1. Difference between System and Application Software?

Parameters	System Software	Application Software
Definition	System software is mainly focused on the efficient management of the computer system.	An application program is mainly focused on solving the specific problems using computer as a tool.
Purpose	System programs support the operation rather than any particular application.	The main focus is on the application not on the computing system.
Machine Dependency	System software is machine dependent.	Application software is machine independent.
Programmer Knowledge	System programmer requires knowledge about the internal computer architecture.	Application programmer requires detailed knowledge about high level language which is used to develop an application.
Portability	System program becomes portable using concept of bootstrapping.	Application program becomes portable using concept of cross compiler.
Examples	Examples are OS, compiler, interpreter, assembler etc.	Examples are Microsoft Access, Notepad, Photoshop etc.

2. Compare Compiler and Interpreter?

Comparison of Compiler and Interpreter

UQ. What is the difference between compiler and interpreter? **MU - May 16, 5 Marks**

UQ. Compare compiler and interpreter? **MU - May 18, Dec. 18, May 19, 5 Marks**

Sr. No.	Parameters	Compiler	Interpreter
1.	Input	It is a system program which compiles complete source program at a time.	Interpreter compiles one line at a time.
2.	Intermediate Code	Compiler generates intermediate code.	Interpreter does not generate intermediate code.

Sr. No.	Parameters	Compiler	Interpreter
3.	Memory	As compiler takes complete source code, it needs more memory to store it during compilation.	Interpreter takes single line at a time so it consumes less memory during interpretation.
4.	Compilation	Source code is compiled once and run anytime.	Every time source code is interpreted and then only it can run.
5.	Errors	The errors are displayed after the entire source code is compiled.	Errors are displayed as soon as encountered in source code.
6.	Examples	Example is gcc compiler.	Example is java byte code interpreter.

3. What are the different types of Editors, debuggers, Device Drivers?

7. Device drivers

Definiton : Device driver is a system program used to control number of devices which are attached to computer.

- The main function of device driver is to translate the instruction given by operating system into a form that will be understandable to the particular devices.
- A device driver tells operating system that how the devices will work on certain commands which are generated by user.

8. Operating system

9. Editor

Definition : Editor is a system program which is used to edit the text in the file. The main tasks of editors are editing the text, traversing through the text, viewing and displaying the text, etc.

- There are various types of editors as shown i Fig. 1.2.6.

