（x+y）^y = x

当 x<0 时，y = 0xFFFFFFFF，（x+y）^y = ~(x-1) = ~x+1 = -x

- **Divpwr2**

```c
int divpwr2(int x, int n) {
  int y = (x>>31);
  int z = y^(y<<n);
  return (x+z)>>n ;
}
```

当 x>=0 时，y=z=0，结果为 x>>n

当 x<0 时，由于负数的移位取整方向和除法不同，需要加上特定的数之后再移位，即加上 z = $2^n - 1$。此时 y = 0xFFFFFFFF，y<<n 再和 y 异或正好为$2^n - 1$。

- **Float_neg**

```c
unsigned float_neg(unsigned uf) {
  unsigned x = 0x80000000;
  if((uf<<1) > 0xFF000000)
    return uf;

  return uf^x;
}
```

当(uf<<1) > 0xFF000000 时，判定为 NaN，return 原数

不然则与 0x80000000 做异或，使第一位符号位取反，然后 return

- **logicalNeg**

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

y= -x

当 x 不为 0 时，x 与 y 必有一个负数，则 z 必定等于 0xFFFFFFFF

而当 x=0 时，z = 0

最后返回 z+1 正好为结果

- **bitMask**

```c
int bitMask(int highbit, int lowbit) {
  int x = ~0;
  int y = x + (2<<highbit);
  int z = x << lowbit;
  return( y & z);
}
```

记最后结果为 0b 000···111···000···

则 y = 0b 000···111111···

z = 0b11111111···000

两者与运算即为结果，并且当 high<low 时正好返回 0

- **isGreater**

```c
int isGreater(int x, int y) {
  int a = (x^y)>>31;
  int b = x + ~(y|a);
  return (b>>31) + 1;
}
```

a 用来判断 x，y 是否异号（因为异号相减需考虑溢出，同号不需）

若 x y 同号，b = x-y-1，然后通过符号位区分结果

若 x y 异号，b = x, 此时若 x 为正则 x>y，x 为负则 x<y

- **logicalShift**

```c
int logicalShift(int x, int n) {
  int y = x>>n;
  int z = (2<<(~n))&y;
  return y+z;
}
```

2<<~n 即 2 右移 31-n 位

若 y 为正数，则 z = 0，返回 y

若 y 为负数，则 z = 2<<~n。y+z 使 y 因为移动产生的 1 全变成 0

- **satMul2**

```c
int satMul2(int x) {
  int ans = x <<1;
  int overFlow = ( x ^ ans) >> 0x1F;
  return ( ans >>  overFlow  ) + ( overFlow << 0x1F ) ;
}
```

ans 为 2*x，若无溢出，则直接 return ans

若有溢出，则 overflow 为 0xFFFFFFFF,

此时，若 x>0，ans 右移 overflow(即 31 位)为 0xFFFFFFFF，加上 0x80000000 即为 Tmax。同理若 x<0，ans 右移 31 位为 0，加上 0x80000000 即为 Tmin

- **subOK**

```c
int subOK(int x, int y) {
    int a = ~x+y;
    int b = x^y;
    int c = a^y;
  return ((b & c) >> 31) + 1;
}
```

a = y - x -1。b 用来判断 x y 是否异号。c 用来判断 a 与 y 是否异号

当 x 与 y 同号时不可能溢出，则 b 的符号位为 0，返回 1

当 x 与 y 异号时，b 的符号位为 1：

　　若 x 为正 y 为负：

　　　　a 若溢出为正，c 的符号位为 1，最终返回 0；

　　　　a 若未溢出则 c 符号位为 0，最终返回 1

　　若 y 为正 x 为负：

　　　　a 若溢出为负，c 的符号位为 1，最终返回 0；

　　　　a 若未溢出则 c 符号位为 0，最终返回 1

- **trueThreeFourths**

```c
int trueThreeFourths(int x) {
  int a = ~x;
  int b =(a >> 2)  + x;
  int c =(a >> 31) & x;
  return b + (!(c & 3));
}
```

a = -x-1

b = a/4 + x = $\frac{3}{4}x$

但此时若 x 为正且第三位不为 1 时或 x 为负会造成误差，故构造 c

当 x<0 时，c = 0，return b+1;

