《计算机系统》

LAB2-datalab 实验报告

目录

1 实验项目一

1.1 项目名称

LAB2-datalab。

1.2 实验目的

1) 本次实验本质上是填写 bits.c 文件中尚未完成的各个函数的内容。但是，本实验要求只使用有限数量、规定的操作符。

2) 对于整型：明确禁止:

1. 使用任何控制结构，如 if、do、while、for、switch 等。
2. 定义或使用任意宏。
3. 在此文件中定义任何附加函数。
4. 调用任意函数。
5. 使用任何其他操作，如&&、||、-或?:
6. 使用除 int 以外的任何数据类型。这意味着不能使用数组、结构体或联合。

3) 对于浮点：对于需要实现浮点运算的问题，编码规则不那么严格。允许使用循环和有条件的控制。可以同时使用 int 型和 unsigned 型。可以使用任意整数和无符号常量。明确禁止您:

1. 定义或使用任意宏。
2. 在此文件中定义任何附加函数。
3. 调用任意函数。
4. 使用除 int 或 unsigned 以外的任何数据类型。这意味着不能使用数组、结构体或联合。
5. 使用任何浮点数据类型、操作或常量。

1.3 实验资源

datalab-handout.tar 以及 ReadmeFirst.txt。

2 实验任务

2.1 bitAnd

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

题目要求：只能用~和|实现两个二进制数的按位与功能。

根据德摩根定律转换为按位取反后，取或操作，再取反。

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

2.2 getByte

```
/*
 * 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
 */
```

题目要求：从低位起字节编号为 0-3，取出编号为 n 的字节。

将所取的字节挪到最右边，与 0xff 进行与操作，即可取出该字节。因为一个字节是 8 位，需要将给的 n 先向左移 3 位，以实现乘 8，即可实现向右移位的过程。

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

}
```

2.3 logicalShift

```
/*
 * 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
 */
```

题目要求：实现逻辑右移

在本实验中，对整数的操作默认为算术右移，而实现逻辑右移 n 位，只需要将算术右移 n 位的内容与 000……111，即 n 个 0，32-n 个 1 进行相与即可。本题重点为如何得到该数字。

需要注意的是：（1）只能使用 0-0xff 之间的数，不允许直接使用大整数。（2）0 在二进制中用 32 个 0 表示，按位非可以得到 32 位为 1 的数。（3）整数使用补码表示（4）不允许使用‘-’。

这里给出了两种思路。

思路 1：将 1 向左移 31 位，得到首位为 1 的 32 位数，再向右移 n-1 位，由于默认为算术右移，会得到前 n 位为 1 的数字，然后取反，即可得到想要的数。

思路 2：将 32 位全 1 的数向左移 32-n 位（右边补 0），然后取反，即可得到想要的数。要注意的是本题中不能使用-，在这里将 32+~n+1，即进行补码运算。

```
int logicalShift(int x, int n) {
  int s=1<<31;
//  int temp=~((s>>n)<<1);
  int temp=~((~0)<<(32+~n)<<1);
  return (x>>n)&temp;
}
```

2.4 bitCount

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

题目要求：数出二进制数中 1 的个数。

本题中不能使用 if,while 等控制结构，所以不能使用每次右移 1 位与 0x1 相与的方法，

查阅资料后发现可使用分治法。

思路 1：

将所有位每两位为 1 组，先计算每 2 位的 1 的个数，再计算每 4 位 1 的个数，每 8 位 1 的个数，每 16 位 1 的个数，最后得到 32 个位 1 的个数，最终结果将被累计为最低位数。

无需担心移位是算术右移，右移得到的最高位在计算时会被丢弃。

计算个数的方式是和特殊的数相与（0x55555555,0x33333333,0x0F0F0F0F，0x00FF00FF,0000FFFF），每次计算的方式是利用移位操作将相邻两组的值相加，存放在较低位处。需要的数按照规则不可以直接使用，需要先通过移位相加得到。

t1:01010101*4，t2:00110011*4，t3:00001111*4，t4:0000000011111111*2，t5:0…0011111111

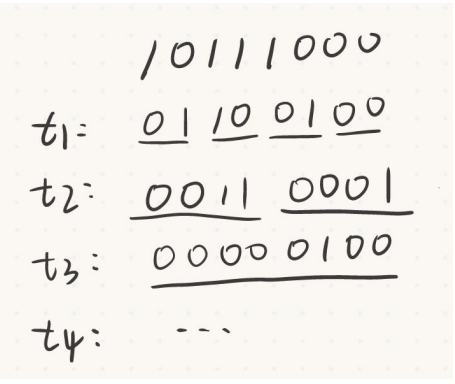
（0 的位置是将高位空出来，累加过程中将划分的每一块当做计数处理，2 的五次方是 32，所以会处理 5 次。）

思路 2：

使用 0x11111111 作为 t1，每次检查八位，也就是每四位中的 1 的个数，在第一次操作时需要累加四次，然后再分别检查高 16 位和低 16 位累加（0xffff，sum>>16，累加后仅剩低 16 位）、相邻四位累加（0x0f0f，sum>>4，累加后还剩低 16 位，这里是先对 8 位内分两块进行了累加）、相邻八位累加（0xff，sum>>8，得到最终结果）。

这里给出了思路 1 的实现。

