task-1

本任务要求我们仅使用 ~ 以及 & 来实现异或运算

考虑两个bit， x 和 y，那么 $x \oplus y = 1\ if\ x! = y$, $x \oplus y = 0\ if\ x == y$

我们假设最终得到的式子的形式为 $P\&Q$, 其中P, Q 均为 x, y 的一个函数。

- 当$x! = y$ 时， $x\&y = 0,\ \ (\sim x)\&(\sim y) = 0$, 然后对二者都取反就得到了两个1

那么 $\sim (x\&y)\& \sim (\sim x)\&(\sim y) = 1$

- 当$x == y$ 时，$x\&y = 1\ or\ 0,\ \ (\sim x)\&(\sim y) = 1\ or\ 0,$ 并且二者一定是不相等的

那么 $\sim (x\&y)\& \sim (\sim x)\&(\sim y) = 0$

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

task-2

本任务要求我们输出最小的two's complement integer

由补码的定义，我们只需让符号位为1，其余位置为0即可

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

task-3

本任务要求我们输出1， 如果x是最大的two's complement integer

最大的补码int为符号位为0， 其余为全部为1的一个数，不妨设为T

我们可以发现这样一个性质 $\sim (T+1) = T$

但是我们能用这条性质来找出T 呢?

考虑-1, 其补码为 32位全为1， 因此其加一后 为 $1 << (32)$

此时发生了溢出， 因此通过截断后为 31全为0 的数，其依然满足$\sim (x+1) = x$, 因此我们需要特判 $x = -1$ 的情况

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

其中当 $x \oplus\ (\sim (x+1))$ 等于0 时，x 只能为 T 或者是 -1 . 如果 $x = -1$, 此时 $!(\ x) = 1$

此时$!((x \oplus (\sim (x+1)))|(!(\sim x))) = 0$, 因此可以排除-1 这种情况.

task-4

本任务要求我们输出1， 如果该integer的每个奇数位都是1， 并且bit的下标从0开始。

我们需要思考怎么把一个数的奇数位全部取出来?

我们容易写出一个32并且每个奇数位上数字为1的数为0xAAAAAAAA，

但是在本实验的最开头有一个要求： Integer constants 0 through 255 (0xFF), inclusive. You are not allowed to use big constants such as 0xffffffff. 这意味着我们不能直接使用 0xAAAAAAAA， 但这个数可以看成是周期生成的，因此我们很容易可以构造出来。

```
int allOddBits(int x) {
  int y = (0xAA << 8) + 0xAA;
    y = (y << 16) + y; //构造出了0xAAAAAAAA
    return !((x & y) ^ y);
}
```

其中 $x\&y$ 相当于取出了 x 的所有奇数位

task-5

本任务需要我们输出一个数的相反数，不能使用负号。

这题考察了补码的性质，$x+\sim x=-1$，那么$-x=\sim x+1$

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

task-6

本任务要求我们输出1，如果 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')

考虑判断 $0x30-x<=0<=0x39-x$, 其中减法可以通过加相反数来实现。

分别考虑两个不等号：$0x30-x<=0$, $0x39-x>=0$ 我们都只需去看 符号位即可.

```
int isAsciiDigit(int x) {
    int a = x + (~0x30 + 1); % x-0x30
    int b = 0x39 + (~x + 1); % 0x39-x
    int c = 0x1 << 31; // 取符号位的mask
    return !(a & c) & !(b & c);
}
```

task-7

本任务要求我们实现 x ? y : z 。

核心在于我们怎么去判断 x 是否为0 。用! 就好了。

如果$x!=0$, 那么$!x=0$, 如果$x==0$, 那么$!x=1$

令$flag =!x$，下面考虑如何实现对y, z 的选择。

我们可以使用一个全为1的mask， 那么$mask\&y=y, mask\&z=z$

因此这个mask需要满足当flag=0时候，mask全为0，当flag=1时，mask全为1。如何构造mask?

令$mask = \ \ flag+1$ 即可，我们可以验证其是满足条件的。

```
int conditional(int x, int y, int z) {
  int flag = !x;
    int mask = ~flag + 1;
    return (~mask & y) | (mask & z);
}
```

task-8

本任务要求我们实现：if x <= y then return 1, else return 0

当x, y 同号时，我们可以直接考虑$x-y$, 并看其符号位即可。

但是如果x, y 不同号时，当我们使用$x-y$, 需要考虑overflow的问题， （比如x>0, y<0)

但此时我们可以直接判断大小了，正数肯定大于等于负数。除此之外特判 $x-y=0$ 的情况

```
int isLessOrEqual(int x, int y){
    int signx = (x>>31)&1;
    int signy = (y>>31)&1;
    int signjudge = signx ^ signy; //%signjudge=1 异号
    int ans1 = signx &(!signy); //signx=1, signy=0, ans1=1; 其余情况均为0.
    int c=x+(~y+1);// x-y
    int ans2=  (((c>>31)&1)&(!signjudge)) | (!c) ; //同号并且x<=y,特判哦;
    return ans1 | ans2;
}
```

task-9

要求： implement the ! operator, using all of the legal operators except ! .

若$x=0, !x=1$, 反之 $!x=0$

问题转化为了我们如何判断一个数是否为0?

我们考虑去看x 和 $-x$ 的符号位是否相同即可

```c
int logicalNeg(int x) {
  // int sign1 = (x>>31)&1;
  // int sign2 =  ((~x+1)>>31) &1;
  // int ans = sign1 ^ sign2; //x为0, ans=0, x不为0, ans=1
  // return ~ans+2;

  return ~((((~x+1)|x)>>31) &1)+2;
}
```

task-10

本任务要求我们找出表示一个数的补码最少需要多少为bit。

如果x为一个正数，我们需要找到最高位的1，然后再加上一位符号位，同理

如果x为一个负数，我们需要找到最高位的0，然后再加上一位符号位。

不妨考虑x为正数，我们先去 它的高16位有没有1，即是否不为0.

