1. Design suitable Data structures and implement Pass-I of two pass assembler to generate intermediate code. Sample input is following assembly language program. START 200 MOVER AREG, ='5' MOVEM AREG, X L1 MOVER BREG, ='2' ORIGIN L1+3 LTORG NEXT ADD AREG, ='1' SUB BREG, ='2' BC LT, BACK LTORG BACK EQU L1 ORIGIN NEXT+5 MULT CREG, ='4' STOP X DS 1 END

   Aim:
   To implement **Pass-I of Two Pass Assembler** to generate **Intermediate Code**, **Symbol Table** and **Literal Table**.

   Theory (Say in viva in 20 seconds):
   Pass-I of the assembler **scans the assembly program line-by-line**, assigns addresses using **Location Counter (LC)**, fills the **Symbol Table**, collects **Literals** in the **Literal Table**, and generates **Intermediate Code (IC)**.
   No machine code is generated in this phase.
   Literal addresses are assigned at **LTORG / END**.

   Algorithm:

1. Initialize **LC = starting address** specified in START instruction.
2. For each line in source program:
   - If label exists → enter into **Symbol Table** with current LC.
   - Identify instruction type (Imperative, Declarative, Assembler Directive).
   - Generate **Intermediate Code** accordingly.
   - If literal appears (=value) → store it in **Literal Table**.
3. On encountering **LTORG or END**, assign addresses to unassigned literals.
4. Print **SYMTAB**, **LITTAB**, and **IC**.

2. Design Pass II of a two pass assembler for following intermediate code and data structures to generate machine code. Symbol table LITERAL TABLE Symbol address LITERAL ADRESS X 214 =’5’ 205 L1 202 =’2’ 206 NEXT 207 =’1’ 210 BACK 202 =’2’ 211 =’4’ 215 Intermediate code (AD,01) (C,200) (IS,04) 1 (L,1) (IS,05) 1 (S,1) (IS,04) 2(L,2) (AD,03) (S,2)+3 (AD,05) (L,1) (L,2) (IS,01) 1 (L,3) (IS, 02) 2 (L,4) (IS,07) 1(S,4) (AD,05) (L,3) (L,4) (AD,04) (S,2) (IS,03) 3 (L,5) (IS,00) (DL,02) (C,1) (AD,02)

Aim

To design and implement **Pass-II of the two-pass assembler** that reads the **Intermediate Code (IC)** produced in Pass-I and generates the **final machine code** using **Symbol Table** and **Literal Table**.

Theory

Pass-II converts **mnemonics and symbolic addresses** into their **actual machine code values**. It reads:

- **Intermediate Code (IC)**
- **Opcode Table (OPTAB)**
- **Symbol Table (SYMTAB)**
- **Literal Table (LITTAB)**

Pass-II replaces:

(IS, opcode) → Actual machine opcode

(S, index) → Address from Symbol Table

(L, index) → Address from Literal Table

(C, value) → Constant value

Assembler directives such as **START, END, ORIGIN, EQU, LTORG** do **not** produce machine code.

Algorithm (Simple and Viva-Friendly)

1. Read **Intermediate Code line by line**.
2. For each line:
    - Identify instruction class:

        - IS → **Imperative Instruction**
        - DL → **Declarative Instruction**
        - AD → **Assembler Directive**
3. If the line is IS:
    - Convert opcode → machine opcode
    - Convert register → register code
    - Replace symbol/literal reference → actual address
4. If DL:
    - Allocate memory but do not generate machine code
5. Print final **Machine Code** instruction line-by-line.

Practical 3 – Pass-I of Macro Processor

Aim

To design and implement **Pass-I of a two-pass macro processor** to construct:

- **MNT (Macro Name Table)**
- **MDT (Macro Definition Table)**
- **ALA (Argument List Array)**

Theory

A **macro** is a sequence of assembly instructions that is given a name and can be reused anywhere in the program.
Macro Processor **expands** these macros before actual assembly.

The **Macro Processor** works in **two passes**:

Pass	Purpose
Pass-I	Identify macro definitions and store them in **MNT** and **MDT**, convert parameters into placeholders.
Pass-II	Perform **macro expansion** by replacing macro invocation with corresponding instructions from MDT.

Algorithm (Pass-I)

1. Read source program **line-by-line**.
2. When encountering MACRO:
    - Read next line → macro header.
    - Extract **macro name** and its parameters.
    - Enter macro name into **MNT** with current **MDT index**.
    - Store parameters in **ALA** as positional #1, #2, #3 ....
3. Copy each instruction of macro body into **MDT** replacing &ARG with corresponding #i.
4. When MEND is encountered:
    - End definition and write MEND into MDT.
5. Continue scanning until end of file.

