```
/*
 * CS:APP Data Lab
 *
 * 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.
 */

#include "btest.h"
#include <limits.h>

/*
 * Instructions to Students:
 *
 * STEP 1: Fill in the following struct with your identifying info.
 */
team_struct team =
{
   /* Team name: Replace with either:
      Your login ID if working as a one person team
      or, ID1+ID2 where ID1 is the login ID of the first team member
      and ID2 is the login ID of the second team member */
    "prof-01",
   /* Student name 1: Replace with the full name of first team member */
   "GABARITO",
   /* Login ID 1: Replace with the login ID of first team member */
   "prof-01",

   /* The following should only be changed if there are two team members */
   /* Student name 2: Full name of the second team member */
   "",
   /* Login ID 2: Login ID of the second team member */
   ""
};

#if 0
/*
 * STEP 2: Read the following instructions carefully.
 */

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

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.

  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;
  }


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. 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.
#endif
```

```
/*
 * STEP 3: 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 btest test harness to check that your solutions produce
 *      the correct answers. Watch out for corner cases around Tmin and Tmax.
 */
/*
 * bitNor - ~(x|y) using only ~ and &
 *   Example: bitNor(0x6, 0x5) = 0xFFFFFFF8
 *   Legal ops: ~ &
 *   Max ops: 8
 *   Rating: 1
 */
int bitNor(int x, int y) {
  return (~x & ~y);
}
/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 *   Max ops: 14
 *   Rating: 2
 */
int bitXor(int x, int y) {
  long int x_and_y = x&y;
  long int x_or_y = ~(~x & ~y);
  return x_or_y & ~x_and_y;



}
/*
 * isNotEqual - return 0 if x == y, and 1 otherwise
 *   Examples: isNotEqual(5,5) = 0, isNotEqual(4,5) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int isNotEqual(int x, int y) {
  return !!(x ^ y);
}
/*
 * 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) {
  /* Shift x n*8 positions right */
  int shift = n << 3;
  int xs = x >> shift;
  /* Mask byte */
  return xs & 0xFF;
}
/*
 * copyLSB - set all bits of result to least significant bit of x
```

```
 *   Example: copyLSB(5) = 0xFFFFFFFF, copyLSB(6) = 0x00000000
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int copyLSB(int x) {
  /* Shift bit to MSB and then right shift (arithmetically) back */
  int result = (x<<31)>>31;
  return result;
}
/*
 * logicalShift - shift x to the right by n, using a logical shift
 *   Can assume that 1 <= n <= 31
 *   Examples: logicalShift(0x87654321,4) = 0x08765432
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 16
 *   Rating: 3
 */
int logicalShift(int x, int n) {
  return (x >> n) & ((1 << (32 + (~n+1))) + ~0);
}
/*
 * bitCount - returns count of number of 1's in word
 *   Examples: bitCount(5) = 2, bitCount(7) = 3
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 40
 *   Rating: 4
 *   Comentários da correção: muitos erraram ou não
 * conseguiram fazer corretamente
*/
int bitCount(int x) {
    /* Sum 8 groups of 4 bits each */
    int m1 = 0x11 | (0x11 << 8);
    int mask = m1 | (m1 << 16);
    int s = x & mask;
    s += x>>1 & mask;
    s += x>>2 & mask;
    s += x>>3 & mask;
    /* Now combine high and low order sums */
    s = s + (s >> 16);
    /* Low order 16 bits now consists of 4 sums,
       each ranging between 0 and 8.
       Split into two groups and sum */
    mask = 0xF | (0xF << 8);
    s = (s & mask) + ((s >> 4) & mask);
    return (s + (s>>8)) & 0x3F;
}
/*
 * bang - Compute !x without using !
 *   Examples: bang(3) = 0, bang(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 *   Comentários da correção: muitos erraram ou não
 * conseguiram fazer corretamente
*/
int bang(int x) {
  int minus_x = ~x+1;
  /* Cute trick: 0 is the only value of x
   * for which neither x nor -x are negative */
  return (~(minus_x|x) >> 31) & 1;
}
```

```
/*
 * leastBitPos - return a mask that marks the position of the
 *               least significant 1 bit. If x == 0, return 0
 *   Example: leastBitPos(96) = 0x20
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 4
 */
int leastBitPos(int x) {
  /* The only bit set in both x and -x will be the least significant one */
  return x & (~x+1);
}
/*
 * TMax - return maximum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmax(void) {
  return ~(1 << 31);
}
/*
 * isNonNegative - return 1 if x >= 0, return 0 otherwise
 *   Example: isNonNegative(-1) = 0.  isNonNegative(0) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 3
 */
int isNonNegative(int x) {
    return ((~x)>>31) & 0x1;
}
/*
 * 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));
}
/*
 * 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
 *   Comentários da correção: dificuldade em fazer a questão corretamente
*/
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;
}
/*
```

```
 * abs - absolute value of x (except returns TMin for TMin)
 *   Example: abs(-1) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 4
 */
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1L;
}
/*
 * 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
 */
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));
}
```