《计算机系统》

datalab 实验报告

班级：信安 2101 班

学号：202109070105

姓名：孙照海

目录

1 实验项目

1.1 项目名称

DataLab 实验

1.2 实验目的

1) 本实验是《深入理解计算机系统》一书中的附带实验。在本次实验中，学生实现简单的逻辑，二进制补码和浮点函数，但使用 C 的高度受限的子集。例如，可能会要求他们仅使用位级操作和直线代码来计算数字的绝对值。

2) 本实验帮助学生了解 C 数据类型的位级表示和数据操作的位级行为。

1.3 实验资源

datalab-handout 文件夹和 bits.c 文件中的说明：

1. 使用 dlc（数据实验室检查器）编译器（在讲义中描述）检查解决方案的合法性。
   使用 DLC 检查合法性
2. 每个函数都有最大数量的运算符（！ ~ & ^ | + << >>）您可以使用它来实现该功能；dlc 检查最大操作数。请注意，'='不是计算;您可以根据需要使用尽可能多的这些而不会受到惩罚。
3. 使用 btest 测试线束检查功能是否正确。
4. 使用 BDD 检查程序正式验证您的功能
5. 每个函数的最大操作数在函数中给出每个函数的标题注释。如果在写入和此文件中的最大操作数之间有任何不一致之处，请考虑这个文件是权威来源。

2 实验任务

2.1 实验任务 1

任务名称：bitAnd 函数：

x&y using only ~ and 使用~和|实现位与操作

Example: bitAnd(6, 5) = 4

Legal ops: ~ |

Max ops: 8

Rating: 1

```
int bitAnd(int x, int y) {
    //使用摩根定律 a&b=~(~a|~b)
    return ~(~x|~y);
}
```

2.2 实验任务 2

任务名称：getByte 函数：

getByte - Extract byte n from word x 从整型 x 中取出第 n 个字节

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) {
    //第0个字节储存的是0x78...
    //n左移3位 扩大8倍
    //x右移3个字节 8bit
    //与&0xff相与，即可使前三个字节清零，而最后一个字节保持不变
    return (x>>(n<<3)) & 0xFF;
}
```

2.3 实验任务 3

任务名称：logicalShift 函数：

logicalShift - shift x to the right by n, using a logical shift　实现逻辑右移

Can assume that 0 <= n <= 31

Examples: logicalShift(0x87654321,4) = 0x08765432

Legal ops: ! ~ & ^ | + << >>

Max ops: 20

Rating: 3

```
int logicalShift(int x, int n) {
    /*
    1.C语言默认用的是算术右移，算术右移是符号位(0/1)补位，而题目要求的是逻辑右移，逻辑右移是0补位。
    故可以先进行算术右移，然后将补位的数字(0/1)全部替换成0;
    2.替换可以使用掩码(参考函数2)。使其前n位为0，后(32-n)位为1;
    3.掩码的构造方法。
        a. 将1左移31位：1<<31; 10000000… .
        b. 取反：~(1<<31) 0111111… .
        c. 右移n位(符号位是0,0补位)：(~(1<<31))>>n 0000011111… .
        d. 左移1位（注意：题目要求不能用减法，故不直接使用右移n-1位）：
        ((~(1<<31))>>n)<<1 \quad 00001111… .110
        e. 和1相或以补最右边的0：(((~(1<<31))>>n)<<1) 00001111… .111
    4.将右移n位后的x与掩码相与
    */
    int mask_code=(((~(1<<31))>>n)<<1)|1;
    return (x>>n)&mask_code;
}
```

2.4 实验任务 4

任务名称：bitCount 函数：

bitCount - returns count of number of 1's in word　求 x 中 1 的个数

Examples: bitCount(5) = 2, bitCount(7) = 3

Legal ops: ! ~ & ^ | + << >>

Max ops: 40

Rating: 4

