□ SPOS Oral / Viva Questions — Two Pass Assembler

1. What is the purpose of Pass-I in the assembler?

Answer: Pass-I is used to **scan the source code** and **build the symbol table** while assigning addresses to each instruction and data item. It also generates an **intermediate code** that will be used by Pass-II.

---

2. What is the purpose of Pass-II in the assembler?

Answer: Pass-II uses the **intermediate code** and the **symbol table** from Pass-I to generate the **final machine (target) code**. It replaces symbols with their actual memory addresses and converts mnemonics into opcodes.

---

3. What is a Symbol Table?

Answer: A **Symbol Table** is a data structure that stores the **symbol names (labels)** along with their **corresponding addresses**. In your code, it's stored in `"symtab.txt"` and used by Pass-II for address resolution.

---

4. What is an Intermediate Code in an assembler?

Answer: Intermediate code is a **transitional representation** of the source program that records each instruction along with its location counter (LC). It simplifies Pass-II processing. In your code, `"intermediate.txt"` acts as the intermediate file.

---

5. What are the main outputs of Pass-I?

Answer:

- **Symbol Table (symtab.txt)**
- **Intermediate Code (intermediate.txt)**
- A message: *"Pass-I complete. Intermediate code and Symbol Table generated."*

---

6. What are the main outputs of Pass-II?

Answer:

- **Target Code (target.txt)**
- A message: *"Pass-II complete. Target code generated in target.txt."*

---

7. Explain the role of the Location Counter (LC).

Answer: LC keeps track of the **address of the next instruction or data**. It starts from the address given by the `START` directive and increments as each instruction is processed.

---

8. What is the function of the `START` and `END` directives?

Answer:

- `START`: Specifies the starting address of the program.
- `END`: Marks the end of the assembly source code and tells the assembler to stop reading further instructions.

---

9. What is the difference between DC and DS?

Answer:

- **DC (Define Constant):** Reserves one memory word and initializes it with a constant value.
- **DS (Define Storage):** Reserves a number of memory locations but does not initialize them.

---

10. What happens when the assembler encounters a label in Pass-I?

Answer: The assembler adds the **label name and its corresponding LC value** into the **symbol table** for later reference in Pass-II.

---

11. How are mnemonics and registers handled in Pass-II?

Answer:

- Mnemonics like `ADD`, `MOVER`, `MOVEM` are mapped to **numeric opcodes** using a **list or table**.
- Registers like `AREG`, `BREG` are assigned **numeric codes** (1, 2, etc.).
- Both are combined with the symbol's address to form machine code.

---

12. What are assembler directives? Give examples.

Answer: Assembler directives are **instructions to the assembler**, not to the CPU. Examples: `START`, `END`, `DS`, `DC`.

---

13. What is the input to Pass-II of the assembler?

Answer:

- Intermediate file (`intermediate.txt`)
- Symbol Table (`symtab.txt`)

---

14. What is the output of Pass-II?

Answer: The **target code (machine code)** written into `target.txt`.

---

15. What does the line `int opCodeVal = inst.indexOf(opcode) + 1;` do in Pass-II?

Answer: It finds the **index (position)** of the given opcode in the list of known instructions and converts it into a **numeric opcode value** by adding 1 (since indexing starts from 0).

---

16. What is the use of the HashMap in Pass-II?

Answer: The HashMap is used to **store and retrieve symbols and their addresses** quickly from the symbol table (`symtab.txt`).

---

17. Why do we need two passes instead of one?

Answer: Because the assembler may encounter **forward references** (symbols defined later in the program). Pass-I records their names and locations, and Pass-II resolves their addresses.

---

18. What does the intermediate file contain in your program?

Answer: It contains each line of source code along with the **location counter**, **opcode**, and **operand**, e.g.:

```
100 START   100
100 MOVER   AREG,DATA
101 ADD AREG,ONE
102 MOVEM   AREG,RESULT
103 END
```

---

19. What is the output format of the target file?

Answer: Each line of the **target.txt** file contains:

```
<LC>  <Opcode>  <RegisterCode>  <Address>
```

Example:

```
100 03  1   104
101 01  1   105
102 04  1   106
```

---

20. What happens if a symbol is not found in the symbol table during Pass-II?

Answer: The assembler assigns a **default address (000)** using:

```
String address = symTable.getOrDefault(symName, "000");
```

---

21. What are the main data structures used in your Pass-I and Pass-II?

Answer:

- **Pass-I:** 2D arrays (`symtab[][]`) for symbol table
- **Pass-II:** `HashMap` for symbol lookup, `List` for instructions and registers

---

22. What are pseudo-ops or assembler directives in your code?

Answer: `START`, `END`, `DC`, and `DS` — they are not machine instructions but control assembler behavior.

