这个lab针对的是CSAPP第二章的内容，要想完成此lab必须对计算机信息的表示和处理非常熟悉，题目有严格的要求，有些题目难度很大，与其说是考编程不如说是考数学。

1、bitAnd

/ \* bitAnd - x&y using only ~ and |

\* Example: bitAnd(6, 5) = 4

\* Legal ops: ~ |

\* Max ops: 8

\* Rating: 1

\*/

int bitAnd(int x, int y) {

return ~(~x|~y);

}

这一题比较容易，原理是“德摩根定律”。

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

\*/

int getByte(int x, int n)

{

return (x >> (n << 3)) & 0xff;

}

从32位的4个字节中取出第n个字节的单字节，一个字节8位，所以8\*n就是要移动的次数，用n<<3移位代替，与0xff取&得到一个字节。

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

\*/

int logicalShift(int x, int n) {

return (x>>n)&(~(((1<<n)+~1+1)<<(32+~n+1))); //(x>>n)&(~(((1<<n)-1)<<(32-n)));

}

系统默认是算数右移，也就是扩展符号位。这里要求逻辑右移，最高位补零，所以思路就是想办法把扩展出来的位跟0取&，其他位跟1与。我的方法是用得到低n位为1的数，再把它移到最高位，取反后与目标值求&即可。这里规定不能用减法，所以-1可以用~0或者~1+1表示，-n也是用同样的办法~n+1，就是取反加一。这里有个可能的潜在问题是当n为0时，要移动32位，这超出了32位系统的范围，c语言没有明确规定超出后的行为，虽然在我的电脑上结果正常，但是可能在别的机器会出问题。下面的代码用了另一个思路，可以避免移位超限的问题。

int logicalShift(int x, int n) {

return (x>>n)&(((0x1<<(32+~n))+~0)|(0x1<<(32+~n)));

}

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

\*/

int bitCount(int x) {

int msk1=0x55|(0x55<<8);

int msk2=0x33|(0x33<<8);

int msk3=0x0f|(0x0f<<8);

int msk4=0xff|(0xff<<16);

int n;

msk1|=msk1<<16;

msk2|=msk2<<16;

msk3|=msk3<<16;

n=(x&msk1)+((x>>1)&msk1);

n=(n&msk2)+((n>>2)&msk2);

n=(n&msk3)+((n>>4)&msk3);

n=(n&msk4)+((n>>8)&msk4);

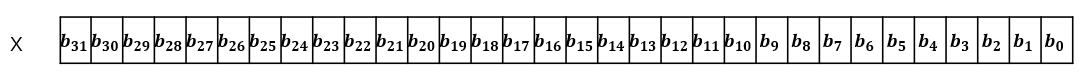
n=(n&((1<<16)+~0))+(n>>16);

return n;

