CSCI247, Winter 2013
Data Lab: Manipulating Bits
Assigned: Jan 26, Due: Friday, Monday Feb.4, 11:30PM

Michael Meehan

1-23-2013

Ramesh Kumar (`Ramesh.Kumar@students.wwu.edu`) is your lab assistant and primary point of contact for this assignment but make use of the mentors in 162 and my office hours. As always, if you can't make my office hours email for an appointment.

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

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course Web page if needed. You must submit your completed work using the place provided on Moodle for "Submit Project 1."

NOTE: You should rename the bits.c file to "your-login-id-bits.c" before submitting it to moodle.

3 Handout Instructions

> **A file called `datalab-handout.tar` is available for download from Moodle under the heading "Project 1 tar file."**

Start by copying `datalab-handout.tar` to a (protected) directory on your home directory space from the file server from a Linux machine in the cf414 lab. The usual process is to make a directory for this

course as in ”mkdir cs367” Download the tar file into this directory. Chage the working directory to that directory as in ”cd cs367” 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` but remember to change the name to your-login-id-bits.c before submitting it. You can submit your work as many times as you wish and only the last one submitted will be graded.

The `bits.c` file contains a skeleton for each of the 13 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 may 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

The following table describes the functions that you are to implement. 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(void)`	Minimum Two’s Complement Integer using only ! & ^ | + << >> and ~	1	4
`isAsciiDigit(int x)`	Return 1 if 0x30 <= x <= 0x39	3	15
`conditional(int x, int y, int z)`	Same as x ? y : z using only ! & ^ | + << >> and ~	3	16
`allEvenBits(int x)`	Return 1 if all even-numbered bits in word set to 1.	2	12
`implication(int x, int y)`	Return x -> y using only ! ~ ^ |	2	5
`isLessOrEqual(int x, int y)`	if x <=y Return 1 else return 0	3	24
`bang(int x)`	Compute !x without using only !	4	12
`howManyBits(int x)`	Return the minimum number of bits need to represent x in 2’s complement	4	90
`byteSwap(int x, int n, int m)`	Swaps the nth byte and the mth byte	2	25
`leastBitPos(int x)`	Return a mask that marks the position of the least significant 1 bit	2	6
`logicalShift(int x, int n)`	Shift x to the right by n using logical shift	3	20
`float_neg(unsigned uf)`	Return bit level equivalent of -f for floating point argument f	2	10
`float_f2i(unsigned uf)`	Return bit level equivalent of expression `(int) f`	4	30

Table 1: Puzzle Functions.

The IEEE standard does not specify precisely how to handle NaN's, and the IA32 behavior is a bit obscure. We will follow a convention that for any function returning a NaN value, it will return the one with bit representation `0x7FC00000`.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
unix> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
unix> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

5 Evaluation

Your score will be computed out of a maximum of 66 points based on the following distribution:

35 Correctness points.

26 Performance points.

5 Style points.

Correctness points. The 13 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 41. 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 allEvenbits
  ```

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

  ```
  unix> ./btest -f implication  -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
  ```

  I will use `driver.pl` to evaluate your solutions.

6 Handin Instructions

Before handing in your solutions rename the `bits.c` file to your-login-name-bits.c. Then, submit it using the place provided on moodle for the DataLab Assignemnt Handin. You may resubmit your solution as many times as you want before the deadline of 11:30 on Febuary 4th. Submissions will be cut off precisely at that time by Moodle.

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. This is similar to the rules in the language Pascal. All declarations must precede any "executable statement." 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 */
}
```