CS 270 Systems Programming Spring 2021

Project 0: Bit-diddling

Out: *29 January 2021* ***Due:*** *12 February 2021*

1 Introduction

The purpose of this assignment is to help you become more familiar with bit-level representations of integers and floating point numbers, and the way they can be manipulated by various C (and C++) operators. 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.

This is an **individual project**. All handins are electronic, via Canvas. Clarifications and corrections will be posted on the page for this assignment. course Canvas page. **Do not look for solutions on the Internet.**

2 Setup

Start by logging into your VM and running the following command there:

```
unix> wget http://www.cs.uky.edu/˜calvert/cs270s21/p0-handout.tar
```

(You may want to create a separate directory for this project; you can do that with `mkdir proj0`. Then `cd proj0`.) Then do:

```
  unix> tar xvf p0-handout.tar.
```

This will cause a number of files to be unpacked in the current 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 14 programming puzzles. Your assignment is to fill in 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 further restrict this list. Also, you are not allowed to use any constants longer than 8 bits—that is, any constant in your code must be between 0 and 255 (inclusive). 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`.

Table **??** lists the puzzles in rough order of difficulty from easiest to hardest. 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
`isZero(x)`	return 1 if `x==0`, and 0 otherwise	1	2
`bitXor(x,y)`	`x^y` using only `&` and `~`	1	14
`bitOr(x,y)`	`x|y` using only `&` and `~`	1	8
`copyLSB(x)`	set all bits of result to least significant bit of `x`	2	5
`isNonNegative(x)`	return 1 if $x \geq 0$, and 0 otherwise	2	6
`getByte(x,n)`	extract byte `n` from word `x`	2	6
`isNotEqual(x,y)`	return 0 if `x==y`, and 1 otherwise	2	6
`oddBits()`	return word with all odd-numbered bits set to 1	2	8
`replaceByte(x,n,c)`	Replace byte `n` in `x` with `c`	3	10
`conditional(x,y,z)`	Same as `x ? y : z`	3	16
`addOK(x,y)`	return 1 if `x+y` will not overflow, else 0	3	20
`bang(x)`	Compute `!x` without using `!`	4	12
`bitReverse(x)`	Reverse bits in a 32-bit word	4	45
`floatUnsigned2Float(u)`	Return bit-level equiv. of `(float)u`	4	30

Table 1: Bit-diddling puzzles. Note that some problems have additional restrictionson operators.

There is one floating-point puzzles, which will require you to understand the bit-level representation of single-precision floating-point numbers, as well as the round-toward-zero convention of IEEE floating-point. For that puzzle *only*, 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.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow` 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.

4 Evaluation

Your score will be computed out of a maximum of 50 points based on the following distribution:

34 Correctness points.

14 Performance points.

2 Style points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4; the total of the ratings is 34. We will evaluate your functions using the `btest` program, which is described in the next section. You will get full credit equal to the rating 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 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 one point for each correct function that satisfies the operator limit. (**Note**: no points for an incorrect function that satisfies the operator limit.)

Style points. Finally, we've reserved 2 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.

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 4 -2 5
  ```

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 produces no output 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 Turning In Your Code

Upload your `bits.c` file as your submission for Project 0 in Canvas. **Do not include anything else.** Be sure that your code passes the `dlc` inspection.

6 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. 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 */
  }
  ```

- Remember that the logical operator `!` returns 0 if its operand is nonzero, and 1 if its operand is 0.
- Don't forget about sign extension when right-shifting signed values.