---

23. What kind of assembler have you implemented — single pass or two pass?

Answer: A **Two-Pass Assembler** implemented in **Java**, where Pass-I and Pass-II are separate programs.

---

24. What are the files generated by your assembler?

Answer:

1. `symtab.txt` – Symbol Table
2. `intermediate.txt` – Intermediate Code
3. `target.txt` – Final Target Code

---

25. What improvements could be added to your assembler?

Answer:

- Add **error handling** for undefined symbols
- Support for **macro processing**
- Include **Literal Table (LITTAB)**
- Use **opcode table files (OPTAB)** instead of hardcoded lists

---

□ SPOS Oral / Viva Questions — Two Pass Macro Processor

1. What is a Macro Processor?

Answer: A **Macro Processor** is a program that **expands macros** (user-defined sequences of instructions) in assembly language programs before actual assembly. It replaces macro calls with the actual set of instructions defined in the macro.

---

2. What is a Macro?

Answer: A **Macro** is a **group of instructions** that can be represented by a **single name**. When invoked, the assembler replaces the macro call with all the statements defined in the macro body.

Example:

```
MACRO
INCR &ARG1
A 1,&ARG1
MEND
```

---

3. What is the need for macros in assembly language?

Answer: Macros help in:

- Reducing repetitive code.
- Increasing program readability.
- Simplifying complex or frequently used instruction sequences.

---

4. What are the two passes in a Macro Processor?

Answer:

- **Pass-I:** Macro **definition processing** (builds MNT and MDT tables).
- **Pass-II:** Macro **expansion phase** (replaces macro calls with actual macro definitions).

---

5. What is the purpose of Pass-I?

Answer: Pass-I:

- Scans the program for **macro definitions**.
- Builds the **Macro Name Table (MNT)** and the **Macro Definition Table (MDT)**.
- Produces **intermediate code** excluding macro definitions.

---

6. What is the purpose of Pass-II?

Answer: Pass-II:

- Reads the **intermediate code** and **expands macro calls** using entries from **MNT** and **MDT**.
- Generates the final **expanded source code** (output file).

---

7. What is MNT (Macro Name Table)?

Answer: The **MNT** stores:

- The **name of the macro**, and
- The **starting index** of its definition in the **MDT**.

Example entry in `MNT.txt`:

```
INCR    0
```

---

8. What is MDT (Macro Definition Table)?

Answer: The **MDT** contains the **body of the macro**, line by line. Each line may contain parameters represented by **&ARG** type arguments. Example `MDT.txt` content:

```
0    INCR &ARG1
1    A 1,&ARG1
2    MEND
```

---

9. What is the role of the Intermediate File in Pass-I?

Answer: It contains the **original program** excluding the **macro definition**, i.e., only the **main program and macro calls** remain. Example:

```
START 101
INCR TERM
MOVEM AREG,TERM
END
```

---

10. What is the function of the MEND directive?

Answer: `MEND` indicates the **end of a macro definition**. When encountered, Pass-I knows the macro definition is complete.

---

11. How are parameters handled in macros?

Answer: Parameters are written with an `&` prefix (e.g., `&ARG1`). During macro expansion, the **actual argument** replaces the **formal parameter** in the macro body.

---

12. What does the line `MNT.add(new String[]{def[0], String.valueOf(mdtIndex)});` do in Pass-I?

Answer: It adds a new **macro entry** to the Macro Name Table (MNT), storing:

- The macro name (`def[0]`)
- The index (`mdtIndex`) where its definition starts in the MDT.

---

13. What does the `isMacro` boolean flag do in Pass-I?

Answer: It helps identify whether the assembler is currently **inside a macro definition**.

