

## Assignment 2: Data Lab

Due: Tuesday, February 5, 2019 at 11:59pm

The purpose of this assignment is to give you familiarity with bit-level representations of integers and floating point numbers. You will accomplish the goal by solving a series of programming “puzzles.” Even though many of the puzzles are quite artificial, you will find yourself thinking much more about bits in working your way through them.

You must work in a group of two people in solving the problems; partners will be assigned for this assignment. *We strongly recommend that you and your partner brainstorm before coding!*

### Getting Started

The materials for the data lab are available on the course web page.

First ssh into `project5.cs.pomona.edu`. Then copy/download `datalab-handout.tar` to a (protected) directory in which you plan to do your work. Next give the command<sup>1</sup>

```
% tar xvf datalab-handout.tar
```

which will cause a number of files to be unpacked in the directory. The only file you will be modifying and submitting is `bits.c`.

Begin by opening the file in an editor and put your names and userids in the comments at the top of the `bit.c` file. Do this right away!!

The `bits.c` file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code (no loops or conditionals) and a limited number of C arithmetic and logical operators. Each function heading tells you what operations are allowed. Further, 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.

### Compiling the Code

We strongly suggest that you do your work on `project5`. You can be sure that the support programs `btest` and `d1c` will work there. In any case, make sure that the version you turn in compiles and runs correctly on `project5`. If it fails to compile there, we cannot grade it.

Check the file `README` for documentation on running the compiler `d1c` and the `btest` program. You will find it helpful to work through the functions one at a time, testing each one as you go. You can use the `-e` flag to instruct `d1c` to count the number of operations you use (in addition to checking for disallowed operations). Once you pass the tests with `d1c`, you can test your function with `btest`. Note: you can use the `-f` flag to instruct `btest` to test only a single function, as in `./btest -f bitAnd`.

Dig more deeply into the `README` file for information on some helper programs.

---

<sup>1</sup>A note on color-coding: In a few cases we display commands that you type in a terminal window and the resulting output. We use `%` for the prompt. The characters that you are to type are in green, and the system’s responses are in blue.

We have given you a Makefile to ease the burden of running the compiler. Type

```
% make btest
```

to compile the program `btest`, or simply

```
% make
```

to compile everything.

## The `d1c` Program

The `d1c` program, a modified version of an ANSI C compiler, will be used to check your programs for compliance with the coding style rules. The typical usage is

```
% ./d1c bits.c
```

- Type `./d1c -help` for a list of command line options. The `README` file is also helpful.
- The `d1c` program runs silently unless it detects a problem.
- Do not include `<stdio.h>` in your `bits.c` file, as it confuses `d1c` and can result in some non-intuitive error messages.
- Running with the `-e` switch causes `d1c` to print counts of the number of operators used by each function.
- ANSI C, and hence `d1c`, disallows `//` comments.
- In ANSI C, you must make all variable declarations at the beginning of a function. The following code is not accepted by `d1c`.

```
int mask = 0x55 + (0x55 << 8);
mask = mask + (mask << 16);
int shift = (x >> 1);
int sum = (shift & mask) + (x & mask);
```

- You may ignore the warning about a “non-includable file.”

## Evaluation

Your code will be run and tested on `project5`. Your score will be computed out of a maximum of 75 points. Each function will be evaluated separately for correctness and performance.

- **Correctness (40 points):** We will use the programs `driver.pl` and `d1c`, supplied with the laboratory materials, to evaluate your code. No points will be given for a function if `d1c` reports an illegal operator, too many operators, or another error.
- **Performance (30 points):** We will use the programs `driver.pl` and `d1c`, supplied with the laboratory materials, to evaluate your code. No points will be given for a function if `d1c` reports an illegal operator, too many operators, or another error.
- **Style (5 points):** Your `bits.c` file will be evaluated by the graders and given up to 3 points for style. For this laboratory, “good style” is easy to attain. It means that your names are present at the top of the file, that your code is understandable and consistently indented, that comments—when necessary to explain—are present and easy to read, and that there is no extraneous material. An additional 2 points will be awarded for submitting a completed feedback file.

## Submission Instructions

When you have finished, submit two files, `bits.c` and `feedback.txt`, to the course submission page.

- Make sure you have included all your team members' names in your files.
- Use all the team members' names when submitting.
- Remove any extraneous print statements before submitting the file.
- Use the submission system, linked from the course web page, to submit the file `bits.c`.
- If you discover a mistake in your code, simply submit the file again.

## Part I: 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.

Function `bitAnd` computes the bitwise and function. You may only use the operators `~` and `|`. Function `bitXor` should duplicate the behavior of the operation `^`, using only the operations `&` and `~`.

Function `isNotEqual` compares `x` to `y` for inequality. As with all *predicate* operations, it should return 1 if the tested condition holds and 0 otherwise. Function `copyLSB` replicates a copy of the least significant bit in all 32 bits of the result. Function `conditional` returns `y` if `x` is true and `z` otherwise.

Function `logicalShift` performs logical right shifts. Function `rotateLeft` rotates the bits to the left by `n`. You may assume the shift/rotation amount `n` satisfies  $0 \leq n \leq 31$ .

Function `bang` computes logical negation without using the `!` operator.

Name	Description	Rating	Max Ops
<code>bitAnd(x,y)</code>	<code>x&amp;y</code> using only <code>~</code> and <code> </code>	1	8
<code>bitXor(x,y)</code>	<code>^</code> using only <code>&amp;</code> and <code>~</code>	1	14
<code>isNotEqual(x,y)</code>	<code>x != y?</code>	2	6
<code>copyLSB(x)</code>	Set all bits to LSB of <code>x</code>	2	5
<code>conditional(x,y,z)</code>	<code>x ? y : z</code>	3	16
<code>logicalShift(x,n)</code>	Logical right shift <code>x</code> by <code>n</code>	3	16
<code>rotateLeft(x,n)</code>	Rotate <code>x</code> to the left by <code>n</code>	3	25
<code>bang(x)</code>	Compute <code>!x</code> without using <code>!</code> operator	4	12

Table 1: Bit-Level Manipulation Functions.

## Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Function `tmax` returns the largest integer.

Function `isNonNegative` determines whether `x` is less than or equal to 0.

Function `addOK` determines whether its two arguments can be added together without overflow.

Function `absVal` is equivalent to the expression `x<0?-x:x`, giving the absolute value of `x` for all values other than `TMin`.

Function `isPower2` determines whether `x` is a power of 2.

Name	Description	Rating	Max Ops
<code>tmax(void)</code>	largest two's complement integer	1	4
<code>isNonNegative(x)</code>	<code>x &gt;= 0?</code>	3	6
<code>addOK(x,y)</code>	Does <code>x+y</code> overflow?	3	20
<code>absVal(x)</code>	absolute value	4	10
<code>isPower2(x)</code>	Is <code>x</code> a power of 2?	4	20

Table 2: Arithmetic Functions

### Part III: Float Arithmetic

Table 3 describes a set of functions that make use of single precision floating point representation of integers. For these puzzles, you may use

Function `float_neg` returns the argument multiplied by `-1`.

Function `float_f2i` casts a float to an int.

Name	Description	Rating	Max Ops
<code>float_neg(f)</code>	Returns <code>-f</code>	2	10
<code>float_f2i(f)</code>	Returns <code>(int) f</code>	4	30

Table 3: Arithmetic Functions

### Part IV: Feedback

Create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

How you answer these questions **will not affect your grade**, but whether you answer them will.