Lab 5

Floating Point Puzzles Lab

Due: Week of February 21, before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

The purpose of this assignment[1] is to become more familiar with bit-level representations of floating point numbers. You'll do this by solving a series of programming "puzzles."

The instructions are written assuming you will edit and run the code on your account on the *csce.unl.edu* Linux server. If you wish, you may edit the code in a different environment, but some of the tools will only run on your account on the *csce.unl.edu* Linux server.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Identify the bit fields of an IEEE 754-compliant floating point number
- Obtain the value of an IEEE 754-compliant floating point number
- Manipulate IEEE 754-compliant floating point numbers in a meaningful way
- Apply IEEE 754 "round-to-nearest-even" rounding

Continuing Forward

The familiarity you gain with the IEEE 754 format will pay off handsomely on the first exam.

During Lab Time

During your lab period, the TAs will provide a refresher of the IEEE 754 format, with a particular emphasis on single-precision floating point numbers, and they will guide students through a discussion and discovery of useful bitmasks for this lab. During the remaining time, the TAs will be available to answer questions.

[1]This lab borrowed from Bryant & O'Halloron and modified by Bohn

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

Scenario

Herb explains that the floating point unit on Eclectic Electronics' experimental microprocessor is nearly finished. Profiling some benchmarks, they've found that there are some common, simple computations that he thinks woudl run faster if they're redesigned. For example, computing the negative value of a floating point number is currently accomplished by subtracting that number from 0.

Herb tasks you with writing C code (that will be used by the C-licon tool) that will compute negative value of a floating point number, the absolute value of a floating point number, and multiply a floating point value by 2 – all without using any floating point operations. You can use any bit operations and, thanks to the ALU you wrote, you can use any arithmetic operations (use the conventional + - * / operators). There are a few other limitations, described below.

Herb rests his arm on your cubicle wall. "Um, yeah. While you're at it," he adds, "if you could write the code to cast between **`float`**s and **`int`**s, that'd be great."

1 Getting Started

Download `puzzlelab.zip` or `puzzlelab.tar` from Canvas or ~cse231 on *csce.unl.edu* and copy it to your account on the *csce.unl.edu* Linux server. Once copied, unpackage the file. 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.

2 The Puzzles

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

2.1 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 `uint32_t`, and any returned floating-point value will be of type `uint32_t`**. 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.

		Points		
Name	Description	Correctness	Performance	Max Ops
`float_neg(f)`	Compute `-f`	2	2	10
`float_i2f(i)`	Compute `(float) i`	4	2	30
`float_twice(f)`	Compute `2*f`	4	2	30
`float_abs(f)`	Compute `abs(f)`	2	2	10
`float_f2i(f)`	Compute `(int) f`	4	2	30

Table 1: Floating-Point Functions. Value `f` is a `uint32_t` that should be interpreted as a single-precision floating-point number, and `i` is a 32-bit signed `int`.

Functions **`float_neg`**, **`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 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:

```
csce> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
csce> ./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:

```
csce> ./fshow 0x7fc00000

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

3 Evaluation

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[2] 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. *You will* not *receive performance points for puzzles that score 0 correctness points.*

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. (You may have to add execute permissions: `chmod +x dlc` and `chmod +x driver.pl`)

- **`btest`:** This program checks the functional correctness of the functions in `bits.c`. To build and use it, type the following two commands:

  ```
  csce> make
  csce> ./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:

  ```
  csce> ./btest -f float_neg
  ```

  You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`:

  ```
  csce> ./btest -f float_neg -1 7 -2 0xf
  ```

  Check the file `README` for documentation on running the `btest` program.

[2] An operator is any arithmetic operator, any bitwise operator, any bit shift operator, any logical operator, or any comparator.

- **`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:

  ```
  csce> ./dlc bits.c
  ```

  The program runs silently unless it detects a problem, such as an illegal operator, too many operators.Running with the `-e` switch:

  ```
  csce> ./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:

  ```
  csce> ./driver.pl
  ```

  We will use `driver.pl` to evaluate your solution.

