Assignment-1

Q.1] Differentiate between application program and system program.

Application Program	System Program
i) The program that helps end user for solving their problem is known as application program.	i) The program that helps in the effective development and execution of application program is called as system Program.
ii) Application software is built for specific tasks.	ii) Low level languages are used to write the system software.
iii) While high level languages are used to write the application software.	iii) While low level languages are used to write the system software.
iv) It is a specific purpose software	iv) It is a general purpose system
v) Without application software, system always run.	v) Without system software, system can't run.
vi) Application software run as per the users requirement.	vi) System software runs when system is turned on and stop when system is off.
vii) Eg: Photoshop, VLC player etc	vii) Operating system, Assembler etc.

Q.2] Write a note on Editor and its type or on text editor.

→ Editors are system tools used to create, modify, and manage text-based files, including program code, configuration files and documents.

Types of Editors:

1) Text Editor: Used for writing and editing plain text files, including configuration files and programming scripts.
Eg: Windows → Notepad, Linux → Nano, MacOS → SimpleText.

2) Code Editor: Specialized editors with features like syntax highlighting, auto-completion and debugging for programming.
Eg: Visual Studio Code, Sublime Text, Atom, Notepad++.

3) Hex Editor: Used for editing raw binary data in files, often for reverse engineering or debugging.
Eg: HxD, Hex Editor Neo, Bless.

4) Word Processor: Designed for creating and editing formatted text documents with rich text features.
Eg: MS Word, Google Docs, Libre Office Writer.

5) Line Editor: Used in command-line environment allowing users to edit text line by line.
Eg. ed (Unix), ex (Vim's line mode)

Q. 3] Compare Compiler and Interpreter.

Compiler	Interpreter
1) A compiler serves the machine language in form of machine code on disks.	1) The Interpreter does not save the machine language.
2) Compiled codes run faster than interpreter.	2) Interpreted code runs slower than compiler.
3) Linking-Loading model is the basic working model of the compiler.	3) The interpretation model is the basic working model of the interpreter.
4) The compiler generates the output in the form of .exe	4) The interpreter does not generate any output.
5) Any change in the source program after the compilation requires recompiling the entire code.	5) Any change in the source code during the translation does not require retranslation of the entire code.
6) Errors are displayed in compiler after compiling together at the current time.	6) Errors are displayed in every single line.

7) Object code is permanently saved for future use.	7) No object code is saved for future use.
8) Used in Production environment	8) Used in Programming & developing environment.

Q4) Write note on software Tools.

⇒ A software tool is a system program that interfaces a program with the entity generating its input data.

- It is a set of computer programs used by developers to create, maintain, debug or support other applications and programs.
- It can be code editors, debuggers, performance analysis tool etc.
- There are some factors that need to be considered before selecting a software tools like usefulness of the tool, integration of one tool with another etc.
- These helps developers to easily maintains the workflow of the project.

Types of software tools:

1) Editors: Editors are system tools used to create, modify, ~~text editor~~ and manage text-based file, including program code, configuration files and documents.

2) Debuggers: A debugger presents compiler-generated errors in a human-readable format and helps in detecting, analyzing and fixing errors (bugs) in software programs. It provides the facility of breakpoints i.e the programmer defines a breakpoint at some specific location that are likely to generate errors the debugger breaks the execution at that point and let the programmer check the value of intermediate variable and function arguments to determine the exact cause of error.

3) Integrated Development Environment (IDE): It is appliation program

that provides tools for software development in a single package. It include - code editor, debugger, compiler / interpreter etc.

4) Profilers : A profiler is a system tool for analyzing time & space complexity of algorithm that analyzes a program's performance by measuring CPU usage, memory consumption and execution time.

5) Project Managers : These tools assits developers in organizing, tracking and managing software development projects. It includes - Task management, version control integration, collaboration tools.

Q.3) Discuss with example forward reference and how it is Handled in assembler design.

⇒ A forward reference problem in an assembler occurs when a label is used in an instruction before it's defined. This means that the assembler doesn't knows the address of the label until it reads its definition. Assembler job is to replace these symbols with the addresses and typically process code sequentially. Encountering an undefined symbol can cause errors or incomplete code generation.

Eg:
```
Start 0
    L  1, FIVE      ← called first
FIVE DC F'4'        ← defined later
```

Solutions of forward reference problem:
There are 2 ways to solve the forward reference problem

1) Multi-pass Approach : Assembler that assembles the code in multiple pass are called multipass assembler. Most commonly used is 2 pass assembler.

2) Single pass Approach : Assembler that assembles the code in single pass are called single pass assembler.

