CS33, Spring 2018
Lab 1: Data Lab
Assigned: Monday April 2**, Due:** Friday April 20, 11:59PM

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers. You will do this by solving a series of programming "puzzles". Many of these puzzles are quite artificial, but you'll find yourself thinking much more about bits in working your way through them.

2 Logistics

You must work on this lab independently, but you may discuss the assignment with other students. All code you write and submit must be your own.

3 Handout Instructions

We will use a shared directory on SEASnet machine lnxsrv.seas.ucla.edu to distribute the files needed for each lab. For this lab, log in to your account on SEASnet machine and start copying the folder datalab1-handout to your local directory using this command:

```
unix> cp -r /w/class.1/cs/cs33/csbin/datalab1-handout .
```

This will make a copy of the handout folder in your directory. The only file you will be modifying and turning in is bits.c. The file btest.c allows you to evaluate the functional correctness of your code. The README file contains additional documentation about btest. Use the command make btest to generate the test code and run it with the command ./btest. The file dlc is a compiler binary that you can use to check your solutions for compliance with the coding rules. The remaining files are used to build btest.

Looking at the file bits.c, you will notice a C structure studentID into which you should insert the requested identifying information about yourself. Do this immediately so that you do not forget.

The bits.c file contains a skeleton for each of the 8 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles (i.e., no loops or 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 further restrict this list. 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.

4 The Puzzles

Tables 1 and 2 describe a set of functions that manipulate and test sets of bits and two's complement (tc) arithmetic. The "Rating" field gives the difficulty rating 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.

4.1 Bit Manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. Refer to the comments in bits.c and the reference versions in tests.c for more information.

Name	Description	Rating	Max Ops
anyOddBit(x)	anyOddBit - return 1 if any odd-numbered bit in word set to 1	3	12
divpwr2(x,n)	Compute x/(2^n), for 0 <= n <= 30	2	15
replaceByte(x,n,c)	Replace byte n in x with c	3	10
bitAnd(x, y)	Implement x&y using only ~ and |.	1	8

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
addOK(x,y)	Determine if can compute x-y without overflow.	3	20
isGreater(x, y)	If x > y then return 1, else return 0.	3	24
negate(x)	Returns -x.	2	5
tc3sm(x)	Convert from two's complement to sign-magnitude	4	15

Table 2: Arithmetic Functions

4.2 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.

5 Evaluation

Your score will be computed out of a maximum of 36 points based on the following distribution:

20 Correctness points.

16 Performance points.

Correctness points. The 8 puzzles you must solve have been given a difficulty rating between 2 and 4. We will evaluate your functions using the btest program, which is described in the next section. 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. Furthermore, 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. 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.

To get a perfect score on this assignment, you need to get 36 points.

6 Handin Instructions

- Make sure it compiles, passes the dlc test, and passes the btest tests on the class machines, i.e. lnxsrv.seas.ucla.edu.
- Make sure you have included your identifying information in your file bits.c.

- Remove any extraneous print statements.
- Submit your bits.c file to CCLE where indicated under Lab 1.

7 Advice

You are welcome to develop your solution using any system or compiler you choose. However, make sure that the version you turn in can compile and run correctly on our class machine (lnxsrv.seas.ucla.edu). If it does not compile, we cannot grade it.

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

unix> ./dlc bits.c

Type ./dlc -help for a list of command line options. The README file is also helpful. Some notes on dlc:

- The dlc program runs silently unless it detects a problem.
- Do NOT include <stdio.h> in your bits.c file, because it confuses dlc and results in some non-intuitive error messages.

Check the README file for documentation on running the btest program. You will 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, e.g. ./btest -f negate.