CMSC 311
Spring 2010
Lab 1: Bit Slinging

Tim Meyer

January 31, 2010

Post questions on this assignment to the CMSC 311 discussion board `http://www.elms.umd.edu/`.

Introduction

The purpose of this assignment 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 are expected to work alone on this assignment. The only "hand-in" will be electronic. All code will be in C (ISO C99). Any clarifications and revisions to the assignment will be posted on the CMSC 311 discussion board `http://www.elms.umd.edu/`.

Hand Out Instructions

We will be using the linuxlab machines for this assignment. Log in to `linuxlab.cs.umd.edu` and download the assignment file (`datalab-handout.tar`) from BlackBoard (`https://elms.umd.edu`) to a (protected) directory in which you plan to do your work. (Directories on linuxlab machines under your home directory should be protected by default)

Then give the command: `tar xzvf 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 (AND SHOULD) 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 yourself. Do this right away so you don't forget. Use your directory ID for the team name and login ID. Remember, **this is an individual project**.

The `bits.c` file also contains a skeleton for each of 17 programming puzzles. Your assignment is to complete each function skeleton using only *straight-line* code (i.e., no loops, conditionals, or function calls) and a limited number of C arithmetic and logical C operators. Specifically, you are *only* allowed to use the following eight operators:

```
! ~ & ^ | + << >>
```

Many 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 the submit server. Your score will be computed out of a maximum of 80 points based on the following distribution:

45 Correctness of code running on the submit server; each puzzle is worth its difficulty rating in points.

34 Two points for each correct puzzle that does not exceed the maximum number of operators.

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

Notice that there are 4 extra points built in for your benefit. Your grade will still be calculated out of 80.

The 17 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 45. 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`.

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. 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 (and none otherwise). ALSO: YOU WILL GET NO POINTS FOR THE FUNCTION IF YOU DID NOT MAKE AN ATTEMPT AT A SOLUTION. Partial credit will be at the discretion of the grader, and is different for each puzzle. Comments detailing your thoughts would be a good idea if you have not passed some of the tests.

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. Do not remove/replace the exiting comments in the code, but add comments on the code that you add.

The functions you will implement

1. `bitAnd` : `(x & y)` using only `~` and `|`
   Example: bitAnd(0x6, 0x5) = 0x4
   Legal ops: `~ |`

2. `minusOne` : return a value of -1
   Legal ops: `! ~ & ^ | + << >>`

3. `tmax` : return maximum two's complement integer Legal ops: `! ~ & ^ | + << >>`

4. `copyLSB` : set all bits of result to least significant bit of x
   Example: copyLSB(5) = 0xFFFFFFFF, copyLSB(6) = 0x00000000
   Legal ops: `! ~ & ^ | + << >>`
   Max ops: 5 * Rating: 2

5. `evenBits` : return word with all even-numbered bits set to 1
   Legal ops: `! ~ & ^ | + << >>`

6. `isEqual` : if $x == y$ then return 1, else return 0
   Example: isLessOrEqual(4,5) = 0.
   Legal ops: `! ~ & ^ | + << >>`

7. `negate` - return -x without using minus
   Legal ops: `! ~ & ^ | + << >>`

8. `addOK` : Determine if can compute x+y without overflow
   Example: addOK(0x80000000,0x80000000) = 0,
   addOK(0x80000000,0x70000000) = 1.
   Legal ops: `! ~ & ^ | + << >>`

9. `bitMask` : Generate a mask with 1's from bits `lowbit` through `highbit` and 0's everywhere else.
   Examples: `bitMask(5,3) = 0x38`
   Legal ops: `! ~ & ^ | + << >>`

10. `conditional` : same as the conditional in C: x ? y : z
    Example: conditional(2,4,5) = 4
    Legal ops: `! ~ & ^ | + << >>`

11. `isLess` : if x $<$ y then return 1, else return 0
    Example: isLess(4,5) = 1, isLess(5,4) = 0
    Legal ops: `! ~ & ^ | + << >>`

12. `isPositive` : return 1 if x > 0, else return 0
    Example: isPositive(-1) = 0.
    Legal ops: `! ~ & ^ | + << >>`
    Max ops: 8 * Rating: 3

13. `reverseBytes`: reverse the bytes of x
    Example: reverseBytes(0x01020304) = 0x04030201
    Legal ops: `! ~ \& \^ | + << >>`

14. `bang`: Compute !x without using !
    Examples: bang(3) = 0, bang(0) = 1
    Legal ops: `~ & ^ | + << >>` * Max ops: 12 * Rating: 4

15. `isPower2` : returns 1 if x is a power of 2, and 0 otherwise
    Examples: isPower2(5) = 0, isPower2(8) = 1, isPower2(0) = 0
    Note that no negative number is a power of 2.
    Legal ops: `! ~ & ^ | + << >>`

16. `leastBitPos` : return a mask that marks the position of the least significant 1 bit. If x == 0, return 0
    Example: leastBitPos(96) = 0x20
    Legal ops: `! ~ & ^ | + << >>`

17. `sm2tc` : Convert from sign-magnitude to two's complement where the most significant bit (msb) is the sign bit
    Example: sm2tc(0x80000005) = -5
    Legal ops: `! ~ & ^ | + << >>`

Advice

How are you supposed to know how to do any of this? Read The Fine Textbook, and spend some quality time with truth tables. Sometimes a problem that seems hard but gets fewer points than you expect is covered in the textbook...

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 submit server. 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.
- Run `make clean` before turning in your code.

You will turn in your `bits.c` file using the submit server. We will provide information on how to do that on the class discussion board.