T-107-TOLH Data Lab Fall 2025

Assigned: Sep 15^{th} Due: Okt. 4^{th} at 11:59PM

Primary instructor is Dr. Gylfi Þór Guðmundsson

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." Some of these puzzles are quite artificial while others teach us how low-level logic gates are used to implement logic operations and perform more complex tasks. Regardless of the puzzle, we hope you'll find yourself thinking much more about bits and bytes while working your way through them.

2 Logistics

This is an individual project. All deliverables are handed in electronically on Skel by running the command `make handin` in the extracted folder (see later). You can check if when you did your last handin by running the command `make check`. Any and all clarifications and corrections to this assignment will be posted on Canvas and will be announced on Discord.

3 Handout Instructions

- This assignment should be done on Skel, `skel.ru.is`
- Skel is reachable via SSH (using command line or Putty). To authenticate you use your regular RU username (first part of your ru e-mail address) and password.
- The compressed tar file (i.e. the codebase) is available at: `/labs/tolh25/assignments/tolh25-datalab-handout.tar.gz`

Once you have connect to Skel via SSH, navigate to your home directory and extract the tar file. This will create a folder called `tolh25-datalab-handout` with several files. **NOTE! The only file you will be modifying is** `bits.c`. The rest of the files are just code that allows you to test if your solution is correct and can help you in the task of solving the problems. At this point you can edit the `bits.c` using your favorite editor (vim, emacs or nano) **but we highly recommend you read the rest of the instructions.**

The `bits.c` file contains a skeleton for each of the 16 programming puzzles. Your assignment is to complete the function skeleton using only *straightline* code for the integer puzzles (**i.e., no loops or conditionals like if/else or ternary statements**) and you are allowed only 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.

4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`. The solutions to first two, the * marked puzzles, are given in class.

4.1 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. 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.

#	Name	Description	Rating	Max Ops
2	`allOddBits(x)`*	Return 1 only if all odd bits are set in `x`	2	12
4	`bitOr(x,y)`	Compute `x | y` using only `&` and `~`	1	8
7	`getByte(x,n)`	Return only the `n`-th byte from `x`	2	6
8	`replaceByte(x,n,c)`	Replace the `n`-th byte in `x` with `c`	3	10
9	`isAsciiDigit(x)`	Return 1 if x is one of the ASCII digits ('0' to '9')	3	15
11	`logicalShift(x,n)`	Shift `x` logical right by `n` bits	3	20
13	`mystery1(x,n,m)`	We need to figure this one out.	2	25
16	`bang(x)`	Compute `!x` without using the `!` operator	4	12
	Rating total		20	

Table 1: Bit-Level Manipulation Functions.

4.2 Two's Complement Arithmetic

Table 2 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.

#	Name	Description	Rating	Max Ops
1	`negate(x)`*	Compute `-x` *Solution given in class*	2	5
3	`tmax(x)`	Return the TMax value of a 4byte Int (shift allowed).	1	4
5	`isEqual(x,n)`	Returns 0x1 only if `x` and `y` are the same	2	5
6	`isTMax(x)`	Returns 0x1 only if `x` is TMax (no shifting)	2	10
10	`subOK(x,y)`	Returns 0x1 only if adding `x` and `y` is safe (no overflow)	3	20
12	`satMul2(x)`	Compute `x` times two, but saturate to TMin/TMax	3	20
14	`mystery2(x,n)`	We need to figure this one out.	3	25
	Rating total		16	

Table 2: Arithmetic and Unknown Functions

4.3 Floating-Point Operations

For this 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.

Table 3 describes a set 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
15	`float_twice(uf)`	Compute `f` * 2 in IEEE 754 32bit format	4	30
	Rating total		4	

Table 3: Floating-Point Functions. Value `f` is the 32 bit floating-point number having the same bit representation as the 32 bit 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. 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.

5 Evaluation

The grading is a little bit complicated. First, 70% of the grade is calculated based on the number of points you collected (see below). That means 30% is based on `STYLE`, i.e., how well you indent and comment your code.

Grade on the 70% point part	You need
11	>56 points
10	>50 points
9	>45 points
7	>35 points
5	>25 points

Table 4: Examples of calculating the 70% grade based on the number of points you get.

The goal of the comments is to show us (the graders) that you understand your solution. **Focus on answering the WHY and not the WHAT**. We can see what the code does, what you want to tell us is why it is done. **–> NOTE! Comments should be well done always. Make sure you comment each puzzle that you solve.** Puzzles that you skip (i.e., leave as is) do not need to be commented but if you leave out too many we will not give full grade for style.

The two types of points:
There are two types of points, *Performance points* and *Correctness points*. For every puzzle you solve with fewer than the maximum number of operators allowed you will get 2 performance points. There are 16 puzzles so a **maximum of 32 Performance points are available**.

Each puzzle has a complexity ranking, we may not always agree on that ranking but it is what we use to get *Correctness points.* For each puzzle you solve correctly (regardless of the number of operators used) you will get the rank value as *Correctness points*. The total ranking of all 16 puzzles is 40 so there are **40 Correctness points available**.

This means that there is a total of 72 points available. However, you DO NOT have to do every puzzle. Basically, you need 50 points to get the full 10 out of 10 for the 70% grade based on points. That means you can skip a total of 22 points. In Table 4 you will find examples of how many points you need for a given grade.

The final grade is then calculated based on the 70% weight of the points and the 30% grade we give for the comments and style. IF you have done extra work, and we have not penalized you for the style points, we will also grant an equivalent bonus for the style part.

—> NOTE! You can not get a higher grade than 11 total.

Auto-grading / Testing 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`:** While using `btest` is great for checking the puzzle you are working on solving rigth now, the driver is what you would use to test ALL of your solutions in one go. The driver program uses both `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.

6 Handin Instructions

When you are ready to hand-in your work, obviously BEFORE the deadline, you need to run the following:

```
[rulogin@skel ~/tolh25-datalab-handout]$ make handin
```

The check will be run for you but you can also check again by running:

```
[rulogin@skel ~/tolh25-datalab-handout]$ make check
```

and the date and time of your last hand-in should be printed out. You can run this hand-in process as often as you like but ONLY THE LAST deliverable is stored. **MAKE SURE that the solutions you hand in are working and that the code complies etc. or you will not get marks for any of the puzzles that you solved.**

7 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 */
}
```

8 The "Beat the Prof" Contest

For fun, we may offer an optional "Beat the Prof" contest that allows you to compete with other students and the instructors to develop the most efficient puzzles. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for each puzzle are winners!

Once the competition opens (last 3 days) you can submit your entry to the contest like this:

```
unix> ./driver.pl -u ''Your Nickname''
```

Nicknames are limited to 35 characters and can contain alphanumerics, apostrophes, commas, periods, dashes, underscores, and ampersands. You can submit as often as you like. Your most recent submission will appear on a real-time scoreboard, identified only by your nickname. You can view the scoreboard by pointing your browser at

```
http://skel.ru.is/tolh25-datalab.html
```

Do not change the functionality of the driver.pl file to submit wrong information to the scoreboard. Such actions are monitored and will not be taken lightly.