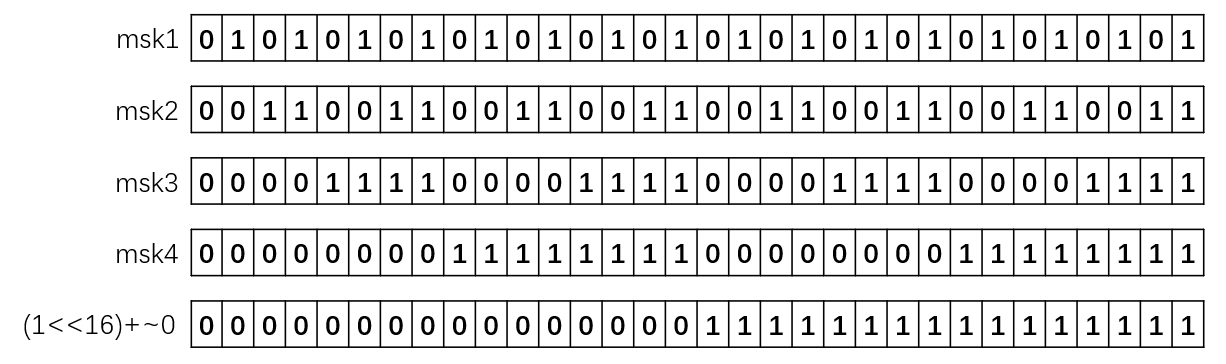
}

这个是我觉得最难的一道题目了，统计x中1的个数，难就难在不能使用循环，而且限制最大操作数40，所以必须找其他思路。下面是整个流程的说明和示意图。

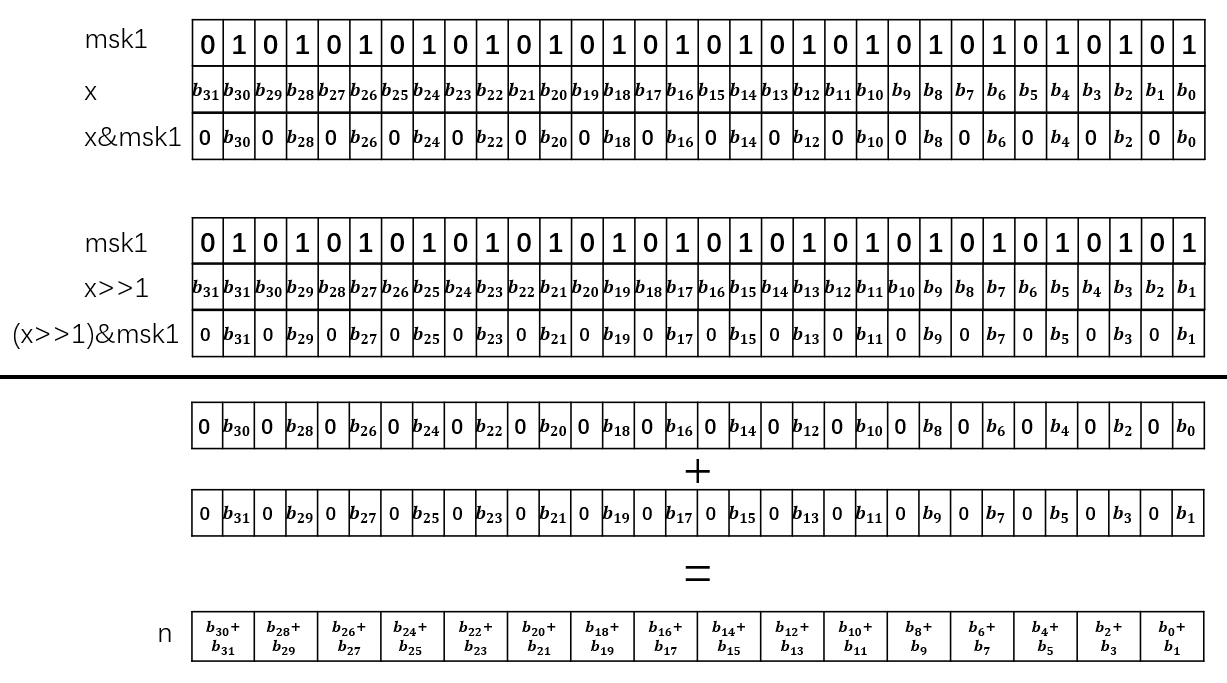
