CS 203L, Fall 2017, Lab 5
Data Lab: Manipulating Bits and Integers

September 30, 2017

1 Goals

- Find a solution for each puzzle, and
- Explain how your solution performs its task.

The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers. 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.

2 Handout Instructions

Start by downloading `datalab-handout.tar` to a 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`.

The `bits.c` file contains a skeleton for each of the 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.** See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

3 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`.

3.1 Bit Manipulations

In your `bits.c` file, provided with each function is a header providing information about that problem. 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.

3.2 Two's Complement Arithmetic

In your `bits.c` file, provided with each function is a header providing information about that problem. There are several 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.

In your `bits.c` file, provides a header providing information about that problem. Refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

4 Evaluation

As usual, Lab 5 will be worth 20 points. The distribution of points is as follows:

Correctness 60% (12 points):

Correctness will be determined by the auto-grading of the problems.

Explanations 20% (4 points):

In the C source file, at the top of each problem solution, write in comments a brief discussion about the problem and solution. Include descriptions of what approach was taken for the solution and why, and any interesting observations about the resulting answers. This set of comments should be brief but cogent.

Style/Commenting 20% (4 points):

In addition to the Explanations, the code should be commented in-line so that it is immediately clear what each instruction or set of instructions is doing. The style also refers to the care taken to make the source code readable (i.e. indenting, spacing between lines, etc.)

Autograding your work

We have included some autograding tools in the handout directory — `btest` 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 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 `b`test program.

- **`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.

- **`dlc`:** If want to work with `dlc` directly, there are several options. Below shows the basic options, to find more use the `-options` option.

  ```
  $ ./dlc -h
  Usage: ./dlc [options] [file]

  Parses <file> as a C program, reporting syntax and type errors, and writes
  processed C program out to <file>.p.c.  If <file> is omitted, uses
  standard input and standard output.

  General Options:
  -help              Print this description
  -options           Print all options
  -copy              Print the copyright information
  ```

```
-v                  Print version information
CS:APP Data Lab Options:
-e                  Emit operator count for each Data Lab function
-z                  Zap illegal Data Lab functions
-Z                  Like -z, but also zaps functions with too many operators
Warning Options:
-ansi               Disable GCC extensions and undefine __GNUC__
-W<n>               Set warning level; <n> in 1-5. Default=4
-Wall               Same as -W5
-il                 Ignore line directives (use actual line numbers)
-offset             Print offset within the line in warnings/errors
-name <x>           Use stdin with <x> as filename in messages
```

5 Handin Instructions

Your submission will be by submitting the `bits.c` file , but please add your name to the file before submission. With your last name first, an underscore, and then the "`bits.c`". An example of this would be "`liew_bits.c`".

6 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.
- Remember this is not number of lines being graded, instead it is number of operations.
- It pays, at least during debugging, to break each step into a single line so you can see what is happening. If you wish to combine lines later, that is OK, but break the problem into smaller parts when you are attempting to develop a solution. Then when you have the solution, explain what each line is doing.
- Start with the low rating problems first, this is an indication of how hard the problem is and will get you warmed-up.