The purpose of this assignment is to become more familiar with bit-level representations of common patterns and integers. You'll do this by solving a series of programming "puzzles." Many of these puzzles are quite artificial, but should help you understand machine representation better.

Coding Rules:

You may use only straight-line code for the integer puzzles (i.e., no loops, function calls, or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are only allowed to use the following eight operators:

`"!", "~", "&", "^", "|", "+", "<<" and ">>"`

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than 8 bits (ranging from hex 0x00 to 0xFF). For the integer puzzles, you may use casting, but only between data types int and long (in either direction.) See the comments in bits.c for detailed rules and a discussion of the coding rules for each function. You can assume the following:

- Values of data type int are 32 bits.
- Values of data type long are 64 bits.
- Signed data types use a two's complement representation.
- Right shifts of signed data are performed arithmetically.
- When shifting a w-bit value, the shift amount should be between 0 and w–1.
- Predicate operators, including the unary operator ! and the binary operators ==, !=, <, >, <=, and >=, return values of type int, regardless of the argument types.

The Puzzles:

This section describes the puzzles that you will be solving in bits.c.

1. Bit Manipulations

Name	Description	Points	Max-Ops
`copyLSB(x)`	Set all bits of result to least significant bit of x	2	5
`allOddBits(x)`	Return 1 if all odd-numbered bits in word are set to 1	2	14
`rotateLeft(x, n)`	Rotate x to the left by n	3	25
`bitMask(highbit, lowbit)`	Generate a mask consisting of all 1's between lowbit and highbit, 0's elsewhere	3	16
`bitCount(x)`	Return count of number of 1's in word	4	50

The table above describes a set of functions that manipulate and test sets of bits. The "Points" field gives the number of points for the puzzle, and the "Max-Ops" field gives the maximum number of operators you are allowed to use to implement each function. See the comments in **`bits.c`** for more details on the desired behavior of the functions. You may also refer to the test functions in tests.c. These are used as reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules for your functions. All arguments and return values for the functions are of type long.

2. Two's Complement Arithmetic

Name	Description	Points	Max-Ops
`isNotEqual(x, y)`	return 0 if x == y and 1 otherwise	2	6
`dividePower2(x, n)`	Return $x/2^n$ for $0 \le n \le 62$, rounding towards 0	2	15
`remainderPower2(x, n)`	Compute $x\%2^n$ for $0 \le n \le 30$	3	20
`isPower2(x)`	Returns 1 if x is a power of 2, and 0 otherwise	3	20
`allAsciiDigits(x)`	Return 1 if each byte b in x satisfies $0x30 \le b \le 0x39$ (ASCII codes for characters '0' - '9')	4	30
`trueThreeFourths(x)`	Multiply and return x by 3/4 rounding towards 0, avoiding errors due to overflow	4	20

The table above describes a set of functions that make use of the two's complement representation of integers. All arguments and return values are of type long. Refer to the comments in bits.c and the reference versions in tests.c for more information. You can use the provided program **`ishow`** to see the decimal and hexadecimal representations of numbers. First, compile the code. Then use it to examine hex and decimal values typed on the command line:

```
prompt> ./ishow 0x8000000000000000
Hex = 0x8000000000000000L, Signed =
-9223372036854775808L,Unsigned = 9223372036854775808L

prompt> ./ishow -123456789
Hex = 0xfffffffff8a432ebL,Signed = -123456789L,Unsigned =
18446744073586094827L
```

Your score will be computed out of a maximum of **54 points** based on the following distribution:
32: Correctness of code.
22: Performance of code, based on number of operators used in each function.

Correctness points: The 11 puzzles you must solve have been given points, which sum to a score of 32. You will receive full correctness points for a puzzle if it passes all coding rules and passes the BDD checker, and no credit otherwise.

Performance points: Our main concern at this point in the course is that you can get the right answer. However, while some of the puzzles can be solved by brute force, it is possible to develop more efficient solutions. Thus, you will receive 2 performance points for each correct function that satisfies the operator limits described above. However, keep in mind that you can still receive correctness points even if the operator limit is exceeded.

Grading the assignment:

The handout directory contains `btest`, `dlc`, and BDD checker to help you check the correctness of your work.

- The `btest` program checks the correctness of the functions in `bits.c` by calling them many times with many different argument values. To build and use it, type the following two commands:

  ```
  prompt> make
  prompt> ./btest
  ```

  Notice that you must rebuild `btest` each time you modify your `bits.c` file. You'll find it very helpful to use `btest` to work through the functions one at a time, testing each one as you go. You can use the -f flag to instruct `btest` to test only a single function:

  ```
  prompt> ./btest -f copyLSB
  ```

  This will call the copyLSB function many times with many different input values. You can feed `btest` specific function arguments using the option flags -1, -2, -3 for the first three function arguments respectively:

  ```
  prompt> ./btest -f copyLSB -1 0xFF
  ```

  This will call copyLSB exactly once, using the specified arguments. Use this feature if you want to debug your solution by inserting printf statements; otherwise, you'll get too much output.
  Warning: the `btest` program does not exhaustively test correctness!
  You also need to run `bddcheck` as described below.

- `dlc` – The `dlc` program checks for compliance with the coding rules for each puzzle. The program will print an error if it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. Running with the -e switch causes dlc to print counts of the number of operators used by each function:
  ```
  prompt> ./dlc -e bits.c
  ```

- The `btest` program simply tests your functions for a number of different cases. For most functions, the number of possible argument combinations far exceeds what could be tested exhaustively. To provide complete coverage, we have created a formal verification program, called `cbit`, that exhaustively tests your functions for all possible combinations of arguments. It does this by using a data structure known as Binary Decision Diagrams (BDDs). You do not invoke `cbit` directly. Instead, there is a series of Perl scripts that set up and evaluate the calls to it. To check all of your functions and get a compact tabular summary of the results, execute:
  ```
  prompt> ./bddcheck/check.pl -g
  ```

You will need to upload your `bits.c` file to receive the grade. You must remove any extraneous print statements from your bits.c file before handing in. Don't include any header files in your bits.c file.