CSE 238/2038/2138 Systems Programming
Project 1
Data Lab: Manipulating Bits
Due: 12.02.2017 11:59PM

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 using the Autolab service. (Autolab will be explained soon).

3 Handout Instructions

You will need the `datalab-handout.tar` for this project, which you can find within the course web site.

`datalab-handout.tar` includes 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 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 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	14
`allOddBits(x)`	Are all odd-numbered bits set to 1?	2	12
`isAsciiDigit(x)`	Is x an ASCII digit?	3	15
`conditional(x, y, z)`	Same as C's "x ? y : z"	3	16
`logicalNeg(x)`	Compute !x 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
`tmin()`	Return smallest two's complement integer	1	4
`negate(x)`	`-x` without negation	2	5
`isTmax(x)`	Is x the largest 32-bit two's complement integer?	2	10
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`howManyBits(x)`	Compute minimum number of bits required to represent x	4	90

Table 2: Arithmetic Functions

4.3 Floating-Point Operations

For this part of the assignment, you will implement some common single-precision floating-point opera- tions. 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.

Name	Description	Rating	Max Ops
`float_i2f(x)`	Compute `(float)x`	4	30
`float_f2i(uf)`	Compute `(int)f`	4	30
`float_twice(uf)`	Computer `2.0*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.

```
unix> ./fshow 0x15213

Floating point value 1.212781782e-40
Bit Representation 0x00015213, sign = 0, exponent = 0x00,
  fraction = 0x015213 Denormalized. +0.0103172064 X 2^(-126)

unix> ./fshow 15.213

Floating point value 15.2130003
Bit Representation 0x41736873, sign = 0, exponent = 0x82,
  fraction = 0x736873 Normalized. +1.9016250372 X 2^(3)
```

5 Evaluation

Your score will be computed out of a maximum of 63 points based on the following distribution:

37 Correctness points.
26 Performance points, based on number of operators used in each function.

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 37. 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.

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 bitXor
```

This will call the `bitXor` function many times with many different input values. You can feed `btest` specific function arguments using the option flags `-1`, `-2`, and `-3`:

```
linux> ./btest -f bitOr -1 7 -2 0xf
```

This will call `bitxOr` exactly once, using the arguments `x=7` and `y=15`. Use this feature if you want to debug your solution by inserting `printf` statements; otherwise, you'll get too much output.

- **`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.

6 Handin Instructions

Unlike other courses you may have taken in the past, in this course you may handin your work as often as you like until the due date of the lab.

To receive credit, you will need to upload your `bits.c` file using the Autolab option "Handin your work." Each time you handin your code, the server will run the driver program on your handin file and produce a grade report (it also posts the result on the scoreboard). The server archives each of your submissions and resulting grade reports, which you can view anytime using the "View handin history" option.

Handin Notes:

- At any point in time, your most recently uploaded file is your official handin. You may handin as often as you like.
- Each time you handin, you should use the "View your handin history and scores" option to confirm that your handin was properly autograded. Manually refresh the page to see the autograded result.
- You must remove any extraneous print statements from your `bits.c` file before handing in.

7 Advice

- Test and debug your functions one at a time. Here is the sequence we recommend:

  Step 1. Test and debug one function at a time using `btest`. To start, use the `-1` and `-2` arguments in conjunction with `-f` to call one function with one specific set of input argument(s):

  ```
  unix> ./btest -f isLess -1 23 -2 0xabcd
  ```

  Feel free to use `printf` statements to display the values of intermediate variables. However, be careful to remove them after you have debugged the function.

  Step 2. Use `btest -f` to check the correctness of your function against a large number of different input values:

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

  If `btest` detects an error, it will print out the specific input argument(s) that failed. Go back to Step 1, and debug your function using those arguments

  Step 3. Use `dlc` to check that you've conformed to the coding rules:

  ```
  unix> ./dlc bits.c
  ```

  Repeat Steps 1–3 for each function.

- Some hints for `dlc`:
  - 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 */
}
```