# 18-213/18-613, Fall 2023
# Data Lab: Manipulating Bits

| | |
|---|---|
| Assigned: | Thursday, August 31 |
| Due: | Tuesday, September 12, 11:59PM |
| Last possible hand in: | Friday, September 15, 11:59PM |

## 1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of common patterns, integers, and floating-point numbers. 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.

## 2 Logistics

- This is an individual project. All handins are electronic using the Autolab service.
- You should do all of your work in an Andrew directory, using either the shark machines or a Linux Andrew machine.
- Before you begin, please take the time to review the course policy on academic integrity at
  `http://www.cs.cmu.edu/~18213/academicintegrity.html`

## 3 Handout Instructions

The only file you will be modifying and handing in is `bits.c`.
The `bits.c` file contains a skeleton for each of the 14 programming puzzles. Your assignment is to complete each function following a strict set of *coding rules*: You may use only *straightline* code for the integer puzzles (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 (ranging from hex `0x00` to `0xFF`). For the integer puzzles, you may use casting, but only between data types `int` and `long` (in either direction.) See the comments in `bits.c` for detailed rules and a discussion of the coding rules for each function.

You can assume the following:

- Values of data type `int` are 32 bits.
- Values of data type `long` are 64 bits.
- Signed data types use a two's complement representation.
- Right shifts of signed data are performed arithmetically.
- When shifting a $w$-bit value, the shift amount should be between 0 and $w - 1$.
- Predicate operators, including the unary operator `!` and the binary operators ==, !=, <, >, <=, and >=, return values of type `int`, regardless of the argument types.

# 4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`.

## 4.1 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. See the comments in `bits.c` for more details on the desired behavior of the functions. You may also refer to the test functions in `tests.c`. These are used as reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules for your functions. All arguments and return values for the functions are of type `long`.

## 4.2 Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers. All arguments and return values are of type `long`. Refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

## 4.3 Floating-Point Operations

For this part of the assignment, you will implement some common single-precision floating-point operations. In this section, the coding restrictions are relaxed:

| Name | Description | Rating | Max Ops |
|---|---|---|---|
| `bitMatch(x,y)` | Create mask indicating which bits in $x$ match those in $y$ using only and & | 1 | 14 |
| `anyOddBit(x)` | return 1 if any odd-numbered bit in word set to 1 where bits are numbered from 0 (least significant) to 63 (most significant) | 2 | 14 |
| `ezThreeFourths(x)` | multiplies by $3/4$ rounding toward 0 | 3 | 12 |
| `bitMask(x,n)` | Generate a mask consisting of all 1's between lowbit and highbit | 3 | 16 |
| `howManyBits(x)` | return the minimum number of bits required to represent $x$ in two's complement | 4 | 70 |
| `hexAllLetters(x)` | return 1 if the hex representation of $x$ contains only characters 'a' through 'f' | 4 | 30 |
| `logicalNeg(x)` | Return !$x$ without using the ! operator | 4 | 12 |

Table 1: Bit-Level Manipulation Functions.

| Name | Description | Rating | Max Ops |
|---|---|---|---|
| `tmax()` | Return maximum two's complement integer | 1 | 4 |
| `isTmin(x)` | Returns 1 if $x$ is the minimum, two's complement number, and 0 otherwise | 1 | 10 |
| `isNegative(x)` | return 1 if $x$ ¡ 0, return 0 otherwise | 2 | 6 |
| `integerLog2(x)` | return floor(log base 2 of $x$), where $x$ ¿ 0 | 4 | 60 |

Table 2: Arithmetic Functions

- You are allowed to use standard control structures, such as `if` statements and `while` loops. However, you should NOT use any nested loops, as they may cause the autograder to time out.
- You may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants.
- You may use all integer operators, as well as logical operators such as == and `&&`.
- You may NOT use any unions, structs, or arrays.
- You may NOT use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function with type `unsigned`, and any returned floating-point value will be of type `unsigned`. Your code should perform the bit manipulations that implement the specified floating point operations.

