The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers. You will solve a series of programming tasks, some of which are quite trivial, but others are indeed implementations of often-used tasks that are part of important applications for network procedures, calculating distances, etc.

Outcome

The goal is for you to find yourself thinking much more about bits in working your way through the programming tasks.

As with the other parts of this class, you can work in groups of two, or solo. However, you are NOT allowed to work in groups of 3 or more, and groups cannot communication solutions nor approaches with other groups.

Files

Download from the files section of Canvas the project 1 zip file, and move it to your CS account. Use sftp, or an ftp client such as FileZilla. Unzip it via `unzip project1_manipulatingBits.zip`

Implementation Details

The only file you will be modifying and turning in is *bits.c*.

The *bits.c* file contains a skeleton for each of several programming tasks (functions). Your assignment is to complete each function skeleton using only straight-line code for the integer puzzles (i.e., no loops nor conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are only allowed to use the following eight operators:

```
!     ~     &     ^     |     +     <<     >>
```

A few of the functions in *bits.c* further restrict this list, so read closely the comments for each function. Also, you are not allowed to use any constants longer than 8 bits. See the comments in *bits.c* for detailed rules and a discussion of the desired coding style.

Important: Carefully read the instructions in the *bits.c* file before you start. These give the coding rules that you will need to follow.

Hint: The comments for each function also provide a hint, which mentions approximately how many lines of code a good solution might require. That is not to imply that your solution must use that few number of lines of code, but if you find yourself writing far more than that many lines of code, perhaps you are overthinking it and you should step back and brainstorm.

The Puzzles (tasks, functions)

There are 3 types of puzzles: Bit manipulation, Two's Complement, and Floating Point Manipulations. This section describes the puzzles that you will be solving in *bits.c*.

Bit Manipulations

Table 1 describes a set of functions that manipulate and test bit patterns. The *Difficulty* field gives the difficulty rating (the number of points) for the puzzle, and the *Max ops* field gives the maximum number of operators you can 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.

Table 1: Bit-Level Manipulation Functions.

Function name	Description	Difficulty	Max Ops
`bitAnd(x,y)`	x & y using only | and ~	1	8
`getByte(x,n)`	Get byte n from x	2	6
`logicalShift(x,n)`	Shift right logical	3	20
`bitCount(x)`	Count the number of 1's in x.	4	40

Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers. Again, refer to the comments in *bits.c* and the reference versions in *tests.c* for more information.

Table 2: Arithmetic Functions

Function name	Description	Difficulty	Max Ops
`tmin()`	Most negative two's complement integer	1	4
`fitsBits(x,n)`	Does x fit in n bits?	2	15
`divpwr2(x,n)`	Compute x/2n	2	15
`negate(x)`	-x without negation	2	5
`isPositive(x)`	x > 0?	3	8
`ilog2(x)`	Compute log2(x)	4	90

Floating-Point Operations

For this part of the assignment, you will implement some common single-precision floating-point operations. You can use standard control structures (conditionals, loops), and you may use both int and unsigned data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having type unsigned, and any returned floating-point value will be of type unsigned. Your code should perform the bit

manipulations that implement the specified floating point operations. Table 3 describes a set of functions that operate on the bit-level representations of floating-point numbers. Refer to the comments in *bits.c* and the reference versions in *tests.c* for more information.

Table 3: Floating-Point Functions. Value f is the floating-point number having the same bit representation as the unsigned integer uf.

Function name	Description	Difficulty	Max Ops
`float_neg(uf)`	Compute -f	2	10
`float_twice(uf)`	Computer 2*f	4	30

Functions `float_neg` and `float_twice` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. The IEEE standard does not specify precisely how to handle NaN's, and the IA32 behavior is a bit obscure. We will follow a convention that for any function returning a NaN value, it will return the one with bit representation 0x7FC00000. The provided program *fshow* helps you understand the structure of floating point numbers. To compile *fshow*, type:

```
unix> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
unix> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized. 1.0000000000 X 2^(121)
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

Auto-grading your work

You are being provided some autograding tools -- *btest*, *dlc*, and *driver.pl* -- to help you check the correctness of your work.

btest

This program checks the functional correctness of the functions in *bits.c*. To build and use it, type the following two commands:

```
unix> make
unix> ./btest
```

The make program is an industry/CS standard macro-like approach for compiling and linking programs, which otherwise you'd need to do manually via the command line.

You must rebuild *btest* each time you modify your *bits.c* file. You'll find it helpful 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:

```
unix> ./btest -f bitAnd
```

You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`:

```
unix> ./btest -f bitAnd -1 7 -2 0xf
```

Check the file *README* for documentation on running the *btest* program.

dlc

This is a modified version of an ANSI C compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

```
unix> /home/clausoa/bin/dlc bits.c
```

This command will invoke the *dlc* program, that is in the `/home/clausoa/bin` directory.

The program runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straight-line code in the integer puzzles. Running with the `-e` switch:

```
unix> /home/clausoa/bin/dlc -e bits.c
```

causes *dlc* to print counts of the number of operators used by each function. Type `dlc -help` for a list of command line options.

NOTE: *dlc* is installed in Dr. Clauson's personal bin directory. If you want to simplify your life, you can add that bin directory to your PATH variable (google *bash PATH*) or create an alias (google *bash alias*).

driver.pl

This is a driver program that uses *btest* and *dlc* to compute the correctness and performance points for your code. It will compile your *bits.c* file, and test each of your functions. The driver.pl program takes no arguments:

```
unix> ./driver.pl
```

The *driver.pl* program will be used to evaluate your solution.

If you get an execution error, you might need to grant execution rights to the driver.pl program. To do that, you would issue the following (which also gives read and write permissions):

```
unix> chmod u+rwx driver.pl
```

Technical caveats

- Don't include the `<stdio.h>` header file in your *bits.c* file, as it confuses *dlc* and results in some non-intuitive error messages. You will still be able to use `printf` in your *bits.c* file for debugging without including the `<stdio.h>` header, although `gcc` will print a warning that you can ignore.
- The *dlc* program enforces a stricter form of C declarations than is the case for C++ or that is enforced by gcc. Any declaration must appear in a block (what you enclose in curly braces) before any statement that is not a declaration. In other words, for each function you must declare all your variables first. For example, it will complain about the following code:

```
int foo(int x)
{
   int a = x;
   a *= 3;    <- Statement that is not a declaration
   int b = a; <- ERROR: Declaration not allowed here
}
```

Rubric and Submission

Upload your *bits.c* file to Canvas. **Important**: If you are not able to complete one or more of the puzzles (functions), then do not upload a *bits.c* file that does not compile. Instead, simply keep the original version of that function (the one that returns 2).

Correctness points. The 12 puzzles are assigned a difficulty rating between 1 and 4, such that their weighted sum totals 30. Your functions will be evaluated using the *btest* program. You will get full credit for a puzzle if it passes all the tests performed by *btest*, and no credit otherwise.

Performance points. Our main concern at this point in the course is that you can get the right answer. However, we want to instill in you a sense of keeping things as short and simple as you can. Some of the puzzles can be solved by brute force, but we want you to be more clever. Thus, for each function we've established a maximum number of operators that you can use for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions. You will receive two points for each correct function that satisfies the operator limit.

Style points. 6 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

Item	Points
Correctness (difficulty level points for each puzzle	30
Performance (using maximum count of operators)	24
Formatting and Style	6