15-213, Spring 2006
Lab Assignment L1: Bit Manipulation
Assigned: Tues., Jan. 24, Due: Thurs., Feb. 2, 11:59PM

Yongjun Jeon (`yongjunj@andrew.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'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.

Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the course message board.

Creating your Autolab Account

All 15-213 labs are being offered this term through a Web service developed by Prof. O'Hallaron called *Autolab*. Before you can download your lab materials, you will need to create your Autolab account. Point your browser at the Autolab front page

```
    http://autolab.cs.cmu.edu
```

and select the "15213-s06" link. Apache will prompt you for a user name and password. Enter your Andrew login ID, leave the password field blank, and press "OK". If you are on Autolab's list of registered students, you will be directed to the Autolab "Create" page, where you will be asked to enter a password, nickname, and email address. After you enter this information, Apache will prompt you again for your user name and password. This time, enter your Andrew login ID and the password you just registered with Autolab, and then press "OK". You will be sent to the main Autolab page for this course, which you should bookmark for future use.

A couple of important notes on creating your account:

- Autolab passwords are encrypted on the network and the server, so you can safely use your Andrew password as your Autolab password if you don't want to have remember another password.
- After you have created your account, you can change your password, nickname, and email address anytime by visiting the Autolab "Update" page.
- If you added the class late, you might not be included in Autolab's list of valid students, and thus won't be redirected to the Autolab "Create" page. If this happens, just send email to Prof. Pfenning (fp@cs.cmu.edu) requesting an Autolab account and he will add you to the list.

Obtaining your Lab Materials

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 through the front page

```
http://autolab.cs.cmu.edu
```

you can retrieve the `datalab-handout.tar` file by selecting "Download lab materials" and then hitting the "Go" button.

Start by copying `datalab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the command "**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 file `bits.c` contains skeletons for the puzzles you will be solving. Your job is to fill each of these skeletons with a solution that is in line with the rules we give (see Puzzle Coding Rules section).
- The file `driver.pl` is the driver program that tests your `bits.c`. You can use it to compute your unofficial total score. To get an official score, you must submit your solution through Autolab (see Handin Instructions section below). This program uses the same grading rules as the tester on Autolab.
- The file `dlc` is a program that should not be run directly. It is automatically invoked by the driver. dlc is a modified simple C compiler that checks your solutions for compliance with the coding rules (see Puzzle Coding Rules section).
- The files in the subdirectory `bddcheck` implement the BDD checker, a tool that formally verifies your code. The BDD checker should not be run directly since it is run by the driver. This utility performs an exhaustive check to ensure that your solution will always produce the same answer as our reference solution.
- One of the properties of the BDD checker is that it stops checking your code once it has found a single set of inputs on which your solution does not match the output of the reference solution. It may be

more useful to have a list of inputs that fail to match. Then you can check this list for any patterns and narrow down the error(s) in your code.

The `btest` utility (compiled from `btest.c`) performs a simple, non-exhaustive check on the functional correctness of your code. It takes a large, but fixed, set of inputs and tests your program with them. It prints out all inputs where your code fails.

`btest` is not run directly by the driver program, so you must run it by hand. Keep in mind that the tests it uses are non-exhaustive, so seeing no errors does not imply a correct solution. The driver ultimately determines your grade.

The file `README` contains additional documentation about btest. Use the command make to generate the test code and run it with the command **./btest**.

Puzzle Coding Rules

The `bits.c` file contains a skeleton for each of the 16 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 operators:

```
 = ! ˜ & ˆ | + << >>
```

A few of the functions further restrict this list. Also, you are only allowed to use constant values between 0 and 255 (`0x0` to `0xFF`).. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

You may **NOT** use any control structures such as loops, function calls, and conditionals within your code. You also may not do any casting or use any data types other than `int`. You may assume that data type `int` is 32 bits long and encodes integers in two's complement format. Both left and right shifts require a shift amount between 0 and 31, and right shifts are performed arithmetically.

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 60 points based on the following distribution:

31 Correctness of code running on one of the class machines.

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

5 Style points, based on your instructor's subjective evaluation of the quality of your solutions and your comments.

The 12 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 31.

For this lab, you can check your work in two different ways. First, the program **btest** performs SIMPLE, NONEXHAUSTIVE tests your functions for just a number of different cases. It is provided for the sake of **easy debugging** and **does not guarantee** your function's correctness. For most functions, the number of possible argument combinations far exceeds what could be tested exhaustively. Recently, Professor Bryant has created an experimental *formal verification* program, `cbit` that, in effect, tests your functions for all possible combinations of arguments. It does this by viewing each bit of the function result as a Boolean function of the bits comprising the function arguments. It uses a data structure known as *Binary Decision Diagrams* (BDDs) (R. E. Bryant, *IEEE Transactions on Computers*, August, 1986) to represent these Boolean functions in a way that the program can efficiently compare the results of your functions with those of a set of reference solutions. If the bit-level functions match, then the two C functions compute identical results. Otherwise, `cbit` can generate a *counterexample*, i.e., a set of function arguments where your function will produce a different result than the reference solution.

You **do not** invoke `cbit` directly. Instead, you will be executing the file **driver.pl**, a script used by the autolab server, to run the BDD checker and DLC and obtain your expected total score. **driver.pl** invokes a series of Perl scripts that set up and evaluate the calls to it. To invoke the individual scripts, execute

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

to check function `fun`. Execute

```
unix> ./bddcheck/check.pl -f
```

to check all of your functions.

