Faculty of Computing and Informatics (FCI) Multimedia University
Cyberjaya

TCP1101 Programming Fundamentals

Trimester 2310

Assignment

Assembly Language Interpreter
(Virtual Machine using C++)

NAME: Venggadanaathan A/L K.Salvam

STUDENT ID: 1231303562

STUDENT ACCOUNT: 1231303562@student.mmu.edu.my

TUTORIAL SECTION: TT3L

LECTURE SECTION: TC1L

LECTURER NAME: Ts. Goh Chien Le (clgoh@mmu.edu.my)

GROUP NUMBER: G40

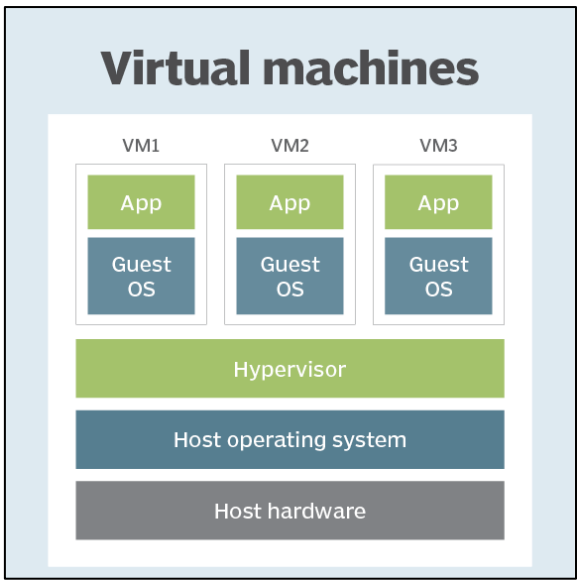
CONTENTS

INTRODUCTION

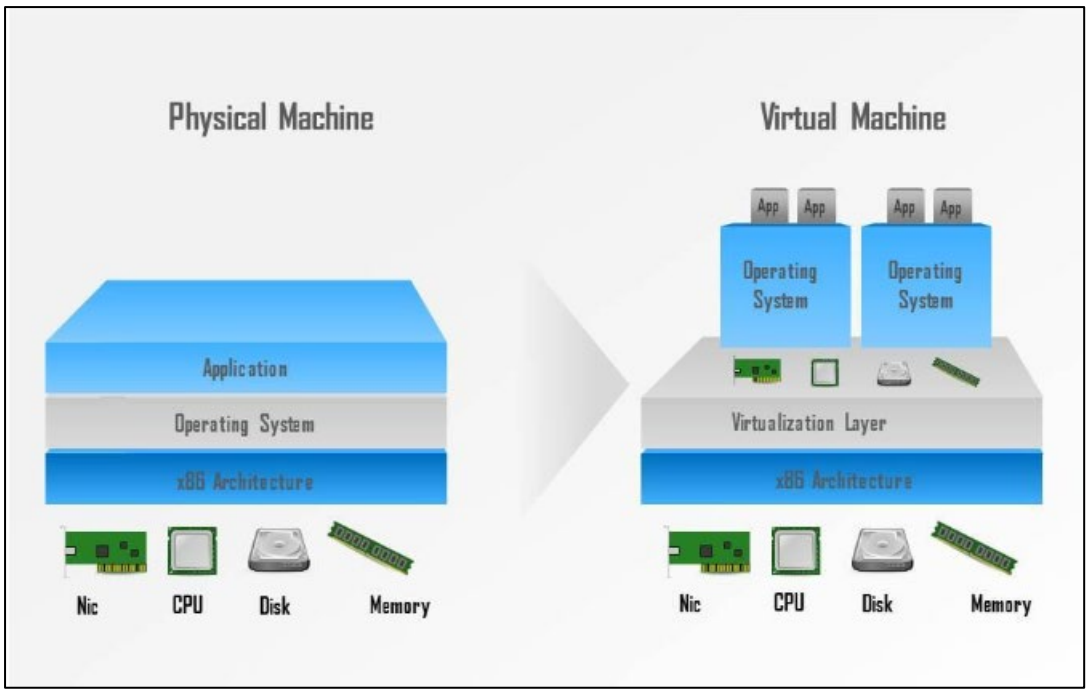
The assembly language interpreter project attempts to develop a stable and efficient virtual machine, as well as a versatile interpreter tailored to a certain architecture. This endeavour aims to bridge the gap between low-level machine instructions and high-level programming languages. By creating an assembly language interpreter, we hope to create a platform for programmers to interface with the underlying hardware in a more human-readable and symbolic way**.**

WHAT IS A VIRTUAL MACHINE?

In general, A virtual machine or abbreviated as (VM) is a simple software-based representation of a real computer. It serves as an isolated environment in which software applications can execute, while abstracting the underlying hardware. The virtual machine runs a set of instructions known as **machine code**, which is usually distinct from the host computer's native instruction sets or from third-party programs. VM can provide information about its CPU, disk, memory, network, and storage.



Above is the diagram, which shows how Virtual Machines works in a computers and devices. A single computer can host multiple VMs running different operating systems (OS) and applications without affecting or interfering with each other. Although the VM is still dependent on the host's physical resources, those resources are virtualized and distributed across the VMs and can be reassigned, as necessary. All these multiple VMs, run on the same hypervisor, host OS and hardware, which means it runs on the same device. A hypervisor is a software that can be used to run multiple VMs on a single device.

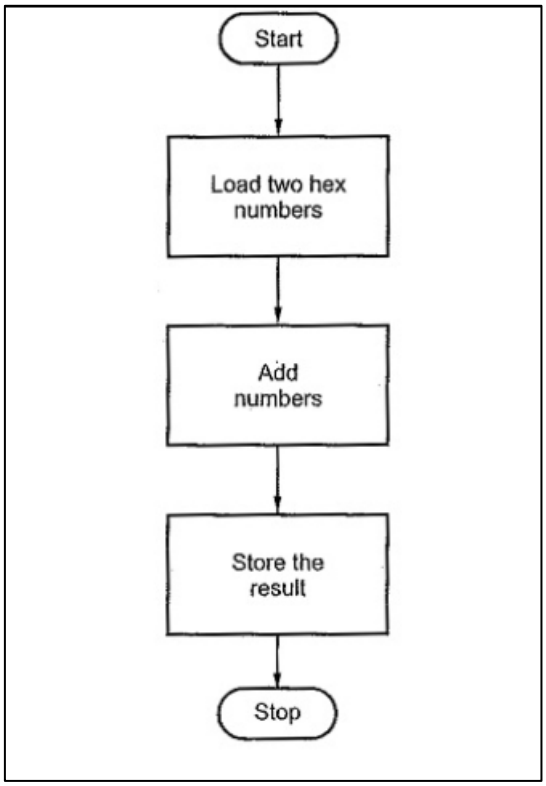


There is a big difference between physical machines and VMs. Virtual machines are virtualization of physical machines where it operates and how it operates the applications through operating systems.

In this assignment, the virtual machine is the output of the program, where the assembly code acts as instructions to toggle it and the interpreter is the C++ program where it will translate the instructions as machine language and run the VM.

WHAT IS ASSEMBLY LANGUAGE?

This low-level programming systems represents a computer's machine code symbolically. Unlike high-level languages, assembly language is related to the architecture and instruction set of the target processor. It uses mnemonic codes to describe machine-level operations, making it more accessible than raw machine code, but it remains intimately connected. For an example, MOV, ADD XOR, OR, AND and LOAD are some of them. It intended to communicate directly with a computer’s hardware.



Above is an example flowchart diagram how the assembly language looks like for the process to add two numbers and store them in microprocessors.

Task 1 instructions :

```
MVI  A,  20H   ; Load 20H as a first
               ; number in register A
MVI  B,  40H   ; Load 40H as a second number
               ; in register B
```

Task 2 Instruction :

```
ADD  B         ; Add two numbers and save
               ; result in register A
```

Task 3 Instruction :

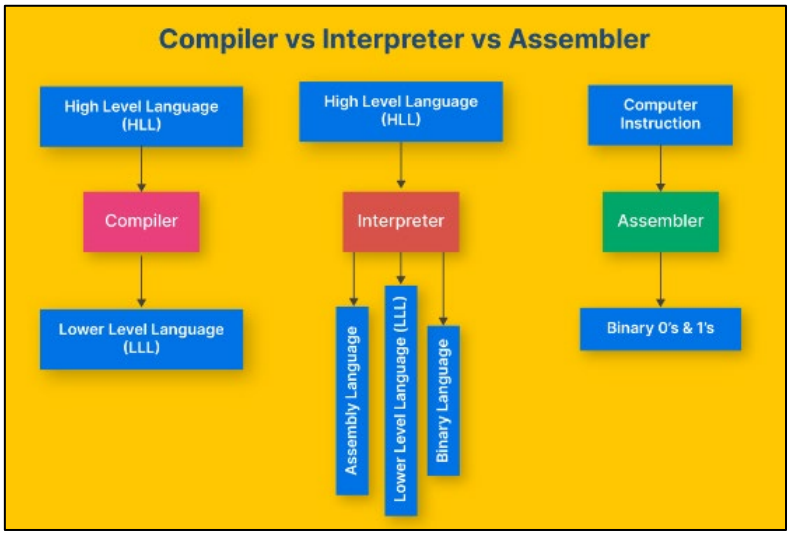
```
STA  2200H     ; Store the result in memory
               ; location 2000H
HLT            ; Stop the program execution.
```

Mnemonics	Hex code		
MVI A, 20H	3EH	←	Opcode
	20H	←	Operand
MVI B, 40H	06H	←	Opcode
	40H	←	Operand
ADD B	80H	←	Opcode
STA 2200H	32H	←	Opcode
	00H	←	Operand (lower byte of address)
	22H	←	Operand (higher byte of address)
HLT	76H	←	Opcode

The similar operations will be used in this assignment, for an example for this process, the assembly language instructions will be LOAD, ADD and STORE.

WHAT IS THE ASSEMBLY LANGUAGE INTERPRETER?

