CS 213, Winter 2022
Data Lab: Manipulating Bits

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." Some of these puzzles are artificial, but you'll find yourself thinking much more about bits in working your way through them. Other puzzles reflect what hardware actually does to implement common operations. Finally, a few puzzles reflect algorithms that are important at all scales, from chip to cloud.

2 Logistics

You may work in a group of up to two people in solving the problems for this assignment. All handins are electronic. Clarifications and revisions will be posted to the course discussion group.

3 Handout Instructions

You will need the file `datalab-handout.tar`, which you will find in the `~cs213/HANDOUT` directory on the class server:

- `moore.wot.eecs.northwestern.edu`

This directory will also be visible on the Wilkinson machines. Generally, `moore` is the best place to work, as it is where we will be testing your code.

Please note that you will hand in your work on `moore`, which is also where it will be tested.

You will need a (protected) directory on a Linux machine in which to do your work. We recommend that you simply use a class server for this. You can create a protected directory like this:

```
unix> cd ~/
unix> mkdir mydatalab
unix> chmod 700 mydatalab
unix> cd mydatalab
```

The `chmod` command will set things so that only the owner of the directory can read, write, or cd into it.

Next, give the command

```
unix> tar xvf ~cs213/HANDOUT/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`.

The `bits.c` file contains a function skeleton for each of the 13 programming puzzles that you will solve. The lab tests each of your solutions for correctness, and for compliance with the set of low-level C operators you are allowed to use and how many you are allowed to use (the *coding rules*.)

Please be sure to read the `README` file.

4 The Puzzles

The puzzles in `bits.c` break down into *integer puzzles* (Sections 4.1) and *floating point puzzles* (Section 4.2). The rules are different for these.

4.1 Integer Puzzles

In the integer puzzles you must 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.

You may find the `ishow` tool to be helpful. [1]

Two's Complement Arithmetic

Table 1 describes a set of functions that make use of the two's complement representation of integers. 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 the function. Refer to the comments in `bits.c` and `tests.c` for more information. `test.c` contains reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules for your functions.

[1] You may also find an 2's complement calculator to be helpful, such as:
`https://www.omnicalculator.com/math/twos-complement`

Name	Description	Rating	Max Ops
`tmin()`	Return the minimum two's complement integer	1	4
`isTmax(x)`	Return 1 if `x` is the maximum two's complement number, else 0	1	10
`negate(x)`	Return `-x`	2	5
`isGreater(x,y)`	If `x > y` then return 1, else return 0	3	24

Table 1: Arithmetic Functions

Bit Manipulations

Table 2 describes a set of functions that manipulate and test sets of bits. 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`.

Name	Description	Rating	Max Ops
`bitOr(x,y)`	Return `x | y`	1	8
`bitAnd(x,y)`	Return `x & y`	1	8
`allOddBits(x)`	Return 1 if all odd-numbered bits in `x` are set to 1	2	12
`getByte(x,n)`	Extract byte `n` from word `x`	2	6
`rotateRight(x,n`	Rotate `x` to the right by `n`	3	25
`bitParity(x)`	Returns 1 if `x` contains an odd number of 0's	4	20

Table 2: Bit-Level Manipulation Functions.

4.2 Floating Point Puzzles

In the floating point puzzles 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
`floatAbsVal(uf)`	Return absolute value of `f`, handling all cases	2	10
`floatInt2Float(x)`	Return `(float)x`, handling all cases	4	30
`floatScale4(uf)`	Return `4*f`, handling all cases	4	30

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

All functions must handle the full range of range of possible argument values, including zeros, infinities, and not-a-numbers (NaNs). The IEEE standard does not specify precisely how to handle NaN's, and the x64 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`.[2]

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.[3]

5 Evaluation

Your score will be computed out of a maximum of 60 points based on the following distribution:

30 Correctness points.

26 Performance points.

4 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 30. 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 learn enough about data representations to be able to get the right answer for these exercises. However, we also 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.

[2] This is the "first positive quiet NaN".

[3] You may also find a floating point calculator to be helpful, such as:
`https://www.h-schmidt.net/FloatConverter/IEEE754.html`

Style points. Finally, we've reserved 4 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. You should test your code with `driver.pl`.

- **`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. They will do so on moore.

- **`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.

6 Handin Instructions

To hand in your work, you will need to be on `moore`.

To hand in your work, you should first include a comment at the beginning of your `bits.c` file that indicates your names and NetIDs. The comment should also indicate any issues you are aware of.

Next, you should create a copy of your file that includes your NetIDs, in alphabetical order, and separated by hyphens, in the name:

```
unix> cp bits.c bits-netid1-netid2.c
```

Finally, copy this file to the class hand-in directory:

```
unix> cp bits-netid1-netid2.c ~cs213/HANDIN/datalab
```

This last command will only work correctly on `moore`.

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 */
}
```