Note: The Perl 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 (...)
```

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.

You will get full credit for a puzzle if the BDD checker determines that your solution is correct, and no credit otherwise. The formal verification provided by the BDD checker will show you that there are no bugs lurking in your code. You'll find yourself wishing you could do this kind of testing with every program you write. Unfortunately, BDDs can handle only relatively simple functions such as the ones you are writing for this assignment. Beyond this, the BDDs get too large to represent and manipulate.

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've established a maximum number of operators that you are allowed to use for each function. Assignment operators ('`=`') aren't counted. This limit is very generous and is designed only to catch egregiously inefficient solutions. You will receive two points for each function that satisfies the operator limit.

Finally, we've reserved 5 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, describing the strategy behind your solution, but they need not be extensive.

Name	Description	Rating	Max Ops
`upperBits(n)`	Pad the upper `n` bits with 1	1	10
`implication(x,y)`	`x` $\Rightarrow$ `y` using only bit operations	1	5
`byteSwap(x, n, m)`	Swap the `n`th and `m`th bytes of `x`	2	25
`sign(x)`	Return 1 if `x` is positive, 0 if zero, and -1 if negative	3	10
`sm2tc(x)`	Sign magnitude to 2's complement conversion	3	15
`bitParity(x)`	Return the parity of the number of bits set in `x`	4	20

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`tmax()`	The largest representable in 2's complement	1	4
`rempow2(x, e)`	`x` mod 2^e	2	20
`isGreater(x, y)`	`x > y`?	3	24
`satMul3(x)`	`3*x` but saturate rather than overflow	3	25
`trueFiveEights(x)`	$Round(0.625 \cdot \texttt{x})$ without overflow	4	25
`howManyBits(x)`	How many bits to show `x` in 2's complement?	4	90

Table 2: Arithmetic Functions

Part I: 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.

Part II: Two's Complement Arithmetic

Table 2 describes a set of functions that make use of the two's complement representation of integers. Again, refer to the comments in `bits.c` and the reference versions in `tests.c` for more information.

Debugging Strategy

Make sure you **DO NOT** include the `<stdio.h>` header file in your bits.c, otherwise DLC WILL get confused when you run it. You will still be able to use the `printf` statements. You may safely ignore the annoying GCC warnings in this case.

When initially debugging, execute

```
unix> make
```

to compile the **btest** binary, which you will use to test your functions. Also run

```
unix> make clean
```

each time you recompile; it is recommended that you do this and remove the traces from the previous compilation, otherwise GCC might get confused and produce unexpected results. If you feel confident about your solution, make sure your bits.c complies with the coding rules by running the **DLC** binary:

```
unix> ./dlc bits.c
```

You can also use the **-h** flag to view its help page. After DLC finishes silently without errors, then finally run the driver script

```
unix> ./driver.pl
```

to make sure your code does not upset the BDD checker. At this stage, you will also be able to see your expected total score on this lab.

Advice

Paying close attention to the precedence of different bitwise operations (available in the Ritchie & Kernighan textbook) could save you hours in doing this assignment. Also, depending on how you choose to implement `trueFiveEights(x)`, you might want to study the following description of the function `ezFiveEights(x)` in order to get started:

`ezFiveEights` -	multiplies by 5/8 rounding toward 0, without worrying about overflow conditions; in other words, you would multiply by 5 first and NOT avoid overflow
Examples:	`ezFiveEights`(11) = 6 `ezFiveEights`(-9) = -5 `ezFiveEights`(0x30000000) = 0xFE000000 (overflow)
Legal ops:	`! ˜ & ˆ | + << >>`
Max ops:	20
Rating:	3

Table 3: Description of the function `ezFiveEights(x)`

You can work on this assignment using one of the class 64-bit "Fish" machines or an Andrew linux machine since they both run similar versions of Linux and you need not be concerned with the word length (64-bit vs. 32-bit) for this lab. The BDD checker and the DLC program are distributed as Linux executables, and so you'll want to be working on a compatible Linux machine in order to use those tools.

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. The typical usage is

```
unix> ./dlc bits.c
```

Type `./dlc -help` for a list of command line options. The README file is also helpful.

- The DLC program runs silently unless it detects a problem.
- Andrew Linux machines have a program called `/usr/local/bin/dlc`, which is *not* the same as our DLC program. So always run DLC using a full path name:

  ```
  unix> ./dlc bits.c
  ```

- DO NOT 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 C declarations than is the case for C++ or 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;     /* Statement that is not a declaration */
    int b = a;  /* ERROR: Declaration not allowed here */
  }
  ```

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 you hand in.

Check the file `README` for documentation on running the `btest` program. 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, e.g., `./btest -f bitXor`. Also, the `-g` option is a nice way to get a compact summary of the correctness of each function.

Hand In 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. There are two types of handins: *unofficial* and *official* handins.

- **Unofficial handins.** As you work on the assignment you can use the driver program `driver.pl` to stream your current results to the Autolab server to be displayed on the class status Web page. The driver is the same program our autograder calls when it grades your handin. If your userid is `bovik`, then typing

  ```
  unix> ./driver.pl -u bovik
  ```

  will stream your results (in the form of an ASCII text line we call an *autoresult string*) to the Autolab server. The autoresult strings are logged, and the last autoresults from each student are periodically summarized on the class status Web page, under each student's Autolab nickname.

  The Autolab page provides options that allow you to view the class status page ("View class status page") as well as the complete history of your autoresult submissions ("View your autoresult history").

- **Official handins.** The autoresult strings sent from your copy of the driver program are unofficial and just for fun. To receive credit, you will need to upload your `bits.c` file using the Autolab option "Handin your work for credit". Each time you handin your code, the server will run the autograder on your handin file and produce a grade report (it also logs an official autoresult string for the class status page). The server archives each of your submissions and resulting grade reports, which you can view anytime using the "View your handin history and scores" option.

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.
- You **MUST** remove any extraneous print statements from your `bits.c` file before handing in.