15-213, Spring 2008
Lab Assignment L1: Manipulating Bits
Assigned: Jan. 15, Due: Wed., Jan. 30, 11:59PM

Randy Bryant (`Randy.Bryant@cs.cmu.edu`) is the lead person for this assignment.

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.

1.1 Logistics

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

1.2 Creating your Autolab Account

All 15-213 labs are being offered this term through a Web service developed by Prof. David 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-s08" 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 `15-213-staff@cs.cmu.edu` requesting an Autolab account, and someone will add you to the list.

1.3 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`.

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 AFS directory and use the Linux `tar` program to extract the files.

The file `btest.c` contains code that performs a simple, non-exhaustive check of the functional correctness of your code. The file `README` contains additional documentation about BTEST. Use the command `make` to generate the test code and run it with the command `./btest`.

The included program DLC can be used to check your solutions for compliance with the coding rules. The included programs ISHOW and FSHOW can be used to help examine the bit representations of integer and floating point numbers.

The files in the subdirectory `bddcheck` implement the BDD checker, a tool that formally verifies your code. The remaining files are used to build the program BTEST.

The `bits.c` file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton according to a strict set of programming rules, intended to help you understand how values are represented at the bit-level and how to manipulate bit patterns using standard C operations.

2 Evaluation

Your code will be compiled with GCC and exhaustively tested with the BDD checker. Your score will be computed out of a maximum of 75 points based on the following distribution:

40 Correctness of code.

30 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 15 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 40.

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 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, there is a series of Perl scripts that set up and evaluate the calls to it. Execute

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

to check function `fun`. Execute

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

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 (...)
```

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.

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.

3 Bit and Integer Manipulations

The first series of puzzles involve creating common bit patterns and manipulating two's complement representations of integers. These puzzles have the strictest set of programming rules. You may only use *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:

```
= ! ~ & ^ | + << >>
```

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 not use any unions, structs, or arrays. 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.

3.1 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.

Name	Description	Rating	Max Ops
`bitAnd(x,y)`	`x & y` using only `|` and `~`	1	8
`thirdBits()`	Create mask with every third bit set to 1	1	8
`conditional(x,y,z)`	`x ? y : z`	3	16
`logicalShift(x,n)`	Logical right shift of `x` by `n`	3	16
`isNonZero(x)`	`x != 0` without using `!`	4	10
`leftBitCount(x)`	Number of 1s on left (most significant) end of `x`	4	50

Table 1: Bit-Level Manipulation Functions.

Name	Description	Rating	Max Ops
`isTmin(x)`	Is `x` the minimum two's complement integer?	1	8
`fitsShort(x)`	Can `x` be expressed as a 16-bit integer?	1	8
`sign(x)`	Indicate whether `x` is negative, zero, or positive	2	10
`isGreater(x,y)`	`x > y`?	3	24
`ezThreeFourths(x)`	`3*x/4`	3	12
`trueThreeFourths(x)`	$3x/4$ without overflow	4	20

Table 2: Arithmetic Functions

3.2 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.

4 Floating-Point Operations

For this part of the assignment, you will implement some common single-precision 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.

4.1 Part III: Floating-Point Arithmetic

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, letting you decipher the results from the BDD checker on floating-point problems. When the BDD checker finds a

Name	Description	Rating	Max Ops
`float_abs(uf)`	|f|	2	10
`float_twice(uf)`	`2*f`	4	30
`float_f2i(uf)`	`(int) f`	4	30

Table 3: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

counterexample, it prints the decimal value of argument(s) that cause a discrepancy between the puzzle code and the reference version. To see what these bit patterns represent as a floating-point number, use FSHOW, e.g.:

```
unix>./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.

Functions `float_abs` and `float_twice` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. The IEEE standard does not specify precisely how to handle NaN's, and the IA32 behavior is a bit obscure. We will follow a convention that for any function returning a NaN value, it will return the one with bit representation `0x7FC00000`. Similarly, when function `float_f2i` encounters a number that is either too big to represent or is a NaN, then it should return `0x80000000`.

5 Advice

You can work on this assignment using one of the class fish machines or one of the the Andrew Linux servers (`ssh linux.andrew.cmu.edu`). The BDD checker and the DLC program are distributed as 32-bit Linux executables, and so you'll need to be working on a compatible Linux machine in order to use those tools. In general, we recommend that you work on the fish machines, since these are the systems you'll be using for most of the labs in this course.

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
```

- 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;     /* 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 bitAnd`. You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`. Also, the `-g` option is a nice way to get a compact summary of the correctness of each function.

The testing provided by BTEST is especially weak for the floating-point problems, where there are tricky issues of denormalized numbers, rounding, and overflow. Use the BDD checker to detect problems in your code, but then use BTEST and FSHOW (or ISHOW for integer problems) to help you better understand what's going on.

6 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.