

CSED211, Fall 2022
Data Lab2: Manipulating Bits (Floating Points)
Due: Sun., Sept. 25, 11:59PM

Jaechang Kim (`jc.kim@postech.ac.kr`) is the lead person for this assignment.

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of floating point numbers. You'll solve six problems in the presentation.

2 Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on PLMS. If you have any question, please use Q & A board. Submitted codes are assumed to be runnable on programming server.

3 Handout Instructions

Start by copying `datalab-floating-point.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar xvf datalab-floating-point.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 6 programming problems.

4 The Problems

This section describes the problems that you will be solving in `bits.c`. The “Rating” field gives the difficulty rating (the number of points) for the problem, 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 don't satisfy the coding rules for your functions.

4.1 Two's Complement Arithmetic

Your assignment is to complete each function skeleton using only *straightline* code for the integer problems (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.

Table 1 describes a set of functions that make use of the two's complement representation of integers. Again, refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Name	Description	Rating	Max Ops
<code>negate(x)</code>	<code>-x</code> without negation	2	5
<code>isLess(x, y)</code>	<code>x < y</code> ?	3	24

Table 1: Arithmetic Functions

4.2 Floating-Point Operations

For this part of the assignment, 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 2 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.

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 IA32 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:

Name	Description	Rating	Max Ops
<code>float_abs(uf)</code>	Compute absolute value of <code>f</code>	2	10
<code>float_twice(uf)</code>	Computer $2*f$	4	30
<code>float_i2f(x)</code>	Compute (float) <code>x</code>	4	30
<code>float_f2i(uf)</code>	Compute (int) <code>f</code>	4	30

Table 2: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

```
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.

5 Evaluation

Your score will be computed out of a maximum of 19 points.

Correctness points. We will evaluate your functions using the `btest` program, which is described in the next section. You will get full credit for a problem if it passes all of the tests performed by `btest`, and no credit otherwise.

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.

6 Handin Instructions

Upload your source file `bits.c` and report in PLMS. You need to explain your answer in the report. The format of file is `(student number)_(your name).c / .doc`.

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 */
}
```