Project Data: Manipulating Bits

Due: October 3, 2012

1 Introduction

Ariel has recently taken up C programming and wants to write programs which take advantage of bit-level representation. However, residents of the ocean have not developed particularly sophisticated compilers or computer hardware, so a lot of the operations that she needs are not supported. She's asked you to help her write those operations using operations which *are* supported.

2 Assignment

For this assignment, you will experience bit-level representations directly, as you solve a series of programming puzzles, which will test your ability to take advantage of bit-level representations. To solve all of the puzzles, you will need to employ your knowledge of bit-level operations, two's complement represenation, and IEEE floating point represenation.

Set up the support code and skeleton for this project by running the script `cs033_install data` from the command line. The script will create a directory called *data/* in your home directory that contains the files needed for the assignment.

The *bits.c* file (inside the directory that you just installed) contains a skeleton for each of the eight programming puzzles. Your assignment is to complete each function skeleton (each puzzle), subject to certain restrictions. The other files contain support code — you are welcome to look at it, but you won't need to for the assignment, and we ask that you not modify it.

For the integer puzzles, you must use only straightline code (i.e., no loops or conditionals) and a limited subset of C arithmetic and logical operators. For most puzzles, you are allowed to use *only* the following eight operators: `!~&^|+ << >>`. A few of the functions further restrict this list. Also, you may only use one-byte constants (between 0 and 255, inclusive (between hexadecimal numbers `0x00` and `0xff`)) (i.e., constants that can be represented in a single byte, using two's complement notation).

For the floating point puzzles, you may use `if` and `while` statements, any C arithmetic/logical operators, and arbitrary-size integer constants.

See the comments in *bits.c* for detailed rules and a discussion of the desired coding style.

2.1 Collaboration

The nature of this assignment is quite different from the others in this course; consequently, the normal CS033 collaboration policy will be modified for this assignment.

For this assignment:

- You may not share any code for any part of this assignment with other students in the course.

- You may not discuss the solutions to any part of this assignment with other students, even at a high level.
- You may not assist other students with debugging, or look at any other student's code. Only you or a member of the course staff may examine your code.
- You may not search for solutions to these puzzles on the internet.

3 The Puzzles

This section describes the puzzles that you will be solving in *bits.c*. For more information, see the comments in *bits.c*. 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 (but they don't satisfy the coding rules for your functions).

3.1 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.

Name	Description	Rating	Max Ops
`bitAnd(int x, int y)`	`x&y` using only `~` and `|`	1	8
`conditional(int x, int y, int z)`	compute `x?y:z`	3	16

Table 1: Bit-Level Manipulation Functions.

3.2 Two's Complement Arithmetic and Representation

Table 2 describes a set of functions that make use of the two's complement representation of integers.

Name	Description	Rating	Max Ops
`negate(int x)`	compute `-x`	2	5
`isEqual(int x, int y)`	compute `x == y`	2	5
`divpwr2(int x, int n)`	compute $x/(2^n)$	2	15
`addOK(int x, int y)`	check whether `x + y` will not overflow	3	20
`absVal(int x)`	compute the absolute value of `x`	4	6

Table 2: Arithmetic Functions

3.3 Floating-Point Operations

For the last puzzle, you will implement a cast from floating point to integer representations. For this puzzle, the bits of the floating-point argument are passed as an integer of type `unsigned`.

Your code should perform the bit manipulations to "decode" this representation and convert it to a 32-bit signed two's complement integer. The coding rules for this puzzle are somewhat different than the integer puzzles; see the comments in *bits.c* for more details.

Table 3 describes a function that operates on the bit-level representations of floating-point numbers.

Name	Description	Rating	Max Ops
`float_f2i(unsigned uf)`	Compute `(int)f`	4	30

Table 3: Floating-Point Functions. The variable `f` represents a float whose bit-level representation is the same as `uf`.