Practical 4 – Pass-II of Macro Processor

Aim

To perform **macro expansion** using the tables generated in Pass-I (MNT, MDT, ALA) and produce the **final expanded source program**.

Theory

In **Pass-II**, the macro processor:

- Reads the input program again
- Whenever a **macro call** is encountered, it:
    1. **Looks up** the macro name in **MNT** to find where its definition starts in **MDT**
    2. Uses **ALA** to map **actual parameters** to **formal parameters**
    3. Copies macro body from **MDT** into output, replacing **positional parameters (#1, #2, …)** with actual arguments
- Non-macro lines are copied to output **directly** without changes.

No new tables are created here.
Pass-II only **expands** macros to produce the final code.

Algorithm (Pass-II)

1. Read program line-by-line.
2. If the line is **not** a macro call:
    - Copy it to output as is.
3. If the line **is** a macro call:
    - Retrieve its entry in **MNT** to get starting index of body in **MDT**.
    - Fill **ALA** with actual arguments.
    - For each line in MDT until MEND:
        - Replace positional parameters (#1, #2, …) with actual arguments.
        - Write the expanded line to output.
4. Continue until END is reached.

Practical 5 – Synchronization Using Mutex

Aim

To solve the classical synchronization problem by using a **Mutex (Mutual Exclusion lock)** to ensure that only one process accesses the critical section at a time.

Theory

A **mutex** is a binary locking mechanism used to prevent **race conditions**.
When a process enters its **critical section**, it locks the mutex so that **no other process** can access shared data at the same time.
After finishing, the process unlocks the mutex, allowing the next waiting process to enter.

Algorithm

1. Initialize **mutex = 1**.
2. When a process wants to enter the **critical section**, it checks:
    - If **mutex == 1**, lock it (mutex = 0).
3. Execute the **critical section** safely.
4. After completing, **release** the mutex (mutex = 1).
5. Other processes wait until mutex becomes free.
6. Continue until all processes finish execution.

Practical 6 – Synchronization Using Semaphore

Aim

To solve the classical synchronization problem using **Semaphores** to coordinate multiple processes accessing shared resources.

Theory

A **semaphore** is an integer variable used for **process synchronization**.
It controls access to shared resources by using two atomic operations:

- **wait() / P()** → decreases semaphore value → blocks if value < 0
- **signal() / V()** → increases semaphore value → wakes up blocked process

Semaphores prevent **race conditions** and ensure **mutual exclusion** in the critical section.

Algorithm

1. Initialize semaphores:
    - **mutex = 1** (for locking critical section)
    - **full = 0** (items produced)
    - **empty = n** (space available)
2. **Producer Process:**
    - wait(empty)
    - wait(mutex)
    - produce item (critical section)
    - signal(mutex)
    - signal(full)
3. **Consumer Process:**
    - wait(full)
    - wait(mutex)
    - consume item (critical section)
    - signal(mutex)
    - signal(empty)
4. Repeat steps until all items are produced and consumed.

Practical 7 – CPU Scheduling: FCFS (First Come First Served)

Aim

To simulate the **First Come First Served (FCFS)** CPU scheduling algorithm and calculate **Waiting Time** and **Turnaround Time** for each process.

Theory

In **FCFS scheduling**, processes are executed in the **order of their arrival** in the ready queue.
It is **non-preemptive**, meaning once a process gets the CPU, it runs until completion.
This scheduling can lead to the **Convoy Effect**, where shorter processes wait behind longer ones.

Algorithm

1. Sort processes in **increasing order of Arrival Time**.
2. Initialize **time = 0**.
3. For each process:
    - If the CPU is idle and the process has not yet arrived, set time = arrival_time.
    - Compute **Waiting Time**:
    - WT = time - arrival_time
    - Compute **Completion Time**:
    - CT = time + Burst_Time
    - Compute **Turnaround Time**:
    - TAT = CT - arrival_time
    - Update **time = CT**.
4. Display **WT** and **TAT** for all processes.

Practical 8 – CPU Scheduling: SJF (Preemptive) / SRTF (Shortest Remaining Time First)

Aim

To simulate the **Shortest Job First (Preemptive)** CPU scheduling algorithm and calculate **Waiting Time** and **Turnaround Time** for each process.

Theory

In **SJF Preemptive**, also called **SRTF**, at any given time the **process with the shortest remaining burst time** is selected to execute.
If a new process arrives with a **shorter burst time** than the currently running one, the CPU is **preempted** and assigned to that new process.
This algorithm minimizes **average waiting time**, but requires knowledge of burst times in advance.

