

Vole machine report

A1-Task4Task5-S5-20231116-20231134-20231042



November 1, 2024

Contents

[Classes 3](#_Toc181569129)

[Memory: 3](#_Toc181569130)

[Register: 3](#_Toc181569131)

[ProgIns 3](#_Toc181569132)

[Simulator: 4](#_Toc181569133)

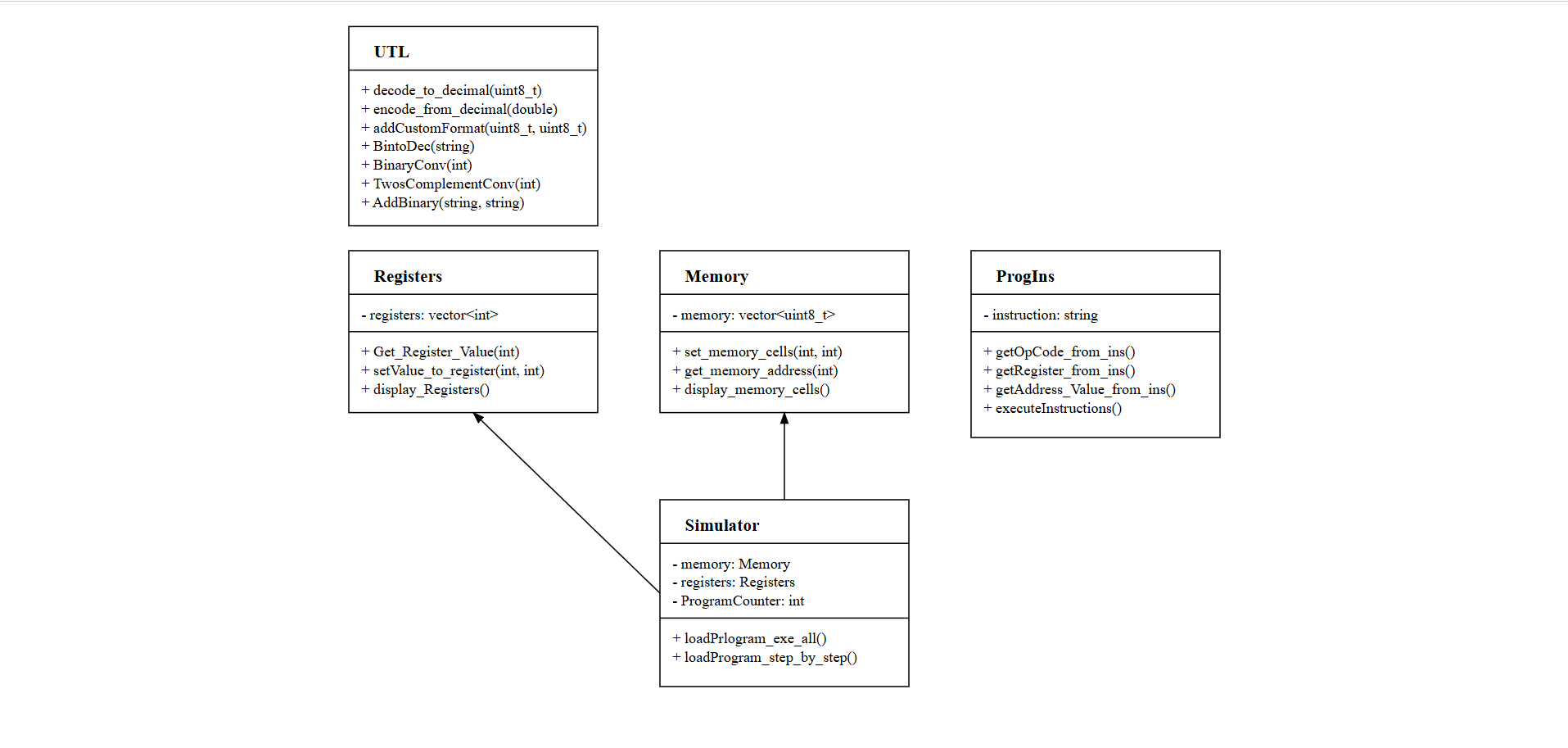
[Other function : 5](#_Toc181569134)