- When `MACRO` is found → `isMacro = true`.
- When `MEND` is found → `isMacro = false`.

---

14. What are the outputs of Pass-I in your program?

Answer:

1. **MNT.txt** — Macro Name Table
2. **MDT.txt** — Macro Definition Table
3. **intermediate.txt** — Intermediate Code
4. A message: *"Pass-I complete. MNT, MDT & Intermediate Code generated."*

---

15. What is the purpose of Pass-II?

Answer: Pass-II reads the intermediate code, detects macro calls, retrieves their definitions from MNT and MDT, and performs **macro expansion** to produce the **final expanded code**.

---

16. How does Pass-II identify a macro call?

Answer: In `Pass2.java`, it checks if the **first token** of each line matches a macro name stored in the **MNT HashMap**:

```
if (MNT.containsKey(t[0]))
```

---

17. What happens when a macro call like `INCR TERM` is found?

Answer:

- The program looks up the macro name in MNT.
- Retrieves the definition from MDT using the stored index.
- Replaces the formal argument (`&ARG`) with the actual argument (`TERM`).
- Writes the expanded instructions to the output.

Result:

```
A 1,TERM
```

---

18. What are the files generated by Pass-II?

Answer:

- **output.txt** — containing the final expanded source program.
- Console message: *"Output written to output.txt"*

---

19. What is the data structure used for MNT and MDT in Pass-II?

Answer:

- **MNT:** `HashMap<String, Integer>` (for fast lookup of macro start index)
- **MDT:** `ArrayList<String>` (for storing macro definitions sequentially)

---

20. What does the `expand()` method in Pass-II do?

Answer: It reads each line of intermediate code, detects macro calls, performs argument substitution, and adds the expanded instructions to the final output list.

---

21. What is stored in the `output.txt` file?

Answer: The **fully expanded source code** after all macro calls have been replaced with their actual instructions.

Example:

```
START
MOV A,10
A 1,TERM
END
```

---

22. How are arguments replaced in macro expansion?

Answer: By using the `replace()` method:

```
MDT.get(i).replace("&ARG", arg)
```

It replaces the formal parameter (&ARG) with the actual argument (e.g., TERM).

---

23. What are the key differences between a macro and a subroutine?

Aspect	Macro	Subroutine
Expansion	Inline code expansion	Control transfers to subroutine
Overhead	No call/return overhead	Requires CALL and RETURN
Memory usage	More (due to repeated expansion)	Less (shared code)
Flexibility	Argument substitution possible	Parameters passed at runtime

---

24. What are the advantages of using macros?

Answer:

- Reduces code repetition
- Improves readability
- Allows parameterized reuse
- Simplifies complex instruction sequences

---

25. What are the limitations of macros?

Answer:

- Increases code size (due to inline expansion)
- Difficult to debug expanded code
- No runtime flexibility — everything is expanded at assembly time

---

26. What is the difference between a macro processor and an assembler?

Feature	Macro Processor	Assembler
Function	Expands macros	Converts assembly to machine code
Input	Assembly with macros	Pure assembly code
Output	Expanded assembly	Object code
Example	Your Pass1 & Pass2 Macro Processor	Two-pass assembler

---

27. What message indicates successful execution of Pass-I and Pass-II?

Pass-I:

```
Pass-I complete. MNT, MDT & Intermediate Code generated.
```

Pass-II:

```
Output written to output.txt
```

---

28. Why is it called a "Two-Pass" Macro Processor?

Answer: Because:

- **Pass-I** handles **macro definition and table generation**, and
- **Pass-II** performs **macro expansion** using those tables.

---

29. How can this macro processor be improved?

Answer:

- Support for **nested macros**
- Handle **multiple arguments** (&ARG1, &ARG2, etc.)
- Include **ALA (Argument List Array)** and **EVTAB (Expansion Variable Table)** for better flexibility
- Add **file-based input/output** instead of hardcoded data

---

30. What are the three main tables used in an advanced macro processor?

Answer:

1. **MNT (Macro Name Table)** – stores macro names and MDT indices
2. **MDT (Macro Definition Table)** – stores macro definitions
3. **ALA (Argument List Array)** – stores formal and actual argument correspondence

---

□ SPOS Viva / Oral Questions — CPU Scheduling Algorithms

1. What is CPU Scheduling?

Answer: CPU Scheduling is the process of **deciding which process** will use the CPU next when multiple processes are ready to execute. It helps in **efficient CPU utilization** and **improves system performance**.

