Data Lab: Manipulating Bits

电02 肖锦松 2020010563

邮箱：xiaojs20@mails.tsinghua.edu.cn

Explanation for the function of my code of each problem.

1 Bit Manipulations

1.1 bitAnd(x,y)

> Description: x & y using only | and ~

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

用非和或运算实现与运算，只需要利用摩根定律A & B=~ (~ A | ~ B) 即可，直接返回~(~x|~y)

1.2 getByte(x,n)

> Description: Get byte n from x.

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

先将x右移n byte，也就是n<<3 bit，然后这时候取最后1 byte，也就是8 bit。只需要最后 8 bit 按位与上1就行，即 `(x>>(n<<3))&0xFF`

1.3 logicalShift(x,n)

> Description: Shift right logical.

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

C语言默认的右移是算数右移，它与逻辑右移的区别在于右移后高位用符号位填充，而逻辑右移则默认用0填充，因此本题应该让最高位的n bit数置为0，而低32-n bit 数不变。怎么让符号位1和0都置为0，可以采用按位与零运算，1&0=0, 0&0=0。因此只需要让最高 n bit 与 n bit的0做按位与运算即可。

构造一个高 n bit 为0，低 32-n bit 为1的数字，然后再与 `x>>n` 做按位与就可以，这样可以保证高 n bit 变为0，低 32-n bit 不变。同理，按位异或运算也可以实现同样功能，只需要让高 n bit 为符号位值，低 32-n bit 为0，但是op 次数应该会更多。

1.4 bitCount(x)

> Description: Count the number of 1's in x.
>
> Reference: 本题没有想法，因此参考网上思路：计算1的个数就相当于计算x的二进制串的所有位的和。首先将int型数据x的32位分成16组，并进行X31+X30，X29+X28，...，X3+X2，X1+X0的运算；然后将x分成8组，并进行X31+X30+X29+X28，...，X3+X2+X1+X0的运算。依次类推，接着将x分成4组，2组并进行相应的运算。最后只剩下1组，此时将所有的位进行相加即得到了最终结果。

```
int bitCount(int x) {
  int by_2, by_4, by_8, by_16, by_32;
  int add;
  by_2 = 0x55 | (0x55<<8); //0x5555
  by_2 = by_2 | (by_2<<16); //0x55555555
  by_4 = 0x33 | (0x33<<8);
  by_4 = by_4 | (by_4<<16);
  by_8 = 0x0F | (0x0F<<8);
  by_8 = by_8 | (by_8<<16);
  by_16 = 0xFF | (0xFF<<16);
  by_32 = 0xFF | (0xFF<<8);
  add = (x&by_2) + ((x>>1)&by_2);
  add = (add&by_4) + ((add>>2)&by_4);
  add = (add&by_8) + ((add>>4)&by_8);
  add = (add&by_16) + ((add>>8)&by_16);
  add = (add&by_32) + ((add>>16)&by_32);
  return add;
}
```

正常来说可以利用循环右移，在每次右移中将`x`和`0x01`进行按位与，然后让`count++`。不能使用循环的情况下不能采用手动的32次移位与按位与，因而考虑"二分法"的思路。由于`x`中1的个数等价于二进制所有位值总和，将32 bit 分为16组，两两相加，再次两两相加...直到最后两组相加。其中，每2位的相加方法为，x[1]+x[0]=x>>1[0]+x[0]，每4位的相加方法x[3:2]+x[1:0]=x>>2[1:0]+x[1:0]...按照这个思路计算，每次算完都得存到一个新int里，直到x[31:16]+x[15:0]=x>>16[15:0]+x[15:0]。该结果即为`x`中1的个数。

这道题目一开始做的时候直接对by_2、by_4...直接赋值，但是最后一直出现报错，最后在代码规范里发现这是被禁止的，因此得利用0xXX进行运算构造出相应的by_2等。