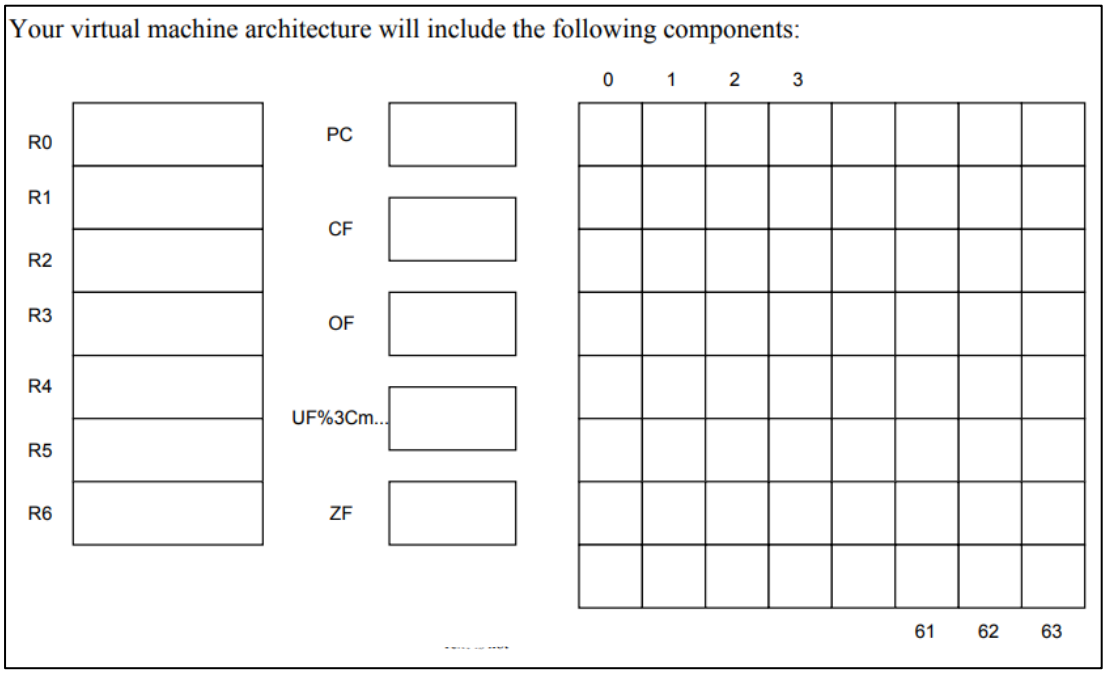
An assembly language interpreter is a software component that understands and runs assembly language programmes. It acts as a bridge between the human-readable assembly code and the machine code recognised by the computer's hardware. The interpreter converts each assembly language instruction into machine code and runs it on the virtual machine. In this assignment, C++ programs and codes will function as the interpreter/runner.



These are the terms to differentiate the types of translators. In this assignment the main part is about the interpreter but these three will be used to for the completion of the instruction that will be input by users.

VIRTUAL MACHINE ARCHITECTURE

The virtual machine includes seven data general registers and a program counter where each of them is one byte wide (char), 4 flag registers (Overflow, Underflow, Carry, and Zero), and a main memory of 64 bytes.



- **Registers**

7 general-purpose data registers: R0, R1, R2, R3, R4, R5, R6. Each register is represented as a one byte (8 bits) in size. The contents of these registers are updated after executing each assembly instruction by the runner. These registers can contain unsigned integer bytes (values between 0 to 255) in every single register, but most commonly only single and double digits will used throughout the processes.

- **Program Counter**

The program counter (will be denoted as PC in the runner) starts always with the value 0, then it is incremented whenever the runner (interpreter) executes an assembly instruction. PC keeps track of the memory address of the next instruction to be executed and it controls the flow of execution by pointing to the current instruction.

- **Flags**

The flags register contains status flags that reflect the outcome of arithmetic and logical operations.

☐ Overflow Flag (OF): A single bit flag (or a byte) indicating arithmetic overflow. It is set when an arithmetic operation results in the range from 0 to 255.

☐Underflow Flag (UF): A single bit flag (or byte) indicating arithmetic underflow. Set when an arithmetic operation results in a value smaller than zero.

☐Carry Flag (CF): A single bit flag (or a byte) indicating carry in arithmetic operations. Set when an arithmetic operation results in a carry. An addition can generate a value larger than 8 bits, therefore the carry flag is set, (Example: 5+6=11, then CF will be triggered), will be also triggered if the result is more than 255.

☐Zero Flag (ZF): A single bit flag indicating the result of an operation is zero. Set when the result of an operation is zero.

These flags will be set in Boolean and will be shown as one when triggered, (0 if no changes) and only be triggered for increment/ decrement, arithmetic, shift and rotate operations.

- **Memory**

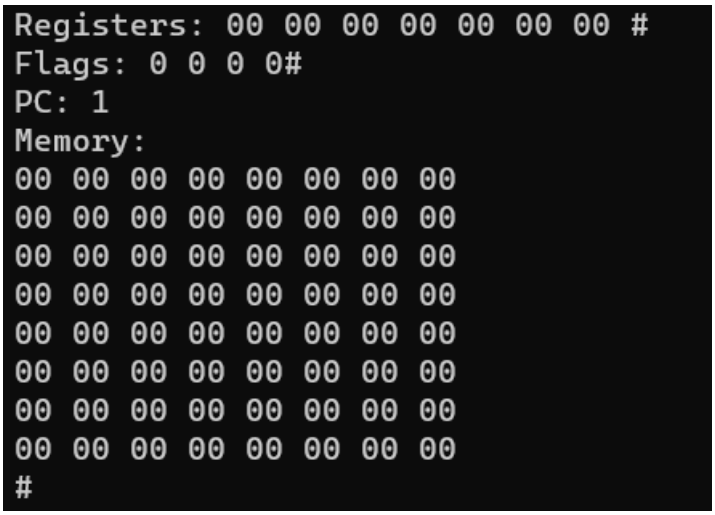
The memory is represented as a one-dimensional array of 64 signed bytes (char). Memory can be accessed only by a load or store operations. Addresses are sequential, starting from 0 to 63, and can be accessed individually. Addresses are used to locate and access data in memory.

- **Assembly Language Runner**

The interpreter reads an assembly program from a ".asm" file (text file). It executes the instructions sequentially on the virtual machine updating the PC (program counter) after each instruction. At the end of execution, it will produce a dump of all registers and memory to the screen in addition to the output window (in command prompt or other respective compilers or note readers).

It Interacts with registers, program counter, flags, and memory during program execution. It also executes instructions by manipulating data in registers and memory.

In the assignment, this how the general overview of the virtual machine looks like:



THE ASSEMBLY LANGUAGE INTREPRETER

This the full-fledged explanation of the C++ program and its contents:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <iomanip>
#include <bitset>
```

These are the header files used in the program:

#include <iostream> : provides the basic input and output streams functionality

#include <fstream> : used for reading from and writing to files.

#include <sstream> : used for string stream handling.

#include <vector> : used for vector functions.

#include <iomanip> : used for formatting output, particularly for setting the width and filling characters in the output.

#include <bitset>: used for bit manipulation, and in this program, it's used to display binary representation of values.

Arrays

`int R[7];` : An array representing the general-purpose registers R0 to R6.

Vectors

`vector<int> memory;` : Vector of memory in integers.

Class

