

CS 203, Fall 2012  
Lab 2: Manipulating Bits  
Assigned: Sept 4, Due: Wednesday, September 12th, 11:59pm

## 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

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic.

If you get stuck, you can choose to talk with people outside your group. However, for this lab, I ask you not to look at each other's code, or discuss solutions in code. If you have to consult with others, please keep the discussion more abstract and high-level. Also, please note in your code the names of anyone you consult with on this lab.

## Hand Out Instructions

Start by downloading the handout to a directory in which you plan to do your work. Then give the command: `tar xvf 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 `bits.c` file contains a skeleton for each 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.

## Evaluation

Your code will be compiled with GCC and run and tested on one of the jazz machines. Your score will be computed out of a maximum of 20 points based on:

Correctness of code (worth the most)

Performance of code, based on number of operators used in each function (worth little)

Style points, based on my subjective evaluation of the quality of your solutions and your comments (worth little)

The puzzles you must solve have been given a difficulty rating between 1 and 3. I 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, my main concern at this point in the course is that you can get the right answer. However, it's also important to try to get used to keeping things as short and simple as you can. Furthermore, some of the puzzles can be solved by brute force, but you should try to be more clever. Thus, for each function I'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 function that satisfies the operator limit.

Finally, I've reserved some 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.

## Advice

I have included some autograding tools in the handout directory — `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
```

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:

```
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> ./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:

```
unix> ./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.

- 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 */
}
```

- Don't forget to check out the README included in the lab's .tar file.

## Hand In Instructions

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- Submit `bits.c` via Moodle. If you work in a group, make it clear in your submission who the group members are.