

College of William and Mary
Computer Science Department
CSCI 304: Computer Organization (Spring 2016)
Data Lab: Manipulating Bits (Due 04/10/16)

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 submissions are electronic.

3 Handout Instructions

You have two file attachments associated with this lab: `datalab-handout.tar` and this file named `datalab.pdf`. Start by copying `datalab-handout.tar` to a protected directory. To have such a protected directory, let's assume you have created a directory named `csci304` under your home directory on your Computer Science Lab machines. After `cd`-ing to the `csci304` directory, you can use the command (assuming the hypothetical name for the machine you are logged on is `unix`)

```
unix> chmod go-rwx .
```

Note the `."` that refers to the current directory as command-line argument. After you have copied the file `datalab-handout.tar` in this directory, type the command,

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

which will create a sub-directory called `datalab-handout` and cause a number of files to be unpacked in that sub-directory. The only file you will be modifying in that sub-directory and turning in is `bits.c`.

The `bits.c` file contains a skeleton for each of the 14 programming puzzles, in which they all return a dummy value of 2. 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
<code>bitNor(x, y)</code>	<code>x & y</code> using only <code>~</code> and <code>&</code>	1	8
<code>getByte(x, n)</code>	Get byte <code>n</code> from <code>x</code> .	2	6
<code>logicalShift(x, n)</code>	Shift right logical.	3	20
<code>conditional(x, y, z)</code>	Same as <code>x ? y : z</code> .	3	16
<code>bang(x)</code>	Compute <code>!n</code> without using <code>!</code> 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
<code>tmin()</code>	Most negative two’s complement integer	1	4
<code>negate(x)</code>	<code>-x</code> without negation	2	5
<code>fitsBits(x, n)</code>	Does <code>x</code> fit in <code>n</code> bits?	2	15
<code>divpwr2(x, n)</code>	Compute $x/2^n$	2	15
<code>isNegative(x)</code>	<code>x < 0</code> ?	2	6
<code>isNonZero(x)</code>	<code>x != 0</code> ?	4	10

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.

Name	Description	Rating	Max Ops
<code>float_neg(uf)</code>	Compute $-f$	2	10
<code>float_f2i(x)</code>	Compute $(\text{int})\ x$	4	30
<code>float_twice(uf)</code>	Compute $2*f$	4	30

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

Functions `float_neg` and `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 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.

5 Evaluation

Your score will be computed out of a maximum of 50 points based on the following distribution:

72 Correctness points: The 14 puzzles you must solve have been given a difficulty rating between 1 and 4. You will get a credit of twice the difficulty rating for correctly solving a puzzle, such that their weighted sum for all correct solutions totals to 72. 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.

28 Performance points: While the main objective is that you can get the right answer, 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 use more clever approaches. 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.

5 Style points (bonus): 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. These points will be awarded over the entire puzzle set, not on individual puzzles.

Note that you can retain your bonus (up to 5% above maximum score), even if it exceeds the full points score and use it for compensating for other components of your total score.

Autograding your work

We have included some autograding tools in the handout directory, namely `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 “`texttt./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. The appropriate scaling will then be used on the correctness score to determine your actual score. It takes no arguments:

```
unix> ./driver.pl
```

Your instructor/grader will use `driver.pl` to evaluate your solution.

6 Hand-in Instructions

The only file you need to submit on Blackboard for grading is your modified `bits.c` solution file. Please do not send as email attachment to the instructor or the graders. Before handing in, make sure that you use `make` command to build and test using `btest` your `bits.c` file intended for submission, on a CS Lab machine, even if you decide to develop your solutions on your own Linux installation (physical or virtual machine).

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 */
}
```