> Each "Expr" is an expression using ONLY the following:
>
> ```
> 1. Integer constants 0 through 255 (0xFF), inclusive. **You are not allowed to use big constants such as 0xffffffff.**
> 2. Function arguments and local variables (no global variables).
> 3. Unary integer operations ! ~
> 4. Binary integer operations & ^ | + << >>
> ```

第二个错误是关于`int add`的定义，一开始定义的位置在add运算前一行，但这会报错`undeclared variable add`，将add的声明改到函数开头即可。**这是C语言的规定！**

1.5 bang(x)

> Description: Compute !n without using ! operator.

```
int bang(int x) {
  int signOR = ((x | (~x+1)) >> 31) & 0x01;
  return ~signOR & 0x01;
}
```

考虑所有非0的数，要么最高位为1，要么其相反数最高位为1，而对于0及其相反数，二者最高位都为1，因此可以利用`x`及其相反数的最高位（也就是符号位）进行按位或得到`signOR`，此时`x`为零`signOR`为0，`x`非零`signOR`为1，所以还得对其进行取反，取最低位。

2 Two’s Complement Arithmetic

2.1 tmin()

> Description: Most negative two’s complement integer

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

最小的有符号数补码，由于最高位权重为-2^k，为负，其余位权重$2^{k-1}, 2^{k-2}...$，为正，因此最小的有符号数补码为100...000，对于32 bit数来说就是`1<<31`。

2.2 fitsBits(x,n)

> Description: Does x fit in n bits?

```
int fitsBits(int x, int n) {
  int minusOne = ~1+1;
  int highBitNotSame = (x>>(n+minusOne)) ^ (x>>31);
  return !highBitNotSame;
}
```

判断`x`能否用`n`位补码表示。`n`位补码能表示的数字范围是$-2^{n-1} \sim (2^{n-1}-1)$，只需要判断`x`的范围是否在其中即可。如果`x`是非负数，直接判断是否在$0 \sim 2^{n-1}-1$，如果`x`是负数，先取其相反数（即取反加一），再判断是否在$1 \sim 2^{n-1}$。

进一步思考，如果`x`的高 32-n+1 bit均为1或0，则可以截取低 n bit作为`x`的“新补码”。因此，可以通过检测`x`的高 32-n+1 bit 是否相同，考虑将`x`右移 n-1 bit，问题就可以转化为32位是否全部相等。此时可以取其中最高位右移31位（其实选取32 bit 均为哪一位都行，只是考虑到选取最高位右移所需的操作数最少），然后进行位异或运算，如果为零则返回可以将`x`表示为 n bit 补码。

遇到的问题：首先是右移 n-1 bit，-1的实现可以借助补码运算$-1 = \sim 1 + 1$，原先我用的是先右移 n bit，再左移 1 bit，但这样会让最后一位始终为0，是错的；另一个遇到的问题是判断`highBitNotSame`是否为零，最后发现本题可以采用 ! 运算。

2.3 divpwr2(x,n)

> Description: Compute x/2^n

```c
int divpwr2(int x, int n) {
  int minusOne = ~1+1;
  int offset = (0x01<<n)+minusOne;
  int sign = x>>31;
  offset &= sign;
  return (x+offset)>>n;
}
```

C语言的右移相当于是向下舍入，而本题要求向零舍入，因此需要给初始数值加上一个偏移量。

如果`x`为非负，向下舍入就是向零摄入，直接右移 n bit 即可。如果`x`为负，必须通过加偏移量使向下舍入变为向零舍入，即$2^n - 1$。

将二者合一，偏移量可以用表达式`offset & sign`表示。之后让`x`加上偏移量再右移即可。

向下舍入	向零舍入	加入偏移量15
-48/16=-3	-48/16=-3	-33/16=-3
-33/16=-3	-33/16=-2	-18/16=-2
-32/16=-2	-32/16=-2	-17/16=-2
-31/16=-2	-31/16=-1	-16/16=-1
-17/16=-2	-17/16=-1	-2/16=-1
-16/16=-1	-16/16=-1	-1/16=-1
-15/16=-1	-15/16=0	0/16=0
-1/16=-1	-1/16=0	14/16=0

