Assigned: Tuesday, October 24, 2017
Due: Friday, November 3, 2017, 6:00 PM

CS-2011, Machine Organization and Assembly Language, B-term 2017

Data Lab: Manipulating Bits

1. Introduction

The purpose of this assignment is to help you to become more familiar with bit-level representations of integers and floating point numbers. You'll do this by solving a series of 15 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. While this assignment has its own server at **`cs2011.cs.wpi.edu`**, final submissions must be electronic via *Canvas* system. In addition, see below for a "Beat the Prof" contest.

Submissions to the contest and the assignment server *are not a substitute* for submissions via *Canvas*. Clarifications and corrections will be posted on the course Web page.

3. Handout Instructions

All of the files for this lab assignment are contained in a single Linux **`tar`** file. There are two versions of this file:–

`datalab-CCC-handout.tar` — specifically for use during the October 25 Recitation session
`datalab-b17-handout.tar` — for use on your Linux virtual machine

The problems and solutions of the two versions are identical, and either one may be used for submission to *Canvas*. However, the CCC version cannot communicate with the "Beat the Prof" server.

Either file may be downloaded from *Canvas*. This will require your WPI username and password.

Save your selected file to a working directory on the CCC system or your Linux virtual machine. Then expand your selected **`tar`** file using the appropriate one following commands:–

```
linux> tar xvf datalab-CCC-handout.tar
linux> tar xvf datalab-Ubuntu-handout.tar
```

Either one will create a folder/directory called **`datalab`** and cause a number of files to be unpacked into that folder. The only file you will be modifying and submitting is **`bits.c`**.

Before you start:– Insert your *name* and your *WPI username* in a comment at the top of the **`bits.c`** file. The penalty for failing to include your name on *any file of any submission* in this course is 5 points per file.[1]

The **`bits.c`** file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton using only *straight-line* code for the integer puzzles (i.e., no loops or conditionals, except where specified) and a limited number of C arithmetic and logical operators. Specifically, you are *only* allowed to use the following eight operators (except where specified):–

`! ~ & ^ | + << >>`

A few of the puzzles 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
`replaceByte(x, n, c)`	Replace byte **`n`** in **`x`** with **`c`**	3	10
`fitsBits(x,n)`	Return **`1`** if **`x`** can be represented as an **`n`**-bit two's complement integer	2	15
`bitParity(x)`	Return **`1`** if **`x`** contains an odd number of **`0`**'s	4	20
`logicalNeg(x)`	Implement the **`!`** operator, using all of the legal operators except **`!`**	4	12
`greatestBitPos(x)`	Return a mask that marks the position of the most significant **`1`** bit. If **`x == 0`**, return **`0`**	4	70

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.

[1] It does not count to include your name and/or username as part of the name of a file or folder, because files can easily become detached from their names.

Name	*Description*	*Rating*	*Max Ops*
`tmax()`	Return maximum two's complement integer	1	4
`isNotEqual(x, y)`	Return **`0`** if **`x == y`**, and **`1`** otherwise	2	6
`isPower2(x)`	Return **`1`** if **`x`** is a power of **`2`**, **`0`** otherwise	4	20
`rotateLeft(x, n)`	Rotate **`x`** to the left by **`n`**	3	25
`rempwr2(x, n)`	Compute **`x%(2^n)`**, for **`0 <= n <= 30`**	3	20
`conditional(x, y, z)`	Return **`x ? y : z`**	3	16

Table 2: Two's Complement 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 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_neg(uf)`	Return bit-level equivalent of expression **`-f`** for floating point argument **`-f`**	2	10
`float_f2i(uf)`	Return bit-level equivalent of expression **`(int) x`**	4	30
`float_twice(uf)`	Return bit-level equivalent of **`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`.

All three of these floating point functions 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:

```
linux> make
```

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

```
linux> ./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 79 points based on the following distribution:–

44 Correctness points.

30 Performance points.

5 Style points.

Correctness points. The 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 79. 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 cleverer than that. Therefore, each function is specified with a maximum number of operators that you are allowed to use. 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 comments. The (human) graders will be looking for solutions that are as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

6. 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:

  ```
  linux> make
  linux> ./btest
  ```

  Note 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:

  ```
  linux> ./btest -f bitXor
  ```

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

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

  Consult 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:

  ```
  linux> ./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:

```
linux> ./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.

Caution: Whereas **gcc** allows declarations to be placed anywhere within a block (i.e., within curly brackets **{}**), **dlc** requires that all declarations be at the *beginning* of the block, before any statements.

Another Caution: Make sure that **dlc** passes *every* puzzle of this lab. If not, the autograder will not run, and your project will not earn *any* performance points.

- **driver.pl:** This is a driver program that uses **btest** and **dlc** to compute the correctness and performance points for your solution. Your instructors will use **driver.pl** to evaluate your solution.

7. Submission Instructions

Please copy your **bits.c** file to a file named **<userName>-bits.c**, where **<userName>** is your WPI username. For example, if your WPI e-mail address is wsmith2@wpi.edu, your file name *must* be

wsmith2-bits.c

It is essential that you rename your file as instructed, so that the autograder can identify your file and return the appropriate grade. The penalty is 25% of the total project points for not naming your file correctly, thereby requiring the TAs to manually rename your file.

Submit this file using the web-based *Canvas* system. This assignment is called *Datalab*.

8. Special Requirement, Notes, and Advice

- This is an *individual assignment*. You may discuss general strategy for solving any puzzle, but you may not copy *or even look at* anyone else's code, including solutions found online.[2]
- Do *not* 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 at the top of its block* — i.e., what you

[2] In a previous year, several students were found to have identical code for a number of the puzzles, all copied from the same online source.

enclose in curly braces, and before any statement that is not a declaration. For example, it will complain about the following code *but it won't tell you what the problem is*:–

```
int foo(int x) {
  int a = x;  // Declaration with initialization -- OK
  a *= 3;     // Statement that is not a declaration
  int b = a;  // ERROR: Declaration not allowed here
}
```

9. The "Beat the Prof" Contest

For fun, we offer an optional "Beat the Prof" contest that allows you to compete with other students and the instructor to develop the most efficient puzzles. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for each puzzle are winners!

To submit your entry to the contest, type the following from your Data Lab working directory:–

```
linux> ./driver.pl -u "Your_Nickname"
```

Nicknames are limited to 35 characters and can contain alphanumeric characters, apostrophes, commas, periods, dashes, underscores, and ampersands. You can submit as often as you like. Your most recent submission will appear on a real-time scoreboard, identified only by your nickname. You can view the scoreboard by pointing your browser at

http://cs2011.cs.wpi.edu:8080

Submitting your solution to the "Beat the Prof" contest is *not a substitute* for submitting it to *Canvas*. That is, the contest server keeps track of scores by nickname, but it does not keep copies of **`bits.c`** files. The graders need the **`bits.c`** files (appropriately named) in order to award grades.

Note: As of this writing, the "Beat the Prof" contest can be accessed from the Ubuntu virtual machines but *not* from the CCC systems. (On the CCC systems, everything seems to work fine except for the **`./driver.pl`** script, which gets compilation errors.)

Note 2: You are strongly encouraged to submit your solution to the "Beat the Prof" contest *every time that you complete or update a puzzle.*