CS341L, Fall 2025
Data Lab - part b: Manipulating Bits using some tools
Assigned: 9/14/25, Due: Monday 9/29/2025 at 11:59PM

Soraya Abad-Mota (`soraya@unm.edu`) is the lead person for this assignment.

1 Introduction

The purpose of part b of the Datalab assignment continues to be to become more familiar withbit-level representations of integers and floating point numbers. Your task is to solve six (6) more programming "puzzles", which are harder than the ones you solved for part a.

As stated before, this semester we are trying something different in the way we generate code in this lab for the harder problems. So we have divided it into two parts.

Part b: Consists of building an experimental setting to find solutions to the more challenging puzzles and evaluate those solutions. Make sure you read this whole document before you start your work. The details of what you must handin are summarized on page 6. Pages 7 and 8 have content that you already read for part a.

2 Logistics

This is an individual project. All handins are electronic on Canvas. Clarifications and corrections will be posted on the course Canvas page. Document any people with whom you discussed the assignment privately in a file in your directory entitled **CREDITS**. If you consult any outside source that helped you find a solution you must also include that in the file CREDITS.

3 Handout Instructions

The *Datalab part b* source files are available in Canvas for this course in a .tar file. These source files are also available in the CS machines in the path specified on canvas. The source files are similar to the ones provided for part a, the only difference is the bits.c file which has the signature of each of the 11 functions presented in this document. You will be working with six (6) out of these 11, you may pick which 6 you are working with. We recommend that you work with lobogit, git, or with another control version software, to keep your work properly backed up.

The `bits.c` file contains a skeleton for each of the programming puzzles for which you will provide a solution. Your assignment is to pick six (6) functions to complete from the pool of 11 described in this document.

All solutions that you submit must be restricted to the rules specified here and in the skeleton `bits.c` provided, i.e. use 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:

```
 ! ˜ & ˆ | + << >>
```

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than eight (8) bits (1 byte). See the comments in `bits.c` for detailed rules and a discussion of the desired coding style. In fact, read those comments thoroughly before you begin.

4 The Puzzles

This section describes the 11 puzzles from which you will pick 6 to solve in `bits.c` for this *part b of the Datalab*. There are some restrictions on which 6 puzzles you pick; at least two (2) of the puzzles must be from table 3, and the other four (4) may be from tables 1 and 2.

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
`greatestBitPos(int x)`	return a mask that marks the position of the most significant 1 bit. If x == 0, return 0	4	70
`leftBitCount(x)`	returns count of number of consecutive 1's in left-hand (most significant) end of word	4	50

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
`fitsBits(x,n)`	Does `x` fit in `n` bits?	2	15
`howManyBits(x)`	return the minimum number of bits required to represent x in two's complement	4	90
`satAdd(int x, int y)`	adds two numbers but accounts for + and - overflow	4	30
`satMul2(int x)`	mult. by 2, saturating to Tmin or Tmax if overflow	3	20
`satMul3(int x)`	mult. by 3, saturating to Tmin or Tmax if overflow	3	25

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
`float_half(uf)`	Return bit-level equivalent of expression 0.5*f	4	30
`float_i2f(int x)`	Return bit-level equivalent of expression (float) x	4	30
`trueThreeFourths(x)`	Times $3/4$ avoid overflow errors	4	20
`trueFiveEighths(x)`	multiplies by 5/8 rounding toward 0, avoiding errors due to overflow	4	25

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

Remember that at least two (2) puzzles from this table 3 must be included in the 6 puzzles that you will work with in this part of the project.

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 Designing a solution and generating the code for each puzzle

Steps to produce solutions for the puzzles selected for this *Datalab part b*.

1. Study the description of the eleven puzzles contained in this document and in the skeleton `bits.c` provided, and decide which six (6) puzzles you will work on. Remember that in section 4 some restrictions on how to select the 6 puzzles were specified.
   You will answer a quiz this week, to mark which puzzles you have selected to work on.

2. For each puzzle, you must understand what the function is supposed to do, think about how to approach a solution to it by breaking it down into simpler tasks, and how will the individual tasks work together to provide the return value. **Document this *design* for each of the functions, in a single document and submit it in pdf format.**

3. Generate code for each function using one of two options: the *Do-it-yourself* and the *Use-genAI-tools*. These two options are described below in sections 5.1 and 5.2. What to submit for each option:

   Do-it-yourself : for this option, in addition to the design, you only need to submit your completed `bits.c` with the code for each puzzle properly commented.

   Use-genAI-tools : for this option you will perform two experiments, analyze the code generated for each function in each experiment, and provide a reflection on your experience with the genAI tool for this task. More details in section 5.2.

