## GROUP - A

### EXPERIMENT NO : 01

##### Title: 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-I (intermediate code file and symbol table) should be input for Pass-II.

## PART – 1

##### Title:

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives.

##### Objectives :

* + To understand Data structure of Pass-1 assembler
  + To understand Pass-1 assembler concept
  + To understand Advanced Assembler Directives

##### Problem Statement :

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature.

##### Outcomes:

After completion of this assignment students will be able to:

* + Implemented Pass – 1 assebmler
  + Implemented Symbol table, Literal table & Pool table.
  + Understood concept Advanced Assembler Directive.

##### Software Requirements:

Latest jdk., Eclipse

##### Hardware Requirement:

* + M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

1. **Theory Concepts:**

Introduction :-

There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*. *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader



**Figure 1**. Executable program generation from an assembly source code

##### Advantages of coding in assembly language are:

* Provides more control over handling particular hardware components
* May generate smaller, more compact executable modules
* Often results in faster execution



##### Disadvantages:

* Not portable
* More complex
* Requires understanding of hardware details (interfaces)

##### Pass – 1 Assembler:

An assembler does the following:

1. Generate machine instructions
   * evaluate the mnemonics to produce their machine code
   * evaluate the symbols, literals, addresses to produce their equivalent machine addresses
   * convert the data constants into their machine representations
2. Process pseudo operations

##### Pass – 2 Assembler:

A two-pass assembler performs two sequential scans over the source code:

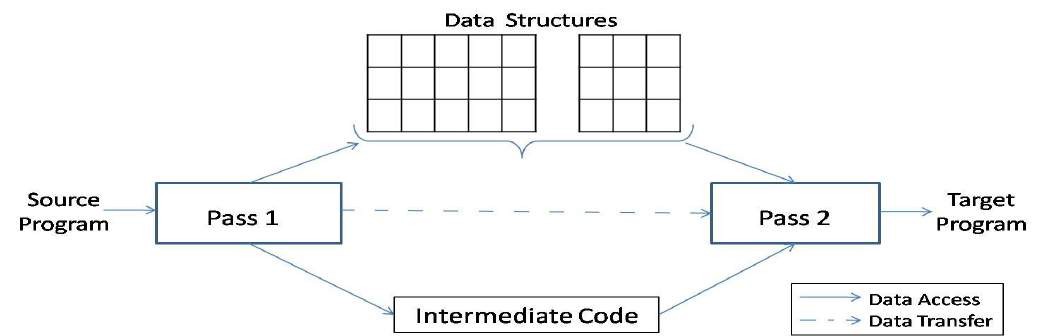
*Pass 1*: symbols and literals are defined

*Pass 2*: object program is generated

*Parsing*: moving in program lines to pull out op-codes and operands

##### Data Structures:

* **Location counter (LC):** points to the next location where the code will be placed
* **Op-code translation table**: contains symbolic instructions, their lengths and their op- codes (or subroutine to use for translation)
* **Symbol table (ST):** contains labels and their values
* **String storage buffer (SSB):** contains ASCII characters for the strings
* **Forward references table (FRT):** contains pointer to the string in SSB and offset where its value will be inserted in the object code



**Figure 2. *A simple two pass assembler***

##### Elements of Assembly Language :

An assembly language programming provides three basic features which simplify programming when compared to machine language.

##### Mnemonic Operation Codes :

Mnemonic operation code / Mnemonic Opcodes for machine instruction eliminates the need to memorize numeric operation codes. It enables assembler to provide helpful error diagnostics. Such as indication of misspelt operation codes.

##### Symbolic Operands :

Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembeler performes memory bindinding to these names; the programmer need not know any details of the memory bindings performed by the assembler.

##### Data declarations :

Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example -5 into (11111010)2 or 10.5 into (41A80000)16

##### Statement format :

An assembly language statement has the following format :

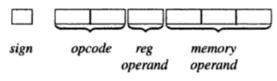
[ Label] <Opcode> <operand Spec> [, operand Spec> ..]

Where the notation [..] indicates that the enclosed specification is optional.