2.4 negate(x)

> Description: -x without negation

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

利用相反数的补码，补码运算规则为取反加一，直接返回即可。

2.5 isPositive(x)

> Description: x > 0?

```c
int isPositive(int x) {
  int notNeg = !(x>>31);
  int isZero = !x;
  return notNeg ^ isZero;
}
```

通过符号位是否为零可以区分非负和负，用`notNeg`表示，在非负的基础上利用！运算可以区分`x`是否为0。因此`x>0`需要满足符号位为0，且`!x==0`。这里为了方便可以采用`!(x>>31) & !!x`，但是使用了两次`!`，采用**异或**可以减少一次操作，因为不可能存在不是非负和是零同时满足的情况。

2.6 isLessOrEqual(x,y)

> Description: x <= y?

```
int isLessOrEqual(int x, int y) {
  int z = y + (~x+1);
  int sign = !(z>>31); // sign should be 1, when z >= 0.
  int xNeg = (x>>31)& 0x01;
  int yNeg = (y>>31)& 0x01;
  int notSame = xNeg ^ yNeg;
  sign = (~notSame & sign) | (notSame & xNeg);
  return sign;
}
```

可以先等价为判断`y-x>=0`，但是考虑到负数减正数或者正数减负数的**溢出**，需要对符号不同的情况进行特殊处理！如果两个参数异号，那么当`x`为负，`y`为正时返回1。

2.7 ilog2(x)

> Description: Compute ⌊log 2(x)⌋

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

由于`floor`向下取整，因此只需要找到`x`二进制数中的第一个1的位置即可。因此可以先检测高16 bit是否有1，若没有则检测低 16 bit是否有1。如果高位有1，则记录此时的移位情况`shift  += 16`，然后再检测其中的高 8 bit 和 低 8 bit，以此类推。最后移位次数`shift`即为答案。

3 Floating-Point Operations

3.1 float_neg(uf)

> Description: Compute -f

```c
unsigned float_neg(unsigned uf) {
  int expMask = 0x7F800000;
  int fracMask = 0xFF800000;
  int expAllOne = uf & expMask;
  int fracZero = uf | fracMask;
  if(expAllOne == expMask && !(fracZero == fracMask))//NaN
    return uf;
  else
    return uf^0x80000000;
}
```

根据浮点数的编码规则`s exp frac`，决定浮点数正负的就只是`s`。那么本题的思路很简单，就是判断输入的参数`unsigned uf`是一个合规的浮点数或者是NaN，如果是合规的浮点数直接改变`s`（和1异或），如果是NaN直接返回原输入参数。

合规的浮点数有两种情况，其实可以避开它去用更少的op去判断NaN。当阶码（23到30bit）全为1，小数域（0到22bit）为非零值时该浮点数为NaN。这里利用了1按位与1为1，任何数按位与0为0来判断阶码是否全为1，利用任何数按位或1都为1，0按位或0为0来判断小数域是否不为0。

3.2 float_i2f(x)

> Description: Compute (float) x

```c
unsigned float_i2f(int x) {
  int tempx=x, absx=x, discarded=0;
  int sign=0;
  int exp=0, frac=0;
  int sigOne=1<<31;
  if(x==0)
    return 0;
  else if(x==(1<<31))
    return 0xCF000000;
  else if(x<0){
    absx=-x;
    tempx=-x;
    sign=sigOne;
  }
  while((tempx&sigOne)==0){
    exp+=1;
    tempx<<=1;
  }
  exp=31-exp;;
  frac=absx-(1<<exp);
  if(exp<=23){
    frac=frac<<(23-exp);
  }
  else{
    frac=frac>>(exp-23);
    discarded=tempx&0xFF;
    if(discarded>0x80 || ((discarded==0x80) && (frac&1))){
      frac+=1;
    }
  }
  exp=(exp+127)<<23;
  return sign+exp+frac;
}
```

