Bit Manipulation

Due at Friday midnight, April 17, 2020

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. Handout Instructions

Start by copying `sp_programming_hw01.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar xvf sp_programming_hw01.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`.

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:

```
! ˜ & ˆ | + << >>
```

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.

3. The Puzzles

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

1. Bit Manipulation and Two's Compliment Arithmetic

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.

Table 1. Bit-Level Manipulation and Arithmetic Functions

Name	Description	Rating	Max Ops
`subOK(x)`	Return 1 if all even-numbered bits are 1	3	20
`byteSwap(x, n, m)`	Swap the `n`th byte and the `m`th byte	2	25
`logicalShift(x, n)`	Count the number of 1's in `x`	3	20
`bitCount(x)`	Count of number of 1's in word	4	40
`isGreater(x, y)`	If `x > y` then return 1, else return 0	3	24
`fitsBits(x, n)`	If `x` can be represented as an `n`-bit return 1	2	15
`rotateLeft(x, n)`	Rotate `x` to the left by `n`	3	25

2. Floating-Point Operaions

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 2 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.

Table 2. Floating-Point Functions

Name	Description	Rating	Max Ops
`float_abs(f)`	Compute absolute value of `f`	2	10
`float_i2f(f)`	Compute `(float) f`	4	30
`float_twice(f)`	Return bit-level equivalent of expression `2*f`	4	30

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.

4. 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 bitAnd
```

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

```
unix> ./btest -f bitAnd -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
```

Your instructors will use `driver.pl` to evaluate your solution.

5. 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 */
}
```

6. Handin Instructions

Your program should be written in the **C language** that can be compiled in the Linux operating system. You will find a lot of different tests of the Linux operating systems. Compete the report in MS Word and upload onto “iCampus”. In the report you must include all of following. Name your file as **StudentID_yourName.docx**.