datalab实验报告

——罗思佳2021201679

1.bitXor

先用`a1 = x & y`将x和y同为1的位置标记为1，再用`a2 = ~x&~y`将x和y同为0的位置标记为1，这两种位置都是要置为0的，所以再将a1和a2取反，然后使用&操作即可得到`x^y`的结果。

```
int a1 = x & y;
  int a2 = ~x & ~y;
  int res = ~a1 & ~a2;
  return res;
```

2.thirdBits

题目是要得到一个二进制数，这个数最低位为1，且从最低位开始每3位为1，因此可以设一个初始数0x49，即01001001，然后向左移9位，与原来的数进行或操作，再左移18位，与上一个数进行或操作，即可得到。

```
int res = 0x49;
  res |= (res << 9);
  res |= (res << 18);
  return res;
```

3.fitsShort

只需将x左移16位，再右移16位，看前后两个数是否相等。

```
int y = x << 16;
  y = y >> 16;
  return !(x ^ y);
```

4.isTmax

1.优化前（6个运算符）

利用Tmax+1和~Tmax相等的特点，同时还需要排除-1的干扰。

```
int a = x + 1;
int b = ~x;
int c = a^b;
return !(c +!a);
```

2.优化后（5个运算符）

将x+1，转换为判断Tmin，因为Tmin+Tmin等于0，再利用！ min 排除0的干扰。

```
int min = x +1;
int b = min + min;
return !(b + !min);
```

5.fitsBits

即将fitsShort的16改为n位

方法1 (6个运算符）：

沿用fitsShort的方法，将16改为`(~n+33)`

方法2（6个运算符）：

如果一个数能用n位表示，那么这个数向右移n-1位后得到的数应该是全1或者全0，全1和全0的特点都是`!x + !(~x) ==1`。

tips：右移n-1可以用右移n+31表示，节省一个运算符。

```
 x = x >> (n+31); //右移n-1
 return !x + !(~x);
```

优化后（5个运算符）

将右移后的数和符号位掩码sign进行异或操作

```
 x = x >> (n+31);
  int sign = x >> 31;
  return !(x ^ sign);
```

6.upperBits

因为n为0时返回0，所以最开始可以定义x = !!n, 当n为0时x也为0，然后左移31位到最高位，然后右移n-1位即可得到。

```
int x = !!n;//排除0的干扰
  x = x << 31;
  n = n + 31;
  x = x >> n;
  return x;
```

7.anyOddBits

由例子可以看出，位数是从0开始的，可以构造一个mask，为0xaa，即最低八位为10101010，先将x的前16位与后16位进行或操作，再将后16位的前8位与后8位进行或操作，最后将x与mask进行与操作，只要x的奇数位出现1，最后的结果就不为0。

```
   x = x | (x >> 16);
   x = x | (x >> 8);
   int mask = 0xaa;
   int res = x & mask;
   return !!res;
```

8.byteSwap

1.优化前（15个运算符）

先将n和m转换成移动的位数n1，m1，再分别将x向右移动n1、m1位得到要移动的字节段，记为y1，y2，再将y1，y2向左移m1、n1，这样两字节的位置就交换了，然后把它们加在一起，并记为y，再将原来数的交换字节的位置置为0，然后与y进行或操作，即可得到结果。

2.优化后（10个运算符）

将要交换的字节分别右移到最右边进行异或操作，然后再移到原来的位置和x进行异或，即可实现交换的目的。

```
  int n1 = n << 3;
  int m1 = m << 3;
  int y = ((x >> n1)^(x >> m1)) & 0xff;
  int res = (y << n1) ^ (y << m1);
  res = res ^ x;
  return res;
```

9.absVal

1.优化前（4个运算符）

如果是正数，返回原值，如果是负数，取反加一。

```
int z = x >>31;
int y = x ^ z;
y = y + (z & 1);
return y;
```

