Tarea programada #1

Data laboratory

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 are to work **alone** on this project, but you may discuss programing tricks and bit-manipulation code with your peers. The entire "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

Hand Out Instructions

You will find the file `datalab-handout.tar` referenced on the homework page of the class website. You will need to download this file so you can use its contents.

Start by copying `datalab-handout.tar` 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 `btest` program allows you to evaluate (by testing) the functional correctness of your code. Below is the help output from the `btest` program.

```
./btest -h
Usage: ./btest [-v 0|1] [-hag] [-f <func name>] [-e <max errors>]
     -e <n>    Limit number of errors to report for single function to n
     -f <name> Check only the named function
     -g        Print compact grading summary (implies -v 0 and -e 0)
     -h        Print this message
     -v <n>    Set verbosity to level n
          n=0: Only give final scores
          n=1: Also report individual correctness scores (default)
```

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 `bits.c` file also contains a skeleton for each of the programming puzzles. Your assignment is to complete each function skeleton using only `straight-line` 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 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 ECCI Linux machines. Your score will be computed out of a maximum of 100 points based on the following distribution:

- **60** Correctness of code running on one of the class machines.
- **30** Performance of code, based on number of operators used in each function.
- **10** Style points, based on your instructor's or grader's subjective evaluation of the quality of your solutions and your comments.

Each of the 10 problems are worth ten points: 6 for its correctness and 3 for it performance. For the 10 problems assigned, this sums to 90 points, with ten points left to the grader's subjective evaluation of your solutions.

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 (continue 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 to satisfy the limitation it may require you to be more clever. Thus, for each function we've established a maximum number of operators that you are allowed to use. This limit is (usually) generous and is designed only to catch egregiously inefficient solutions. You will receive points for each function that satisfies the operator limit.

Finally, we've reserved 10 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

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	(x&y) using only | and ~	1	8
`bitXor(x,y)`	^ using only & and ~	2	14
`isEqual(x,y)`	x = y?	2	5
`evenBits()`	Return word with all even-numbered bits set to 1	2	8
`fitsBits(x,n)`	Return 1 if x fits in n bits	2	15
`bitMask(x,y)`	Generate a mask consisiting of 1's	3	16
`conditional(x,y,z)`	Compute x ? y : z	3	16
`reverseBytes(x)`	Reverse the bytes of x	3	25
`bang(x)`	Compute !x without using ! operator	4	12
`bitCount(x)`	Count number of 1's in x	4	40

Table 1: Bit-level manipulation functions

Table 1 describes a set of functions that manipulate and test sets of bits. The "Rating" field gives the difficulty rating 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 `~`. Function `bitXor` should duplicate the behavior of the bit operation `^`, using only the operations `&` and `~`.

Function `isEqual` compares x to y for equality. As with all predicate operations, it should return 1 if the tested condition holds and `0` otherwise. Function `evenBits` should generate a constant with all even-numbered bits set to `1` and the odd-numbered bits set to `0`.

Function `fitsBits` should return `1` if its input can be represented as a n-bit two's complement integer. Function `bitMask` generates a mask consisting of 1's from the two positions specified. The `conditional` function implements the C conditional function (? :), and `reverseBytes` reverses the bytes in its input.

Function `bang` computes the C ! without using the ! operator. Function `bitCount` returns a count of the number of 1's in the argument.

Part II: Two's Complement Arithmetic

Name	Description	Rating	Ops
`isZero(x)`	x = 0	1	2
`isNegative(x)`	x < 0?	3	6
`multFiveEights(x)`	Multiplies by 5/8 rounding toward 0	3	12
`sum3(x,y,z)`	x+y+z using only a single '+'	3	16
`addOK(x,y)`	Does x+y overflow?	3	20
`isLess(x,y)`	Is x < y ?	3	24
`abs(x)`	find the absolute value of x	4	10
`isNonZero(x)`	Is x not zero ?	4	10
`tc2sm(x)`	Two's complement to sign-magnitude	4	15

Table 2: Arithmetic functions

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

Function `isZero` decides if its input is zero. Function `isNegative` determines whether x is less than 0. Function `multFiveEights` multiplies its input by 5/8's rounding the result toward 0.

Function `sum3` adds its three inputs only using the '+' operator at most once. Function `addOK` determines whether its two arguments can be added together without overflow.

Function `isLess` determines whether x is less than y. Function `abs` returns the absolute value of its argument, except in the case of the most negative number where it returns its argument. Function `isNonZero` determines whether x is not equal to 0. Function `tc2sm` converts its input from twos-complement representation to sign-magnitude representation.

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 laboratories public Linux (Ubuntu) machines. 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 bitXor`.

Hand In Instructions

You will submit this lab via your control version repository. Create a new directory for "TareasProgramadas", then another for "DataLab", commit all your source code there.