CS 2505 Spring 2018
Data Lab 1
Assigned: Monday, March 12
Due: Friday March 23, 11:59PM
Ends: Friday March 23, 11:59PM

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 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 pairs for this assignment. If you work with a partner, list both names and both PIDs in a comment at the beginning of your submitted file(s).

3 Handout Instructions

Download the file `datalab-handout.tar` from the course website to a (protected) directory on a Linux machine on 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 assigned 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 arithmetic and logical operators:

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

(Of course, you are allowed to use the assignment operator.) 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 for the puzzle, the "Max Ops" field gives the maximum number of operators you are allowed to use to implement each function, and the "Points" field shows how many points a solution to the puzzle is worth. 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	Points
`evenBits()`	return word with all even-numbered bitsset to 1	1	8	6
`thirdBits()`	return word with every third bit (starting from the LSB) set to 1	1	8	8
`upperBits(n)`	pads n upper bits with 1's	1	10	10
`divPwr2(x,n)`	Compute $x/(2^n)$, for $0 \leq n \leq 30$; round toward zero	2	15	12

Table 1: Required Functions

5 Evaluation

Your score will be computed out of a maximum of 44 points based on the following distribution:

Correctness points. **(maximum 36 points)** The 4 puzzles you must solve come with a difficulty rating between 1 and 4 (note that you only have 1s and 2s... this time). Each of the puzzles will be worth the number of points shown in the tables above. 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 `driver.pl`, and no credit otherwise. Note that when you run and autograding `btest` and `driver.pl` tests, the number of points displayed for correctness is equal to the difficulty rating of the puzzle. Your grade for the lab will be computed based on the Points column of the table.

Performance points. **(maximum 8 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 two points for each correct function that satisfies the operator limit.

Explanation and analysis points. **(maximum deduction 22 points)** You must add to the header comment for each function an explanation of the logic employed in your solution. This comment must be accurate, precise and complete. For example:

```
/*
 * isNotMultOf4 - returns 0 if x is a multiple of 4,
 *                        non-0 otherwise
 *   Examples: isMultOf4(0x033B104C) = 0
 *             isMultOf4(0x033B1046) != 0
 *   Legal ops: ~ & ^ |
 *   Max ops: 5
 *   Rating:  1
 *   Logic:
 *     x is a multiple of 4 if and only if x % 4 = 0.  But, since
 *     x % 4 returns the remainder when x is divided by 4, and
 *     dividing by 4 will simply chop off the two low bits of x,
 *     x % 4 will yield the two low bits of the representation
 *     of x.  So, x is a multiple of 4 if and only if its
 *     representation ends in two 0s.
 *
 *     We can obtain the desired bits by applying the right
 *     mask to x.  The key is to set the 30 high bits to 0,
 *     so we could do this:  x & 0x00000003 (C -> 1100).
 *
 */
int isMultOf4(int x) {

   int mask = 0x03;      // OK, it's a one-byte constant '0011',
                         //     which will be sign-extended to a
                         //     32-bit value:
                         //      00000000 00000000 00000000 00000011.
   return x & mask;      // 0 if low bits are 00, non-0 otherwise
}
```

We will evaluate your explanations for some, but not all, of the functions. If your explanation for a function is unsatisfactory, we will apply a deduction of up to 50% to your score for that function.

The evaluation of these comments will be performed by the TAs, not by the autograding tools described below. It is up to you to make sure your explanations are worthy of credit.

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.

- **`make`:** If you have never used a Makefile before, you are encouraged to take a look inside the Makefile file included in the archive. Makefiles are a clean way to automate complex compilation processes. Rather than entering a sequence of `gcc` commands, you can simply type `make` to run a stored sequence.

  You may encounter some errors when using the Makefile to compile. It seems that most, if not all, 64-bit distros fail to include some libraries that are needed to compile 32-bit executables. If you

are running 64-bit CentOS, you should be able to fix the problem by performing one installation command:

```
yum install glibc-devel.i686
yum install libgcc.i686
```

If this is unsuccessful, you may need to complete this assignment using rlogin rather than your virtual machine.

- **`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.

- **`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. If your code does not compile and run correctly when tested with `driver.pl` you will receive a 0 for the assignment.

6 Handin Instructions

You will submit your `bits.c` file to the Curator under the heading `C06`. Unlike earlier programming assignments, this is not autograded at the time of submission. Instead, we will periodically run autograding code on your submissions, and post the results to the Curator system.

Of course, if you apply the tools `btest` and `dlc` properly, you will already know whether your solution passes testing.

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 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 */
}
```