THE UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

CSCI 2467, Fall 2020

The Data Lab: Manipulating Bits in C

Assigned: Tues., September 8, Due: Tues., September 22, 11:59PM

1 Introduction

The purpose of this assignment is to become more familiar with boolean algebra and bit-level representations of integer numbers. You'll do this by solving a series of programming "puzzles." After completing this assignment, you should understand more about machine-level representations at the bit level, as well as how to use the bitwise operations in the C programming language.

Sections `2.1` and `2.2` in Bryant & O'Hallaron's CS:APP textbook contain the background information needed to complete these puzzles. We highly recommended that you read these sections closely.

This is an individual assignment. You must hand in your own code and comments.

2 Instructions

Start by logging in to `https://autolab.cs.uno.edu` using your UNO credentials, then choose this course: **CSCI 2467**. From there, click **Data Lab** and then **Download handout**.

If you've done the above step using a browser in room 209 or 212 (HP terminal connected to systems-lab), all you need to do is move the tar file from your `Downloads` directory to your `2467` directory like this:

```
(systems-lab) $ mv ~/Downloads/datalab-handout.tar ~/2467
```

If you've downloaded `datalab-handout.tar` onto your own computer, you'll need to copy it onto `systems-lab`. You could use a graphical program like `Filezilla` or `WinSCP`. You can also use the command line. The example below assumes you've changed to the directory where your downloaded file is located (and of course, you'll replace UNOusername with yours):

```
(your own computer) $ scp datalab-handout.tar UNOusername@systems-lab.cs.uno.edu:~/2467
```

3 The 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.

This section describes the puzzles that you will be solving in `bits.c`. In the table below, the "Points" field gives the point value for the puzzle, which should correlate roughly with the puzzle's difficulty. The "Max ops" field gives the maximum number of operators you are expected to use to implement each function.

See the comment above each puzzle in `bits.c` for more details on the desired behavior of the functions. You may also refer to the test functions in `tests.c` if you wish. These are used as reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules you must follow.

Name	Description	Points	Max Ops
`bitOr(x,y)`	`x | y` using only `&` and `~`	1	8
`bitAnd(x,y)`	`x & y` using only `|` and `~`	1	8
`bitXor(x,y)`	`x ^ y` using only `&` and `~`	2	14
`isNotEqual(x,y)`	return 0 if `x == y`, 1 otherwise	2	6
`copyLSB(x)`	set all bits of result to least significant bit of x	2	5
`specialBits()`	return bit pattern `0xffca3fff`	2	6
`conditional(x,y,z)`	implement `x ? y : z` (ternary operator[1])	4	16
`bitParity(x)`	returns 1 if `x` contains an odd number of 0s	4	20
`minusOne()`	return a value of `-1`	1	2
`tmax()`	return maximum two's complement integer	1	4
`negate(x)`	return `-x` without negation operator	2	5
`isNegative(x)`	return 1 if `x < 0`, otherwise return 0	2	6
`isPositive(x)`	return 1 if `x > 0`, otherwise return 0	4	8
`bang(x)`	Compute `!n` without using `!` operator.	4	12
`addOK(x,y)`	Determine if `x+y` can compute without overflow	4	20
`absVal(x)`	absolute value of `x`	4	10
Total		40 pts	
† `byteSwap(x,n,m)`	swaps the n^{th} byte and the m^{th} byte	3	25
† `bitCount(x)`	returns count of number of 1s in word	4	40
† `logicalShift(x,n)`	shift x to the right by n, using a logical shift	3	20
Bonus		10 pts	

(† indicates *bonus* puzzles, we suggest you don't work on these until after the others.)

4 Evaluation

You must comment your code! Any puzzle solutions which are not accompanied by adequate comments will receive no credit.

The puzzles have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 40 (plus up to 10 bonus points for the additional puzzles at the end). 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`
- does not use illegal operators (as determined by `dlc`)
- contains one or more comments explaining *why* the solution works

Incorrect and illegal solutions will receive no credit, be sure to test with `btest` and `driver.pl` often!

Regarding comments: Please note the emphasis above on the word *why*. Comments which merely restate what the code does ("shift left by 31 bits", "add one") are not sufficient; these are obvious to your instructors who can also read C. Your comment must explain why you decided to perform that operation.

[1] Returns `y` if `x` is true (nonzero), returns `z` if `x` is false (zero).

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. These are the same tools the course instructor (and Autolab) will use to evaluate your work, so you will be able to anticipate your score by running these. (This is assuming you have adequately commented your code; if not, you will be penalized regardless of what Autolab says your score is)

- `btest:` This program checks the functional correctness of the functions in `bits.c`. To build and use it, type the following two commands:

  ```
  $ make
  $ ./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:

  ```
  $ ./btest -f bitAnd
  ```

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

  ```
  $ ./btest -f bitAnd -1 7 -2 0xf
  ```

  The line above calls your function with arguments as follows: `bitAnd(7, 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:

  ```
  $ ./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.

- `driver.pl:` This is a driver program that uses `btest` and `dlc` to compute the correctness points for your solution. It takes no arguments:

  ```
  $ ./driver.pl
  ```

5 Handin Instructions

You will hand in only the `bits.c` file using `https://autolab.cs.uno.edu`. (You will not create a tar file in this case.) You can do this using the Autolab web interface, or the *autolab-cli* (using the command `autolab submit bits.c`).

You may hand in to Autolab as often as you'd like. You may hand in partial solutions as you make progress.

A scoreboard is availble on Autolab as your classmates submit solutions. You can set a nickname on Autolab if you'd like to participate in a friendly competition, or you can simply leave your score anonymous (except to the course instructor).

6 Advice

- This is a time-consuming lab, you need to start early!
- Come to course help hours!
- Everyone needs help (see previous advice), but do **NOT** copy or look at anyone else's solutions, online or in person.
- 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.
- You must remove any `printf` statements you used for debugging before handing in your solution.
- Your code must work with both `btest` *and* `driver.pl`. See the note below if your code works with btest but generates errors when run with `driver.pl`.
- 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 */
}
```

  The way aroud this is to declare all variables at the top of the function before using them. (This used to be common.) For example:

```
int foo(int x)
{
  int a,b;   /* variables declared before using them */
  a = x;
  a *= 3;    /* Statement that is not a declaration */
  b = a;     /* Previously ERROR: Declaration not allowed here (now ok) */
}
```