Computer Systems
Fall 2018
Lab Assignment 1: Manipulating Bits
Assigned: September 4, Due: Wed., Sept. 12, **20:00**.

Giulio Stramondo (giuliostramondo+kicks2018@gmail.com) is the lead person for this assignment.

1 Introduction

The purpose of this assignment is to challenge your familiarity 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

Please read the entire assignment carefully before you start working.

For this assignment you are requested to work in pairs. Each pair is requested to submit code and a "lab report" (see section 6 for more details). Clarifications and corrections will be posted on the Canvas course page, if the need arises.

3 Machines

This course expects that you bring a laptop with you. Please find below the alternative ways to use your laptop, ordered by the ease-of-use for the CS course.

3.1 Native Linux machine or dual-boot machine

If you already have a Linux machine, you are good to go. If you have a dual-boot machine, please remember that all Computer Systems labs require the use of the Linux OS, so please boot your machine when working

on the CS assignments.

Before you proceed further, please use `sudo apt-get install`, to install `valgrind` and `kcachegrind`.

3.2 Virtual machine

You should already have a virtual machine installed.

If not, in datanose, you will find under Tools→Sofware a list of packages (`https://datanose.nl/#byod`). One of those is VMWare, select Player 7 (Linux or Windows) or Fusion (Mac).

Within the player, you open the Virtual Machine file that you have downloaded from: `http://sbt.science.uva.nl/bterwijn/KI2016`.

3.3 Mac machine

Note that if you have a Mac, you might still be able to run the assignments natively. However, due to the large diversity of compilers and tools, as well as the different ways to configure them, the TAs cannot guarantee support for Mac machines. You are free to use them, but you must manage the installation, compilers, and tools (gcc, gdb, valgrind, kcachegrind) yourselves.

Note that special attention must be paid to the `gcc` compiler, which is often (on a Mac) NOT the expected version.

3.4 Student server `acheron`

Inside the uva-domain you could access a Linux-server dedicated for students (`acheron.ic.uva.nl`). To access the server `acheron`, one first has to use an VPN-connection with your UvAID, even when you are in the fnwi-domain. Please make sure your VPN client is working correctly before attempting this.

The server is always available and could be used for programming. However, it is a relatively old machine and, given the large numbers of students enrolled in this course, we strongly recommend you use your own machine as much as possible.

Still, when needed, to login from your local machine to `acheron`, you need a program such as `ssh` (or `putty` under Windows). To copy files to and from the machine, you need a program such as `scp` (or WinSCP under Windows).

4 Submission instructions

Each pair of students will get a personalized puzzle. To obtain your group's puzzle, ask the TA to assign a number for your team-of-2. As soon as you got a puzzle number, you can download the puzzle from Canvas.

To work locally (i.e., using a native or VM Linux on your own machine), you should just download the code from Canvas, store it in an appropriate folder (e.g., datalab) and decompress it.

To execute on `acheron`, you must download the code from Canvas to your local machine, copy the code from the local machine to the server (using `scp`), and finally login to the server (using `ssh`) to work.

IMPORTANT: The only file you will be modifying and turning in is `bits.c`, which is in the folder `puzzle??` (where "??" is replaced by your puzzle number). Note that when you are working on acheron, you will have to retrieve the file (also using `scp`) to be able to submit it in Canvas.

The `bits.c` file contains a skeleton for each of the 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`. The puzzles increase in difficulty, but you do not necessarily need to solve them in order.

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; the comments also specify a "Rating" and a "Max ops" restriction. 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.

