CS 33, Fall 2012
Lab Assignment L1: Manipulating Bits
Assigned: Oct. 4th, 2012. Due: Oct 14th, 11:59pm

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 must work on this lab independently - you may discuss the assignment with other students, but the code you write and submit must be your own. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the CourseWeb page.

Hand Out Instructions

Start by downloading `datalab-handout.tar` from CourseWeb to a (protected) directory in which you plan to do your work. Then give 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`.

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`.

Looking at the file `bits.c` you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. Do this right away so you don't forget.

The `bits.c` file also contains a skeleton for each of the 13 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:

```
 ! ˜ & ˆ | + << >>
```

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.

Evaluation

Your code will be compiled with GCC and run and tested on one of the class machines. Your score will be computed out of a maximum of 75 points based on the following distribution:

40 points Correctness of code running on one of the class machines.

30 points Performance of code, based on number of operators used in each function.

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

The 13 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 40. 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. You will receive two points for each function that satisfies the operator limit.

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.

Part I: 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.

Function `bitAnd` computes the AND function. That is, when applied to arguments `x` and `y`, it returns `(x&y)`. You may only use the operators `˜` and `|`.

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	`&(x|y)` using only `|` and `~`	1	8
`bitMask(highbit,lowbit)`	bit mask of all 1's between `highbit` and `lowbit`	3	16
`bitXor(x,y)`	`^` using only & and `~`	2	14
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`getByte(x,n)`	Extract byte `n` from `x`	2	6

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`isNonZero(x)`	`x != 0?` without using `!`	4	10
`abs(x)`	compute absolute value of x	4	10
`isPower2(x)`	`x power of 2?`	4	60
`addOK(x, y)`	if `x + y` can be computed without overflow return 1, else return 0	3	20
`isLessOrEqual(x, y)`	if `x <= y` return 1, else return 0	3	24
`multFiveEights(x)`	multiplies x by `5/8`	3	12
`sm2tc(x)`	converts from sign-magnitude to two's complement	4	15
`tmax()`	produces the maximum two's complement integer	1	4

Table 2: Arithmetic Functions

Function `bitMask` creates a bit mask of all ones between a low bit and a high bit.

Function `bitXor` should duplicate the behavior of the bit operation `^`, using only the operations `&` and `~`.

Function `copyLSB` sets all the bits of the result to the least significant bit of x.

Function `getByte` extracts a byte from a word. The bytes within a word are ordered from 0 (least significant) to 3 (most significant).

Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Function `isNonZero` determines whether `x` is nonzero using the legal operators except `!`.

Function `abs` returns the calculated absolute value of `x`.

Function `isPower2` returns 1 if `x` is a power of 2.

Function `addOK` returns 1 if the addition of `x` and `y` can be computed without overflow, and 0 otherwise.

Function `isLessOrEqual` returns 1 if `x` is less than or equal to `y`, and 0 otherwise.

Function `multFiveEights` multiplies `x` by 5/8 rounding towards 0.

Function `sm2tc` translates `x` from an integer in sign-magnitude format to an integer in two's complement format.

Function `tmax` returns the maximum possible representable two's complement integer.

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 our class machines(lnxsrv01.seas.ucla.edu, lnxsrv02.seas.ucla.edu, lnxsrv03.seas.ucla.edu). 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 it compiles, passes the dlc test, and passes the btest tests on the class machines i.e. lnxsrv01.seas.ucla.edu, lnxsrv02.seas.ucla.edu, lnxsrv03.seas.ucla.edu.
- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- Submit your `bits.c` file to CourseWeb where indicated under Lab 1.