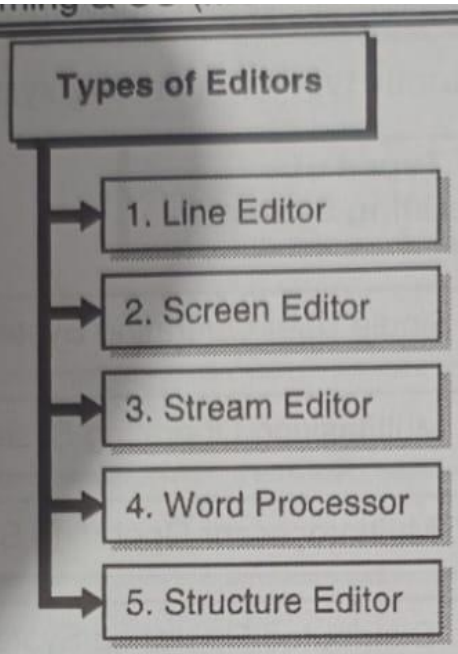


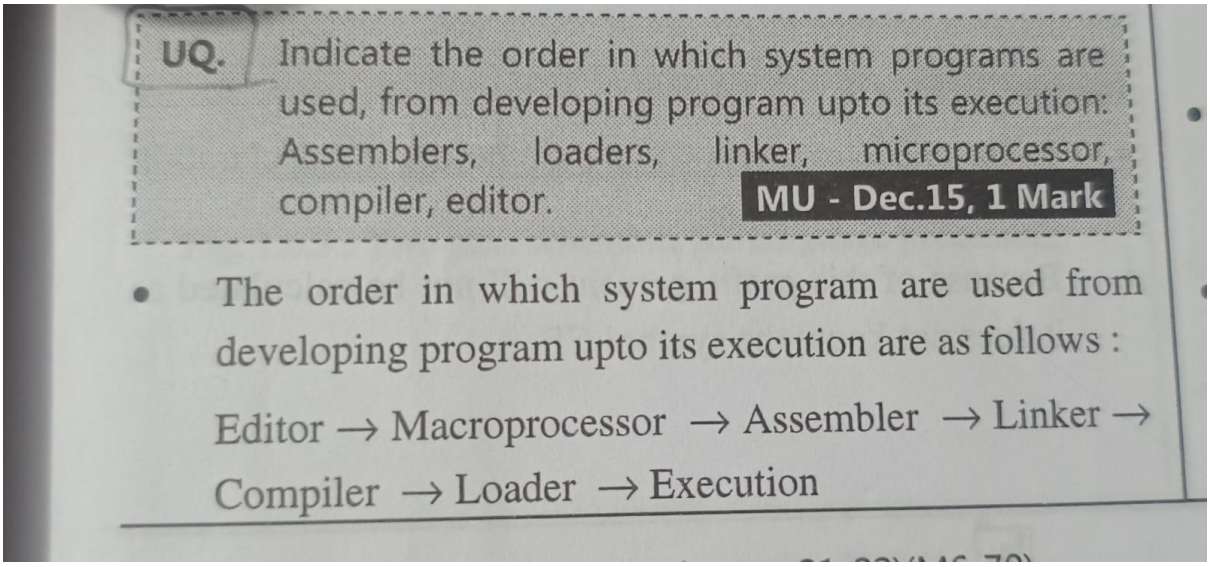
Fig. 1.2.6 : Types of Editors

- Examples of Editors are notepad++, WordPad on windows, Vi and Vim on Linux, Micro-Edit on MS-DOS, etc.

10 Debuggers

Definition : A **debugger** is a computer program used to find out errors which are also called as bugs in source program.

- Debugger provides the facility to halt the program at any certain point and check the changes made in the program. Most debugger runs programs step by step; because of this the programmer can examine states of program changing during the execution.
- Some debuggers allow setting breakpoint in the source program and debugging program up to that breakpoint and examine the changes.



UQ. Indicate the order in which system programs are used, from developing program upto its execution: Assemblers, loaders, linker, microprocessor, compiler, editor. **MU - Dec.15, 1 Mark**

- The order in which system program are used from developing program upto its execution are as follows :

Editor → Macroprocessor → Assembler → Linker → Compiler → Loader → Execution

Note sequence is wrong in book correct sequence as per sir:-

The correct sequence of execution in this case should be:

1. **Editor** – Used to write and edit the source code.
2. **Macro Processor** – Expands macros in the source code (if macros are used).
3. **Compiler** – Translates high-level language source code into assembly or machine code.
4. **Assembler** – Converts assembly language code into machine code (if needed).
5. **Linker** – Combines object files and libraries into a single executable file.
6. **Loader** – Loads the executable file into memory for execution by the CPU.

Explanation:

7 Marks:-

1. Pass 1 of 2 pass assembler

Pass 1 in a Two-Pass Assembler

A **two-pass assembler** processes the source code in two passes. The first pass is responsible for **building the symbol table** and **calculating addresses.** Here's a detailed breakdown of what happens in **Pass 1**:

Main Tasks in Pass 1:

1. **Scanning the Source Code**
   - The assembler reads the source code line by line to identify **labels, opcodes, and operands.**
2. **Assigning Addresses (Location Counter - LC)**
   - The assembler maintains a **Location Counter (LC)** to track memory addresses for each instruction.
   - The LC starts at a specified address (often 0 or a predefined starting point).
   - As each instruction is processed, the LC is updated to reflect the memory location of the next instruction.
3. **Building the Symbol Table**
   - When a **label** is encountered, it is added to the **Symbol Table** along with its corresponding memory address (from the LC).
   - This table is used in **Pass 2** for resolving address references.
4. **Processing Directives (ORG, EQU, etc.)**
   - The assembler processes assembler directives such as:
     - **ORG (Origin)** → Sets the starting address.
     - **EQU (Equate)** → Assigns a constant value to a label.
     - **DS (Define Storage)** → Reserves memory space.
5. **Handling Forward References**
   - If a symbol is used before being defined, it is marked as a **forward reference** and resolved in **Pass 2.**

Example of Pass 1 Execution

Input Assembly Code

assembly Copy Edit

```
START 1000
LOOP    ADD A       ; Add value at A
        SUB B       ; Subtract value at B
        JMP LOOP    ; Jump back to LOOP
A       DW 5        ; Define A with value 5
B       DW 3        ; Define B with value 3
END
```

Pass 1 Processing

Label	Address	Instruction
START	1000	(Starting point)
LOOP	1000	ADD A
	1001	SUB B
	1002	JMP LOOP
A	1003	DW 5
B	1004	DW 3

Symbol Table After Pass 1

Symbol	Address
LOOP	1000
A	1003
B	1004

Pass 1 Output

- **Symbol Table:** Contains labels with their assigned addresses.
- **Intermediate Code:** Stores parsed instructions and their memory locations.
- **Location Counter (LC) Final Value:** 1005 (next available address).

Purpose of Pass 1

- Establishes addresses for **labels and symbols.**
- Prepares the **symbol table** for use in **Pass 2**, where actual machine code is generated.
- Helps resolve **forward references** in the second pass.