```
class VirtualMachine {
private:
    int R[7];
    unsigned int PC;
    bool OF, UF, CF, ZF;
    vector<int> memory;
    ofstream outputFile;
```

VirtualMachine: A class representing a simple virtual machine with registers, flags, memory, and methods to execute a program.

Files

`ofstream outputFile;` Used to log the output state of the virtual machine.

`ifstream inputFile(inputFilePath);` Used to read the program from the input file specified in executeProgram function.

GENERAL Overview of the Assembly Language Interpreter

i) Pseudocode

Create a VirtualMachine object.

Display "Enter the path of the program file:".
Input filePath.

Call executeProgram method of the VirtualMachine object with filePath.

VirtualMachine Class:
  Define a constructor initializing the virtual machine:
    Initialize registers, flags, memory, and open an output file for logging.

  Define stringToInt method:
    Convert the input string to an integer using stringstream.

  Define displayState method:
    Display the current state of the virtual machine, including registers, flags, program counter, and memory.

  Define executeProgram method:
    Open the input file specified by the filePath.
    Loop through each line in the file:
      Extract instruction and operands from the line.
      Check if the instruction is valid.
      Perform the corresponding operation based on the instruction.
      Update program counter and display the state.
    Close the input file.

  Define private method isValidInstruction:
    Check if the given instruction is one of the valid instructions.

  Define private method getOperandValue:
    Determine the value of an operand based on its type (register, immediate value, or memory).

  Define private method updateFlags:
    Update flags (OF, UF, CF, ZF) based on the result of an operation.

  Define private method shift (with rotate functions):
    Perform shift or rotate operation based on the provided instruction.

  Define private method move:
    Move data between registers, memory, or immediate values.

  Define private method arithmeticOperation:
    Perform arithmetic operations (ADD, SUB, MUL, DIV).

  Define private method incrementDecrement:
    Increment or decrement a register.

  Define private method input:
    Read input and store it in a register.

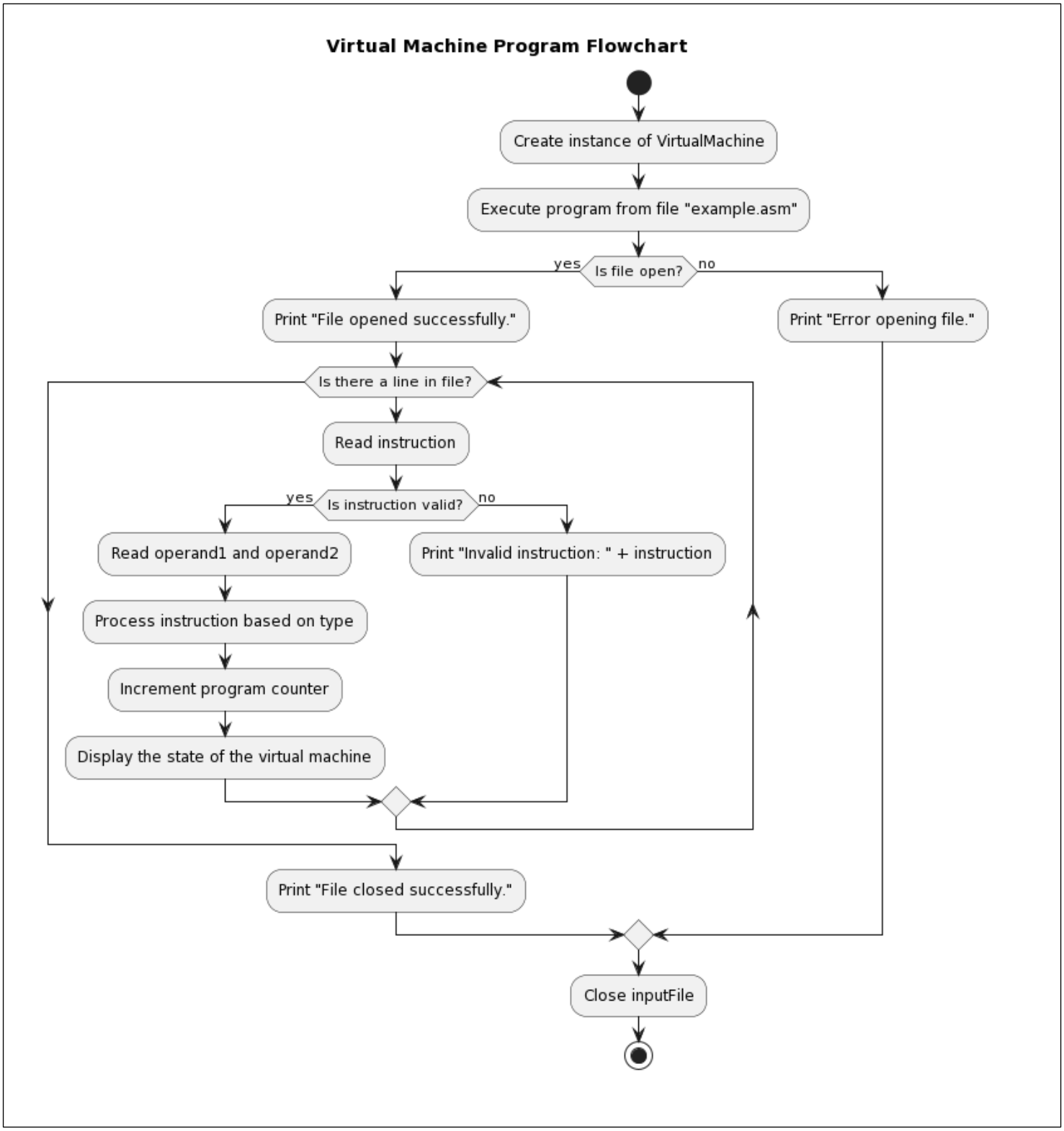
  Define private method output:
    Output the value of a register.

  Define private method loadandStore:
    Load or store data between registers and memory.

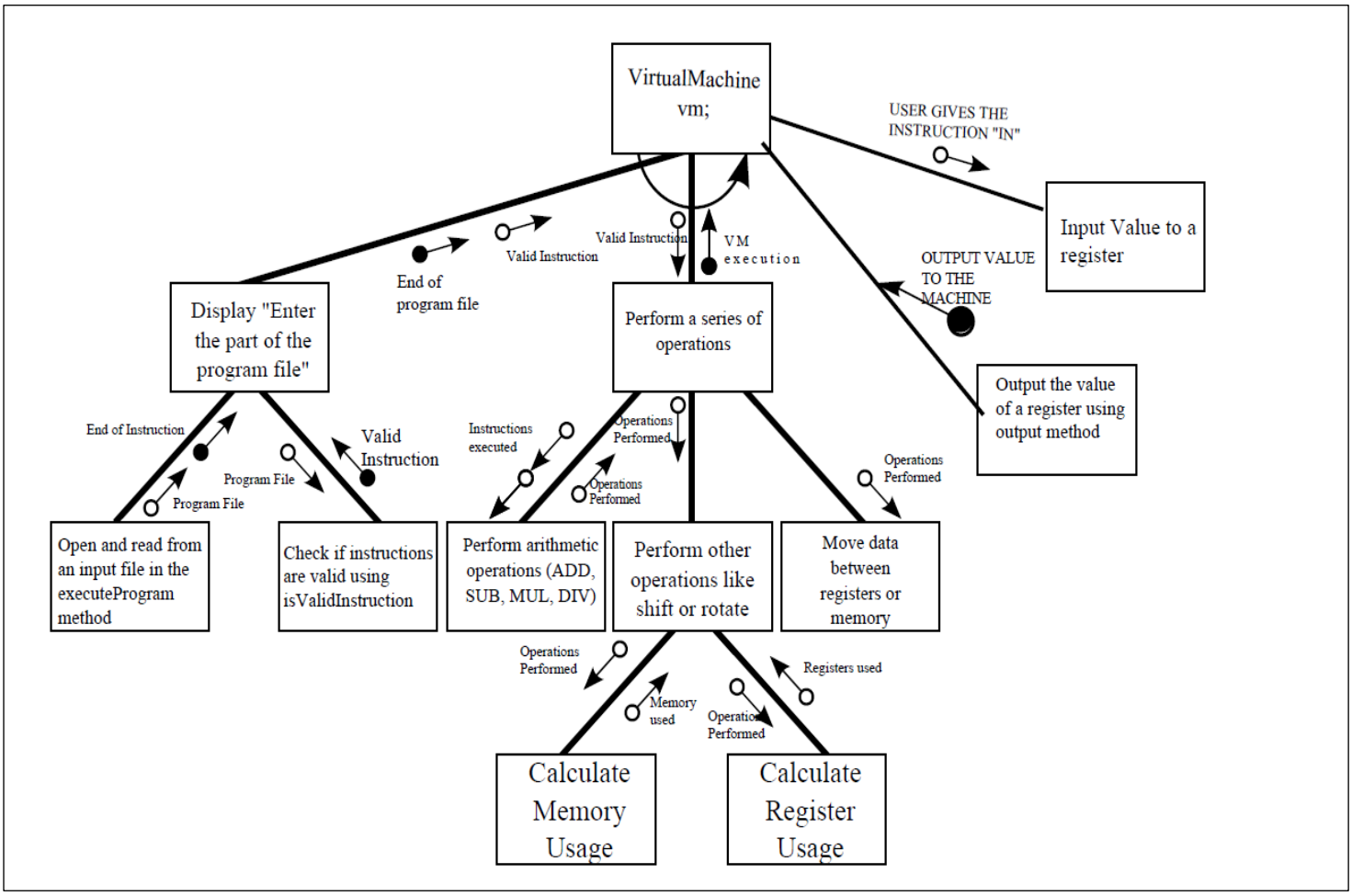
Define main function:
  Create a VirtualMachine object.
  Call executeProgram method of the VirtualMachine object with the path of the program file.

End main function.

ii) Flowchart



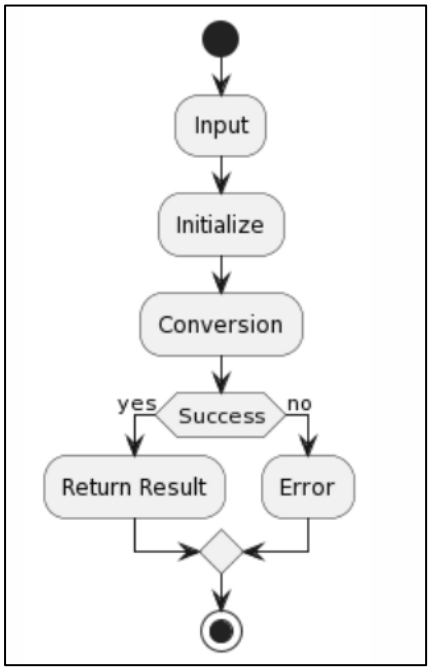
iii) Structured Chart



Functions

1. stringToInt(const string& str)

```
int stringToInt(const string& str)
{
    stringstream ss(str);
    int result;
    ss >> result;
    return result;
}
```

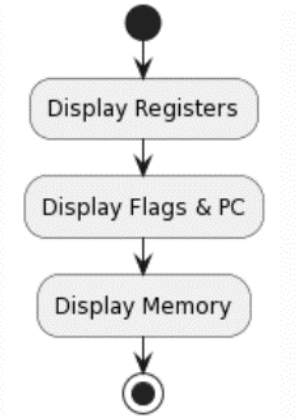


Pseudocode:

Function stringToInt takes a constant string reference str as input and returns an integer:
Create a stringstream ss and initialize it with str.
Declare an integer variable result.
Extract data from ss into result
Return result.
End Function.

2. displayState function

```
void displayState() const // Display the current state of the virtual machine
{
    cout << "Registers: "; // Display registers
    for (int i = 0; i < 7; ++i) //Loop through each register
    {   //Display the value of the register in decimal format, padded with zeros
        cout << setw(2) << setfill('0') << dec << R[i] << " ";
    }
    //Display the flags and program counter
    cout <<"#\nFlags: " << OF << " " << UF << " " << CF << " " << ZF << "#\nPC: " << dec << PC << "\nMemory:\n";
    // Display memory
    for (int i = 0; i < memory.size(); ++i) //Loop through each memory location
    {
        cout << setw(2) << setfill('0') << dec << memory[i] << " ";
        if ((i + 1) % 8 == 0) //If the memory location is a multiple of 8
        {
            cout << endl; //Start a line
        }
    }
    cout << "#\n"; // Separator
}
```



Pseudocode:

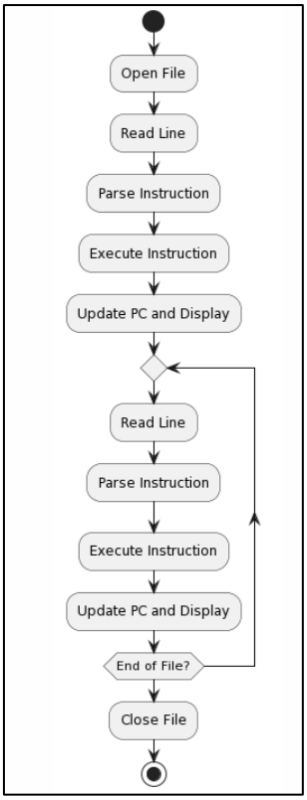
Function displayState:
    Print "Registers: ", register values, "Flags: ", flag values, "PC: ", PC value
    Print "Memory: ", memory values in lines of 8
End Function.

3. executeProgram(const string& inputFilePath) function.

```
void executeProgram(const string& inputFilePath) // Execute the program loaded from the input file
{
    ifstream inputFile(inputFilePath);
    if (inputFile)
        {
        cout << "File opened successfully.\n";
        string line;

        while (getline(inputFile, line))
        {
            istringstream ins(line);
            string instruction;
            ins >> instruction;

            if (isValidInstruction(instruction))
            {
                string operand1, operand2;
                ins >> operand1 >> operand2;

                if (isdigit(operand2[0]) && instruction != "STORE" && instruction != "LOAD")
                    {
                    operand2 = "#" + operand2;
                    }

                if (instruction == "MOV") // Process the instructions
                    {
                    move(operand1, operand2);
                    }
                else if (instruction == "ADD" || instruction == "SUB" || instruction == "MUL" || instruction == "DIV")
                    {
                    arithmeticOperation(instruction, operand1, operand2);
                    }
                else if (instruction == "INC" || instruction == "DEC")
                    {
                    incrementDecrement(instruction, operand1);
                    }
                else if (instruction == "IN")
                    {
                    input(operand1);
                    }
                else if (instruction == "OUT")
```

Pseudocode:

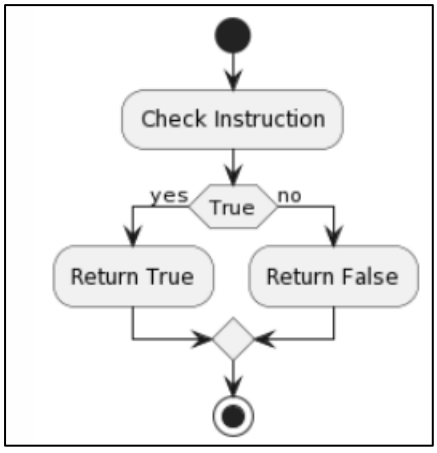
Function executeProgram takes a file path as input:
    Open the file.
    If the file is open:
        Read each line of the file.
        For each line, parse the instruction and operands.
        If the instruction is valid, execute the corresponding operation.
        Increment the program counter and display the state.
    Close the file.
End Function



In the private class of the VirtualMachine class:

4. isValidInstruction function