---

2. What are the types of CPU scheduling?

Answer:

1. **Preemptive Scheduling** – CPU can be taken away from a process before completion. *(Example: Round Robin, Preemptive Priority, SRTF)*
2. **Non-Preemptive Scheduling** – Once the CPU is given, the process keeps it until completion. *(Example: FCFS, SJF, Non-Preemptive Priority)*

---

3. Define important scheduling terms.

Term	Meaning
Arrival Time (AT)	Time when process enters ready queue
Burst Time (BT)	Time required by process for execution
Waiting Time (WT)	Time process spends waiting in ready queue
Turnaround Time (TAT)	Time from process arrival to completion (TAT = WT + BT)
Response Time	Time from arrival to first CPU response

---

4. What are the goals of CPU Scheduling?

Answer:

- Maximize **CPU utilization**
- Minimize **waiting time**
- Minimize **turnaround time**
- Minimize **response time**
- Maximize **throughput**

□□ FCFS (First Come First Serve) Scheduling

---

5. What is FCFS scheduling?

Answer: FCFS (First Come First Serve) is the simplest non-preemptive scheduling algorithm where the process that **arrives first** gets the CPU first.

---

6. What is the rule used in FCFS?

Answer: Processes are **executed in order of arrival time** (FIFO order).

---

7. What are the advantages of FCFS?

- Simple to implement.
- Fair — every process gets a turn.

---

8. What are the disadvantages of FCFS?

- **Poor average waiting time.**
- **Convoy effect:** Small processes wait for a large process to complete.

---

9. Formula used in FCFS?

```
Waiting Time (WT) = Start Time - Arrival Time
Turnaround Time (TAT) = WT + Burst Time
```

---

10. Example Output Format (from your code):

```
Process  AT  BT  WT  TAT
P1       0   5   0   5
P2       1   3   4   7
```

Average WT and **TAT** are displayed at the end.

---

11. Why is sorting done by Arrival Time in your FCFS code?

Answer: Because the FCFS rule executes the process **which arrives first**; sorting ensures correct order of execution.

---

12. What is CPU idle time handling in your FCFS program?

Answer: If currentTime < arrival time of next process, then CPU remains idle until the next process arrives.

□□ SJF (Shortest Job First) Scheduling

---

13. What is SJF Scheduling?

Answer: SJF selects the process with the **shortest burst time** next for execution.

---

14. What are the two types of SJF?

1. **Non-Preemptive SJF** – Once process starts, it completes.
2. **Preemptive SJF (SRTF)** – Process with smaller burst time can preempt the current process.

---

15. What is the advantage of SJF?

- Produces **minimum average waiting time** among all algorithms.

---

16. What is the disadvantage of SJF?

- Requires **knowledge of future burst time** (difficult in practice).
- May cause **starvation** for long processes.

---

17. Formula used in SJF?

```
WT[i] = sum of burst times of all previous processes
TAT[i] = WT[i] + BT[i]
```

---

18. Why do you sort processes by burst time in SJF?

Answer: Because in SJF, the CPU is assigned to the process with the **shortest burst time** first.

---

19. Is SJF preemptive or non-preemptive in your code?

Answer: Your code implements **Non-Preemptive SJF** (since once a process starts, it finishes).

---

20. What is starvation in SJF?

Answer: Long processes may **never get CPU** if shorter processes keep arriving — this is known as **starvation**.

□□ Priority Scheduling

---

21. What is Priority Scheduling?

Answer: In this algorithm, each process is assigned a **priority value**. The CPU is allocated to the **highest priority process**.

---

22. What is the rule of Priority Scheduling?

Answer: Processes are executed **in decreasing order of priority value** (higher number = higher priority in your code).

---

23. What is the type of Priority Scheduling in your program?

Answer: Your code performs **Non-Preemptive Priority Scheduling.**

---

24. How does your code handle equal priorities?

Answer: If two processes have the same priority, the one appearing **first in input order** executes first.