```
14 int bitCount(int x) {
15   int t1,t2,t3,t4,t5;
16   t1=0x55+(0x55<<8);
17   t1=t1+(t1<<16);
18   t2=0x33+(0x33<<8);
19   t2=t2+(t2<<16);
20   t3=0x0f+(0x0f<<8);
21   t3=t3+(t3<<16);
22   t4=0xff+(0xff<<16);
23   t5=0xff+(0xff<<8);
24
25   x=(t1&(x>>1))+(t1&x);
26   x=(t2&(x>>2))+(t2&x);
27   x=(t3&(x>>4))+(t3&x);
28   x=(t4&(x>>8))+(t4&x);
29   x=(t5&(x>>16))+(t5&x);
30   return x;
31 }
```



2.5 bang

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

题目要求：实现取非操作

思路 1：

等价于求二进制串中是否含有 1，与上一题的思路类似，但不用计数，直接取或即可。高 16 与低 16，16-8 与低 8，8-4 与低 4，4-2 与低 2，2 与 1，最终返回 32 位相或的结果。

无需担心算术右移会影响结果，只有含有 1 的时候才会补 1。

因为没有进行与操作，高位的 1 或 0 会被保留，最终返回结果的时候记得与 0x1，使最终的结果返回 1 或 0 而不是其他的数字。

思路 2：

查资料的时候发现了另一种解法，0 取反后+1 仍为 0，只有 0 有该特点，因此将 x 与 x 取反加一后的数按位或，将只有 0 会得到符号位为 0，其余的数符号位均为 1，直接返回符号位右移 31 位并加一的结果即可（符号位为 1 的会发生溢出，返回结果为 0）。

```
int bang(int x) {
  int t1=x|x>>16;
  int t2=t1|t1>>8;
  int t3=t2|t2>>4;
  int t4=t3|t3>>2;
  int t5=t4|t4>>1;
  return (t5&0x1)^0x1;
//x=(~x+1)|x;
//return (x>>31)+1;
}
```

2.6 tMin

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

题目要求：返回补码中的最小值

32 位补码表示的最小值为 0x80000000。补码负数是对应二进制无符号数减表示范围，如四位表示时，最小值为 1000，-8，对应无符号数的 8-16。

```
int tmin(void) {
  return 1<<31;
}
```

2.7 fitsBits

```
/*
 * 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
 */
```

题目要求：求 x 是否可用 n 位补码表示。

思路 1：

符合要求的数的特点应该是从第 n 位到最高位没有有意义的数值，对于正数，前 33-n 位为 0，对于负数，前 33-n 位为 1（加入了符号位，32-n+1）。左移 32-n 位和右移 32-n 位得到的数值应该相同，用异或取反判断。

思路 2：

如果是正数或 0，向右移动 n-1 位后，该数的数值一定为 0，如果是负数，数值一定为 1（32 位全 1 的数），可以移动后判断是否为全 0 或全 1 判断。

```
int fitsBits(int x, int n) {
  int t = 32+(~n+1);
  return !(x^(x<<t)>>t);
//int temp=n+~0;
//int tempx=x>>temp;
//int ans = (!temp|!(tempx+1));
//return ans;
}
```

2.8 divpwr2

