

CS 135, Fall 2010  
Programming Project 1: Manipulating Bits  
Assigned: September 21, 2010, Due: October 5th, 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 must work in your assigned team in solving the problems for this assignment. The "hand-in" will be two electronic documents – (i) a Report in PDF or MS-word document describing your solution technique, i.e., your Algorithm and (ii) your solutions implemented as C code and specifically into the file `bits.c`. Any clarifications and revisions to the assignment will be posted on the course Web page.

The file (named `bits.c`) that we will provide will contain 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. Detailed rules and a discussion of the desired coding style will be provided in the comments in the file `bits.c`.

**How to start working on this project:** During the first week your team should work on the puzzles and come up with solutions that can be coded (in C) during the following week. The puzzles, which are specified on pages 3 and 4 of this document, essentially require you to perform a number of boolean operations.

## Evaluation

Your code will be compiled with GCC and run and tested on Hobbes or my Mac. You can earn a maximum score of 75 points based on the following distribution:

**40** Correctness of code

**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 41. We will evaluate your functions using various test arguments and by reading your solution included in your report. You will get full credit for a puzzle if it passes all of the tests performed by us, half credit if it fails one test, and no credit otherwise. The grading of your solutions will be based on both (a) your report (describing your solutions and why they work) and (b) your C code submitted. If you do not provide a proper explanation of your solutions in the report or do not submit the report, then you will receive zero points for that puzzle. It is important that you work out the solution and are able to explain your logic correctly – simply “hacking” up a solution will not earn you any points. A correct solution in your report but an incorrect implementation will get you partial credit (and not a zero).

**Example:** In the example problems in the bits.c file, you will have a puzzle that asks you to implement *pow2plus1*, i.e., given  $x$  as input the function should return  $2^x + 1$ .

- The code you will insert into the bits.c file will be described in the bits.c file.
- In the report, you will explain why this “algorithm” works – i.e., a sample explanation can be. The number 1 is represented in  $n$ -bit two's complement binary as  $0^{n-1}1$ . Shifting left  $x \geq 0$  times results in the 1 being shifted left  $x$  places into bit  $x$ . More formally, given input  $A = a_{n-1}, a_{n-2}, \dots, a_1, a_0$  where  $a_0 = 1$  and all other  $a_i = 0$ , then shifting  $A$  left  $x$  times results in the number  $B = b_{n-1}, b_{n-2}, \dots, b_x, \dots, b_1, b_0$  where  $b_x = 1$  and all other  $b_i = 0$  for  $i \neq x$ . From definition of binary representation, this number  $B$  (with a 1 only in bit position  $x$ ) represents decimal number  $2^x$ . Adding one to this number gives us  $2^x + 1$ .

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. Note that we also require a separate document where you need to provide detailed logic of your solutions.

Name	Description	Rating	Max Ops
<code>bitNor(x, y)</code>	$\sim(x y)$ using only <code>&amp;</code> and <code>~</code>	1	8
<code>bitXor(x, y)</code>	$\wedge$ using only <code>&amp;</code> and <code>~</code>	2	14
<code>isNotEqual(x, y)</code>	$x \neq y?$	2	6
<code>getBytes(x, n)</code>	Extract byte <code>n</code> from <code>x</code>	2	6
<code>copyLSB(x)</code>	Set all bits to LSB of <code>x</code>	2	5
<code>logicalShift(x, n)</code>	Logical right shift <code>x</code> by <code>n</code>	3	16
<code>bitCount(x)</code>	Count number of 1's in <code>x</code>	4	40
<code>bang(x)</code>	Compute $\neg x$ without using <code>!</code> operator	4	12
<code>leastBitPos(x)</code>	Mark least significant 1 bit	4	30

Table 1: Bit-Level Manipulation Functions.

## Project Grading Requirements

Note that for each puzzle you solve correctly (i.e., function that you implement) you can get up to 2 points for meeting performance requirements. So your final score will depend on the sum of the difficulty rating and the performance points. For example, in the tables below, if you correctly solve all the 9 puzzles in Table 1 (which have a total rating of 24) then you will earn  $24 + 9.2 + 5 = 47$  points.

**Report Requirement:** You must submit both the report and the code. When possible try to describe your solution formally as an algorithm; at the very least explain the logic in your solution. **Failure to submit a report will result in a grade of zero on Project 1 without exceptions.** The objective behind getting you to write out your solutions in a report is to separate the problem solving process from the coding (i.e., hacking) process. We want you to focus first on the problem to be solved without worrying about “hacking” it up, and develop the solution. It is a habit worth learning regardless of which project or which course you are doing in computer science or any engineering discipline.

## 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.

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. For example, in function `bitNor` you can only use the bitcomplement and bitwise and operators as specified in the second column of the table. Furthermore, you can only use a maximum of 8 legal operators in your solution. Also, you are not allowed to use any constants longer than 8 bits.

Name	Description	Rating	Max Ops
<code>tmax(void)</code>	largest two's complement integer	2	4
<code>isNonNegative(x)</code>	$x \geq 0$ ?	3	6
<code>isGreater(x, y)</code>	$x > y$ ?	3	24
<code>divpwr2(x, n)</code>	$x / (1 \ll n)$	3	15
<code>abs(x)</code>	absolute value	4	10
<code>addOK(x, y)</code>	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 `getBytes` 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 greater 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 required to first look at the definitions of the puzzles and the constraints (i.e., the operators you are allowed to use), and design your algorithm and explain it – *i.e.*, first work on writing your report. Once you have understood the problem and solution and are able to show that it works then examine the bits.c file (which will be posted one week later), write code for your solution and insert it into the bits.c file. You should also refer to the syntax for the C operators – you can look up the C reference book or even the Textbook's chapters on C programming.

## Hand In Instructions

**Each team must hand in their (1) Report file and (2) the code solution file) using Blackboard. If you do not hand in your Report then you will receive a zero for the project.**

More details on submission format will be posted with the code next week.

- Your Report file should be in PDF or MS-Word format, and should be titled with the team number of the team members. For example, if you are in Team 2 then the filename must be team2.pdf (or .doc).