**CSAPP**

**实 验 报 告**

**学生姓名** 吴语港

**学生学号** SA19225404

**实验日期** 2019/10/8

实 验 报 告

**一、实验名称：位操作**

**二、实验学时： 3**

1. **实验内容和目的：**

1.了解各种数据类型在计算机中的表示方法

2.掌握C语言数据类型的位级表示及操作

3.本实验总共包括有关位操作的15个编程题

4.你们的目标就是实现这15个编程题

5.所有要实现的代码都在bits.c里面

1. **实验原理：**

1. <<：左移就是: 丢弃最高位,0补最低位 （算术和逻辑都是这样）

2. >>：再说右移,明白了左移的道理,那么右移就比较好理解了.

右移的概念和左移相反,就是往右边挪动若干位,运算符是>>.

右移对符号位的处理和左移不同,对于有符号整数来说,比如int类型,右移会保持符号位不变,例如:

int i = 0x80000000;

i = i >> 1; //i的值不会变成0x40000000,而会变成0xc0000000

就是说,符号位向右移动后,正数的话补0,负数补1,也就是汇编语言中的算术右移.同样当移动的位数超过类型的长度时,会取余数,然后移动余数个位.

负数10100110 >>5(假设字长为8位)，则得到的是 11111101

总之,在C中,左移是逻辑/算术左移(两者完全相同),右移是算术右移,会保持符号位不变 .实际应用中可以根据情况用左/右移做快速的乘 /除运算,这样会比循环效率高很多.

3. !：逻辑非

4. ˜ ：位非

5. & ：位与

6. ˆ：异或

7. | ：位或

**实验步骤及结果：**

1. tar xvf datalab-handout.tar解压代码，包含如下文件

Bits.c:唯一需要修改的文件

Btest.c :该文件的作用是对我们实现的bits.c功能的正确性行评估，

README:关于btest的一些说明。

Dlc：语法检查

2. 打开bits.c文件，该文件包含了一个结构体team，我们需要首先补充该team中的数据

3. Bits.c中包含需要实现的15个函数，文件中规定了实现每个函数需要的逻辑和算术操作符（规定数量）。

只能使用规定的操作符! ˜ & ˆ | + << >>

不能使用循环或者条件语句

不能使用超过8位的常数（ff)

4. 完成后用./dlc bits.c检查bits.c的语法是否正确，就是是否按照要求使用规定数量的操作符

5. 如果语法检查无误，那么使用make btest，生成btest可执行文件，该文件检查bits.c中实现的函数功能是否与要求的一致，具体用法如下./btest

如果还需要修改bits.c那么需要make clean；make btest重新生成btest文件

./btest -f isPositive单独测试某一个函数

6.实验思路以及程序代码

//1

/\*

 \* bitXor - x^y using only ~ and &

 \*   Example: bitXor(4, 5) = 1

 \*   Legal ops: ~ &

 \*   Max ops: 14

 \*   Rating: 1

 \*/

/\*

 \* 思路：根据离散数学的逻辑公式

 \* x^y = ~(x&y)&~(~x&~y)

 \*/

int bitXor(int x, int y) {

  return ~((~(x&~y))&(~(~x&y)));

}

/\*

 \* tmin - return minimum two's complement integer

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 4

 \*   Rating: 1

 \*/

/\*

 \* 思路：0x8000 0000为补码表示的最小的负数 -128

 \* 将1左移31位就可以得到

 \*/

int tmin(void) {

  return 1<<31;

}

//2

/\*

 \* isTmax - returns 1 if x is the maximum, two's complement number,

 \*     and 0 otherwise

 \*   Legal ops: ! ~ & ^ | +

 \*   Max ops: 10

 \*   Rating: 1

 \*/

/\*

 \* 思路：补码表示的最大值为127，即Tmax是  0x7FFFFFFF

 \* 判断相等使用异或操作，相等则为全零

 \* Tmax满足Tmax == ~(tmax+1)

 \* 不过要用|位或来排除同样满足条件的0xFFFFFFFF

 \* 如果两边有一处不满足条件，那么就无法让位或运算为0，从而输出1

 \*/

int isTmax(int x) {

  return !((x^(~(x+1)))|(!(~x)));

}

/\*

 \* allOddBits - return 1 if all odd-numbered bits in word set to 1

 \*   where bits are numbered from 0 (least significant) to 31 (most significant)