Pass 2 will now use this information to replace labels with actual addresses and generate the final machine code. 🚀

Flowchart of Pass-I of Two-Pass Assembler

UQ. Draw a neat flowchart of pass-I of two pass assembler design and explain in detail. **MU - May 19, 10 Marks**

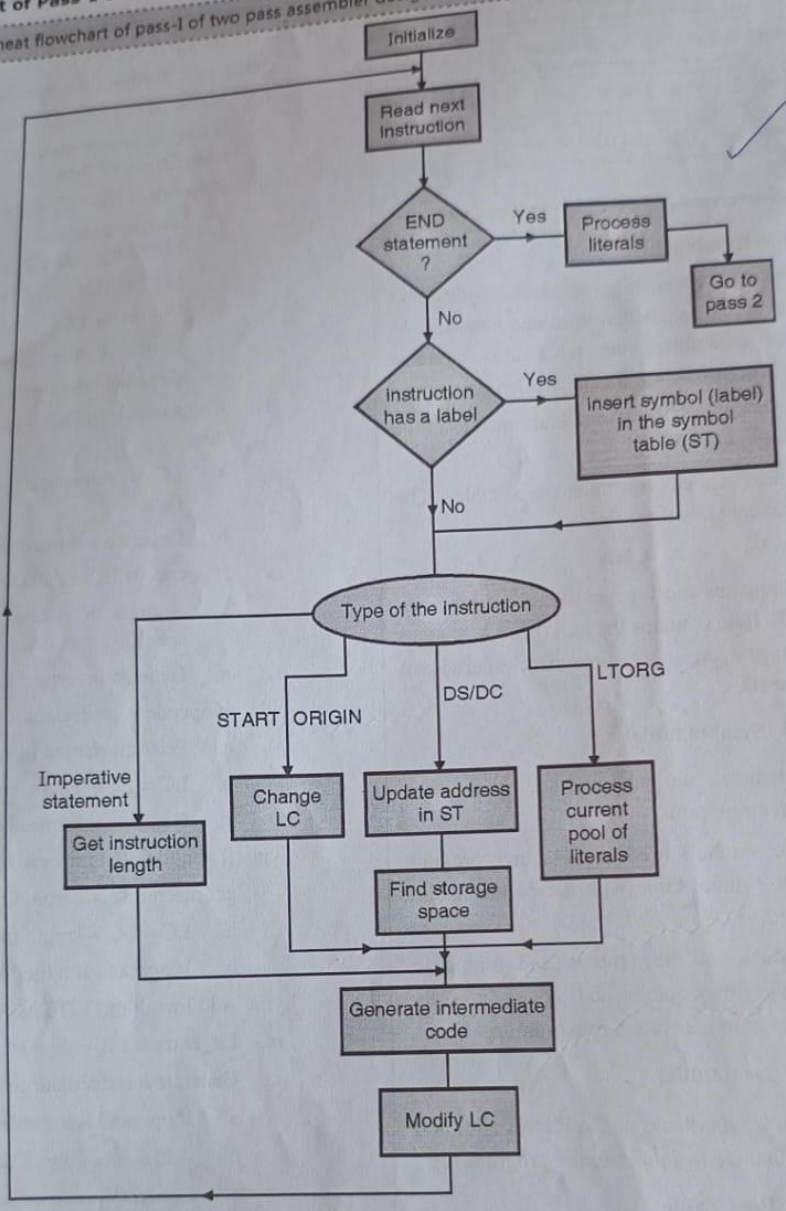


Fig. 2.6.2 : Flowchart of Pass-I of Two-Pass Assembler

2. With reference to assemble explain following tables with suitable examples :POT, MOT, ST, LT ,BT

with reference to assemble explain following tables with suitable examples :POT, MOT ST LT BT

In assembly language, different types of tables are used to store information about instructions, opcodes, and memory addresses. Below are explanations of the tables you mentioned, with suitable examples:

1. POT (Pseudo Opcode Table)

- This table contains pseudo-opcodes (also known as pseudo-instructions or assembler directives) that are used by the assembler for various purposes such as memory allocation, defining constants, or controlling the assembly process.
- These directives are not actual machine-level instructions but are necessary for program assembly.

Example:

pgsql Copy Edit

```
START 1000   ; Start execution at address 1000
BYTE  C'HELLO'  ; Store character constant
WORD  5       ; Store integer constant
RESW  1       ; Reserve one word in memory
END   MAIN    ; Mark end of program
```

POT Table Example:

Pseudo Opcode	Meaning
START	Program start address
END	Marks end of the program
BYTE	Stores character or hexadecimal constant
WORD	Stores integer constant
RESW	Reserves memory words

2. MOT (Mnemonic Opcode Table)

- This table contains machine-level opcodes for corresponding assembly language mnemonics.
- It helps in the translation of assembly language instructions into their machine code equivalents.