```
bool isValidInstruction(const string& instruction) const // Check if the instruction is valid
{
    return (instruction == "MOV" || instruction == "ADD" || instruction == "SUB" || instruction == "MUL" ||
            instruction == "DIV" || instruction == "INC" || instruction == "DEC" || instruction == "IN" ||
            instruction == "OUT" || instruction == "LOAD" || instruction == "STORE" || instruction == "SHL" ||
            instruction == "SHR" || instruction == "ROL" || instruction == "ROR");
}
```

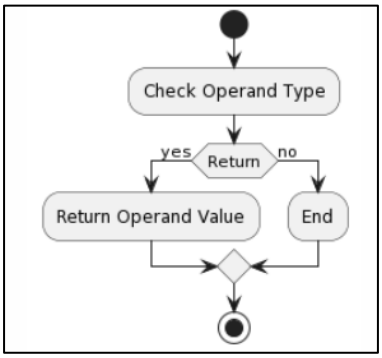


Pseudocode:

Function isValidInstruction takes an instruction as input:
    Return true if the instruction matches any of the valid instructions.
End Function

5. int getOperandvalue function

```
int getOperandValue(const string& operand) const // Get the value of an operand based on its type
{
    // If the operand is a register, return the value in the register
    if (operand[0] == 'R') return R[operand[1] - '0'];
    if (operand[0] == '#') // If the operand is an immediate value
        {
        int value = 0;
        // For each character in the operand, starting from the second character
        for (size_t i = 1; i < operand.size(); ++i)
        {
            // Convert the character to a digit and add it to the value
            value = value * 10 + (operand[i] - '0');
        }
        return value; // Return the value
       }
      // If the operand is a memory location
    if (operand[0] == '[' && operand.back() == ']')
    {
        int memoryIndex = 0; // Initialize memory index
        // For each character in the operand, excluding the first and last characters
        for (size_t i = 1; i < operand.size() - 1; ++i)
        {
            // Convert the character to a digit and add it to the memory index
            memoryIndex = memoryIndex * 10 + (operand[i] - '0');
        }
        return memory[memoryIndex]; // Return the value at the memory location
    }
    return 0; // If the operand is not recognized, return 0
}
```



Pseudo code:

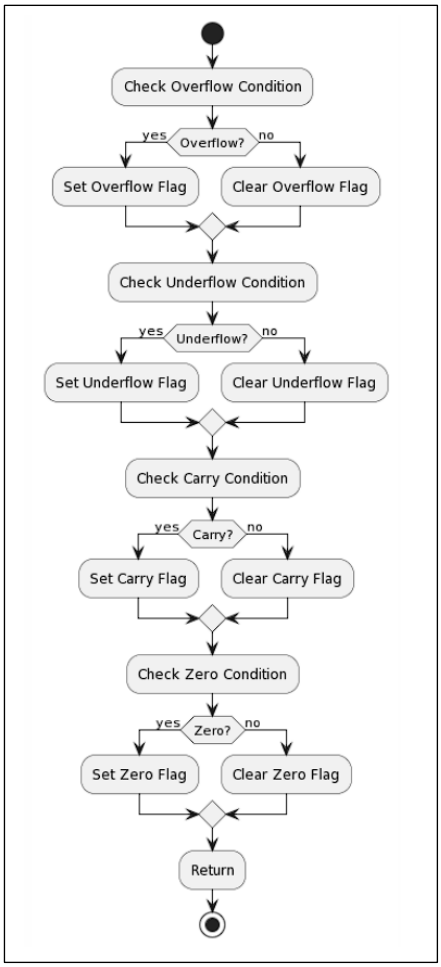
Function getOperandValue takes an operand as input:
    If the operand is a register, return the register value.
    If the operand is an immediate value, return the value.
    If the operand is a memory location, return the memory value.
    If the operand is not recognized, return 0
End Function

6. void updateflags function

```
void updateFlags(int result)
{
    OF = (result > 0 && result < 255);
    UF = (result < 0);
    CF = (result > 255);
    ZF = (result == 0);
}
```

Pseudo code:

Function updateFlags takes a result as input:
    Set flags based on the result value.
End Function



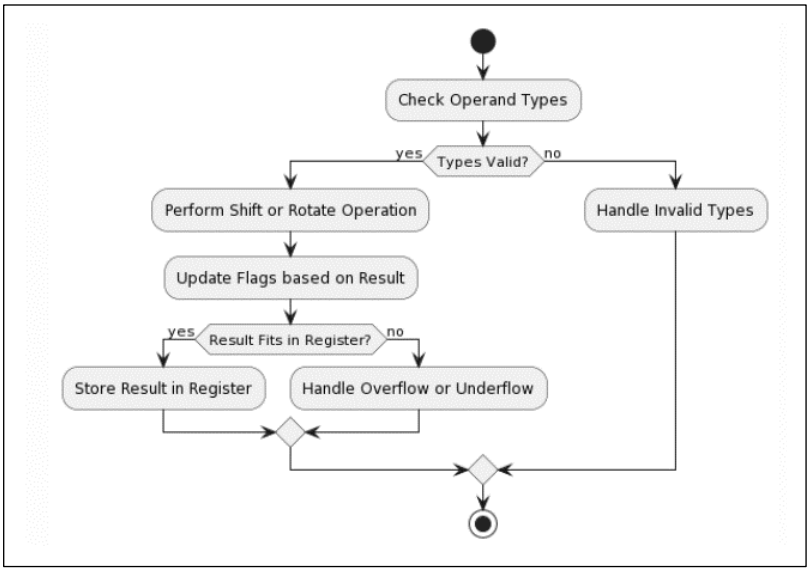
7. void shift function

```
void shift(const string& instruction, const string& operand1, const string& operand2)
{
 int shiftAmount = getOperandValue(operand2); //Get the shift amount from the second operand
 unsigned char value = getOperandValue(operand1); //Get the value to be shifted from the 1st operand
 unsigned char result = 0; //Initialize result
 // Display binary representation of operand before operation
 cout << "Binary of operand before shift/rotate: " << bitset<8>(value) << endl;
 if (instruction == "SHL")// If instruction is Shift Left
     {// Shift the bits of value to the left by shiftAmount and mask with 0xFF to keep within 8 bits
     result = (value << shiftAmount) & 0xFF;
     }
 else if (instruction == "SHR")
     { // If instruction is Shift Right
     result = value >> shiftAmount;// Shift the bits of value to the right by shiftAmount
     }
 else if (instruction == "ROL")// If instruction is Rotate Left
     {
     result = (value << shiftAmount) | (value >> (8 - shiftAmount));
     // Rotate the bits of value to the left by shiftAmount
     result |= (value << (shiftAmount - 1)); // Set the carry bit in the rotated value
     }
 else if (instruction == "ROR")
     {// Rotate the bits of value to the right by shiftAmount and mask with 0xFF to keep within 8 bits
     result = ((value >> shiftAmount) | (value << (8 - shiftAmount))) & 0xFF;
     }
 else // If instruction is not recognized
     {
     cout << "Invalid shift/rotate instruction: " << instruction << endl; // Display error message
     return; // Exit function
     }
 cout << "Binary of result after shift/rotate: " << bitset<8>(result) << endl;
 // Display binary representation of result after operation
 cout << "Decimal of result after shift/rotate: " << static_cast<int>(result) << endl;  // Display the decimal value
 // Display decimal value of result after operation
 updateFlags(result); // Update flags based on the result of the operation
 if (operand1[0] == 'R' && isdigit(operand1[1]))// If the first operand is a register
     {
     R[operand1[1] - '0'] = result; // Store the result in the specified register
     }
}
```

Pseudo code:

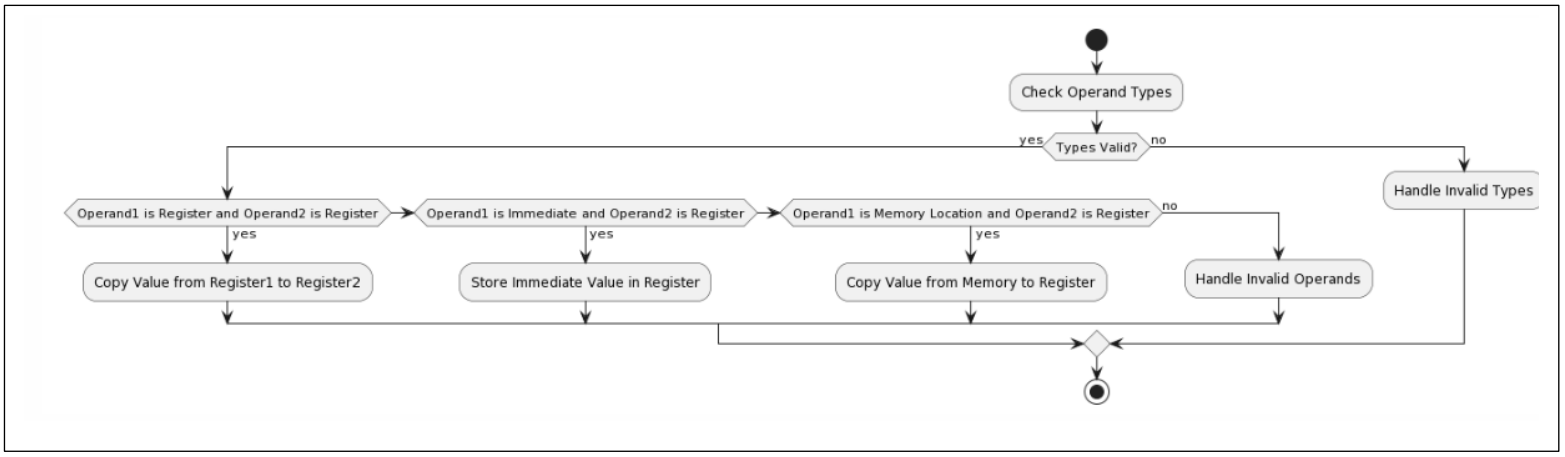
Procedure SHIFT(instruction, operand1, operand2)

shiftAmount <- get value from operand2
value <- get value from operand1
result <- 0
If instruction is "SHL" Then
result <- Shift value left by shiftAmount (within 8 bits)
Else If instruction is "SHR" Then
result <- Shift value right by shiftAmount
Else If instruction is "ROL" Then
result <- Rotate value left by shiftAmount (set carry bit)
Else If instruction is "ROR" Then
result <- Rotate value right by shiftAmount (within 8 bits)
Else
Display error message and exit function
Display binary and decimal of result
Update flags based on result
If operand1 is a register Then Store result in register
End Procedure



8. void move function

```
void move(const string& operand1, const string& operand2)
{
  if (operand1[0] == 'R' && operand2[0] == 'R')
      {
      int sourceValue = R[operand1[1] - '0'];// Get the value from the source register
      R[operand2[1] - '0'] = sourceValue;// Copy the value to the destination register
      cout << "Copied value " << sourceValue << " from register R" << (operand1[1] - '0')
      << " to register R" << (operand2[1] - '0') << endl;
      }// Display the operation

  else if (isdigit(operand1[0]) && operand2[0] == 'R') // If the first operand is an immediate v
      {
      int sourceValue = stringToInt(operand1);// Convert the string to an integer
      R[operand2[1] - '0'] = sourceValue;// Store the value in the specified register
      cout << "Stored value " << sourceValue << " to register R" << (operand2[1] - '0') << endl;
      } // Display the operation

