DataLab实验报告

> 软件2203班——丁海桐——202226010304

一、实验项目

在给定规则限制下完成bits.c中的函数。其中最主要的规则如下：

整数规则

- 不能使用for while if等
- 只能使用! ~ & ^ | + << >>运算符
- 只能使用int
- 只能使用0-0xFF的常数
- 使用运算符数不超过限制(Max ops)
- 不能使用全局变量或调用函数等其他规则

浮点数规则

- 可以使用for while if
- 只能使用int，unsigned int
- 使用运算符数不超过限制(Max ops)
- 不能使用数组，函数调用等其他规则

完成bits.c后使用 `./dlc` 检查代码是否符合规范， `make btest` 进行编译， `./btest` 进行函数测试

二、实验内容

1. bitAnd

- ***题目要求***：使用按位或和按位取反实现按位与。
- ***思路***：使用德摩根定律将与操作转换为按位取反。

```
/*
 * bitAnd - x&y using only ~ and |
 *   Example: bitAnd(6, 5) = 4
 *   Legal ops: ~ |
 *   Max ops: 8
 *   Rating: 1
 */
int bitAnd(int x, int y) {
        return ~(~x | ~y);
}
```

2. getByte

- ***题目要求***：从低位起字节编号为0-3，取出编号为n的字节。
- ***思路***：将需要的字节右移n*8位到最低位，再和0xFF与清除其他位。

```
/*
 * getByte - Extract byte n from word x
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: getByte(0x12345678,1) = 0x56
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int getByte(int x, int n) {
        //num = 8 * n
        int num = n << 3;
        return x >> num & 0xff;

}
```

3. logicShift

- ***题目要求***：实现逻辑右移。
- ***思路***：和00..0011..11(n个0)相与，清除前n位。要得到00..0011..11(n个0)，就要先得到11..1100..00(n个1)，再取反。

```
/*
 * logicalShift - shift x to the right by n, using a logical shift
 *   Can assume that 0 <= n <= 31
 *   Examples: logicalShift(0x87654321,4) = 0x08765432
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int logicalShift(int x, int n) {

        // a = 11..10000..0, 共有n个1
        int a = 1 << 31 >> n << 1;
        // ~a = 00..011111, 共有n个0
        return x >> n & ~a;
}
```

4. bitCount

- ***题目要求***：数出二进制数中1的个数
- ***思路***：
    1. 采用分治的策略
    2. 将所有位分成32组，一组中只有1位；
    3. 将相邻两组合为一组，组中的数值为原来两组中的数值相加；
    4. 重复第2步，直到合成只有1组，组中的数值即为结果。

5. 开始中每组中的数值即为每组中1的数量，然后将相邻两组中的数值相加的过程就相当于将之前一级的1的数量汇总，不断重复这个过程就可以将1的数量汇总到最后的一个数中。

```
int bitCount(int x) {
        int m1, m2, m3, m4, m5;
        m1 = 0x55 + (0x55 << 8); //m1 = 01010101...
        m1 = m1 + (m1 << 16);
        m2 = 0x33 + (0x33 << 8);
        m2 = m2 + (m2 << 16);    //m2 = 00110011...
        m3 = 0x0f + (0x0f << 8);
        m3 = m3 + (m3 << 16);    //m3 = 0x0f0f0f0f
        m4 = 0xff ;              //m4 = 0x00ff00ff
        m4 = m4 + (m4 << 16);
        m5 = 0xff + (0xff << 8); //m5 = 0x0000ffff

        //前面的 + 后面的
        x = (x & m1) + ((x >> 1) & m1); //2bits 为1组
        x = (x & m2) + ((x >> 2) & m2); //4bits
        x = (x & m3) + ((x >> 4) & m3); //8
        x = (x & m4) + ((x >> 8) & m4); //16
        x = (x & m5) + ((x >> 16) & m5);//32
        return x;
}
```