```
int bitCount(int x) {
    /*
    可用分治法来做
    求32位二进制数里有多少个1，可以先2位2位的看，再4位4位的看，再8位8位的看，再16位16位的看，最后看32位。
    举例说明：x=10110100（8位）
    第一步：shift_2-1=(10|11|01|00)&01010101=00010100
    X右移1位：shift_2-2=(01|01|10|10)&01010101=01010000
    Sum_2=shift_2-1+shift_2-2=00010100+01010000=01|10|01|00
    第二步：shift_4-1=（0110|0100）&00110011=0010|0000
    X右移2位：shift_4-2=（0001|1001）&00110011=0001|0001
    Sum_2=shift_4-1+shift_4-2=00100000+00010001=0011|0001
    第三步：shift_8-1=（00110001）&00001111=00000001
    X右移4位：（00000011）&00001111=00000011
    Sum_2=shift_2-1+shift_2-2=00000001+00000011=00000100
    即含有4个1
    因为实验要求中整数常数的范围是：0-255（0x0-0xff）
    故需要对掩码进行转换，这可以通过移位和或运算实现
    */

    //构造掩码
    int m_1,m_2,m_4,m_8,m_16;
    m_1=0x55|(0x55<<8);//01010101=0x55
    m_1=m_1|(m_1<<16);//m_1=01010101 01010101 01010101 01010101
    m_2=0x33|(0x33<<8);//00110011=0x33
    m_2=m_2|(m_2<<16);//m_2=00110011 00110011 00110011 00110011
    m_4=0x0f|(0x0f<<8);//00001111=0x0f
    m_4=m_4|(m_4<<16);//m_4=00001111 00001111 00001111 00001111
    m_8=0xff|(0xff<<16);//11111111=0xff m_8=00000000 11111111 00000000 11111111
    m_16=0xff|(0xff<<8);//m_16=00000000 00000000 11111111 11111111

    x=(x&m_1)+((x>>1)&m_1);
    x=(x&m_2)+((x>>2)&m_2);
    //最多32个1，故下面三组的最高位不会是1
    x=(x&m_4)+((x>>4)&m_4);
    x=(x&m_8)+((x>>8)&m_8);
    x=(x&m_16)+((x>>16)&m_16);
    return x;
}
```

2.5 实验任务 5

任务名称：bang 函数：

bang - Compute !x without using ！不用！符号求得！x

Examples: bang(3) = 0, bang(0) = 1

Legal ops: ~ & ^ | + << >>

Max ops: 12

Rating: 4

```
int bang(int x) {
    //求反的过程实际就是判断x是否为0。x若为0，则!x为1，否则为0；
    //只有0的原码和补码的或的最高位为0，即转化为判断原补码或运算后的最高位是否为0的问题，而这可以用右移31位后&1来判断
    //~和！不同。~是按位求反；！是逻辑求反，其结果只有0或1
    int com=~x+1;//求补码
    return ~((com|x)>>31)&1; //只有0的原补码的或的最高位为0
}
```

2.6 实验任务 6

任务名称：tmin 函数：

tmin - return minimum two's complement integer 返回补码整数的最小整数数值

Legal ops: ! ~ & ^ | + << >>

Max ops: 4

Rating: 1

```
int tmin(void) {
    //最小的二进制补码是10000000 00000000 00000000 00000000（负数）
    return 1<<31;
}
```

2.7 实验任务 7

任务名称：fitsBits 函数：

fitsBits - return 1 if x can be represented as an n-bit, two's complement integer.如果 x 可以表示为 n 位二进制补码形式则返回 1

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) {
    /*
    1.题目要求判断x能否用n位二进制数表示，而x肯定能用32位二进制数表示，可通过将32位二进制数左移(32-n)位
    再右移(32-n)位后的数是否还和原来相等来判断(由于补位原则，相当于只保留了n位有效数)
    2.因为题目不允许用-号，故可以用+上n的反码再+1的方式求差
    3.通过异或^来判断是否相等
    4.算术右移：最高位填充符号位，正数填0，负数填1
    逻辑右移：都是填0
    左移都是填0
    */
    int count=32+(~n)+1;
    return !(x^(x<<count>>count));
}
```

2.8 实验任务 8

任务名称：divpwr2 函数：

divpwr2 - Compute x/(2^n), for 0 <= n <= 30　计算 x 除以 2 的 n 次方

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) {
    //正数：直接右移一位
    //负数：加上一个偏置位，避免向负取整的情况
    //如果x为正数，则直接右移n位，即偏移量为0；如果x为负数则需要加偏移量2*n-1之后再右移n位
    //C语言默认为算术右移
    int sign,bias;
    sign=x>>31;    //符号位
    bias=sign&((1<<n)+(~0));    //偏置位 正数时为0 其中~0为-1
    return (x+bias)>>n;
}
```

2.9 实验任务 9

任务名称：negate 函数：

negate - return -x　求相反数

Example: negate(1) = -1.

Legal ops: ! ~ & ^ | + << >>

Max ops: 5

Rating: 2

按位取反，末位加 1