  // If the first operand is a memory location and the second operand is a register
  else if (operand1[0] == '[' && operand1.back() == ']' && operand2[0] == 'R')
      {
      int memoryAddress = R[operand1[2] - '0'];// Get the memory address from the register speci
      int sourceValue = R[operand2[1] - '0'];// Get the value from the register specified in the
      memory[memoryAddress] = sourceValue;// Store the value in the specified memory location
      cout << "Copied value " << sourceValue << " from register R" << (operand2[1] - '0')
      << " to memory location " << &memory[memoryAddress] << endl;
      }
  else
      {// If the operands are not recognized, display error message
      cout << "Invalid operands: " << operand1 << ", " << operand2 << endl;
      }
}
```

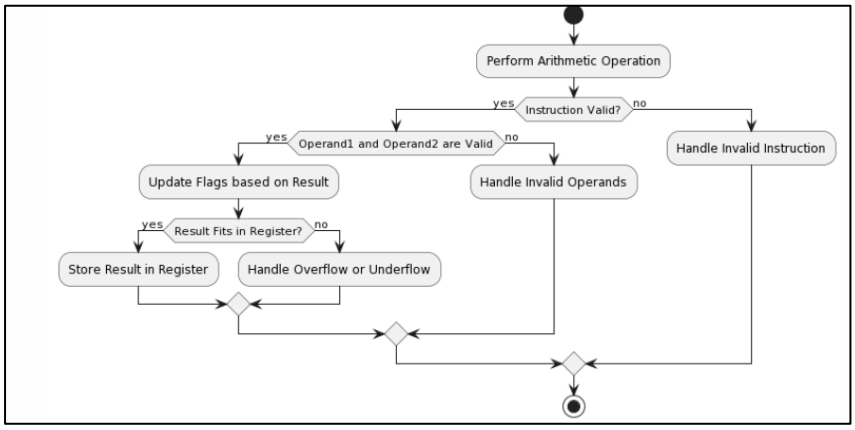


Pseudo code:

Procedure MOVE(operand1, operand2)
  If operand1 and operand2 are registers Then
    Copy value from source register to destination register
  Else If operand1 is an immediate value and operand2 is a register Then
    Store the value in the specified register
  Else If operand1 is a memory location and operand2 is a register Then
    Store the value in the specified memory location
  Else
    Display error message "Invalid operands"
  End If
End Procedure

9. Void arithmeticOperation function

```
void arithmeticOperation(const string& instruction, const string& operand1, const string& operand2)
{
  int result;
  if (instruction == "ADD")
      {
      result = getOperandValue(operand1) + getOperandValue(operand2);
      }
  else if (instruction == "SUB")
      {
      result = getOperandValue(operand1) - getOperandValue(operand2);
      }
  else if (instruction == "MUL")
      {
      result = getOperandValue(operand1) * getOperandValue(operand2);
      }
  else if (instruction == "DIV")
      {
      int divisor = getOperandValue(operand2);
      if (divisor != 0) //If dividing not by an integer
          {
          result = getOperandValue(operand1) / divisor; //store the result
          }
    else {
          cout << "Error: Division by zero.\n"; //If yes, then print this error message
          return;
         }
     }
    }
```



Pseudocode:

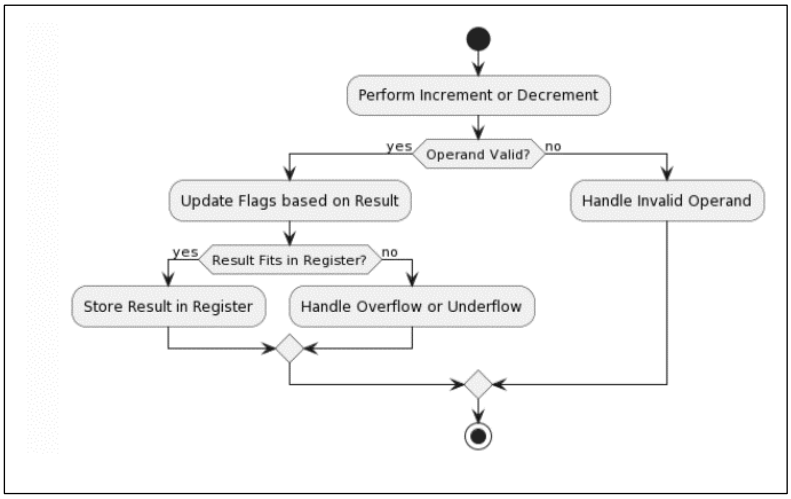
```
Procedure arithmeticoperation (instruction, operand1, operand2)
    If instruction is "ADD" Then
        result <- operand1 + operand2
    Else If instruction is "SUB" Then
        result <- operand1 - operand2
    Else If instruction is "MUL" Then
        result <- operand1 * operand2
    Else If instruction is "DIV" Then
        If operand2 is not zero Then
            result <- operand1 / operand2
        Else
            Display error message "Error: Division by zero."
            Exit function
        End Procedure
```

10. Void incrementDecrement function

Pseudocode:

Procedure INCREMENTDECREMENT(instruction, operand)
  If operand is a register Then
    If instruction is "INC" Then
      Increment the register value
    Else If instruction is "DEC" Then
      Decrement the register value
    Else
      Display error message "Invalid increment/decrement instruction"
      Exit function
    End If
    Update flags based on result
    Store result in register
    Display updated register value
  Else
    Display error message "Invalid operand for increment/decrement instruction"
  End If
End Procedure

```
void incrementDecrement(const string& instruction, const string& operand)
{
  int result;
  if (operand[0] == 'R')
      {
      if (instruction == "INC")
      {
          result = R[operand[1] - '0'] + 1;
      }
      else if (instruction == "DEC")
      {
          result = R[operand[1] - '0'] - 1;
      }
      else
      {
      cout << "Invalid increment/decrement instruction: " << instruction << endl;
      return;
      }
      updateFlags(result);
      R[operand[1] - '0'] = result;
      cout << "Register R" << (operand[1] - '0') << " incremented/decremented to " << result << endl;
      }
  else {
      cout << "Invalid operand for increment/decrement instruction: " << operand << endl;
       }
}
```

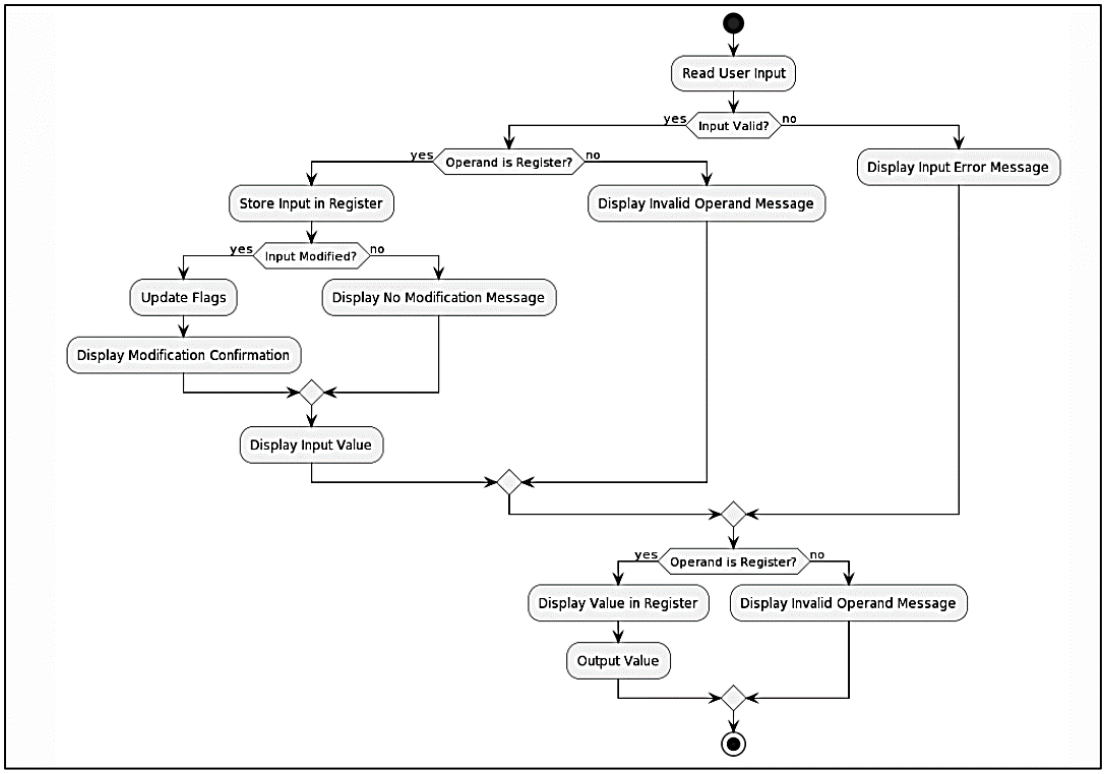


11. void input and output function (both are different functions and combined for this explanation)

Pseudo code:

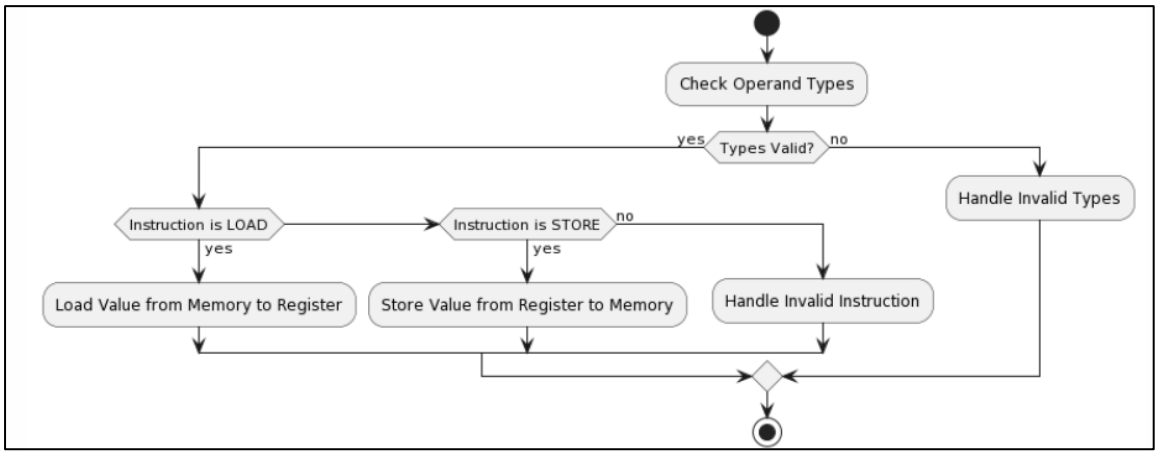
Procedure INPUT(operand)
  If operand is a register, Then
    Prompt user for a value
    Store the value in the specified register.
    Display stored value.
  Else
    Display error message "Invalid operand for IN instruction"
End Procedure
Procedure OUTPUT(operand)
  If operand is a register, Then
    Retrieve value from the specified register
    Display value
  Else
    Display error message "Invalid operand for OUT instruction"
End Procedure

