《计算机系统》

DataLab 实验报告

班级：计科 210X

学号：202108010XXX

姓名：甘晴void

目录

一.实验题目及要求

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

1、不能使用 for while if 等

2、只能使用! ~ & ^ | + << >>运算符

3、只能使用 int

4、只能使用 0-0xFF 的常数

5、使用运算符数不超过限制(Max ops)

6、不能使用全局变量或调用函数等其他规则

浮点数

1、可以使用 for while if

2、只能使用 int，unsigned int

3、使用运算符数不超过限制(Max ops)

4、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. 实验过程

1、bitAnd

实验要求：使用按位或和按位取反实现按位与

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

思路：使用德摩根定律将“按位与”操作转换为“按位取反”和“按位或”操作。

解答：

```
int bitAnd(int x, int y)
{
  return ~(~x | ~y);
}
```

2、getByte

实验要求：从 x 中取第 n 个有效位的 4 位数字

```
/*
 * 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
 */
```

思路：先将 x 右移 8*n 个位置，再与 0x000000ff 相与消去前 24 位。

解答：

```
int getByte(int x, int n)
{
  return ((x >> (n << 3)) & 0xff);
}
```

3、logicalShift

实验要求：将 x 逻辑右移 n 位

```
 * 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
```

思路：由于直接对 x 右移的结果是算术右移，即会将可能的符号位向右补齐。所以我们要构造一个左边 n 位都是 0 的数与我们的结果相与，就可以消掉这个影响。

解答：

```
int logicalShift(int x, int n)
{
  return ((x >> n) & (~(((1 << 31) >> n) << 1)));
}
/*
```

4、bitCount

实验要求：统计出 x 的二进制表示中有多少个 1

```
 * bitCount - returns count of number of 1's in word
 *   Examples: bitCount(5) = 2, bitCount(7) = 3
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 40
 *   Rating: 4
```

思路：先构造出 0x55555555、0x33333333、0x0f0f0f0f、0x00ff00ff、0x0000ffff，

之后采用分治的算法，分别统计每 2 个、每 4 个、每 8 个、每 16 个、与最后的整体 32 个所含 1 的个数。

解答：

```
int bitCount(int x)
{
  int t1, t2, t3, t4, t5;
  t1 = 0x55;
  t1 = t1 + (t1 << 8);
  t1 = t1 + (t1 << 16);
  // t1=0x55555555
  t2 = 0x33;
  t2 = t2 + (t2 << 8);
  t2 = t2 + (t2 << 16);
  // t2=0x33333333
  t3 = 0x0f;
  t3 = t3 + (t3 << 8);
  t3 = t3 + (t3 << 16);
  // t3=0x0f0f0f0f
  t4 = 0xff;
  t4 = t4 + (t4 << 16);
  // t4=0x00ff00ff
  t5 = 0xff;
  t5 = t5 + (t5 << 8);
  // t5=0x0000ffff
  x = (x & t1) + ((x >> 1) & t1);
  x = (x & t2) + ((x >> 2) & t2);
  x = (x & t3) + ((x >> 4) & t3);
  x = (x & t4) + ((x >> 8) & t4);
  x = (x & t5) + ((x >> 16) & t5);
  return x;
}
```

5、bang

实验要求：不用！达到逻辑反的结果

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

思路：本题是经过了学习之后才理解的。这道题的本质就是判断 x 是否为 0，若为 0，就输出 1，否则就输出 0。从补码上来看，一个数的补码必然与本身不同，补码等于本身的数只有 0x00000000 和 0x80000000，即 0 和补码最小数。但是我们发现，只有 0 与其补码按位或之后为 0，结果输出 1。包括补码最小数在内的其他数与其补码按位或结果为 1，此时右移 31 位出现-1，结果输出 0。

解答：

```
int bang(int x)
{
  return ((x | (~x + 0x1)) >> 31) + 1;
}
```

6、tmin

实验要求：返回补码最小值

```
* tmin - return minimum two's complement integer
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 4
*   Rating: 1
```

思路：输出最小的补码 0x80000000，即符号位为 1，后面全是 0

解答：

```
int tmin(void)
{
  return (0x1 << 31);
}
```

7、fitsBits

实验要求：判断 x 是否能被 n 位补码表示

```
* 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
```

思路：若 x 可以被 n 位补码表示，则 x 的第（n+1）位到第 32 位应该都是无效位，则将 x 先左移（32-n）位再右移（32-n）位，若与原来的 x 相同（使用异或来判断），则它的确可以被表示。

解答：

```
int fitsBits(int x, int n)
{
  return !(((x << (32 + (~n) +1)) >> (32 + (~n) +1)) ^ x);
}
```

8、divpwr2

实验要求：计算 x 整除 2^n 后的结果

```
* 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
```

思路：右移 n 位，对负数进行偏移处理。

解答：

```
int divpwr2(int x, int n)
{
  int bias = ((1 << n) + (~1 + 1)) & (x >> 31);
  return ((x + bias) >> n);
}
```

