```
/*

 *
 * bits.c - Source file with your solutions to the Lab.
 *          This is the file you will hand in to your instructor.
 *
 */

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

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.


#endif
/*
 * bitAnd - x&y using only ~ and |
 *   Example: bitAnd(6, 5) = 4
 *   Legal ops: ~ |
 *   Max ops: 8
 *   Rating: 1
 */
int bitAnd(int x, int y) {
   // negating each input , bitwise ORinng them and then negating the result is eqivalent to
   AND . ~((~)|(~))
   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) {

  int shift = n * 8;
  x = x >> shift;         // bring the required byte to lower byte level
  x = x & 0x000000FF;      // extract the byte alone

  return x ;
}
/*
 * 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) {
  // right shift the numbers and make the n ms bits as 0
  x = x >>n ;    // right shift
  int shift = 1 <<31;    // 1 followed by 31 0s
  shift = (shift >> n )<<1; // n number of msb bits set
  shift = ~shift;          // negate
  return x&shift;          // set n number of bits to 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
 */
int bitCount(int x) {
    // right shift x  check eavh lsb, add lsb bit value to the count.
    int count =0, temp ;

    temp = x&1;        // bit 0
    count = count+ temp;

    x = x >>1;          // bit 1
    temp = x&1;
    count = count+ temp;

    x = x>>1;            // bit 2
    temp = x&1;
    count = count+ temp;

    x = x>>1;           // bit 3
    temp = x&1;
    count = count+ temp;

    x = x>>1;           // bit 4
    temp = x&1;
    count = count+ temp;

    x = x>>1;         // bit 5
    temp = x&1;
    count = count+ temp;

    x = x>>1;        // bit 6
```

```
    temp = x&1;
    count = count+ temp;

    x = x>>1;            // bit 7
    temp = x&1;
    count = count+ temp;

    return count;

}
/*
 * bang - Compute !x without using !
 *   Examples: bang(3) = 0, bang(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
int bang(int x) {
    // for x != 0 x & -x have different MSBs .. for X =0 , x & -x have MSB =0;
    int  minus_x = (~x)+1;  // -x with 2's complement
    x = ~(minus_x ^ x);   // x != 0 -> MSB = 0   x=0-> MSB = 1;
    x = (x>>31)& 1 ;   // extract MSB
    return x;


}
/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
  return 2;
}
/*
 * 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 2;
}
/*
 * 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) {



}
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
```

```
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x)
 //  2's compliment of a number is negative of the number - bitwise negate and add 1
  x = ~x;   //  1's complement
  x = x+1;    // add 1
  return x ;
}
/*
 * isPositive - return 1 if x > 0, return 0 otherwise
 *   Example: isPositive(-1) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 3
 */
int isPositive(int x) {
    // if MSB is 1 -> the number is negative else positive.
    x = x & 128;    // obtain MSB bit
    x = x >>7;      // bring it to LSB
    return !x;      // return negation of the bit
}
/*
 * 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) {

}
/*
 * 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) {
     /*
    Below is the logic used :
    while (x >0){
        x = x/2;
        count++;
    }
    x right shift by 1 in each step . if result is non zero value increment the count by 1
    otheriwse by 0
    */
    int temp, y, count=0;

    y= x>>1;           //-y1     x/2
    temp = (~y)+1;      //-y
    temp = temp ^ y;          // if x =0 MSB =0; if x != 0 MSB =1
    temp = (temp >>31)& 1;       // extract MSB
    count = count +temp;        // increment count by temp   if x =0 -> temp =0 else temp =1;

    y= y>>1;                    //x/4
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    temp = temp <<1;
    count = count +temp;
```

```
        y= y>>1;                //x/8
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    count = count +temp;

    y= y>>1;               // x/16
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    count = count +temp;

    y= y>>1;                    //x/32
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    count = count +temp;

    y= y>>1;                    //x/64
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    count = count+temp;

    y= y>>1;                  // x/128
    temp = (~y)+1;  //-y1
    temp = temp ^ y;
    temp = (temp >>31)& 1;
    count = count+temp;

    return count-1;
}
/*
 * 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) {
  return 2;
}
/*
 * 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) {
   /*
    31_1__30_____8_________23________ 23__________0
    |  S  | Exponent       |  Fraction            |
    ____________________________________________

    */

    // -(1)^ s x (1+ Fraction) X 2 ^ (Exponent -127)
```

```
    // x = x.0 = (-1)^ MSB x  (1+(x-1)) x  2 ^ (127 -127)
    /*
       Compare the two equations:
       s = MSB   Fraction = x-1 Exponent = 127
    */

   unsigned sign ;
    unsigned fraction;
    unsigned exponent ;

    sign = 1<<31;
    sign = x&sign;

    if (sign){
      x = (~x)+1;
    }
    int y =x;
    int shift =0;
    while (y !=0){
        y = y>>1;
        shift++;
    }

    shift = shift-1;
    int temp = 1<<31;


    temp = temp >>(31-shift);
    temp = ~temp;


    fraction = (temp & x)<<(23-shift) ;


    exponent = 127+shift ;

    if ((exponent == 255)&& (fraction != 0)){
      return 0x7FC00000 ;  // Not a number
    }

    exponent = exponent <<23;

    return sign|exponent|fraction;




}
/*
 * 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)
{
```

```
}
```