Data Assignment: Manipulating Bits

COMP 222: Introduction to Computer Organization
Assigned: 8/29/25, **Due: Friday, 9/12/25, 11:55PM**

Important: This project must be done individually. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp222/html/assignments.html`

1 Introduction

The objective of this assignment is for you to become more familiar with the bit-level representations of 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 Getting Started

To get started on this assignment, visit the web page at `https://classroom.github.com/a/IwzjQBjw`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP222-F25-classroom" and "Accept the assignment — Data". Moreover, it should have a green button labeled "Accept this assignment"[1]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR` system if you have not already done so. Type the following:

```
git clone [Copy the URL for your repo here]
```

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

```
data-[YOUR github ID]
```

Please `cd` into this directory, and run the command `ls`. You should see the following files:

- `Makefile` - Specifies how the `make` command should compile `btest`, `fshow`, and `ishow`
- `README` - Describes these files
- `bits.c` - The *only* file you will be modifying and turning in

[1] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

- `bits.h` - Header file
- `btest.c` - The main `btest` program
  - `btest.h` - Used to build `btest`
  - `decl.c` - Used to build `btest`
  - `tests.c` - Used to build `btest`
- `dlc` - Rule checking compiler executable (data lab compiler)
- `driver.pl` - Driver program that uses `btest` and `dlc` to autograde `bits.c`
- `fshow.c` - Utility for examining floating-point representations
- `ishow.c` - Utility for examining integer representations

If you do ***NOT*** see these files, contact the course staff immediately!

The only file you will be modifying and turning in is `bits.c`. This file contains a skeleton for each of the 14 programming puzzles. Your assignment is to complete each function skeleton using only *straight-line* 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. In contrast, there are fewer and different restrictions on the floating-point puzzles. See the comments in `bits.c` for detailed rules.

3 The Puzzles

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

Table 1 lists the puzzles in their estimated order of difficulty from easiest to hardest. 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 that 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.

For the floating-point puzzles, you will implement some common single-precision floating-point operations. For these puzzles, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having 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.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, `cd` into your data assignment directory and type the command[2]:

```
CLEAR> make
```

[2]When a line begins with `CLEAR>`, you must perform the command that follows on the CLEAR system, not your personal computer. Do not type `CLEAR>` at the command prompt from the CLEAR system, only type the text that follows it.

Name	Computes	Rating	Max ops
`bitOr(x,y)`	`x|y` using only `~` and `&`	1	8
`specialBits()`	Returns the bit pattern `0xffca3fff`	1	3
`isZero(x)`	Returns `1` if `x == 0`, and `0` otherwise	1	2
`anyEvenBit(x)`	Returns `1` if any even-numbered bit in `x` is set to `1` where bits are numbered from `0` (least significant) to `31` (most significant)	2	12
`negate(x)`	Returns `-x`	2	5
`leastBitPos(x)`	Returns a mask that marks the position of the least significant `1` bit; if `x == 0`, returns `0`	2	6
`dividePower2(x,n)`	Computes `x/(2^n)`, for `0 <= n <= 30`	2	15
`rotateLeft(x,n)`	Rotate `x` to the left by `n`	3	25
`isLess(x,y)`	If `x < y` then returns `1`, else returns `0`	3	24
`isPower2(x)`	Returns `1` if `x` is a power of `2`, and `0` otherwise	4	20
`bitReverse(x)`	Reverse bits in a 32-bit word	4	45
`floatAbsVal(uf)`	Bit-level equiv. of absolute value of f.p. arg. `f`	2	10
`floatInt2Float(x)`	Bit-level equiv. of `(float)x` for integer arg. `x`	4	30
`floatScale64(uf)`	Bit-level equiv. of `64*f` for f.p. arg. `x`	4	35

Table 1: Data assignment puzzles. For the floating point puzzles, value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
CLEAR> ./fshow 2080374784
Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

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

4 Evaluation

Your score will be computed out of a maximum of 63 points based on the following distribution:

35 Correctness points.

28 Performance points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 35. We will evaluate your functions using the `btest` program, which is described in the next section. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, 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. You will receive two points for each correct function that satisfies the operator limit.

Autograding your work

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

- **`btest`:** This program checks the functional correctness of the functions in `bits.c`. To build and use it, type the following two commands:

  ```
  CLEAR> make
  CLEAR> ./btest
  ```

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

  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:

  ```
  CLEAR> ./btest -f bitOr
  ```

  You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`:

  ```
  CLEAR> ./btest -f bitOr -1 4 -2 5
  ```

  Check the file `README` for documentation on running the `btest` program.

- **`dlc`:** This is a modified version of a 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:

  ```
  CLEAR> ./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:

  ```
  CLEAR> ./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.

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. It takes no arguments:

  ```
  CLEAR> ./driver.pl
  ```

  Your instructors will use `driver.pl` to evaluate your solution.

5 Turnin Instructions

To turn in your assignment, you ***must*** use `git push` to copy your work to the `github` remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code, we encourage you to `push` periodically. Please note, the *only* file that you need to turn in is `bits.c`. In other words, this is the only file on which you should ever perform `git add`. For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.

6 Advice

- 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.

- While compiling your code, `gcc` may print error and/or warning messages. Error messages describe defects in the code that prevent the compilation from completing, such as syntax errors or undefined variables. In contrast, warning messages describe code that can be compiled, but the compiler suspects that the code contains logic errors that will result in unexpected behavior or runtime errors. However, the compiler may be wrong about warnings. For example, one of our solutions generated the following warning because we were using `~` and `!` in an unusual way.

```
bits.c: In function puzzle:
bits.c:221:18: warning: ~ on a boolean expression [-Wbool-operation]
  221 |     int zmask = (~!n)+1;
      |                  ^
bits.c:221:18: note: did you mean to use logical not?
  221 |     int zmask = (~!n)+1;
      |                  ^
      |                  !
```

  In other words, the compiler thought we made a typo, and that we meant to write `!!`. We did not, so we ignored this warning.

- The `dlc` program enforces an older form of C variable definitions than is allowed by the modern C language and `gcc`. In particular, any definition must appear in a block (what you enclose in curly braces) before any statement that is not a definition. For example, it will complain about the following code:

```
int foo(int x)
{
  int a = x;
  a = a * 3;  /* Statement that is not a definition */
  int b = a;  /* ERROR: Definition not allowed here */
}
```