9、negate

实验要求：返回 x 的相反数

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

思路：对于补码来说，取反加一即可。

解答：

```
int negate(int x)
{
  return ((~x) + 0x1);
}
```

10、isPositive

实验要求：

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

思路：一个正数 x 等价于 x 符号位为 0 并且 x 不为 0x00000000，这里由于不能使用条件符，需要使用德摩律做一个小小的转换。

解答：

```
int isPositive(int x)
{
  return !((x>>31)|(!x));
}
```

11、isLessOrEqual

实验要求：比较 x 是否小于等于 y

```
* isLessOrEqual - if x <= y  then return 1, else return 0
*   Example: isLessOrEqual(4,5) = 1.
*   Legal ops: ! ~ & ^ | + << >>
*   Max ops: 24
*   Rating: 3
```

思路：分情况讨论；若 x 与 y 同号，则 x-y 的符号位为 1 或者 x-y=0；若 x 与异号，则 x 小于 0，使用按位或来连接这两个情况即可。

解答：

```
int isLessOrEqual(int x, int y)
{
  int sign = !((x >> 31) ^ (y >> 31));
  return (sign & (((x + (~y) + 0x1) >> 31) | !(x + (~y) + 0x1)) | (!sign & (x >> 31)));
}
```

12、ilog2

实验要求：求 floor(log2(x))

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

思路：题目本质是求 x 的二进制码最高位在哪里。采用分治的思想，分区域寻找，用 y 来保存当前寻找的地址。每进一步寻找就重新保存更新后的地址，这样逐步缩小范围直至最终找到答案。

解答：

```
int ilog2(int x)
{
  int y = 0;
  y = (!!(x >> 16)) << 4;
  y = y + ((!!(x >> (y + 8))) << 3);
  y = y + ((!!(x >> (y + 4))) << 2);
  y = y + ((!!(x >> (y + 2))) << 1);
  y = y + (!!(x >> (y + 1)));
  return y;
}
```

13、float_neg

实验要求：

返回浮点参数 f 的表达式-f 的位等效项。参数和结果都作为无符号 int 传递，但是它们将被解释为单精度浮点值的位级表示。当参数为 NaN 时，返回参数。

```
* 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
```

思路：

函数的参数可能为 NaN，所以需要进行判断。如果参数 uf 的第 23 位到第 30 位全为 1，而且 uf 的低 23 位不为 0，这说明 uf 解释为单精度浮点值的位级表示时是一个 NaN，所以应该直接返回，否则应直接将 uf 的最高位（符号位）取反即可得到-f。

解答：

```
unsigned float_neg(unsigned uf)
{
  unsigned w, x, y, z, t;
  w = (1 << 23) - 1;
  x = 0xff << 23;
  y = uf & w;
  z = uf & x;
  if ((z == x) && y)
  {
    return uf;
  }
  t = (1 << 31) ^ uf;
  return t;
}
```

14、float_i2f

实验要求：返回表达式（浮点数）x 的等价位。 结果以 unsigned int 形式返回，但是将其解释为单精度浮点值的位级表示。

```
* 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
```

思路：若 x 为 0，则直接返回，否则按照 int 分别求阶码、尾数并赋给 float 的相应位置。这

里使用 while 将 absx 不断向左移位，当 absx 的最高位 1 所在位置到达最左边的时候，表示位移已经到了尽头，此时记录下向左移位的次数，用 31 减去它，可以得到未修正的阶码 e，e 再考虑偏置量 127 就可以得到阶码 E。这个过程就相当于将 absx 向右移位直到形成 1.M 这样的结构，同时刚刚 absx 移位的结果也就是尾数 M 的值。这里的 tail 是 absx 移位之后的末八位，如果 tail 比 0x10000000 大，表示这里是“舍入”的“入”，最后还需进 1；如果 tail 恰好等于 0x10000000，此时考虑的是“舍入”的特殊情况，向偶舍入。此时考虑前面尾数 M 的末位是否为 1，若为 1，则要进位，向偶数舍入。

解答：

```
unsigned float_i2f(int x)
{
  unsigned sign = 0, enow = 0, fnow = 0, absx = x,
           shiftLeft = 0, tail = 0, result = 0;
  unsigned pos = 1 << 31;
  if (x == 0)
  {
    return 0;
  }
  else if (x < 0)
  {
    absx = -x;
    sign = pos;
  }
  while ((pos & absx) == 0)
  {
    absx <<= 1;
    shiftLeft += 1;
  }
  enow = 127 + 31 - shiftLeft;
  tail = absx & 0xff;
  fnow = (~(pos >> 8)) & (absx >> 8);
  result = sign | (enow << 23) | fnow;
  if (tail > 0x80)
  {
    result += 1;
  }
  else if (0x80 == tail)
  {
    if (fnow & 1)
    {
      result += 1;
    }
  }
  return result;
}
```

