SPOS Practical codes
GROUP A

Practical- 1
Design suitable Data structures and implement Pass-I and Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. The output of Pass-1 (intermediate code file and symbol table) should be input for Pass-II.

Ans.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>
#include <algorithm>

using namespace std;

// Data structures for symbol table, literal table, and intermediate code
unordered_map<string, int> symbolTable; // Symbol Table
vector<string> literalTable;             // Literal Table
vector<string> intermediateCode;         // Intermediate Code

unordered_map<string, string> opcodeTable = {
    {"LOAD", "01"},
    {"ADD", "02"},
    {"STORE", "03"},
    {"END", "04"},
};

void passI(const string& filename);
void passII(const string& outputFilename);
void generateFiles();
void generateMachineCode();

int main() {
    const string sourceFilename = "source.asm";
    // Pass I
    passI(sourceFilename);
    // Pass II
    generateMachineCode(); // Note: Using a separate function for Pass II
    // Generate output files
    generateFiles();
    return 0;
}

void passI(const string& filename) {
    ifstream sourceFile(filename);
```

```
    string line;
    int locationCounter = 0;

    if (!sourceFile.is_open()) {
        cerr << "Error opening source file!" << endl;
        return;
    }

    // Read each line from the source file
    while (getline(sourceFile, line)) {
        istringstream iss(line);
        string label, opcode, operand;

        iss >> label >> opcode >> operand;

        // Process START directive
        if (opcode == "START") {
            locationCounter = stoi(operand);
            continue; // Skip the rest for START
        }

        // Process the label
        if (!label.empty() && label != "END") {
            symbolTable[label] = locationCounter;
        }

        // Process the opcode
        if (opcode == "END") {
            intermediateCode.push_back(to_string(locationCounter) + "\t" + opcode);
            break; // End of processing
        } else {
            intermediateCode.push_back(to_string(locationCounter) + "\t" + opcode + "\t" +
operand);
            locationCounter += 1; // Increment for each instruction
        }

        // Identify literals
        if (!operand.empty() && operand[0] == '=') {
            string literal = operand.substr(1); // Remove '='
            if (find(literalTable.begin(), literalTable.end(), literal) == literalTable.end()) {
                literalTable.push_back(literal);
            }
        }
    }
    sourceFile.close();
}

void generateMachineCode() {
    cout << "Pass II Output (Machine Code):" << endl;
    cout << "Address\tMachine Code" << endl;

    // Generate machine code from intermediate code
```

```
    ofstream machineCodeFile("machine_code.txt");
    for (const auto& code : intermediateCode) {
        istringstream iss(code);
        string address, opcode, operand;
        iss >> address >> opcode >> operand;

        // Convert opcode to machine code
        string machineCode;
        if (opcodeTable.find(opcode) != opcodeTable.end()) {
            machineCode += opcodeTable[opcode];
            if (!operand.empty()) {
                if (operand[0] == '=') { // Literal
                    string literal = operand.substr(1); // Get the literal value
                    machineCode += literal; // Assuming direct representation for literals
                } else if (symbolTable.find(operand) != symbolTable.end()) { // Symbol
                    machineCode += to_string(symbolTable[operand]);
                } else {
                    machineCode += "00"; // Invalid operand
                }
            } else {
                machineCode += "00"; // No operand
            }
        } else {
            machineCode += "00"; // Invalid opcode
        }

        // Print and write the machine code to the file
        cout << address << "\t" << machineCode << endl;
        machineCodeFile << address << "\t" << machineCode << endl;
    }
    machineCodeFile.close();
}

void generateFiles() {
    // Write symbol table to a file
    ofstream symbolFile("symbol_table.txt");
    symbolFile << "Symbol Table:\n";
    for (const auto& entry : symbolTable) {
        symbolFile << entry.first << "\t" << entry.second << endl;
    }
    symbolFile.close();

    // Write literal table to a file
    ofstream literalFile("literal_table.txt");
    literalFile << "Literal Table:\n";
    for (const auto& literal : literalTable) {
        literalFile << literal << endl;
    }
    literalFile.close();

    // Write intermediate code to a file
    ofstream intermediateFile("intermediate_code.txt");
```