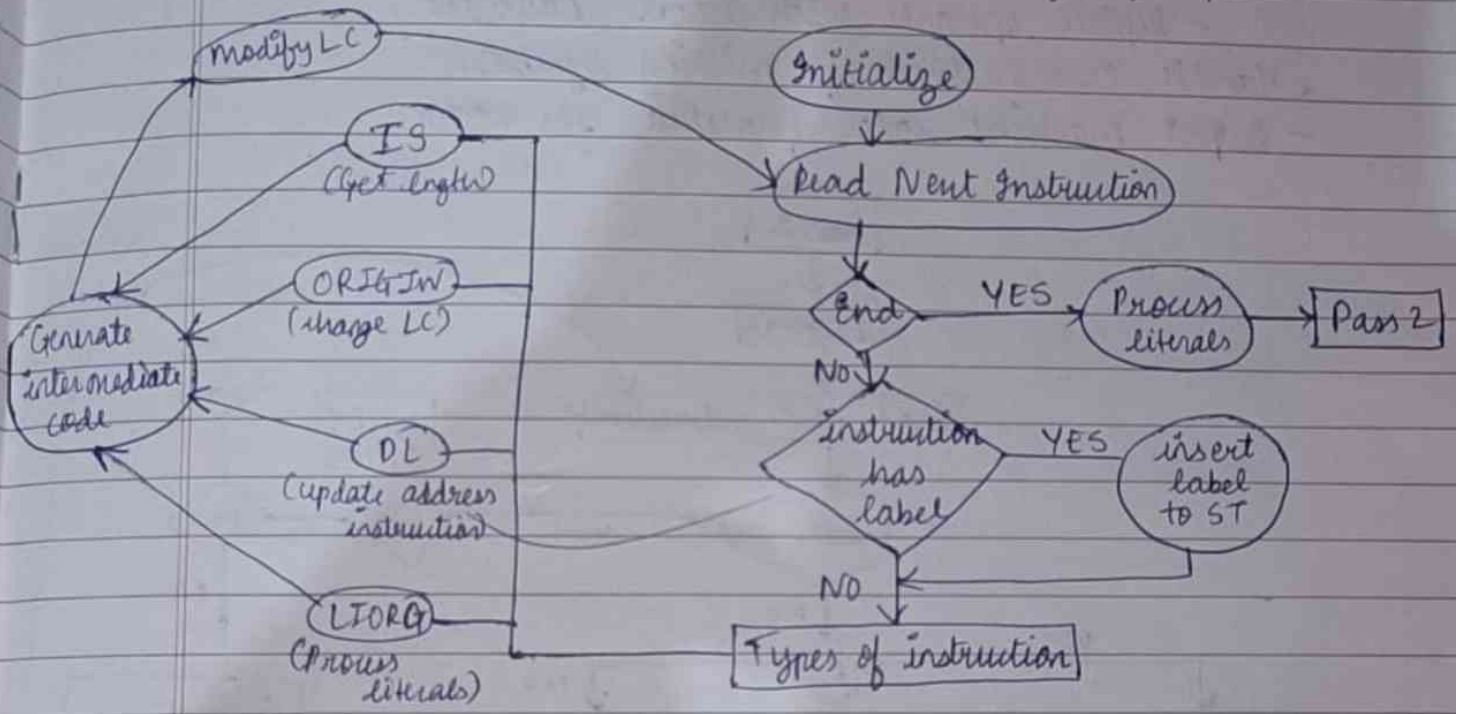
Q.6) Explain design of 2-pass assembler with flowcharts and databases.
⇒ Assembler is a program that is used to convert the instruction written in Low level assembly code to relocatable machinery code. It is necessary to convert the user written language into the machine code. This is called a translation of high-level language to a low level language. This type of translation is performed by a system software. An assembler can be defined a program that translates an assembly language into the machine language.

Working of 2-pass Assembler: 2-pass Assembler divides the task in 2 passes.

* Data structures of Pass1:
- Source code: Assembly program as input.
- Location counter: Tracks memory address for instruction and data.
- Symbol table: Stores label names with corresponding address.
- Literal table: Stores literals (constant value) and design address at the end of the pass1.
- Machine Opcode Table (MOT): Contains Mnemonics, machine opcode, instruction format and length.
- Pseudo Opcode Table (POT): Contains assembler directives and related processing information.
- Intermediate Code file: Store partially processed code with symbol reference.

