CSED211, Fall 2025
Data Lab: Manipulating Bits
Due: Wed., Sept. 17, 11:59PM

Instructor: Prof. Jisung Park

Seunghun Oh is the lead person for this assignment. If you have problem doing assignment, check the *FAQ* for Data Lab2 or use the Q&A board in PLMS.

1 Introduction

The purpose of this assignment is 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 Handout Instructions

Download the `datalab-floating-point.tar` from PLMS. Upload the file to a programming server, move it to the directory you want to work on. 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 code for each of the 2 programming puzzles about two's complement and 4 programming puzzles about floating point. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

3 The Puzzles

This section describes the puzzles 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.

3.1 Two's Complement Arithmetic

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 | + << >>
```

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
`negate(x)`	Return `-x`	2	5
`isLess(x,n)`	Returns 1 if `x < y`, and return 0 otherwise	3	24

Table 1: Arithmetic Functions

3.2 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.

Name	Description	Rating	Max Ops
`float_abs(uf)`	Compute absolute value of `f`	2	10
`float_twice(uf)`	Compute `2 * f`	4	30
`float_i2f(x)`	Compute `(float) x`	4	30
`float_f2i(uf)`	Compute `(int) ff`	4	30

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

Functions `float_abs` 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 when of these functions is given a NaN value as ab argument, it will return the same value for the results (Refer to the comments in `bits.c`).

4 Evaluation

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

`Correctness points`. We will evaluate your functions using the *btest* and *dlc* 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 verified with *dlc*, 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.

- **`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.

5 Handin Instructions

Upload your `bits.c` file and report in PLMS with the following format:

- Rename your `bits.c` file as: `[student id].c` (e.g., `20231234.c`)
- Submit your report with **pdf** format: `[student id]`**`.pdf`** (e.g., `20231234`**`.pdf`**)

Submissions with incorrect format will not be graded, no exception (i.e., you will get 0 points for this entire lab).

6 Tips & Advice

- We provide a program `fshow` that will help you to 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.

- 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 */
}
```