15、float_twice

实验要求：返回浮点参数 f 的表达式 2 * f 的位等效项。参数和结果都作为 unsigned int 传递，但是它们将被解释为的位级表示。当参数为 NaN 时，返回参数。

```
* 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
```

思路：按照如下方式，分别分离原 float 的阶码、尾数并计算得到答案 float 的阶码，尾数最后返回即可。具体步骤如下。先判定，若为 0x00000000 或者 0x80000000（就是 0），则没有对它乘的必要，直接返回即可。否则分离出它的符号 S、阶码 E 和尾数 M。若阶码为 0xff（即 11111111），那么这是一个表示 NaN 或者 inf 的数，不必处理，直接返回即可。否则考虑若其阶码为 0x0（即 00000000），那么这是一个就绝对值较小数，我们先对它的尾数 M 左移，如果有进位，就表示阶码需要变了，阶码进一；若没有进位，则不需要特殊处理。若其阶码不是特殊值，那么只需要对阶码进行+1 操作就可以使它倍增了。最后将这些 S、E、M 部分拼接即可得到最终的答案。

解答：

```c
unsigned float_twice(unsigned uf)
{
  unsigned sign = 0, enow = 0, fnow = 0;
  unsigned pos = 1 << 31;
  unsigned frule = (1 << 23) - 1;
  if (uf == 0)
  {
    return 0;
  }
  if (uf == pos)
  {
    return uf;
  }
  sign = uf & pos;
  enow = (uf >> 23) & 0xff;
  if (enow == 0xff)
  {
    return uf;
  }
  fnow = uf & frule;
  if (enow == 0)
  {
    fnow = fnow << 1;
    if (fnow & (1 << 23))
    {
      fnow = fnow & frule;
      enow += 1;
    }
  }
  else
  {
    enow += 1;
  }
  return sign | (enow << 23) | fnow;
}
```

16、验证

（1）使用./dlc -e bits.c 检验操作合规性以及操作数合法性

```
wolf@wolf-VirtualBox:~/datalab/datalab-handout$ ./dlc -e bits.c
dlc:bits.c:143:bitAnd: 4 operators
dlc:bits.c:155:getByte: 3 operators
dlc:bits.c:167:logicalShift: 6 operators
dlc:bits.c:202:bitCount: 36 operators
dlc:bits.c:213:bang: 5 operators
dlc:bits.c:223:tmin: 1 operators
dlc:bits.c:236:fitsBits: 10 operators
dlc:bits.c:249:divpwr2: 8 operators
dlc:bits.c:260:negate: 2 operators
dlc:bits.c:271:isPositive: 4 operators
bits.c:282: Warning: suggest parentheses around arithmetic in operand of |
dlc:bits.c:283:isLessOrEqual: 18 operators
dlc:bits.c:300:ilog2: 27 operators
dlc:bits.c:325:float_neg: 9 operators
dlc:bits.c:371:float_i2f: 23 operators
dlc:bits.c:418:float_twice: 20 operators
```

（2）使用./btest 检验代码对数据正确性

```
Compilation Successful (1 warning)
wolf@wolf-VirtualBox:~/datalab/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
```

（3）上述两步也可以被简化为使用./driver.pl 进行验证。

```
Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       4       bitAnd
2       2       0       2       3       getByte
3       3       0       2       6       logicalShift
4       4       0       2       36      bitCount
4       4       0       2       5       bang
1       1       0       2       1       tmin
2       2       0       2       10      fitsBits
2       2       0       2       8       divpwr2
2       2       0       2       2       negate
3       3       0       2       4       isPositive
3       3       0       2       18      isLessOrEqual
4       4       0       2       27      ilog2
2       2       0       2       9       float_neg
4       4       0       2       23      float_i2f
4       4       0       2       20      float_twice

Score = 71/71 [41/41 Corr + 30/30 Perf] (176 total operators)
```

（4）可以验证代码正确。

三、实验总结

DataLab 是一个用来检验我们对于计算机系统机器表示以及位运算的理解的一个非常好的实验。通过实验，我熟悉了位的基本操作，更进一步熟悉了数的机器表达形式以及特点。熟悉了异或取反判等，移位和 0x1 与得符号位等基本的位操作，通过使用一些数的特性，算数右移左移操作特点进一步熟练了整数和浮点数在位级的表示与一些运算方法。

此外还有与老师和同学的讨论等，这些都能为最终的结果带来很大的帮助。当然最重要的还是知识本身的理解，通过不断地试错我一步一步地改进方法，最终向着正确迈进。

另外的是一些解题的技巧与经验的总结。

（1）清除(保留)特定的位：使用特殊的数进行与运算（0xFF，0x1FF..）

（2）分治：将问题分解为 16 位，8 位，4 位..逐步处理(或自底向上处理)

（3）取反+1 得到相反数

（4）0x0 和 0x80000000 的相反数是本身

（5）作差判断大小，考虑符号不同的溢出情况

（6）浮点数乘 2：规格化阶码+1，非规格化左移 1 位