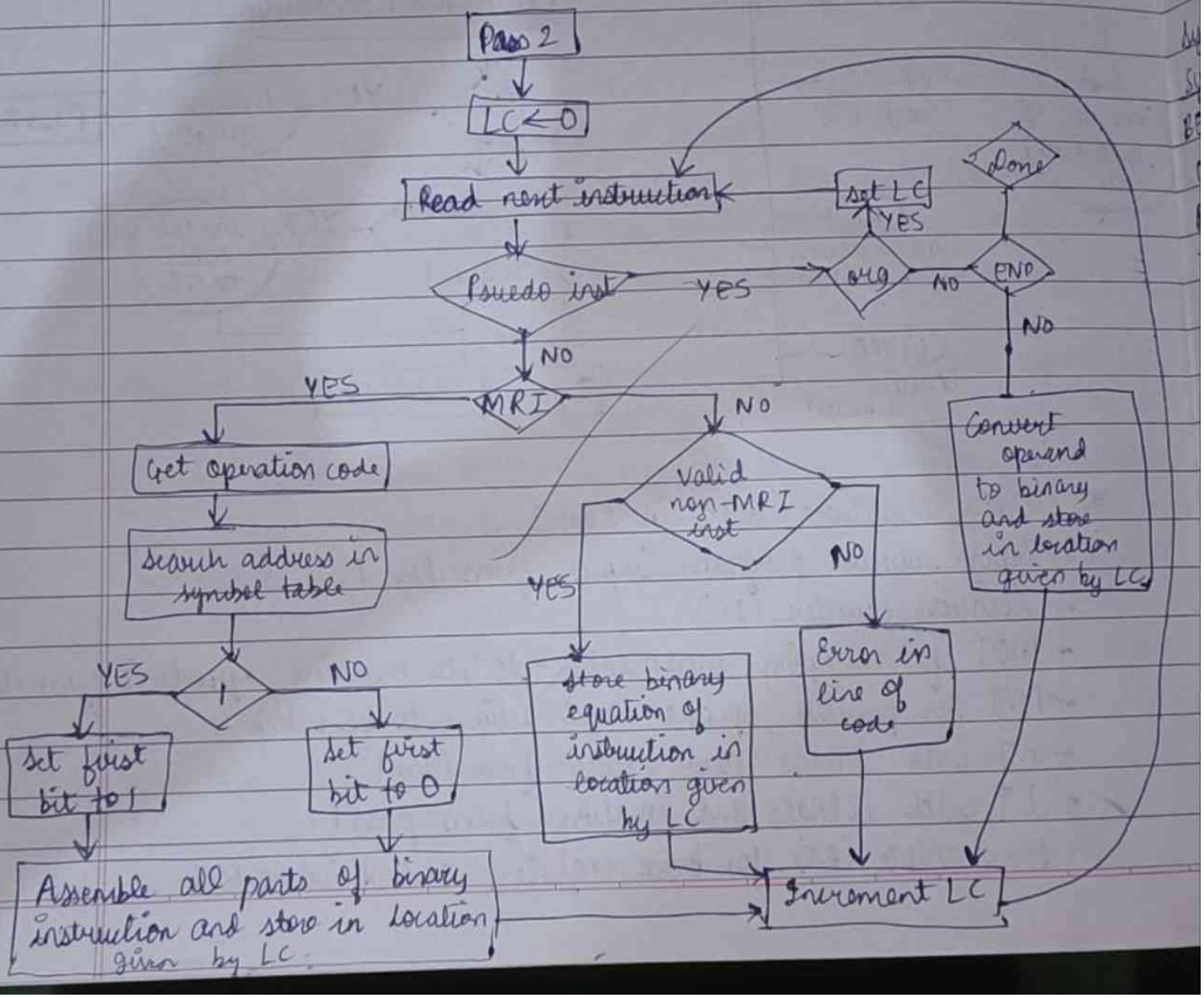
* Design flow of Pass1:
- Initialize the location counter to the start address.
- Read each line of source code.
- Differentiate the line into label, opcode and operand.

- Check for label: Add to Symbol Table with correspond address of location counter
- Search opcode in MOT and POT:
  - if found in MOT, get instructions length and update LC
  - if found in POT, handle directives like LTORG etc.
- Handle literal: Add literal to L.T, assigning address at the end of pass1
- Generate intermediate code: Output each line with placeholder address.
- Write ST, LT and intermediate code to file for pass 2.



* Data structure used in Pass 2
  - Copy source program input from Pass 1
  - Location Counter (LC)
  - MOT for mapping mnemonics to its machine opcode & format
  - POT for pseudo opcode and actions to be taken.
  - ST with labels and values from Pass 1.
  - LT with literals and locations from pass 1.
  - Base Table (BT) for Base register and their content.

* Design flow of Pass 2.
  - Load ST, LI and intermediate code.
  - Read each line of intermediate code: By initializing LC.
  - Check Opcode:
    - if machine instruction: Replace symbols with simple table values and literals with LT addresses.
    - if pseudo opcode: Handle pseudo opcodes like USING (set base register), DROP. (release register) etc.
  - Generate machine code:
    - Convert mnemonic to binary using MOT.
    - Replace operands with actual addresses
  - Handle Errors: Check for missing symbols.
  - Output machine code formatted for loader.



