CS 213, Fall 2000
Lab Assignment L1: Twiddling Bits
Assigned: Aug. 31, Due: Wed., Sept. 13, 11:59PM

Randy Bryant (`Randy.Bryant@cs.cmu.edu`) is the lead person for this assignment.

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 2 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 Web page `assigns.html` in the class WWW directory.

All files for this assignment are in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-f00/L1
```

Start by copying the file `L1.tar` from that directory to a (protected) directory in which you plan to do your work. Then give the command: `tar xvf L1.tar`. This will cause 8 files to be unpacked into the directory: `README`, `Makefile`, `bits.h`, `btest.h`, `bits.c`, `btest.c`, `decl.c`, and `test.c`. 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`.

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 skeletons 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 8 operators:

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

A few of the functions further restrict this list. See `bits.c` for the detailed rules.

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	& using only | and ~	1	12
`bitXor(x,y)`	^ using only & and ~	2	21
`isEqual(x,y)`	`x == y`?	2	6
`evenBits()`	Mask with even bits set	2	8
`bitMask(lowbit,highbit)`	Generate bit mask	3	24
`bitParity(x)`	Set if `x` has odd number of 1's	4	20

Table 1: Bit-Level Manipulation Functions.

Evaluation

Your code will be compiled with GCC and run and tested on an Intel Cluster machine. Your score will be computed out of a maximum of 75 points based on the following distribution:

40 Correctness of code running on Intel Cluster machine.

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

5 Style points, based on a 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 innefficient 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.

Functions `bitAnd` and `bitXor` should duplicate the behavior of the bit operations & and ^, respectively. In `bitAnd` you may only use the operations | and ~, while in `bitXor` you may only use the operations &

Name	Description	Rating	Max Ops
`tmax(void)`	largest two's complement integer	1	6
`isPositive(x)`	`x > 0?`	3	12
`negate(x)`	`-x`	2	6
`fitsBits(x,n)`	Can `x` be represented with `n` bits	2	20
`isLess(x,y)`	`x < y`	3	39
`sum3(x,y,z)`	`x+y+z` with one '+'	3	24
`sm2tc(x)`	Convert sign-magnitude to two's complement	4	15
`isNonZero(x)`	`x!=0` without using `!`	4	18
`isPower2(x)`	Does `x` equal 2^i for some i?	4	60

Table 2: Arithmetic Functions

and `^`.

Function `isEqual` compares `x` to `y`. As with all *predicate* operations, it should return `1` if the tested condition holds and `0` otherwise.

Function `evenBits` generates a word in which the even-numbered bits are set to 1. Function `bitMask` generates a mask in which all bits between `lowbit` and `highbit` are set to 1.

Function `bitParity` returns 1 if its argument contains an odd number of 1's, and 0 otherwise.

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 `tmax` returns the largest integer.

Function `isPositive` determines whether `x` is greater than 0, while `negate` computes `-x`.

Function `fitsBits` determines whether argument `x` could be represented as an `n`-bit, two's complement number.

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

Function `sum3` should compute the sum of three numbers using a single addition operation.

Function `sm2tc` converts a number from sign-magnitude format to two's complement format. That is, the high order bit of `x` is a sign bit s, while the remaining bits denote a nonnegative magnitude m. The function should then return the two's complement representation of $(-1)^s \times m$.

Function `isNonZero` determines whether $x \neq 0$. To make things more interesting, you may not use the operation `!` for this problem.

Finally, function `isPower2` determines whether `x` is of the form 2^i for some value of i.

Advice

You are welcome to do your code development using any system or compiler you choose. Note, that the version you turn in must compile and run correctly using gcc on the Intel Cluster machines. If it doesn't compile, we can't grade it.

A modified version of the c2c ANSI C compiler will be used to check your programs for compliance with the coding style rules. You can also use it to measure the operator counts of your functions. You can run these tests from one of the Intel Cluster machines by executing the command:

```
/afs/cs.cmu.edu/academic/class/15213-f00/L1/c2c -e bits.c
```

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

- Make sure you have included your identifying information in your file `bits.c`.
- Remove any extraneous print statements.
- To handin your `bits.c` file, type

  ```
  make handin NAME=username
  ```

  where `username` is the Andrew ID of the first person on your team.
- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  make handin NAME=username VERSION=2
  ```
- You can verify your handin by looking in

  ```
  /afs/cs.cmu.edu/academic/class/15213-f00/L1/handin
  ```

  You have list and insert permissions in this directory, but no read or write permissions.