Assignment #2: DataLab II (due on Sun, Feb. 10, 2019 by 11:59pm)

Introduction

This assignment continues on bitwise operations, and asks you to manipulate the encoding of 32-bit IEEE 754 floating-point variables (`float` in C) at the bit level.

Notably, problems on the manipulation of floating point variables allow you to use a larger set of operations, including loops (`for`, `while`, `do while`) and conditional statements (`if`).

To solve the problems, you must know:

- The bit-level format of `float` (1 bit for sign, 8 bits for exponent in excess-127 format, 23 bits for fraction).
- How to extract (and combine) sign/exponent/fraction fields using bitwise operations.
- The bit-level encoding of special values (positive/negative zero and infinity, NaN).
- How denormalized floating point numbers work.

Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course Piazza page. Remember that:

- Any time you receive help from a TA/CP, you should acknowledge that in your code with a comment starting with the string “assistance from”.
- You are not allowed to search for help online!
- You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.
- Reconsideration requests must be made within one week of our release of grades for the assignment.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why a few weeks later, if needed.

Handout Instructions

Similarly to the previous assignment, we will add files of this assignment to the private GitHub repository that was shared with you. You will find these files in the `proj2` directory. Be sure to pull the GitHub repository inside the class VM (`git pull`).

The only file you will be modifying and turning in is `bits.c`, although other files will help you, such as to find out what your grade will be. The `bits.c` file contains a skeleton for each of the programming puzzles. Your assignment is to complete each function skeleton using only straight-line 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 puzzles/functions may further restrict this list so read the comments of each puzzle/function carefully. 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.

In contrast, **floating-point puzzles allow you to use 4-byte constants, loops and conditionals.**

The Puzzles

Bitwise Manipulations

- `int getByte(int x, int n)` (2 point, 6 ops allowed): extract the `n`-th byte of `x` (where `n` can be equal to 0, 1, 2, or 3, and bytes are counted from the right).
- `int replaceByte(int x, int n, int c)` (3 points, 10 ops allowed): replace the `n`-th byte of `x` with `c` (an `int` between 0 and 255).

Floating-Point Operations

- `unsigned floatNegate(unsigned uf)` (2 points, 10 ops allowed): given the bit-level encoding of a `float`, return the bit-level encoding of its opposite (i.e., invert the sign).
- `unsigned floatScale2(unsigned uf)` (4 points, 30 ops allowed): given the bit-level encoding of a `float`, return the bit-level encoding resulting after its multiplication by `2.0`. This is a challenging puzzle! Keep in mind the following cases:
  - Denormalized floats have an exponent of all zeros. To multiply them by `2.0`, you need to operate on the fraction. And if the fraction becomes too large, they become normalized (in which case, you need to change the exponent too).
  - For normalized numbers, modifying the exponent should be enough, until you hit positive/negative infinity (in which case the fraction should change too).
  - Special values such as positive/negative infinity and NaN do not change when multiplied by `2.0`.
- `int floatFloat2Int(unsigned uf)` (4 points, 30 ops allowed): given the bit-level encoding of a `float`, return the `int` obtained after a `float` to `int` cast. This puzzle is also challenging! Keep in mind the following facts.
  - Truncation happens during the cast from `float` to `int`, so that `(int)1.9f == 1`, `(int)-1.9f == -1`, and `(int)0.01f == 0`.
  - So, you need to compute the integer part of the float from its sign/exponent/fraction fields and drop the rest. Remember the implicit `1.` before the fraction in the IEEE 754 encoding.
  - You also need to figure out when there is an overflow/underflow in the cast; in this case, you need to return `0x80000000u` (`MIN_INT`) because this is the value returned by the cast in C. You can tell if there is an overflow from the exponent: For which values of `exponent` is the following true? (There can be a few additional cases to cover.)

    ```
    abs_value =  1.(fraction) * 2^(exponent-127)  >  2^31-1  = MAX_INT
    ```

Evaluation

Your score will be computed out of a maximum of **25 points** including **15 correctness points** and **10 performance points**.

- *Correctness points.* The puzzles have been given a number of correctness points between 1 and 4, such that their weighted sum totals to 15. We will evaluate your functions using the `btest` program (described in the next section) to check for correctness. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, and no credit otherwise. Note that we will be running the `driver.pl` program to do the grading.

- *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 2 points for each correct function that satisfies the operator limit. You can use the `dlc` program (described in the next section) to check if your program satisfies these limits.

Autograding your work

We have included the following grading tools in the handout.

- `btest` checks the functional correctness of the functions in `bits.c`. Every time you modify `bits.c`, you should recompile it with `make btest` and run it as `./btest` to see how many correctness points you would receive. You can run `./btest -f funcName` to test only a specific function, or `./btest -f funcName -1 123 -2 456` to test `funcName(123, 456)`. You can ignore any warnings about `btest.c:528:9: warning: variable errors set but not used`.

- `dlc` 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 (allowed operators and maximum number of operations). You can run it as `./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. Run as `./dlc -e bits.c` to also print the number of operations used by each function. You can ignore any warnings about `/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includable file`.

- `driver.pl` uses `btest` and `dlc` to compute correctness/performance points and produce your final score (the instructors will use this program to evaluate your solution). Run it as `./driver.pl`.

Handin Instructions

Assignment collection will be automatic: right after the assignment deadline, our grading system will pull the most recent commit on the `master` branch of your repository. To ask for late days, fill out this form: we will grade the GitHub commit with the specified hash and use the form submission timestamp to compute the number of late days that you are taking.

Be sure to run `./driver.pl` and verify that your program passes not only the functional tests, but also the performance tests. The grade you see from `./driver.pl` will be the grade you get.

Advice

- Do not 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 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 */
}
```

- Run the `driver.pl` program! We'll be running the same program to determine your grade. You want to make sure it will work when we run it.

Acknowledgements. This lab was developed by the authors of the course textbook and their staff. It has been customized for use by this course.

CS356