Q7) Generate Pass I and Pass 2 assembler and show the content of databases involved in it.

			LC	
	START	0	0	
BEGIN	BALR	15,0	0	2
	USING	*,15		2
	L	3,=F'34'	2	
	A	3,OLDOH	6	
	S	3,RECPT	10	
	ST	3,ISSUE	14	
OLDOH	DC	F'9'	20	18
RECPT	DC	F'4'	24	24
ISSUE	DS	1F	28	28
	END			32

Symbol table

Symbol	Value	R/A
BEGIN	0	R
OLDOH	20	R
RECPT	24	R
ISSUE	28	R

Literal Table

literal	Value	R/A
=F'34'	32	R

Base Table

Content	Register
2	15

Machine code

Relative location counter	Content/Mnemonics
0	BALR BEGIN 0,(15,0)
2	L 3, 30(15,0)
6	A 3, 18 (15,0)
10	S 3, 22(15,0)
14	ST 3, 26 (15,0)
20	9
24	4
28	F

P.8) Explain design of single pass assembler with flowchart and databases.

→ A single pass assembler scans the program only once and creates the equivalent binary program. The assembler substitutes all of the symbolic instruction with machine code in one pass.

Single pass assembler are used when:
- It is necessary to avoid a second pass over the source program
- The external storage for the intermediate file between two passes is slow.

Main problem
- Forward references:
Rules for an assembly program states that the symbol should be defined somewhere in the program. But in some cases a symbol may be used prior to its definition. Such a reference is called forward reference.

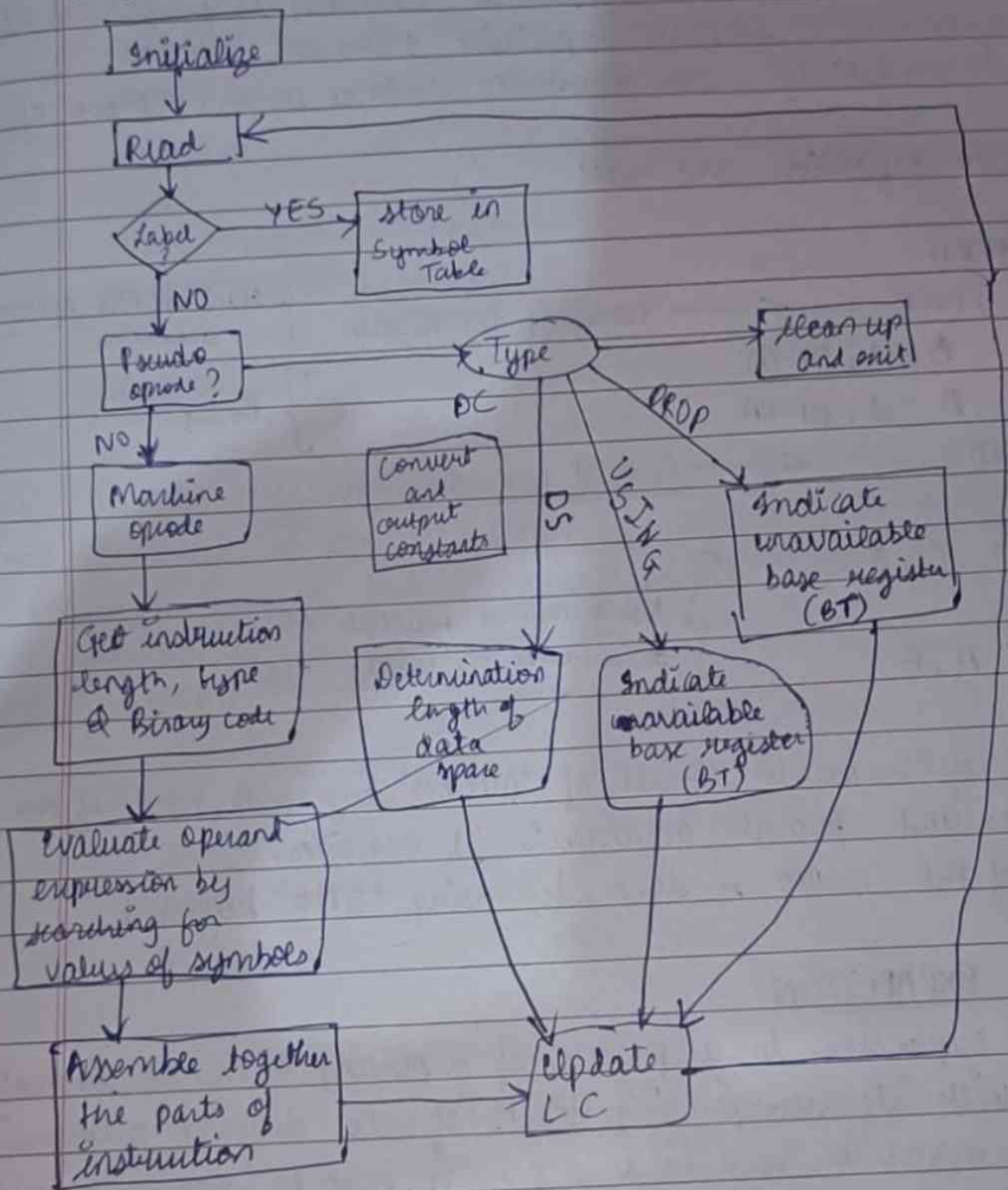
Eg:
```
     START  0
     L  1, FIVE      ← called first
FIVE DC   F '4'      ← defined later
```