Label associated as a symbolic name with the memory word(s) generated for the statement

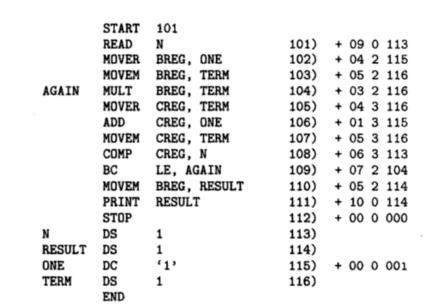
##### Mnemonic Operation Codes :

**Instruction Format :**



Sign is not a part of Instruction

##### An Assembly and equivalent machine language program :(solve it properly)



**Assembly Language Statements :**

Three Kinds of Statements

1. Imperative Statements
2. Declaration Statements
3. Assembler Directives

**Imperative Statements :** It indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction

1. **Declaration Statements :** Two types of declaration statements is as follows

[LABEL] DS <CONSTANT>

[LABEL] DC ‘<VALUE>’

The DS (Declare Storage) statement reserves areas of memory and associates names with them.

|  |  |  |
| --- | --- | --- |
| Eg)**A** | **DS** | **1** |
| **B** | **DS** | **150** |

First statement reserves a memory of 1 word and associates the name of the memory as A. Second statement reserves a memory of 150 word and associates the name of the memory as B.

The DC (Declare Constant) Statement constructs memory word containing constants. Eg ) **ONE DC ‘1’**

Associates the name ONE with a memory word containing the value ‘1’ . The programmer can declare constants in decimal,binary, hexadecimal forms etc., These values are not protected by the assembler. In the above assembly language program the value of ONE Can be changed by executing an instruction MOVEM BREG,ONE

##### Assembler Directives :

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some Assembler directives are described in the following

##### START <Constant>

Indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <Constant>

##### END [ <operand spec>]

It Indicates the end of the source program

##### Pass Structure of Assembler :

One complete scan of the source program is known as a pass of a Language Processor. Two types 1) Single Pass Assembler 2) Two Pass Assembler.

##### Single Pass Assembler :

First type to be developed Most Primitive Source code is processed only once.

The operand field of an instruction containing forward reference is left blank intially Eg) MOVER BREG,ONE

Can be only partially synthesized since ONE is a forward reference

During the scan of the source program, all the symbols will be stored in a table called **SYMBOL TABLE.** Symbol table consists of two important fields, they are symbol name and address.

All the statements describing forward references will be stored in a table called Table of Incompleted Instructions (TII)

##### TII (Table of Incomplete instructions)

|  |  |
| --- | --- |
| **Instruction Address** | **Symbol** |
| 101 | ONE |

By the time the END statement is processed the symbol table would contain the address of all symbols defined in the source program.

##### Two Pass Assembler :

Can handle forward reference problem easily.

*First Phase : (Analysis)*

* Symbols are entered in the table called Symbol table
* Mnemonics and the corresponding opcodes are stored in a table called Mnemonic table
* LC Processing

*Second Phase : (Synthesis)*

* + Synthesis the target form using the address information found in Symbol table.
  + First pass constructs an Intermediated Representation (IR) of the source program for use by the second pass.

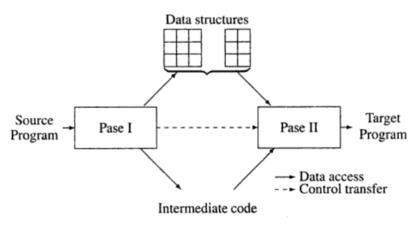
**Data Structure used during Synthesis Phase :**

* + 1. Symbol table
    2. Mnemonics table





Processed form of the source program called Intermediate Code (IC)



##### ADVANCED ASSEMBLER DIRECTIVES

1. ORIGIN
2. EQU
3. LTROG

ORIGIN :

Syntax : ORIGIN < address spec>

< address spec>can be an <operand spec> or constant

Indicates that Location counter should be set to the address given by < address spec>

This statement is useful when the target program does not consist of consecutive memory words. Eg) ORIGIN Loop + 2

EQU : *Syntax*

<symbol> EQU <address spec>

<address spec>operand spec (or) constant

Simply associates the name symbol with address specification No Location counter processing is implied

Eg ) Back EQU Loop

LTORG :

Where should the assembler place literals ?

It should be placed such that the control never reaches it during the execution of a program.

By default, the assembler places the literals after the END statement.