2.优化后（3个运算符）

发现用减一取反可以节省一个操作符。

```
int z = x >> 31;
x = x + z;
x = x ^ z;
return x;
```

10.divpwr2

因为是向0取整，所以正数除以2^n可以直接向右移位，负数如果直接右移是向负无穷取整，所以需要先加一个偏移量再右移，经数值计算得到偏移量是2^n-1。

11.float_neg

先将uf的符号位取反，然后判断uf的阶码是不是NaN，即判断uf的阶码是否全为1，frac是否不为0,如果是的话返回原值，否则返回符号位相反的数。

```
int res = uf ^ 0x80000000;//取反
int t = uf & 0x7fffffff;//取绝对值
  if(t > 0x7f800000) return uf;
 return res;
```

12.logicalNeg

除了0以外的任何数，和它的相反数进行或操作的结果都是负数。

```
int y = (~x) + 1;
  x = x | y;
  x = x >> 31;
  return(x+1);
```

13.bitMask

先构造全1的掩码b，然后将b左移highbit位，再左移1位（要分开移），然后取反得到x；然后将b左移lowbit位得到y，x&y得到结果。

```
int b = ~0;
   int x = b << highbit;
   x = x << 1; x = ~x;
   int y = b << lowbit;
   return x & y;
```

这里附上之前的错误做法：

1.错误做法一

使用^操作符,不能兼顾highbit < lowbit 的情况

```
int b = ~0;
   int x = b << (highbit+1);
   int y = b << lowbit;
   return x ^ y;
```

2.错误做法二

直接左移highbit+1位，当highbit为31时，效果是没有移动。

```
int b = ~0;
   int x = b << (highbit+1);
   x = ~x;
   int y = b << lowbit;
   return x & y;
```

14.isGreater

优化前（12个运算符）

先将y右移31位得到y1，再用`x^y`来判断x与y的符号是否相同，设`z = ~x + y + 1`，即y-x，然后将z右移31位，return的判断条件是，如果x与y符号相同，就看y-x的符号，如果x与y的符号不同，就直接看y的符号。

优化后（10个运算符）

最后进行右移31位的操作，可以节省2个运算符。

```
int m = x^y;
 int z = ~x + y + 1;
 int res = (m & y) | ((~m) & z);
 return (res >> 31) & 1;
```

优化后2.0（9个运算符）

定义变量z = ~y + x; 即x-y-1,当x > y 时，x-y-1的符号位为0，当 x <= y 时，x-y-1的符号位为1。

```
int m = x^y;
int z = ~y + x ;
int res = (m & y) | (~(m|z));
return (res >> 31) & 1;
```

15.logicalShift

需要考虑n==0的情况

法一（9个运算符）

先将x右移n位，然后根据x的符号以及n的值确定掩码，最后进行异或操作，去掉左边可能的1。

```
x = x >> n;
  int y = (x >> 31) & !!n;
  int c = (y << 31) >> (n+31);
  return x ^ c;
```

法二（7个运算符）

思路与法一相似，不同的是构造掩码的方式不同，这里是先将x右移31位得到sign，然后将sign左移31-n位，再左移1位（分开移位同样是考虑了n==0的特殊情况）。

法三（6个运算符）

构造掩码的方式不同

```
x = x >> n;
  int sign = x & (1 << 31);
  sign = sign >> n;
  return x ^ (sign << 1);
```

16.satMul2

关键是x的最高两位，如果是00或11就直接移位，如果是01或10就需要saturate to TMax or TMin.

优化前（19个运算符）

分别构造了min和max,判断条件不够简洁。

```
int w1 = (x >> 31) ; int w2 = (x >> 30) ;
  int min = 1 << 31; int max = ~min;
  int y = x << 1;
  int w = (w1 ^ w2) + (~0);
  w1 = (w1 << 31) >> 31;
  w2 = (w2 << 31) >> 31;
  return (w & y) | (~w & ((w2 & max) | (w1 & min))) ;
```

优化1.0（12个运算符）

只构造TMin，根据左数第二位确定是TMax还是TMin，并根据左边第一位和第二位是否相同来决定是否左移1位。

```
int min = 1 << 31;
  int z1;
  int y = x << 1;
  z1 = y >> 31;
  int z2 = (x ^ y) >> 31;
  x = ((min ^ z1) & z2) | (x & ~z2);
  x = x << (z2 + 1);
  return x;
```

