CSCE 230J, Spring 2004
Homework 1: Manipulating Bits
Assigned: Jan. 22, Due: Thur., Feb. 05, 9:00PM

Byron Blunk (`bblunk@unlnotes.unl.edu`) is the lead person for this assignment.

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 may work in a group of up to two people in solving the problems for this assignment. The only "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 need access to a machine running Linux, or FreeBSD. If you do not have a machine of your own, you can use huron.unl.edu for your work.

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 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 15 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 a machine running Unix. Your score will be computed out of a maximum of 75 points based on the following distribution:

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

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. 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.

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)`	return 1 if x == y, 0 otherwise	2	5
`evenBits()`	return word with all even-number bits = 1	2	8
`bitMask(highbit,lowbitx)`	Generate a mask of all 1's	3	16
`conditional(x,y,z)`	same as x ? y : z	3	16
`bitParity(x)`	returns 1 if x contains an odd number of 0's	4	20
`logicalNeg(x)`	implement the ! operator, without using !	4	12

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`isZero(x)`	returns 1 if x == 0, and 0 otherwise	1	2
`negate(x)`	return -x	2	5
`fitsbits(x, n)`	returns 1 if x can fit in n bits	2	15
`isPositive(x)`	`x > 0?`	3	8
`isLess(x, y)`	returns 1 if `x < y`	3	24
`sm2tc(x)`	Convert from sign magnitude to 2s complement	4	15
`satAdd(x,y)`	Adds 2 numbers, accounting for overflow	4	30

Table 2: Arithmetic Functions

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` returns a word with all evennumbered bits set to 1.

Function `bitMask` generates a mask consisting of all 1's between lowbit and highbit.

Function `conditional` performs the same function as the ternary (or trinary) operator in c/c++. `x ? y : z` evaluates to if x, return y, else return z.

Function `bitParity` returns 1 if x contains an odd number of 0's. Function `logicalNeg` computes logical negation without using the `!` operator.

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 `isZero` determines whether `x` is zero or nonzero.

Function `negate` returns the negation of x. e.g. negate(1) = -1

Function `fitsBits` determines whether x can be represented as an n-bit, two's complement integer.

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

Function `isLess` determines whether `x` is less than `y`.

Function `sm2tc` Converts x from sign-magnitude to two's complement where the MSB is the sign bit.

Function `satAdd` adds two numbers, but when positive overflow occurs returns maximum possible value, and when negative overflow occurs returns miniumum positive value.

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. 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 you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- In order to be graded for this assignment, you must turn in `bits.c`
- Use the webhandin program to submit bits.c before the deadline.