```
    intermediateFile << "Intermediate Code:\n";
    for (const auto& code : intermediateCode) {
        intermediateFile << code << endl;
    }
    intermediateFile.close();

    cout << "Output files generated: symbol_table.txt, literal_table.txt, intermediate_code.txt,
machine_code.txt" << endl;
}
```

Practical NO- 2

Design suitable data structures and implement Pass-1 and Pass-II of a two-pass macro-processor. The output of Pass-I (MNT, MDT and intermediate code file without any macro definitions) should be input for Pass-II.

Ans.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.LinkedHashMap;

public class SPOS2 {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("MACRO.asm"));
        FileWriter mnt = new FileWriter("mnt.txt");
        FileWriter mdt = new FileWriter("mdt.txt");
        FileWriter kpdt = new FileWriter("kpdt.txt");
        FileWriter pnt = new FileWriter("pntab.txt");
        FileWriter ir = new FileWriter("intermediate.txt");

        LinkedHashMap<String, Integer> pntab = new LinkedHashMap<>();
        String line;
        String macroname = null;
        int mdtp = 1, kpdtp = 0, paramNo = 1, pp = 0, kp = 0, flag = 0;

        while ((line = br.readLine()) != null) {
            String parts[] = line.split("\\s+");

            if (parts[0].equalsIgnoreCase("MACRO")) {
                flag = 1; // Indicate we're in a macro definition
                line = br.readLine();
                parts = line.split("\\s+");
                macroname = parts[0];

                if (parts.length <= 1) {
```

```
            mnt.write(parts[0] + "\t" + pp + "\t" + kp + "\t" + mdtp + "\t" + (kp == 0 ? kpdtp :
(kpdtp + 1)) + "\n");
            continue;
        }

        for (int i = 1; i < parts.length; i++) { // Processing of parameters
            parts[i] = parts[i].replaceAll("[&,]", "");
            if (parts[i].contains("=")) {
                ++kp;
                String keywordParam[] = parts[i].split("=");
                pntab.put(keywordParam[0], paramNo++);
                if (keywordParam.length == 2) {
                    kpdt.write(keywordParam[0] + "\t" + keywordParam[1] + "\n");
                } else {
                    kpdt.write(keywordParam[0] + "\t-\n");
                }
            } else {
                pntab.put(parts[i], paramNo++);
                pp++;
            }
        }

        mnt.write(parts[0] + "\t" + pp + "\t" + kp + "\t" + mdtp + "\t" + (kp == 0 ? kpdtp :
(kpdtp + 1)) + "\n");
        kpdtp = kpdtp + kp;

    } else if (parts[0].equalsIgnoreCase("MEND")) {
        mdt.write(line + "\n");
        flag = kp = pp = 0;
        mdtp++;
        paramNo = 1;
        pnt.write(macroname + ":\t");
        Iterator<String> itr = pntab.keySet().iterator();
        while (itr.hasNext()) {
            pnt.write(itr.next() + "\t");
        }
        pnt.write("\n");
        pntab.clear();

    } else if (flag == 1) {
        for (int i = 0; i < parts.length; i++) {
            if (parts[i].contains("&")) {
                parts[i] = parts[i].replaceAll("[&,]", "");
                mdt.write("(P," + pntab.get(parts[i]) + ")\t");
            } else {
                mdt.write(parts[i] + "\t");
            }
        }
        mdt.write("\n");
        mdtp++;
    } else {
        ir.write(line + "\n");
```

```
            }
        }

        br.close();
        mdt.close();
        mnt.close();
        ir.close();
        pnt.close();
        kpdt.close();

        System.out.println("Macro Pass 1 Processing done. :)");
    }
}
```

