CMSC 313, Fall 2010
Project 3: Manipulating Bits
Assigned: Oct. 18, Due: Wed., Oct. 27, 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 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

Start by copying `datalab-handout.tar` from Mr. Frey's public directory `\afs\umbc.edu\users\f\r\frey\pub\313\proj3` 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:

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

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than 8 bits. Use of `unsigned` is strictly forbidden. You may not declare variables as `unsigned` or cast a variable to be `unsigned`. 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 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.

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

Name	Description	Rating	Max Ops
`bitXor(x,y)`	^ using only & and ~	2	14
`isNotEqual(x,y)`	`x != y?`	2	6
`copyLSB(x)`	Set all bits to LSB of `x`	2	5
`bitMask(hi, lo)`	return mask for bit positions lo to hi	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: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`isZero(x)`	returns 1 if x is zero	1	2
`minusOne()`	returns the value -1	1	2
`divpwr2(x,n)`	`x/(1<<n)`	2	15
`isGreater(x,y)`	`x > y?`	3	24
`isNegative(x)`	`x < 0?`	3	6
`isLessOrEqual(x, y)`	`x <= y?`	3	24
`multFiveEights(x)`	multiples x by 5/8 and returns the result	3	12
`conditional(x, y, z)`	`x ? y : z`	3	16

Table 2: Arithmetic Functions

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 `copyLSB` returns a 1-bit bitmask indicating the position of the least signficant 1 bit.

Function `bitMask` returns a mask with all 1s for bit positions `lbit` to `hbit` and all other bits 0. For example, bitMask(5, 3) returns 0x38.

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

Function `isZero` returns 1 if `x` is zero, 0 otherwise.

Function `minusOne` returns an integer whose value is -1.

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

Function `isNegative` returns 1 if `x` is negative, 0 otherwise.

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

Function `multFiveEights` multiples `x` by 5/8, rounding toward zero and returns the result. For example, multFiveEights(77) = 48.

Function `conditional` simulates the trinary ?: operators. It returns `y` if `x` is "true" and `z` if `x` is "false".

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 UMBC GL Linux 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.
- Create a team name of the form:
  - "*ID*" where *ID* is your Andrew ID, if you are working alone, or
  - "ID_1+ID_2" where ID_1 is the UMBC email IDof the first team member and ID_2 is the UMBC email ID of the second team member.

This should be the same as the team name you entered in the structure in `bits.c`.

- To handin your `bits.c` file, type:

  ```
  make handin TEAM=teamname
  ```

  where `teamname` is the team name described above.

- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  make handin TEAM=teamname VERSION=2
  ```

  Keep incrementing the version number with each submission.

- You can verify your handin by looking in

  ```
  /afs/umbc.edu/users/c/m/cmsc313/pub/cmsc313_submissions/proj3
  ```

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