CS 367-001 Homework 2 (Spring 2017)

Due: Thursday, February 9 11:59 PM.

This is an assignment where you can work in groups of two. You can collaborate ONLY with your partner (if you have one). GMU and CS department Honor Codes apply.

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.

Handout Instructions

Start by copying datalab-handout.tar from the *Piazza H2-code link* to your account on zeus.

Once you have the file, create a directory to work in. Move the tar file to this directory and type the command:

```
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**. Make sure to set the access rights of bits.c so that it can be accessed only by you!

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 use to check your solutions for compliance with the coding rules. The remaining files are used to build btest.

The source file that you will be submitting (that is, the modified bits.c) should include in the first commented lines the names and G numbers of the students in your group. Do this right away so you don't forget.

The bits.c file also contains a skeleton for each of the 6 programming puzzles.

IMPORTANT:

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:

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

A few of the functions may further restrict this list. In the beginning of each puzzle in bits.c, you will see the list of operators that are allowed for that specific puzzle. Also, you are not allowed to use any constants longer than 8 bits (constants that cannot be represented with only 8 bits are disallowed). See the comments in bits.c for detailed rules and a discussion of the desired coding style.

As specified in bits.c, you may assume that the system:

1. Uses 2s complement, 32-bit representations of integers (word size = 32 bits).
2. Performs right shifts arithmetically.
3. Has unpredictable behavior when shifting an integer by more than the word size.

You cannot:

1. Use any data type other than "int"
2. Call any functions or macros
3. Use any form of casting
4. Use any control constructs such as "if", "while", "for", "switch", ..
5. Use any other operations such as &&, ||, ..

See the comments in bits.c for the full set of rules and a discussion of the desired coding style. During grading, we will be using the "dlc" program (see below) to check if you comply with these rules. If the "dlc" program indicates a violation of these rules, then you will not get credit for that puzzle.

Evaluation

Your code will be compiled with *gcc* and run and tested on zeus.vse.gmu.edu. Your score will be computed based on:

- Correctness of code running on zeus.
- Performance of code, based on number of operators used in each function.
- Style, based on your instructor's subjective evaluation of the quality of your solutions and your comments.

The puzzles you must solve have been given a difficulty rating between 1 and 3. 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, half credit if it fails one test, and no credit otherwise. 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 for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions. Finally, we've reserved 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.

You will be solving the following six puzzles:

```
/*
 * isTmin - returns 1 if x is the minimum, two's complement
 * number, and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 1
 */
int isTmin(int x) {
  return 2;
}

/*
 * thirdBits - return word with every third bit (starting
```

```
 * from the LSB) set to 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 1
 */
int thirdBits(void) {
  return 2;
}


/*
 * anyEvenBit - return 1 if any even-numbered bit
 *   set to 1
 *   Examples anyEvenBit(0xA) = 0, anyEvenBit(0xE) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
int anyEvenBit(int x) {
  return 2;
}




/*
 * getByte - Extract byte n from word (32-bit) x
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: getByte(0x12345678,1) = 0x56
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int getByte(int x, int n) {
  return 2;
}


/*
 * addOK - Determine if can compute x+y without overflow
 *   Example: addOK(0x80000000,0x80000000) = 0,
 *            addOK(0x80000000,0x70000000) = 1,
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int addOK(int x, int y) {
  return 2;
```

```
}


/*
 * isPositive - return 1 if x > 0, return 0 otherwise
 *   Example: isPositive(-1) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 3
 */
int isPositive(int x) {
  return 2;
}
```

Advice

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 zeus system. If it doesn't compile, we can't grade it so you won't get any points for doing it.

Checking your Program

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. (This program only runs on a Linux x86 system such as the zeus system in the VS&E Lab.) 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`.**

Submission Instructions

Submit a copy of your modified bits.c file through the Blackboard by 2/9, 11:59 PM. Before submitting your file through the Blackboard, you should rename it to reflect your name(s) as well as homework and section numbers. For example, if Mike Smith and Amy Jones are making a submission, then the file should be named:

`hw2-s002-m-smith-a-jones.c`

On Blackboard, select the *Homework 2* submission link. In addition to the homework specification file, you will see a link to submit your C source file (named properly, i.e., reflecting your name).

IMPORTANT: Both students of a group must make the submission separately BEFORE the deadline; moreover, their submissions must be identical. Make sure to coordinate properly.