当 x>=0 时，c = x

当 x 第三位不为 1 时，return b+1

当 x 第三位为 1 时，return b

- **isPower2**

```c
int isPower2(int x)
{
  int y=x+~0;
  return !((x&y)|(y>>30));
}
```

y = x - 1

当 x 是 2 的幂时，x 形同 0b000…10000…

y 形同 0b000…01111…

故 x&y = 0，但是我们还要排除 x 是负数和 x 是 0 的干扰，

注意到当 x 是负数(不是 Tmin)或 x 是 0 时，y<0，(x&y)|y 可以排除干扰

注意到当 x>0 时，x 若是幂，y 最大为 0x1FFFFFFF，则 y 应至少右移 30 位才能消除对正确结果的影响

注意到当 x = Tmin 时，y = 0x7FFFFFFF，则 y 右移最多 30 位才能排除干扰

综上应该返回!((x&y)|(y>>30))

- **Float_i2f**

```c
unsigned float_i2f(int x) {
  unsigned ux = x;
  unsigned s_and_e = 0x4E800000;
  int flag = 0;
  int y = 0;
  if (x) {
    while (1) {

      if (ux & 0x80000000) {
        if(flag)
          break;
        ux = -x;
        s_and_e = 0xCE800000;
      }
      else {
        ux <<= 1;
        s_and_e -= 0x00800000;
      }

      flag = 1;

    }

    if (ux & 0x80 && ux & 0x17F)
      y = 1;

    return s_and_e + (ux >> 8) + y;
  }

  return 0;
}
```

易知把 x 变成浮点数，E 的取值在 0~30。则把 s_and_e 初始化为 0x4E800000，代表符号位为 0，exp 为 10011101(即 E 为 30)。

然后进入循环，flag 仅仅用来辅助第一次循环，若第一次循环发现 x<0，则把 x 取正然后把 s_and_e 的符号位赋值为 1

之后的循环就是不断让 ux 左移，每成功左移一次就让 exp-1，直到 ux 的最左位为 1，但是注意到这样会使 exp 多减一次 1

所以让 ux 右移 8 位空出符号位和 exp 的左 7 位，然后加上 s_and_e 正好让 exp 补上多减的 1，同时也满足 frac 省略第一个 1 的特性

但是这样做会造成精度的损失，寻找规律发现当 ux&0x80 与 ux&0x17F 同时满足时，结果需要加 1

- **howManyBits**

```c
int howManyBits(int x) {
    x = x ^ (x << 1);
    int n5 = !!(x >> 16) << 4;
    x = x >> n5;
    int n4 = !!(x >> 8) << 3;
    x = x >> n4;
    int n3 = !!(x >> 4) << 2;
    x = x >> n3;
    int n2 = (!!(x >> 2) << 1) + 1;
    x = x >> n2;
    int n1 = x & 1;
    return n5 + n4 + n3 + n2 + n1;
}
```

由于返回数 1<= n <= 32,所以 n-1 用二进制表示有 5 位，n1 到 n5 就分别是最低位到最高位，最后再加 1 就是 n

第一步异或是为了消除负数左边全是 1 的影响

接下来的操作相似，即检查 x 的左$2^i(0 \le i \le 4)$位是否有数字，有则 ni 赋值成再乘2^{i-1}，同时移动 x 以进行下一步计算

由于最后的返回值要+1，我们发现如果把这个 1 加在 n2 上并使 x 接下来多移动一位，就可以直接通过 x&1 来判断 n1，节省了操作符

- **Float_half**

```c
unsigned float_half(unsigned uf) {

  unsigned x=uf<<1;
  if (x >= 0xFF000000)
    return uf;

  if (x <= 0x01FFFFFE){
    int f = uf;
    int y = f>>1;
    int z = y & f & 1;
    return (y&0xBFFFFFFF)+z;
  }

  return uf-0x00800000;
}
```

x 为 uf 去掉符号位的结果，用来判断不同情况

x>=0xFF000000 时，exp 为 FF，uf 即为无穷或 NaN，返回 uf。

X<=0x01FFFFFE 时，exp 为 1 或 0，此时 uf 除以 2 后 exp 为 0，frac 为原来的右移 1 位。因此我们让 y=f>>1，由于 f 是 int 类型，y 会自动保留符号位，我们只需让 y&0xBFFFFFFF 以消除符号位右移进入 exp(即从左向右第二位)的影响。然后，考虑舍入会有 1 的误差，令变量 z，当 f 的右边第一位和第二位都为 1 时，我们让结果加上 1 以平衡误差

除了上述两种情况外，其他时候我们只需让 exp-1 即可，即 uf-0x00800000