LTROG statement permits a programmer to specify where literals should be placed

**Algorithms :**

PASS 1

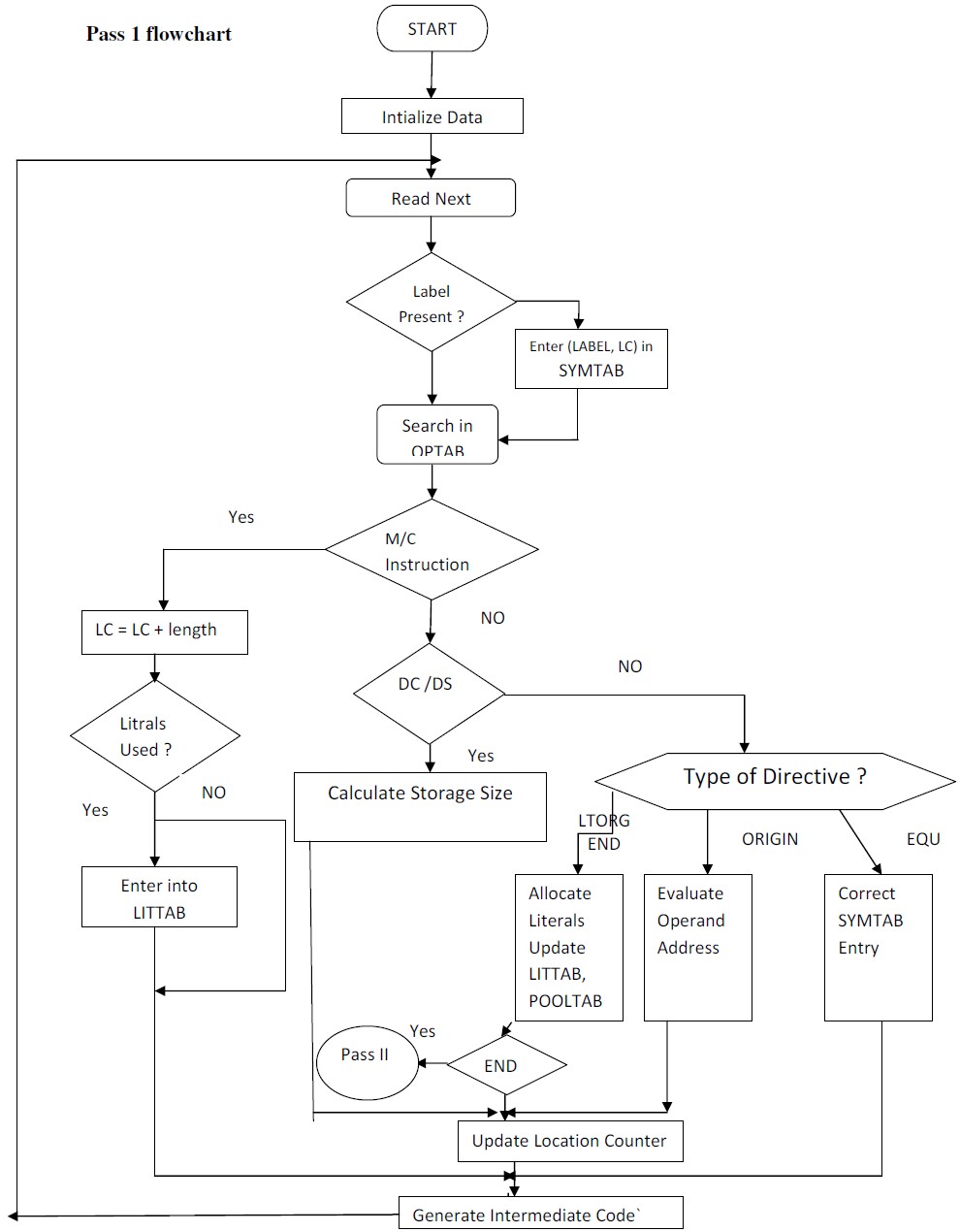
• Initialize location counter, entries of all tables as zero.

• Read statements from input file one by one.