---

25. What are advantages of Priority Scheduling?

- Allows **important processes** to execute first.
- Improves system responsiveness for critical tasks.

---

26. What are disadvantages of Priority Scheduling?

- May lead to **starvation** for low-priority processes.
- Requires careful assignment of priority values.

---

27. How can starvation be avoided?

Answer: By using **aging**, which gradually increases priority of waiting processes.

---

28. Output from your Priority Scheduling code:

```
Process BT  WT  TAT  Priority
P1      5   0   5    3
P2      3   5   8    1
```

Then **average WT** and **average TAT** are displayed.

□□ Round Robin Scheduling

---

29. What is Round Robin Scheduling?

Answer: Round Robin (RR) is a **preemptive scheduling algorithm** where each process is given a **fixed time quantum** to execute, then placed back in the ready queue if unfinished.

---

30. What is Time Quantum?

Answer: A small fixed time slice during which a process is allowed to execute. Example: 2 ms, 4 ms, etc.

---

31. What is the main advantage of Round Robin?

- **Fairness:** Every process gets equal CPU share.
- **Good response time** for interactive systems.

---

32. What happens if a process doesn't finish within its time quantum?

Answer: It is **preempted** and moved to the **end of the ready queue**.

---

33. What is shown by your Gantt Chart in the program output?

Answer: The **order and duration** in which each process executes, along with time points. Example:

```
Gantt Chart:
0 | P1 | 2 | P2 | 4 | P3 | 6
```

---

34. What are the limitations of Round Robin Scheduling?

- High **context switching overhead** if quantum is too small.
- Poor average turnaround time if quantum is too large.

---

35. What is an ideal time quantum?

Answer: Should be **large enough** to avoid overhead, but **small enough** for quick response. (Usually between 10–100 milliseconds.)

---

36. Is your Round Robin implementation preemptive or non-preemptive?

Answer: Preemptive, because CPU is forcibly taken after each time quantum.

---

37. What happens if CPU is idle in Round Robin?

Answer: The next available process immediately gets CPU time; idle time is minimal.

---

38. Why is Round Robin suitable for time-sharing systems?

Answer: Because it ensures **equal time allocation** to all users/processes — ideal for interactive or multitasking OS.

□ Comparison Summary Table

Algorithm	Type	Basis of Selection	Preemptive	Starvation	Fairness
FCFS	Non-Preemptive	Arrival Time	No	No	Yes
SJF	Non-Preemptive	Shortest Burst Time	No	Yes	No
Priority	Non-Preemptive	Priority Value	Optional	Yes	No
Round Robin	Preemptive	Time Quantum	Yes	No	Yes

□ Bonus Viva Questions

39. What are the criteria for selecting a good scheduling algorithm?

- Minimum average waiting time
- Minimum turnaround time
- Maximum CPU utilization
- Fairness among processes

---

40. What are real-world examples of these algorithms?

Algorithm	Used In

Algorithm	Used In
FCFS	Batch Systems
SJF	Short jobs in compilers
Priority	Real-time systems
Round Robin	Time-sharing OS like Unix/Linux

Viva Questions and Answers – Page Replacement Algorithms

1. What is a page replacement algorithm?

A page replacement algorithm decides which memory page to replace when a new page needs to be loaded into memory, but all frames are full.

---

2. Why do we need page replacement algorithms?

Because in virtual memory systems, physical memory (RAM) is limited. When a process requests a page that is not in memory (page fault), the operating system must replace one of the existing pages using a replacement algorithm.

---

3. What is a page fault?

A page fault occurs when a program tries to access a page that is not currently in main memory. The OS must bring the required page from secondary storage (disk) to main memory.

---

4. What is a page hit?

A page hit occurs when the requested page is already present in the main memory, so no replacement or disk access is needed.

---

5. What is the difference between a page hit and a page fault?

- **Page Hit:** Requested page found in main memory.
- **Page Fault:** Requested page not found in main memory and must be fetched from secondary storage.

---

6. What is FIFO page replacement algorithm?

FIFO (First-In-First-Out) replaces the oldest page in memory — the one that was loaded first. It uses a simple queue structure to keep track of the order of pages.

