Csapp Lab1：datalab

实验环境：Kubuntu20.04 64位

前置条件：$ sudo apt update

$ sudo apt install build-essential（该命令将安装一堆新包，包括gcc，g ++和make。）

$ sudo apt install gcc-multilib（既能生成32也能生成64位两种格式，这样在64位机器上生成32位的项目 也能跑）

1. 只用~和&运算符实现异或操作

```
/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 *   Max ops: 14
 *   Rating: 1
 */
```

分析：

- 主要考察了**数字电路**和**离散数学**相关知识。
- 首先列出**真值表**，画出**卡诺图**，然后得到公式$A\hat{}B=A'B+AB'$
- 之后根据离散数学的德摩根对上述公式进行变形

简单推导一下

真值表:

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

根据真值表可以画出卡诺图（卡诺图有最小项表达式和最大项表达式）

卡诺图：因为markdown好像不支持斜头表格，就用下面这种方式表示

	B：0	B：1
A：0	A^B：0	A^B：**1**
A：1	A^B：**1**	A^B：0

对于卡诺图的最小项表达式我们只需要观察结果为1的部分

$F = AB' + A'B$

反之，对于卡诺图的最大项表达式，我们只需要观察结果为0的部分

$F = (A' + B')(A + B)$

得到上述公式后，就可以任意拿其中一个公式然后根据离散数学的德摩根律进行推导即可。

$F = AB' + A'B = ((A' + B)(A + B'))' = ((AB')'(A'B)')'$

好了，到此为止，此题就顺利解决了。

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

2. **返回最小的二进制补码整数（只能用 ! ~ & ^ | + << >>运算符）**

```
/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
```

分析：

- 考察补码表示法

不必多说，在32位系统下，补码表示法的最大值：$2^{31} - 1$，最小值为：2^{31}

附上答案：

```
int tmin(void) {
    return 1<<31;
}
```

3. **判断x是不是最大的二进制补码整数，是返回1，否则返回0（只能用 ! ~ & ^ | +运算符）**

```
/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *     and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 1
 */
```

在第二题中已经分析过了，在32位系统下，补码表示法的最大值：$2^{31} - 1$

但是本题加强了对运算符的限制，因此还需要进行稍微变通一下。

举个例子：5的补码表示：0101，-5的补码表示1010+1=1011

分析最大值的情况，倘若x是最大值，那么x+1=最小值，产生了溢出，因此可以很容易得到等式

x+1=最小值，最小值+1=~x+1，所以可得最小值=~x，所以最终可以得到x+1=~x

此处需要特别注意一个情况，当x为**-1**或者**0**的时候上述公式都成立，所以需要排除这两个的影响

利用!!(x+1)能屏蔽掉**-1**的影响，但是屏蔽不了0的影响

屏蔽0的影响：x能屏蔽0的影响

直接附上答案

```
int isTmax(int x) {
    return !((x+1)^(~x)) & (!!(x+1) & x);
}
```

4. **对于奇数位置的比特位若全是1则返回1，否则返回0（只能用! ~ & ^ | + << >>运算符）**

```
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 *   where bits are numbered from 0 (least significant) to 31 (most significant)
 *   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
```

要求：只能用0-255的常数

例子：allOddBits(0XFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1

分析：首先很容易想到，判断奇数位置的比特位，只需每次右移与1取与，看结果即可，但是这种情况很麻烦

需要构造出常数：10101010 10101010 10101010 10101010，之后用x与该常数进行过滤

用了一个小技巧：x^x=0

```
int allOddBits(int x) {
  int a = 0xAA;
     int b = a << 8 | a;
     int c = b << 16 | b;
     int d = x & c;
     return !(d^c);
}
```

5. **取负操作**

```
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
```

例子：negate(1) = -1

负数的补码表示为其对应正数的补码表示按位取反最后再+1，例如在4位的情况下：

5的补码表示：0101，-5的补码表示1010+1=1011

因此，对于只需对x进行按位取反最后再+1即可实现取负操作

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

6. **判断一个数是不是ASCII码（只能用! ~ & ^ | + << >>）**

```
/*
 * isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')
 *   Example: isAsciiDigit(0x35) = 1.
 *            isAsciiDigit(0x3a) = 0.
 *            isAsciiDigit(0x05) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 3
 */
```

