CMSC 313, Spring 2011
Project 4: Manipulating Bits
Assigned: Mar. 29
Due: Wednesday., Apr. 6, 11:59PM

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers. 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. A Blackboard discussion board is also available for your questions.

3 Handout Instructions

Start by copying `datalab-handout.tar` from the course's public directory `\afs\umbc.edu\users\c\m\cmsc313\pub` to a (protected) directory on a Linux machine in which you plan to do your work. 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 nine `(9)` 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 and use of `unsigned` is strictly forbidden. You may not declare variables as `unsigned` or cast a variable to be `unsigned`. See the comments in `bits.c` for detailed rules and a discussion of appropriate 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
`anyEvenBit(x)`	returns 1 if any even numbered bit of `x` is 1	2	12
`negate(x)`	returns `-x`	2	5
`reverseBytes(x)`	reverses the order of the bytes in `x`	3	25
`rotateRight(x, n)`	Rotate `x` to the right by `n` bits	3	25

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.

Name	Description	Rating	Max Ops
`tmin( )`	returns the minumum two's complement integer	1	4
`isAsciiDigit(x)`	determines if `x` is an Ascii digit character	3	15
`subOK(x, y)`	determines if `x - y` will result in overflow	3	20
`absValue(x)`	returns the absolute value of `x`	4	10
`satAdd(x,y)`	returns TMAX or TMIN if `x + y` has positive or negative overflow	4	30

Table 2: Arithmetic Functions

5 Evaluation

Your score will be computed out of a maximum of 50 points based on the following distribution:

25 Correctness points.

18 Performance points.

7 Style points.

Correctness points. The 9 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 25. 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.

Style points. Finally, we've reserved 7 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

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 subOK
  ```

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

```
unix> ./btest -f subOK -1 26 -2 0xf
```

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.

6 Handin Instructions

- Remove any debugging and extraneous printing code from `bits.c`.
- To submit your `bits.c` file, use the command

  ```
  unix> make handin USERNAME=YourUserName
  ```

  where `YourUserName` is your UMBC email ID. For example, if your UMBC email is `bob@umbc.edu`, then use `bob` as your `YourUserName`

- If you want to resubmit your code, use the command

  ```
  unix> make handin USERNAME=YourUserName VERSION=2
  ```

  Increment the VERSION number for each subsequent submission.

- You can verify your handin by looking in
  `/afs/umbc.edu/users/c/m/cmsc313/pub/cmsc313_submissions/Proj4`
  You have list and insert permissions in this directory, but no read or write permissions.

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 C99 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 */
}
```