- Solution of forward reference:
1) Omit the address translation
2) Insert the symbol into SYMTAB, and mark this symbol undefined.
3) The address that refers to the undefined symbol is added to a list of forward reference.
4) When the definition for a symbol is encountered, the proper address for the symbol is then inserted

- Solutions for single pass assembler
1) Eliminating forward references:
Either all labels used in forward references are defined in the source program before they are referenced, or forward reference

to data items are prohibited.

2) Generating the object code in memory:
No object program is written out and no loader is needed. The program needs to be re-assembled every time.



Q9) Explain Macro and Macro Expansion with example.

⇒ MACRO:

A macro is a sequence of instructions, statements or code that is defined once and can be used multiple times within a program. It is essentially a name given to a block of code or a single line abbreviation of a small sequence of statement that can be expanded wherever required.
Macros are commonly used in assembly language and HLL programming to simplify repetitive tasks and improve
- code readability, code reusability, reduces error, improve efficiency

* Macro definition and call:

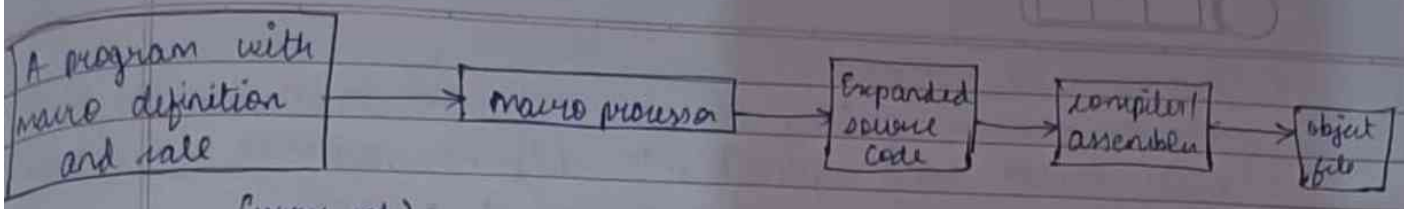
```
MACRO                          } start of the macro definition
   INCR   <-- Name of the macro
     A  1, DATA                } Body
     A  2, DATA
MEND   <-- End of the definition

INCR  A, B
.
.
INCR  A, B
```

} Using macro multiple time it is called as macro call

- Header of the macro consist of 'MACRO' keyword, name of the macros and formal arguments it requires.
- End of the macro is done by using 'END' keyword.

MACRO EXPANSION:
Macro expansion is a process of replacing a macro invocation (call) with its corresponding block of code during preprocessing. When a macro is used in a program, macro processor replaces the macro name with its actual definition before the program is compiled or assembled.



(Source code)

Eg: MACRO
Add1
LOAD A
ADD B
STORE B
MEND
.
.
Add1
.
.
Add1

(expanded code)

LOAD A
ADD B
STORE B

LOAD A
ADD B
STORE B

Q.10) Explain different features of macro with suitable example.

→ Features of Macro Facility:

1) Simple macro processor: In a simple macro processor each macro processor is expanded with its code.

Eg: MACRO //macro defined
TEST
L1, R1
A1, R1
ST 1, SWAP
MEND
.
.
TEST
.
.
TEST
.
.
TEST

L 1, R1
A1, R1
ST1, SWAP

L1, R1
A1, R1
ST 1, SWAP

L1, R1
A 1, R1
ST 1, SWAP

2) Parameterised macro:
- Macros provides facility to pass parameters to it resulting in same macro being expanded into different sequences of instruction. There are two types of parameterised macro:

i) formal parameter:
They are the ones that appears in the macro header and are proceeded by '&' sign. Parameter names are local symbols, which are known within the macro only.

ii) Actual parameter:
They are the ones that appears in the macro call i.e the value /variable actually passed to the macros on which sequence of instructions will be performed.

