```
/*
 * CS:APP Data Lab
 *
 * <Please put your name and userid here>
 *
 * bits.c - Source file with your solutions to the Lab.
 *          This is the file you will hand in to your instructor.
 *
 * WARNING: Do not include the <stdio.h> header; it confuses the dlc
 * compiler. You can still use printf for debugging without including
 * <stdio.h>, although you might get a compiler warning. In general,
 * it's not good practice to ignore compiler warnings, but in this
 * case it's OK.
 */

#if 0
/*
 * Instructions to Students:
 *
 * STEP 1: Read the following instructions carefully.
 */

You will provide your solution to the Data Lab by
editing the collection of functions in this source file.

INTEGER CODING RULES:

  Replace the "return" statement in each function with one
  or more lines of C code that implements the function. Your code
  must conform to the following style:

  int Funct(arg1, arg2, ...) {
      /* brief description of how your implementation works */
      int var1 = Expr1;
      ...
      int varM = ExprM;

      varJ = ExprJ;
      ...
      varN = ExprN;
      return ExprR;
  }

  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 & ^ | + << >>

  Some of the problems restrict the set of allowed operators even further.
  Each "Expr" may consist of multiple operators. You are not restricted to
  one operator per line.

  You are expressly forbidden to:
  1. Use any control constructs such as if, do, while, for, switch, etc.
  2. Define or use any macros.
  3. Define any additional functions in this file.
  4. Call any functions.
  5. Use any other operations, such as &&, ||, -, or ?:
  6. Use any form of casting.
  7. Use any data type other than int.  This implies that you
      cannot use arrays, structs, or unions.
```

```
   You may assume that your machine:
   1. Uses 2s complement, 32-bit representations of integers.
   2. Performs right shifts arithmetically.
   3. Has unpredictable behavior when shifting an integer by more
      than the word size.

 EXAMPLES OF ACCEPTABLE CODING STYLE:
   /*
    * pow2plus1 - returns 2^x + 1, where 0 <= x <= 31
    */
   int pow2plus1(int x) {
      /* exploit ability of shifts to compute powers of 2 */
      return (1 << x) + 1;
   }

   /*
    * pow2plus4 - returns 2^x + 4, where 0 <= x <= 31
    */
   int pow2plus4(int x) {
      /* exploit ability of shifts to compute powers of 2 */
      int result = (1 << x);
      result += 4;
      return result;
   }


 FLOATING POINT CODING RULES

 For the problems that require you to implent floating-point operations,
 the coding rules are less strict.  You are allowed to use looping and
 conditional control.  You are allowed to use both ints and unsigneds.
 You can use arbitrary integer and unsigned constants.

 You are expressly forbidden to:
   1. Define or use any macros.
   2. Define any additional functions in this file.
   3. Call any functions.
   4. Use any form of casting.
   5. Use any data type other than int or unsigned.  This means that you
      cannot use arrays, structs, or unions.
   6. Use any floating point data types, operations, or constants.


 NOTES:
   1. Use the dlc (data lab checker) compiler (described in the handout) to
      check the legality of your solutions.
   2. Each function has a maximum number of operators (! ~ & ^ | + << >>)
      that you are allowed to use for your implementation of the function.
      The max operator count is checked by dlc. Note that '=' is not
      counted; you may use as many of these as you want without penalty.
   3. Use the btest test harness to check your functions for correctness.
   4. Use the BDD checker to formally verify your functions
   5. The maximum number of ops for each function is given in the
      header comment for each function. If there are any inconsistencies
      between the maximum ops in the writeup and in this file, consider
      this file the authoritative source.

 /*
  * STEP 2: Modify the following functions according the coding rules.
  *
  *    IMPORTANT. TO AVOID GRADING SURPRISES:
  *    1. Use the dlc compiler to check that your solutions conform
  *       to the coding rules.
  *    2. Use the BDD checker to formally verify that your solutions produce
  *       the correct answers.
```

```
   */


 #endif
 /*
  * negate - return -x
  *   Example: negate(1) = -1.
  *   Legal ops: ! ~ & ^ | + << >>
  *   Max ops: 5
  *   Rating: 2
  */
 int negate(int x) {
   return ~x+1;
 }
 /*
  * 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);




 }
 /*
  * anyOddBit - return 1 if any odd-numbered bit in word set to 1
  *   Examples anyOddBit(0x5) = 0, anyOddBit(0x7) = 1
  *   Legal ops: ! ~ & ^ | + << >>
  *   Max ops: 12
  *   Rating: 2
  */
 int anyOddBit(int x) {
     int m8 = 0xAA;
     int m16 = m8 | m8 << 8;
     int m32 = m16 | m16 <<16;
     int oddx = x & m32;
     return !!oddx;
 }
 /*
  * 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) {
     /* Handle rounding by generating bias:
        0 when x >= 0
        2^n-1 when x < 0
     */
     int mask = (1 << n) + ~0;
     int bias = (x >> 31) & mask;
     return (x+bias) >> n;
 }
 /*
  * addOK - Determine if can compute x+y without overflow
  *   Example: addOK(0x80000000,0x80000000) = 0,
  *            addOK(0x80000000,0x70000000) = 1,
  *   Legal ops: ! ~ & ^ | + << >>
  *   Max ops: 20
  *   Rating: 3
```

```c
 */
int addOK(int x, int y) {
  int sum = x+y;
  int x_neg = x>>31;
  int y_neg = y>>31;
  int s_neg = sum>>31;
  /* Overflow when x and y have same sign, but s is different */
  return !(~(x_neg ^ y_neg) & (x_neg ^ s_neg));
}
/*
 * isGreater - if x > y  then return 1, else return 0
 *    Example: isGreater(4,5) = 0, isGreater(5,4) = 1
 *    Legal ops: ! ~ & ^ | + << >>
 *    Max ops: 24
 *    Rating: 3
 */
int isGreater(int x, int y) {
  int x_neg = x>>31;
  int y_neg = y>>31;
  return !((x_neg & !y_neg) | (!(x_neg ^ y_neg) & (~y+x)>>31));
}
/*
 * replaceByte(x,n,c) - Replace byte n in x with c
 *    Bytes numbered from 0 (LSB) to 3 (MSB)
 *    Examples: replaceByte(0x12345678,1,0xab) = 0x1234ab78
 *    You can assume 0 <= n <= 3 and 0 <= c <= 255
 *    Legal ops: ! ~ & ^ | + << >>
 *    Max ops: 10
 *    Rating: 3
 */
int replaceByte(int x, int n, int c) {
  /* Mask out current byte value and OR in replacement */
  int n8 = n << 3;
  int mask = 0xff << n8;
  int cshift = c << n8;
  return (x & ~mask) | cshift;
}
/*
 * tc2sm - Convert from two's complement to sign-magnitude
 *    where the MSB is the sign bit
 *    You can assume that x > TMin
 *    Example: tc2sm(-5) = 0x80000005.
 *    Legal ops: ! ~ & ^ | + << >>
 *    Max ops: 15
 *    Rating: 4
 */
int tc2sm(int x) {
  int mask = x >> 31;
  int sign = mask & 1;
  /* Compute absolute value of x */
  int abs = (mask ^ x) + sign;
  return abs | (sign << 31);
}
```