如果有1，我们就要往最终答案中添加16. 如果无1，我们就添加0， 用bit_16来代表这个权重。

```c
int bit_16 = (!!(x>>16))<<4 //如果x>>16 不为0， 那么!!(x>>16)=1, 1<<4=16, 太巧妙了！！！
```

下一步我们需要更新当前的x, 令$x = x >> bit_16$ 即可。

接下来继续考虑x的高8位，高4位，高2位，高1位， 流程都是一样的。

```c
int howManyBits(int x) {
    int flag = x >> 31;
    x = ((~flag) & x) | (flag & (~x)); //如果x是负数，那么考虑~x.
    int bit_16 = (!!(x >> 16)) << 4;
    x = x >> bit_16;
    int bit_8 = !!(x>>8)<<3;
    x = x >> bit_8;
    int bit_4 = !!(x >> 4) << 2;
    x = x >> bit_4;
    int bit_2 = !!(x >> 2) << 1;
    x = x >> bit_2;
    int bit_1 = !!(x >> 1);
    x = x >> bit_1;
    int bit_0 = x;
    return bit_16+bit_8+bit_4+bit_2+bit_1+bit_0+1;
}
```

task-11

本任务要求：Return bit-level equivalent of expression 2*f for   floating point argument f.

我们先来回顾以下float是怎么存储的。

- **case 1: Normalized Values**

  三个field： sign s; exp ; frac

  其中 s 编号为31， expt编号为[23:30]， frac编号为[0：22]

  exp $\neq$ 0, $\neq 255$，其中E = e- bias（127）

- **case 2: Denormalized Values**

  exp为0，此时E = 1-bias

- **case 3: Special Values**

  exp全为1

综上: 当exp 为0， 直接对frac乘2即可

当exp $\neq$ 0, $\neq 255$ 时，直接对exp加1

```
unsigned floatScale2(unsigned uf) {
    unsigned s = (uf >> 31) & 0x1;
    unsigned exp = (uf >> 23) & 0xFF;
    unsigned frac = (uf & 0x7FFFFF); // 分别取出三个field
    //NaN
    if(exp == 0xFF) //exp全为1
        return uf;
    //
    else if(exp == 0){ //exp全为0
        frac <<= 1;
        return (s << 31) | (exp << 23) | frac;
    }
    //else
    exp++;
    return (s << 31) | (exp << 23) | frac;

}
```

task-12

本任务要求：给你一个float f，输出(int) f.

比如输入1.123，输出1； 输入1.5， 输出1， 输入0.12213， 输出0。

依然分情况讨论：

- **case 1: Normalized Values**

  此时$Value = (-1)^s 2^E M$, 其中$E = e - bias,\ M = 1 + frac$, 并且$1 \leq M < 2$

  和十进制的乘法类似，乘一个二无非是将小数点向左边移动一位。

  此时M的小数点后面有23位，

  - 若E>=23， 那么最后是一个整数，且需要再M后面填上$E - 23$个0
  - 若E<23 那么最后是一个小数，并且后$23 - E$ 个bits需要略去
  - 当E>=31时， 发生overflow

- **case 2: Denormalized Values**
  - 此时E=1-bias = -126， 显然输出0
- **case 3: Special Values**

  exp=255, E=255-127

```c
int floatFloat2Int(unsigned uf) {
    unsigned s = (uf >> 31) & 1;
    unsigned exp = (uf >> 23) & 0xFF;
    unsigned frac = (uf & 0x7FFFFF);
    int E = exp - 127;
    frac = frac | (1 << 23);
    if(E < 0) return 0;
    else if(E >= 31) return 0x1 << 31;
    else{
        if(E<23) {
            frac>>=(23 - E);
        }else{
            frac <<= (E - 23);
        }
    }
    if (s)
        return ~frac + 1;
    return frac;
}
```

task-13

本人任务要求我们输出： 2.0^x

x 大到什么程度会超出我们能表示的数的边界?

我们可以计算得到:

规格化float的范围： $[2^{-126}, 2^{127} \times (2 - 2^{-23})]$

非规格化float的范围: $[2^{-149}, 2^{-126} \times (1 - 2^{-23})]$

因此我们可以做出以下判断:

- x>127, 越界
- x< -149 越界
- -126<=x<=127, 规格化， frac全为0， exp = 127 + x
- -149<=x<-126, 非规格化, 此时E固定为-126， exp为0， 并且 M = frac， 此时有 $M * 2^{-126} = 2^x \rightarrow M = 2^{126+x}$

  由于M是2的幂次，因此frac只能有一个bit为1， 对于frac的23个bits， 其权重从左到右边分别为 $-1, -2, -3, \cdots, -23,$

  那么从右边开始，我们需要将哪个唯一的1左移多少位呢？设为n位

  那么，$-(23-n) = x + 126$, 得到 $n = x + 149$ 位

```c
unsigned floatPower2(int x) {
    if(x < -149)
        return 0;
    else if(x < -126)
        return 1 << (x + 149);
    else if(x <= 127)
        return (x + 127) << 23;
    else
        return (0xFF) << 23;
}
```

总结

真的好难，难以想象在计算机发展的早期，计算机科学家们花了多大的力气，并且只能用一些最朴素原始的计算方式， 去解决这些最底层的问题。