CS 210, Fall 2016
Programming Assignment PA1: Manipulating Bits
Due: Oct 4th, 2016 @ 1:30 PM

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

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course piazza site.

3 Handout Instructions

Start by downloading a copy of datalabhandout.tar from Homework section of

```
https://piazza.com/bu/fall2016/cs210/resources
```

Then copying `datalabhandout.tar` to a (protected) directory on one of the Linux machines ($csa2.bu.edu$ or $csa3.bu.edu$) in which you plan to do your work. Then issue the command

```
tar xvf datalabhandout.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 10 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 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
`bitXor(x, y)`	`x ^ y` using only `&` and `~`	1	8
`upperBits(n)`	Pads upper `n` bits with 1's	1	10
`leastBitPos(x)`	Return a mask that marks position of least significant 1 bit	2	6
`anyOddBit(x)`	Return 1 if any odd-numbered bit in word set to 1	2	12
`bang(x)`	Compute `!n` without using `!` operator.	4	12

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
`fitsBits(x, n)`	Does `x` fit in `n` bits?	2	15
`divpwr2(x, n)`	Compute $x/2^n$	2	15
`logicalShift(x, n)`	Logical right shift `x` by `n`	3	20
`satMul2(x)`	Multiplies by 2, saturating to Tmin or Tmax	3	20
`isNonNegative(x)`	`x >= 0`?	3	6

Table 2: Arithmetic Functions

5 Evaluation

Your score will be computed out of a maximum of 48 points based on the following distribution:

23 Correctness points.

20 Performance points.

5 Style points.

Correctness points. The 10 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 23. 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 5 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 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.

- **`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

Use gsubmit to submit your `bits.c` solutions file.

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- Create a team name of the form:
  - "*ID*" where *ID* is your login ID, if you are working alone, or
  - "ID_1+ID_2" where ID_1 is the login ID of the first team member and ID_2 is the login ID of the second team member.

  This should be the same as the team name you entered in `bits.c`.
- To handin your `bits.c` file, follow the gsubmit instructions to submit it under a directory named `DataLab-teamname`, where `teamname` is the team name described above.

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 */
}
```