cs281: Introduction to Computer Systems

Project Lab 1: The Datalab

Assigned: Sept. 4, Due: Sept. 16

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of the integral data types and their bit-level operations. 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.

3 Handout Instructions

All the files you require for this assignment have been gathered together in a Unix "tar" file. This is an single archive file containing a set of files. On a Linux lab machine in Olin 219, you should run a web browser and navigate to the "Schedule" tab of the course web page. Then right-click the `datalab-handout.tar` link available in the *Homework* column.

Save the file to a directory specific to your cs281 work. If you left-click or download to somewhere else, you should start by copying `datalab-handout.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then, from the directory containing the tar file, give the command

```
  unix> tar xvf datalab-handout.tar
```

This will cause a number of files to be unpacked in a created datalab-handout 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 15 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 functions further restrict this list; you can see the legal operators in the comments for each function in the header comments in `bits.c`. Also, you are *not allowed to use any constants longer than 8 bits*. This means that an assignment like:

```
int m = 0xFE;
```

is legal, while one like:

```
int m = 0xFEEDBEEF;
```

is not. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style. In particular, you should take care to make all variable declarations *first* in each function, followed by the assignments and operations used to solve the puzzle.

4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`.

Table 1 describes a set of functions that manipulate and test sets of bits, as well as functions that use knowledge of two's complement representation to perform their function. 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
`tmin()`	Most negative two's complement integer	1	4
`bitOr(x,y)`	`x | y` using only `~` and `&`	1	8
`addOk(x,y)`	No overflow on addition of `x` and `y`	3	20
`conditional(x,y,z)`	same as `x ? y : z`	3	16
`anyEvenBit(x)`	return 1 if any even-numbered bit in x is set to 1.	2	12
`allOddBits(x)`	return 1 if all odd-numbered bits in x are set to 1.	2	12
`sign(x)`	return 1 if positive, 0 if zero, and -1 if negative.	2	10
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`bang(x)`	Compute `!x` without using `!` operator.	4	12
`rotateLeft(x,n)`	Rotate x to the left by n bits.	3	25
`fitsBits(x,n)`	Does `x` fit in `n` bits?	2	15
`tc2sm(x)`	Convert x from two's complement to sign-magnitude	4	15
`byteSwap(x,n,m)`	Swaps the nth byte and the mth byte of x	2	25
`leastBitPos(x)`	return mask marking position of least significant bit	2	6

Table 1: Puzzle Functions.

5 Evaluation

Your score will be computed out of a maximum of 70 points based on the following distribution:

34 Correctness points.

28 Performance points.

8 Style points.

Correctness points. The 14 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 34. 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 8 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

The only file I require from you is the bits.c file. At present, turning in your assignment simply involves sending the bits.c file to me as an attachment in an email. The send timestamp on the email will determine any lateness penalty, with anything after 11:59 pm on the due date incurring the mandatory 10% per day penalty.

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 */
  }
  ```

8 The "Beat the Prof" Contest

For fun, we're offering an optional "Beat the Prof" contest that allows you to compete with other students and the instructor to develop the most efficient puzzles. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for the set of puzzles are winners!

To submit your entry to the contest, type:

```
unix> ./driver.pl -u "Your Nickname"
```

Nicknames are limited to 35 characters and can contain alphanumerics, apostrophes, commas, periods, dashes, underscores, and ampersands. You can submit as often as you like. Your most recent submission will appear on a real-time scoreboard, identified only by your nickname. You can view the scoreboard by pointing your browser at

```
http://tashi:8080
```