CS281, Fall 2025
Data Lab: Manipulating Bits

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers. 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. Often, hardware designers are after the smallest and fastest design that will solve a problem, even if the design is more confusing to understand. This is in contrast to the ethos of writing "good code" in which readability is often more important than slight improvements in speed which add a high degree of confusion. These puzzles may require obscure, but fast and efficient solutions to solve problems which can normally be solved very easily in regular code.

2 Logistics

You will complete this project as a team. Your team will be preassigned (randomly selected). One person from the team is designated as the team captain. Their responsibility is to allocate the workload evenly among the other team members and also to set a rigid schedule that allows for completion and testing of the lab. The team MUST share the work effort.

All handins are electronic. Your team captain will submit one bits.c file for the whole group.

3 Handout Instructions

Obtain the `datalab-handout.tar` file from the canvas page.

Start by copying `datalab-handout.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar -xvf 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`.

Each of you should download and untar this file so that you each have access to the code individually.

The `bits.c` file contains a skeleton for each of the 10 programming puzzles. 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.

Stop now and read the comments in `bits.c` for detailed rules and a discussion of the desired coding style. You will need to fully understand these before you begin solving puzzles.

4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`. Note there are 10 puzzles listed, but we will do one of them together in class as examples. Thus you will have 9 puzzles remaining to solve on your own.

Table 1 lists the puzzles in roughly order of difficulty from easiest to hardest. 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	8
`bitAnd(x,y)`	`x \& y` using only `|` and `~`.	1	8
`tmin()`	Smallest two's complement integer	1	4
`bitMatch(x,y)`	Return 1 for all the bits that match.	1	14
`implication(x,y)`	Return logical implication x implies y.	2	5
`negate(x)`	Return `-x` with using - operator.	2	5
`isPositive(x)`	return 1 if x greater than 0, return 0 otherwise	2	8
`isPallindrome(x)`	return true if x is a palindrome, false otherwise	4	45
`bang(x)`	Compute !x without using !	4	12
`absVal(x)`	Compute —x—	4	10

Table 1: Datalab puzzles.

5 Evaluation

Your score will be computed out of a maximum of 100 points based on the following distribution:

60 Correctness points (6 pts per puzzle).

20 Performance points (2 pts per puzzle).

20 Style points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4. 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 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 minimizes the operator limit.

Style points. Finally, we have 20 points for documenting your solutions. You MUST FULLY describe the design logic of each puzzle using generous comments. These puzzles are tricky and their solution (from just looking at code) is not obvious. It is your responsibility to fully explain the logic behind the steps of your solutions by using comments in each of your functions. You should also add all of your team members names in the global comment block at the top, and also add invididual team members names to each function that they worked on.

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:

  ```
  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 bitXor
  ```

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

  ```
  unix> ./btest -f bitXor -1 4 -2 5
  ```

Check the file `README` for documentation on running the `b`test 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 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.

6 Handin Instructions

Submit your `bits.c` file in canvas. This is the only file you will turn in.

7 Advice

- These puzzles are tricky, especially the higher rated ones. Be sure to spread your work over the time period alloted so that you have plenty of time to not only find a solution, but to also think deeply about improving your solutions once they are complete. There may be multiple ways to solve each puzzle, some with a lower number of operators!

- 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 */
}
```