Example:

```css
ADD  A, B  ; Add contents of B to A
SUB  A, C  ; Subtract contents of C from A
```

MOT Table Example:

Mnemonic	Opcode	Operand Format
ADD	18	Register, Register
SUB	1A	Register, Register
MOV	89	Register, Memory
MUL	20	Register, Register

---

3. ST (Symbol Table)

- Stores information about symbols (labels, variables, constants) used in the program.
- The assembler maintains this table to resolve memory addresses.

Example:

```pgsql
MAIN  START 1000
LOOP  ADD A, B
END
```

Symbol Table Example:

Symbol	Address	Type
MAIN	1000	Label
LOOP	1003	Label

4. LT (Literal Table)

- Stores literals (constants explicitly written in the code) used in the program.

Example:

```ini
LDA =5  ; Load immediate value 5
```

Literal Table Example:

Literal	Address
=5	1050

---

5. BT (Base Table)

- Used in **base register addressing mode** to keep track of base registers in a program.
- The assembler maintains this table to facilitate address calculations when base-relative addressing is used.

Example:

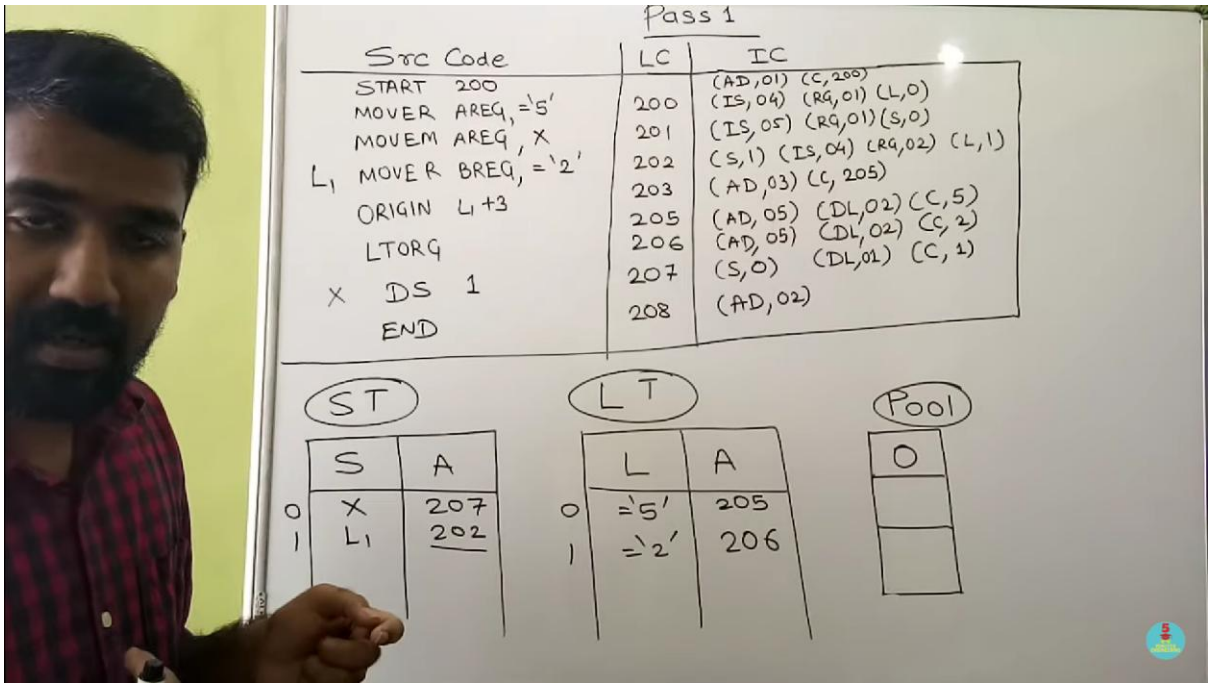
```csharp
USING *,15  ; Use register 15 as the base
```

Base Table Example:

