CSCE 231, Fall 2020
Lab 3: Manipulating Bits
Code Due: 11:59pm on the day prior to your assigned lab section

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of 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 submissions are electronic. Clarifications and corrections will be posted on the course Web page.

3 Handout Instructions

Start by copying `datalab2-handout.tar` to a (protected) directory on *csce.unl.edu* in which you plan to do your work. Note that you must use *cse.unl.edu* You can login to *csce.unl.edu* by using the same login information as that used to login to *csce.unl.edu*. Once you have copied the file to *csce.unl.edu*, give the following command to unpack the provided archive file:

```
unix> tar xvf datalab2-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 5 programming puzzles. Your assignment is to complete each function skeleton.

4 The Puzzles

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

4.1 Floating-Point Operations

You will implement some common single-precision floating-point operations. In this section, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, **any floating-point operand will be passed to the function as having type `unsigned`, and any returned floating-point value will be of type `unsigned`**. Your code should perform the bit manipulations that implement the specified floating point operations.

Table 1 describes a set of functions that operate on the bit-level representations of floating-point numbers. Refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Name	Description	Rating	Max Ops
`float_neg(uf)`	Compute `-f`	2	10
`float_i2f(x)`	Compute `(float) x`	4	30
`float_twice(uf)`	Compute `2*f`	4	30
`float_abs(x)`	Compute `abs(x)`	2	10
`float_twice(uf)`	Compute `(int) f`	4	30

Table 1: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

Functions `float_neg` and `float_twice` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. The IEEE standard does not specify precisely how to handle NaN's, and the x86 behavior 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:

```
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. For example:

```
unix> ./fshow 0x7fc00000
```

```
Floating point value nan
Bit Representation 0x7fc00000, sign = 0, exponent = 0xff, fraction = 0x400000
Not-A-Number
```

5 Evaluation

Your score will be computed out of a maximum of 38 points based on the following distribution:

16 Correctness points.

10 Performance points.

Correctness points. The 5 puzzles have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 16. 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.

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` (using `make` 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 TAs will use `driver.pl` to evaluate your solution.

6 Handin Instructions

Submit your `bits.c` solution file via Canvas no later than 11:59pm the evening prior to your lab section.

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 */
  }
  ```