[Menu: 5](#_Toc181569135)

|  |  |  |
| --- | --- | --- |
| Name | ID | What he makes |
| Amr Khaled Ahmed Abd El-Hamid   20231116@stud.fci-cu.edu.eg | 20231116 | 1. Register: Class:    1. Get\_Register\_Value: function    2. setValue\_to\_register: function    3. display\_Registers: function 2. Instructions: Class    1. getOpCode\_from\_ins: function    2. getRegister\_from\_ins: function    3. getAddress\_Value\_from\_ins: function    4. executeInstructions: function 3. Simulator: Class    1. loadProgram\_exe\_all: function    2. loadProgram\_step\_by\_step: function 4. Opcode:1,2,4 5. Slow print: function 6. Menu 7. Defensive programming |
| Mohamed Ahmed Mohamed Abd El-Wahab  20231134@stud.fci-cu.edu.eg | 20231134 | 1. Memory: Class    1. set\_memory\_cells: function    2. get\_memory\_address: function    3. display\_memory\_cells: function 2. hexa\_char: function 3. hexa\_unsigned\_chars: function 4. char\_to\_hex: function 5. decode\_to\_decimal: function 6. encode\_from\_decimal: function 7. Normalize: function 8. AddFloatsBitwise: function 9. Opcode: 3,6, B,C |
| George Malak Magdy   20231042@stud.fci-cu.edu.eg | 20231042 | 1. BintoDec: function 2. BinaryConv: function 3. TwosComplementConv: function 4. AddBinary: function 5. Opcode: 5,7,8,9, A |
| Each member on team contributes on: | 1. Testing 2. Associate All program components 3. Validation | |

Vole Machine Simulator

# UML Diagram:



# GitHub repository:

GitHub link: <https://github.com/mohamedahmed2005/vole_machine.git>

# Classes

## Memory:

* **Memory definition:** memory is a vector which contains 256 cells on hexadecimal representation.
* We will make constructor to make all elements on vector equal to 0
* **Set memory cells:** On execution program, instructions must be loaded on memory cells, and we should handle overflow.
* **Get memory cells:** after execution program, we will return value on cells to be used on program later.
* **Display memory cells:** program will output 256 cells of memory and values which is inside each memory cell.
* We will make destructor to avoid memory leak.

## Register:

* **Definition:** The Registers class manages a set of registers used for temporary storage of values during program execution.
* **Constructor:** Initializes all register values to 0.
* **Get\_Register\_Value:** Retrieves the value stored in a specified register.
* **setValue\_to\_register:** Updates the value of a specified register, ensuring the index is valid.
* **display\_Registers:** Outputs the current values of all registers for debugging and tracking purposes.
* **Destructor:** Cleans up any resources, although simple integer storage typically requires minimal management.

## ProgIns

* **Definition:** The ProgIns class encapsulates program instructions, providing methods to extract components from instructions and execute them.
* **Constructor:** Initializes the instruction string.
* **getOpCode\_from\_ins:** Extracts the opcode from the instruction for processing.
* **getRegister\_from\_ins:** Retrieves the register index specified in the instruction.
* **getAddress\_Value\_from\_ins:** Extracts the address or value from the instruction for memory operations.
* **executeInstructions:** Implements logic to execute the instruction based on the opcode and interacts with the memory and registers as necessary.

## Simulator:

* **Definition:** The Simulator class coordinates the operation of the virtual machine, managing both memory and registers while executing instructions.
* I use an association from Registers Class and Memory class.
* **Constructor:** Initializes the simulator, including the memory and registers.
* **loadProgram\_exe\_all:** Loads all instructions into memory for execution and give all data afte r execute it.
* **loadProgram\_step\_by\_step:** Allows instructions to be loaded and executed step by step, with ask for continue or not.

