

## 实验要求

### 测试相关命令

#### 题目

实现异或  
输出补码形式的最小值  
检查Tmax  
所有奇数位为1  
取相反数  
判断是否为0到9的数字  
三目运算符  
小于等于  
实现 `!` 操作  
补码表示所需要的最小位数  
将浮点数乘以2  
将float转化为int  
返回 $2^x$

#### 总结

# 实验要求

- 第二章：信息的表示和处理

本章我们研究在计算机上如何表示数字和其他形式数据的基本属性，以及计算机对这些数据执行操作的属性

- 实验要求

`data1ab` 着重于让学生理解**数字在位级**上的表示与操作，通过限制学生的操作集（如仅能使用 `~`, `|`, `+` 等操作），让学生在**位级**上实现基础的运算操作

# 测试相关命令

只需修改bits.c

可以使用括号

```
# 检查编译错误
./dlc bits.c

# 检查操作符号的数量
./dlc -e bits.c

# 每次修改bits.c都要执行一次
make btest

# 测试
./btest [optional cmd line args]

# 输出全部信息
./btest

# 省略错误信息
./btest -g
```

```
# 测试某个函数
./btest -f foo
# 测试某个函数的某个用例
./btest -f foo -1 27 -2 0xf

# 检查数字格式
./ishow 0x27
    Hex = 0x00000027,    Signed = 39,    Unsigned = 39
./ishow 27
    Hex = 0x0000001b,    Signed = 27,    Unsigned = 27
./fshow 0x15213243
    Floating point value 3.255334057e-26
    Bit Representation 0x15213243, sign = 0, exponent = 0x2a, fraction =
0x213243
    Normalized. +1.2593463659 x 2^(-85)
./fshow 15213243
    Floating point value 2.131829405e-38
    Bit Representation 0x00e822bb, sign = 0, exponent = 0x01, fraction =
0x6822bb
    Normalized. +1.8135598898 x 2^(-126)
```

## 题目

### 实现异或

$a \oplus b$

1.  $(a|b) \& (\sim a | \sim b)$
2.  $\sim(\sim a \& \sim b) \& \sim(a \& b)$
3.  $(a \& \sim b) | (\sim a \& b)$

```
/*
 * bitXor - x^y using only ~ and &
 * Example: bitXor(4, 5) = 1
 * Legal ops: ~ &
 * Max ops: 14
 * Rating: 1
 */
int bitXor(int x, int y) {
    int a = ~x & ~y;
    int b = x & y;
    int res = ~a & ~b;
    return res;
}
```

### 输出补码形式的最小值

将1左移31位，得到0x80000000，即只有首位为1

```

/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
    int res= 1<<31 ;
    return res;
}

```

## 检查Tmax

Tmax = 0x7fffffff

令Tmax+1得到Tmin=0x80000000，再将Tmin与自身相加得到0

而当输入的x=-1=0xffffffff时，也符合上述性质，因此要排除这种情况

其他情况则**zero**不会等于0

```

/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *   and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 1
 */
int isTmax(int x) { //x=Tmax=0x7fffffff
    int Tmin = x+1; //Tmin=0x80000000
    int zero = Tmin + Tmin; //zero=0
    int flag=!Tmin; //flag=0
    int res=zero + flag; //res=0+0=0
    return !res ; //返回1
}
/*
int isTmax(int x) { //x=-1=0xffffffff
    int Tmin = x+1; //Tmin=0
    int zero = Tmin + Tmin; //zero=0
    int flag=!Tmin; //flag=1
    int res=zero + flag; //res=0+1=1
    return !res ; //返回0
}
*/

```

## 所有奇数位为1

最低位为0，最高位为31

设置掩码mask，作用是得到所有奇数位的值，而将其他位置为0（**奇数位不变，0还是0,1还是1**）

再将结果与mask取**异或**，若完全相同则为0，即所以奇数位都为1，其他位不管

**注意：**

1. 声明的变量值不能超过255
2. 变量要先声明再使用

```
/*
 * 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(0xFFFFFFFF) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
int allOddBits(int x) {
    int mask=0xAA; //变量值的大小有限制
    int allodd; //变量要先声明再使用
    int res;
    mask=(mask<<8) | mask; //得到0xAAAA
    mask=(mask<<16) | mask; //得到0xAAAAAAAA
    allodd=x & mask;
    res=allodd ^ mask;
    return !res;
}
```