输入x如下所示：



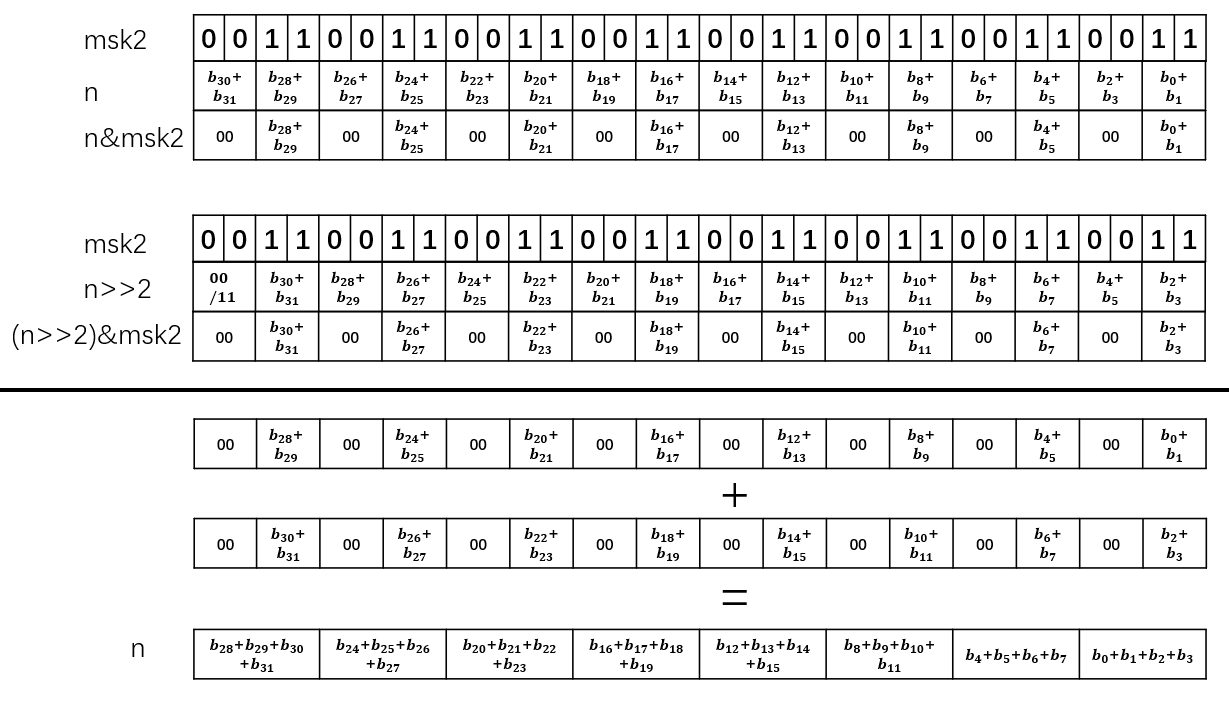
先构造辅助位，如下所示：



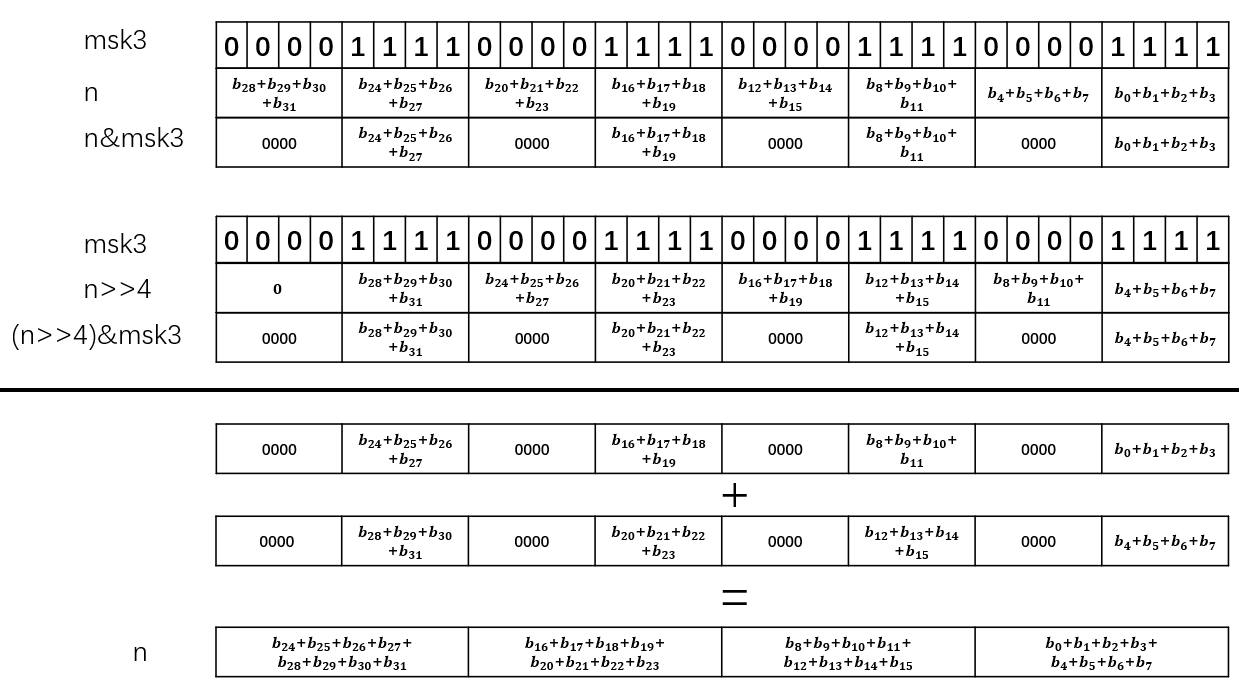
具体办法是先将相邻位两两相加，结果就是这两位中1的个数，保存在它俩对应的两个位中，



