CS354, Spring 2016
Data Lab: Manipulating Bits
Assigned: Feb. 12, Due: Wed., Feb. 26, 09:00 AM

Urmish Thakker (`uthakker@cs.wisc.edu`) is the lead person for this assignment.

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 and Piazza.

3 Handout Instructions

> **Please copy the file** `/p/course/cs354-common/public/src/datalab-handout.tar` **file to your private directory.**

Start by copying `datalab-handout.tar` to your private 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 the 8 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles (**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 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. Each function has a “Difficulty” field which gives the difficulty for the puzzle. You can find the difficulty of each problem in bits.c. The difficulty helps you plan things or helps you decide what problem to pick up first. The “Max Ops” fields (again found in bits.c) refers to the maximum number of operations allowed to complete a puzzle. Points will be deducted for solutions that exceed maximum number of operations. 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
getByte(x,n)	Get byte n from x.
isNotEqual(x,y)	Returns true if x is not equal to y
isEqual(x,y)	Returns true if x is equal to y

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
fitsBits(x,n)	Does x fit in n bits?
negate(x)	-x without negation
isPositive(x)	x > 0?
isNegative(x)	x < 0?
sign(x)	Returns 1 if x is positive, 0 if zero and -1 if negative?

Table 2: Arithmetic Functions

5 Evaluation

Your score will be computed out of a maximum of 120 points. The 8 puzzles you must solve are equally weighted. For each puzzle that you solve, you get a total of 13 points if it is correct. You additionally get 2 points if you are within the maxops limit. 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.

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 `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

Handin the `bits.c` solution file. Note it should be renamed to `cslogin-bits.c`. Eg, my cslogin is uthakker, and I would submit the file as *uthakker-bits.c*. Copy this file to /p/course/cs354-common/public/spring16.handin/cslogin/p2.

7 Advice/Notes

- 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 */
  }
  ```

- The tests.c should only be used as a reference to get an idea of what is the expected output of a function.