

课程实验报告

课 程 名 称：计算机系统课程实验

实验项目名称：datalab

专 业 班 级：

姓 名：

学 号：

指 导 教 师：

完 成 时 间： 2016 年 5 月 5 日

信息科学与工程学院

|  |  |
| --- | --- |
| 实验题目：The CS:APP Data Lab | |
| 实验目的：Your goal is to modify your copy of bits.c so that it passes all the  tests in btest without violating any of the coding guidelines. | |
| 实验环境：PC | |
| 实验内容及操作步骤：  int bitAnd(int, int);  int test\_bitAnd(int, int);  int getByte(int, int);  int test\_getByte(int, int);  int logicalShift(int, int);  int test\_logicalShift(int, int);  int bitCount(int);  int test\_bitCount(int);  int bang(int);  int test\_bang(int);  int tmin();  int test\_tmin();  int fitsBits(int, int);  int test\_fitsBits(int, int);  int divpwr2(int, int);  int test\_divpwr2(int, int);  int negate(int);  int test\_negate(int);  int isPositive(int);  int test\_isPositive(int);  int isLessOrEqual(int, int);  int test\_isLessOrEqual(int, int);  int ilog2(int);  int test\_ilog2(int);  unsigned float\_neg(unsigned);  unsigned test\_float\_neg(unsigned);  unsigned float\_i2f(int);  unsigned test\_float\_i2f(int);  unsigned float\_twice(unsigned);  unsigned test\_float\_twice(unsigned);  实验结果及分析：  /\*  \* 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)）;//a&b=-(-a+-b)  }  用的摩根定律，把合取用析取表示。  /\*  \* 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))<<24>>24;//最后  }  通过移动，把需要的那八位移动到最右  /\*  \* 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) {  int y=32+(~n);  return (x>>n)&((1<<y)+(~0)+(1<<y));  }  这个很简单，也是通过移动来找出我需要剩下来的数是什么，但是需要注意的是这个会溢出，所以要加上(1**<<**y)来防止溢出。  /\*  \* 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 \_mask1=(0x55)|(0x55<<8);  int \_mask2=(0x33)|(0x33<<8);  int \_mask3=(0x0f)|(0x0f<<8);  int mask1=\_mask1|(\_mask1<<16);  int mask2=\_mask2|(\_mask2<<16);  int mask3=\_mask3|(\_mask3<<16);  int mask4=(0xff)|(0xff<<16);  int mask5=(0xff)|(0xff<<8);  int ans=(x&mask1)+((x>>1)&mask1);  ans=(ans&mask2)+((ans>>2)&mask2);  ans=(ans&mask3)+((ans>>4)&mask3);  ans=(ans&mask4)+((ans>>8)&mask4);  ans=(ans&mask5)+((ans>>16)&mask5);  return ans;  }  他的解题思路特别像二分排序，就是先把每两位中的1找出来，存在这个两位数里面，然后把每两位也就是四位中的一保存在这个四位数中，再八位……  在实验心得会有详细说明  /\*  \* bang - Compute !x without using !  \* Examples: bang(3) = 0, bang(0) = 1  \* Legal ops: ~ & ^ | + << >>  \* Max ops: 12  \* Rating: 4  \*/  int bang(int x) {  return ((~(x|(~x+1)))>>31)&1;  }  对0返回一，那就找0和其他数不同的地方就可以了，那么发现其他数字取反与原数字一定有一个最高位是一，也就是说一定有一个是负数，而0取反就是0.  /\*  \* tmin - return minimum two's complement integer  \* Legal ops: ! ~ & ^ | + << >>  \* Max ops: 4  \* Rating: 1  \*/  int tmin(void) {  return 1<<31;  }  找最小的数  /\*  \* 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>>(n+(~0)))+1)>>1);  }  x的补码——如果x是证书，那么他第n位肯定是0，相反，如果是负数，他的第n位是1，0只能存在在后面n-1个数里面。所以，通过右移，就会变成要么全是一，要么全是零，再+1，再右移，就全是零了。当然，要是不能用n位表示的话，那就会是1  /\*  \* 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+(((x>>31)&1)<<n)+(~n)+(!((x>>31)&1)))>>n;  }  因为有符号数，正数除法是向下取整，而负数除法是向上取整。但是移位运算都是向下取整。  