例子：x在0x30 <= x <= 0x39这个范围内就是，否则就不是

首先写出上界和下界的位级表示：0011 0000，0011 1001，可以发现第4-7位的位级表示都是0011

条件1：二进制位第4-7位的位级表示必须为0011

条件2：二进制位第0-3位的位级表示在0000-1001范围之间，观察发现只需要对低四位的数进操作，然后在第3位为1的前提下，第1-2位有出现1的情况，那么范围就超过了1001，不符合条件2

整理一下，便可以得到下面答案

```
int isAsciiDigit(int x) {
int lc = 0x3;
int condition1 = !!((x >> 4) ^ lc); //如果满足条件1为0,否则为1，再次利用了!!运算符的技巧
int low = x & 0xF; //取出x低四位的位级表示,只要低四位的位级表示比1001要大,那么就不对了

int condition2 = low >> 3; //原来是(low >> 3) & 1.但是运算符个数超了,就把&去了
int tmp = low;
tmp = tmp & 0x6; //与0110做与
tmp = !!tmp; //如果tmp不为0并且condition2为1那么范围就超过了1001,不符合条件
condition2 = (!condition2 | (condition2 & !tmp));
return !condition1 & condition2;
}

```

7. **条件表达式（只能用! ~ & ^ | + << >>）**

```
/*
 * conditional - same as x ? y : z
 *   Example: conditional(2,4,5) = 4
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 16
 *   Rating: 3
 */
```

分析：x ? y : z

分析技巧：利用!!X判断x是否为0，之后构造出-1，0的补码是0，1求补之后是-1，-1的补码表示全是1

比较简单，只需要用与运算判断x是否为0即可

```
int conditional(int x, int y, int z) {
    int a = !!x;//判断x是否是0,x若为0,a为0,否则a为1
    a = ~a + 1;
    return (a&y) | (~a&z);
}
```

8. 判断小于等于（只能用! ~ & ^ | + << >>）

```
/*
 * isLessOrEqual - if x <= y  then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
```

例子：x <= y，返回1，否则返回0

对式子进行变形：y-x>=0，所以只需对结果抽出符号位进行判断即可，但需要考虑正溢出的问题，

比如4位数，y=-5，x=4，y-x>=0，在x与y异号的时候是有可能会产生溢出的，比如-5-4>0，这种情况下需要特判一下y的符号位

- 条件1：y与x同号，结果的符号位为0
- 条件2：y与x异号，y符号位为0（x,y异号的情况下相减有可能会产生正溢出,因此要加上y的符号位为0这个限定条件）

上面两个条件取得是或的关系

```
int isLessOrEqual(int x, int y) {
    int mx = ~x+1;//-x
    int result = y+mx;//y-x
    int rsign = (result >> 31) & 1;
    int xsign = (x >> 31) & 1;
    int ysign = (y >> 31) & 1;
    return (!(xsign ^ ysign)& !rsign) | ((xsign^ysign)&!ysign);
}
```

9. 用其他运算符实现逻辑取反（!）的功能

```
/*
 * logicalNeg - implement the ! operator, using all of
 *              the legal operators except !
 *   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
```

分析：注意到一个比特位和1取异或得到的是这个数的取反，例如0^1=1, 1^1=0

- 对于为0的数，那么就取反取出最高位即可
- 不为0的数，如果是正数，利用其符号位为0；如果是负数，利用其符号位为1

所以对上述两个条件进行观察可以发现((x | (~x+1))可以涵盖为0和不为0的情况，若x为0，则((x | (~x+1))的最高位一定也为0，否则最高位一定是1

上面两个条件是或的关系，关键是如何对这两个条件进行合并，对((x | (~x+1))算数右移31位后发现若x为0,则结果是0；若x不为0，那么结果为-1

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

10. 计算x最小有多少位数（用二进制补码表示法）

```
/* howManyBits - return the minimum number of bits required to represent x in
 *             two's complement
 *  Examples: howManyBits(12) = 5
 *            howManyBits(298) = 10
 *            howManyBits(-5) = 4
 *            howManyBits(0)  = 1
 *            howManyBits(-1) = 1
 *            howManyBits(0x80000000) = 32
 *  Legal ops: ! ~ & ^ | + << >>
 *  Max ops: 90
 *  Rating: 4
 */
```

