

# Lab Assignment 1 - Manipulating Bits

EECS-213 - Spring 2007

## Important Dates

**Out:** April 4th, 2007.

**Due:** April 17th, 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 handin will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

## Hand Out Instructions

SSH into any of the tlab machines that is running Linux (e.g., `tlab-login.cs.northwestern.edu`). You will need to supply your CS account name and password. Begin by making a `cs213` directory by executing `mkdir cs213`. Download the handout from the course website and place it in your `cs213` directory. You can download it in Windows and transfer it to your tlab machine using the SSH Secure File Transfer Client.

Then give the command: `tar -xvf data1ab-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 `le README` contains additional documentation about `btest`. Use the command `make btest` to generate the test code and run it with the command `./btest`. The `le 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 `le bits.c` you'll notice a C structure template 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` `le` 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:

Name	Description	Rating	Max Ops
bitNor(x,y)	$\sim(x y)$ using only $\&$ and $\sim$	1	8
bitXor(x,y)	$\wedge$ using only $\&$ and $\sim$	2	14
isNotEqual(x,y)	$x \neq y$ ?	2	6
getBytes(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	30

Table 1: Bit-Level Manipulation Functions.

! ~ & ^ | + << >>

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 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.

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

Table 2: Arithmetic Functions

Function `bitNor` computes the NOR function. That is, when applied to arguments  $x$  and  $y$ , it returns  $\sim(x|y)$ . 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  $n$  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 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 our class machines. If it doesn't compile, we can't grade it.

The `d1c` 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

```
./d1c 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 NETID, if you are working alone, or
  - *ID1 + ID2* where *ID1* is the NETID of the first team member and *ID2* is the NETID 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 `make handin TEAM=teamname VERSION=2`  
Keep incrementing the version number with each submission.
- There is a contest for this programming assignment based on how few perform each task. To enter the contest, execute the `submit.pl` script by typing  
`./submit.pl`  
in your project directory. You can view the live results of the contest at <http://cs213.cs.northwestern.edu/dlcontest>