CMPSC 311, Fall 2019, Project 1
Data Lab: Manipulating Bits
Assigned: Friday 9/13/2019, Due: Friday 9/27/2019, 11PM

Yanling Wang (`yuw17@psu.edu`) is the lead person for this assignment.

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 Logistics

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

3 Handout Instructions

Download `datalab-handout.tar` file from Canvas.

Start by copying `datalab-handout.tar` to a (protected) directory on a Linux machine (for exmaple, 311 directory in your home directory on W204 machines) in which you plan to do your work. Then give the command

```
unix> tar xvf datalab-handout.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`.

First thing you should do is to make sure that you modify the header of the file to add your name and psu userid by replacing the line:

```
 * &lt;Please put your name and userid here&gt;
```

with

```
 * Author: Yanling Wang
 * Email: yuw17@psu.edu
```

The `bits.c` file contains a skeleton for each of the 10 programming puzzles. 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:

```
 ! ~ & ^ | + << >>
```

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.

4 The Puzzles

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

4.1 Bit Manipulations

Table 1 describes a set of functions that manipulate and test sets of bits. 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 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.

Name	Description	Rating	Max Ops
`thirdBits(void)`	Return word with every third bit set to 1	1	8
`allEvenBits(x)`	Return 1 if all even-numbered bits are set to 1	2	12
`rotateLeft(x, n)`	Rotate x to the left by n bits	3	25
`bang(x)`	Computer !x without using !	4	12

Table 1: Bit-Level Manipulation Functions.

4.2 Two's Complement Arithmetic

Table 2 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
`isTmax(x)`	Return 1 if x is the maximum 2's complement int, 0 otherwise	1	10
`isNonNegative(x)`	Return 1 if `x >= 0`	2	6
`isLess(x,y)`	`x < y`?	3	24
`signMag2TwosComp(x)`	convert from sign-magnitude to 2's complement	4	15

Table 2: Arithmetic Functions

4.3 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 3 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
`floatIsEqual(uf, ug)`	Compute `f==g`	2	25
`floatUnsigned2Float(u)`	Compute bit-level equivalent of float(u)	4	30

Table 3: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

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:

```
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 50 points based on the following distribution:

26 Correctness points.

20 Performance points.

4 Style points.

Correctness points. The 10 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 26. We will evaluate your functions using the `btest` program, which is described in the next section. Make sure you use `dlc` or `driver.pl` that includes `dlc` to check that your code has followed the coding rules and compiles. **Code that doesn't pass the `dlc` test will get a zero.**

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 two points for each correct function that satisfies the operator limit.

Style points. We've reserved 4 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive. Other style requirement is the same as previous homework. Indentation with 2 spaces for each level; no more than 80 characters per line; good variable names; no compiler warnings, in particular pass the more restrictive compiler "dlc" provided. If you have added printf for debugging purposes in your code, please remove them in your final submission.

Beat the Professor contest. For fun, we're offering a "Beat the Prof" contest that allows you to compete with other students and the instructor to develop the most efficient puzzles. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for each puzzle are winners!

To submit your entry, you will have to have your code/solution on a W204 linux machine. The server for this contest can only accept submissions from W204 machine.

To submit your entry to the contest, type:

```
unix> ./driver.pl -u ``Your Nickname''
```

Nicknames are limited to 35 characters and can contain alphanumerics, apostrophes, commas, periods, dashes, underscores, and ampersands. You can submit as often as you like. Your most recent submission will appear on a real-time scoreboard, identified only by your nickname. If you are in psu network or connected to psu's VPN, You can view the scoreboard by pointing your browser at

```
http://cse-cmpsc311-7.cse.psu.edu:8080
```

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 bitXor
  ```

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

  ```
  unix> ./btest -f bitXor -1 7 -2 0xf
  ```

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

- **`dlc`: Make sure your code is accepted by dlc. Failure to comply to dlc compiler will result a ZERO for the project. See advice section for a common failure with dlc about declarations.** 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.

6 Handin Instructions

Handin your `bits.c` file only to Gradescope. Remember, your code will be autograded with `driver.pl` file included in the handout, plus the style points that will be graded by TA.

Your code will be checked by automated tool against each other's code and popular solutions you may find online. So please submit your work and only your work. We would like you to succeed with integrity. Violations will result in an academic integrity form filed to College of Engineering.

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 during the testing. Delete these debugging printfs for your final version.
- 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 */
}
```