```
int negate(int x) {
    //按位取反，末位加1
    return (~x)+1;
}
```

2.10 实验任务 10

任务名称：isPositive 函数：

isPositive - return 1 if x > 0, return 0 otherwise　如果 x 为正数则返回 1，否则返回 0

Example: isPositive(-1) = 0.

Legal ops: ! ~ & ^ | + << >>

Max ops: 8

Rating: 3

```
int isPositive(int x) {
    //1.正数的符号位为0，负数的符号位为1，0的符号位为0
    //2.判断正数：符号位为0
    //3.判断0：!0=1 ！1=0 ！-1=1
    //4.x|0=x,x|1=1;
    return !((x>>31)|(!x));
}
```

2.11 实验任务 11

任务名称：isLessOrEqual 函数：

isLessOrEqual - if x <= y　then return 1, else return 0　判断 x<=y 则返回 1，否则 0

Example: isLessOrEqual(4,5) = 1.

Legal ops: ! ~ & ^ | + << >>

Max ops: 24

Rating: 3

```
int isLessOrEqual(int x, int y) {
    //要判断x<=y通常我们都会想到用y-x>=0来判断，这时候就碰到了一个问题，如果x与y异号且x为负时会发生溢出
    //因此分情况讨论，如果x与y同号的话，看y-x的符号位（进行符号扩展右移31位）
    //如果x与y异号的话，之需要根据x来判断，若x的符号位为1则y>=x
    int signx=(x>>31) & 0x1;     //求得x的符号位
    int signy=(y>>31) & 0x1;     //求得y的符号位
    int is_notSame=(signx ^ signy); //判断x、y是否异号，进行异或操作，如果结果为1则异号，否则同号
    int tmp=((y+(~x)+1)>>31) &0x1;  //x与y同号时进行减法操作之后得到结果的符号位
    return ((!tmp) & (!is_notSame))|(is_notSame & signx));
}
```

2.12 实验任务 12

任务名称：ilog2 函数：

ilog2 - return floor(log base 2 of x), where x > 0　求以 2 为底 x 的对数

Example: ilog2(16) = 4

Legal ops: ! ~ & ^ | + << >>

Max ops: 90

Rating: 4

```
int ilog2(int x) {
    //要求以2为底，x的对数，只需要知道最高的1在哪一位即可
   //采用二分法，第一次右移16位，判断结果是否大于0，进行两次!! 变为0/1之后左移4位
   //第二次右移8+bitsNumber位，判断结果是否大于零，进行两次!! 变为0/1之后左移3位
   //依次分下去
   //!为算数运算符 返回结果为0或1
   int bitsNumber;
   bitsNumber = (!!(x>>16)) << 4;
   bitsNumber = bitsNumber + ((!!(x >> (bitsNumber + 8))) << 3);
   bitsNumber = bitsNumber + ((!!(x >> (bitsNumber + 4))) << 2);
   bitsNumber = bitsNumber + ((!!(x >> (bitsNumber + 2))) << 1);
   bitsNumber = bitsNumber + ( !!(x >> (bitsNumber + 1)));
   return bitsNumber;
}
```

2.13 实验任务 13

任务名称：float_neg 函数：

float_neg - Return bit-level equivalent of expression -f for floating point argument f.

返回和浮点数参数-f 相等的二进制

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.

参数和返回结果都是无符号整数，但是可以解释成单精度浮点数的二进制表示

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) {
    /*
    1.题目翻译：题目的要求就是更改一下浮点数的符号位，特别需要注意的是当参数为NaN的时候返回参数本身
    2.IEEE浮点表示（32bits）：
    |s（符号位，1位）|exp（阶码，8位）| Frac（尾码，23位）|
    3.通过判断exp是否为11111111，frac是否不等于000… 来判断参数是否为NaN(不是一个数)
    4.通过^改变符号位
    5.exp的几种特殊情况
    exp=00000000:阶码E=-Bias+1; 尾数M=0.xxx…
    exp=11111111,frac=000… 0:s=1时表示负无穷s=0时表示正无穷
    exp=11111111,frac！=000… 0:NaN 一些无法表示的数
    */
    int exp=(uf<<1)>>24;//左移一位以去除符号位
    if(exp==0xff)
    {
        if(uf<<1!=0xff000000) return uf;
    }
    return uf^(1<<31);//和0异或不变 和1异或取反
}
```