```
/*
 * 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
 */
```

题目要求：给出 x/2^n 的结果。

正数直接向右移 n 位即可得到 x/2^n 的答案，负数直接移会得到(x/2^n) -1（向下取整），我们可以根据符号位判断正负，并将负数加上 2^n-1 的偏移量（即不会改变正确结果，也可以让除不尽的部分向 0 取整）。temp 得到的是符号位取反+1，也就是将符号位扩展，符号位为 0 得到 0，符号位为 1 得到全 1 的值；q 是偏移量；ans 将符号位扩展与偏移量相与后右移，即可得到结果。

```
int divpwr2(int x, int n) {
  int temp=(~((x>>31)&0x1))+1;
  int q=(1<<n)+~0;
  int ans=(x+(temp&q))>>n;
  return ans;
```

2.9 negate

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

题目要求：求-x

由补码运算可知，x 与-x 的和为总长度，如果是四位表示，和应该为 10000，即刚刚好溢出的值。求一个补码数的负数，取~x+1 即可。

```
int negate(int x) {
  return ~x+1;
}
```

2.10 isPositive

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

题目要求：判断是否为正数，是返回 1，否则返回 0

本来写的是(x>>31)+0x1，发现漏考虑了 0 的情况，输出错误。

故特殊考虑 0 的情况，使用取反操作，若为 0，取反操作位 1，与 temp 进行或操作，最后取反得到结果。

```
int isPositive(int x) {
  int temp =(x>>31)&0x1;
  int ans = !(temp | !x);
  return ans;
}
```

2.11 isLessOrEqual

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

题目要求：判断 x<=y，若是，返回 1；否则返回 0。

对于符号不同的情况，可直接判断；符号相同时，将两数作差判断结果。

xi,yi 分别提取了 x,y 的符号位，same 用异或判断了符号位是否相同，p 是 y-x 的符号位，若 y-x 大于 0，p 返回 1，即表示 y>x。最后如果相同，返回 p 值，如果不同，返回 xi 的符号值，若 xi 大于 0，结果为 0，即 y<x。

```
int isLessOrEqual(int x, int y) {
  int xi=(x>>31)&0x1;
  int yi=(y>>31)&0x1;
  int same=!(xi^yi);
  int p=!((~x+1+y)>>31);
  return (same&p)|(!same & xi);
}
```

2.12 ilog2

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

题目要求：对一个整数求 log2(x)，向下取整。

思路 1：

相当于找最高位 1 的位置。可以将最高位 1 右侧的位全部通过移位覆盖，并进行 bitCount，将最终的结果-1，就是想要的值。

思路 2：

通过右移并两次取反检测高 16 位是否有 1，若有，结果至少为 16，将 res 加上 16，继续检测前 16 位的高 8 位；如果没有，检测后 16 位的高 8 位，以此类推，逐步检测高 4 位、高 2 位、高 1 位，最终得到结果。每次检测的结果要乘以对应的权重值。

```
int ilog2(int x) {
  int t1,t2,t3,t4,t5;
  t1=0x55+(0x55<<8);
  t1=t1+(t1<<16);
  t2=0x33+(0x33<<8);
  t2=t2+(t2<<16);
  t3=0x0f+(0x0f<<8);
  t3=t3+(t3<<16);
  t4=0xff+(0xff<<16);
  t5=0xff+(0xff<<8);

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

  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&x);
  return x+(~1+1);
```

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

2.13 float_neg

```
/*
 * 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
 */
```

题目要求：返回浮点数 f 的负数，对于 NAN，直接返回 NAN。

注意：该数以 Unsigned 的形式给出，但它是浮点数的二进制表示。

正常情况时只需要用异或操作改变首位，本题目中的特殊情况是 NAN，它的结构是阶码位全部为 1，而小数位不为 1。uf<<9 用于判断小数位的值，((uf>>23)&0xff)==0xff 用于判断阶码的 8 位是否为 1（没有判断符号位），需要注意的是要在==前加括号，因为==的优先级大于&。

```
unsigned float_neg(unsigned uf) {
  if((uf<<9)&&(((uf>>23)&0xff)==0xff)) return uf;
  else return uf^0x80000000;
}
```

2.14 float_i2f