Eg:
```
MACRO
Add 2 &arg1, &arg2        <- formal parameter
LOAD &arg1
ADD &arg2
STORE &arg1
MEND
:
Add 2 ABC, DEF            <- Actual parameter
```

3) Nested Macro:
There are two ways of using macro facility:

i) Nested macro call: A model statement in a macro may constitute a call on another macro such macros are known as nested macro calls.
- A macro containing the nested call is known as outer macro

Eg:
```
MACRO
    ADD2 &arg1, &arg2
    ADD &arg2
    STORE &arg1
MEND
MACRO
    ADD3 &arg1, &arg2, &arg3
    LOAD &arg1
    ADD2 &arg1, &arg2   ← Call to inner macro
    STORE &arg3
MEND
    :
ADD3 A, B, C   ← Call to outer macro
```

inner macro

outer macro

Expanded code

```
LOAD A
ADD A
STORE A
STORE C
```

ii) **Nested macro definition:**

- Nested macro definition means a macro is defined within the another macro. The nested macros definitions aren't valid until the outer macro has been expanded.

Eg:
```
MACRO
    ADD1 &arg1
    MACRO
        ADD2 &arg2
        A 1, &arg1
        ST 1, &arg1
    MEND
MEND
```

4) **Conditional macro expansion:**

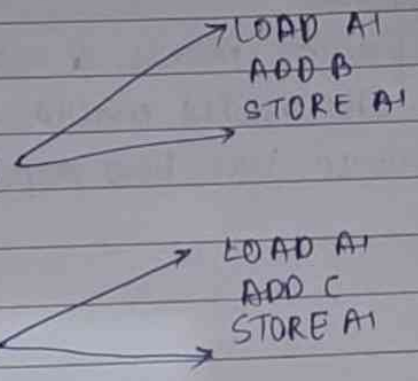
This means based on evaluation of certain condition different sequences of instructions can be inserted into the macro body using the same macro definition.

- Pseudo opcodes that are used for conditional macro expansion:

i) AIF: Used to evaluate a condition and depending on its result, the flow of control inside the macro body is changed.

ii) AGO: It is an unconditional branching statement for the macro processor.

Eg:
```
MACRO
   C_ADD &arg1, &arg2, &arg3
   LOAD &arg1
   AIF (&arg1 > 10). down2
   ADD &arg3
   AGO . down
down2  ADD &arg2
   STORE &arg1
MEND
START 0
MOV A1, 11
C_ADD A1, B, C
   :
MOV A1, 09
C_ADD A1, B, C
```



5) Recursive Macro:
Recursive macros are special cases of nested macro calls; where we make a call from within a macro itself, it uses stack for maintaining status of macro calls.

Eg:
```
MACRO
   FACTORIAL N
   IF N GT 1        ← condition to terminate loop
      M = N-1
   FACTORIAL M      } Recursive loop
      MUL N
   ENDIF
MEND
```

call to itself

START
MOV A,1
FACTORIAL 5 // calculate factorial of 5

Q.11] Draw flowchart and explain the two pass macro processor with its databases.

→ Step 1: Recognizing Macro Definition :- (Pass 1)

↳ Design part - Identifying macro names by scanning the source code before any macro calls are expanded.

↳ Data structure: MNT - stores macro names with index to their definition in the MDT.

Step 2: Storing Macro Definition :- (Pass 1)

↳ Design and data structure: MDT contains the entire macro definition. It stores the definition in a format suitable for expansion, omitting comment lines and converting parameters references into a positional notation for efficiency.

Step 3: Recognizing Macro calls :- (Pass 2)

↳ Design part: Scans the program again, identifying macro calls.

↳ Data structure: MNT: checks in MNT to determine if an instructure is a macro call.
MDT: If it a macro call, retrieves the macro definition from MDT

Step 4: Expanding macro calls and substituting arguments (pass 2)

↳ Design part: Expands the macro by replacing positional notation with actual arguments.

