CS 356, Fall 2018
Data Lab (Part 1): Manipulating Bits
Due: Wednesday, Sep. 5, 11:59PM

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.

Project 2 will continue this assignment with more integer and some floating point puzzles.

2 Logistics

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

3 Handout Instructions

First, complete the Github form. A link is available on the *Assignments* webpage. At some point after you complete it, we will create your Github repos. **You will not be able to start this project until it happens - complete the form now to minimize the delay. Do not skip or delay this step by getting the files from a classmate.**

When we create your repo, you will find files in a `proj1` directory. Be sure to pull these file so you can begin.

To repeat, you need to complete the Github form immediately. No excuses related to git problems will be accepted if you are unable to correctly submit your work.

The only file you will be modifying and turning in is `bits.c`, although other files will help you, such as to find out what your grade will be.

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:

```
 ! ˜ & ˆ | + << >>
```

A few of the puzzles/functions may further restrict this list so read the comments of each puzzle/function carefully. 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
`bitAnd(x,y)`	`x & y` using only `|` and `˜`	1	8
`allEvenBits(x)`	return 1 if all even-numbered bits in word set to 1.	2	12
`bitMask(highbit, lowbit)`	Generate a mask consisting of all 1's between lowbit and highbit.	3	16

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.

5 Evaluation

Your score will be computed out of a maximum of 30 points based on the following distribution:

16 Correctness points.

Name	Description	Rating	Max Ops
`isTmax(x)`	Determine if `x` is the maximum 32-bit 2's comp. integer	1	10
`addOk(x,y)`	Compute `x + y`	3	20
`isGreater(x,y)`	Is `x` greater-than `y`?	3	24
`satMul2(x)`	Multiplies by 2, saturating to `Tmin` or `Tmax` if overflow	3	20

Table 2: Arithmetic Functions

14 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 16. 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.

Note that we will be running the `driver.pl` to do the grading.

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. In the past, we gave points for "style" - proper commenting, indenting, etc. There aren't any explicit points for style this semester, but be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted - your comments should be such that you can determine what your code does and why a few weeks later, if needed.

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 may ignore any warnings about:

  ```
  btest.c:528:9: warning: variable errors set but not used
  ```

```
[-Wunused-but-set-variable]
```

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.

  You may ignore any warnings about:

  ```
  /usr/include/stdc-predef.h:1: Warning: Non-includable file
  <command-line> included from includable file
  /usr/include/stdc-predef.h
  ```

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. Your instructors will use `driver.pl` to evaluate your solution. The typical usage is:

  ```
    unix> ./driver.pl
  ```

  Optionally, you may specify a nickname with the `-u` flag, which will run the driver and submit your score to bytes.usc.edu:3000, where you can view the leaderboard for the least amount of overall operations.

  ```
    unix> ./driver.pl -u <nickname>
  ```

6 Handin Instructions

Go to the class website and click on the submission link for this assignment. You will need to submit your github repo name and the SHA of your commit (the full SHA, not the 7-10 digit abbreviation). No submissions will be accepted via email and no re-submissions will be allowed due to incorrect SHAs. Click the ”Check My Submission” button once you enter your info to ensure your submission information was accurate.

Be sure to run `./driver.pl` and verify your program passes not only the functional tests but the performance tests. The grade you see from `./driver.pl` will be the grade you get (barring any submission errors).

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.
- 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 */
}
```

- Run the `driver.pl` before submitting your code. We'll be running that to determine your grade. You want to make sure it will work when we run it.

8 Acknowledgements

This lab was developed by the authors of the course text and their staff. It has been customized for use by this course.