CS 33, Spring 2023
Lab 0: Warm-up Lab

Assigned: Monday, April 3rd

Due: Friday, April 7, 11:59PM

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers numbers. You'll 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.

We use the shared directory `/w/class.1/cs/cs33/csbin/` on the SEASnet machine to distribute the files. Log in to your account on the SEASnet machine, and start copying the folder `datalab0-handout` to your local directory using this command:

```
cp -r /w/class.1/cs/cs33/csbin/datalab0-handout .
```

Once you have copied the folder across to your home directory, you can change to the folder via:

```
cd datalab0-handout
```

The first time you copy it, use the chmod command to set the permission of `dlc` to be executable:

```
chmod a+x dlc
```

You only need to do that once. Now modify the `bits.c` file to implement your solution using your favorite editor (e.g. emacs, vim). The only file you will be modifying and turning in is `bits.c`.

The `bits.c` file contains a skeleton of the 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.

We recommend you to code and debug on the SEASnet machine (preferably `cs33.seas.ucla.edu`). You could choose to copy the handout to your personal machine and code and debug locally. *However, the grading script would be run on the SEASnet machine (specifically, `cs33.seas.ucla.edu`),* ***so please make sure to test on it prior to submission.***

2 Evaluation

Your bits.c must first comply with the coding rules - check this with the dlc program, see section 2.2. Otherwise, your program will fail to compile during grading.

The maximum score of the lab is 5. Your score will be computed as the sum of the following:

Total 3 Correctness points.

Total 2 Performance points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4. The correctness point for each puzzle is determined by its difficulty rating. 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 of 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 are allowed to 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.

2.1 **`btest`**

The `btest` program checks the functional correctness of the functions in `bits.c`. To build your code and test it, type the following two commands:

```
make

./btest
```

Notice that 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:

```
./btest -f bitAnd
```

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

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

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

2.2 `dlc`

`dlc` 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:

```
./dlc bits.c
```

The program runs silently unless 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:

```
./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.

3 Advice

- 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`. In particular, any declaration must appear in a block (what you enclose in curly braces) before any statement that is not a declaration. 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 */
}
```

4 Before Submitting

- Make sure it compiles, passes the dlc test, and passes the btest tests on `cs33.seas.ucla.edu`.
- Make sure you have included your identifying information (name and UID) in your `bits.c` file.
- Remove any extraneous print statements.