 \*   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 12

 \*   Rating: 2

 \*/

/\* 判断二进制数奇数位是否全为1

 \* 思路：若一个二进制数偶数位为0，奇数位为1，则这个数为0xAAAAAAAA

 \* 先将x=x&0xAAAAAAAA，将这个数偶数位为变为0，

 \* 之后x^0xAAAAAAAA判断该数是否为0xAAAAAAAA

 \* 就可以完成判断了

 \*/

int allOddBits(int x) {

  int mask = 0xAA | 0xAA << 8;

  mask = mask | mask << 16;

  x = x & mask;

  return !(mask^x);

}

/\*

 \* negate - return -x

 \*   Example: negate(1) = -1.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 5

 \*   Rating: 2

 \*/

/\*

 \* 由于用补码表示，取反加一即可

 \*/

int negate(int x) {

  return (~x+1);;

}

//3

/\*

 \* isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')

 \*   Example: isAsciiDigit(0x35) = 1.

 \*            isAsciiDigit(0x3a) = 0.

 \*            isAsciiDigit(0x05) = 0.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 15

 \*   Rating: 3

 \*/

/\*

 \* x需要>=’0’且<=’9’，将x与临界点作差，然后判断符号位的为0还是1即可

 \* 48 - 57为ascii码数字部分，右边用58是因为0的符号位算正数，所以要多减一次

 \*/

int isAsciiDigit(int x) {

  return (!((x+~48+1)>>31))&!!((x+~58+1)>>31);

}

/\*

 \* conditional - same as x ? y : z

 \*   Example: conditional(2,4,5) = 4

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 16

 \*   Rating: 3

 \*/

/\*

 \* 首先使用t=!x，当x为0时返回1，当x不为0时，返回0，

 \* 根据题意得到( \_ &y)|( \_ &z)，首先空1，当x不为0，即t=0时，

 \* 需要t转换为0xffffffff（-1），当t=1时，需要将t转换为0x0（0），

 \* 将t-1即可，得到空1为“!x+~1+1”，同理空2为“~!x+1”

 \*/

int conditional(int x, int y, int z) {

  return ((!x+~1+1)&y)|((~!x+1)&z);

}

/\*

 \* isLessOrEqual - if x <= y  then return 1, else return 0

 \*   Example: isLessOrEqual(4,5) = 1.

 \*   Legal ops: ! ~ & ^ | + << >>

 \*   Max ops: 24

 \*   Rating: 3

 \*/

/\*

 \* 直接用y-x可能会超出int的表示范围，故而分类讨论：

 \* A：在x与y同号的情况下转换为p=y-x>=0，然后p符号位(p>>31）&1为0则返回1，符号位1则返回0；

 \* B：异号时，只要x>=0，就要返回0，否则返回1，由(x>>31)&1能达到该效果；

 \* C：c=a+b可作为x,y同号异号的判断

 \* p = y - x,q判断p的符号

 \*/

int isLessOrEqual(int x, int y) {

  int a=x>>31;

  int b=y>>31;

  int c=a+b;

  int p=y+(~x+1);

  int q=!((p>>31)&1);

  int r=(c&(a&1))|((~c)&q);

  return r;

}

//4

/\*

 \* logicalNeg - implement the ! operator, using all of

 \*              the legal operators except !

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

 \*   Legal ops: ~ & ^ | + << >>

 \*   Max ops: 12

 \*   Rating: 4

 \*/

/\*

 \* 令y=~x+1，考虑x与y的符号位：

 \* A：当x为0时，两者符号位都为0；

 \* B：当x=0x8000 0000时，两者符号位都为1；

 \* C：否则，两者符号位为01或10；

 \* D：根据离散数学的真值表得出(~x)&(~y).

 \*/

int logicalNeg(int x) {

  return ((~(~x+1)&~x)>>31)&1;

}

/\* howManyBits - return the minimum number of bits required to represent x in

 \*             two's complement

 \*  Examples: howManyBits(12) = 5

 \*            howManyBits(298) = 10

 \*            howManyBits(-5) = 4

 \*            howManyBits(0)  = 1

 \*            howManyBits(-1) = 1

 \*            howManyBits(0x80000000) = 32

 \*  Legal ops: ! ~ & ^ | + << >>

 \*  Max ops: 90

 \*  Rating: 4

 \*/

/\* 求表示补码的最少位数

 \* 思路：我们只需要异或相邻的数x^=(x<<1)，找出为1的最高位在哪一位就可以了

 \* 构造了一个函数g(x)去寻找0与1的分界线n，寻找分界线使用了二分法，不断缩小排查范围

 \*/

int howManyBits(int x) {

  int n = 0;

  x ^= (x<<1);

  n += ((!!(x&((~0)<<(n+16)))) << 4);

  n += ((!!(x&((~0)<<(n+8)))) << 3);

  n += ((!!(x&((~0)<<(n+4)))) << 2);

  n += ((!!(x&((~0)<<(n+2)))) << 1);

  n += (!!(x&((~0)<<(n+1))));

  return n+1;

