Computer Systems
baiCOSY06, Fall 2011
Data Lab: Manipulating Bits
Assigned: Sep. 05, Due: Wed., Sept. 13, 12:59

Arnoud Visser (`A.Visser@uva.nl`) is the lead person for this assignment. Carsten van Weelden and/or Maarten van der Velden will be the teaching assistants.

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of 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.

2 Logistics

For this assignment you are requested for pairs. All handins are electronic. As part of this assignment, each pair is requested to keep a labjournal. See section 6 for more details. Clarifications and corrections will be posted on the course Web page.

3 Linux server

At the Universiteit van Amsterdam a dedicated server is available for programming in a Linux environment. This server is called 'deze.science.uva.nl'. Note that this server is only available in the edu-domain. To access the server from elsewhere, one first has to build up an vpn-connection with the UvAID, followed by a login into the server sremote.science.uva.nl. The server sremote is a file-server with only limited computer power. In addition, no programming tools are installed on sremote. For programming, please continue with `ssh deze`. Note that sremote does not allow X-forwarding, so graphical applications can only be used locally.

The classroom A1.16A is provided with Windows machines equiped with Exceed. Locate Exceed via the Start-Programs-Open Text Exceed button, and place a shortcut on your Desktop. When you start Exceed,

you get the graphical Login-screen towards the server 'deze.science.uva.nl'. Once logged in on deze, you can start a webbrowser by typing in `firefox &`.

4 Handout Instructions

Each group of students will get a personalized puzzle. To obtain your group's puzzle, one (and only one) of the group members should point deze's firefox webbrowser to the following address:

`http://deze.science.uva.nl:15213`

Fill out the HTML form with email addresses and names of your team members and then submit the form by clicking the "Request" button. The request daemon will build your puzzle and return it immediately to your browser in a compressed archive called `puzzle`*k*`.tgz`, where *k* is the unique number of your puzzle.

Start by copying `puzzle*.tgz` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
unix> tar zxvf puzzle*.tgz
```

This will cause a number of files to be unpacked in the directory. The only file you will be modifying and turning in is `bits.c`.

The `bits.c` file contains a skeleton for each of the 15 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles (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 eight operators:

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

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than 8 bits. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

5 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`. You will solve puzzles with an increasing difficulty, it is not required that you solve them all.

5.1 Bit Manipulations

The first part of the assignment is a set of functions that manipulate and test sets of bits. See the comments in `bits.c` for more details on the desired behavior of the functions. In the comments also the "Rating" and 'Max ops" fields are specified. 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. 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.

5.2 Two's Complement Arithmetic

The intermediate part of the assignment involve 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.

5.3 Floating-Point Operations

For the last 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.

Functions `float_neg` 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`.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
unix> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
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.

6 Labjournal

To solve the puzzles, you need creativity, knowledge and experimental skills. It is required that you document your progress on finding a solution on the puzzles in a labjournal. See `http://www.science.uva.nl/~arnoud/education/ZSB/2011/Experiment/labjournaal.html` for some general pointers about what a labjournal should be contain.

7 Evaluation

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

16 Correctness points.

12 Performance points.

12 Academic style points.

Correctness points. The 6 puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 16. 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 for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions. You will receive two points for each correct function that satisfies the operator limit.

Style points. Finally, your labjournal will be evaluated by your tutors on the following criteria:

- *Experiment description*: describe who you are, the assignment in your own words and the circumstances in which you perform the experiment.
- *Progress*: describe the logical steps you made to solve assignment. Include tests, code-snippets, observations
- *Analysis*: describe in retrospect how you could have solved the assignment easier, and what the solution direction could be the towards a higher performance.
- *Style*: is the labbook well organized and well written.

Autograding your work

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

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

```
unix> make
unix> ./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:

```
unix> ./btest -f bitAnd
```

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

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

Check the file `README` for documentation on running the `b`test program.

- **`dlc`:** This 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:

  ```
  unix> ./dlc 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:

  ```
  unix> ./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.

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. It takes no arguments:

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

  Your instructors will use `driver.pl` to evaluate your solution.

8 Handin Instructions

To handin your solution, type:

```
make handin TEAM=teamname
```

in your directory in the edu-domain.

9 Advice

- 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 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 */
}
```