5. bang

- ***题目要求***：实现取非操作。。
- ***思路***：等价于求二进制数中是否含有1。高16位与低16位相或，低8位与8-16位相或，低4位与4-8位相或，低2位与2-4位相或，最低位和第二位相或，最终得到的结果是32位相或的结果，也就是是否含1，即最终结果。这个思路与bitCount的分治思路类似。

```
/*
 * bang - Compute !x without using !
 *   Examples: bang(3) = 0, bang(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
int bang(int x) {
        //这里不用像上一题那样计算数量，只用用`|`来保证有就行了
        int or16, or8, or4, or2, or1;
        or16 = x | x >> 16;
        or8 = or16 | or16 >> 8;
        or4 = or8 | or8 >> 4;
        or2 = or4 | or4 >> 2;
        or1 = or2 | or2 >> 1;
        //`& 0x01`是为了清除31-1位上的字符，`^ 0x01`采用异或取反
        return (or1 & 0x01) ^ 0x01;
}
```

6. tMin

- ***题目要求***：最小的补码。
- ***思路***：0x80000000。

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

7. fitsBits

- ***题目要求***：求x是否可用n位补码表示。
- ***思路***：对于n位补码表示的数，这里以负数为例：其分为符号位和数据位两部分，符号位是从左起连续的1，遇到第1个0停止不包含0；符号位就是剩下的位。若是x能用n位补码表示，那么右移n-1位，剩下的肯定全是符号。也就是正数右移n-1位得到0x0，负数右移n-1位得到0xffffffff。

```
/*
 * fitsBits - return 1 if x can be represented as an
 *  n-bit, two's complement integer.
 *   1 <= n <= 32
 *   Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int fitsBits(int x, int n) {
        //00...000[1011]
        //<------>所有的0都可以看成符号位
        //11...111[0111]
        //<------>所有的1都可以看成符号位
        int sign = x >> 31 & 1; //sign,  1或者0
        int off = n + ~1 + 1;   //data, 对于n位补码，用n-1个字符保存数据
                                //这里off表示偏移量，也就是数据的最大长度，值为n - 1
        //正数的情况，移位后正确应该是000..000，错误是000..0001..
        int pos = !sign & !(x >> off);
        //负数的情况，移位后正确应该是111..111，错误是111..1110..
        int neg = sign & !~(x >> off);
        return pos | neg;

}
```

8. divPwr2

- ***题目要求***：计算x除2的n次方。
- ***思路***：对于正数可以直接右移，对于负数需要增加偏置量off使其向0进位。负数中若是右移n位出去的都是0，那说明x能被2^n整除，off=0；负数中若是右移n位出去的存在1，就应该在移位后+1，或者在移位前+2^n-1。

```
/*
 * divpwr2 - Compute x/(2^n), for 0 <= n <= 30
 *  Round toward zero
 *   Examples: divpwr2(15,1) = 7, divpwr2(-33,4) = -2
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int divpwr2(int x, int n) {
        /*
          除法要向0取整，使用移位操作是向下取整，正数无所谓，负数就要修正
          这里负数也只是部分要修正，下面我们以4位补码举例：
          1010 = -6   -6
          0101 = -3  -6/2 = -3，此时并不需要修正，
          只要右移移出去的是0，也就是这个数还是2的倍数，直接右移就不需要修正
          0010 = -2  -6/4 = -1，此时需要修正，右移移出去1
          这里修正需要“+1”，【但是】这个“+1”是在【除完之后的结果】上“+1”，
          所以在【原数】上 + 【2^n - 1】，这样只要后n位不全是0，有1，在加上【2^n - 1】        后再移位会自动“+1”
         */

        //00..011..11 = (1 << n) + ~1 + 1)
        //     <---->n个1
        // 后面的(x >> 31)是确保当x为负数时，off才会有值
        int off = ((1 << n) + ~1 + 1) & (x >> 31);
        return (off + x)>> n;
}
```