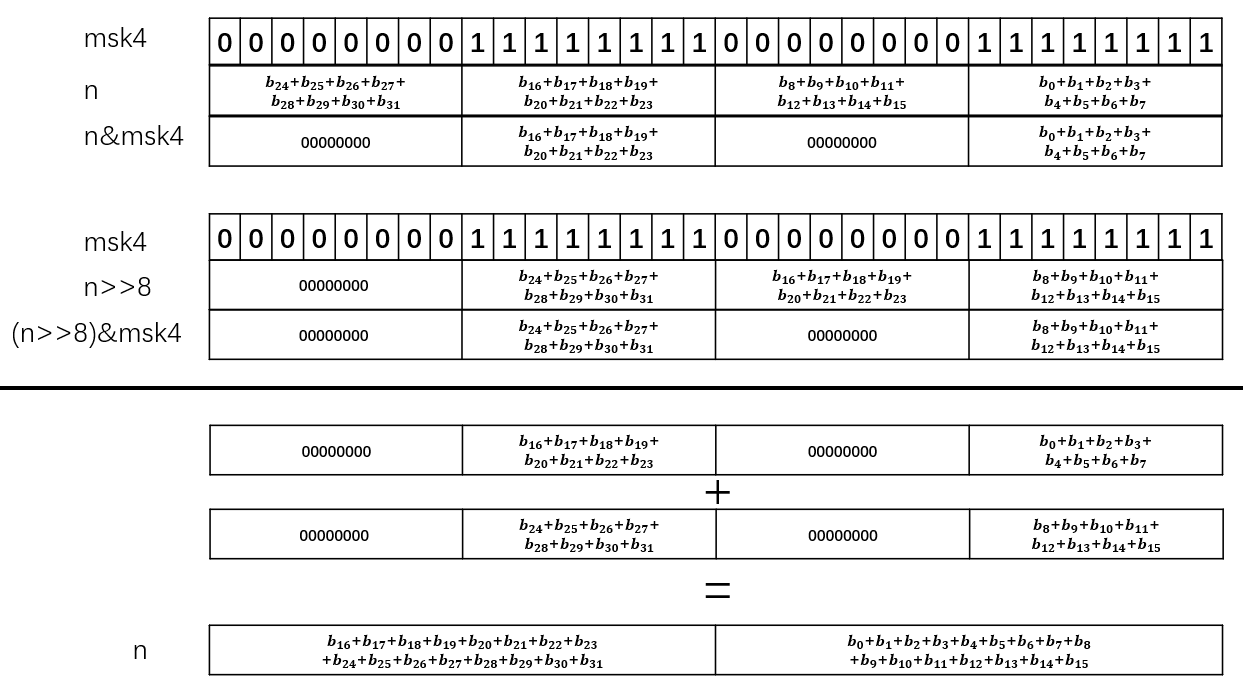
然后再相邻的2位两两相加，保存在他们对应的4个位中，



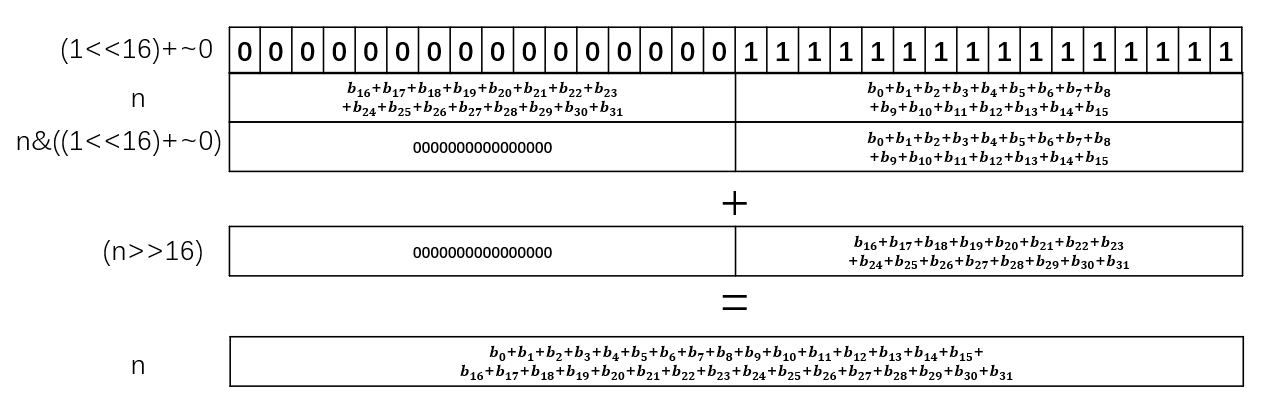
之后是相邻的4位两两相加，保存在他们对应的8个位中，



再然后相邻的8位两两相加，保存在他们对应的16位中，



最后高16位和低16位相加，得到最终的结果。



此题值得好好研究，相信还有其他解法，欢迎大家交流讨论。

5、bang

/\*

\* bang - Compute !x without using !