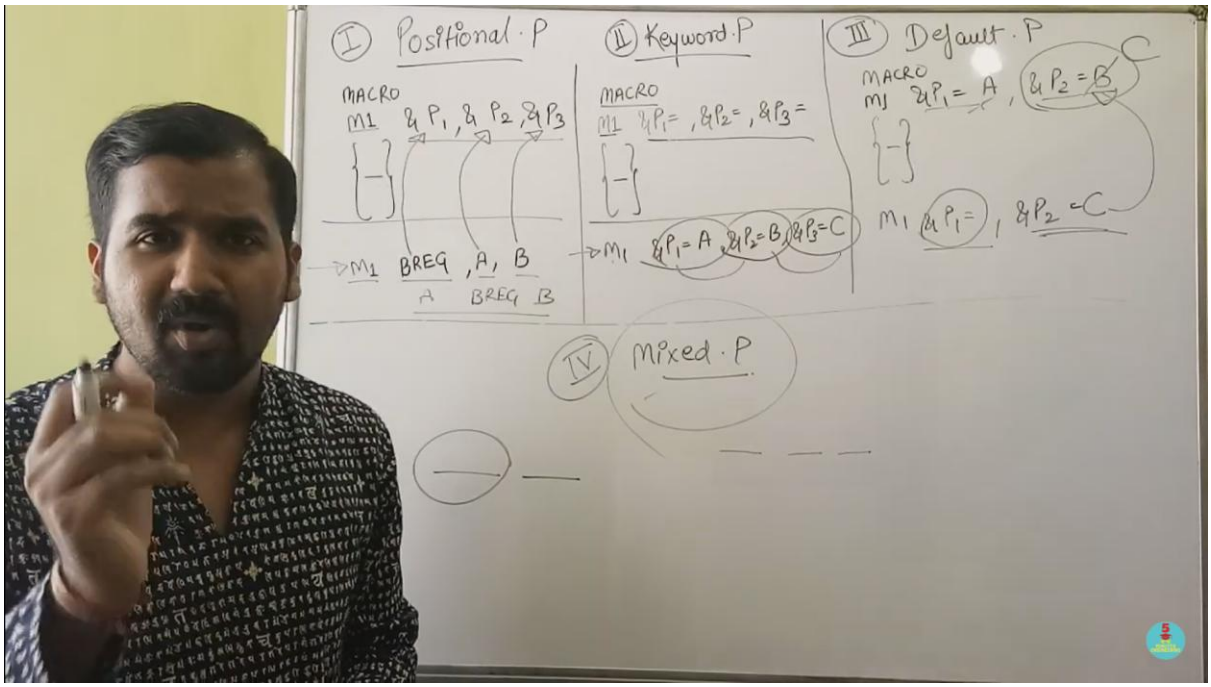
Base Register	Address
15	2000

These tables help in the translation, execution, and management of assembly language programs efficiently.

