CSci2021, Spring 2015
Data Lab: Manipulating Bits
Assigned: Jan.26, Due: Feb.06, 23:55

Ran Hu (`huxxx662@umn.edu`) is the lead person for this assignment.

The intent of this lab is to make you more familiar with bit representations and bit operators in C. The assignment involves solving a series of programming puzzles in which you will be restricted from using certain operations. These puzzles will get you thinking deeply about bit representation and manipulation, and how to use them to get better performance in Real World applications. All questions can be directed primarily to the TAs. Feel free to ask questions on the lab forum or contact us. Our goal is to maximize your understanding of the class materials.

1 Logistics

This is an **individual project**. All handins are electronic. This lab includes the grading program that will be used to grade your submissions. You should be able to know your approximate grade before submitting your program.

2 Handout Instructions

Start by copying `datalab-handout.tar` 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` which is in the `datalab-handout` directory.

---

bits.c —— File with function outlines, **The only file you are asked to modify**

bits.h —— Header for bits.c

tests.c —— Tests that will be used to check your versions

btest.c —— Source for grading program

btest.h —— Header for grading program

decl.c —— Declarations of functions

dlc —— Binary that checks for rules compliance, compiled to Linux Intel

driver.pl —— Autograding driver program

ishow.c —— Helper that shows the bit representation of arbitrary integer numbers.

fshow.c —— Helper that shows the bit representation of arbitrary floating point numbers.

README —— Extra details

Makefile —— Makefile for building/testing your work

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:

```
 ! ˜ & 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.**

Any other files in your submission will be overwritten with our own versions for grading. Don't count on us using anything other than the code you write in bits.c!

Grades will be based on the results of running `driver.pl` which utilizes the `btest` and `dlc` binaries. `btest` will give a score based on the correctness of your result. However that score will be reduced for violation of rules, as reported by `dlc`. Also, note that any partial credit reported by btest may differ from the actual partial credit that you end up receiving for your submission.

3 Submission Instructions

For this lab, you will submit only one file: "**bits.c**". Submit your .c file on Moodle site before the deadline. Do not rename the file. Put your name and X500 id at the top of the bits.c file, using comment. For example, change `<Please put your name and userid here>` to `<Ran Hu, huxxx662>`.

Make sure that your bits.c file is ready to submit by running driver.pl one last time before you submit. If your file could only be graded by `btest`, but cannot pass the `driver.pl`, you will still get no points.

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
`bitNor(x,y)`	`~(x | y)` using only `~` and `&`	1	8
`anyOddBit(x)`	return 1 if any odd numbered bit of x set to 1.	2	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
`isTmax(x)`	return 1 if x is the maximum, two's complement number	1	10
`negate(x)`	`-x` without negation	2	5

Table 2: Arithmetic Functions

4.3 Floating-Point Operations

For this part of the assignment, you will implement some common single-precision floating-point operations. In this section, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having type `unsigned`, and any returned floating-point value will be of type `unsigned`. Your code should perform the bit manipulations that implement the specified floating point operations.

Table 3 describes a set of functions that operate on the bit-level representations of floating-point numbers. Refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Functions `float_twice` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. 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 when the argument is NaN, return the argument.

Name	Description	Rating	Max Ops
`float_twice(uf)`	Computer `2*f`	4	30

Table 3: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

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 100 points based on the following distribution:

69 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. Easy, Moderate, Hard and Tricky problems are worth 1, 2, 3 and 4 points respectively. The maximum attainable score for correctness is 29 points which will be scaled up to 69 points by the grader using the following computation ((score * 69) / 29). We will evaluate your functions using the `btest` program, which is described in subsequent sections. You will get full credit for a puzzle if it passes all of the tests performed by `btest`.

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 very 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.

Some Preparation for the CSELabs machines

Before you start to test the grading or compiling tools, you may need to change some settings if you are using the CSELabs machines.

- Every time you open a new terminal on the CSELabs machines, you must run the following command before compiling your code:

```
unix> module unload soft/gcc
```

-If you get permission denied for the use of '.pl' files. You must change the access permissions to file system objects (files and directories) using the command under the directory of the '.pl' files:

```
unix> chmod 700 *.pl
```

After the preparation, you are now ready for the grading commands:

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 bitNor
  ```

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

  ```
  unix> ./btest -f bitNor -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 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 */
  }
  ```

Some puzzles are substantially more difficult than others. Start by working on ones that seem easier, and make sure you get those right before moving on to trickier problems.

Use the programs that are included! You should know ahead of time what your score is going to be.

Some (not all) of the problems will be easier if you can think of them as not just bit manipulation, but as mathematical problems.

A deep understanding of two's complement arithmetic will be very useful.