↳ Data structure: ALA: stores the actual arguments passed to the

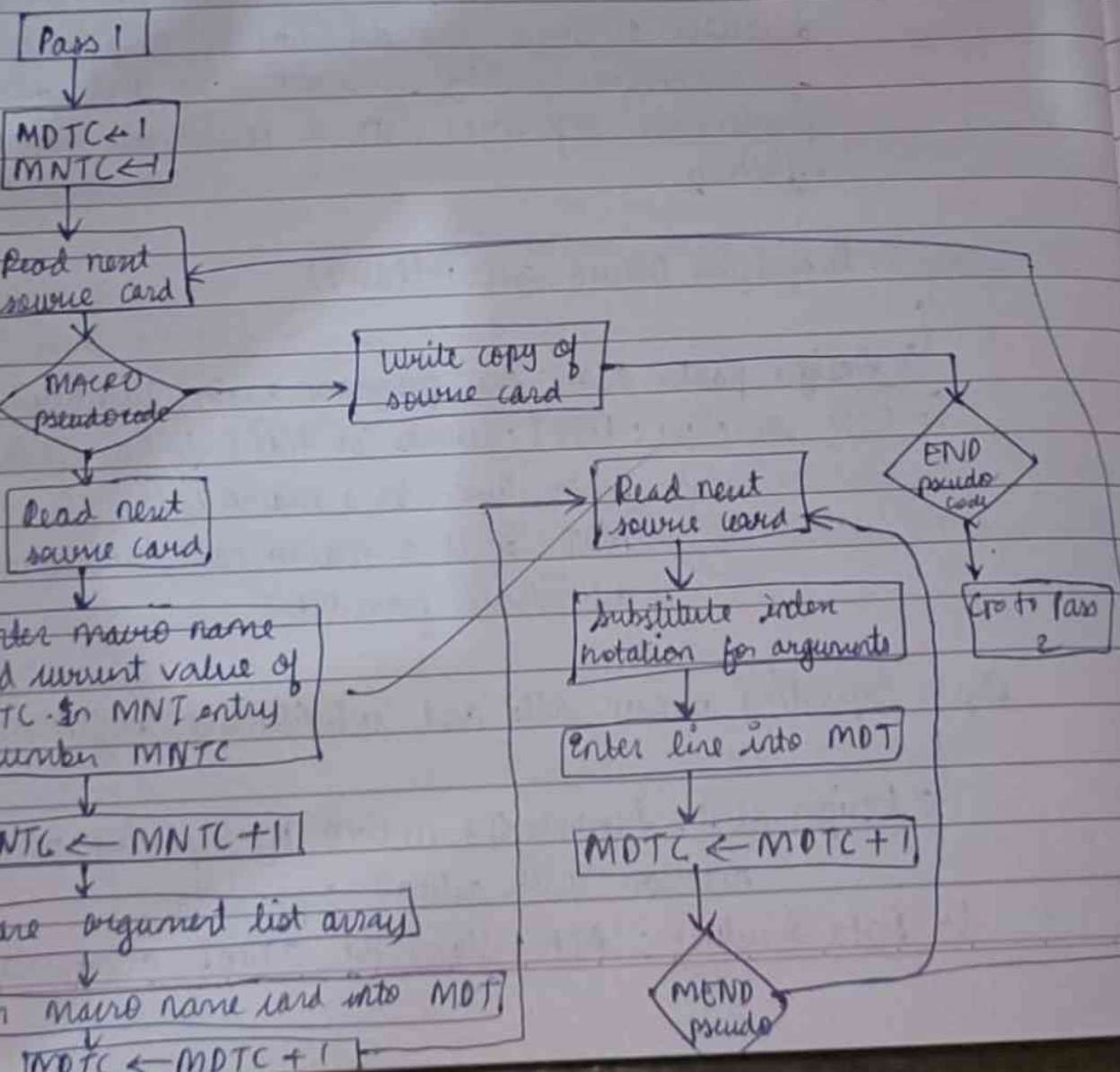
macro in ALP with positional notation.

Step 5: Processing Macro Expansion: (Pass 2)