\* Examples: bang(3) = 0, bang(0) = 1

\* Legal ops: ~ & ^ | + << >>

\* Max ops: 12

\* Rating: 4

\*/

int bang(int x) {

int a = (x>>16)|x;

a = (a>>8)|a;

a = (a>>4)|a;

a = (a>>2)|a;

return (((a>>1)|a)&0x1)^1;

}

此题不用！取非，原理就是判断x中是否有1，“或 |” 可以将1保留下来，用二分法5步或下来结果就跑到最低位了，用1取&就知道是0还是1了，再与1异或就取！了。

6、tmin

/\*

\* tmin - return minimum two's complement integer

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 4

\* Rating: 1

\*/

int tmin(void) {

return 1<<31;

}

二进制补码的最小值，就是符号位1，其他位0的值,0x80000000

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

\*/

int fitsBits(int x, int n) {

return (x>>31)&(x>>(n+~0))&1;

}

这题有点绕，不过理解二进制补码最高位是符号位就不难了。我的思路是将当前32位的符号位与n位的符号位进行比较，如果相同就能表示。这题测试不过，我试过网上的其他答案也过不了，看来是测试本身的问题。这题还有其他思路，欢迎大家交流讨论。

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

\*/

int divpwr2(int x, int n) {

return (x+(((1<<n)+(~1+1))&(x>>31)))>>n; //~1+1 = ~0

}

除以2的n次就是向右移动n位，但是要求向0舍入，负数需要加一个偏置（1<<n）-1。（x>>31）根据符号位得到全0或者全1的值。

9、negate

/\*

\* negate - return -x

\* Example: negate(1) = -1.

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 5

\* Rating: 2

\*/

int negate(int x) {

return ~x+1;

}

取反加一

10、isPositive

/\*

\* isPositive - return 1 if x > 0, return 0 otherwise

\* Example: isPositive(-1) = 0.

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 8

\* Rating: 3

\*/

int isPositive(int x) {

return (((x>>31)&1)^1)&(!(!x));

}

此题主要是对0的处理，(((x>>31)&1)^1)判断符号位，(!(!x))判断是否为0。

11、isLessOrEqual

/\*

\* 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) {

return (((x^y)>>31)&(!(((x^y)&y)>>31)))+((!((x^y)>>31))&(!((y+(~x+1))>>31)));

}

此题解法也很多，思路都是异号直接通过符号位判断大小，同号就相减判断大小。(x^y)>>31同号为0，计算+后边的表达式，异号为1，计算+前面的表达式。

12、ilog2

/\*

\* 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 n=0;

n=(!(x>>1));

n+=(!(x>>2));

n+=(!(x>>3));

n+=(!(x>>4));

n+=(!(x>>5));

n+=(!(x>>6));

n+=(!(x>>7));

n+=(!(x>>8));

n+=(!(x>>9));

n+=(!(x>>10));

n+=(!(x>>11));

n+=(!(x>>12));

n+=(!(x>>13));

n+=(!(x>>14));

n+=(!(x>>15));

n+=(!(x>>16));

n+=(!(x>>17));

n+=(!(x>>18));

n+=(!(x>>19));

n+=(!(x>>20));

n+=(!(x>>21));

n+=(!(x>>22));

n+=(!(x>>23));

n+=(!(x>>24));

n+=(!(x>>25));

n+=(!(x>>26));

n+=(!(x>>27));

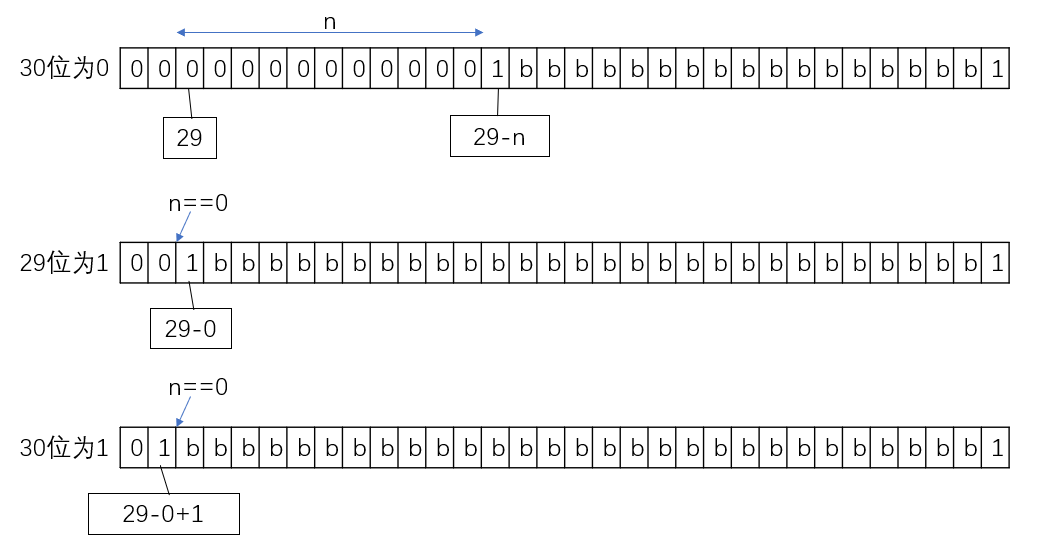
n+=(!(x>>28));

