Data Lab: Manipulating Bits

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 Handout Instructions

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.

3 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.

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
```

4 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.