

Assignment #1: DataLab I (due on Sun, Jan. 27, 2019 by 11:59pm)

Introduction

The purpose of this assignment is to become familiar with the bit-level representation of integers and their operations. You'll achieve 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 while working your way through them.

Assignment #2 will continue this assignment with more integer and some floating point puzzles.

Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course Piazza page. Remember that:

- Any time you receive help from a TA/CP, you should acknowledge that in your code with a comment starting with the string "assistance from".
- You are not allowed to search for help online!
- You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.
- Reconsideration requests must be made within one week of our release of grades for the assignment.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why a few weeks later, if needed.

Handout Instructions

First, complete the form announced on Piazza: we will create a private GitHub and share it with you. (You must submit the form first! Do not skip or delay this step by getting the files from a classmate.)

When we create your repository, you will find files in a `proj1` directory. Be sure to clone the GitHub repository inside the class VM and pull these files.

The only file you will be modifying and turning in is `bits.c`, although other files will help you, such as to find out what your grade will be. The `bits.c` file contains a skeleton for each of the 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**) 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 puzzles/functions may further restrict this list so read the comments of each puzzle/function carefully. 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.

The Puzzles

Bit Manipulations

- `int bitOr(int x, int y)` (1 point, 6 ops allowed): compute `x | y` using only `~` and `&`.
- `int oddBits(void)` (2 points, 8 ops allowed): return an `int` with all odd-numbered bits set to 1.
- `int leastBitPos(int x)` (2 points, 6 ops allowed): return an `int` where the only bit set to 1 is at the position of the least-significant, nonzero bit of `x`.

Two's Complement Arithmetic

- `int isPositive(int x)` (2 points, 8 ops allowed): return 1 if `x > 0` and 0 otherwise (no comparison operators allowed!)
- `int isLess(int x, int y)` (3 points, 24 ops allowed): return 1 if `x < y` and 0 otherwise (no comparison operators allowed!)

- `int` subtractionOK(`int` x, `int` y) (3 points, 20 ops allowed): return 1 if x-y does not overflow (e.g., subtractionOK(0x80000000, 0x80000000) == 1 but subtractionOK(0x80000000, 0x70000000) == 0).
- `int` ezThreeFourths(`int` x) (3 points, 12 ops allowed): return x*3/4 (i.e., multiply by 3/4 rounding toward zero, replicating overflow behavior).

Evaluation

Your score will be computed out of a maximum of **30 points** including **16 correctness points** and **14 performance points**.

- *Correctness points.* The puzzles have been given a number of correctness points between 1 and 3, such that their weighted sum totals to 16. We will evaluate your functions using the btest program (described in the next section) to check for correctness. You will get full credit for a puzzle if it passes all of the tests performed by btest, and no credit otherwise. Note that we will be running the driver.pl program to do the grading.
- *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 2 points for each correct function that satisfies the operator limit. You can use the dlc program (described in the next section) to check if your program satisfies these limits.

Autograding your work

We have included the following grading tools in the handout.

- btest checks the functional correctness of the functions in bits.c. Every time you modify bits.c, you should recompile it with `make btest` and run it as `./btest` to see how many correctness points you would receive. You can run `./btest -f funcName` to test only a specific function, or `./btest -f funcName -1 123 -2 456` to test funcName(123, 456). You can ignore any warnings about `btest.c:528:9: warning: variable errors set but not used`.
- dlc 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 (allowed operators and maximum number of operations). You can run it as `./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. Run as `./dlc -e bits.c` to also print the number of operations used by each function. You can ignore any warnings about `/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includable file`.
- driver.pl uses btest and dlc to compute correctness/performance points and produce your final score (the instructors will use this program to evaluate your solution). Run it as `./driver.pl`.

Handin Instructions

Assignment collection will be automatic: right after the assignment deadline, our grading system will pull the most recent commit on the master branch of your repository. To ask for late days, [fill out this form](#): we will grade the GitHub commit with the specified hash and use the form submission timestamp to compute the number of late days that you are taking.

Be sure to run `./driver.pl` and verify that your program passes not only the functional tests, but also the performance tests. The grade you see from `./driver.pl` will be the grade you get.

Advice

- 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 in a block 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 */  
}
```

- Run the driver.pl program! We'll be running the same program to determine your grade. You want to make sure it will work when we run it.

Acknowledgements. This lab was developed by the authors of the course textbook and their staff. It has been customized for use by this course.

CS356 