```
void input(const string& operand) // Read input and store it in a register
{
  int value;
  cout << "Enter a value: ";
  cin >> value; //User inputs the value

  if (operand[0] == 'R')
      {
      R[operand[1] - '0'] = value; //Stores at the register
      cout << "Entered value " << value << " stored in register R" << (operand[1] - '0') << endl;
      }
  else
     {
     cout << "Invalid operand for IN instruction: " << operand << endl;
     }
}

void output(const string& operand)// Output the value of a register
{
  if (operand[0] == 'R')
      {
      int value = R[operand[1] - '0'];
      cout << "Value in register R" << (operand[1] - '0') << ": " << value << endl;
      }
  else
  {
   cout << "Invalid operand for OUT instruction: " << operand << endl;
  }
}
```



12. Void LoadStore function

```cpp
void loadandStore(const string& instruction, const string& operand1, const string& operand2) {
 if (instruction == "LOAD") {
 // If the first operand is a register and the second operand is a digit
     if (operand1[0] == 'R' && isdigit(operand2[0])) {
         int memoryIndex = 0;// Initialize memory index
         for (char c : operand2) { // For each character in the second operand
             memoryIndex = memoryIndex * 10 + (c - '0'); // Convert the character to a digit and add it to the memory index
         }
         R[operand1[1] - '0'] = memory[memoryIndex];// Load the value from the memory location into the register
         cout << "Loaded value " << R[operand1[1] - '0'] << " from memory location " << memoryIndex << " into register R" << (operand1[1] - '0') << endl;
     } else {
         cout << "Invalid operands for LOAD instruction: " << operand1 << ", " << operand2 << endl;
     }
 } else if (instruction == "STORE") {
     if (operand1[0] == 'R' && isdigit(operand2[0])) {
         int memoryIndex = 0;
         for (char c : operand2) {
             memoryIndex = memoryIndex * 10 + (c - '0');
         }
         memory[memoryIndex] = R[operand1[1] - '0'];
         cout << "Stored value " << R[operand1[1] - '0'] << " from register R" << (operand1[1] - '0') << " into memory location " << memoryIndex << endl;
     } else {
         cout << "Invalid operands for STORE instruction: " << operand1 << ", " << operand2 << endl;
         }
 }
}
```

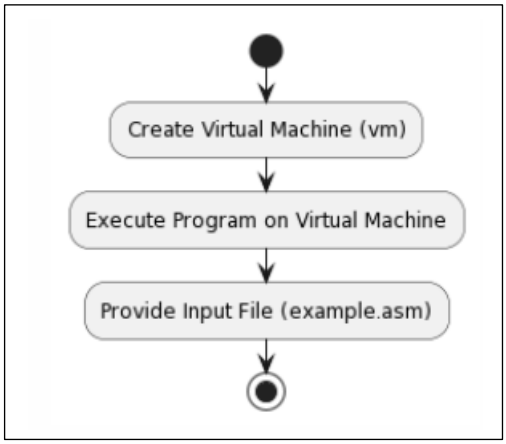


Pseudo code:

```
Procedure LOADANDSTORE(instruction, operand1, operand2)
    If instruction is "LOAD" and operand1 is a register Then
        Convert operand2 to an integer to get memory index
        Load value from memory location into register
        Display loaded value and memory location
    Else If instruction is "STORE" and operand1 is a register Then
        Convert operand2 to an integer to get memory index
        Store register value into memory location
        Display stored value and memory location
    Else
        Display error message "Invalid operands for instruction"
End Procedure.
```

13. Main function

```
int main()
{
VirtualMachine vm;
vm.executeProgram("example.asm");
return 0;
}
```



Pseudo code:

```
Procedure MAIN
    Create an instance of the VirtualMachine class named vm
    Execute the program "assembly_lang.asm" using vm
    Return 0
End Procedure
```

ASSEMBLY LANGUAGE INSTRUCTIONS

1) Input and Output operations

☐IN Rdst: read a value from the keyboard and store it into Rdst.

☐OUT Rsrc: write to the screen the value inside the register Rsrc.

The assembly instruction:

```
1    IN R0
2    OUT R0
```

The output:

For an example, if the user entered the value 14,

```
File opened successfully.
Enter a value: 14
Entered value 14 stored in register R0
Registers: 14 00 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 1
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

And for the OUT instruction:

```
Value in register R0: 14
Registers: 14 00 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 2
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

The flags are updated for these instructions, in here the Overflow Flag is triggered because the value 14 is in between 0 to 255 range. Each flag will be triggered for the size of the value that wished to be input and when the value wants to be output.

2) MOV Operations

MOV Rsrc, Rdst: copies the values stored in the source register to the destination register. The mov operation supports three modes only, these modes are explained in the following 3 examples.

The assembly instructions:

```
1   MOV 10, R0
2   MOV 12, R1
3   MOV R1, R0
```

The output:

For operation 1 and 2,

```
Stored value 10 to register R0
Registers: 10 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 1
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Stored value 12 to register R1
Registers: 10 12 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 2
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

The MOV (Number), (Register) operations will move the number given in the assembly instructions to the respective registers.

Meanwhile, for operation 3, the instruction given is to move the value from R1 to R0, so in R0 it changes value from 10 to 12 which is from R1.

No flags will be updated as these functions does not involve any numerical operations just moving data as numbers.

For operation 3:

```
Copied value 12 from register R1 to register R0
Registers: 12 12 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 3
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
File closed successfully.
```

3) Arithmetic Operations

☐ADD Rsrc, Rdst: Add the value of Rsrc to Rdst.

☐SUB Rsrc, Rdst: Subtract the value of Rsrc from Rdst.

☐MUL Rsrc, Rdst: Multiply the value of Rdst by the value of Rsrc.

☐DIV Rsrc, Rdst: Divide the value of Rdst by the value of Rsrc.

The above 4 instruction operate in a similar way. These operations only operate on the 7 registers directly.

The assembly instructions:

- Addition (ADD) and subtraction (SUB) operations:

```
1  MOV 5, R1
2  MOV 7, R0
3  ADD R0, 8
4  ADD R0, R1
5  SUB R1, 3
6  SUB R1, R0
```

The output:

```
Result after ADD: 15
Registers: 15 05 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 3
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Result after ADD: 20
Registers: 20 05 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 4
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

First, value 5 and 7 are moved to R1 and R0 respectively.

ADD R0, 8 (PC: 3)

Then, value in R0, 7 is added with 8 which results in 15.

ADD R0, R1 (PC :4)

R0, 15 now is added with value in R1, 5 and the value resulting in 20.

The Overflow flag is triggered for both operations as in 0 to 255 number range.

```
Result after SUB: 2
Registers: 20 02 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 5
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Result after SUB: -18
Registers: 20 -18 00 00 00 00 00 #
Flags: 0 1 0 0#
PC: 6
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

SUB R1, 3 (PC: 5)

For the subtraction, the value in R1 which is 5 is subtracted by 3 resulting in 2 now at R1.

SUB R1, R0 (PC :6)

Then, R1-R0 occurs, which is 02-20, giving a negative value, -18 to R1.

The Underflow flag is triggered for the sixth operation, as the result is in negative which is lesser than 0.

- Multiplication and division operations

```
1   MOV 5, R1
2   MOV 7, R0
3   MUL R1, 8
4   MUL R0, R1
5   DIV R1, 8
6   DIV R0, R1
```

The output:

```
Result after MUL: 40
Registers: 07 40 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 3
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Result after MUL: 280
Registers: 280 40 00 00 00 00 00 #
Flags: 0 0 1 0#
PC: 4
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

Now, for the same moved values from previous operation, 5 in R1 and 7 in R0, multiplication begins.

MUL R1, 8

This simply multiplies the value in R1, 5 by 8 giving the value 40 in R1.

MUL R0, R1

Then, this instruction multiplies value 40 and 7 from R0, resulting in 280 now at R1.

The OF triggered for the first operation as 40 is in the range between 0 to 255, meanwhile the Carry Flag is triggered because the value 280 exceeds 255 in the second operation.

```
Result after DIV: 5
Registers: 280 05 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 5
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Result after DIV: 56
Registers: 56 05 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 6
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

For the division processes later,

DIV R1, 8

The value in R1, 40 is divided by 8 resulting in 05 to R1.

DIV R0, R1

Now, the value in R0 will be divided by R1's value. Which is, 280 divided by 05, giving 56 to R0.

OF is set for both as 56 and 40 falls in the range of 0 to 255.

4) Increment and Decrement Operations

☐INC Rdst: Adds 1 to the value stored in the Rdst.

☐Dec Rdst: Subtracts 1 from the value stored in the Rdst.

The instructions:

```
1  MOV 1, R1
2  MOV 7, R0
3  INC R0
4  DEC R1
```

The output:

After moving the values 7 and 1 to R0 and R1 respectively, (the first two operations are omitted for brevity)

```
Register R0 incremented/decremented to 8
Registers: 08 01 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 3
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Register R1 incremented/decremented to 0
Registers: 08 00 00 00 00 00 00 #
Flags: 0 0 0 1#
PC: 4
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

INC R0

The value in R0, 7 is incremented (added 1) to 8 and stored back to R0.

DEC R1

The value in R1,1 is decremented (minus 1) to 0, (1-1=0) and stored back to R1.

For both operations, flags are updated. For the INC operation, the number 8 falls within range from 0 to 255, so the OF is triggered and for the DEC operation, the number 0 triggers Zero Flag (ZF), the last 0 digit changed to 1 as the result is 0.

5) LOAD and STORE operations

☐LOAD Rdst, addr: Load the value at memory address 'addr' into Rdst.

☐LOAD Rdst, [Raddr]: Load the value at memory address stored in register Raddr into Rdst.

☐STORE Rsrc, addr: Store the value in Rsrc into memory address 'addr'.

☐STORE Rsrc, [Raddr]: Store the value in Rsrc into memory address stored in register Raddr.

The instructions:

```
1  MOV 1, R1
2  MOV 7, R0
3  STORE R1, 43
4  LOAD R0, 40
```

The output:

```
Stored value 1 from register R1 into memory location 43
Registers: 07 01 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 3
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
Loaded value 0 from memory location 40 into register R0
Registers: 00 01 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 4
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 01 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

Assuming value 1 and 7 are moved to R1 and R0 respectively by the MOV operations,

STORE R1, 43

The instruction stores 43 from R1 register to the memory location number 43, which is 01.

LOAD R0, 40

This instruction loads out the value stored in memory location number 40 to register R0. Since, there are no numbers stored in memory location number 40, it returns 0 to R0.

No flags will be updated for these instructions.

6) Rotate operations.

☐ROL Rdst, count: Rotate the bits in Rdst left by 'count' positions.
☐ROR Rdst, count: Rotate the bits in Rdst right by 'count' positions.

ROL (Rotate on Left)

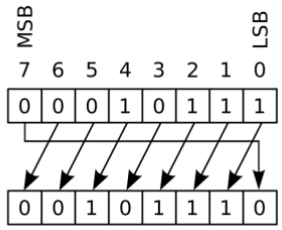
The instructions:

```
1   MOV 23, R1
2   ROL R1, 1
```

The output for 2nd instruction, (after 23 moved to R1)

```
Binary of operand before shift/rotate: 00010111
Binary of result after shift/rotate: 00101110
Decimal of result after shift/rotate: 46
Registers: 00 46 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 2
```

The value in R1 is converted to a binary value first, then the Bits are rearranged in a similar manner to the following example. After the rotation is done, convert back to decimal and store into R1.



