Information regarding Project Components

1. 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 followed by an instruction
- 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 <= 255 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 label name consists of alphanumeric characters and underscores.
  A label followed by the instruction may looks like:
  ```
  mylabel: add R1 R2 R3
  ```
- 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.** A variable name consists of alphanumeric characters and underscores.
- Each line may be preceded by whitespace.

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 (less than 0 or more than 255)
    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
j. Wrong syntax used for instructions (For example, `add` instruction being used as a type B 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.**

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 256 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 the 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 R1 R2
st R3 X
hlt
```

The above program will be converted into the following machine code

```
0001000100001010
0001001001100100
0011000011001010
0010101100000101
1001100000000000
```

2. 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 8 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 (8 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 256 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 8 bit address and returns a 16 bit value as the data. The MEM stores 512bytes, initialized to 0s.
2. Program Counter (PC): The PC is an 8 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.

<u>The simulator should follow roughly the following pseudocode:</u>

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

white(not halted)
{
       Instruction = MEM.getData(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 memory state
```

3. Scatter Plot

In Q2, generate a scatter plot with the cycle number on the x-axis and the memory address on the y-axis. You need to plot which memory address is accessed at what time.