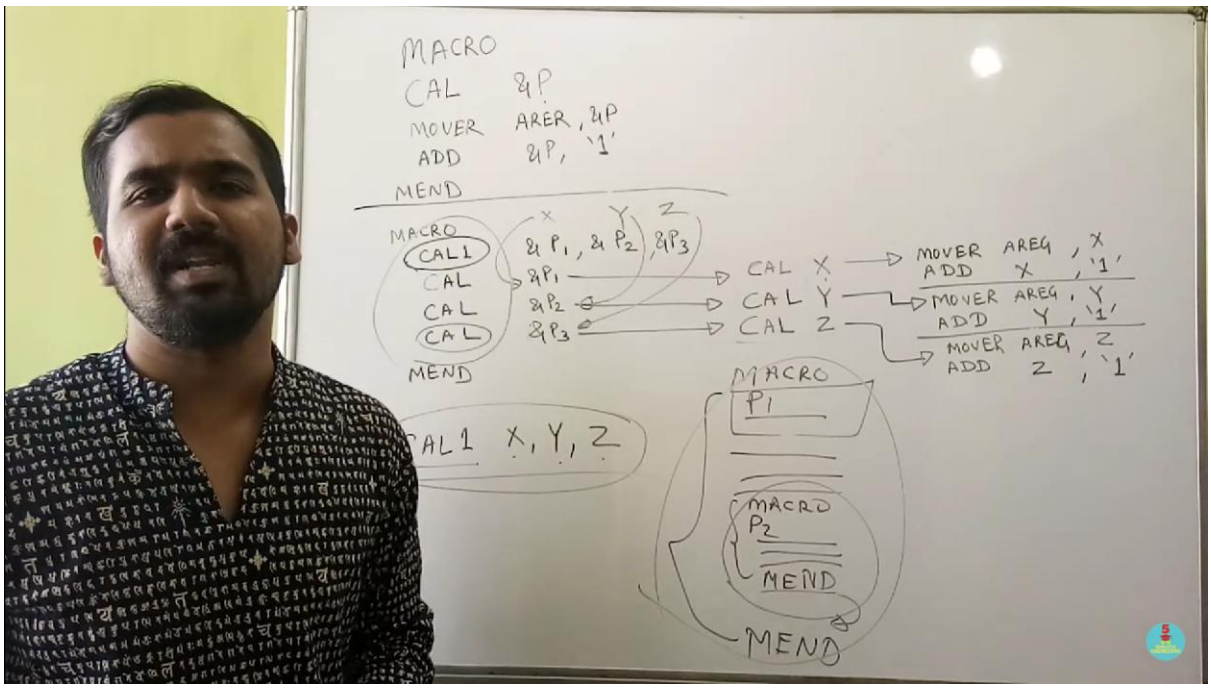
Numerical on Pass 1



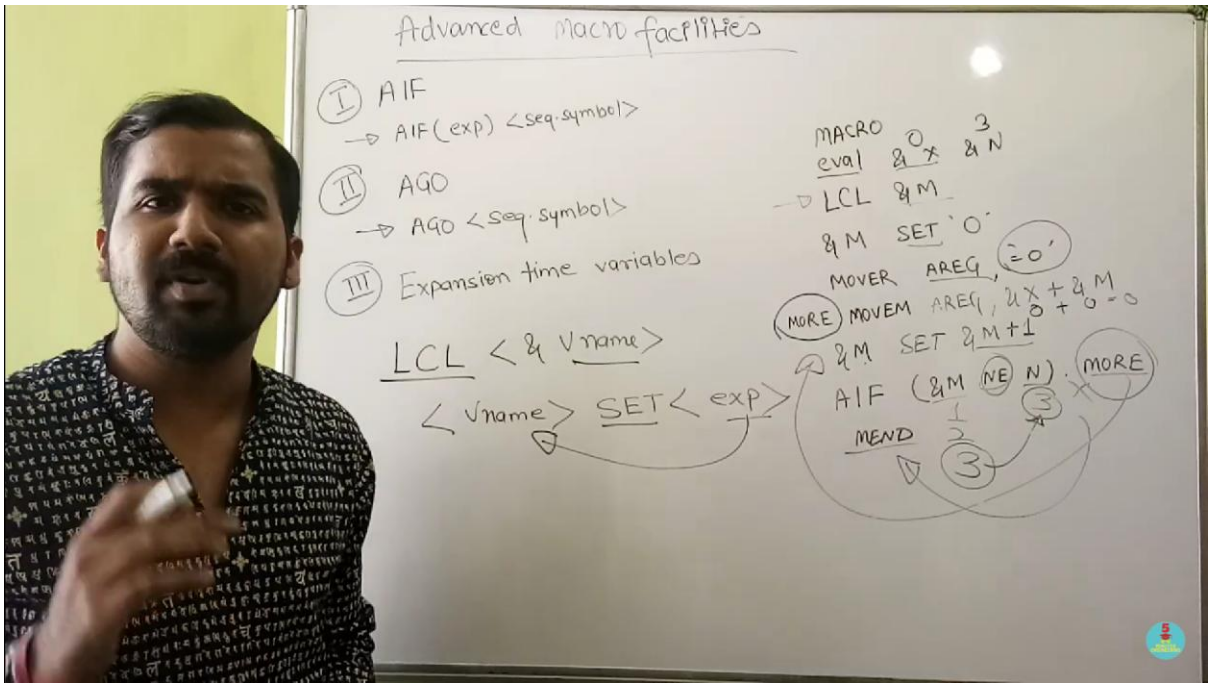
Parameters and types



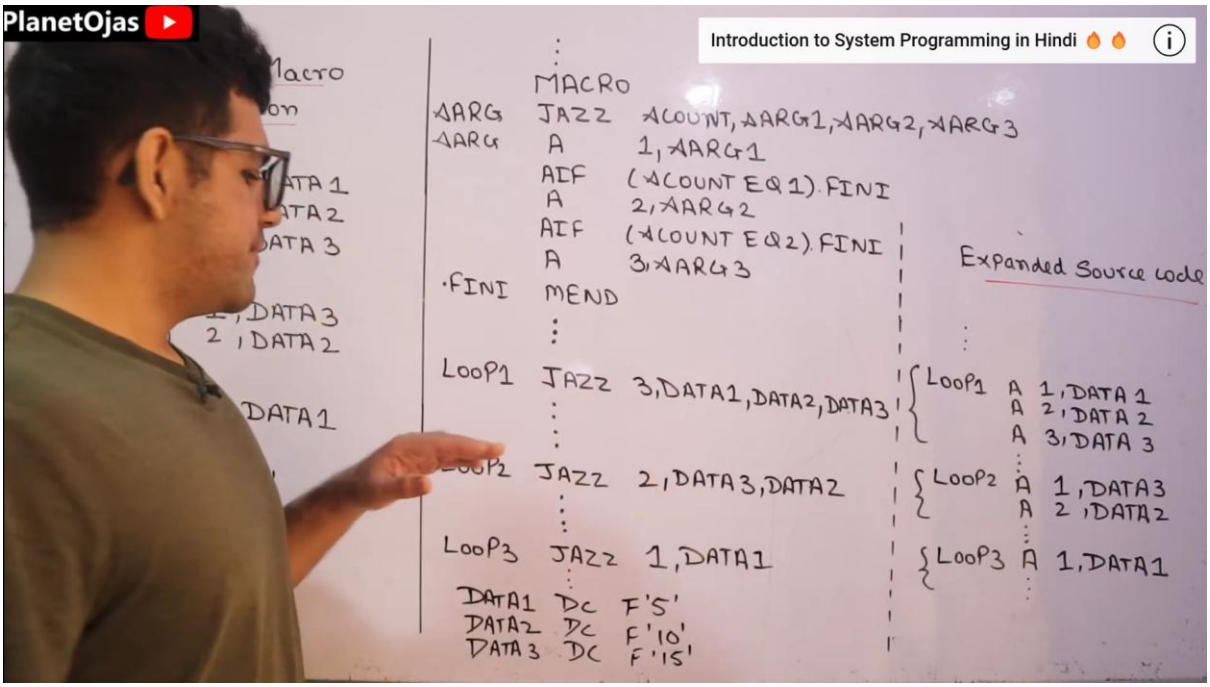
Nested macro

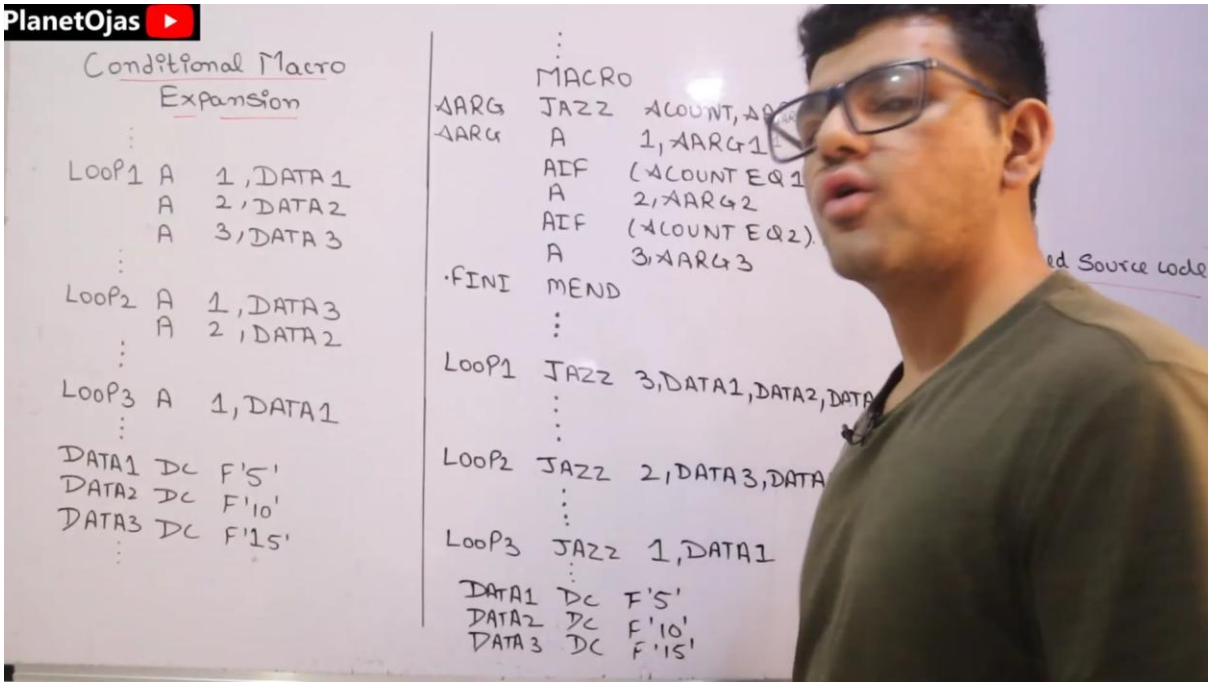


Advanced macro facilites

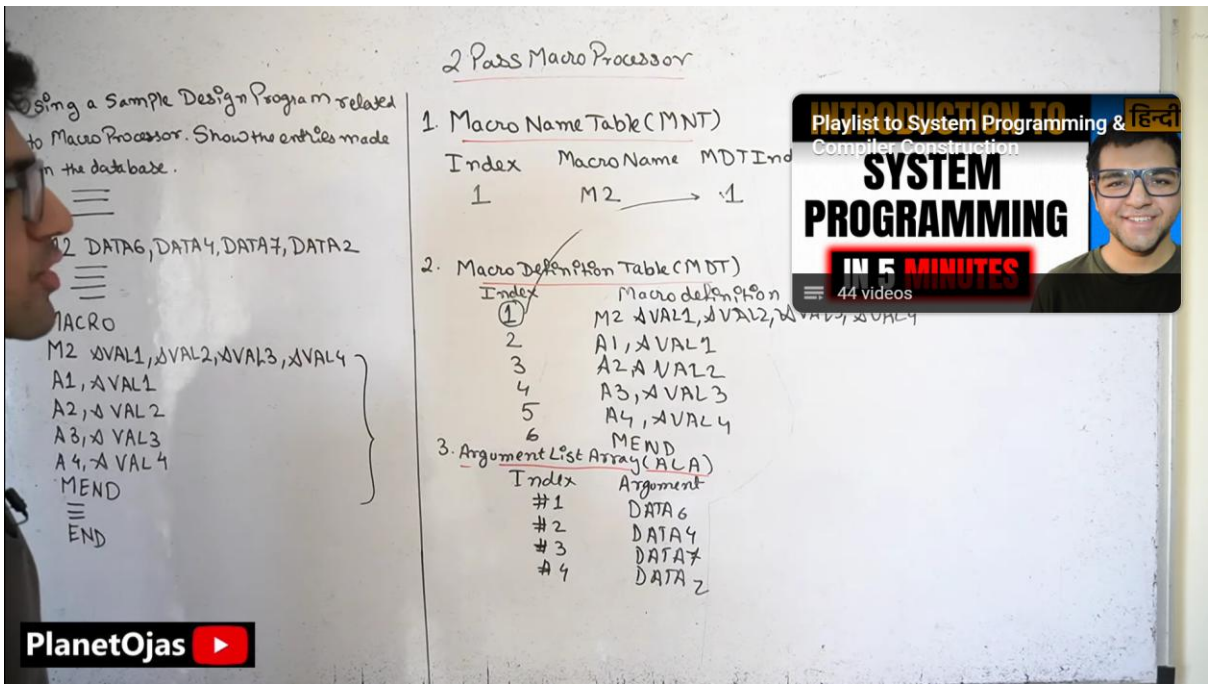


Expansion of macro (conditional)




PlanetOjas
Conditional Macro Expansion
LOOP1 A 1,DATA1
A 2,DATA2
A 3,DATA3
LOOP2 A 1,DATA3
A 2,DATA2
LOOP3 A 1,DATA1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
MACRO
&ARG JAZZ &COUNT,&A
&ARG A 1,&ARG1
AIF (&COUNT EQ 1
A 2,&ARG2
AIF (&COUNT EQ 2)
A 3,&ARG3
.FINI MEND
LOOP1 JAZZ 3,DATA1,DATA2,DATA
LOOP2 JAZZ 2,DATA3,DATA
LOOP3 JAZZ 1,DATA1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
ed Source code

Macro prosessor pass 2:



Explain different features of Marco processors with Suitable example

Features of Macro Processors

A **macro processor** is a system software component that expands macros in an assembly language program before it is processed by the assembler. Macros help programmers write modular and reusable code by defining a set of instructions that can be invoked multiple times with different parameters.