Algorithm

1. Maintain arrays for **Arrival Time (AT)** and **Remaining Time (RT)** for all processes.
2. Set **current time = 0**, **completed = 0**.
3. At each time unit:
    - From all processes that **have arrived** and **not completed**, select the process with **minimum remaining time**.
    - Decrease its remaining time by 1 (process executes for 1 unit).
4. If a process's remaining time becomes 0:
    - Mark process as **completed**.
    - Calculate:
    - Completion Time (CT) = current time
    - Turnaround Time (TAT) = CT - AT
    - Waiting Time (WT) = TAT - Burst Time
5. Repeat until all processes are completed.
6. Display **WT** and **TAT**.

Practical 9 – CPU Scheduling: Priority (Non-Preemptive)

Aim

To simulate the **Priority Scheduling (Non-Preemptive)** CPU scheduling algorithm and calculate **Waiting Time** and **Turnaround Time** for each process.

Theory

In **Priority Scheduling**, each process is assigned a **priority value**.
The CPU is allocated to the process with the **highest priority** (usually **lowest priority number** = highest priority).
This algorithm is **non-preemptive**, meaning once a process starts execution, it runs until completion.
If multiple processes have the same priority, the one that arrived earlier is selected.

Algorithm

1. Accept input of processes with:
    - Process ID
    - Arrival Time
    - Burst Time
    - Priority Value
2. Sort all processes by:
    - **Arrival Time**, and if equal,
    - **Priority (lowest number first)**.
3. Set **current time = 0**.
4. For each process in sorted order:
    - If current time < arrival time, set current time = arrival time.
    - Compute:
    - Waiting Time (WT) = current time - Arrival Time
    - Completion Time (CT) = current time + Burst Time
    - Turnaround Time (TAT) = CT - Arrival Time
    - Update current time = CT.
5. Display **WT** and **TAT** for each process.

Practical 10 – CPU Scheduling: Round Robin (Preemptive)

Aim

To simulate the **Round Robin** CPU scheduling algorithm using a given **time quantum**, and calculate **Waiting Time** and **Turnaround Time** for each process.

Theory

Round Robin scheduling assigns each process a fixed **time quantum**.
Processes are executed in a **circular queue**.
If a process does not finish within its allotted time slice, it is **preempted** and placed back into the ready queue.
This ensures **fair CPU sharing** among all processes and is widely used in **time-sharing systems**.

Algorithm

1. Accept input of:
    - Process ID
    - Arrival Time
    - Burst Time
    - Time Quantum (Q)
2. Initialize:
    - Remaining Time = Burst Time of each process
    - Current Time = 0
    - Ready Queue = empty
3. Add processes to the ready queue as their **arrival time ≤ current time**.
4. While there are unfinished processes:
    - Take the **first process** from the ready queue.
    - Execute it for:
    - min(Time Quantum, Remaining Time)
    - Reduce Remaining Time.
    - Increase Current Time accordingly.
    - If Remaining Time becomes 0:
        - Compute:

        - Completion Time (CT) = current time
        - Turnaround Time (TAT) = CT - AT
        - Waiting Time (WT) = TAT - BT
    - Else:
        - Place the process back into the queue.
    - Add any newly arrived processes to the queue.
5. Display **WT** and **TAT** for all processes.

Practical 11 – Memory Allocation: Best Fit

Aim

To simulate the **Best Fit** memory allocation strategy and allocate processes to memory blocks with **minimum internal fragmentation**.

Theory

In **Best Fit**, each process is allocated to the **smallest memory block** that is **large enough** to accommodate it.
This helps in reducing **internal fragmentation**, because we try to fit the process in the block that leaves the **least leftover space**.

However, Best Fit may lead to **external fragmentation**, since many small unused gaps can be left across memory.

Algorithm

1. Accept sizes of memory blocks.
2. Accept sizes of processes to be allocated.
3. For each process:
    - Search all memory blocks.
    - Select the block that:
        - Is **greater than or equal** to the process size.
        - Has the **smallest size** among all eligible blocks.
4. If such a block exists:
    - Allocate the process to that block.
    - Reduce the block size by the allocated process size.
5. If no suitable block is found:
    - The process **cannot be allocated**.
6. Display allocation results.

Practical 12 – Memory Allocation: First Fit

Aim

To simulate the **First Fit** memory allocation strategy and allocate processes to memory blocks in the order they appear.

Theory

In **First Fit**, memory blocks are scanned **from the beginning**, and a process is allocated to the **first available block** that is **large enough** to hold it.
It is **simple** and **fast** because it does not search for the optimal block — it stops as soon as it finds a suitable one.

However, it can lead to **external fragmentation**, as memory becomes scattered with small free gaps.

Algorithm

1. Take input for sizes of memory blocks.
2. Take input for sizes of processes to be allocated.
3. For each process:
    - Scan the list of memory blocks from the start.
    - If a block is **large enough**:
        - Allocate the process to that block.
        - Reduce block size by process size.
        - Stop searching for this process.
4. If no suitable block is found:
    - The process **remains unallocated**.
5. Display allocation results.

