Questions:

Q1: Assembler:

Program an assembler for the aforementioned ISA and assembly. The input to the assembler is a text file containing the assembly instructions. Each line of the text file may be of one of 3 types:

- Empty line: Ignore these lines
- A label
- An instruction
- A variable definition

Each of these entities have the following grammar:

- The syntax of all the supported instructions is given above. The fields of an instruction are whitespace separated. The instruction itself might also have whitespace before it. An instruction can be one of the following:
  - The opcode must be one of the supported mnemonic.
  - A register can be one of R0, R1, … R6, and FLAGS.
  - A mem_addr in jump instructions must be a label.
  - A Imm must be a whole number <= 127 and >= 0.
  - A mem_addr in load and store must be a variable.
- A label marks a location in the code and must be followed by a colon (:). No spaces are allowed between label name and colon(:)
- A variable definition is of the following format:

  `var xyz`

  which declares a 16 bit variable called xyz. This variable name can be used in place of mem_addr fields in load and store instructions.

All variables must be defined at the very beginning of the assembly program.

The assembler should be capable of:

1) Handling all supported instructions
2) Handling labels
3) Handling variables
4) Making sure that any illegal instruction (any instruction (or instruction usage) which is not supported) results in a syntax error. In particular you must handle:
   a) Typos in instruction name or register name
   b) Use of undefined variables
   c) Use of undefined labels
   d) Illegal use of FLAGS register
   e) Illegal Immediate values (more than 7 bits)
   f) Misuse of labels as variables or vice-versa
   g) Variables not declared at the beginning
   h) Missing `hlt` instruction
   i) `hlt` not being used as the last instruction

   You need to generate distinct readable errors for all these conditions. If you find any other illegal usage, you are required to generate a “General Syntax Error”.

   The assembler must print out all these errors.
5) If the code is error free, then the corresponding binary is generated. The binary file is a text file in which each line is a 16bit binary number written using 0s and 1s in ASCII. The

assembler can write less than or equal to 128 lines.

Input/Output format:

- The assembler must read the assembly program as an input text file (stdin).
- The assembler must generate the binary (if there are no errors) as an output text file (stdout).
- The assembler must generate the error notifications along with line number on which the error was encountered (if there are errors) as an output text file (stdout). **In case of multiple errors, the assembler may print any one of the errors.**

Example of an assembly program:

```
var X
mov R1 $10
mov R2 $100
mul R3 R2 R1
st R3 X
hlt
```

The above program will be converted into the following machine code

```
0001000100001010
0001001001100100
0011000011010001
0010101100000101
1101000000000000
```

Q2: Simulator:

You need to write a simulator for the given ISA. The input to the simulator is a binary file (the format is the same as the format of the binary file generated by the assembler in **Q1**. The simulator should load the binary in the system memory at the beginning, and then start executing the code at address 0. The code is executed until `hlt` is reached. After execution of each instruction, the simulator should output one line containing an 7 bit number denoting the program counter. This should be followed by 8 space separated 16 bit binary numbers denoting the values of the registers (R0, R1, … R6 and FLAGS).

<PC (7 bits)><space><R0 (16 bits)><space>...<R6 (16 bits)><space><FLAGS (16 bits)>.

The output must be written to *stdout*. Similarly, the input must be read from *stdin*. After the program is halted, print the memory dump of the whole memory. This should be 128 lines, each having a 16 bit value.

<16 bit data>
<16 bit data>
.....
<16 bit data>

Your simulator must have the following distinct components:

1. Memory (MEM): MEM takes in an 7 bit address and returns a 16 bit value as the data. The MEM stores 256 bytes, initialized to 0s.
2. Program Counter (PC): The PC is an 7 bit register which points to the current instruction.
3. Register File (RF): The RF takes in the register name (R0, R1, … R6 or FLAGS) and returns the value stored at that register.
4. Execution Engine (EE): The EE takes the address of the instruction from the PC, uses it to get the stored instruction from MEM, and executes the instruction by updating the RF and PC.

The simulator should follow roughly the following pseudocode:

```
initialize(MEM); // Load memory from stdin
PC = 0; // Start from the first instruction
halted = false;

while(not halted)
{
	Instruction = MEM.fetchData(PC); // Get current instruction
	halted, new_PC = EE.execute(Instruction); // Update RF compute new_PC
	PC.dump(); // Print PC
	RF.dump(); // Print RF state
	PC.update(new_PC); // Update PC
}
MEM.dump() // Print the complete memory
```

Q3: Floating-Point Arithmetic:

*CSE112_Floating_point_representation: **NO** sign bit, 3 exponent bits, and 5 mantissa bits.*

- *In the registers, only the last 8 bits will be used in computations and initialization for the floating-point numbers.*

Modify the assembler and simulator to include arithmetic operations for floating-point numbers of the form(precision) given above.

Specifically, include the following functions:

10000	F_Addition	Performs reg1 = reg2 + reg3. If the computation overflows, then the overflow flag is set and reg1 is set to 0	addf reg1 reg2 reg3	A
10001	F_Subtraction	Performs reg1 = reg2 - reg3. In	subf reg1 reg2 reg3	A

		case reg3 >reg2, 0 is written to reg1 and overflow flag is set.		
10010	Move F_Immediate	Performs reg1 = $Imm where Imm is an 8-bit floating-point value.	movf reg1 $Imm	B

Note:

- For moving 1.5 into reg1. The instruction(in assembly language) should be: **movf reg1 $1.5**
- Keep in mind that in floating point multiplication $Imm is 8 bit so you need to make a new Type B syntax with 8 bit.
- The students must only apply the operations for the floating-point numbers that can be represented in the given system(8 bits), else they should report it as an error.

Q4: (Bonus) Designing New Instructions

In Q4. You need to design five instructions on your own or take help of online resources. After their design you need to upgrade your assembler to include those instructions.Similarly you need to upgrade your simulator to simulate those instructions.

Upgrade Assembler 5%
Upgrade Simulator 5%

A proper readme file with description of the instructions (opcode, semantics etc) should be included and proper explanation should be given to respectiveTA during evaluation.
N.B. You are not allowed to change the existing opcode.