## UTL :

* Slow Print Function: This function takes a millisecond delay and prints every character after this delay.
* BintoDec: This function takes a binary number represented as a string (e.g., "101") and returns its decimal equivalent as an integer. For example, inputting "101" will return the decimal value 5.
* hexa\_char: the Function of this function is to convert a single hex digit to an unsigned character.
* hexa\_unsigned\_chars: the Function of this function is to convert a hex string to an unsigned char.
* char\_to\_hex: the function of this function is to convert an unsigned char to a hex string.
* decode\_to\_decimal: this function Decode from custom format to decimal.
* encode\_from\_decimal: this function Encode from decimal to custom format.
* Normalize\_IEEE: This function converts floating numbers to IEEE “8-bits” Representation.
* Add\_two\_floating\_numbers: this Function to add two 8-bit floating-point numbers in custom format.
* BinaryConv: This function accepts an integer and returns its binary representation as a string. For instance, if you input the integer 5, it will return "101". (Note: this function is only called within **TwosComplementConv**).
* TwosComplementConv: This function first checks if a given number is positive or negative. If the number is positive, it converts it directly to binary using **BinaryConv**. If the number is negative, it takes the positive version of the number, converts it to binary, inverts all binary digits (changing 1s to 0s and 0s to 1s), and finally adds 1, following the rules of two’s complement representation.
* AddBinary: This function takes two binary numbers represented as strings, adds them, and returns the result as a binary string, ignoring any overflow.

# Menu:

* **Options**:
  + **1 - Load a new program**: This option allows the user to load and execute a program.
  + **2 - Exit**: This option terminates the program and displays a farewell message.
* **User Input**: The user is prompted to enter a choice. Input validation ensures only valid choices are accepted.

**Execution Menu** (appears after selecting to load a new program):

* **Options**:
  + **1 - Execute line by line**: This option allows the user to step through the program one instruction at a time.
  + **2 - Execute the entire program at once**: This option loads and executes all instructions in a single step.
* **User Input**: Similar to the main menu, the user is prompted to enter a choice, with error handling for invalid inputs.

# The Machine’s Language

|  |  |  |  |
| --- | --- | --- | --- |
| **c** | **Format** | **Operand** | **Description** |
| 1 | Load | RXY | LOAD the register R with the bit pattern found in the memory cell whose address is XY. Example. |
| 2 | Load | RXY | LOAD the register R with the bit pattern XY. Example: 20A3 would cause the value A3 to be placed in register 0. |
| 3 | Store | RXY | STORE the bit pattern in register R into the memory cell at address XY. Example. |
| 4 | Move | 0RS | MOVE the bit pattern found in register R to register S. Example. |
| 5 | Add | RST | ADD the bit patterns in registers S and T as though they were two’s complement representations and leave the result in register R. we have used the functions:   * TwosComplementConv * BinaryConv * AddBinary * BintoDec |
| 6 | FAdd | RST | ADD the bit patterns in registers S and T as though they represented values in floating-point notation and leave the floating-point result in register R.  we have used the functions:   * **hexa\_char** * **hexa\_unsigned\_chars** * **char\_to\_hex** * **decode\_to\_decimal** * **encode\_from\_decimal** * **Normalize\_IEEE** * **Add\_two\_floating\_numbers** |
| 7 | OR | RST | OR the bit patterns in registers S and T and place the result in register R. Example. |
| 8 | AND | RST | AND the bit patterns in registers S and T and place the result in register R. Example. |
| 9 | EXCLUSIVE OR | RST | EXCLUSIVE OR the bit patterns in registers S and T and place the result in register R. Example. |
| A | Rotate | R0X | ROTATE the bit pattern in register R one bit to the right X times. Each time place the bit that started at the low-order end at the high-order end. Example. |
| B | Jump | RXY | JUMP to the instruction located in the memory cell at address XY if the bit pattern in register R is equal to the bit pattern in register number 0. Otherwise, continue with the normal sequence of execution. (The jump is implemented by copying XY into the program counter during the execute phase. |
| C | Halt | 000 | HALT execution. Example: C000 would cause program execution to stop |