优化2.0（10个运算符）

将x左移的结果直接写在最后判断的语句中，如果满足就直接返回左移的结果。

```
int min = 1 << 31;
  int z1;
  int y = x << 1;
  z1 = y >> 31;
  int z2 = (x ^ y) >> 31;
  int res = ((min ^ z1) & z2) | (y & ~z2);

  return res;
```

17.subOK

关键是比较x和y符号，y和z的符号。

优化前（10个运算符）

```
int z = ~y + x + 1;//z = x - y
  int z1 = (x^y) >> 31;//取符号位
  //z1 = z1 & 1;
  int c = (y^z) >> 31;
  return (z1+1) | (c & 1);
```

优化后（8个运算符）

设`z = ~y + x + 1`计算x-y的值，`m = x ^ y`，如果x和y同号则m的最高位为0，异号则为1，`n = x ^ z`，返回值是`((m & n) >> 31) + 1`，如果x和y同号，相减一定不会溢出，m最高位为0，和n进行&后最高位还是0，右移31位加1后返回1；如果x和y异号，m最高位为1，这时候要看x和z的最高位，如果x和z异号，则溢出，n最高位为1，`(m&n)>>31`结果是-1，加1后结果为0，如果x和z同号，则不溢出，n最高位为0，`(m&n)>>31`结果是0，加1后结果为1。

```
int z = ~y + x + 1;//x-y
int m = x ^ y;  int n = x ^ z;
return ((m & n) >> 31) + 1;
```

18.tureThreeFourths

优化前（14个运算符）

先将x的最低两位保存，记为low2，x右移2位（除以4）得到y，余数是low2，通过找规律发现，当x为正数且不整除4时，y×3后要加上low2-1，当x是负数或0或整除4时，要加上low2。

```
int low2 = x & 0x3;
  int sign = (x >> 31) & 1;
  sign = (sign | !x) | !low2;
  sign = sign + ~0;
  int y = x >> 2;
  int z = y << 1;
  z = z + y;
  int res = z + (low2 + sign);
  return res;
```

优化后（11个运算符）

将x分为整除4的部分和不整除4的余数部分，整除部分进行先除以4再乘3的操作，余数部分因为比较小，进行先乘3再除以4的操作（有利用到divpwr2的技巧），最后加在一起。

```
  int low2 = x & 0x3;
  int y = x >> 2;
  y = (y << 1) + y;
  low2 = low2 + low2 + low2;
  int sign = x >> 31;
  int z = low2 + ( sign & 3);
  z = z >> 2;
  return y + z;
```

19.isPower2

优化前（11个运算符）

如果一个数是2的幂次，那么它和它的相反数进行&操作的结果等于它本身，同时要排除0x80000000和0的特殊情况。

优化后1.0（8个运算符）

如果x是2的幂次，x和x-1的与是0，同时要排除负数和0的情况。

```
int y = x + ~0;//x-1
int z = x & y;//如果是2的幂次，x 和 x-1 的并是0
int sign = x >> 31;
return !(z + sign + !x);//排除负数和0的情况
```

优化后2.0（7个运算符）

定义变量m = x << 1, 这样TMin和0左移一位后都是0，而2的次幂不为0。

```
int y = x + ~0;
  int z = x & y;
  int m = x << 1;
  return !(z + !m);
```

20.float_i2f

首先判断是否是负数，负数的话需要取绝对值。然后用while循环找到第一个1出现的位置index，如果没找到就返回0。然后根据index计算exp的值；将x左移到第二位，记下后八位的值，用low8表示，同时记下尾数最后一位的值（0或1），将low8与最大值的一半halfMax（值为128）相比较，如果大于halfMax要进位，如果等于halfMax，尾数最后一位的1的话就进位，如果小于halfMax则舍弃。得到尾数后右移8位到自己的位置上，再加上符号位和exp。