GROUP B-
Practical no 3-

Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

Ans.

```
import java.util.*;

class Process {
    String name;
    int arrivalTime;
    int burstTime;
    int priority; // Only used for Priority Scheduling
    int startTime;
    int completionTime;
    int waitingTime;
    int turnaroundTime;
    int remainingTime; // Only used for SJF and Round Robin

    Process(String name, int arrivalTime, int burstTime, int priority) {
        this.name = name;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.priority = priority;
        this.remainingTime = burstTime; // Initialize remaining time
    }
}

public class CPUSchedulingAlgorithms {
    // Helper function to print the Gantt chart and completion times
    private static void printGanttChart(List<Process> processes, String title) {
        System.out.println("\n" + title);
        int time = 0;

        // Print the Gantt chart timeline
```

```
        for (Process p : processes) {
            System.out.print(time + "\tl");
            time += p.burstTime;
        }
        System.out.println(time);
        System.out.println("Processes:");
        for (Process p : processes) {
            System.out.print(p.name + " ");
        }
        System.out.println();
        System.out.println("Completion Times:");
        for (Process p : processes) {
            System.out.print(p.completionTime + " ");
        }
        System.out.println();
        System.out.println("Turnaround Times:");
        for (Process p : processes) {
            System.out.print(p.turnaroundTime + " ");
        }
        System.out.println();
        System.out.println("Waiting Times:");
        for (Process p : processes) {
            System.out.print(p.waitingTime + " ");
        }
        System.out.println();

        // Calculate and print average turnaround and waiting times
        double avgTurnaroundTime = processes.stream().mapToInt(p ->
p.turnaroundTime).average().orElse(0.0);
        double avgWaitingTime = processes.stream().mapToInt(p ->
p.waitingTime).average().orElse(0.0);
        System.out.printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);
        System.out.printf("Average Waiting Time: %.2f\n", avgWaitingTime);
    }

    // FCFS Scheduling Algorithm
    public static void fcfsScheduling(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        int time = 0;

        for (Process p : processes) {
            if (time < p.arrivalTime) {
                time = p.arrivalTime; // Idle until process arrives
            }
            p.startTime = time;
            time += p.burstTime;
            p.completionTime = time;
            p.turnaroundTime = p.completionTime - p.arrivalTime;
            p.waitingTime = p.turnaroundTime - p.burstTime;
        }
        printGanttChart(processes, "FCFS Scheduling");
    }
```

```
    // SJF (Preemptive) Scheduling Algorithm
    public static void sjfPreemptiveScheduling(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        int time = 0;
        PriorityQueue<Process> queue = new PriorityQueue<>(Comparator.comparingInt(p
-> p.remainingTime));
        Map<String, Process> processMap = new HashMap<>();
        int index = 0;

        while (!queue.isEmpty() || index < processes.size()) {
            while (index < processes.size() && processes.get(index).arrivalTime <= time) {
                Process p = processes.get(index);
                queue.add(p);
                processMap.put(p.name, p);
                index++;
            }
            if (!queue.isEmpty()) {
                Process p = queue.poll();
                if (p.remainingTime == p.burstTime) {
                    p.startTime = time; // Set start time if it's the first time running
                }
                int timeSlice = Math.min(p.remainingTime, 1); // Run for 1 unit of time
                time += timeSlice;
                p.remainingTime -= timeSlice;
                if (p.remainingTime == 0) {
                    p.completionTime = time;
                    p.turnaroundTime = p.completionTime - p.arrivalTime;
                    p.waitingTime = p.turnaroundTime - p.burstTime;
                } else {
                    queue.add(p); // Re-add process if not completed
                }
            } else {
                time++; // Idle time
            }