2.14 实验任务 14

任务名称：float_i2f 函数：

float_i2f - Return bit-level equivalent of expression (float) x　返回 int x 的浮点数的二进制形式

Result is returned as unsigned int, but

it is to be interpreted as the bit-level representation of a

single-precision floating point values　返回的是 unsigned 型但是表示的时二进制单精度形式

Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

Max ops: 30

Rating: 4

```
unsigned float_i2f(int x) {
    unsigned sign,shift_count,tail,rank,flag,temp;
    if(!x) return 0; //参数为0直接返回0
    sign=x&(1<<31);  //取得符号位
    if(sign) x=-x; //参数为负数则转化为其绝对值
    shift_count=0;
    tail=x;
    while(1)
    {
        temp=tail;
        tail<<=1;  //tail多左移一次，最终保存的是去除首位1后的数
        shift_count++;
        if(temp&0x80000000) break; //temp最终保存的是最高位为1的数
    }
    //tail<<=1; //去除首位1
    if((tail&0x01ff)>0x0100) flag=1;
    else if((tail&0x03ff)==0x0300) flag=1;
    else flag=0;
    tail=(tail>>9)+flag;
    rank=(127+32-shift_count)<<23;
    return sign+rank+tail;
}
```

2.15 实验任务 15

任务名称：float_twice 函数：

float_twice - Return bit-level equivalent of expression 2*f for floating point argument f.

返回以 unsinged 表示的浮点数二进制的二倍的二进制 unsigned 型

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) {
    //如果阶码为0，那么就是非规格数，直接将尾数左移1位到阶码域上，其他不变即可
    //例如 0 00000000 1000… 001 变成 0 00000001 000… 0010
    //这样可以做的原因正是由于非规格化数的指数E = 1 - bias，而不是-bias
    //这样使得可以平滑地从非规格数过渡到规格化数。
    //如果阶码不为0且不是255，那么直接阶码加1即可。
    //如果阶码为255，那么是NaN，∞ ，-∞ ，直接返回。
    if((uf & 0x7F800000)==0)//阶码全为0
    uf = (((uf & 0x007FFFFF)<<1)|(uf & 0x80000000));
    else if((uf & 0x7F800000)!=0x7F800000)//阶码不全为1
    uf = uf + 0x00800000;
    return uf;
}
```

2.16 实验任务 16

任务名称：在 Ubuntu 中编译函数并检查错误

使用./btest 来检查测试所有函数的正确性并打印出错误消息。

```
szh@ubuntu:~$ cd datalab-handout
szh@ubuntu:~/datalab-handout$ ./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
szh@ubuntu:~/datalab-handout$
```

3 总结

3.1 实验中出现的问题

理解说明文档和英文的 bits.c 文件中的内容花了我较长时间，希望能将 bits.c 中的注释改为中文，方便学生理解。同时有些运算方式较难想到或者容易出现纰漏，但在查阅资料和深入思考后都得以解决

3.2 心得体会

在编写代码的过程中一开始只按照寻常逻辑的思路完成，后来经过调试发现按位操作要考虑到正负数之分，尤其是对于除法之类的运算，不能简单向下舍入。

对于一些较难的函数（例如 ilog2），我的第一想法都是运用 while 循坏去实现，但是发现不能使用 while 循环时，陷入了迷茫。后来经过查阅网上资料，经过自己的理解之后再独立编码，最终成功解决了所有问题。

当写到浮点数部分时，由于对浮点数的掌握还不太熟悉，只能根据老师讲解的内容慢慢推导，从而写出代码。但是在实现 float_i2f 函数时，因为没考虑到 0 和-1 的特殊性，所以第一次运行 btest 程序时该函数无法通过，根据终端打印的信息才考虑到两种情况，因此把它们作为特殊情况进行考虑。

同时，对于本次实验，我认为我们应该摆脱寻常套路，深入位运算操作之中，才能成功完成实验。