

Project Data: Manipulating Bits

Due: September 27, 2017 at 11:59pm

1 Introduction	1
2 Assignment	1
2.1 Collaboration	2
2.2 TA Hours	3
3 The Puzzles	3
3.1 Bit Manipulations	3
3.2 Two's Complement Arithmetic	3
3.3 Boolean Operations	4
3.4 Floating Point Operations	4
4 General Advice	4
5 Hints	5
6 Grading	5
6.1 Evaluating Your Work	6
6.1.1 btest	6
6.1.2 dpc	6
6.1.3 driver.pl	7
7 Handing In	7

1 Introduction

Mike and Sully need to pass their Monsters University computer science exams. However, they have forgotten 9 essential functions that they need for their test. Additionally, they have also forgotten many basic coding operators. Help them reimplement these functions so they can graduate!

2 Assignment

For this assignment, you will 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 and two's complement representation.

Set up the support code and skeleton for this project by running the script `cs0330_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 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 all puzzles in this assignment, 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)

The exception to this rule is `float_f2i()`, for which you can use any integer operators, if/else statements (including `x ? y : z` syntax), loops, and constants of any size.

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

Note that **this assignment is not fully supported on Mac, because the dpc binary will not work properly**. If you choose to work on this assignment with your personal computer, just make sure it runs on a department machine before you hand it in!

2.1 Collaboration

The nature of this assignment is quite different from the others in this course; consequently, the normal CS0330 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.

2.2 TA Hours

The TAs are also limited in how they can help you. They are only allowed to give specific hints for each puzzle, and they are only allowed to help debug your code to make sure your **syntax** is correct. They may also talk about topics covered in lecture.

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). For each of the tables, 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.

3.1 Bit Manipulations

Table 1 describes a set of functions that manipulate and test sets of bits.

Name	Description	Rating	Max Ops
<code>bit_and(int x, int y)</code>	<code>x&y</code> using only <code>~</code> and <code> </code>	1	8
<code>add_ok(int x, int y)</code>	check whether <code>x + y</code> will overflow	3	20

Table 1: Bit Manipulation Functions

3.2 Two's Complement Arithmetic

Table 2 describes a set of functions that make use of two's complement arithmetic.

Name	Description	Rating	Max Ops
<code>negate(int x)</code>	compute <code>-x</code>	2	5
<code>div_pwr_2(int x, int n)</code>	compute <code>x/(2ⁿ)</code>	2	15
<code>abs_val(int x)</code>	compute the absolute value of <code>x</code>	5	6

Table 2: Arithmetic Functions

3.3 Boolean Operations

Table 3 describes a set of functions that implement basic boolean operations.

Name	Description	Rating	Max Ops
<code>is_equal(int x, int y)</code>	compute <code>x == y</code>	2	5
<code>conditional(int x, int y, int z)</code>	compute <code>x?y:z</code>	3	16
<code>bang(int x)</code>	compute <code>!x</code>	4	12

Table 3: Boolean Operation Functions

3.4 Floating Point Operations

Table 4 shows the final puzzle which employs both floating point and two's complement representation. You have access to certain control statements (`if` and `while`) for this function.

Name	Description	Rating	Max Ops
<code>float_f2i(unsigned uf)</code>	convert floating point to integer	5	30

Table 4: Floating Point Operations

4 General Advice

- **Read Chapter 2 of the textbook and the lecture slides on data representation.** The information in these resources will help greatly with many of the puzzles. Any time you spend reading these materials 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.

- 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 100 points based on the following distribution:

54 Correctness points.

36 Performance points.

10 Style points.

Correctness points. The 9 puzzles you must solve have been given a rating between 1 and 5. Each puzzle is worth 2 times its rating, and their weighted sum totals 54. We will evaluate your functions using the `btest` program, which is described in the next section. Barring extenuating circumstances, 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 four points for each correct function that satisfies the operator limit.

Style points. Finally, we've reserved 10 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 bit_and
```

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

```
./btest -f bit_and -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:

```
cs0330_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.

Important note: *If you have already handed in your assignment, but plan to hand in again after the TAs start grading at noon on Saturday, September 30rd, you must run `cs0330_grade_me_late data` to inform us. You must run the script by noon on 9/30. If you run this script, you will get grades back later than other students.*

If you do not run this script, the TAs will proceed to grade whatever you have already handed in, and you will receive a grade report with the rest of the class that is based on the code you handed in before we started grading.

If something changes, you can run the script with the `--undo` flag (before noon on 9/30) to tell us to grade you on-time and with the `--info` flag to check if you're currently on the list for late grading.

These instructions apply to all projects unless otherwise stated on the handout.