n+=(!(x>>29));

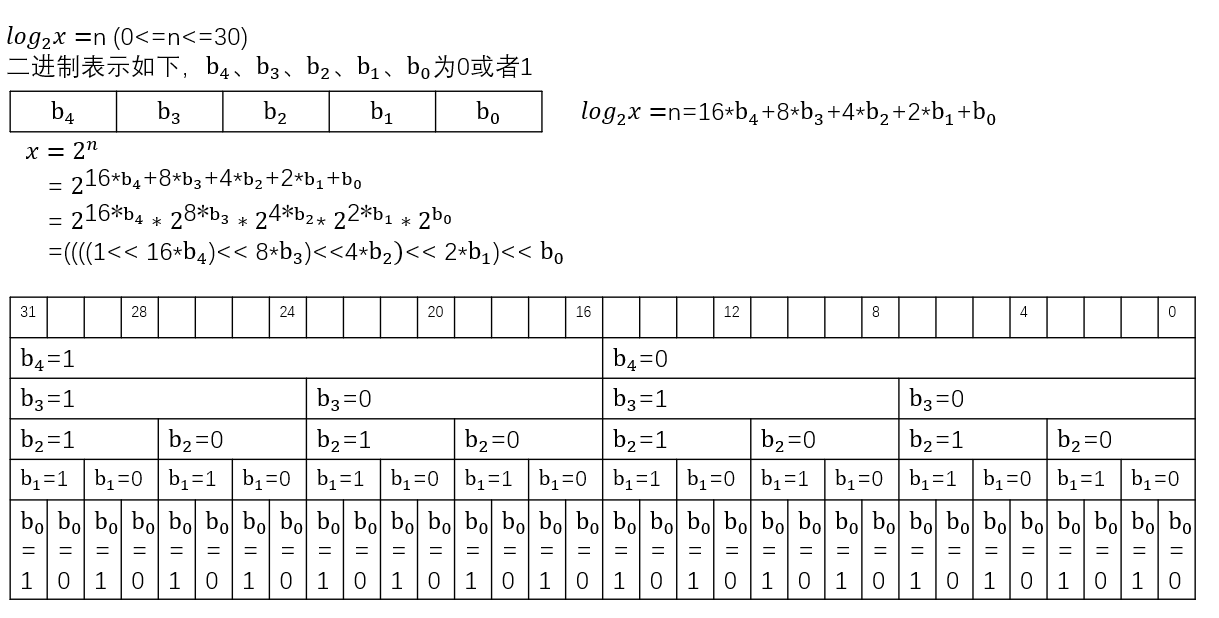
return 30+(x >> 30)+(~n); //29-n+(x>>30)

}

此题我用了笨办法，刚刚好卡在所允许的最大操作数上，log2的结果刚好是x中最高位1的位置，x>0，x符号位肯定是0，最低位肯定是1，然后右移1位判断是否是0，如果是0，就说明上一位是最高位1，之后的移位都是0，n记录了移位后结果为0的次数，也就是低29位高位0的个数，29-n就是前29位最高位1的位置，如果第30位为1，n必然为0，30就是最高为1的位，否则29-n就是最高为1的位。之所以把30位特殊处理是可以减少操作数，n+=(!(x>>30))多一个！。