9. negate

- ***题目要求***：求-x。
- ***思路***：取反+1即可。

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

10. isPositive

- ***题目要求***：大于0返回1。
- ***思路***：通过符号位判断是否>=0, 在通过~0 + 1 = 0排除0。

```
/*
 * isPositive - return 1 if x > 0, return 0 otherwise
 *   Example: isPositive(-1) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 3
 */
 int isPositive(int x) {
        /*
左边是为了排除0，~0 + 1 = 0，这里还要 >> 31，否则~x+1结果为0xfffffff0时return 0
右边是通过符号位，符号位是0时!为 0x00000001
         */
        //(!(x >> 31)) --> 0x80000000, no ~ -->0xffffffff
        return ((~x + 1) >> 31) & (!(x >> 31));
}
```

11. isLessOrEqual

- ***题目要求***：实现<=判断。
- ***思路***：
  - 【x - y +】 ->yes ->(sx & !sy)
  - 【x + y -】 ->no
  - 【x + y +】->yes -> y - x >= 0
  - 【x - y -】->yes -> y - x >= 0

```
/*
 * 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 sx, sy, res;
        sx = !!(x >> 31);
        sy = !!(y >> 31);
        res = y + ~x + 1; //res = y - x >= 0
                          // sx和sy同或 //借助上一题，这里不需要排除0
        return (sx & !sy) | (!(sx ^ sy)  & (!(res >> 31)));
}
```

12. ilog2

- ***题目要求***：求以2为底的对数，向下取整。
- ***思路***：求最靠前的1在第几位，可以通过把x转换成00...0011..11，再统计x中1的个数。

```
int ilog2(int x) {
        int m1, m2, m3, m4, m5;
        //起初x = 00..0011100110110, 第一个1在第n位
        x = x | x >> 1; //n、n-1 = 1
        x = x | x >> 2; //n、n-1、n-2、n-3 = 1
        x = x | x >> 4; //n、n-1、n-2、n-3、n-4、n-5、n-6、n-7
        x = x | x >> 8; //...
        x = x | x >> 16;//...
        //x = 00..00111111..111
        //下面数一下有多少个1


        m1 = 0x55 + (0x55 << 8); //m1 = 01010101...
        m1 = m1 + (m1 << 16);
        m2 = 0x33 + (0x33 << 8);
        m2 = m2 + (m2 << 16);    //m2 = 00110011...
        m3 = 0x0f + (0x0f << 8);
        m3 = m3 + (m3 << 16);    //m3 = 0x0f0f0f0f
        m4 = 0xff ;              //m4 = 0x00ff00ff
        m4 = m4 + (m4 << 16);
        m5 = 0xff + (0xff << 8); //m5 = 0x0000ffff

        x = (x & m1) + ((x >> 1) & m1); //2bits
        x = (x & m2) + ((x >> 2) & m2); //4bits
        x = (x & m3) + ((x >> 4) & m3); //8
        x = (x & m4) + ((x >> 8) & m4); //16
        x = (x & m5) + ((x >> 16) & m5);//32

        //x = 00...001 时，logx = 0，所以还要-1
        return x + ~1 + 1;

}
```

13. float_neg

- ***题目要求***： 浮点数取-x。
- ***思路***：将需要的字节右移到最低位，再和0xFF与清除其他位。右移的位数为n*8。
  - 首先判断uf是否是NAN，通过阶码 == ff和尾数 != 0
  - 剩下的符号位取反就行

```
/*
 * float_neg - Return bit-level equivalent of expression -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 representations of
 *   single-precision floating point values.
```

```
 *   When argument is NaN, return argument.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 10
 *   Rating: 2
 */
unsigned float_neg(unsigned uf) {
        if(((uf & 0x7fffffff) >> 23) == 0xff && uf << 9) return uf;
        return uf ^ 0x80000000;
}
```

