ECEn 324
Data lab: Manipulating Bits

Introduction

The purpose of this lab is to become more familiar with bit-level representations and manipulations. 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.

Logistics

You may work in a group of up to two people in solving the problems for this lab. The only "hand-in" will be electronic. Any clarifications and revisions to the lab will be posted on the course Web page.

Hand Out Instructions

Stop! If you do not have an account in the CAEDM labs, then please get one here. While you are free to complete the labs anywhere, we can't guarantee that everything will run smoothly. When in doubt, try it on a SPICE machine in the CAEDM lab.

The datalab-handout.tar contains the code you will need for this lab.

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

The file `btest.c` allows you to evaluate the functional correctness of your code. The file `README` contains additional documentation about `btest`. Use the command `make btest` to generate the test code and run it with the command `./btest`. The file `dlc` is a compiler binary that you can use to check your solutions for compliance with the coding rules. The remaining files are used to build `btest`.

Looking at the file `bits.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. Do this right away so you don't forget. (Be sure to use your Route Y ID in the appropriate field.)

The `bits.c` file also contains a skeleton for each of the 15 programming puzzles. Your assigment is to complete each function skeleton using only *straightline* code (i.e., no loops conditionals, or switches) 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 8 bits. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

Evaluation

Your code will be compiled with GCC and run and tested on one of the class machines. Your score will be computed out of a maximum of 76 points based on the following distribution:

41 Correctness of code running on one of the class machines.

30 Performance of code, based on number of operators used in each function.

5 Style points, based on our subjective evaluation of the quality of your solutions and your comments.

The 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 41. We will evaluate your functions using the test arguments in `btest.c`. You will get full credit for a puzzle if it passes all of the tests performed by `btest.c`, half credit if it fails one test, and no credit otherwise.

Regarding 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 function that satisfies the operator limit.

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.

Puzzles

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. Complete descriptions for each function are given in the `bits.c` file included in the `datalab-handout.tar` file. The descriptions include the restricted set of operators that can be used in each function.

Table 2 describes a set of functions that make use of the two's-complement representation of integers. Like Table 1, this table includes columns for the "Rating" and "Max Ops" for each function. Complete

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	`&` using only `~` and `|`	1	8
`bitNor(x,y)`	`~(x|y)` using only `&` and `~`	1	8
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`evenBits(void)`	even bits set to 1	2	8
`logicalShift(x,n)`	Logical right shift `x` by `n`	3	16
`bang(x)`	Compute `!x` without using `!` operator	4	12
`leastBitPos(x)`	Mark least significant 1 bit	4	6

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`isNotEqual(x,y)`	`x != y?`	2	6
`negate(x)`	return -x	2	5
`isPostive(x)`	1 if x > 0	3	8
`isNonNegative(x)`	`x >= 0?`	3	6
`sum3(x,y,z)`	x+y+z with one '+'	3	16
`addOK(x,y)`	Does `x+y` overflow?	3	20
`abs(x)`	absolute value	4	10
`isNonZero(x)`	1 if x nonzero	4	10

Table 2: Arithmetic Functions

descriptions are found in the `bits.c` file with the restricted set of operators that can be used in each function.

Advice

You are welcome to do your code development using any system or compiler you choose. Just make sure that the version you turn in compiles and runs correctly on the SPICE linux machines in the CAEDM lab. If it doesn't compile, we can't grade it.

The `dlc` 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

```
./dlc bits.c
```

Type `./dlc -help` for a list of command line options. The README file is also helpful. Some notes on `dlc`:

- The `dlc` program runs silently unless it detects a problem.
- Don't include `<stdio.h>` in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages.

Check the file `README` for documentation on running the `btest` program. 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, e.g., `./btest -f isPositive`.

Hand In Instructions

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- To handin your `bits.c` file, type:

  ```
  ./submit.pl
  ```

  This will check your solutions in bits.c, email the results with your bits.c file to the instructor, and update the class webpage. Results on the class webpage are identified by team name.
- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  ./submit.pl
  ```

  This will overwrite your previous solution and update the results. We only grade your most recent submission.