        }
        printGanttChart(new ArrayList<>(processMap.values()), "SJF (Preemptive)
Scheduling");
    }

    // Priority Scheduling (Non-Preemptive) Algorithm
    public static void priorityScheduling(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        int time = 0;
        Queue<Process> queue = new PriorityQueue<>(Comparator.comparingInt(p ->
p.priority));
        Map<String, Process> processMap = new HashMap<>();
        int index = 0;

        while (!queue.isEmpty() || index < processes.size()) {
            while (index < processes.size() && processes.get(index).arrivalTime <= time) {
                Process p = processes.get(index);
```

```
            queue.add(p);
            processMap.put(p.name, p);
            index++;
        }
        if (!queue.isEmpty()) {
            Process p = queue.poll();
            if (p.startTime == 0) {
                p.startTime = time; // Set start time if it's the first time running
            }
            time += p.burstTime;
            p.completionTime = time;
            p.turnaroundTime = p.completionTime - p.arrivalTime;
            p.waitingTime = p.turnaroundTime - p.burstTime;
        } else {
            time++; // Idle time
        }
    }
    printGanttChart(new ArrayList<>(processMap.values()), "Priority Scheduling (Non-
Preemptive)");
  }

  // Round Robin Scheduling Algorithm
  public static void roundRobinScheduling(List<Process> processes, int quantum) {
    processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
    int time = 0;
    Queue<Process> queue = new LinkedList<>();
    Map<String, Process> processMap = new HashMap<>();
    int index = 0;

    while (!queue.isEmpty() || index < processes.size()) {
        while (index < processes.size() && processes.get(index).arrivalTime <= time) {
            Process p = processes.get(index);
            queue.add(p);
            processMap.put(p.name, p);
            index++;
        }
        if (!queue.isEmpty()) {
            Process p = queue.poll();
            if (p.startTime == 0) {
                p.startTime = time; // Set start time if it's the first time running
            }
            int timeSlice = Math.min(p.remainingTime, quantum); // Run for the quantum
time
            time += timeSlice;
            p.remainingTime -= timeSlice;

            if (p.remainingTime == 0) {
                p.completionTime = time;
                p.turnaroundTime = p.completionTime - p.arrivalTime;
                p.waitingTime = p.turnaroundTime - p.burstTime;
            } else {
                queue.add(p); // Re-add process if not completed
```

```
            }
        } else {
            time++; // Idle time
        }
    }
    printGanttChart(new ArrayList<>(processMap.values()), "Round Robin Scheduling");
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Read number of processes
    System.out.print("Enter the number of processes: ");
    int n = scanner.nextInt();
    scanner.nextLine(); // Consume newline

    List<Process> processes = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        System.out.println("Enter details for process " + (i + 1) + ":");
        System.out.print("Name: ");
        String name = scanner.nextLine();
        System.out.print("Arrival Time: ");
        int arrivalTime = scanner.nextInt();
        System.out.print("Burst Time: ");
        int burstTime = scanner.nextInt();
        System.out.print("Priority (use 0 if not needed): ");
        int priority = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        processes.add(new Process(name, arrivalTime, burstTime, priority));
    }

    // Clone processes for each algorithm
    List<Process> processesFCFS = new ArrayList<>(processes);
    List<Process> processesSJF = new ArrayList<>(processes);
    List<Process> processesPriority = new ArrayList<>(processes);
    List<Process> processesRR = new ArrayList<>(processes);

    // Run FCFS Scheduling
    fcfsScheduling(processesFCFS);
    // Run SJF Scheduling
    sjfPreemptiveScheduling(processesSJF);
    // Run Priority Scheduling
    priorityScheduling(processesPriority);
    // Run Round Robin Scheduling
    System.out.print("Enter the time quantum for Round Robin Scheduling: ");
    int quantum = scanner.nextInt();
    roundRobinScheduling(processesRR, quantum);

    scanner.close();
}
}
```