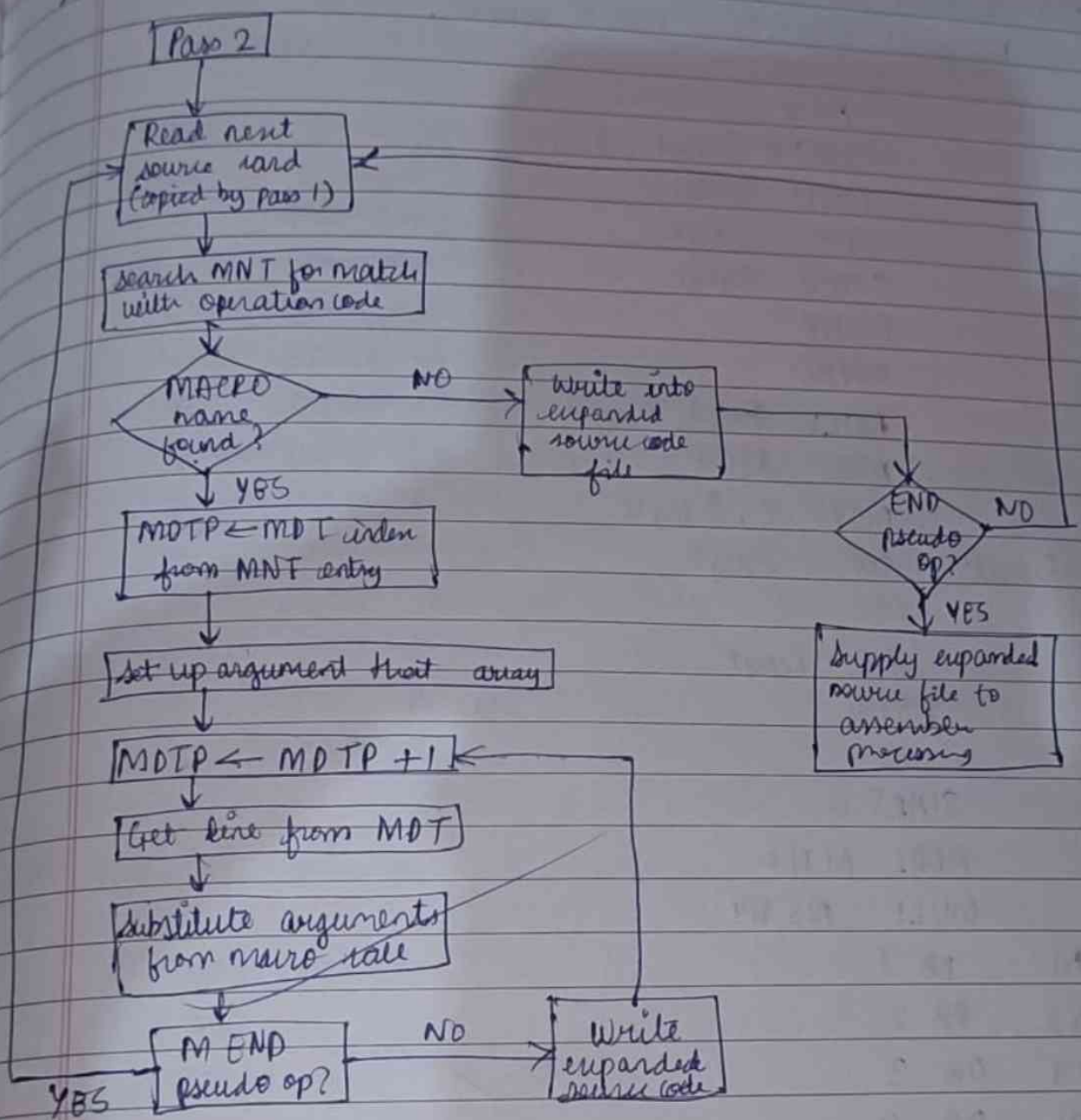
↳ Design part: Expand the macro by replacing positional replacing body line by line parameters and handling nested macros.

↳ Data structure: MDT:- Reads macro body from MDT.
- Replace placeholders with actual arguments from ALP
- Processes expansion control statements like AIF & GO.

* Pass 1 - FLOW CHART



* Pass 2 - Flowchart



Q12) Construct the necessary data structures after compiling the above code by pass I of two pass macro processor.

```
        MACRO
        ADDI  &arg1, &arg2
        LOAD  &arg1
        ADD   &arg2
        STORE &arg1
        MEND
        MACRO
        MULI  &arg3, &arg4
        MOV   A, 00
        MOV   C, &arg4
Repeat  ADD   &arg3
        DEC   C
        JNC   Repeat
        MEND

        START 0
        ADDI  N1, N2
        MULI  N3, N4
N1      DB 1
N2      DB 2
N3      DB 3
N4      DB 4
        END
```

→ MNT:

Index	Macro name	MDT index
1	ADDI	1
2	MULI	6

→ ALA (definition):

ADD1: &arg1 ⇒ #1
&arg2 ⇒ #2

MUL1: &arg3 ⇒ #1
&arg4 ⇒ #2

→ MDT:

Index	statement
1	ADD1 &arg1, &arg2
2	LOAD #1
3	ADD #2
4	STORE #1
5	MEND
6	MUL1 &arg3, &arg4
7	MOV A, 00
8	MOV C, #2
9	Repeat ADD #1
10	DEC C
11	JNC Repeat
12	MEND

⇒ ALA (call):

ADD1:

Positional Parameter	Formal Parameter
#1	N1
#2	N2

MUL1:

Positional Parameter	Formal Parameter
#1	N3
#2	N4

→ Expanded code

```
START 0
LOAD N1
ADD N2
STORE N1
MOV A,00
MOV C,N4
Repeat ADD N3
DEC C
JNC Repeat
N1 DB 1
N2 DB 2
N3 DB 3
N4 DB 4
END
```