## 取相反数

取反加1 (0也符合这条性质)

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

## 判断是否为0到9的数字

0 的后8位是 0011 0000 , 9 的后8位是 0011 1001

所以0到9的数字符合 0011 0xxx 或 0011 100x

第一种情况, 将x右移3位, 利用掩码进行异或操作观察是否后5位为00110

第二种情况, 将x右移1位, 利用掩码进行异或操作观察是否后5位为11100

**异或操作: 两个数只有完全相同时结果才为0**

```
/*
 * 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
*/
int isAsciiDigit(int x) {
    int mask1=0x6; //00110
    int shift1=x>>3;
    int flag1=mask1 ^ shift1; //为0即符合
    int mask2=0x1c; //11100
    int shift2=x>>1;
    int flag2=mask2 ^ shift2; //为0即符合
    return !flag1 | !flag2; //任何一种情况为0即返回1
}

```

## 三目运算符

当x为真时令其全为1，当x为假时令其全为0

因此只有两种情况：Y=y, Z=0 和 Y=0, Z=z

```

/*
*   conditional - same as x ? y : z
*   Example: conditional(2,4,5) = 4
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 16
*   Rating: 3
*/
int conditional(int x, int y, int z) {
    x=!!x; //x只能为1或0
    x=~x+1; //取相反数，x只能为-1或0，即全1或全0
    // int Y=x&y; //x全1时Y=y, x全0时Y=0
    // int Z=~x&z; //x全1时Z=0, x全0时Z=z
    return (x&y) | (~x&z); //两者取或
}

```

## 小于等于

判断x<=y是否成立，成立则返回1，否则返回0

直接想法：判断x-y<=0，但当两个数异号时可能会发生溢出

因此先判断是否为异号，如果异号则x<0且y>0就成立

如果同号的话就利用x-y<=0去判断，此时不会发生溢出

>> 算数右移：首位是0就补0，首位是1就补1

>>> 逻辑右移：全补为0

y的相反数-y为~y+1

$x - y = x + (\sim y + 1) \leq 0$ , 即  $x + \sim y \leq -1 < 0$ , 结果必为负值

```
/*
 * isLessOrEqual - if x <= y then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
int isLessOrEqual(int x, int y) {
    int signX=x>>31 & 1; //获取x的符号
    int signY=y>>31 & 1; //获取y的符号
    int signSame=signX^signY; //x与y同号, 则signSame为0
    int signDiff=signX & !signY; //x为负且y为正, 则signDiff为1
    int negateY=~y; //取y的相反数-1
    int minus=x+negateY; //x-y<=0
    int res=(minus>>31) & 1; //移位得到x-y的符号
    return signDiff | ((!signSame)&res); //两种情况: 异号 或 同号且x-y<=0
}
```

## 实现！操作

思考0与其他数的区别：0的相反数是它本身

注意：Tmin的相反数也是它本身

将x和x的相反数取异或操作，只有两者相同时结果为0，结果的符号位为0，其他的符号位为1

判断符号位，右移31位，`0000+1=1`，`1111+1=0`（溢出）

因为不能使用取反操作！，所以要特殊处理

最后再排除Tmin的情况

```
/*
 * 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
 */
int logicalNeg(int x) {
    int negate=~x+1; //取相反数
    int res1=((x^negate)>>31) + 1; //判断异或结果的符号位, 符号为0得到1, 符号为1得到0
    int res2=(x>>31) + 1; //当x为0时, 符号为0得到1; 当x为Tmin时, 符号为1得到0
    return res1 & res2;
}
```