After 23 moved to R1, the second instruction says 23 to rotate by left 1 time(s). In this case, the binary value of 23 is 00010111 (in 8 bits only) and once rotation on left, or in other words, multiplies by 2 gives 00101110. This is 46 in decimals, which will be stored at R1 back.

Basically (23x2= 46) but the switching happens in the binary form of the value, which will be stored at the register indicated.

Flags will be updated as usual for these operations also, OF triggered because 46 falls in the range between 0 to 255.

Rotating a value left by 8 times will give the same value as input in the register.

ROR (Rotate on Right)

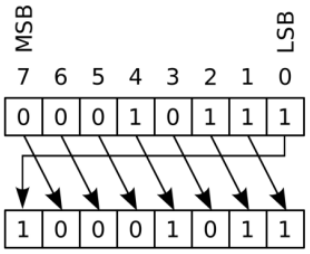
The instructions:

```
1   MOV 23, R1
2   ROR R1, 1
```

The output for 2nd instruction:

```
Binary of operand before shift/rotate: 00010111
Binary of result after shift/rotate: 10001011
Decimal of result after shift/rotate: 139
Registers: 00 139 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 2
```

The value in R1 is converted to a binary value first, then the Bits are rearranged in a similar manner to the following example. After the rotation is done, convert back to decimal and store into R1.



After 23 moved to R1, the second instruction says 23 to rotate by right 1 time(s). In this case, the binary value of 23 is 00010111 (in 8 bits only) and once rotation on right gives 10001011. This is 139 in decimals, which will be stored at R1 back.

Flags will be updated as usual for these operations also, OF triggered because 139 falls in the range between 0 to 255.

7) Shift Operations

SHL (Shift Left)

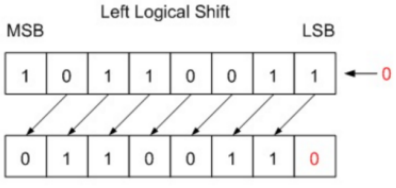
The instructions:

```
1   MOV 179, R1
2   SHL R1, 1
```

The output for 2nd instruction (after 179 is moved to R1):

```
Binary of operand before shift/rotate: 10110011
Binary of result after shift/rotate: 01100110
Decimal of result after shift/rotate: 102
Registers: 00 102 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 2
```

The value in R1 is converted to a binary value first, then the Bits are rearranged in a similar manner to the following example. After the shifting is done, convert back to decimal and store into R1.



After 179 moved to R1, the second instruction says 179 to shift left 1 time(s). In this case, the binary value of 179 is 10110011 (in 8 bits only) and once shifting left gives 01100110. This is 102 in decimals, which will be stored at R1 back.

Flags will be updated as usual for these operations also, OF triggered because 102 falls in the range between 0 to 255.

SHR (Shift Right)

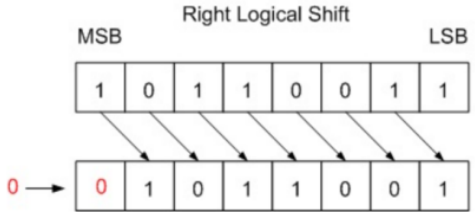
The instructions:

```
1   MOV 179, R1
2   SHR R1, 1
```

The output for the 2nd instruction:

```
Binary of operand before shift/rotate: 10110011
Binary of result after shift/rotate: 01011001
Decimal of result after shift/rotate: 89
Registers: 00 89 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 2
```

The value in R1 is converted to a binary value first, then the Bits are rearranged in a similar manner to the following example. After the shifting is done, convert back to decimal and store into R1.



After 179 moved to R1, the second instruction says 179 to shift right 1 time(s). In this case, the binary value of 179 is 10110011 (in 8 bits only) and once shifting right gives 01011001. This is 89 in decimals, which will be stored at R1 back.

Flags will be updated as usual for these operations also, OF triggered because 89 falls in the range between 0 to 255.

- This operation is basically dividing the input number by 2, which 179 gives nearest value of 89 once divided by 2.

INPUT VALIDATIONS

Input validations are crucial to ensure that the instructions and operands provided in the input file are correctly formatted and valid for the virtual machine. The program performs various checks to validate instructions and operands. Let's discuss the input validations in different parts of the code.

1. **isValidInstruction Function:**

```
bool isValidInstruction(const string& instruction) const // Check if the instruction is valid
{
    return (instruction == "MOV" || instruction == "ADD" || instruction == "SUB" || instruction == "MUL" ||
            instruction == "DIV" || instruction == "INC" || instruction == "DEC" || instruction == "IN" ||
            instruction == "OUT" || instruction == "LOAD" || instruction == "STORE" || instruction == "SHL" ||
            instruction == "SHR" || instruction == "ROL" || instruction == "ROR");
}
```

This function checks if the given instruction is valid by comparing it against a list of allowed instructions. If the instruction is not recognized, it is considered invalid.

2. **getOperandValue Function:**

```
int getOperandValue(const string& operand) const // Get the value of an operand based on its type
{
    // If the operand is a register, return the value in the register
    if (operand[0] == 'R') return R[operand[1] - '0'];
    if (operand[0] == '#') // If the operand is an immediate value
        {
        int value = 0;
        // For each character in the operand, starting from the second character
        for (size_t i = 1; i < operand.size(); ++i)
        {
            // Convert the character to a digit and add it to the value
            value = value * 10 + (operand[i] - '0');
        }
        return value; // Return the value
       }
       :
       :
```

This function is responsible for extracting the value of an operand based on its type. It validates the operand format and returns the corresponding value. If the operand is not recognized, it returns 0.

3. **executeProgram Function:**

```
void executeProgram(const string& inputFilePath)
{
    ifstream inputFile(inputFilePath);
    if (inputFile)
        {
        cout << "File opened successfully.\n";
        string line;

        while (getline(inputFile, line))
        {
            istringstream ins(line);
            string instruction;
            ins >> instruction;
            :
            :
```

Inside the **executeProgram** function, the program reads lines from the input file and checks if the instruction is valid using **isValidInstruction** before processing it. If an invalid instruction is encountered, an error message is displayed.

4. **Various Operand Validations:**

   The **move**, **arithmeticOperation**, **incrementDecrement**, **input**, and **output** functions perform operand validations specific to their operations. They check whether the operands are of the expected types and formats. If an operand is not valid, an error message is displayed.

```
    cout << "Invalid increment/decrement instruction: " << instruction << endl;
    return;
    }
    updateFlags(result);
    R[operand[1] - '0'] = result;
    cout << "Register R" << (operand[1] - '0') << " incremented/decremented to " << result << endl;
    }
else {
    cout << "Invalid operand for increment/decrement instruction: " << operand << endl;
     }
```

5. **loadandStore Function:**

```
void loadandStore(const string& instruction, const string& operand1, const string& operand2) {
 if (instruction == "LOAD") {
 // If the first operand is a register and the second operand is a digit
     if (operand1[0] == 'R' && isdigit(operand2[0])) {
         int memoryIndex = 0;// Initialize memory index
         for (char c : operand2) { // For each character in the second operand
             memoryIndex = memoryIndex * 10 + (c - '0'); // Convert the character to a digit and add it to the memory index
         }
         R[operand1[1] - '0'] = memory[memoryIndex];// Load the value from the memory location into the register
         cout << "Loaded value " << R[operand1[1] - '0'] << " from memory location " << memoryIndex << " into register R" << (operand1[1] - '0') << endl;
     } else {
         cout << "Invalid operands for LOAD instruction: " << operand1 << ", " << operand2 << endl;
     }
```

The **loadandStore** function validates operands specific to the LOAD and STORE instructions, ensuring that the operands are in the expected formats.

SAMPLE RUNS OF THE EXAMPLE ASSEMBLY PROGRAM

There will be 3 types of step-by-step executions for this assembly program, which is.

i) ADD, MUL, SUB, MOV, SHL, STORE
ii) MOV, DIV, INC, ROR, STORE
iii) IN, MOV, ROL, SHR, DEC, STORE, LOAD, OUT

For all these sample runs, first we must make sure that the “assembly_lang.asm” file is open along with the “virtual_machine.cpp” file in the same directory, like both windows in preferred programming platform like Codeblocks or Visual Studio. Apart from this, later will be a user manual will be attached, to show on how to compile and run using command prompt or compiler and writing the data to the “output.txt” file.

i) ADD, MUL, SUB, MOV, SHL, STORE

In the assembly_lang.asm file, these are the instructions given:

```
1  MOV 5, R1
2  MOV 13, R0
3  ADD R1, 7
4  ADD R0, R1
5  MUL R1, 25
6  SUB R0, 5
7  SHL R1, 1
8  STORE R1, 45
```

Step by step executing the program above:

1) **MOV 5, R1**

```
File opened successfully.
Stored value 5 to register R1
Registers: 00 05 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 1
```

5 is moved to register R1, and the program counter is incremented. No flags will be updated for these instructions.

2) **MOV 13, R0**

```
Stored value 13 to register R0
Registers: 13 05 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 2
```

13 is moved to register R0, and the program counter is incremented again resulting in 2. No flags will be updated for these instructions.

3) **ADD R1, 7**

```
Result after ADD: 12
Registers: 13 12 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 3
```

05 in R1 is added by 7 giving the value 12 to R1, the PC is 3. The Overflow Flag is triggered as 12 falls in the range of 0 to 255.

4) **ADD R0, R1**

```
Result after ADD: 25
Registers: 25 12 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 4
```

R0, 13 and R1, 12 are added together resulting in 25, this 25 will be at R0. The OF is triggered.

5) **MUL R1, 25**

```
Result after MUL: 300
Registers: 25 300 00 00 00 00 00 #
Flags: 0 0 1 0#
PC: 5
```

12 in R1 will be multiplied by 25, resulting into 300 and the value now in R1 is 300. The Carry Flag is set because 300 exceeds the range and more than 255.

6) **SUB R0, 5**

```
Result after SUB: 20
Registers: 20 300 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 6
```

25 in R0 is subtracted by 5 giving the value 20 into R0. The OF is triggered.

7) **SHL, R1, 1**

```
Binary of operand before shift/rotate: 00101100
Binary of result after shift/rotate: 01111100
Decimal of result after shift/rotate: 124
Registers: 20 124 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 7
```

300, R1 in binary form is 00101100 and if we shifted left it once, we get 01111100 which is 124 in decimals. This value stored back at R1. The OF is triggered as the 124 falls in the range of 0 to 255.

8) **STORE R1, 45**

Value in R1, which is 124 is now stored at memory location 45, as seen below. There are 8 instructions so the last PC is 8. A message “File closed successfully” printed if it has finised reading data and closed it without errors.

```
Stored value 124 from register R1 into memory location 45
Registers: 20 124 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 8
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 124 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
File closed successfully.
```

ii) MOV, DIV, INC, ROR, STORE operations

In the assembly_lang.asm file, these are the instructions given:

```
1  MOV 30, R1
2  MOV 15, R0
3  DIV R1, R0
4  INC R0
5  ROR R1, 1
6  STORE R1, 63
```

Step by step executing the program above:

1) MOV 30, R1

```
Stored value 30 to register R1
Registers: 00 30 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 1
```

Value 30 is moved to R1.