Table 3 describes a set of functions that operate on the bit-level representations of single-precision, floating-point numbers. Refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

| Name | Description | Rating | Max Ops |
|---|---|---|---|
| `floatNegate(uf)` | Return bit-level equivalent of expression $-f$ for floating point argument $f$. | 2 | 10 |
| `floatFloat2Int(f)` | Return bit-level equivalent of expression (int)$f$ for floating point argument $f$. | 4 | 30 |
| `floatScale1d4(f)` | Return bit-level equivalent of expression $0.25 * f$ for floating point argument $f$. | 4 | 30 |

Table 3: Floating-Point Functions.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
linux> make
```

You can use `fshow` to see how a bit pattern represents a floating-point number, using either a decimal or hex representation of the pattern:

```
linux> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)

linux> ./fshow 0x15213

Floating point value 1.212781782e-40
Bit Representation 0x00015213, sign = 0, exponent = 0x00, fraction = 0x015213
Denormalized.  +0.0103172064 X 2^(-126)
```

You can also give `fshow` floating-point values, and it will decipher their bit structure.

```
linux> ./fshow 15.213

Floating point value 15.2130003
Bit Representation 0x41736873, sign = 0, exponent = 0x82, fraction = 0x736873
Normalized.  +1.9016250372 X 2^(3)
```

# 5 Evaluation

Your score will be computed out of a maximum of 67 points based on the following distribution:

**39** Correctness of code.

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

*Correctness points.* The 14 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 39. We will use the the `dlc` compiler to check that your function follows the coding rules. We will use the BDD checker to verify that your function is correct. You will get full credit for a puzzle only if it follows all of the coding rules and it passes all of the tests performed by the BDD checker, and no credit otherwise.

*Performance points.* 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. We will use the `dlc` compiler to verify that you've satisfied the operator limit. You will receive two points for each correct function that satisfies the operator limit.

# 6 Autograding your work

We have included some handy autograding tools in the handout directory—`btest`, `dlc`, BDD checker, and `driver.pl`—to help you check the correctness of your work.

- **`btest`:** This program checks the correctness of the functions in `bits.c` by calling them many times with many different argument values. To build and use it, type the following two commands:

  ```
  linux> make
  linux> ./btest
  ```

  Notice that you must rebuild `btest` each time you modify your `bits.c` file.

  You'll find it very helpful to use `btest` 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:

```
linux> ./btest -f allOddBits
```

This will call the `allOddBits` function many times with many different input values. You can feed `btest` specific function arguments using the option flag `-1`:

```
linux> ./btest -f allOddBits -1 0xFF
```

This will call `allOddBits` exactly once, using the specified arguments. Use this feature if you want to debug your solution by inserting `printf` statements; otherwise, you'll get too much output.

- **`dlc`:** This is a modified version of an ANSI C compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

  ```
  linux> ./dlc bits.c
  ```

  The program runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. Running with the `-e` switch:

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

  causes `dlc` to print counts of the number of operators used by each function. Type `./dlc -help` for a list of command line options.

- **`BDD checker`:** The `btest` program simply tests your functions for a number of different cases. For most functions, the number of possible argument combinations far exceeds what could be tested exhaustively. To provide complete coverage, we have created a *formal verification* program, called `cbit`, that exhaustively tests your functions for all possible combinations of arguments. It does this by using a data structure known as *Binary Decision Diagrams* (BDDs).

  You do not invoke `cbit` directly. Instead, there is a series of Perl scripts that set up and evaluate the calls to it. Execute

  ```
  linux> ./bddcheck/check.pl -f fun
  ```

  to check function `fun`. Execute

  ```
  linux> ./bddcheck/check.pl
  ```

  to check all of your functions. Execute

  ```
  linux> ./bddcheck/check.pl -g
  ```

  to check all of your functions and get a compact tabular summary of the results.

- **`driver.pl`:** This is a driver program that uses `dlc` and the BDD checker to compute the correctness and performance points for your solution. This is the same program that Autolab uses when it autogrades your handin. Execute