• While next statement is not END statement

1. Tokenize or separate out input statement as label,numonic,operand1,operand2
2. If label is present insert label into symbol table.
3. If the statement is LTORG statement processes it by making it’s entry into literal table, pool table and allocate memory.
4. If statement is START or ORIGEN Process location counter accordingly.
5. If an EQU statement, assign value to symbol by correcting entry in symbol table.
6. For declarative statement update code, size and location counter.
7. Generate intermediate code.
8. Pass this intermediate code to pass -2

**Flowchart :**



**Conclusion :**

Thus , I have implemented pass-1 assembler with symbol table, literal table and pool table.

## PART – 2

##### Title:

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

##### Objectives :

* + To understand Data structure of Pass-1 & Pass-2 assembler
  + To understand Pass-1 & Pass-2 assembler concept
  + To understand Advanced Assembler Directives

##### Problem Statement :

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

##### Outcomes:

After completion of this assignment students will be able to:

* + Implemented Pass – 2 assebmler
  + Implemented machine code from intermediate code.
  + Understood concept Pass-2 Assembler.

##### Software Requirements:

Latest jdk., Eclipse

##### Hardware Requirement:

* + M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

**Theory Concepts:**

**Introduction :**

There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*. *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.



**Figure 1**. Executable program generation from an assembly source code

##### Advantages of coding in assembly language are:

* Provides more control over handling particular hardware components
* May generate smaller, more compact executable modules
* Often results in faster execution



##### Disadvantages:

* Not portable
* More complex
* Requires understanding of hardware details (interfaces)

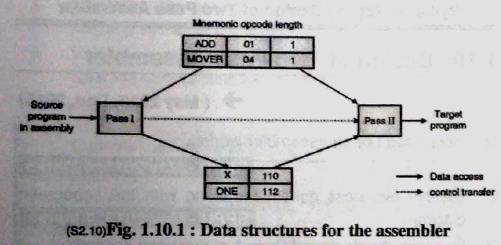
##### DESIGN OF TWO PASS ASSEMBLER:

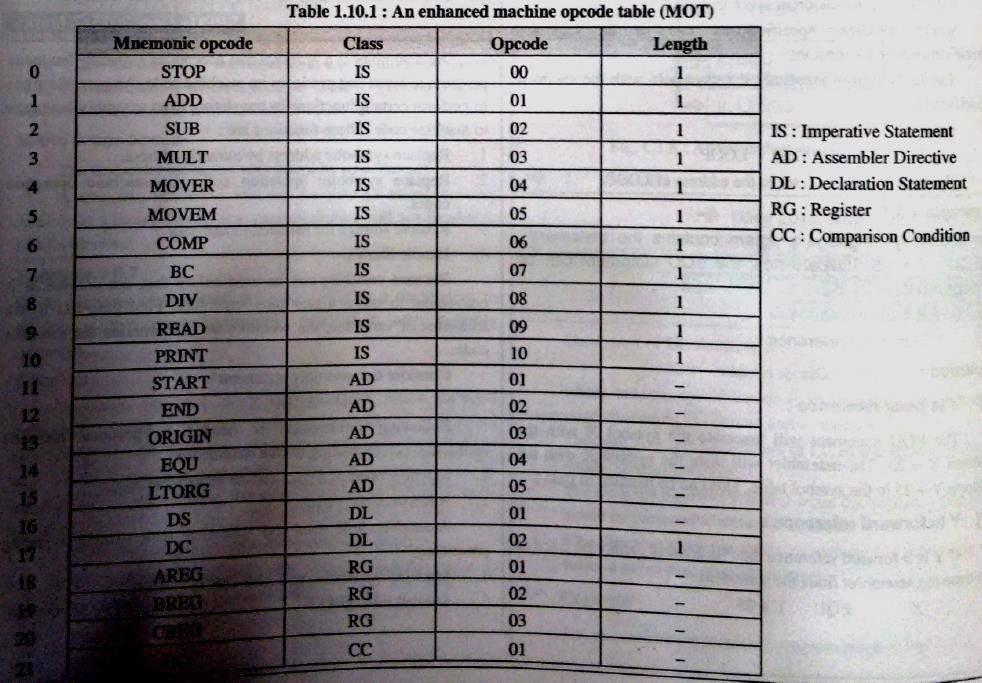
Pass I : (Analysis of Source Program)

