CSCI247, Winter 2017
Data Lab: Manipulating Bits
Assigned: Monday January 16th
Due: Friday Feb 3 by 5:30PM

Dr. J. Michael Meehan

1-15-2017

Your lab assistant is your primary point of contact for this assignment but make use of the mentors in the cf162 lab and my office hours. As always, if you can't make my office hours email for an appointment. Email:
(`meehan@wwu.edu`)

1 Introduction

The purpose of this assignment is for you to become more familiar with bit-level representations of integers and floating point 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.

2 Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course Web pages if needed. You must submit your final program to Canvas in the place provided by the submission deadline.

3 Handout Instructions

> A file called `datalab-handout.tar` is available for download from Canvas under Assignments Project 1 Datalab.

Start by downloading `datalab-handout.tar` to a (protected) directory under your home directory space from Canvas using one of the cs department lab machines running Linux. The usual process is to make a directory for this course as in ”mkdir cs247” make a subdirectory like “Datalab” Download the tar file into this directory. Change the working directory to that directory as in ”cd cs247/Datalab” Then give the command

```
unix> 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`. You can submit your work as many times as you wish and only the last one submitted will be graded but it must be submitted prior to the deadline. Canvas will cut off submissions at the deadline.

The `bits.c` file contains a skeleton for each of the programming puzzles you will be solving. 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:

```
! ˜ & ˆ | + << >>
```

A few of the functions in `bits.c` may 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

This section describes the puzzles that you will be solving in `bits.c`.

4.1 Bit Manipulations

The following table describes the functions that you are to implement. 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. See the comments in `bits.c` for more details on the desired behaviour of the functions. The comments show a C function that you are trying to mimic the behaviour of. These functions don’t satisfy the coding rules for your functions. You are trying to come up with a function that produces the exact same output while only using the allowed operations.

The IEEE standard does not specify precisely how to handle NaN’s, and the IA32 behaviour 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 included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
unix> make
```

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

Name	Rating	Max Ops
dl1	4	10
dl2	3	20
dl3	2	12
dl4	2	12
dl5	2	10
dl6	2	10
dl7	4	10
dl8	1	06
dl9	4	38
dl10	3	12
dl11	1	05
dl12	1	08
dl13	4	20
dl14	1	14
dl15	2	25
dl16	3	16
dl17	2	05
dl18	2	15
dl19	1	06
dl20	3	10
dl21	4	30
dl22	4	30
dl23	4	30
dl23	4	90

Table 1: Puzzle Functions.

```
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.

5 Evaluation

Your score will be computed out of a maximum of 114 points based on the following distribution:

63 Correctness points.

48 Performance points.

5 Style points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 63. 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 instil 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.

Style points. Finally, we've reserved 5 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.

Autograding your work

We 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 functionName
```

You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`: This is useful to test some parameter edge case values where bugs often hide.

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

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. It takes no arguments:

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

  I will use `driver.pl` to evaluate your solutions and generate your scores.

6 Handin Instructions

Handin your work by uploading your modified bits.c file to Canvas in the place provided. You can submit your solutions as often as you like. Only the last submission uploaded before the sue date and time will be used for grading. The autograding tools I have given you will tell you exactly what your score will be.

7 Beat the Prof Contest

For fun, we have a Beat the Prof contest, where students compete against themselves and the instructor. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who beat the instructor's operator count for each puzzle are winners. I have submitted a set of solutions that sets the standard for the number of operators for each function. If you submit a set of solutions that uses fewer operations for each function you have won the contest. A prize will be awarded to the student who beats the prof with the lowest total operations. The solutions I have submitted have been artificially padded with a few opertions more than a known solution to give you a little wiggle room to beat the one I posted.

You submit your entries for the contest by running the driver with the -u option from a Linux box:

linux¿ ./driver.pl -u "Your Nickname"

Nicknames are limited to 35 characters and can contain alphanumerics, apostrophes, commas, periods, dashes, underscores, and ampersands. The most recent entry from each user will appear on the real-time scoreboard, which you can view by pointing a browsers at scofield.cs.wwu.edu:24701 The scoreboard server is updated every 30 seconds.

8 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. This is similar to the rules in the language Pascal. All declarations must precede any "executable statement." 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 */
}
```

- Read the README file included in the datalab.tar file for further explanation of how to run the various helper and testing programs.