15-213/18-213/15-513, Summer 2020
Data Lab: Manipulating Bits

Assigned:	Fri, May 22, 2020
Due:	Fri, May 29, 11:59 pm Eastern Time
Last possible hand in:	Mon, June 1, 11:59 pm Eastern Time

For the fastest response, please use Piazza. Your posts will be private by default. Before asking a question, though, please read this handout in its entirety, and also look at the FAQ page. This lab involves a combination of many tools, each of which has its own quirks. The FAQ covers many of these.

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 the Shark machines.
- Before you begin, please take the time to review the course policy on academic integrity at
  http://www.cs.cmu.edu/~213/academicintegrity.html

3 Logging in to Autolab

All 213/513/613 labs are being offered this term through a Web service developed by CMU students and faculty called *Autolab*, located at `https://autolab.andrew.cmu.edu`.

You must be enrolled to receive an Autolab account. If you added the class late, you might not be included in Autolab's list of valid students. In this case, you won't see the 213 course listed on your Autolab home page. If this happens, contact the staff and ask for an account.

If you are still on the waitlist for the course, then download a copy of the file `datalab-handout.tar` from the course schedule web page. You can get working on the lab and then get an Autolab account once you are enrolled.

4 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 13 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, function calls, 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.

5 The Puzzles

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

5.1 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`.

Name	Description	Rating	Max Ops
`bitNor(x,y)`	Compute `~(x|y)` using only `~` and `&`	1	8
`anyOddBit(x)`	Return 1 if any odd-numbered bit in word is set to 1	2	14
`logicalShift(x, n)`	Shift x to the right by n, using a logical shift	3	20
`bang(x)`	Compute `(long) !x` without using `!`	4	12

Table 1: Bit-Level Manipulation Functions.

5.2 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.

Name	Description	Rating	Max Ops
`negate(x)`	Return -x	2	5
`sign(x)`	Return 1 if positive, 0 if zero, and -1 if negative	2	10
`subtractionOK(x, y)`	Determine if can compute x-y without overflow	3	20
`isLess(x)`	if $x < y$ then return 1, else return 0	3	24
`howManyBits(x)`	Return the minimum number of bits required to represent x in two's complement	4	70

Table 2: Arithmetic Functions

You can use the provided problem `ishow` to see the decimal and hexadecimal representations of numbers. First, compile the code as follows:

```
linux> make
```

Then use it to examine hex and decimal values typed on the command line:

```
linux> ./ishow 0x8000000000000000
Hex = 0x8000000000000000L,Signed = -9223372036854775808L,Unsigned = 9223372036854775808L

linux> ./ishow  -123456789
Hex = 0xfffffffff8a432ebL,Signed = -123456789L,Unsigned = 18446744073586094827L
```

5.3 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:

- 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(f)`	Return bit level equivalent of expression $-f$	2	10
`floatIsLess(f, g)`	Compute $f < g$. If either argument is NaN, return 0	3	30
`floatScale4(x)`	Compute $4 \cdot x$	4	30
`floatUnsigned2Float(u)`	Convert unsigned integer u to float	4	30

Table 3: Floating-Point Functions.

The included program `fshow` helps you understand the structure of floating point numbers. When you ran `make` to compile `ishow`, it should also have compiled `fshow`. 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)
```

6 Evaluation

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

37 Correctness of code.

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

Correctness points. The 13 puzzles you must solve have been given a difficulty rating between 1 and 4 as described above, which sum to a score of 37. You will receive full correctness points for a puzzle if it passes all coding rules and passes 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, while some of the puzzles can be solved by brute force, it is possible to develop more efficient solutions. Thus, you will receive 2 performance points for each *correct* function that satisfies the operator limits described above. *However,* keep in mind that you can still receive correctness points even if the operator limit is exceeded.

7 Autograding your work

We have included some handy autograding tools in the handout directory — `btest`, `dlc`, and BDD checker — to help you check the correctness of your work. We have also included `driver.pl`, which is the exact program used to grade your work on Autolab.

7.1 btest

The `btest` 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 copyLSB
```

This will call the `copyLSB` function many times with many different input values. You can feed `btest` specific function arguments using the option flags `-1`, `-2`, `-3` for the first three function arguments respectively:

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

This will call `copyLSB` 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.

Warning: the btest program does not exhaustively test correctness! You also need to run bddcheck as described below.

7.2 dlc

The `dlc` program 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 program will print an error if 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 causes `dlc` to print counts of the number of operators used by each function:

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

7.3 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. To check all of your functions and get a compact tabular summary of the results, execute:

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

7.4 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. To check all of your functions, execute:

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

7.5 Formatting

This lab will not be style graded. The score you receive on Autolab will be your final score. However, since `bits.c` is the only file uploaded, Autolab will not enforce formatting rules for this lab.

As with Lab 0, we provide the `clang-format` tool for you to help format your code. To invoke it, run `make format`. You can modify the `.clang-format` file to reflect your preferred code style. More information is available in the style guideline at `https://www.cs.cmu.edu/~213/codeStyle.html`.

8 Handin Instructions

To receive credit, you will need to upload your `bits.c` file using one of the following options:

1. Run `make submit` from the Shark machine command line.
2. Upload your file directly to the Autolab website.

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).

Handin Notes:

- At any point in time, your most recently uploaded file is your official handin. You may handin as often as you like, with no penalty.
- Each time you handin, you should check your score to confirm that your handin was properly autograded. You can click on your score in Autolab to see the autograder output.
- You must remove any extraneous print statements from your `bits.c` file before handing in.

9 Formatting of C code for Datalab

The `dlc` program requires special care in how you write your code:

- 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`. All declarations must appear before any statement that is not a declaration.
  - NOT OK: Mixing declarations and statements

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

– OK: Forward-declaring all variables

```
long foo(long x) {
  long a, b;
  a = x;
  a *= 3;
  b = a;
  return b;
}
```

– OK: Writing a sequence of declarations at the start of a function

```
long foo(long x) {
  long a = x;
  long a3 = a * 3;
  long b = a;
  return b;
}
```

The BDD checker introduces other considerations in the formatting of your C code.

- 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.
- What to do when `bddcheck/check.pl` reports a syntax error:
  - The syntax errors reported by `bddcheck/check.pl` are indexed from the first line of the function, not the file! This can be confusing.
  - The BDD checker scripts are a bit picky about the formatting of your functions. They expect the function to open with a line of one of the following forms:

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

    They also expect the function to end with its closing brace on a separate line, with no whitespace before it.
  - Some of the scripts also don't understand comments very well. So if you comment out part of your code, make sure the first line of each function is explicitly commented out with `//`, instead of relying on block comments with `/*` and `*/`.

10 Advice

- Start early.
- See http://www.cs.cmu.edu/~213/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 copyLSB -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 copyLSB
    ```

    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 copyLSB
    ```

  - **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
    ```

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!