`float_f2i` must handle the full range of possible argument values, including not-a-number (NaN) and infinity. If the input is NaN or is too large to store in a 32-bit signed integer, you should return `0x80000000u`. To convert regular floating-point numbers to integers, truncate all digits after the decimal point.

The `fshow` program provided with the stencil code helps you understand the structure of floating point numbers. To compile `fshow`, switch to your project working directory and type:

```
make fshow
```

You can use `fshow` to see what an arbitrary pattern (unsigned hex or decimal integer) represents as a floating-point number.

```
$ ./fshow 2080374784

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

`fshow` can also be used to see how a floating point value will be represented

```
$ ./fshow 2080374784

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

4 General Advice

- **Read Chapter 2 of the textbook**. The information in this chapter will help greatly with many of the puzzles. Any time you spend reading the book before beginning some of the puzzles will reduce time you spend poring over them later.
- If you get stuck, ask the TAs! Course staff members will try to point you in the right direction without giving away the answer.

- Use the provided data project checker (`./dpc bits.c`, described in section 6.1.2) to make sure that your code follows the rules for each of the puzzles. A consequence of using this checker is that you will not be able to include the `<stdio.h>` header file in your code. Doing so confuses `dpc` and results in 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). Remove all print statements before handing in.

- You are allowed to use conditionals and comparison/boolean operators in the last problem, `float_f2i` (but only in that problem). When in doubt as to whether you can use a given operator, you can test whether it is valid by using it in the function in question and running `./driver.pl`, which will tell you if you have invalid operations in your code. If you're still confused, you're of course more than welcome to email the TAs and ask.

5 Hints

- You are only allowed to declare constants that fit in one byte of data. However, this does *not* mean that you may not *use* constants that don't fit in one byte of data. If you find yourself needing a large constant, consider how you can generate such a value using only small constants and binary operators.

- You will find *bit masks* invaluable in this assignment. A bit mask is an integer whose binary representation is intended to combine with another value using either `&` or `|` to extract or set a particular bit or set of bits respectively. For example, to extract the fourth bit of an integer:

```
int mask = 0x8;         /* 1000 in binary                          */
int value = 0xc;        /* 1100 in binary                          */
return mask & value;    /* returns 0x8, indicating that the fourth bit
                           of the "value" variable was set          */
```

6 Grading

Your score will be computed out of a maximum of 40 points based on the following distribution:

21 Correctness points.

16 Performance points.

3 Style points.

Correctness points. The 8 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals 21. 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. This limit is designed to catch inefficient solutions; a few of the problems require some clever thinking to satisfy the limit. You will receive two points for each correct function that satisfies the operator limit.

Style points. Finally, we've reserved 3 points for a qualitative 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, but they need not be extensive.

6.1 Evaluating Your Work

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

6.1.1 `btest`

The `btest` program checks the functional correctness of the functions in *bits.c*. To build and use it, type the following two commands within the *data/* directory:

```
make btest
./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:

```
./btest -f bitAnd
```

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

```
./btest -f bitAnd -1 7 -2 0xf
```

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

6.1.2 `dpc`

The `dpc` binary 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:

```
./dpc 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:

```
./dpc -e bits.c
```

causes `dpc` to print counts of the number of operators used by each function. Type `./dpc -help` for a list of command line options.

Note that this compiler enforces a stricter form of C declarations than `gcc` does. In particular, any declaration in a given scope (what you enclose in curly braces) must appear before any statement that is not a declaration. For example, `dpc` will reject 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 */
}
```

At one time, this was required by the official C language specification. Thankfully, the modern standard has been updated to no longer require this restriction.

6.1.3 `driver.pl`

The `driver.pl` script uses `btest` and `dpc` to compute the correctness and performance points for your solution. It takes no arguments:

```
./driver.pl
```

`driver.pl` will be used to evaluate your solution.

7 Handing In

To hand in your project, run the command

```
cs033_handin data
```

from your project working directory. You should hand in your *bits.c* file only; you need not include any other files.

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.