Spring 2019

Lab 1b: Bits in C

Assigned: Friday, April 12, 2019

Due Date: *Monday, April 22, 2019 at 11:59 pm*

Video(s): This video (../../videos/tutorials/lab1-print_binary.mp4) (with captions) (https://www.youtube.com/watch?v=R0R4MDG3-mM) shows how to use the optional helper function `print_binary()` as well as a few more bit tricks you might find helpful for this lab.

Overview

Learning Objectives:

- Gain familiarity with data representation at the level of bits.
- Gain practical knowledge of bit manipulation in C.

You will solve a series of programming "bit puzzles." Many of these may seem artificial, but bit manipulations are very useful in cryptography, data encoding, implementing file formats (e.g. MP3), and certain job interviews.

Code for this lab

Browser: Download here (lab1b.tar.gz)

Terminal: `wget https://courses.cs.washington.edu/courses/cse351/19sp/labs/lab1b.tar.gz`

Unzip: Running `tar xzvf lab1b.tar.gz` from the terminal will extract the lab files to a directory called `lab1b` .

Lab 1b Instructions

`bits.c` contains skeletons for the programming puzzles, along with a comment block that describes exactly what the function must do and what restrictions there are on its implementation. Your assignment is to complete each function skeleton using:

- only straightline code (i.e., no loops or conditionals)
- a limited number of C arithmetic and logical operators (you can also use shorthand versions of "legal" operators--ex. you can use ++ and += if + is legalå)
- no constants larger than 8 bits (i.e., 0 - 255 inclusive)--however, you are allowed to combine constants to values greater than 255 or less than 0. e.g. 250 + 250 = 500, so long as the operator you're using to combine the constants is listed as "legal" at the top of the method you're writing
- as many "(", ")", and "=" as you need

The intent of the restrictions is to require you to think about the data as bits - because of the restrictions, your solutions won't be the most efficient way to accomplish the function's goal, but the process of working out the solution should make the notion of data as bits completely clear.

Bit Manipulation Puzzles

The table below describes a set of functions that manipulate and test sets of bits. The Rating column gives the difficulty rating (the number of points) for each puzzle and the Description column states the desired output for each puzzle along with the constraints. 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.

Rating	Function Name	Description
1	bitAnd	Compute `x & y` using only `~` and `|`. **Hint:** DeMorgan's Law.
1	bitXor	Compute `x ^ y` using only `~` and `&`. **Hint:** DeMorgan's Law.
1	thirdBits	Return an int with every third bit (starting from the least significant bit) set to 1 (*i.e.* 0100 1001 0010 0100 1001 0010 0100 1001_2). **Hint:** Remember the restrictions on integer constants.
2	getByte	Extract the `n` th byte from `int x`. **Hint:** Bytes are 8 bits.
3	logicalShift	Shift `x` to the right by `n` bits, using a *logical* shift. You only have access to *arithmetic* shifts in this function.
3	invert	Invert (0 ↔ 1) `n` bits from position `p` to position `p+n-1`. **Hint:** Use a bitmask.
Extra Credit:		
4	bang	Compute `!x` without using the `!` operator. **Hint:** Recall that 0 is false and anything else is true.

Two's Complement Puzzles

The following table 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.

Rating	Function Name	Description
2	sign	Return 1 if positive, 0 if zero, and -1 if negative. **Hint:** Shifting is the key.
3	fitsBits	Return 1 if `x` can be represented as an `n` -bit, two's complement integer. **Hint:** -1 = ~0.
3	addOK	Return 1 if `x+y` can be computed *without* overflow. **Hint:** Think about what happens to sign bits during addition.
Extra Credit:		
4	isPower2	Return 1 if `x` is a power of 2, and 0 otherwise.

Floating Point Puzzles

The following table describes a set of functions that make use of the IEEE 754 floating point representation. **Note:** these functions use `unsigned int` to pass the floating point numbers, but you should interpret their bit-level representations as floating point values.

Rating	Function Name	Description
2	floatNegate	Return the bit-level equivalent of the expression `-f` for floating point argument `f` . `NaN` should be returned for argument `NaN` .
2	floatIsEqual	Compute `f == g` for floating point arguments `f` and `g` . `NaN` cannot be equal to any float. ±0 are equal.
Extra Credit:		
4	floatInt2Float	Return the bit-level equivalent of `(float) x` . **Warning:** you will need to implement round to nearest, ties to even (https://en.wikipedia.org/wiki/IEEE_754#Rounding_rules).

Checking Your Work

We have included the following tools to help you check the correctness of your work:

- We have included a `print_binary` function, which takes an integer and outputs its binary representation. This can be useful in debugging your code, but its use is optional and all calls to the function should be commented out in your final submission. See the video link at the top of this page for usage examples.

- `btest` is a program that ***checks the functional correctness of the code*** in `bits.c` . To build and use it, type the following two commands:

```
$ make
$ ./btest
```

Notice that you must rebuild `btest` each time you modify your `bits.c` file. (You rebuild it by typing `make`.) 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 bitXor
```

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

```
$ ./btest -f bitXor -1 7 -2 0xf
```

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

We may test your solution on inputs that btest does not check by default and we will check to see that your solutions follow the coding rules.

- The `make` command additionally produces two helper executables called `ishow` and `fshow` that can be used to view conversions between decimal values and bit representations for integers and floating point numbers, respectively. For more information about using them, see the end of the `README` file.
- `dlc` 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:

```
$ ./dlc bits.c
```

Note: dlc will always output the following warning, which can be ignored:

```
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includable file /usr/include/stdc-predef.h.
```

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:

```
$ ./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.

- The `dlc` program enforces a stricter form of C declarations than is the case for C++ or that is enforced by `gcc`. In particular, in a block (what you enclose in curly braces) all your variable declarations must appear before any statement that is not a declaration. For example, `dlc` 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 */
}
```

  Instead, you must declare all your variables first, like this:

```
int foo(int x) {
  int a = x;
  int b;
  a *= 3;
  b = a;
}
```

- Do NOT include the `<stdio.h>` header file in `bits.c`, as it confuses `dlc` and results in some non-intuitive error messages. You will still be able to use `printf` for debugging without including the `<stdio.h>` header, although `gcc` will print a warning that you can ignore.

Advice

- Puzzle over the problems yourself, it is much more rewarding to find the solution yourself than stumble upon someone else's solution.
- If you get stuck on a problem, move on. You may find you suddenly realize the solution the next day.
- There is partial credit if you do not quite meet the operator limit, but often times working with a suboptimal solution will allow you to see how to improve it.
- You can use `gdb` (GNU debugger) on your code. See this transcript (lab1-gdb.html) for an example.

Lab 1b Reflection

Make sure your answers to these questions are included in the file `lab1reflect.txt` !

Assuming that `x = 351` , as in the original code of `lab0.c` :

1. Find a *positive* value of `y < x` such that `x & y = 0` . Answer in hex. [2 pt]
2. Find a *negative* value of `y` such that `x ^ y = -1` . Answer in decimal. [2 pt]
3. Consider the following two statements:
   - `y = -1;`
   - `y = 0xFFFFFFFF;`

   Is there a difference between using these two statements in your code? Explain. If there is a difference, make sure to provide an example. [3 pt]

Submission

Please submit your completed `bits.c` and `lab1Breflect.txt` files to the assignments page (../submit.php).