所以需要给负数加入一个偏差(1<<n)-1，判断负数就是看最高位是否为1  /\*  \* 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  /\*  \* 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)+(((x+(~0))>>31)&1));  }  要是正数的话，-1然后最高位也要是0就可以，就是x和x-1最高位都是1  /\*  \* 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) {  int fuhaox=(x>>31)&1;  int fuhaoy=(y>>31)&1;  int zjf=(!fuhaoy)&fuhaox;  int putongjian=(((fuhaox+(~fuhaoy))>>31)&1)&(!fuhaoy)&fuhaox;  return zjf|putongjian;  }  要好几种可能  a+ b+  a+ b-   1. b+ 2. b-   当第三种情况的时候，一定是正数。所以只用把这种情况给合取上就可以了  /\*  \* 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) {  ans = ans + ((!!(x>>(16 + ans)))<<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;  }  log2(x)=16×a+8×b+4×c+2×d+1×e  那么就每一位每一位的判断就可以了  知道了a之后，右移（8+16×a）位，判断是否为0，得到b的值。  /\*  \* 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 tmp = uf&(0x7fffffff);  unsigned result = uf^(0x80000000);  if(tmp>0x7f800000) result=uf;  return result;  }  这个之前作业做过，判断nan就if(tmp>0x7f800000)  取反用到异或，和1000000000000异或，后面不影响，就是符号位相反  /\*  \* 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 result;  int signx = x&(1<<31);  int res = 31;  int ss=0,sss=0;  int ff=0;  int tmp;  if(x == 0) result=0;  else{  if(signx) x = (~x)+1;  while(!((1<<res)&x)) res=res-1;  x = x^(1<<res);  if(res<23) x=x<<(23-res);  else{  tmp = res-24;  if(tmp>=0) ss = (x>>tmp)&1,ff = ((1<<tmp)-1)&x;  x=(x>>(res-23));  }  x=x|((res+127)<<23);  if(ff==0) ss=(ss&x);  x=x+ss;  x=x|signx;  result = x;  }  return result;  }  另一种方式：  /\*  \* 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) {    int sign=x>>31&1;//获取符号位（1位）  int i;  int exponent; //指数域（8位）  int fraction; //小数域  int delta;//偏差（用于舍入）  int fraction\_mask;//小数域的掩码（23位）  if(x==0)//如果为0就直接返回  return x;  else if(x==0x80000000)//如果为TMin，解释为-（2^31），对应float就是（1）x1.0x2^31，所以S=1，M=1，E=31，指数域=31+127=158  exponent=158;  else{  if (sign)//通过前面的操作已经确定了符号，先把int的绝对值获取，之后利用浮点数的计算公式即可计算出float的值，现在要获取绝对值存入内存中  x = -x;    i = 30;//最高位是符号位，次高位是有效数字的起始位  while ( !(x >> i) )//从左往右查找有效数字第一个不为零的位，对应的位置就是最终的i（这里的位置从0开始标号）  i--;  //printf("%x %d\n",x,i);  exponent = i + 127;//数值的最高位已经找到是第i位（有效数据共有i+1位），又因为int类型不可能是非规格数据的范围（为0的情况在前面已排除），所以小数域就是d第i为后面的位向量（小数域一共有i个位），故阶码E=i（小数部分x2的E次方），指数域等于i+127；  x = x << (31 - i);//清除有效数据前面的所有0，包括符号位，得到有效数据开头的数据  fraction\_mask = 0x7fffff;//设置23位的小数域掩码  fraction = fraction\_mask & (x >> 8);//虽然按照浮点数格式，最前面的9个位不加入小数位，按道理应该右移9位，  //但是由于int类型的参数不可能是非规格数，所以最前面的一个有效数据也被舍弃（默认M=1+f），  //当向右移动8位，舍弃了有效数据的低8位，再和掩码处理以后，一共舍弃了9位  //除了低八位还包括有效数据的最高位，类比二进制小数中小数点左边的那一位数字，在float存储的时候，小数点左边数字不存入内存  //者9个位用来存储符号位+指数域  x = x & 0xff;//由于右移8位，舍弃了有效数据第八位，现在获取低八位用于舍入操作  delta = x > 128 || ((x == 128) && (fraction & 1));//如果低八位超过八位二进制能表示的无符号数的一半，  //要在小数域+1，普通的四舍五入思想  //如果低八位刚好等于八位二进制能表示的无符号数的一半，而且小数域目前最后一位是1，  //根据向偶数舍入的模式，也要在小数域+1，向上舍入的思想，  //如果低八位刚好等于八位二进制能表示的无符号数的一半，如果小数域目前最后一位是0，则向下舍入，不加1  //如果低八位刚好小于八位二进制能表示的无符号数的一半，直接丢弃，不加1，普通的四舍五入思想  fraction += delta;//进行舍入  if(fraction >> 23) {//如果舍入过后，小数域多余23位，则只取低23位，高位舍弃，但是阶码E要加1，所以指数域也就要加1  fraction &= fraction\_mask;  exponent += 1;  }  }  return (sign<<31)|(exponent<<23)|fraction;//符号位最高位（31），指数域（30--23），小数域（22-0）  }  /\*  \* 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 tmp = uf;  if((tmp&0x7f800000)==0){  tmp = (tmp&0x80000000)|((tmp&0x007fffff)<<1);  }  else if((tmp&0x7f800000)!