Practical 13 – Memory Allocation: Next Fit

Aim

To simulate the **Next Fit** memory allocation strategy and allocate processes to memory blocks starting the search from the **last allocated position**.

Theory

Next Fit works similar to **First Fit**, but instead of always starting the search from the **beginning** of the memory, it remembers the **last block** where a process was allocated.
For the next process, the search begins **from that point onward**, looping back to the start if needed.

This strategy helps in reducing time spent repeatedly scanning memory from the beginning, but may still lead to **external fragmentation**.

Algorithm

1. Take input for sizes of memory blocks.
2. Take input for sizes of processes to be allocated.
3. Maintain an index **j** that marks the **last allocated block position**.
4. For each process:
    - Starting from block index j, scan memory sequentially.
    - If the block is **large enough**:
        - Allocate the process to that block.
        - Reduce the block size by process size.
        - Update j to the current block position.
        - Stop searching for this process.
    - If the end of the list is reached, wrap around to the start.
5. If no suitable block is found, the process remains **unallocated**.
6. Display allocation results.

Practical 14 – Memory Allocation: Worst Fit

Aim

To simulate the **Worst Fit** memory allocation strategy and allocate processes to the **largest available memory block**.

Theory

In **Worst Fit**, each process is allocated to the **largest block** that is big enough to hold it. This aims to **reduce external fragmentation** by leaving larger remaining spaces available for future allocations rather than creating many small unusable gaps.

However, this method may still produce fragmentation but tends to create **larger free blocks**.

Algorithm

1. Input the sizes of memory blocks.
2. Input the sizes of processes.
3. For each process:
    - Search all memory blocks.
    - Identify the block that:
        - Is **greater than or equal** to the process size.
        - Has the **largest size** among all such blocks.
4. If such a block exists:
    - Allocate the process to that block.
    - Reduce the block size by the process size.
5. If no suitable block is found:
    - Mark the process as **Not Allocated**.
6. Display allocation results.

Practical 15 – Page Replacement: FIFO (First In First Out)

Aim

To simulate the **FIFO Page Replacement** algorithm and count the number of **page faults**.

Theory

In **FIFO**, the page that has been in memory for the **longest time** is the one that gets **replaced first** when a new page needs to be loaded.
A **queue** is used to track the order of pages.
FIFO is simple but can cause **Belady's Anomaly**, where increasing the number of frames can **increase** page faults.

Algorithm

1. Take the **reference string** and number of **frames**.
2. Initialize an empty **queue** to store pages currently in memory.
3. For each page in the reference string:
    - If the page is **already in memory**, do nothing.
    - Else:
        - If memory is **not full**, add the page to the queue.
        - If memory is **full**, remove the **front (oldest)** page and insert the new page.
    - Count page faults whenever a new page is added.
4. Display total **page faults**.

Practical 16 – Page Replacement: Optimal

Aim

To simulate the **Optimal Page Replacement** algorithm and count the number of **page faults**.

Theory

The **Optimal Page Replacement** algorithm replaces the page that will **not be used for the longest duration in the future**.
It gives the **minimum possible number of page faults**, meaning **no other algorithm can perform better** for a given reference string.
However, it is **not practical** in real systems because it requires **future knowledge** of memory references.

Algorithm

1. Take the **reference string** and number of **frames**.
2. Initialize memory frames as **empty**.
3. For each page in the reference string:
    - If the page is **already in memory**, continue.
    - Else:
        - If there is a **free frame**, insert the page.
        - Otherwise:
            - For each page currently in memory, look **ahead** in the reference string to determine when it will be used next.
            - Replace the page that will be used **farthest in the future** (or **not used again**).
    - Increment page fault count whenever a page is replaced or newly inserted.
4. Display total **page faults**.

Practical 17 – Page Replacement: LRU (Least Recently Used)

Aim

To simulate the **LRU (Least Recently Used)** page replacement algorithm and count the number of **page faults**.

Theory

In **LRU**, the page that has **not been used for the longest time in the past** is replaced first. The assumption is that pages used recently are **more likely to be used again**, so LRU tries to minimize page faults by replacing the **least recently accessed** page.
It requires tracking the **order of page usage**, often maintained using a **stack** or **list**.

Algorithm

1. Take the **reference string** and number of **frames**.
2. Initialize the set or list of pages in memory as **empty**.
3. For each page in the reference string:
    - If the page is **already in memory**:
        - Update its position to mark it as **most recently used**.
    - Else:
        - If memory has a **free frame**, insert the page.
        - If memory is **full**, remove the **least recently used** page.
        - Insert the new page.
        - Increment the **page fault counter**.
4. Continue until all references are processed.
5. Display the total **page faults**.