The included program `ishow` helps you translate integer numbers, signed and unsigned, into their binary representation. To compile `ishow`, make sure you are in the puzzle directory (`datalab/puzzle??` and type:

```
make
```

You can use `ishow` like this:

```
./ishow 75
Hex = 0x0000004b,Signed = 75,Unsigned = 75
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

5.3 Floating-Point Operations

For the last phase 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.

Table 1 describes a couple of examples 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.

Name	Description	Rating	Max Ops
`float_neg(uf)`	Compute `-f`	2	10
`float_i2f(x)`	Compute `(float) x`	4	30
`float_twice(uf)`	Compute `2*f`	4	30

Table 1: Floating-Point Functions. Value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

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. If you've already compiler `ishow`, you can skip the compilation. Otherwise, to compile `fshow`, switch to the puzzle directory (`datalab/puzzle??` and type:

```
make
```

You can use `fshow` to see what an arbitrary pattern 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)
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

6 Lab Report

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 lab report.

See the page 'Het labboek'[1] on the website on Academic Skills for some general pointers about what a lab journal/lab report should contain.

For this assignment, you will receive a template of such a lab report, and you will be requested to answer the questions in the provided text (a bit like "fill in the blanks", only with longer answers). You will be graded for these answers only, but make sure they are coherent, they make sense in the context of the report you submit, and they are readable. We do not correct grammar mistakes, but text that is not readable will not be graded. Correct answers need not be excessive in length (usually, one paragraph per (sub)question suffices), and can include images, data, or (psuedo)code snippets, if needed.

Each question has an allocated score. When a total score (i.e., the sum of all questions' scores) of a report exceeds 10, you simply have more opportunities to get a 10. Moreover, if your grade does exceed 10, excess points can be used to improve scores for other reports (your TA will manage these grades). Thus, it is in your overall advantage to try your best to answer all the questions.

Once you have filled in your answers, generate the PDF file of the report using your favorite Latex compiler (e.g., online, Overleaf or SharedLatex). The PDF of your report is the file you hand in as your lab report.

Like most of the course material, the lab report is provided in English. If you have problems writing your answers in English please inform your TA or contact the coordinators (Giulio, Ana) asap.

7 Evaluation

Your score will consist of two grades: code and report. Your code is graded for correctness and performance.

Correctness points These are the points you get for solving your team's six puzzles. Each puzzle has a difficulty rating between 1 and 4, such that their weighted sum is 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 (for that puzzle) otherwise.

Performance points. Each function also has a recommended maximum number of operators that you are allowed to use. This limit is usually generous, and is designed only to catch the most inefficient solutions. You will receive 2 points for each *correct function* that satisfies the operator limit.

Report points. Your overall lab report will be evaluated by the TAs. They will focus on the correctness and completeness of your answers, as well as on their fit in the report. Therefore, please read your report before you submit: it should be readable!

[1] `http://www.practicumav.nl/onderzoeken/labboek.html`

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, respectively:

  ```
  make
  ./btest
  ```

  Important: you must rebuild `btest` (i.e., by using `make`) every 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 a more in-depth documentation for running the `btest` 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. To be able to use it, make sure it has executable rights. To set that, in your puzzle folder (datalab/puzzle??), type:

  ```
  chmod u+x dlc
  ```

  Next, the typical usage is:

  ```
  ./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:

  ```
  ./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. To be able to use it, make sure it has executable rights. To set that, in your puzzle folder (datalab/puzzle??), type:

```
chmod u+x driver.pl
```

The driver takes no arguments:

```
./driver.pl
```

This will output the combined number of points for correctness and performance.

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

8 Submission instructions

To submit your solution, upload your lab report and code (`bits.c`) on Canvas. Please make sure you will rename the code as `bits_XXXXXX_YYYYYY.c` where "XXXXXX" and "YYYYYY" stand for the student numbers in your pair. Likewise, the report should be named `Report_DataLab_xxxxx_yyyyy.pdf`.

Delayed submissions (up to 4h late after 20:00) are penalized with 1 point per every 1h. This applies to whichever part of the submission is delayed (solution, report, or both). Submissions that are received later than 4h (i.e., after midnight) are NOT graded (thus, will receive 0 points).

9 Advice

- Homework assignments 2.58 - 2.70 are the same style of puzzles as the one in this assignment. Also look at page 192 for the example code of problem 2.42.
- 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 */
}
```