Handout 6

COMP 264: Introduction to Computer Systems (Section 001) Fall 2015 R. I. Greenberg
Comp. Sci. Dept., Loyola U., 820 N. Michigan Ave., Chicago, IL

Assignment #3

Issued 9/25
Due 10/9

1 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.

2 Logistics

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic.

3 Hand Out Instructions

Start by copying `~rig/public/c264/datalab-handout.tar` to a (protected) directory in which you plan to do your work. (e.g., type the following, including the period at the end of each of the last two lines):

```
mkdir hw3
cd hw3
chmod go-rwx .
cp ~rig/public/c264/datalab-handout.tar .
```

Then give the command

```
tar xvf datalab-handout.tar
```

which will cause a number of files to be unpacked into a subdirectory where you can proceed to work. The only file you will be modifying and turning in is `bits.c`.

Looking at the file `bits.c` you'll notice a line near the top with the text
`<Please put your name and userid here>`
Rather than doing exactly what this says, replace that text with the name and principal email address of the one or two individuals comprising your programming team. (Look at the pair programming information on `http://www.cs.luc.edu/~rig/courses` to see how team programming should be done to optimize the experience, but I won't ask for the formalities of a log reporting on the process; just submit by yourself or with a partner as appropriate.)

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 (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:

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

Some functions may 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.

4 The Puzzles

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

4.1 Bit Manipulations

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. 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
`bitNor(x,y)`	`~(x|y)` using only `&` and `~`	1	8
`bitXor(x,y)`	`^` using only `&` and `~`	1	14
`isNotEqual(x,y)`	`x != y?`	2	6
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`replacByte(x,n)`	Replace byte `n` in `x` with `c`	3	10

Table 1: Bit-Level Manipulation Functions.

4.2 Two's Complement Arithmetic

Table 2 describes a set of functions for which it may be helpful to 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.

Name	Description	Rating	Max Ops
`tmax(void)`	largest two's complement integer	1	4
`isNonNegative(x)`	`x >= 0?`	3	6

Table 2: Arithmetic Functions

5 Evaluation

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

65 Correctness points from multiplying the autograder's correctness points by 5.

28 Performance points from multiplying the autograder's performance points by 2.

7 Style points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 13. 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. The autograder gives two points for each correct function that satisfies the operator limit.

Style points. Finally, we've reserved 7 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.

5.1 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:

  ```
  make
  ./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:

  ```
  ./btest -f bitAnd
  ```

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

  ```
  ./btest -f bitAnd -1 7 -2 0xf
  ```

  Check the file `README` for documentation on running the `btest` 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:

  ```
  ./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:

  ```
  ./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:

  ```
  ./driver.pl
  ```

  Your instructors will use `driver.pl` to evaluate your solution.

6 Hand In Instructions

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 our class machines.

1. Make sure you have included your identifying information in your file `bits.c`. *(Please reread carefully the instructions in the second paragraph under "Hand Out Instructions".)*
2. Remove any extraneous print statements.
3. Homework is due at the time of day that class starts. Two options to submit your .c file:
   - Easiest if you are working on a Loyola GNU/Linux machine: Copy your file to the directory `~rig/c264hw3sub` with a filename in the form `Email-X.c`, where `Email` is your email address, and `X` is a "random" string of at least 8 alphanumeric characters. The Unix command for this would look *similar* to:

     ```
     cp bits.c ~rig/c264hw3sub/YOUREMAILADDRESS-RANDOM.c
     ```

     where you must put your own things for "YOUREMAILADDRESS" and "RANDOM". (Don't cut and paste from the PDF, or your tilde might not come out right.) *Remember that if you submit this way the file must be readable by everybody, though you will want to have used* `chmod` *to protect the directory containing the file. Protections show with the* `ls -l` *command illustrated below.* You can verify successful submision by using the "ls" command with the same file name you just copied to, specifically you can use a command *similar* to:

     ```
     ls -l ~rig/c264hw3sub/YOUREMAILADDRESS-RANDOM.c
     ```

   - Or if you prefer: Submit the file through the online submission mechanism on my course web page. Submit it as `bits.c` or `3.c`.

7 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.
- 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 */
}
```