In the quiz were you will pick your 6 puzzles, you will also specify which of the two options you prefer.

5.1 Do-it-yourself

If you don't want to use a generative AI tool, you may work on these puzzles on your own as you did for part a; since these are harder puzzles, we ask you to first design the solution before coding it.

5.2 *Use-genAI-tools*

Plenty has been said about the ability of *Generative Artificial Intelligence (genAI)* tools to produce code. Solutions for simple tasks have been successful, but harder problems have not produced good code and require human intervention for debugging and correcting. These tools are evolving very rapidly and solutions to harder problems have improved significantly.

If you are using this option, you may **select any one of these three tools** to work with: *Claude, coPilot,* or *chatGPT 5* (apparently you have to explicitly say that it is a hard problem, for the chatbot to really use the chatGPT 5 "machinery", otherwise it uses earlier versions). All of these tools are available for free. *Copilot* is offered to all students, faculty, and staff at UNM. Pick only one of these to do your experiments, the same tool for all the functions. Answer in the quiz, which tool you will be using if you selected the *Use-genAI-tools* option.

You will **execute two experiments prescribed** for this option.

Experiment Y: use the chatbot to generate code for each function; you will do this by providing the prompts that describe your own design and tell it that it must follow that design. Save the code generated and the prompts given.

Run `btest` on the code for each function and make a note of the score obtained, the errors produced, and any other relevant information for you to understand the behavior of the code generated.

Experiment Z: Ask the genAI tool you are using to first break the problem without implementation. Save the results of the "design" generated by the tool. Compare the design produced with your own, write down your comparison. Then, ask the genAI tool to implement each piece that it has "designed"; make sure you separate the design from the actual implementation, even though both will be made by the genAI tool. And make sure that you tell it to use that design.

Run `btest` on the code generated for each function and make a note of the score obtained, the errors produced, and any other relevant information for you to understand the behavior of the code generated.

For each experiment, Y or Z, you need to document the following:

1. Prompts used. In addition to the description of each function and your own design for it (for Experiment Y), you need to remember to provide the restrictions of each function, in terms of maximum number of operators, forbidden C constructs, etc.
2. GenAI tool responses.
3. Your own reflection on the design (when produced by the genAI tool) and on the code generated. Make sure you can follow the design on each.

6 Evaluation of the option *Do-it-yourself*

If you do not want to use genAI tools for this portion, you may provide your own solution, but it has to be your own, not borrowed from somebody/something else.

Your score will be computed based on which puzzles you pick, and it will include the following aspects:

A Correctness. The puzzles you must solve have been given a difficulty rating between 2 and 4. We will evaluate your functions using the `btest` program. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, and no credit for this aspect otherwise.

B Performance. 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.

C Design and style points. Each solution to a puzzle must come from the design you did as explained in bullet 2 of the *Steps to produce solutions*, at the beginning of section 5. On both options for this part b of the project, you will have to first provide a design for the solution of each puzzle. So, this will be evaluated the same way for both options.

We might ask you to provide the explanations in a session with the TA or the instructor. Or you may get a question in an exam about one of your solutions.

7 General Rubric for the two options

(3 pts) You picked 6 puzzles according to the restrictions, puzzles picked outside of the required ones will not count and will not be graded.

(42 pts) You broke each of the 6 puzzles by yourself into a design of components that must work together; this requires understanding the problem and designing a solution; the design involves describing several simpler tasks and how they work together to return the expected value for each function.

(55 pts) Evaluated depending on which option you selected, i.e. the *Do-it-yourself* option of the *Use-genAI-tools* option.

8 Handin Instructions for this *Datalab Part b*

Submit the following at the appropriate place for this assignment on Canvas.

1. You will upload a pdf document with your design for all six (6) puzzles you worked with.
2. If your option is the *Do-it-yourself*, You will upload your final version of `bits.c` (plain text) and a CREDITS file in pdf format into Canvas, if applicable.
3. If you selected the *Use-genAI-tools* option, you will submit all the documentation for the two experiments, Y and Z, each one will contain their own bits.c file, call them, `bitsY.c` and `bitsZ.c`, respectively.

9 Autograding the solutions to the puzzles

We have included the same autograding tools that were provided for part a of this project, these are in the handout directory — `btest`, `dlc`, and `driver.pl` — with them you can check the correctness of the solutions. You may also run the code with your own test cases if you want.

- **`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 correctness and performance for each solution. It takes no arguments:

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

10 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 aFunc(int x)
{
  int a = x;
  a *= 3;     /* Statement that is not a declaration */
  int b = a;  /* ERROR: Declaration not allowed here */
}
```