例子：howManyBits(12) = 5 //01100

分析：对于正数而言，从高往低找找到第一个0；对于负数而言，从高往低找到第一个1即可。

利用**二分**的思想去找，对0-31位进行二分，对32位数进行二分即可

x是32位的数，先用a1=!!(x >> 16)，判断x的高16位中是否含1。如果含1，那么至少需要>16位数；否则至多需要16位数，于是可以根据a1的结果来判断需要有多少位的数。分析发现，在第一次二分的时候，a1 << 4即可实现所需要的位数（a1=0, 则a1<<4=0，此时不需要高16位的数；a1=1，则a1<<4=16，此时至少需要>16位）

对于正数来说，我们需要找的是最高的0，对于负数，对其进行取反，然后就等价为找最高的0，之后再统一加上符号位即可。

人间疑惑：-1的最少位数为什么是1？？？？？？这题参考了网上的做法，不是很懂

思考了一下大概是因为 比如-2可以用10表示，但是再32位补码表示中-2被表示为1111...1110，有很多1其实是多余的，所以对负数的情况要进行取反操作

```
int howManyBits(int x) {
     int sign = (x >> 31); //是算术右移，符号位会跟着一块移动
     int a1, a2, a3, a4, a5, a6;
     x = (sign & ~x) | (~sign & x);

     a1 = !!(x >> 16);
     a1 = a1 << 4;
     x = x >> a1;

     a2 = !!(x >> 8);
     a2 = a2 << 3;
     x = x >> a2;

     a3 = !!(x >> 4);
     a3 = a3 << 2;
     x = x >> a3;

     a4 = !!(x >> 2);
     a4 = a4 << 1;
     x = x >> a4;

     a5 = !!(x >> 1);
     a5 = a5 << 0;
     x = x >> a5;

     a6 = !!(x >> 0);
     a6 = a6 << 0;

    return a1 + a2 + a3 + a4 + a5 + a6 + 1;//最后加1是为了加上符号位
}
```

11. 计算浮点数的规模

```
/*
 * floatScale2 - Return bit-level equivalent of expression 2*f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representation of
 *   single-precision floating point values.
 *   When argument is NaN, return argument
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
```

题目说要返回2*f

分析：要知道浮点数在计算机当中的表示方法，浮点数在计算机中的表示方法是IEEE 754表示法

分为符号位s（1位），exp（8位），尾数M（23位）

然后依次把s，exp，M的数值通过位级运算操作取出来即可，之后再根据规格化和非规格化的定义进行相应的操作。

- 对于规格化数：
  - 如果exp=255，尾数为0，表示INF；
  - 如果exp=255，尾数非0，表示NaN；
- 对于非规格化数：
  - 如果exp=0，尾数为0，表示0
  - 如果exp=0，尾数不为0，表示接近0的数

本质还是在考察IEEE 754浮点表示方法

```
unsigned floatScale2(unsigned uf) {
    int exp = (uf & 0x7F800000) >> 23;
    int sign = (uf >> 31) & 1;
    int m = uf & 0x7FFFFF;
    if (exp == 0xFF) return uf;
    else if (exp == 0) {
        m <<= 1;//*2小数点左移，相当于尾数整体往左挪了一位
        return (sign << 31) | (exp << 23) | m;
    }
    else {
        exp++;
        return (sign << 31) | (exp << 23) | m;
    }
}
```

总结：对于本题来说，最关键的是将规格化数和非规格化数的*2分开考虑。

在非规格化数的范围内，exp始终保持为0，此时*2就通过移动尾数来实现；

在规格化数的范围内，exp>0，此时*2就可以直接通过exp++来实现；

11. 浮点数转为整数

```
/*
 * floatFloat2Int - Return bit-level equivalent of expression (int) f
 *   for floating point argument f.
 *   Argument is passed as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point value.
 *   Anything out of range (including NaN and infinity) should return
 *   0x80000000u.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
```