}

//float

/\*

 \* floatScale2 - 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

 \*/

/\* 计算浮点数2\*f，flaot\_twice

 \* 思路：

 \* 1.若原数为非规格化小数或0时，处理小数部分

 \* if(exp\_ == 0) return (uf<<1)|s\_;

 \* 2.若为NaN或INF时

 \* if(exp\_ == 255) return uf;

 \* 直接返回。

 \* 3.若为其他情况，即指数加一

 \* ++exp\_,

 \* 4.若加了之后为INF时，保证其不为NaN，即小数部分全为0，

 \* if(exp\_ == 255) return 0x7f800000|s\_;

 \* 5.最后为一般情况，直接输出2\*f

 \* return (uf&0x807fffff)|(exp\_<<23);

 \*/

unsigned floatScale2(unsigned uf) {

  int exp\_ = (uf&0x7f800000)>>23;//阶码

  int s\_ = uf&0x80000000;//数符

  if(exp\_ == 0) return (uf<<1)|s\_;

  if(exp\_ == 255) return uf;

  ++exp\_;

  if(exp\_ == 255) return 0x7f800000|s\_;

  return (uf&0x807fffff)|(exp\_<<23);

}

/\*

 \* floatFloat2Int - Return bit-level equivalent of expression (int) f

 \*   for floating point argument f.

 \*   Argument is passed as unsigned int, but

 \*   it is to be interpreted as the bit-level representation of a

 \*   single-precision floating point value.

 \*   Anything out of range (including NaN and infinity) should return

 \*   0x80000000u.

 \*   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

 \*   Max ops: 30

 \*   Rating: 4

 \*/

/\*

 \* 将浮点数转换为有符号整数，float\_f2i

 \* 思路：

 \* 先将浮点数分成三段，

 \* 符号部分s\_ = uf>>31，

 \* 指数大小exp\_ = ((uf&0x7f800000)>>23)-127，

 \* 获取小数部分，并补上浮点数缺省的1，

 \* frac\_ = (uf&0x007fffff)|0x00800000。

 \* 处理特殊情况：

 \* 若全为0是返回0，

 \* 若指数大于31，整数无法表示溢出返回0x80000000。

 \* 若指数小于0，该数0<x<1返回0

 \* 若指数部分大于23则将小数部分向左移动frac\_ <<= (exp\_ - 23) ，exp\_代表指数大小。

 \* 若指数部分小于23则将小数部分向右移动frac\_ >>= (23 - exp\_) ，exp\_代表指数大小。

 \* 考虑最后符号，正数转换为负数不会产生溢出。

 \* 若frac\_为正数，则根据s\_调整正负输出即可。

 \* 若frac\_为负数，唯一正确情况为0x80000000。

 \*/

int floatFloat2Int(unsigned uf) {

  int s\_    = uf>>31;

  int exp\_  = ((uf&0x7f800000)>>23)-127;

  int frac\_ = (uf&0x007fffff)|0x00800000;

  if(!(uf&0x7fffffff)) return 0;

  if(exp\_ > 31) return 0x80000000;

  if(exp\_ < 0) return 0;

  if(exp\_ > 23) frac\_ <<= (exp\_-23);

  else frac\_ >>= (23-exp\_);

  if(!((frac\_>>31)^s\_)) return frac\_;

  else if(frac\_>>31) return 0x80000000;

  else return ~frac\_+1;

}

/\*

 \* floatPower2 - Return bit-level equivalent of the expression 2.0^x

 \*   (2.0 raised to the power x) for any 32-bit integer x.

 \*

 \*   The unsigned value that is returned should have the identical bit

 \*   representation as the single-precision floating-point number 2.0^x.

 \*   If the result is too small to be represented as a denorm, return

 \*   0. If too large, return +INF.

 \*

 \*   Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while

 \*   Max ops: 30

 \*   Rating: 4

 \*/

/\* 计算浮点数2.0^x

 \* 思路：

 \* 一般情况下，

 \* 把x当成浮点数的阶码就可以了

 \* 考虑特殊情况，

 \* 指数exp<-127和 exp>128，分别返回0和0x7f800000

 \*/

unsigned floatPower2(int x) {

  if(x<-127) return 0;

  if(x>128) return 0x7f800000;

  x += 127;

  x = x << 23;

  return x;

}

7.运行测试程序结果