## 补码表示所需要的最小位数

正数：找到最高位为1的位n，n+1（要加一个0）

负数：找到最高位为0的位n，n+1（要加一个1）

```

/* 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
 */
int howManyBits(int x) {
    int b16,b8,b4,b2,b1,b0;
    int sign=x>>31;
    x = (sign&~x)|(~sign&x); //如果x为正则不变，否则按位取反（这样都统一为找最高位为1的位）

    // 不断缩小范围
    b16 = !(x>>16)<<4; //高十六位是否有1
    x = x>>b16; //如果有（至少需要16位），则将原数右移16位
    b8 = !(x>>8)<<3; //剩余16位的高8位是否有1
    x = x>>b8; //如果有（至少需要16+8=24位），则右移8位
    b4 = !(x>>4)<<2; //剩余8位的高4位是否有1
    x = x>>b4;
    b2 = !(x>>2)<<1; //剩余4位的高2位是否有1
    x = x>>b2;
    b1 = !(x>>1); //剩余2位的高1位是否有1
    x = x>>b1;
    b0 = x;
    return b16+b8+b4+b2+b1+b0+1; //+1表示加上符号位
}

```

## 将浮点数乘以2

不同情况：特殊值（无穷大或NaN），非规格数，规格数

- 非规格数
  - 除去符号位，将剩余位全部左移一位
- 规格数
  - 乘以2之后溢出（阶码位加1后全为1），返回无穷大
  - 正常则只需将阶码位加1

```

/*
 * 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
 */

```

```

unsigned floatScale2(unsigned uf) {
    int exp = uf&0x7F800000; //阶码位
    int sign = uf&0x80000000; //符号位
    int frac = uf&0x7FFFFFFF; //尾数位
    if((exp>>23)==0xFF){return uf;} //无穷大或NaN, 阶码位全1
    else if((exp>>23)==0) //非规格数, 阶码位全0
    {
        return sign | ((uf&0x7FFFFFFF)<<1);
    }
    else //规格数
    {
        int tmp=(exp>>23)+1; //乘以2相当于阶码位加1
        if(tmp==0xFF) //如果加1后阶码位全1, 相当于溢出, 返回无穷大
        {
            return sign | (tmp<<23);
        }
        else //未溢出则正常
        {
            return sign | (tmp<<23) | frac;
        }
    }
}

```

## 将float转化为int

先将sign, exp和frac按位表示出来

exp全为1表示特殊值, exp全为0表示非规格值

**frac记得要加1**, 即此时frac变为0000 0000 1xxx xxxx xxxx xxxx xxxx

将exp减去bias即可得到真正的指数E, E的范围是[-128, 127]

int类型只有32位, 因此当E**大于等于**31时, 小数点右移后会超出int的表示范围

而当E小于0时, 会向0舍入

- 当 $0 \leq E < 31$ 时
  - $0 \leq E \leq 23$ , 小数点右移E位后, 小数部分舍去, 相当于frac部分右移了(23-E)位
    - frac=1.11001, 小数点右移3位得到1110.01, 小数部分舍去得到1110, 相当于111001右移了(5-3)=2位, 得到1110
  - $23 < E < 31$ , 小数点右移E位后, frac后面会多出(E-23)位0, 相当于frac部分左移了(23-E)位
    - frac=1.11001, 小数点右移7位得到11100100, 后面会多出(7-5)=2位0, 相当于111001左移了(7-5)=2位, 得到11100100

### 最后判断符号

首先转化完的frac的表示形式(32位)是**无符号整数**, 而转化为int类型(32位)时变为**有符号整数**

四种情况:

- sign=0, frac首位为0
  - 首位是0的无符号数, 转化为有符号数, 形式不变, 而原本就是正数, 返回frac
- sign=1, frac首位为0



- 首位是0的无符号数，转化为有符号数，形式不变，而原本是负数，返回相反数 $\sim \text{frac} + 1$
- sign=0, frac首位为1 (E=31)
  - 首位是1的无符号数，转化为有符号数，发生上溢出，无法表示，返回0x80000000u (与原本的正负无关)
- sign=1, frac首位为1 (E=31)
  - 首位是1的无符号数，转化为有符号数，发生上溢出，无法表示，返回0x80000000u (与原本的正负无关)