注意的点：当尾数为全1，进位后阶码要加1。

```
int index = 32;
  unsigned sign = x & 0x80000000;
  if(sign) //如果是负数 (之前没有考虑负数要取反加一)
   x = -x;
  unsigned tmp = x;
  while(index)
  {
 int s = (tmp >> (index-1)) ;
    if(s)  break;
    index = index - 1;;
  }

  if(!index) return 0;

  int exp = index + 126;
          //index - 1 + 127
  tmp = tmp << (33 - index);
  tmp = tmp >> 1;
  int low8 = tmp & 0xff;
  int tail = (tmp >> 8) & 1;
  int halfMax = 128;
  int t = 0x100;
  if(low8 > halfMax)
  {
    tmp = tmp + t;//进位
  }
  if(low8 == halfMax)
```

```
  {
    tmp = tmp + tail;
  }
  tmp = tmp >> 8;
  int high9 = sign + (exp << 23);
  unsigned res = tmp + high9;
  return res;
  //当尾数位最高位有进位时，阶码要加一
```

21.howManyBits

思路：二分查找，先分为左16位和右16位，如果左16位出现要找的位置就向右移16位，sum加16，如果右16位出现要找的位置就不移位，sum加0，然后8位，4位，2位，1位同理。

优化前（37个运算符）

```
int sum = 1;
 int sign = x >> 31;
 int x1 = x >> 16;
 int step1 = !!(sign ^ x1);
 step1 = step1 << 4;//step等于0或16
 x = x >> step1;
 //sum = sum + step1;

 int x2 = x >> 8;
 int step2 = !!(sign ^ x2);
 step2 = step2 << 3;
 x = x >> step2;
 //sum = sum + step2;

 int x3 = x >> 4;
 int step3 = !!(sign ^ x3);
 step3 = step3 << 2;
 x = x >> step3;
 //sum = sum + step3;

 int x4 = x >> 2;
 int step4 = !!(sign ^ x4);
 step4 = step4 << 1;
 x = x >> step4;
 //sum = sum + step4;

 int x5 = x >> 1;
 int step5 = !!(sign ^ x5);
 x = x >> step5;
 //sum = sum + step5;

 sum = sum + step1 + step2 + step3 + step4 + step5 + (x^sign) ;
 return sum;
```

优化后（29个运算符）

最高有效位和它的右边一位一定是相反的，根据这一特点，设y = (x << 1)^x， 这样y的最高的1就是x的最高有效位（0除外）

```
int y = (x << 1)^x;
```

```
  int step1 = !!(y >> 16);
  step1 <<= 4;
  y >>= step1;

 int step2 = !!(y >> 8);
 step2 <<= 3;
 y >>= step2;

 int step3 = !!(y >> 4);
 step3 <<= 2;
 y >>= step3;

 int step4 = !!(y >> 2);
 step4 <<= 1;
 y >>= step4;

 int step5 = (y >> 1) & 1  ;
 return step1 + step2 + step3 + step4 + step5 + 1;
```

22.float_half

优化前（19个运算符）

先判断是不是NaN或者无穷大，然后判断如果×0.5前后都是规格化数，就返回阶码位减一的值；然后计算尾数是否需要进位，如果前后都是非规格化数，返回尾数加符号位；如果阶码为1，×0.5后可能会变为非规格化数，也可能还是规格化数（尾数为全1），统一采用尾数加0x400000(不必分类讨论)，然后加上符号位。

优化1.0（17个运算符）

省掉了判断前后都是非规格化数的if分支，将这种情况和阶码为1的情况合并。

优化2.0（13个运算符）

思路和1.0一样，省掉了一些不必要的语句，比如截取尾数和符号位的语句。

```
unsigned tmp = uf & 0x7fffffff;
  //如果是NaN
  if(tmp >= 0x7f800000) return uf;
  int exp = uf & 0x7f800000;
  int low2 = uf & 0x3;//末尾2位

  if(exp > 0x800000)//如果前后都是规格化数
  {
    return uf - 0x800000;
  }

  int temp = uf;
  temp = temp >> 1;//右移1位后的尾数
  if(low2 == 3) temp += 1;//是否需要进位
  temp = temp & 0xbfffffff;
  return temp;
```