此题还有更好的办法，如下所示：



如果b4=1,x>=(1<<16)，高16位为非0，否则b4=0高16位肯定为0，利用这点可以根据x>>16后判断是否为0得出b4的值，然后在上一步的基础上判断高8位是否为0得出b3，之后在上两步的基础上判断高4位是否为0得出b2，以此类推得出b1和b0。这种方法看着优雅多了。

int ilog2(int x) {

int ans=0;

ans=(!!(x>>(16)))<<4;

ans=ans+((!!(x>>(8+ans)))<<3);

ans=ans+((!!(x>>(4+ans)))<<2);

ans=ans+((!!(x>>(2+ans)))<<1);

ans=ans+((!!(x>>(1+ans)))<<0);

return ans;

}

受此方法启发，还可以用二分法找最高为1的位，效率应该比我第一个笨办法高不少，这里就不列代码了，大家自行尝试实验。

下面是对浮点数的题目，虽然考察的是浮点，但参数和返回值都是整型，只是二进制层面解释为浮点，所以要对浮点的二进制结构非常熟悉。

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

\*/

unsigned float\_neg(unsigned uf) {

unsigned t;

if((((uf>>23)&0xff)==0xff)&&(uf&0x7fffff))

{

t=uf;

}

else

{

t=uf^(1<<31);

}

return t;

}

浮点取负，对于NaN直接返回，其他情况只需要将最高位取反即可。

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

\*/

unsigned float\_i2f(int x) {

unsigned uf,s=0,e;

unsigned i,t;

if(x==0)

{

uf = 0;

}

else if(x==-x)

{

uf = 0xcf000000;

}

else

{

if(x<0)

{

uf=-x;

s=(1<<31);

}

else

uf=x;

i=30;

while(!(uf&(1<<i)))

{

i--;

}

uf=uf-(1<<i); //浮点数的1是隐含的，只保留小数部分，这里把高位1去掉

if(i<24)

{

uf=uf<<(23-i);

}

else

{

t=i-23;

uf+=(1<<(t-1)); //加即将丢弃的值的一半，四舍五入

if(uf&((1<<(t+1))-1)) //判断奇偶，如果是1后跟着0，说明不该进位。

{

uf-=1;

}

uf=uf>>t;

}

e=(127+i)<<23;

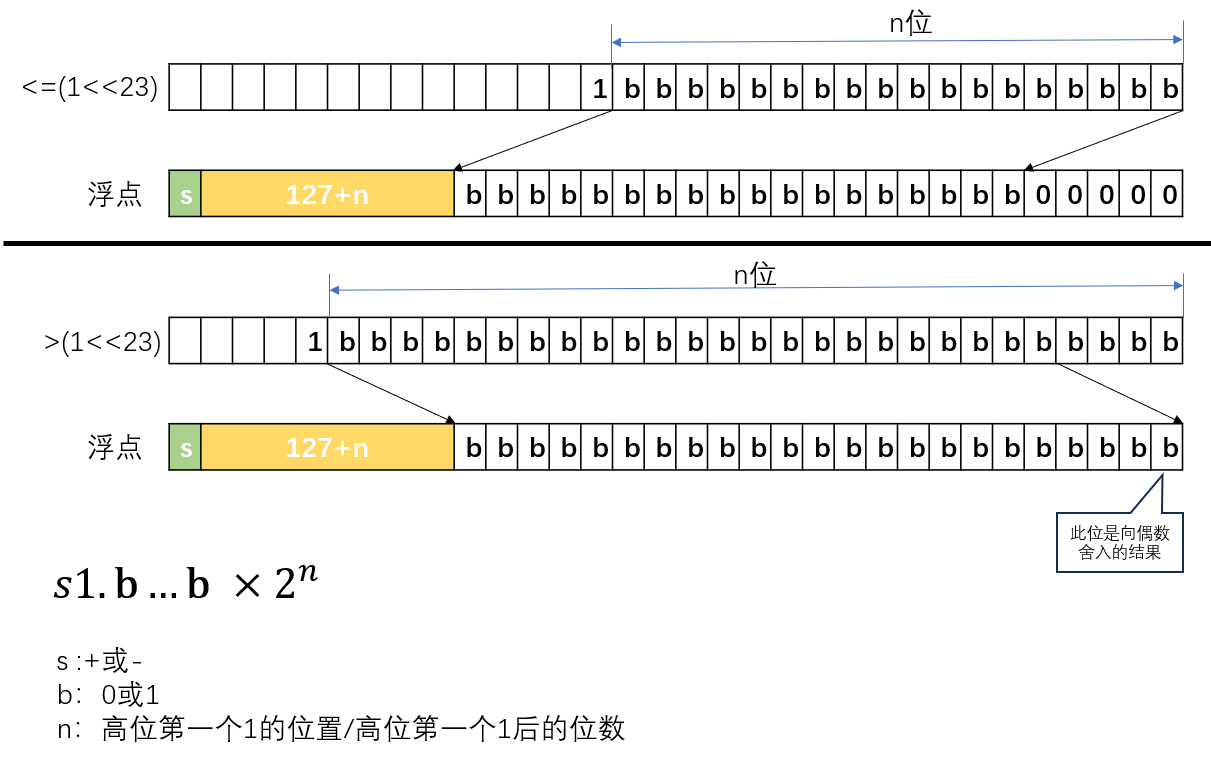
uf+=e|s;