2) MOV 15, R0

```
Stored value 15 to register R0
Registers: 15 30 00 00 00 00 00 #
Flags: 0 0 0 0#
PC: 2
```

Value 15 is moved to R0.

3) DIV R1, R0

```
Result after DIV: 2
Registers: 15 02 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 3
```

Result in R1, 30 is divided by result in R0, 15 giving the value 02 to R1. The OF id triggered because 2 is within 0-255 range.

4) INC R0

```
Register R0 incremented/decremented to 16
Registers: 16 02 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 4
```

15 in R0 is incremented and now 16 in R0, the OF is triggered.

5) ROR R1, 1

```
Binary of operand before shift/rotate: 00000010
Binary of result after shift/rotate: 00000001
Decimal of result after shift/rotate: 1
Registers: 16 01 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 5
```

2 in R1 is 00000010 in binary. When rotated right for one time, the binary value is 00000001, which results 1 in decimals. The value 1 will be then stored back at R1, and the OF flag is triggered.

6) STORE R1, 63

```
Stored value 1 from register R1 into memory location 63
Registers: 16 01 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 6
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01
#
File closed successfully.
```

Value in R1, which is 01 is now stored at memory location 63. There are 6 instructions so the last PC is 6. A message “File closed successfully” printed if it has finised reading data and closed it without errors.

iii) IN, MOV, ROL, SHR, DEC, STORE, LOAD, OUT

In the assembly_lang.asm file, these are the instructions given:

```
1  IN R0
2  MOV 16, R6
3  ROL R0, 1
4  SHR R6, 3
5  DEC R0
6  STORE R6, 15
7  LOAD R3, 15
8  OUT R6
```

Step by step executing the program above:

1) IN R0

```
File opened successfully.
Enter a value:
```

Assuming the input value entered is 50,

```
Enter a value: 50
Entered value 50 stored in register R0
Registers: 50 00 00 00 00 00 00 #
Flags: 1 0 0 0#
PC: 1
```

The value entered, 50 will be stored in register R0 and the OF is triggered.

2) MOV 16, R6

```
Stored value 16 to register R6
Registers: 50 00 00 00 00 00 16 #
Flags: 1 0 0 0#
PC: 2
```

16 is moved to register R6.

3) ROL R0, 1

```
Binary of operand before shift/rotate: 00110010
Binary of result after shift/rotate: 01100100
Decimal of result after shift/rotate: 100
Registers: 100 00 00 00 00 00 16 #
Flags: 1 0 0 0#
PC: 3
```

50 in R0 is 00110010 in binary. When rotated left for one time, the binary value is 01100100, which results 100 in decimals. The value 100 will be then stored back at R0, and the OF flag is triggered.

4) SHR R6, 3

```
Binary of operand before shift/rotate: 00010000
Binary of result after shift/rotate: 00000010
Decimal of result after shift/rotate: 2
Registers: 100 00 00 00 00 00 02 #
Flags: 1 0 0 0#
PC: 4
```

16, R6 in binary form is 00010000 and if we shifted right it three times, we get 00000010 which is 2 in decimals. This value stored back at R6. The OF is triggered as the 2 falls in the range of 0 to 255.

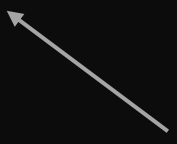
5) DEC R0

```
Register R0 incremented/decremented to 99
Registers: 99 00 00 00 00 00 02 #
Flags: 1 0 0 0#
PC: 5
```

100 in R0 is decremented and now 99 in R0, the OF is triggered.

6) STORE R6, 15

```
Stored value 2 from register R6 into memory location 15
Registers: 99 00 00 00 00 00 02 #
Flags: 1 0 0 0#
PC: 6
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```



Value in R6, which is 02 is now stored at memory location 15.

7) LOAD R3, 15

```
Loaded value 2 from memory location 15 into register R3
Registers: 99 00 00 02 00 00 02 #
Flags: 1 0 0 0#
PC: 7
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
```

Value stored in memory location 15, 02 just now will be loaded into register R3.

8) OUT R6

```
Value in register R6: 2
Registers: 99 00 00 02 00 00 02 #
Flags: 1 0 0 0#
PC: 8
Memory:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
#
File closed successfully.
```

The value in R6 which is 2 will be written out to the screen.

INSTRUCTIONS TO COMPILE AND RUN THE VIRTUAL MACHINE

1. Prerequisites:

C++ Compiler:
Ensure you have a C++ compiler installed on your system. Popular choices include GCC (GNU Compiler Collection) for Unix-like systems or MinGW for Windows.

C++ Standard Library:
The code utilizes standard C++ libraries, which are typically included with the compiler.

2. Upload the Code:

Download and upload the C++ code for the virtual machine interpreter. Save the code to a directory of your choice.

3. Compilation:

Open a terminal or command prompt and navigate to the directory where you saved the code.

Unix-like Systems (Using GCC):

```
g++ virtual_machine.cpp -o virtual_machine
```

Windows (Using MinGW):

```
g++ virtual_machine.cpp -o virtual_machine.exe
```

This command compiles the code and generates an executable named virtual_machine (or virtual_machine.exe on Windows).

4. Running the Virtual Machine Interpreter:

Unix-like Systems:

```
./virtual_machine assembly_lang.asm
```

Windows:

```
virtual_machine.exe assembly_lang.asm
```

Replace assembly_lang.asm with the path to your assembly code file.

5. Writing the data to output.txt file

Since the C++ program prints information to the console using cout, can use the > operator to redirect the output to a file.

Here's how you can modify the execution command to write the output to output.txt:

Unix-like Systems:

```
./virtual_machine assembly_lang.asm > output.txt
```

Windows:

```
virtual_machine.exe assembly_lang.asm > output.txt
```

This command will execute your virtual machine interpreter and redirect the output to output.txt. The > operator is used for output redirection.

After running the program with this modified command, the execution log, including the state of the virtual machine after each instruction, will be written to the output.txt file in the same directory. You can then open and review this file to see the program's detailed execution log.

6. Interpreting Assembly Programs:

Once the virtual machine is running, it will execute the assembly program specified in the command-line argument. Follow the on-screen instructions, and the interpreter will display the state of registers, flags, program counter, and memory after each instruction is executed.

7. Output Log:

The interpreter generates an output log file named output.txt in the same directory. This file contains the program's execution log, including the state of the virtual machine after each instruction. You can refer to this log for detailed information about the program's execution.

8. Troubleshooting:

Ensure the assembly code file is in the correct format and contains valid instructions.
Check for any error messages displayed during compilation and execution.
Make sure the input file path provided in the command-line argument is correct.

CONCLUSION

In conclusion, this task provided extensive hands-on experience in developing and building a virtual machine using a simplified assembly language. The virtual machine architecture consists of general-purpose data registers, a programme counter, flags, and memory. The assembly language allows a wide range of operations, including input/output, mathematical calculations, rotation and shifting, and memory operations.

The implementation entailed writing a strong C++ interpreter (runner) that reads assembly programmes from a file, executes them sequentially, and displays the virtual machine's state. The study described the virtual machine's architecture, interpreter algorithms, assembly language syntax and examples, flags handling, and a step-by-step explanation of how to execute a sample programme.

The task improved the understanding of low-level programming, memory management, and the complexities of running programs in a virtual machine. The significance of comprehensive documentation, adherence to coding norms, and modular design concepts have been emphasized throughout the implementation.

Overall, the assignment is a wonderful exercise in bringing theoretical knowledge to practical implementation, which helps better grasp computer architecture and assembly programming.

REFERENCES

1. ISO/IEC. (2017). ISO/IEC 14882:2017 - Programming Languages - C++. International Organization for Standardization.

2. Website links:

- https://www.google.com/search?q=what+is+assembly+language+interpreter%3F&tbm=isch&ved=2ahUKEwiwm-7-tYWEAxVMvGMGHWL4BpgQ2-cCegQIABAA&oq=what+is+assembly+language+interpreter%3F&gs_lcp=CgNpbWcQAzoFCAAQgAQ6BwgAEIAEEBhQsglY_EVgqEhoA3AAeACAAYYBiAGXCpIBBDEwLjSYAQCgAQGqAQtnd3Mtd2l6LWltZ8ABAQ&sclient=img&ei=8hS5ZbDaB8z4juMP4vCbwAk&bih=608&biw=1356&rlz=1C1ONGR_enMY1020MY1020&hl=en#imgrc=KRdTIvLG3epUGM

- https://cplusplus.com/reference/bitset/bitset/

- MinGW original page.
  https://www.mingw.org/ or
  https://sourceforge.net/projects/mingw/
  or https://mingw.osdn.io/

- Codeblocks homepage.
  https://www.codeblocks.org/

3. Individuals

- Ts. Goh Chien Le
  Assistant Professor
  FACULTY OF COMPUTING AND INFORMATICS (FCI)
  BR4015 , Multimedia University, Persiaran Multimedia
  63100, Cyberjaya, Selangor
  03-83125251
  clgoh@mmu.edu.my

- Mr. Sharaf El-Deen Sami Mohammed Al- Horani
  Specialist 2
  FACULTY OF COMPUTING AND INFORMATICS (FCI)
  BR4007 , Multimedia University, Persiaran Multimedia
  63100, Cyberjaya, Selangor
  03-83125232
  sharaf.horani@mmu.edu.my

-THE END-

TCP1101 Assignment Evaluation Form (40%)

Student ID:	1231303562	Total Score 40%
Student Name:	VENGGADANAATHAN A/L K.SALVAM	

Assignment implementation (30%)

Item	**Maximum marks**	**Actual Marks**
Inline comments, function and class comments, indentation, following proper C++ naming and styling conventions. Any violation is penalized by reduction of 0.5 mark.	1	
Reading from a file and writing to a file (0 if now files used)	2	
Use a project and multiple header and source files (0 if no project format).	1	
Must use vectors or arrays, functions and classes	2	
Implementing all the components of the virtual machine. Any missing requirement is penalized by deducting 1 mark.	4	
Implementing all the instructions of the assembly language. Any missing instruction or wrongly implemented is penalized by deducting 1 mark.	8	
The program demonstrates sufficient abstraction, modularity, and code reusability through classes and functions. [0: Below Expectation, 1: Within Expectation, 2: Exceed Expectation]	2	
Total:	20	

Report and Video (10%)

Item	**Maximum marks**	**Actual Marks**
Correct structured charts and the general flowchart of the Runner	2	
Correct flowcharts (any missing flowchart or pseudo code will cause you to lose 1 mark). For all the assembly instructions.	2	
Sample assembly programs (at least 3) and their outputs.	2	
User Documentation done and is coherence with the implementation	2	
The video presentation (How to compile and run the program and show how to run a sample program).	2	
Total:	10	

Examined with the midterm.

Item	**Maximum marks**	**Actual Marks**
Questions added to the midterm test as part of the assignment (individual marks)	10	
Total:	10	

Additional Comments
