# 15-213 Spring 2003
# Lab 1: Twiddling Bits
# Assigned: Tue, Jan 14, Due: Fri, Jan. 24, 11:59PM

Greg Reshko (reshko@cs.cmu.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 will do this by solving a series of programming puzzles. Many of these puzzles are quite artificial, but you will find yourself thinking much more about bits in working your way through them.

## Logistics

You should work individually 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

All files you need are in the directory `/afs/cs/academic/class/15213-s03/labs/L1`
Start by copying the file `bits-handout.tar` from that directory to a protected directory (i.e. your `213hw/L1` directory) in which you plan to do your work. Then give the command:

```
tar xvf bits-handout.tar
```

This will cause 9 files to be unpacked into the directory: `Makefile`, `README`, `bits.c`, `bits.h`, `btest.c`, `btest.h`, `decl.c`, `dlc`, `tests.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 will 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. See `bits.c` for the detailed rules.

Furthermore, you are not allowed to use arrays or look-up tables and you are not allowed to use casting, i.e. `int a = (unsigned) x;` is illegal.

## 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:

**45 Points**: Correctness of code running on one of the class machines.

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

The 15 puzzles you must solve have been given a difficulty rating between 1 (easiest) and 4 (most difficult), such that their weighted sum totals to 45. 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 have 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 2 points for each function that satisfies the operator limit.
Also, 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

You will implement the following functions that that manipulate and test sets of bits. Please note the *Rating* which gives the difficulty rating (the number of points) for the puzzle, and the *Max Ops* which gives the maximum number of operators you are allowed to use to implement each function.

- `bitAnd(x,y)`
  Duplicate the behavior of the bit operation `&`. Only use operations `|` and `~`.
  Rating: 1 / Max Ops: 8
- `bitOr(x,y)`
  Duplicate the behavior of the bit operation `|`. Only use the operations `&` and `~`.
  Rating: 1 / Max Ops: 8
- `isEqual(x,y)`
  Compares `x` to `y`, i.e. `x==y`. It should return 1 if the tested condition holds and 0 otherwise.
  Rating: 2 / Max Ops: 5
- `logicalShift(x,n)`
  Does a logical right shift of `x` by `n`.
  Rating: 3 / Max Ops: 16
- `bitParity(x)`
  Returns 1 if `x` contains an odd number of 1's, and 0 otherwise.
  Rating: 4 / Max Ops: 20
- `leastBitPos(x)`
  Returns a mask that marks the position of the least significant 1 bit of `x` with a 1. All other positions of the mask should be 0.
  Rating: 4 / Max Ops: 30
- `bitCount(x)`
  Returns a count of the number of 1's in the argument .

Rating: 4 / Max Ops: 40

- `bang(x)`
  Compute `!x` without using `!` operator.
  Rating: 4 / Max Ops: 12

## Part II: Two's Complement Arithmetic

Now you will implement the following functions that make use of the two's complement representation of integers:

- `tmax(void)`
  Returns the largest integer.
  Rating: 1 / Max Ops: 4
- `negate(x)`
  Compute `-x` without using `-` operator.
  Rating: 2 / Max Ops: 5
- `addOk(x,y)`
  Determines whether argument `y` can be added to argument `x` without overflow.
  Rating: 3 / Max Ops: 20
- `isNonZero(x)`
  Determines whether $x \neq 0$. You may not use the operation `!` for this problem.
  Rating: 4 / Max Ops: 10
- `sm2tc(x)`
  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 $(-1)^s \cdot m$.
  Rating: 4 / Max Ops: 15
- `abs(x)`
  Compute absolute value of `x`. Except it returns TMin for TMin.
  Rating: 4 / Max Ops: 10
- `satAdd(x,y)`
  Adds two values and if the result (`x+y`) has a positive overflow it returns the greatest possible positive value (instead of getting a negative result). If the result has a negative overflow, then it should return the least possible negative value.
  Rating: 4 / Max Ops: 30

## Advice

You are welcome to do your code development using any system or compiler you choose. However, please note that the version you turn in must compile and run correctly using our class machines. If it does not compile, we cannot 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. You can also use it to measure the operator counts of your functions. You can run these tests by executing the command:

```
./dlc -e bits.c
```

Check the file `README` for documentation on running the `btest` program. You will 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.
- To handin your `bits.c` file, type:
  `make handin TEAM=ID`
  where `ID` is your Andrew ID
- After the handin, if you discover a mistake and want to submit a revised copy, type
  `make handin TEAM=ID VERSION=2`
  Keep incrementing the version number with each submission.
- You can verify your handin by looking in
  `/afs/cs.cmu.edu/academic/class/15213-s03/labs/L1/handin`
  You have list and insert permissions in this directory, but no read or write permissions.