```
/*
 * 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
 */
```

题目要求：将整型转化为浮点数，返回为 IEEE 格式

符号位使用原浮点数的符号位。

阶码部分，参考 ilog2 的操作，取得原值 2 的最高次方。而 ilog2 的条件是正数，将负数取反+1 可得到相反数，需要注意两个特殊值：0，返回 0；0x8000000 取反+1 仍然为原值，是负数，直接手动计算得 0xcf000000 返回。while 循环部分寻找阶码部分最高位的 1，退出时的 E 值是第一个 1 后面的位的个数，因此，E+1 位是从低位起 1 的位置，也就是说所得到的 E 就是所求的对数结果，加上偏移量 127 后移位到对应的位置，得到了阶码部分。（ilog2 不能使用 while）

尾数部分，需要得到完整的尾数，左移 32-E 位可以得到完整的尾数部分（不包括最高位的 1），浮点数尾数有 23 位，所以需要右移 9 位，并清除补充的位。

此时将三个部分相与，得到未处理舍入时的结果。

处理舍入：根据向偶舍入的原则，如果最高位不为 1，小于舍入部分的 1/2，返回原值；如果为 1，判断后 8 位，如果全为 0，根据向偶舍入，判断 res 的最后一位，如果为 1，返回 res+1，如果为 0，返回 res。

整型转换为浮点数不会溢出，不会出现非规格化数。

```
unsigned float_i2f(int x) {
  int s,exp,frac,E=30,bias=127,res=0;
  unsigned int t;
  s=x&0x80000000;
  if(s) x=~x+1;
  //
  if(x==0) return 0;
  if(x==0x80000000) return 0xcf000000;
  //
  while(!((x>>E)&0x1)) E--;
  exp=(E+bias)<<23;
  x=(x<<(32-E));
  frac=(x>>9)&0x007fffff;
  res=s|exp|frac;
  //
  t=x&0x000001ff;
  if(t&0x00000100){
    if(t&0xff)
        return res+1;
    else{
        if(res&0x1) return res+1;
        else return res;
   }
  }
  return res;
}
```

2.15 float_twice

```
/*
 * 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
 */
```

题目要求：返回给定 f 的两倍。

对于规格化数，直接将阶码值+1，对于非规格化数，左移 1 位并补符号位，对于特殊值 NAN 和无穷，返回原值。s 提取符号位，exp 提取阶码位，如果不是全 1 或 0 就将阶码值+1,全为 0 左移补符号，否则不做处理。注意给定的 uf 是无符号数，右移补 0。

```
unsigned float_twice(unsigned uf) {
  int s,exp;
  s=uf&0x80000000;
  exp=(uf<<1)>>24;
  if(exp!=0xff && exp!=0) uf+=0x800000;
  else if(exp==0) uf=(uf<<1)|s;
  return uf;
}
```

3 总结

3.1 心得体会

在实验过程中，我发现我对二进制整数的操作以及浮点数的操作不够熟练，在遇到问题的时候很难想到很好的解决办法，因而在处理问题的时候参考了许多网上的解法，在这个过程中我也发现了许多不同的思路，位处理的灵活性不言而喻。

通过该实验，我进一步熟悉了位的基本操作和对浮点数的处理，复习了许多更加细小的知识点，我也通过本实验中一些灵活的问题开拓了视野，知道了位运算不只是简单的操作，它们组合起来有很多奥妙的地方，甚至可以实现分治这种复杂的思想。

从二进制这一层面出发，我对机器存储、对数的操作都有了更深一步的理解。

以下对学习到的许多思路进行总结：

- ✓ 分治
- ✓ 用高位 1 将剩余位全覆盖/全异或
- ✓ 用偏移量使改变取整方向
- ✓ 补码数的相反数，取~x+1
- ✓ 特殊值：0 的符号位是 0, 0x8000000 按位取反加 1 仍为它本身（0 也是）
- ✓ 异或取反判断相等
- ✓ 两次取非判断 0
- ✓ 浮点数乘 2：规格化阶码+1，非规格化左移该符号，特殊直接返回
- ✓ 负数转正数求 E（最高位 1）
- ✓ 向偶舍入