Key Features of Macro Processors

1. Macro Definition and Invocation

- A macro processor allows users to define macros using a specific syntax.
- Once defined, macros can be invoked multiple times in the program.

Example:

assembly Copy Edit

```
MACRO
ADDNUM &X, &Y
   ADD &X, &Y
MEND

START 1000
ADDNUM A, B   ; Macro invocation (expands to ADD A, B)
END
```

2. Macro Expansion

- When a macro is invoked, the macro processor replaces the macro call with the actual instructions defined in the macro.
- This eliminates the need for writing repetitive code manually.

Expansion of the above example:

assembly Copy Edit

```
START 1000
   ADD A, B
END
```

---

3. Parameters and Arguments Support

- Macros can accept parameters, making them flexible and reusable for different operations.

Example:

assembly Copy Edit

```
MACRO
MULTIPLY &A, &B
   MUL &A, &B
MEND

MULTIPLY X, Y   ; Expands to MUL X, Y
MULTIPLY P, Q   ; Expands to MUL P, Q
```

4. Conditional Processing in Macros

- Some macro processors support conditional assembly, allowing macros to have **IF-ELSE** conditions.
- This helps in creating macros that behave differently based on given parameters.

Example:

```assembly
MACRO
COMPARE &X, &Y
IF &X GT &Y
   SUB &X, &Y
ELSE
   SUB &Y, &X
ENDIF
MEND
```

---

5. Recursive Macro Expansion

- Some macro processors allow macros to call other macros, enabling hierarchical code structures.

Example:

```assembly
MACRO
ADD_SQUARE &A, &B
   SQUARE &A   ; Calls another macro SQUARE
   SQUARE &B
   ADD &A, &B
MEND
```

---

6. Iteration in Macros (LOOP Constructs)

- Many macro processors allow iteration using **WHILE** or **REPEAT** constructs.

Example:

assembly Copy Edit

```
MACRO
LOOPPRINT &N
WHILE &N GT 0
   PRINT &N
   DEC &N
ENDW
MEND
```

Advantages of Macro Processors

- **Code Reusability:** Reduces code repetition.
- **Simplifies Coding:** Makes assembly programs more readable.
- **Efficient Execution:** Expands macros at compile-time, reducing runtime overhead.
- **Parameterization:** Allows flexibility in defining operations with different inputs.

Conclusion

Macro processors are essential in assembly language programming, making code more efficient, modular, and easy to manage. By using macros, programmers can reduce manual effort, minimize errors, and optimize assembly code for better maintainability.