

# CASAPP : datalab 实验报告

## 实验目标

datalab 实验要求我们使用限定的运算符 `&` `|` `<<` `>>` `!` 等，完成每一个函数的功能并满足所有限制条件。

文件包中有以下文件，注释了各自的功能。

```
Makefile      - Makes btest, fshow, and ishow
README        - This file
bits.c        - The file you will be modifying and handing in
bits.h        - Header file
btest.c       - The main btest program
  btest.h     - Used to build btest
  decl.c      - Used to build btest
  tests.c     - Used to build btest
  tests-header.c - Used to build btest
dlc*          - Rule checking compiler binary (data lab compiler)
driver.pl*    - Driver program that uses btest and dlc to autograde bits.c
Driverhdrs.pm - Header file for optional "Beat the Prof" contest
fshow.c       - Utility for examining floating-point representations
ishow.c       - Utility for examining integer representations
```

## 题目及解法

### bitAnd

第一题较为简单  $A \& B = \sim((\sim A) | (\sim B))$

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

```
int res = ~((~x)|(~y));
return res;

}
```

## getBytes

题目要求返回 32 位数 x 中的第 n 个字节，通过移位运算和掩码即可。

```
/*
 * 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) {

    int shifts = n << 3;
    int res = x >> shifts;
    res &= 0xff;

    return res;

}
```

## logicalShift

题目要求实现逻辑右移，我的想法是通过加法将算术右移填充的 1 溢出成 0，还可以在 x >> 1 之后将符号位直接清 0。

```
/*
 * 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) {

    int bias = (x >> 31) & 2;
    int res, tem;
```

```

    tem = (31 ^ n);
    bias <= tem;
    res = bias + (x >> n);

    return res;

}

```

## bitCount

题目要求我们统计 32 位数字二进制表示中数字 1 出现的次数，并且限定操作符在 40 个以内。

老师上课时讲过一种统计方法，如图。因为有  $2^n - 2^{n-1} - 2^{n-2} \dots - 1 = 1$  通过图中计算 tmp 的方式，可以每 3 位为一组，计算这一组中 1 的个数。  $(tmp + (tmp \gg 3)) \& 0x030707070707$  将相邻红蓝块数值合并入蓝块中。因为  $1 \equiv 64 \pmod{63}$  所以对 63 取模就得到了蓝块之和，即答案。

统计一个机器字的二进制表示中“1”的个数

```

int bitcount(unsigned int n)
{
    unsigned int tmp;

    tmp = n - ((n >> 1) & 033333333333)
           - ((n >> 2) & 011111111111);
    return ((tmp + (tmp >> 3)) & 030707070707) % 63;
}

```

MIT 6.034 Lecture 4.60

L2 Chp2.35

Wu Spring 19 ©USTC

而我采用了类似线段树值合并的思想，有点类似上述方法，通过移位和加法，逐步将 1 的个数统计出来。

```

/*
 * bitCount - returns count of number of 1's in word
 * Examples: bitCount(5) = 2, bitCount(7) = 3
 * Legal ops: ! ~ & ^ | + << >>

```

```

*   Max ops: 40
*   Rating: 4
*/
int bitCount(int x) {

    int bias1 = 0x55;
    int bias2 = 0x33;
    int bias3 = 0x0f;
    int bias4 = 0xff;
    int bias5 = 0xff;

    bias1 |= bias1 << 8;
    bias1 |= bias1 << 16;

    bias2 |= bias2 << 8;
    bias2 |= bias2 << 16;

    bias3 |= bias3 << 8;
    bias3 |= bias3 << 16;

    bias4 |= bias4 << 16;

    bias5 |= bias5 << 8;

    x = (x & bias1) + ((x >> 1) & bias1);
    x = (x & bias2) + ((x >> 2) & bias2);
    x = (x & bias3) + ((x >> 4) & bias3);
    x = (x & bias4) + ((x >> 8) & bias4);
    x = (x & bias5) + ((x >> 16) & bias5);

    return x;

}

```

## bang

当  $x \neq 0$  时  $!x = 0$  ,  $x = 0$  时  $!x = 1$  题目要求不使用 `!` 求出  $!x$  , 只需要检查  $x$  二进制表示中是否有 `1` 即可。

```

/*
* bang - Compute !x without using !
* Examples: bang(3) = 0, bang(0) = 1
* Legal ops: ~ & ^ | + << >>
* Max ops: 12
* Rating: 4
*/
int bang(int x) {

