Operating Systems and C, Fall 2022
Data Lab: Manipulating Bits

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."

This is an individual project. Handins are electronic. Clarifications and corrections will be posted on Slack.

Instructions for setting up datalab and submitting your work are on github::

`https://github.itu.dk/OSC/22-Lab1-datalab`.

Datalab provides you with an environment that will make it possible for you to focus on bit manipulation, by editing a single file (bits.c) without the overhead of writing an entire C program. A self-correction tool is also provided to guide you through the assignment.

The `bits.c` file contains a skeleton for each of the 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.

2 The Puzzles

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

Table 1 lists the puzzles in 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
`bitXor(x,y)`	`x^y` using only `&` and `~`.	1	14
`tmin()`	Smallest two's complement integer.	1	4
`allOddBits(x)`	True only if all odd-numbered bits in `x` set to 1.	2	12
`negate(x)`	return `-x` without using the `-` operator.	2	5
`conditional(x, y, z)`	Same as `x ? y : z`	3	16
`isLessOrEqual(x, y)`	True if $x \leq y$, false otherwise	3	24
`logicalNeg(x))`	Compute `!x` without using the `!` operator.	4	12
`howManyBits(x)`	Min. nr. of bits to represent `x` in two's comp.	4	90

Table 1: Datalab puzzles.

3 Evaluation

Your score will be computed out of a maximum of 40 points based on the following distribution:

20 Correctness points.

16 Performance points.

4 Style 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 20. You can check your correctness points with `btest`.

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. You can test correctness using `dlc`.

Style points. Finally, we've reserved 4 points for a subjective evaluation of the style of your solutions and most importantly your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative and clear.

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
  ```

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

4 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 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 */
}
```