=0x7f800000){  tmp+=(1<<23);  if((tmp&0x7f800000)==0x7f800000){  tmp=(tmp>>23)<<23;  }  }  return tmp;  }  首先，规格化数\*2是阶码+1，但是如果本来阶码就已经是可以达到的最大值了的话，那变成无穷大。  其次，非规格化数就是把尾数左移1位，前面的符号位不变。    收获与体会：  太难了！！！  不过通过对if的限制，我感觉对二进制的规律了解的更清晰了。就像是之前卡了我很久的第四题，研究了半天都是步数超过，后来发现居然可以分治的去思考。  第三题也是，始终弄不明白为什么会有问题，当移动0的时候就会有问题，显示0    ../../../../../Library/Containers/com.tencent.qq/Data/Library/Caches/Images/D4F0545286A1420951757BD7B3B9A4C2  更改之后就ok了  在做统计一的个数的时候，查找了资料  **Counting bits set, in parallel**  unsigned int v; // count bits set in this (32-bit value)  unsigned int c; // store the total here  static const int S[] = {1, 2, 4, 8, 16}; // Magic Binary Numbers  static const int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF};  c = v - ((v >> 1) & B[0]);  c = ((c >> S[1]) & B[1]) + (c & B[1]);  c = ((c >> S[2]) + c) & B[2];  c = ((c >> S[3]) + c) & B[3];  c = ((c >> S[4]) + c) & B[4];  The B array, expressed as binary, is:  B[0] = 0x55555555 = 01010101 01010101 01010101 01010101  B[1] = 0x33333333 = 00110011 00110011 00110011 00110011  B[2] = 0x0F0F0F0F = 00001111 00001111 00001111 00001111  B[3] = 0x00FF00FF = 00000000 11111111 00000000 11111111  B[4] = 0x0000FFFF = 00000000 00000000 11111111 11111111  We can adjust the method for larger integer sizes by continuing with the patterns for the *Binary Magic Numbers,* B and S. If there are k bits, then we need the arrays S and B to be ceil(lg(k)) elements long, and we must compute the same number of expressions for c as S or B are long. For a 32-bit v, 16 operations are used.  The best method for counting bits in a 32-bit integer v is the following:  v = v - ((v >> 1) & 0x55555555); // reuse input as temporary  v = (v & 0x33333333) + ((v >> 2) & 0x33333333); // temp  c = ((v + (v >> 4) & 0xF0F0F0F) \* 0x1010101) >> 24; // count  The best bit counting method takes only 12 operations, which is the same as the lookup-table method, but avoids the memory and potential cache misses of a table. It is a hybrid between the purely parallel method above and the earlier methods using multiplies (in the section on counting bits with 64-bit instructions), though it doesn't use 64-bit instructions. The counts of bits set in the bytes is done in parallel, and the sum total of the bits set in the bytes is computed by multiplying by 0x1010101 and shifting right 24 bits.  A generalization of the best bit counting method to integers of bit-widths upto 128 (parameterized by type T) is this:  v = v - ((v >> 1) & (T)~(T)0/3); // temp  v = (v & (T)~(T)0/15\*3) + ((v >> 2) & (T)~(T)0/15\*3); // temp  v = (v + (v >> 4)) & (T)~(T)0/255\*15; // temp  c = (T)(v \* ((T)~(T)0/255)) >> (sizeof(T) - 1) \* CHAR\_BIT; // count  ../../../../../Downloads/20170311222033378 | |
| 实  验成绩 |  |