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 modern versions of C. 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 */
  }
  ```

- Read each function's comments in `bits.c` that describe the expected behavior.

- For some puzzles, NaN is the only special case. In other puzzles, you will need to explicitly think about NaN, normal & subnormal numbers, zero & infinity, and positive & negative values.
- You may want to create some bit pattern constants that correspond to values (or non-values) that you'll use frequently, and also bit pattern constants that you might use as bitmasks frequently. Do not express these in decimal, as it difficult to recognize the bit patterns of all but a very small subset of the decimal values – instead use hexadecimal or, if you must, binary. For example, it isn't obvious that `1610612736` has all 0s except for 1s in bits 30 and 29. It takes only a little effort to see that `0x60000000` has that bit pattern. `0b01100000000000000000000000000000` is quite explicit about its bit pattern, though readability does suffer.
  - *Note* The `dlc` compiler does not recognize the `const` keyword, so use `#define` to create named constants. Other than named constants, you may *not* use `#define` to create macros.
- For some puzzles, you may find it easier if you break the bit pattern for your floating-point value into its sign, exponent, and fraction fields to deal with each separately.
- Do the 2-point puzzles first as a warmup and then move on to the 4-point puzzles.
  - My personal opinion of the puzzles sorted by increasing difficulty: **`float_neg`**, **`float_abs`**, **`float_twice`**, **`float_f2i`**, **`float_i2f`**. Your experience may differ.
- Worry about functional correctness first. If you exceed the allowable number of operations, *then* worry about improving the efficiency of your solution. A horribly-inefficient but correct solution will receive its correctness points, but a very efficient but incorrect solution will receive 0 points.
- When you run `make`, you will see a couple of warnings for *btest.c*. You may safely ignore those. You should, however, address any warnings for *bits.c* (except for warnings about **`printf()`**). You will not be penalized for having warnings; however, warnings are the compiler's way of telling you that your code might not do what you want it to do.
- Frequent Questions' Answers:
  - That edge case is explained in the function's header comment
  - You need to round the fraction
  - The fraction overflowed when you rounded it
  - That works for normal numbers but not subnormal numbers
  - That works for finite numbers but not infinity
  - Notice that $-(-2,147,483,648_{10}) \neq 2,147,483,648_{10}$

- I have a collection of online solutions that I've found over the years. You should assume that I'll look for student solutions that are duplicates of others' solutions.

5 Beat the Professor

Here is the score from the sample solution I wrote:

```
    Correctness Results     Perf Results
    Points  Rating  Errors  Points  Ops  Puzzle
    2       2       0       2       3    float_neg
    4       4       0       2       20   float_i2f
    4       4       0       2       10   float_twice
    2       2       0       2       2    float_abs
    4       4       0       2       15   float_f2i

Score = 26/26 [16/16 Corr + 10/10 Perf] (50 total operators)
```

You can earn bonus points for this lab by scoring close to my score. If your raw score is 26/26, and if you have fewer than 54 total operators, then you will get bonus credit: $54 - \textit{total operators}$. If you match my operator count, then you would get 4 bonus points, for a score of 30/26.

If you beat my score (if your raw score is 26/26 and you have strictly fewer than 50 total operators), then you get an additional flat bonus of 4 points applied to this lab on top of the calculated bonus. For example, if you had 48 total operators, then your score would be 36/26 $(26 + (54 - 48) + 4 = 36)$.

Turn-in and Grading

When you have completed this assignment, upload *bits.c* to Canvas.

This assignment is worth 26 points.

_____ **+2 `float_neg()`** meets functional specification

_____ **+2 `float_neg()`** meets operator count specification (must also meets functional specification)

_____ **+4 `float_i2f()`** meets functional specification

_____ **+2 `float_i2f()`** meets operator count specification (must also meets functional specification)

_____ **+4 `float_twice()`** meets functional specification

_____ **+2 `float_twice()`** meets operator count specification (must also meets functional specification)

_____ **+2 `float_abs()`** meets functional specification

_____ **+2 `float_abs()`** meets operator count specification (must also meets functional specification)

_____ **+4 `float_f2i()`** meets functional specification

_____ **+2 `float_f2i()`** meets operator count specification (must also meets functional specification)

_____ **Bonus +(54-*total operators*)** has a raw score of 26/26 and uses fewer than 54 operators

_____ **Bonus +4** has a raw score of 26/26 and uses fewer than 50 operators

Penalties

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Epilogue

Lil enters the room. Herb challenges her: "*Gulliver's Travels*, 'endian'," and Lil answers, "ends."

Lil walks up to you and says, "We have the insurance situation taken care of, and it's time to get the Zoo ready for guests. We're reassembling the tech team, and there's plenty of work to do."

You smile. "That's good news!"

Lil's face is hard to read. "Well, yes and no. It's good that you'll be able to resume work on the Zoo's systems. But while Archie was waiting for us to fix the insurance situation, he picked up a book on assembly code. He has some ideas. There's another 'opportunity to succeed' waiting for you."

To be continued...