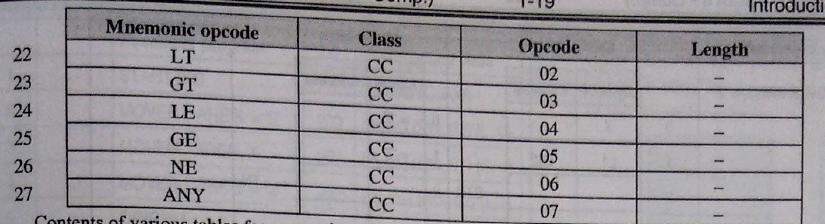
1. Separate the symbol, mnemonic opcode and operand fields
2. Build the symbol table.
3. Perform LC processing.
4. Construct intermediate representation

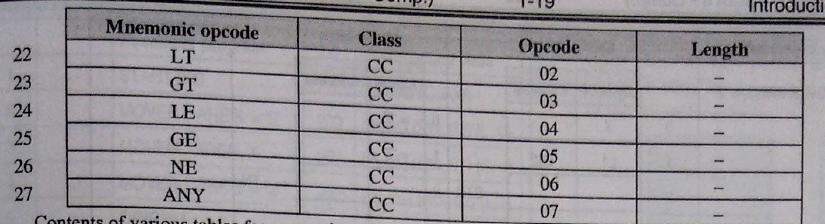
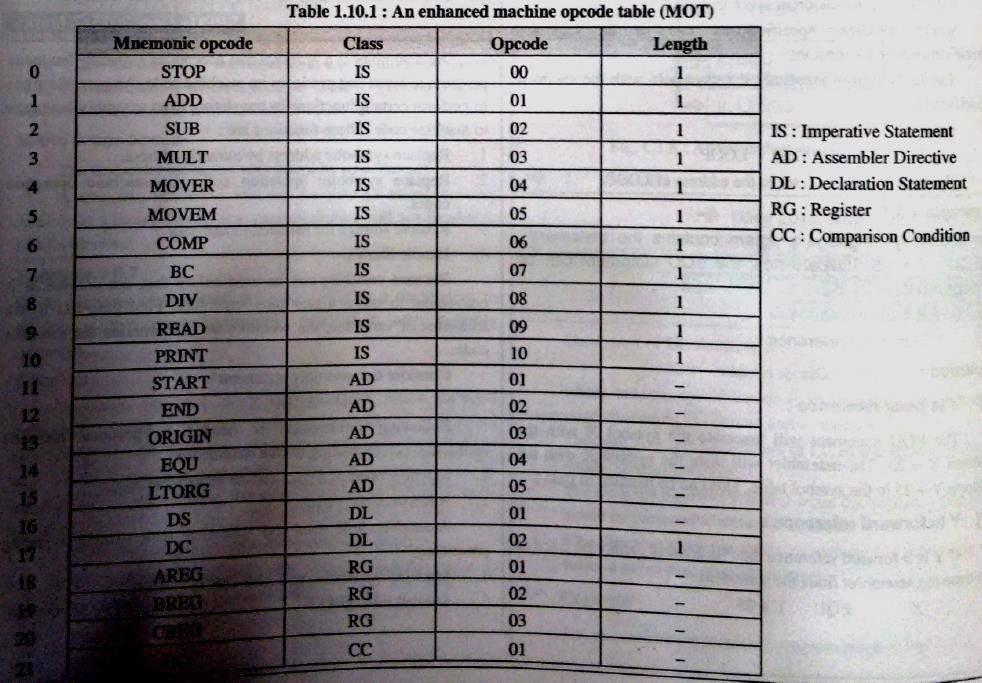
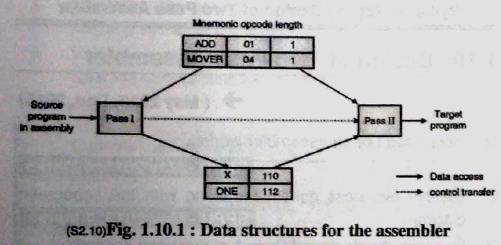
PASS 2:-

Processes the intermediate representation (IR) to synthesize the target program.









