

# CMSC 311, Fall 2009

## Lab Assignment L1: Manipulating Bits

Assigned: Sept. 9, Due: Sept. 16, midnight

William A. Arbaugh

### Introduction

The purpose of this assignment is to become more familiar with bit-level representations and manipulations. 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.

### Logistics

The only "hand-in" will be electronic (see the section "Hand In Instructions" below for details). Any clarifications and revisions to the assignment will be posted on the class mailing list.

### Hand Out Instructions

You can download the `datalab-handout.tar.gz` file from the course syllabus under Lab1 in assignments.

Start by copying `datalab-handout.tar.gz` to a (protected) directory in which you plan to do your work. Then give the command: `tar xvfz datalab-handout.tar`. This will cause a number of files to be unpacked in the 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 file `README` 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'll notice a C structure `team` into which you should insert the requested identifying information about yourself. Do this right away so you don't forget.

The `bits.c` file also contains a skeleton for each of the 5 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code (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: `! ~ & ^ | + << >>`

| Name                        | Description   | Rating | Max Ops |
|-----------------------------|---|--------|---------|
| <code>evenBits()</code>     | return a word with all even-numbered bits set                           | 10     | 8       |
| <code>fitsBits(x, n)</code> | return 1 if x can be represented as an n-bit, two's complement integer. | 10     | 15      |

Table 1: Bit-Level Manipulation Functions.

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.

## Evaluation

Your code will be compiled with GCC and run and tested on one of the class linux machines. Your score will be computed out of a maximum of 100 points based on the following distribution:

**60** Correctness of code running on one of the class machines.

**40** Performance of code, based on number of operators used in each function.

The 5 puzzles you must solve have been given a difficulty rating between 5 and 20, such that their weighted sum totals to 60. We will evaluate your functions using the test arguments in `btest.c`. You will get full credit for a puzzle if it passes all of the tests performed by `btest.c`, half credit if it fails one test, and no credit otherwise.

Regarding performance, 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 eight points for each function that satisfies the operator limit.

## Part I: Bit manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. The "Rating" field gives the difficulty rating (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.

Function `evenBits` returns a machine-word sized value where all the even-numbered bits are set to 1.

Function `fitsBits` returns 1 if x can be represented in n bits. Examples: `fitsBits(5, 3) = 0`; `fitsBits(-4, 3) = 1`.

| Name                       | Description  | Rating | Max Ops |
|----------------------------|--|--------|---------|
| <code>log2(x)</code>       | return $\text{floor}(\log_2(x))$ , where $x > 0$   | 20     | 90      |
| <code>sum3(x, y, z)</code> | return the sum of 3 integers                       | 15     | 16      |
| <code>tmin()</code>        | return the lowest valued two's complement integer. | 5      | 4       |

Table 2: Arithmetic Functions

## Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Function `log2` returns the LOG BASE 2 of `x` rounded down to an integer. Note that  $x > 0$ .

Function `sum3` returns the sum of it's 3 arguments: `x`, `y`, `z`. Example: `sum3(3, 4, 5)` returns 12. You are not allowed to use the '+' operator, but can make *a single call* to the `sum(x, y)` helper function.

Function `tmin` returns the possible minimum two's complement integer.

### Advice

You are welcome to do your code development using any system or compiler you choose. Just make sure that the version you turn in compiles and runs correctly on the Grace Cluster Linux machines (`linux.grace.umd.edu`). If it doesn't compile, we can't grade it. To access the class accounts you will need to:

```
ssh linux.grace.umd.edu
```

and login using your directory ID and password.

You can confirm that you're logged into one of the Linux systems (as opposed to the Solaris systems) by running the command:

```
uname
```

You should receive the response `Linux`.

The `dlc` program, a modified version of an ANSI C compiler, will be used to check your programs for compliance with the coding style rules. The typical usage is

```
./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.
- Don't include `<stdio.h>` in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages.

Check the file `README` for documentation on running the `btest` program. 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, e.g., `./btest -f isPositive`.

## Hand In Instructions

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- To handin your `bits.c` file, one and only one student must upload it via the `makefile`. Assuming you have edited `bits.c` in the same directory that it was uncompressed into, this is most easily done by changing into that directory and typing:

```
make submit
```