EECS-213, Fall 2017
Data Lab: Manipulating Bits
Assigned: Oct. 2, Due: Mon., Oct. 16, 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   IMPORTANT: Notes on Academic Integrity

It is very important to come up with you own solutions. You may discuss among yourselves but copying code from others or any online source is strictly forbidden. Before we do any grading, we will run your submissions through MOSS, an automatic tool for source code plagiarism detection. This means that if you cheat, we **WILL** catch you and report you to the dean.

3   Logistics

This is a group project. All handins are electronic through Canvas. Clarifications and corrections will be posted on Canvas or Piazza.

4   Handout Instructions

For this assignment you will need to download `datalab-handout-f17.tar` from Canvas. Start by copying `datalab-handout-f17.tar` to a (protected) Linux directory in which you plan to do your work. Then give the command

```
unix> tar xvf datalab-handout-f17.tar
```

* Note that " unix> " is merely indicating you should be working on a UNIX based machine. It is not part of the command you need to type. This applies to all commands in this lab and in future labs.

* If you are unfamiliar with working under a UNIX/Linux command line environment, please come to the office hours or refer to these quick guides on basic UNIX commands (`http://mally.stanford.edu/~sr/computing/basic-unix.html`) and VIM (`https://scotch.io/tutorials/getting-started-with-vim-an-interactive-guide`).

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 14 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.

5 The Puzzles

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

5.1 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
`allEvenBits(x)`	Returns 1 if all even-numbered bits in word set to 1	2	12
`copyLSB(x)`	Set all bits of result to least significant bit of `x`	2	5
`bitMask(h,l)`	Generate a mask consisting of all 1's between `l` and `h`	3	16
`isPower2(x)`	Returns 1 if `x` is a power of 2, and 0 otherwise	4	20
`logicalNeg(x)`	Implement the `!` operator, using all of the legal operators except `!`	4	12

Table 1: Bit-Level Manipulation Functions.

5.2   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)	Returns 1 if x is the maximum two's complement number, 0 otherwise	1	10
fitsBits(x,n)	Return 1 if x can be represented as an n-bit, two's complement integer	2	15
isLess(x,y)	If x < y then return 1, else return 0	3	24
absVal(x)	Absolute value of x	4	10
isNonZero(x)	Check whether x != 0 using the legal operators except !	4	10

Table 2: Arithmetic Functions

5.3   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.

Name	Description	Rating	Max Ops
float_abs(uf)	Compute $\|f\|$	2	10
float_neg(uf)	Compute $-f$	2	10
float_i2f(x)	Compute (float) x	4	30

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

Functions `float_abs` and `float_neg` 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 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.

6 Evaluation

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

38 Correctness points.

28 Performance 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 38. 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 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.

7 Handin Instructions

One person in your group should submit your bits.c file to Canvas. Please write the **names and netids of both group members** in line 4 of bits.c. Multiple submissions are allowed, but only the last one will be graded. Also please make sure all submissions are under the same Canvas account. If you update your solutions after the deadline, the most recent one will be graded but penalties will apply.

8 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.

5

- 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 */
}
```