**Algorithms(procedure)**

1. **c**ode\_area\_address=address of code area; Pooltab\_ptr:=1;

loc\_cntr=0;

1. While next statement is not an END statement
2. clear the machine\_code\_buffer
3. if an LTORG statement
   1. process literals in LITTAB[POOLTAB[pooltab\_ptr]]…

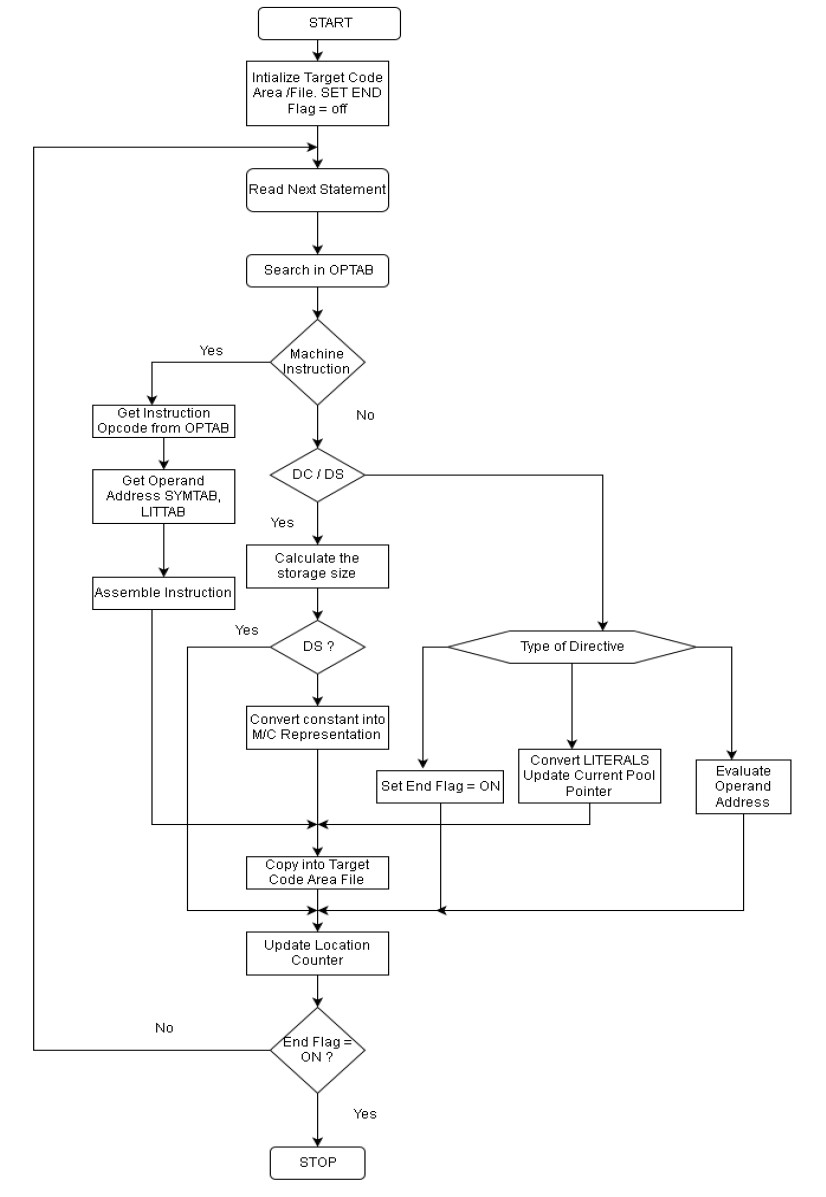
LITTAB[POOLTAB[pooltab\_ptr+1]]-1 similar to processing of constants in a dc

statement.

* 1. size=size of memory area required for literals
  2. pooltab\_ptr=pooltab\_ptr+1

1. if a START or ORIGIN statement then
   1. loc\_cntr = value specified in operand field
   2. size=0;
2. if a declaration statement
   1. if a DC statement then assemble the constant in machine\_code\_buffer
   2. size=size of memory area required by DC or DS:
3. if an imperative statement then
   1. get operand address from SYMTAB or LITTAB
   2. Assemble instruction in machine code buffer.
   3. size=size of instruction;
4. if size # 0 then
   1. move contents of machine\_code\_buffer to the address code\_area\_address+loc\_cntr ;
   2. loc\_cntr=loc\_cntr+size;
5. (Processing of END statement)

**Flowchart : .**

 **Conclusion :**

Thus , I have implemented Pass-2 assembler by taking input as output of Pass1.