由于输入是`int`，浮点数形式只会是规格化。由于整数和浮点数表示负数的方法不同，浮点数是利用1位符号位来判断正负，然后加上其绝对值，而整数是利用补码表示。因此要把整数转化成浮点数需要先判断整数的正负，然后再求其绝对值。浮点数的表示形式为$(-1)^s M2^E$，因此需要先找出阶数E，也就是整数绝对值最高位1所在的位置，因为对应$1 \cdot 2^E$。然后在该位后面的数可以看成小数部分，但是小数部分是以阶数后一位开始的，然而浮点数规定小数部分只有23位，因此需要对其进行移位处理，只保留小数部分的高23位。如果小数部分少于23位，需要左移填充到23位，如果小数部分多于23位，需要右移减少到23位。最后需要注意的就是小数部分的四舍五入，因为如果小数部分多于23位，如果需要舍弃掉的尾数的首位是1，那么是需要向小数部分进位的。

原先采用的取**被舍弃部分**的方法是利用原先的小数部分减去最后得到的小数部分，需要多次移位操作，导致最后操作数超过了40。因此重新思考了一下如何更好地得到被舍弃的小数部分。把`x`左移直到最高位为1，此时最高位的位置就是阶数，那么也就是说此时`x`的8到30正是小数部分，而0到7就是前面用了很多次移位才得到的舍弃部分！最后还有一个小细节，就是向偶数进位，需要在条件判断的时候多注意。

由于没有足够的时间思考，最后这道题还是卡着op=30过了。

3.3 float_twice(uf)

> Description: Computer 2*f

```c
unsigned float_twice(unsigned uf) {
  int sign = uf&0x80000000;
  int exp = uf&0x7F800000;
  int frac = uf&0x007FFFFF;
  int expAllZero = (exp == 0);
  int expAllOne = (exp == 0x7F800000);
  int twiceExpAllOne = (exp == 0x7F000000);
  if(expAllOne) ///Nan or Inf
    return uf;
  if(!expAllZero){ //normalized
    if(twiceExpAllOne)
      return sign+0x7F800000;
    else
      return uf+0x00800000;
  }
  //denormalized
  frac<<=1;
  return sign+exp+frac;
}
```

这时候的输入也是浮点数，因此得分不同情况讨论。

1. Normalized. 指数部分不为全0和全1：将指数加1即可，但如果此时指数加1后全为1则输出inf。
2. Denormalized. 指数部分全为0：将小数部分左移1位即可，这样即使原先小数的最高位为1，也能够直接让指数加1。
3. NaN or Inf. 直接将输入参数输出。

Summary And Reflections

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	7	bang
1	1	0	2	1	tmin
2	2	0	2	7	fitsBits
2	2	0	2	8	divpwr2
2	2	0	2	2	negate
3	3	0	2	4	isPositive
3	3	0	2	14	isLessOrEqual
4	4	0	2	27	ilog2
2	2	0	2	7	float_neg
4	4	0	2	30	float_i2f
4	4	0	2	12	float_twice

Score = 71/71 [41/41 Corr + 30/30 Perf] (168 total operators)

168 total operators

- 这个作业有些题目很难一下子想到解决方法，主要是因为题目对操作进行了限制。
- 将一次大作业分成多个小题的形式还是比较适合“分次”地完成作业，不至于隔天来做大作业还得熟悉之前写的部分。
- 这次大作业让我对整数和浮点数的知识点更加印象深刻，总体来说得到了较大锻炼，同时也发现我对浮点数的掌握并不到位。
- 思考过程有时候可以用8位或4位来思考，然后再将做法拓展到32位。

在做受限制的位级运算时，可以巧妙地利用别的运算代替，比如减法可以用加补码表示，没有了循环可以利用”分而治之“的思想进行求解，也可以减少次数到$log(N)$。

关于如何减少op次数的总结：

1. 有时候异或运算是减少op次数的一个重要工具，能够减少一些判断次数。
2. 利用变量初始化默认为0，如果要对某变量赋值0，有时候可以省略。
3. 有时候32位数据之间存在奇妙的关系，如果发现可能可以减少很多不必要的操作。