4471020, Fall 2021
Data Lab: Manipulating Bits
Assigned: Oct. 9, Due: Wed., Oct. 23, 11:59PM

System Programming TA: Namgyu Lee (`namtac2@naver.com`)
Please contact your TA directly for questions.

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(E-Ruri).

3 Handout Instructions

The handout also contains a Tar file (`datalab-handout.tar`). Please download it. Start by copying `datalab-handout.tar` to a (protected) directory on a Linux machine assigned to you(e.g., `Dakgalb` server). 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`.

The `bits.c` file contains a skeleton for each of the 14 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 lists the puzzles in rought 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
`bitAnd(x)`	`x & y` using only `|` and `~`	1	8
`anyEvenBit(x)`	Determine if any even-numbered bit in `x` is set.	2	12
`allOddBits(x)`	Determine if all odd-numbered bit in `x` is set.	2	12
`logicalShift(x,n)`	Implement the logical right shift.	3	20
`logicalNeg(x)`	Implement the `!` operator.	4	12
`bitCount(x)`	Count the number of 1's in `x`.	4	40

Table 1: Bit-Level Manipulation Functions.

4.2 Two's complement Arithmetic

For the two's complement puzzles, you will implement operations for two's complement integers that has a sign bit in MSB. Table 2 describes a set of functions that make use of the two's complement representations of integers. Again, please refer to the comments in bits.c and the reference version in tests.c for more information.

4.3 Floating-Point Operations

For the floating-point puzzles, 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

Name	Description	Rating	Max ops
`tmin()`	Most negative two's complement interger.	1	4
`isPositive(x)`	Determine if `x` is positive.	2	8
`sign(x)`	1 if `x` is positive, 0 if zero, and -1 if negative.	2	10
`addOK(x,y)`	Determine if `x` and `y` can be added together without overflow.	3	20
`isLess(x,y)`	Determine if `x < y`.	3	24

Table 2: Functions on the two's complement representation

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 on the bit-level representations of floating-point numbers. Refer to the comments in bits.c and the reference versionts in tests.c for more information as well.

Name	Description	Rating	Max ops
`floatAbsVal(uf)`	Compute absolute value of floating point value `x`.	2	10
`floatNegate(uf)`	Compute `-f`.	2	10
`floatFloat2Int(uf)`	Compute `(int)f`.	4	30

Table 3: Functions on the two's complement representation

Functions such as `floatAbsVal`, `floatNegate`, and `floatFloat2Int` must handle the full range of possible argument values, including not-a-number (NaN) and infinity.

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 63 points based on the following distribution:

35 Correctness points.

28 Performance points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 35. 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 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.

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 `b`test 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
  ```

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

6 Handin Instructions

If you are ready to submit your `bits.c` file, you can upload the file on E-Ruri. You don't need to hand in Tar file or something, just submit the '*bits.c*' which you wrote. Some students may use their grace(late) day for the lab. In this case, you couldn't upload your hand-in(`bits.c`) on E-ruri. If you use the grace day and then want to submit your hand-in late, please submit your hand-in by e-mail(Recipient: *both* wjsong@kangwon.ac.kr and namtac2@naver.com).

7 Advice

- **We're going to check code plagiarism with a specific tool. So, please don't cheat. Otherwise, you'll get the failing grade. NO EXCUSE.**
- 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 */
}
```