需要知道IEEE 754浮点数表示法，符号位，exp，以及尾数

int的范围最大是$2^{31}-1$

主要是根据指数和23的关系来进行尾数的移位，特别需要注意的是要考虑到尾数的**舍入**问题，这也是本题的难点

```
int floatFloat2Int(unsigned uf) {
 //将浮点数转为整数，int的表示范围(-2^{31} ~ 2^{31}-1)
    int sign = (uf >> 31) & 1;//符号位
    int m = uf & 0x7fffff; //尾数
    int exp = (uf & 0x7f800000) >> 23; //exp
```

```
    int E = exp - 127; //浮点数的指数
    if (E >= 31) return 0x80000000; //NAN或者INF
    else if (E < 0) { //对于浮点数中的小数,直接置为0即可
        return 0;
    }
    else if (E >= 23) {
        m = m | 0x800000; //真正的尾数
        m <<= (E - 23);
    }
    //需要考虑舍入误差的问题,出问题的应该在这部分
    else { //E < 23
        m = m | 0x800000;
        int tmp = 1 << (23 - E - 1);
        int ttmp = tmp - 1;
             int next = m & tmp;
        int post = !!(m & ttmp);
        if (next == 0) {
            m = m >> (23 - E);
        }
        else if (next == 1 && post) {
            m = m >> (23 - E);
            m += 1;
        }
        else { //向偶数舍入
            m = m >> (23 - E);
            if (m & 1) {
                m += 1;
            }
        }
    }
    if (sign) {
        return -m;
    }
    else {
        return m;
    }

}
```

11. 浮点数的次幂

```
/*
 * floatPower2 - Return bit-level equivalent of the expression 2.0^x
 *   (2.0 raised to the power x) for any 32-bit integer x.
 *
 *   The unsigned value that is returned should have the identical bit
 *   representation as the single-precision floating-point number 2.0^x.
 *   If the result is too small to be represented as a denorm, return
 *   0. If too large, return +INF.
 *
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while
 *   Max ops: 30
 *   Rating: 4
 */
```

分析：exp的位模式既不全为0，也不全为1。E=exp-Bias，对于单精度数值而言，Bias=127，

对于规格化数：$exp \neq 0$，对于非规格化数：$exp = 0$

- 规格化数：$1 \leq exp \leq 254$，所以$-126 \leq E \leq 127$，尾数为0，当指数为-126且尾数全为0的时候取到最小值，当指数为127且23位尾数全为1的时候取到最大值，最小值：$1.0 * 2^{-126}$，最大值：$1.11111... * 2^{127}$，所以能够表示的浮点数范围2^{-126}~$2^{127} * (2 - 2^{-23})$
- 非规格化数：exp=0，所以E=1-Bias=-126，当尾数全为0的时候取到最小值$0.0...01 * 2^{-126}$，当尾数全为1的时候取到最大值$0.1111 * 2^{-126}$，所以能够表示的浮点数范围2^{-149}~$2^{-126} * (1 - 2^{-23})$

```
unsigned floatPower2(int x) {
  if (x < -149) {//小于非规格化数的表示范围
       return 0;
   }
   else if (x > 127) { //大于规格化数的表示范围
       return 0x7F800000;//23位尾数全为0，exp全为1
   }
   else if (x >= -126) { //考虑规格化数的最小值
       return (x + 127) << 23;
   }
   else {//考虑非规格化数的范围（-149 <= x < -126）
       return 1 << (x+149);
   }
}
```

总结

- csapp lab1总体难度还是适中的，其中howManyBits是比较难的，我也是看了不少教程之后才理解，还有计算一个数用补码表示法需要的最小位数是比较灵活的，最开始没有转过弯来，一直不理解为什么-1可以只用一位数来表示
- 浮点数部分的三题特别有意思，特别是第二题，需要考虑舍入误差，这个真的容易忽视，并且需要向偶数舍入
- 对浮点数的数据表示有了稍微更加深刻的理解
- 加油，冲冲冲