PKU-ICS
Data Lab: Manipulating Bits

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 your working dir, using the ICS Linux server.

3 Logging in to Autolab

All ICS labs are being offered this term through a Web service developed by CMU students and faculty called *Autolab*. Before you can download your lab materials, you will need to update your Autolab account. Point your browser at the Autolab front page

```
https://autolab.pku.edu.cn
```

You will be asked to authenticate. After you authenticate this first time, Autolab will prompt you to update your account information with a *nickname*. Your nickname is the external name that identifies you on the public scoreboards that Autolab maintains for each assignment, so pick something interesting! You can change your nickname as often as you like. Once you have updated your account information, click on "Save Changes" button, and then select the "Home" link to proceed to the main Autolab page.

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.

4 Handout Instructions

Your lab materials are contained in a Unix tar file called `datalab-handout.tar`, which you can download from Autolab. After logging in to Autolab at

```
https://autolab.pku.edu.cn
```

you can retrieve the `datalab-handout.tar` file by selecting "Data Lab->Download handout". Start by copying `datalab-handout.tar` to the Linux working directory where you plan to do your work. Then give the command

```
linux> tar xvf datalab-handout.tar
```

This will create a directory called `datalab-handout` that contains a number of files. The only file you will be modifying and handing in is `bits.c`.

The `bits.c` file contains a skeleton for each of the 16 programming puzzles. Your assignment is to complete each function skeleton 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. See the comments in `bits.c` for detailed rules and a discussion of the coding rules for each functions.

WARNING: Do not let the Windows WinZip program open up your `.tar` file (many Web browsers are set to do this automatically). Instead, save the file to your Linux working directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files, doing so can cause loss of data (and important work!).

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.

Name	Description	Rating	Max Ops
`bitXnor(x)`	`~(x^y)` using only `~` and `|`	1	7
`bitConditional(x,y,z)`	`x?y:z` for each bit respectively	1	4
`byteSwap(x,n,m)`	Swaps the `n`th byte and the `m`th byte	2	16
`logicalShift(x,n)`	Shift `x` to the right by `n`, using a logical shift	3	16
`cleanConsecutive1(x)`	Clean any consecutive `1` in the binary form of `x`	4	16
`leftBitCount(x)`	Count the leading ones in the binary form of `x`	4	50

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. In this part, you will get a better understanding of how intergers are represented in a computer system. Again, refer to the comments in `bits.c` and the reference versions in `tests.c` for more details!

Name	Description	Rating	Max Ops
`counter1To5(x,n)`	Return `1+x` if `x<5`, return `1` otherwise	2	15
`sameSign(x,y)`	Return `1` if `x` and `y` have same sign, return `0` otherwise	2	5
`satMul3(x)`	Multiply `x` by `3`, saturating to T_{min} or T_{max} if overflow	3	25
`isGreater(x, y)`	Return `1` if `x>y`, return `0` otherwise	3	24
`subOK(x,y)`	Determine whether can compute `x-y` without overflow	3	20
`trueFiveEighths(x)`	Multiply `x` by `5/8`, avoiding overflow errors	4	25

Table 2: Arithmetic Functions

5.3 Floating-Point Operations

For this part of the assignment, you will implement some common floating-point operations. In this section, 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.

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

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
```

Name	Description	Rating	Max Ops
`float_half(x)`	Compute `x/2`	4	30
`float_i2f(x)`	Convert `int` `x` to `float`	4	30
`float64_f2i(x)`	Convert `double` `x` to `int`	4	20
`float_negpwr2(x)`	Compute `2^(-x)`	4	20

Table 3: Floating-Point Functions.

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

```
linux> ./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.

```
linux> ./fshow 0x15213

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

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 80 points based on the following distribution:

48 Correctness of code.

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

Correctness points. The 16 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 48. 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.

7 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 functional 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 bitXnor
  ```

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

  ```
  linux> ./btest -f bitXnor -1 7 -2 0xf
  ```

  This will call `bitXnor` exactly once, using the arguments `x=7` and `y=15`. 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 code in `btest` 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
  ```

  the 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.

8 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.

9 Advice

- You are recommended to use the instruction `make clean` each time before using `make`.
- See `http://autolab.pku.edu.cn/faq.html` for answers to frequently-asked questions.
- You can work on this assignment using one of the class machines

```
linux> ssh -p 1x22 id@ics9.pku.edu.cn
```

- 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` and `-2` arguments in conjunction with `-f` to call one function with one specific set of input argument(s):

    ```
    linux> ./btest -f bitXnor -1 23 -2 0xabcd
    ```

    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 bitXnor
    ```

    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 bitXnor
    ```

  - **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 that 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:
  - 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.

Good luck!