特例: 0xc0000000

sign=1, exp=158, frac=0x00800000

E=158-127=31

将frac左移(31-23)=8位，得到frac=0x80000000

此时frac是一个无符号数，它向有符号数转化时发生了**向上溢出**，与sign的符号无关

```
/*
 * 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
 */
int floatFloat2Int(unsigned uf) {
    int sign = uf & 0x80000000; //原始符号
    int exp = uf & 0x7F800000;
    int frac = (uf & 0x007FFFFF) | 0x00800000; //小数部分记得加上1，即frac是大于等于1的
    //数(前8位是0，第九位是1)
    int E; //记录真正的指数E=exp-bias

    if((exp^0x7F800000)==0) return 0x80000000u; //特殊值（无穷大或NaN）
    if(exp==0) return 0; //非规格数（接近0）

    E = (exp>>23)-127;
    if(E>=31) return 0x80000000u; //向上溢出，注意当E=31时，frac向左移动8位，此时frac首位
    //必为1发生溢出，其他情况首位必为0
    else if(E<0) return 0; //向0舍入
    else if(E>23) frac<<=(E-23); //当24<=E<=30时，frac只能左移1到7位，首位一定还是0
    else frac>>=(23-E); //当0<=E<=23时，frac只能右移0到23位，首位一定还是0

    if(sign==0) return frac; //sign与无符号数frac的首位都为0，未发生溢出，返回原值
    else return ~frac+1; //sign=1且无符号数frac的首位为0，未发生溢出，返回相反数
}
```

## 返回 $2^x$

四种情况

- 情况一，**都是0**：E=1-127=-126； -126-24=-**150**
- $2^{-150}$ 的二进制表示为0|00000000|000000000000000000000000
- 情况二，**阶码0**：E=1-127=-126； -126-23=-**149**，-126-1=-**127**

0.xxxx 小数部分有且只有一位为1，其余位为0，相当于小数点位置固定了，1的位置不断变化

- $2^{-149}$ 的二进制表示为0|00000000|000000000000000000000001
- $2^{-148}$ 的二进制表示为0|00000000|000000000000000000000010
- $2^{-147}$ 的二进制表示为0|00000000|0000000000000000000000100
- .....
- $2^{-129}$ 的二进制表示为0|00000000|001000000000000000000000
- $2^{-128}$ 的二进制表示为0|00000000|010000000000000000000000
- $2^{-127}$ 的二进制表示为0|00000000|100000000000000000000000

- 情况三，**阶码非0**：E=exp-127； 1-127=-**126**，254-127=**127**

1.0000 小数部分全为0，只有整数部分加上一个1，相当于小数点位置不断变化

- $2^{-126}$ 的二进制表示为0|00000001|000000000000000000000000
- $2^{-125}$ 的二进制表示为0|00000010|000000000000000000000000
- $2^{-124}$ 的二进制表示为0|00000011|000000000000000000000000
- .....
- $2^0$ 的二进制表示为0|01111111|000000000000000000000000
- $2^1$ 的二进制表示为0|10000000|000000000000000000000000
- .....
- $2^{127}$ 的二进制表示为0|11111110|000000000000000000000000

- 情况四，**无穷大**：E=exp-127； 255-127=**128**

- $2^{128}$ 的二进制表示为0|11111111|000000000000000000000000
- $2^{129}$ 的二进制表示为0|11111111|000000000000000000000000

```
/*
 * 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
 */
unsigned floatPower2(int x) {
    int exp;
    if(x<=-150) return 0;
    else if(-149<=x && x<=-127)
    {
        int pos=-126-x; //23到1
        int bias=23-pos; //0到22
        return 1<<bias;
    }
    else if(-126<=x && x<=127)
    {
        exp=x+127; //1到254
        return exp<<23;
    }
}
```

```
else return 0xFF<<23;  
}
```

```
csapp@ubuntu:~/Desktop/Lab1/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  
csapp@ubuntu:~/Desktop/Lab1/datalab-handout$
```

## 总结

对于整数和浮点数的表示方式有了深刻的认识，之前很少使用位操作符号，现在才知道有这么大的用处。

尤其是 `floatFloat2Int` 将浮点数转化为整数，让我感叹浮点数设计之巧妙，指数要减去bias得到阶码、小数部分用无符号数表示，都是为了让非规格数平滑过渡到规格数，同时这道题还体现了无符号数向有符号数转化的溢出现象，实在是太妙了。

最后 `howManyBits` 这道题感觉难度很大，可以理解，但很难自己对照思路写出来。

整数的表示方式，位操作符号