```

```

x |= x >> 16;
x |= x >> 8;
x |= x >> 4;
x |= x >> 2;
x |= x >> 1;

return (x & 1) ^ 1;

}

```

## tmin

要求输出最小的有符号整数( int )

```

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

    int res = 1 << 31;
    return res;

}

```

## fitsBits

题目问  $x$  能否表示为  $n$  位二进制数。我一开始的做法十分复杂，其实只要将  $x$  截断为  $n$  位再还原回去，比较与原来的数是否相等就行。因为  $n$  位二进制数也是有符号的，所以如果有操作引起数值符号的改变导致与原本不相等的情况，正是压缩为  $n$  位之后最高位翻译成符号位的表现。

```

/*
 * 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) {

    int m = 33 + ~n;
    int t = x << m >> m;

    return !(t ^ x);

}

```

## divpwr2

就是对  $x$  进行向  $0$  舍入的移位，当  $x < 0$  时加一个偏置即可。

```

/*
 * 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) {

    int s = (x >> 31) & 1;
    int res = x + ((s << n) + ~0 + !s) >> n;

    return res;

}

```

## negate

$-x = \sim x + 1$

```

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

```

```
int negate(int x) {

    return ~x + 1;

}
```

## isPositive

判断 `x` 是否大于 `0`，注意特判 `0` 的情况。

```
/*
 * isPositive - return 1 if x > 0, return 0 otherwise
 * Example: isPositive(-1) = 0.
 * Legal ops: ! ~ & ^ | + << >>
 * Max ops: 8
 * Rating: 3
 */
int isPositive(int x) {

    int res = (x >> 31) & 1;
    res |= !x;
    return !res;

}
```

## isLessOrEqual

询问是否 `x <= y`，则可以通过先比较符号位，不等时可以直接出答案，当符号位相等时则使用减法判断大小而不会产生溢出的问题。

```
/*
 * 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 s1 = (x >> 31) & 1;
    int s2 = (y >> 31) & 1;
    int res = (!s2) & s1;
```

```

    res |= (!(s1 ^ s2)) & (!(y + ~x + 1 >> 31));

    return res;

}

```

## ilog2

题目问 `x` 的最高位 `1` 在第几位上，但给出了比较严格的限制。这题的思路有点类似二分法，运用了整数能被唯一地表示为一个二进制数的知识。（通过 `!!` 符号可以将非零整数映射为 `1`）

```

/*
 * ilog2 - return floor(log base 2 of x), where x > 0
 *   Example: ilog2(16) = 4
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 90
 *   Rating: 4
 */
int ilog2(int x) {

    int res = 0;
    int t = !(x >> 16);
    res |= t << 4;
    t = !(x >> res + 8);
    res |= t << 3;
    t = !(x >> res + 4);
    res |= t << 2;
    t = !(x >> res + 2);
    res |= t << 1;
    t = !(x >> res + 1);
    res |= t;

    return res;
}

```

## float\_neg

输出 `-f`，但特别注意 `NaN` 的情况。

```

/*
 * 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 & 0x7F800000) != 0x7F800000 || !(uf & 0x007FFFFF))
        uf ^= 1 << 31;

    return uf;

}

```

## float\_i2f

题目要求将 `int` 类型转化为 `float` 类型。先判断 `x` 的符号，但由于 `-tmin = tmin` 所以先将其特判掉。接着按照类型转换的逻辑写下去即可，但要注意规格化问题，由于 `float` 与 `int` 精度差别，会导致舍入问题，还应考虑舍入导致的进位问题。

```

/*
* 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) {

    unsigned res = 0;
    unsigned cnt = 0;
    unsigned y;
    int i, j;
    int zero;

    if (!x) return 0;
    if (x == 0x80000000) return 0xc0000000;

    if (x < 0) {
        x = -x;
        res = 1 << 31;
    }
}

```

```

    for (i = 31; i > 0; --i) {
        if (x & (1 << i)) break;
    }
    cnt = i;
    res |= cnt + 127 << 23;
    y = x << 32 - cnt;
    i = y & 0x100;
    j = y & 0x0ff;
    y >>= 9;
    res |= y;
    if (i == 0x100){
        if (j != 0 || (y & 1)) ++res;
    }

    return res;

}

