ECEN 324
Lab Assignment: Manipulating Bits

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.

This lab is "Open Book with Internet Solutions but With a Waiting Period (See: Modules -> Resources -> What resources may I use?)." The waiting period is 8 hours. There are also other restrictions on getting help with this lab. See the "Help Guidelines" section at the end of the handout. Your goal should be to ***understand the code that you submit***. You will work in groups of two for this assignment and may submit the same code, if you choose to do so. Both partners are to submit a file. Don't forget to run the `dlc` tool on your code before submitting it (see the "Advice" section of this handout).

Logistics

The only "hand-in" for this lab will be electronic. Any clarifications and revisions to the assignment will be given in class and should be posted on I-Learn.

Hand Out Instructions

Start by creating a directory in which to work and making the new directory your current directory:

```
mkdir datalab
cd datalab
```

Extract `datalab-handout.tar` into the directory you created, with the command:

```
tar xvf /home/ecen324/datalab-handout.tar
```

This will cause a number of files to be unpacked in the directory. Note the useful information in the `README` and additional information in the `bits.c` file. The only file you will be modifying and submitting 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`.

WARNING: You will most likely have the environment variable MAKEFILES set for you. You need to 'unset' it for each terminal window you use. In the terminal windows you are working in, do: MAKEFILES=""

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 15 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code (i.e., no loops or conditionals). You may wish to read and understand "Programming without using an if statement". Another restriction is 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 the ECEN 324 Linux VM. Your score will be computed out of a maximum of 75 points based on the following distribution:

40 Correctness of code running on the ECEN 324 Linux VM.

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

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

The 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 40. Your functions will be evaluated 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, the main concern at this point in the course is that you can get the right answer. However, the desire is 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 be more clever. Thus, for each function there is 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, there are 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. **Note: The score in I-Learn for the lab is 140. The 75 points here will be rescaled to 140. I may adjust the point structure slightly. If so, I'll let you know.**

Part I: Bit manipulations

Table 1: Bit-Level Manipulation Functions

Name	Description	Rating	Max Ops
bitNor (x, y)	~(x | y) using only & and ~	1	8
bitXor (x, y)	x^y using only & and ~	2	14
isNotEqual (x, y)	x != y?	2	6
getByte (x, n)	Extract byte n from x	2	6
copyLSB (x)	Set all bits to LSB of x	2	5
logicalShift (x, n)	Logical right shift x by n	3	16
bitCount (x)	Count number of 1's in x	4	40
bang (x)	Compute !x without using ! operator	4	12
leastBitPos (x)	Mark least significant 1 bit	4	6

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 `bitNor` computes the NOR 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 `isNotEqual` compares `x` to `y` for inequality. As with all *predicate* operations, it should return `1` if the tested condition holds and `0` otherwise.

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

Function `copyLSB` replicates a copy of the least significant bit in all 32 bits of the result. Function `logicalShift` performs logical right shifts. You may assume the shift amount satisfies $1 \leq n \leq 31$.

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

Function `leastBitPos` generates a mask consisting of a single bit marking the position of the least significant one bit in the argument. If the argument equals 0, it returns 0.

Part II: Two's Complement Arithmetic

Table 2: Arithmetic Functions

Name	Description	Rating	Max Ops
tmax (void)	largest two's complement integer	1	4
isNonNegative (x)	x >= 0?	3	6
isGreater (x, y)	x > y?	3	24
divpwr2 (x, n)	x/ (1<<n)	2	15
abs (x)	absolute value	4	10
addOK (x, y)	Does x+y overflow?	3	20

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

Function `tmax` returns the largest integer.

Function `isNonNegative` determines whether `x` is less than or equal to 0.

Function `isGreater` determines whether `x` is greater than `y`.

Function `divpwr2` divides its first argument by 2^n, where *n* is the second argument. You may assume that $0 \leq n \leq 30$. It must round toward zero.

Function `abs` is equivalent to the expression `x<0?-x:x`, giving the absolute value of `x` for all values other than *TMin*.

Function `addOK` determines whether its two arguments can be added together without overflow.

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 ECEN 324 Linux VM. If it doesn't compile, it can't be graded.

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 (constant values 0x00 to 0xFF, etc.). 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.
- The `dlc` binary was compiled on a Linux machine, so it will not run on other machines.
- One of the tests performed by the script is to check your code for the strings "unsigned", "long", "short", and "char" to make sure they have not been used. Unfortunately, they cannot occur even in the comments. If this is a problem, simply revise your comments.
- The `dlc` requires that all variables used in a function be declared at the start of the function.

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`.

Help Guidelines

You and your partner are to attempt all functions, with both of you spending at least 20 minutes each per function, before accessing materials other than the book and notes from class discussions. This time (20 minutes) may be working separately or together as lab partners. After trying each function, you may:

- ask for helps and hints from other students in the class. These hints or helps should be given verbally.
- access Internet resources. If you do this, you need to wait 8 hours before coding up and testing your solution. You also need to place into the comments of the code the resource you used.

There is to be no copying and pasting of code from Internet resources, or students (current or former ones), except from your partner for this lab. You are to be able to document what your code is doing, that is, you are to understand what is being done.

Hand In Instructions

Use the ECEN 324 Linux VM and the following command to submit your `bits.c` file (without quotes): "`submit bits.c`" command to submit your `bits.c` file. A header for the submit command has been added to the bits.c file. The submit process should provide you with the following prompt:

```
Submit homework to allred ecen324 and labDatalab. (y/n)
```

You will get a warning and an error when doing the submit. Ignore them and you should see "Submit successful."