}

return uf;

}

此题实现将整型数强转为浮点型，考察的是整型与浮点二进制表示之间的关系。0和tmin比较特殊，需要单独处理，负数需要转成正数，正数与浮点的关系如下图所示：

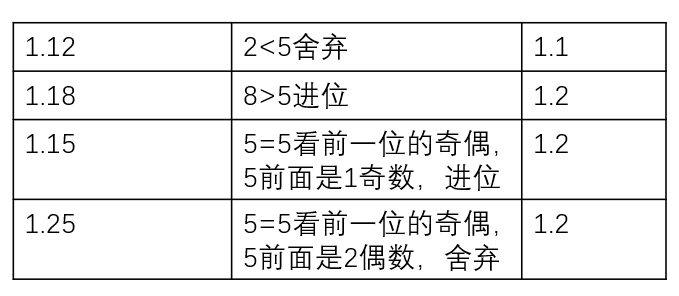


浮点使用科学计数法，所以我们只需要得到符号、指数部分和小数部分，把他们合起来就可以了。

符号位简单，整数最高位是啥浮点最高位就是啥；

指数部分需要找出高位第一个1的位置n，127+n就是指数部分的值，但是要将它左移23位，保存在符号位后8位中；

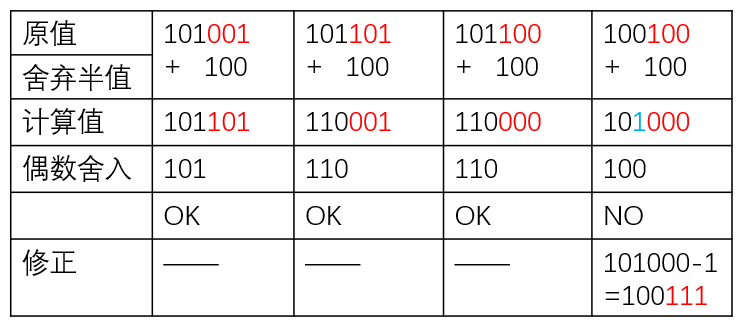
小数部分如图所示，通过移位操作移动到低23位中左对齐即可。对于左移低位补零，没有舍入问题，而右移由于将最低几位舍弃了，所以需要考虑舍入问题，通常使用向偶数舍入，它类似四舍五入，但对于中间值需要根据舍入后的最低有效位的奇偶进位（奇）或者舍弃（偶）。我们先举个10进制的例子：1.12、1.18、1.15、1.25舍入到小数点后1位，中间值是10/2=5



再看个二进制的例子1.0100、1.0101、1.0111、1.0110、1.0010，舍入到小数点后2位，中间值是4/2=2,二进制10。



理解了向偶数舍入就好办了。可以分别判断上面的几种情况进行处理，我觉得麻烦，所以我的思路是先加上舍弃位的半值，利用四舍五入，小于半值的加上半值也不会进位，移位后自然舍弃，大于半值的加上半值会产生进位，完全符合我们的需求。对于刚好半值，加上半值会进位，但是它比较特殊，加完后舍弃的值是全0，如果最低有效位是1，说明原来是0，不该进位，所以再减1去掉进位。



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

\*/

unsigned float\_twice(unsigned uf) {

unsigned t=uf,a; //Nan或无穷大

a = (uf>>23)&0xff; //get e

if(a==0) //非规格化

{

t=(uf&0x80000000)|(uf<<1);

}

else if(a!=0xff) //规格

{

t=uf+(1<<23); //指数+1

}

return t;

}

浮点数乘2，Nan和无穷大直接返回即可。对于非规格化的值，二进制层面就是小数位左移1位，溢出也没关系，刚好变成指数部分为1的规格化数，这是浮点数设计的妙处。对于规格化数，直接指数部分加1即可。

以上就是所有data lab的题目和解答，欢迎大家交流讨论。