```

## float\_twice

题目要求计算  $2 * u2f(f)$ 。非规格化数只需整体左移再维护符号位不变就能达到乘二的目的，因为若在指数部分产生溢出正好将数字转为规格化(隐藏最高位 1)，对于规格化数指数加一就能达到要求，而对于 NaN 及 无穷 特判返回即可。

```

/*
 * 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) {

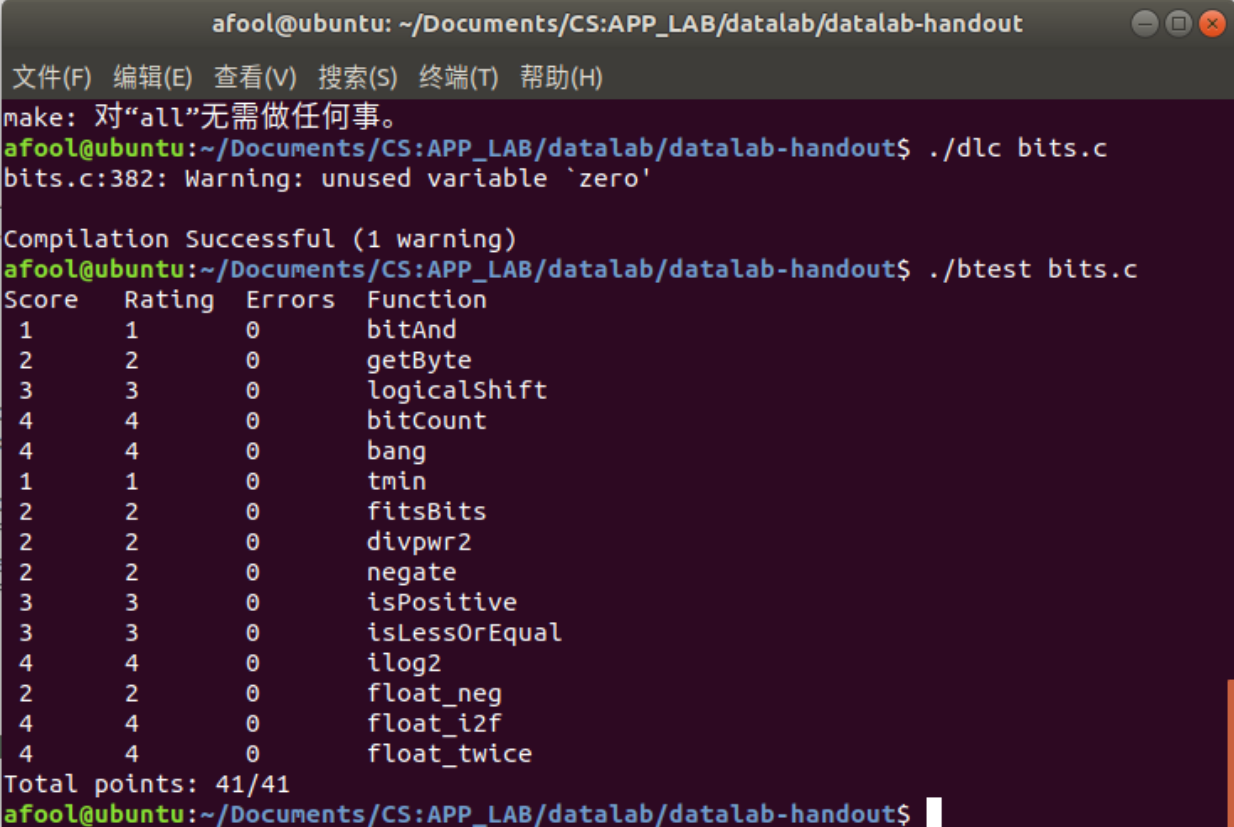
    unsigned s = uf & (1 << 31);
    unsigned e = uf & 0x7f800000;
    unsigned f = uf & 0x007fffff;
    if (e == 0x7f800000) return uf;
    if (e == 0) {
        uf <<= 1;
        uf = (uf & 0x7fffffff) | s;
        return uf;
    }
}

```

```
e = (e >> 23) + 1 << 23;  
return s | e | f;  
  
}
```

## 实验结果

本实验所有资源在 <https://github.com/Afool1999/CSAPP-Labs/tree/master/datalab>



```
afool@ubuntu: ~/Documents/CS:APP_LAB/datalab/datalab-handout  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
make: 对“all”无需做任何事。  
afool@ubuntu:~/Documents/CS:APP_LAB/datalab/datalab-handout$ ./dlc bits.c  
bits.c:382: Warning: unused variable `zero'  
  
Compilation Successful (1 warning)  
afool@ubuntu:~/Documents/CS:APP_LAB/datalab/datalab-handout$ ./btest bits.c  
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  
afool@ubuntu:~/Documents/CS:APP_LAB/datalab/datalab-handout$
```