  ```
  linux> ./driver.pl
  ```

  to check all of your functions and to display the result in a compact tabular format.

# 7 Handin Instructions

Unlike other courses you may have taken in the past, in this course you may handin your work as often as you like until the due date of the lab.

To receive credit, you will need to upload your `bits.c` file using the Autolab option "Handin your work." Each time you handin your code, the server will run the driver program on your handin file and produce a grade report (it also posts the result on the scoreboard). The server archives each of your submissions and resulting grade reports, which you can view anytime using the "View handin history" option.

**Handin Notes:**

- At any point in time, your most recently uploaded file is your official handin. You may handin as often as you like.
- Each time you handin, you should use the "View your handin history and scores" option to confirm that your handin was properly autograded. Manually refresh the page to see the autograded result.
- You must remove any extraneous print statements from your `bits.c` file before handing in.

# 8 Advice

- Start early.
- See `http://www.cs.cmu.edu/~18213/faq.html` for answers to frequently-asked questions.
- You can work on this assignment using one of the class shark machines

  ```
  linux> ssh -X andrewid@shark.ics.cs.cmu.edu
  ```

  or one of the Andrew Linux servers

  ```
  linux> ssh -X andrewid@unix.andrew.cmu.edu
  ```

- Test and debug your functions one at a time. Here is the sequence we recommend:
  - **Step 1.** Test and debug one function at a time using `btest`. To start, use the `-1` argument in conjunction with `-f` to call one function with one specific set of input argument(s):

    ```
    linux> ./btest -f allOddBits -1 7
    ```

    Feel free to use `printf` statements to display the values of intermediate variables. However, be careful to remove them after you have debugged the function.
  - **Step 2.** Use `btest -f` to check the correctness of your function against a large number of different input values:

    ```
    linux> ./btest -f allOddBits
    ```

If `btest` detects an error, it will print out the specific input argument(s) that failed. Go back to Step 1, and debug your function using those arguments

- **Step 3.** Use `dlc` to check that you've conformed to the coding rules:

```
linux> ./dlc bits.c
```

- **Step 4.** After your function passes all of the tests in `btest`, use the BDD checker to perform the definitive correctness test:

```
linux> ./bddcheck/check.pl -f allOddBits
```

- **Step 5.** Repeat Steps 1–4 for each function. At any point in time, you can compute the total number of correctness and performance points you've earned by running the driver program:

```
linux> ./driver.pl
```

- Some hints for `dlc`:
  - Don't include the `<stdio.h>` header file in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages. You will still be able to use `printf` in your `bits.c` file for debugging without including the `<stdio.h>` header, although `gcc` will print a warning that you can ignore.
  - The `dlc` program enforces a stricter form of declarations than is the case for C++ or Java or even than is enforced by `gcc`. In particular, any declaration must appear in a block (what you enclose in curly braces) before any statement that is not a declaration. For example, it will complain about the following code:

```
int foo(int x)
{
  int a = x;
  a *= 3;      /* This statement is not a declaration */
  int b = a;  /* ERROR: Declaration not allowed here */
}
```

- Some hints for the BDD checker:
  - Test your code with `btest` before using the BDD checker. The BDD checker does not provide very good error messages when given malformed code.
  - The BDD checker cannot handle functions that call other functions, including `printf`. You should use `btest` to evaluate code with debugging `printf` statements. Be sure to remove any of these debugging statements before handing in your code.
  - The BDD checker scripts are a bit picky about the formatting of your code. They expect the function to open with a line of the form:

```
int fun (...)
```

or

```
unsigned fun (...)
```

and to end with a single right brace in the leftmost column. That should be the only right brace in the leftmost column of your function.

If you have any questions about this lab, first reread this entire handout and check the FAQ. They contain lots of details that may require multiple readings to fully appreciate. If you still have lab questions, or you have questions about the Autolab system or the course in general, please contact the staff via Piazza. We respond days and evenings and are very good about getting back to you fast. Remember: We're here to help. Good luck!