cs281: Introduction to Computer Systems

Project Lab: The Datalab – Floats Version

Assigned: Dec. 5, Due: Dec. 12 at 11:59pm

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of the float data types use bit-level operations to manipulate those representations. You'll do this by solving a series of programming "puzzles," like we did early in the semester, but with fewer restrictions.

2 Logistics

This is an individual project. All handins are electronic, through `submitbox`. Clarifications and corrections will be posted through Piazza.

3 Handout Instructions

All the files you require for this assignment have been gathered together in a Unix "tar" file. This is an single archive file containing a set of files. On a Linux lab machine in Olin 219, you should run a web browser and navigate to the "Schedule" tab of the course web page. Then right-click the `datalab-floats-handout.tar` link available in the *Homework* column.

Save the file to a directory specific to your cs281 work. If you left-click or download to somewhere else, you should start by copying `datalab-floats-handout.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then, from the directory containing the tar file, give the command

```
unix> tar xvf datalab-floats-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, adhering to the following rules:

You are allowed to use looping and conditional control. You are allowed to use both ints and unsigneds. You can use *arbitrary* integer and unsigned constants.

You are expressly forbidden to:

1. Define or use any macros.
2. Define any additional functions in this file.
3. Call any functions.

4. Use any form of casting.
5. Use any data type other than int or unsigned. This means that you cannot use arrays, structs, or unions.
6. Use any floating point data types, operations, or constants.

You are still restricted in how many operators from the following set you are allowed to employ:

```
 ! ~ & ^ | + << >>
```

See the comments in `bits.c` for detailed rules and a discussion of the desired coding style. In particular, you should take care to make all variable declarations *first* in each function, followed by the assignments and operations used to solve the puzzle.

4 The Puzzles

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

Table 1 describes a set of functions on floating point representations. 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 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 dont satisfy the coding rules for your functions.

Name	Description	Rating	Max Ops
`float_abs(x)`	Absolute value of float representation in unsigned `x`	2	10
`float_f2i(x)`	Compute (int) of float representation in unsigned `x`	4	30
`float_half(x)`	Compute x * 0.5 as a bit-level float	4	30
`float_i2f(x)`	Compute a bit-level float of the same value as integer x	4	30
`float_neg(x)`	Compute bit-level float of the negation of x	2	10

Table 1: Float Bit-Level Manipulation Functions.

5 Evaluation

Your score will be computed out of a maximum of 30 points based on the following distribution:

16 Correctness points.

10 Performance points.

4 Style points.

Correctness points. The 5 puzzles you must solve have been given a difficulty rating between 2 and 4, such that their weighted sum totals to 16. We will evaluate your functions using the `btest` and `driver.pl` programs, which are described in the next section. You will get full credit for a puzzle if it passes all of the tests performed by the evaluating programs, 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.

Style points. Finally, we've reserved 4 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 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.

- **`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
  ```

  Your instructors will use `driver.pl` to evaluate your solution.

6 Handin Instructions

The only file I require from you is the bits.c file. At present, turning in your assignment simply involves uploading the file via submitbox. The timestamp on the upload will determine any lateness penalty, with anything after 11:59 pm on the due date incurring the mandatory 10% per day penalty.

7 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 */
  }
  ```