➢ 2nd Homework

- Overview

➢ **Released date**: 10/10 (Mon.)
➢ **Due date**: 11/7 (Mon.)
➢ **Where to submit**: to e-class (http://eclass.seoultech.ac.kr)
  - Late submission is not allowed.
➢ **Assigned score**: 7 points

For this project, this module refers to the original source provided by CMU[1]. In this module, we build our own environments to conduct projects. The environments are provided to the students so that they can check their results are correct or not themselves.

The purpose of this project 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.

- Handout Instructions

Start by copying datalab-handout.tar to a directory on a Linux machine in which you plan to do your work. Then give the command

```
linux> tar xvf datalab-handout.tar.
```

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.

In bit.c, you have different problems to solve. This project consists of different three problems. 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.

[1] http://csapp.cs.cmu.edu/3e/datalab.pdf

- Evaluation

Some autograding tools are include to help you check the correctness of your work. All the evaluations are conducted by the autograding tools. There are three measures to evaluate: 1) correctness of the program, 2) performance of the program, and 3) coding style.

If you submit all the three problems, they are evaluated together with autograding tools.

- How to check your results

  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:

  ```
  linux> make
  linux> ./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:

  ```
  linux> ./btest –f bitAnd
  ```

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

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

  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:

  ```
  linux> ./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:

  ```
  linux> ./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:

linux> ./driver.pl

- Problem #1

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
`bitAnd(x,y)`	`x & y` using only `|` and `~`	1	8
`getByte(x,n)`	Get byte `n` from `x`.	2	6
`logicalShift(x,n)`	Shift right logical.	3	20
`bitCount(x)`	Count the number of 1's in `x`.	4	40
`bang(x)`	Compute `!n` without using `!` operator.	4	12

Table 1: Bit-Level Manipulation Functions.

- Problem #2

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
`tmin()`	Most negative two's complement integer	1	4
`fitsBits(x,n)`	Does `x` fit in `n` bits?	2	15
`divpwr2(x,n)`	Compute $x/2^n$	2	15
`negate(x)`	`-x` without negation	2	5
`isPositive(x)`	`x > 0`?	3	8
`isLessOrEqual(x,y)`	`x <= y`?	3	24
`ilog2(x)`	Compute $\lfloor \log_2(x) \rfloor$	4	90

Table 2: Arithmetic Functions

- Problem #3

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
`float_neg(uf)`	Compute `-f`	2	10
`float_i2f(x)`	Compute `(float) x`	4	30
`float_twice(uf)`	Computer `2*f`	4	30

Table 3: 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.

- Submission Instructions
  - Rename your bit.c into bit_*studentID*.c
  - Submit bit_*studentID*.c into e-class system in SeoulTech: http://eclass.seoultech.ac.kr.

- **CAUTION: penalty for cheating**
  - If cheating is detected, you will get an F
  - What is cheating?
    - Sharing code: by **copying**, **retyping**, **looking at**, or supplying a file
    - Searching the **Web for solutions**
    - Helping your friends line by line
  - Some **automatic tools** are used to detect cheating
    - Copying and making **some modifications will also be automatically detected**