Practical No -4

Write a program to simulate Memory replacement algorithm.

Ans.

```
import java.util.*;

class Block {
    int id;
    int size;
    int start; // Start of allocation
    int end;   // End of allocation

    Block(int id, int size) {
        this.id = id;
        this.size = size;
        this.start = -1; // Initially, no allocation
        this.end = -1;   // Initially, no allocation
    }
}

class Process {
    String name;
    int size;
    int allocatedBlockId;

    Process(String name, int size) {
        this.name = name;
        this.size = size;
        this.allocatedBlockId = -1; // No block allocated
    }
}

public class MemoryPlacementStrategies {

    public static void firstFit(List<Block> blocks, List<Process> processes) {
        System.out.println("\nFirst Fit Allocation:");
        for (Process p : processes) {
            boolean allocated = false;
            for (Block b : blocks) {
                if (b.size >= p.size && b.start == -1) {
                    p.allocatedBlockId = b.id;
                    b.start = 0; // Start of allocation
                    b.end = p.size; // End of allocation
                    b.size -= p.size; // Reduce block size
                    allocated = true;
                    System.out.println(p.name + " allocated to Block " + b.id);
                    break;
                }
            }
            if (!allocated) {
                System.out.println(p.name + " could not be allocated.");
            }
        }
    }

    public static void bestFit(List<Block> blocks, List<Process> processes) {
        System.out.println("\nBest Fit Allocation:");
```

```
    for (Process p : processes) {
        Block bestBlock = null;
        int minSize = Integer.MAX_VALUE;
        for (Block b : blocks) {
            if (b.size >= p.size && b.size < minSize) {
                minSize = b.size;
                bestBlock = b;
            }
        }
        if (bestBlock != null) {
            p.allocatedBlockId = bestBlock.id;
            bestBlock.start = 0; // Start of allocation
            bestBlock.end = p.size; // End of allocation
            bestBlock.size -= p.size; // Reduce block size
            System.out.println(p.name + " allocated to Block " + bestBlock.id);
        } else {
            System.out.println(p.name + " could not be allocated.");
        }
    }
}

public static void nextFit(List<Block> blocks, List<Process> processes) {
    System.out.println("\nNext Fit Allocation:");
    int lastBlockIndex = 0;
    for (Process p : processes) {
        boolean allocated = false;
        int startIndex = lastBlockIndex;
        do {
            Block b = blocks.get(startIndex);
            if (b.size >= p.size && b.start == -1) {
                p.allocatedBlockId = b.id;
                b.start = 0; // Start of allocation
                b.end = p.size; // End of allocation
                b.size -= p.size; // Reduce block size
                allocated = true;
                System.out.println(p.name + " allocated to Block " + b.id);
                lastBlockIndex = (startIndex + 1) % blocks.size();
                break;
            }
            startIndex = (startIndex + 1) % blocks.size();
        } while (startIndex != lastBlockIndex);
        if (!allocated) {
            System.out.println(p.name + " could not be allocated.");
        }
    }
}

public static void worstFit(List<Block> blocks, List<Process> processes) {
    System.out.println("\nWorst Fit Allocation:");
    for (Process p : processes) {
        Block worstBlock = null;
        int maxSize = Integer.MIN_VALUE;
        for (Block b : blocks) {
            if (b.size >= p.size && b.size > maxSize) {
                maxSize = b.size;
                worstBlock = b;
            }
        }
        if (worstBlock != null) {
            p.allocatedBlockId = worstBlock.id;
```

```java
            worstBlock.start = 0; // Start of allocation
            worstBlock.end = p.size; // End of allocation
            worstBlock.size -= p.size; // Reduce block size
            System.out.println(p.name + " allocated to Block " + worstBlock.id);
        } else {
            System.out.println(p.name + " could not be allocated.");
        }
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Read number of memory blocks
    System.out.print("Enter the number of memory blocks: ");
    int numBlocks = scanner.nextInt();
    List<Block> blocks = new ArrayList<>();
    for (int i = 0; i < numBlocks; i++) {
        System.out.print("Enter size for Block " + (i + 1) + ": ");
        int size = scanner.nextInt();
        blocks.add(new Block(i + 1, size));
    }

    // Read number of processes
    System.out.print("Enter the number of processes: ");
    int numProcesses = scanner.nextInt();
    scanner.nextLine(); // Consume newline
    List<Process> processes = new ArrayList<>();
    for (int i = 0; i < numProcesses; i++) {
        System.out.print("Enter name for Process " + (i + 1) + ": ");
        String name = scanner.nextLine();
        System.out.print("Enter size for Process " + name + ": ");
        int size = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        processes.add(new Process(name, size));
    }

    // Run First Fit
    List<Block> blocksForFirstFit = new ArrayList<>(blocks);
    List<Process> processesForFirstFit = new ArrayList<>(processes);
    firstFit(blocksForFirstFit, processesForFirstFit);

    // Run Best Fit
    List<Block> blocksForBestFit = new ArrayList<>(blocks);
    List<Process> processesForBestFit = new ArrayList<>(processes);
    bestFit(blocksForBestFit, processesForBestFit);

    // Run Next Fit
    List<Block> blocksForNextFit = new ArrayList<>(blocks);
    List<Process> processesForNextFit = new ArrayList<>(processes);
    nextFit(blocksForNextFit, processesForNextFit);

    // Run Worst Fit
    List<Block> blocksForWorstFit = new ArrayList<>(blocks);
    List<Process> processesForWorstFit = new ArrayList<>(processes);
    worstFit(blocksForWorstFit, processesForWorstFit);

    scanner.close();
}
}
```