---

7. What are the advantages and disadvantages of FIFO?

- **Advantages:** Simple and easy to implement.
- **Disadvantages:** May replace frequently used pages (causes Belady's anomaly).

---

8. What is Belady's Anomaly?

Belady's Anomaly is the situation where increasing the number of page frames results in **more page faults** in FIFO. It does not occur in algorithms like LRU and Optimal.

---

9. How does the FIFO algorithm work?

- Pages are inserted in the order they come.
- When memory is full, the oldest page (first inserted) is removed and replaced by the new one.

---

10. What is the LRU (Least Recently Used) page replacement algorithm?

LRU replaces the page that has not been used for the longest period of time — assuming that pages used recently will likely be used again soon.

---

11. How does LRU differ from FIFO?

- **FIFO:** Based on arrival order.
- **LRU:** Based on usage order (recentness). LRU uses the history of references, while FIFO does not.

---

12. How is LRU implemented?

LRU can be implemented using:

- **Stack or List:** Move recently used pages to the top.
- **Counter or Timestamp:** Keep track of when each page was last used.

---

13. What is the main advantage of LRU over FIFO?

LRU generally gives better performance because it uses the principle of temporal locality — recently used pages are likely to be used again soon.

---

14. What is the Optimal Page Replacement algorithm?

The Optimal algorithm replaces the page that **will not be used for the longest period in the future**. It gives the **minimum number of page faults** possible.

---

15. Why is the Optimal algorithm not practical for real systems?

Because it requires future knowledge of the reference string, which is impossible to predict in real-time systems. It is mainly used for theoretical comparison.

---

16. What is the difference between LRU and Optimal?

- **LRU:** Uses past information (recent usage).
- **Optimal:** Uses future information (next usage). Optimal gives the best results theoretically; LRU is practical.

---

17. Which algorithm gives the least number of page faults?

The **Optimal Page Replacement** algorithm gives the least number of page faults among all algorithms.

---

18. What data structures are used in these algorithms?

- **FIFO:** Queue or circular buffer.
- **LRU:** Stack, list, or map with timestamps.
- **Optimal:** Array with look-ahead checking.

---

19. What is the hit ratio?

The hit ratio is the fraction of memory accesses that result in a page hit. **Formula:** `Hit Ratio = Number of Hits / Total Number of References`

---

20. What is the fault ratio?

The fault ratio is the fraction of memory accesses that result in a page fault. **Formula:** `Fault Ratio = Number of Faults / Total Number of References`

---

21. In your FIFO program, what is the role of the pointer?

The pointer indicates which frame should be replaced next. It moves circularly through all frames (using modulo operation).

---

22. In your LRU program, how is the recently used page tracked?

An ArrayList (stack) is used — whenever a page is used, it is moved to the end (most recently used position). The least recently used page is at the front.

---

23. In your Optimal program, how is the replacement page decided?

For each page currently in memory, the algorithm checks how far in the future it will be used. The page with the farthest next use is replaced.

---

24. What happens when all frames are empty in the beginning?

Initially, all page references will cause page faults until all frames are filled.

---

25. Which algorithm is used in modern operating systems?

Most modern systems use variations of **LRU**, such as **Clock algorithm** or **Enhanced Second Chance**, because they provide a good balance between accuracy and efficiency.

---

26. Which algorithm may cause Belady's anomaly?

The **FIFO algorithm** may cause Belady's anomaly. **LRU** and **Optimal** do not.

---

27. What is the time complexity of these algorithms?

- **FIFO:** $O(n \times f)$
- **LRU:** $O(n \times f)$ (if implemented with list)
- **Optimal:** $O(n \times f^2)$ (because it looks ahead in the reference string)

Where `n` = number of references, `f` = number of frames.

---

28. What is temporal locality in memory management?

Temporal locality means that recently accessed pages are likely to be accessed again soon. LRU takes advantage of this property.

---

29. What is spatial locality?

Spatial locality means that pages near the recently accessed page are likely to be accessed soon. It is used in prefetching techniques.

---

30. Give one example where LRU performs better than FIFO.

If the reference string frequently reuses recently accessed pages, LRU performs better because it retains those pages, while FIFO may remove them too early.
