CSE 361 Fall 2015
Lab Assignment L1: Manipulating Bits
Assigned: Wednesday Aug. 26
Due: Wednesday Sept. 09 at 11:59 pm

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers. You'll do this by solving a series of programming "puzzles." Many of these puzzles seem artificial, but you'll understand much more about bits as you work your way through them.

2 Logistics

This is an individual project. All handins are electronic via svn. Code will be vigorously tested for integrity violations; honor code violations will result in -100% on the assignment. Clarifications and corrections will be posted on the course Piazza page.

3 SVN Instructions

So far, your repo contains only `lab0`. As the semester progresses, we will add new lab directories to your repo. In order to see the new directories that we push to your repositories, you will need to perform an `svn update` as follows:

```
yourhome/cse361/yourwustlkey> ls
lab0
yourhome/cse361/yourwustlkey> svn update
-- long printout of files being updated --
yourhome/cse361/yourwustlkey> ls
lab0 lab1
```

Inside `lab1` will be a bunch of files. The `README` will give you a brief explanation of each file and what it's used for. **The only file you will be modifying is `bits.c`. Any modifications you make to any other files will not be considered when we compile and run your code.**

The `bits.c` file contains a skeleton for each of the 10 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:

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

(Keep in mind that when you operate on a signed integer in C, the operator $>>$ likly performs an *arithmetic* shift.)

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.

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
`isZero(x)`	returns 1 is x == 0 and returns 0 if x is non-zero	1	2
`bitXor(x,y)`	compute `x^y` using only `~` and `&`	1	14
`thirdBits(void)`	return a word with every third bit (starting from LSB) set to 1	1	8
`isEqual(x,y)`	return 1 if `x == y`, and 0 otherwise	2	5
`conditional(x,y,z)`	same as `x ? y : z`	3	16
`replaceByte(x,n,c)`	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 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.

Name	Description	Rating	Max Ops
`isTmax(x)`	return 1 if x is the maximum, tow's complement number	1	10
`negate(x)`	return -x	2	5
`divpwr2(x)`	compute `x/(2^n)`	3	15
`addOK(x,y)`	determine if can compute `x+y` without overflow	3	20

Table 2: Arithmetic Functions

5 Evaluation

Your score will be computed out of a maximum of 30 points based on the following distribution:

20 Correctness points.

10 Performance points.

Correctness points. The 10 puzzles you must solve have been given a difficulty rating between 1 and 3, such that their weighted sum totals to 20. 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. You will receive one point for each correct function that satisfies the operator limit.

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:

  ```
  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 `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:

  ```
  unix> ./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:

```
unix> ./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:

  ```
  unix> ./driver.pl
  ```

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

6 Handin Instructions

Once your code is complete, perform an `svn commit -m "finished lab1"`. If you'd like a sanity check to see the status of your repository, look at it in a web browser. Remember the time stamp of the `svn commit` is how we will determine whether the assignment was completed on time or not. Don't forget to commit before midnight!

Do not add any new files to the repository. Particularly, do not add any binaries. We do not need anything but `bits.c`, nor will we look at any other files you may or may not have modified while completing this assignment.

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 */
  }
  ```

- Be sure to check out the `ishow` binary in the distribution and its instruction in README; it will come in handy!
- You probably won't need the `fshow` binary in the distribution, but it may help you understand the floating point representation, which we will cover down the road.