14. float_i2f

- ***题目要求***：将整型转换为浮点数表示。
- ***思路***：
    1. 两种特殊情况提前判断，这里由于后面要取绝对值全部变成正数，但是0x80000000的绝对值用in没法表示，所以提前判断。
    2. 求符号位
    3. 将x全部转换成正数
    4. 确定exp
    5. 确定frac
    6. 舍入

```
/*
 * float_i2f - Return bit-level equivalent of expression (float) x
 *   Result is returned as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point values.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_i2f(int x) {
        int s;
        int exp;
        int frac;
        int ans;
        int front_zero_cnt;
        int low_9;

        //0、两种特殊情况提前判断，这里由于后面要取绝对值全部变成正数，但是0x80000000的绝对值用in没法表示，所以提前判断
        if(x == 0) return 0;
        if(x == 0x80000000) return 0xcf000000; // 1 10011110 ... - 127+31

        //1、符号位
        s = x & 0x80000000;

        //2、变成正数
        if(x < 0) x = -x;
```

```
        //3、统计从31位到第一个1有多少个0，以此来确定exp
        front_zero_cnt = 0;
        while(!(x & 0x080000000))
        {
                front_zero_cnt++;
                x = x << 1;
        }

        //4、确定exp，-1是减去尾数省略的那个[1.]
        exp = (127 + 32 - front_zero_cnt - 1) << 23;
        x = x <<  1 ; //front_zero_cnt已经去掉了，现在只用去掉尾数省略的那个[1.]

        //5、确定frac
        frac = x >> 9 & 0x007fffff;
        ans = s + exp + frac;

        //6、舍入
        low_9 = x & 0x000001ff;  //low_9是没有进入frac的尾数，用其来判断舍入
        if(low_9 > 0x00000100) ans++;
        if((low_9 == 0x00000100) && (ans & 0x1) ) ans++;

        return ans;
}
```

15. float_twice

- ***题目要求***：返回2f。
- ***思路***：对于规格化数阶码+1即可，非规格化数由于可以和规格化数平滑衔接，只需要左移1位，补充符号位，特殊值直接返回原值。

```
/*
 * float_twice - 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 float_twice(unsigned uf) {
        int s;
        int exp;
        s = uf & 0x80000000;
        exp = uf & 0x7f800000;
        // +0，-0，无穷，NAN直接判断
        if(uf == 0 ||uf == 0x80000000 || exp == 0x7f800000) return uf;
        if(exp == 0) //非规格化数 ，从尾数左移，把移出来的1放到阶码上
```

```
        {
                uf = uf << 1;
                uf = uf + s;
        }
        //规格化数 // 阶码 + 1
        else uf += 0x00800000;

        return uf;
}
```

结果测试

```
dinghaitong@ubuntu:~$ cd dataLab/
dinghaitong@ubuntu:~/dataLab$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitAnd
 2      2       0       getByte
 3      3       0       logicalShift
 4      4       0       bitCount
 4      4       0       bang
 1      1       0       tmin
 2      2       0       fitsBits
 2      2       0       divpwr2
 2      2       0       negate
 3      3       0       isPositive
 3      3       0       isLessOrEqual
 4      4       0       ilog2
 2      2       0       float_neg
 4      4       0       float_i2f
 4      4       0       float_twice
Total points: 41/41
```

三、实验总结

3.1 实验中出现的问题

1. 对于各个操作符的优先级不清楚。
2. 有些题目过于困难，苦思冥想之后仍不知所以。在处理问题时也参考了许多网上的解法，用更易理解的方式重写以及补充必要的注释，以便复习。而在参考解法时也发现了一些题的多种解法，这体现了位处理的灵活性。
3. 总是忽略0、0x80000000这类特殊的数导致出错。

3.2 心得体会

1. 熟练掌握了位的各种操作。
2. 对于有符号整数的补码形式有了更深刻的理解。
3. 掌握了分治法在数字上的应用。

4. 对于0、0x80000000这类特殊的数的性质更加了解。
5. 掌握了整数转换成浮点数的全过程。