Assemblers

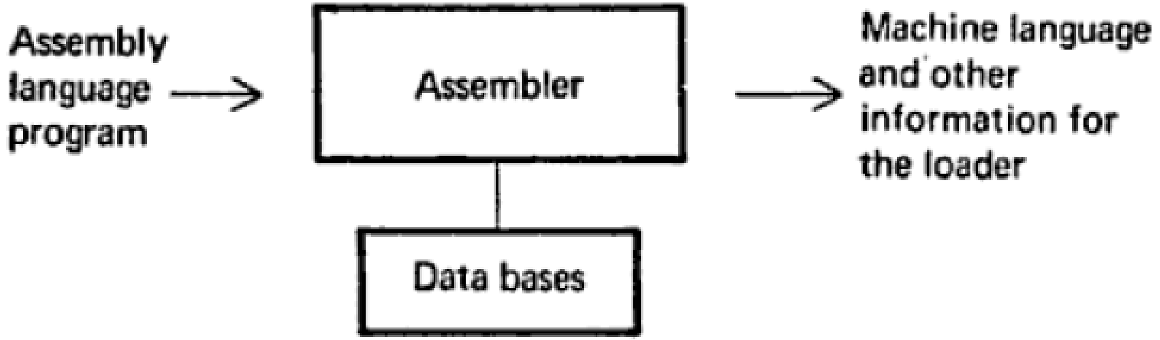
Module 2

Topics to Cover :-

1. Elements of Assembly Language programming
2. Assembly scheme
3. Pass structure of assembler
4. Assembler Design: Two pass assembler Design
5. Single pass Assembler Design for X86 processor
6. Data structures used.

ASSEMBLER

- An assembler is a program that accepts as input ASSEMBLY LANGUAGE PROGRAM and produces its MACHINE LANGUAGE equivalent along with information for the Loader


Assembly language program
Assembler
Data bases
Machine language and other information for the loader

BASIC ASSEMBLER FUNCTIONS

1. Translating mnemonic operation codes to their machine language equivalents.
2. Assigning machine addresses to symbolic labels

Elements of Assembly Language programming

1. **Command:** A command is an instruction in assembly code that tells the assembler what action to take. Assembly language commands frequently use abbreviations to keep the terminology short while also using self-descriptive abbreviations, such as "ADD" for addition and "MOV" for data transfer.

2. **Label:** A label is a word, number, or symbol that the assembly code uses to locate the date or instruction to use.

3. **Operand:** A variable or piece of data is an operand, which the assembler can manipulate.

4. **Mnemonic:** A mnemonic is a name given to a machine function or an abbreviation for an assembly language operation. In assembly, each mnemonic represents a machine instruction.

Elements of Assembly Language programming

5. Comments: Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code.

Comments can be specified in two ways:

a)**Single-line comments**, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.

```
MOV eax, 5;   I am a comment.
```

b)**Block comments**, beginning with **the COMMENT directive** and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. For example,

```
COMMENT !
This line is a comment.
This line is also a comment.
!
```

FORMAT OF ASSEMBLY LANGUAGE

- A format for a typical line from assembly language program can be given as

Label :   Mnemonic Operand1, Operand2 ; Comment

The first field, which is optional, is the label field, used to specify symbolic labels. As mentioned earlier, the presence of a label is optional, but if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction.

The second field is mnemonic, which is compulsory. All instructions, must contain a mnemonic.

The third and following fields are operands. The presence of the operand depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma.

The last field is a comment field. It begins with a delimiter such as the semicolon and continues to the end of the line. The comments are for our benefits, they tell us what the program is trying to accomplish.

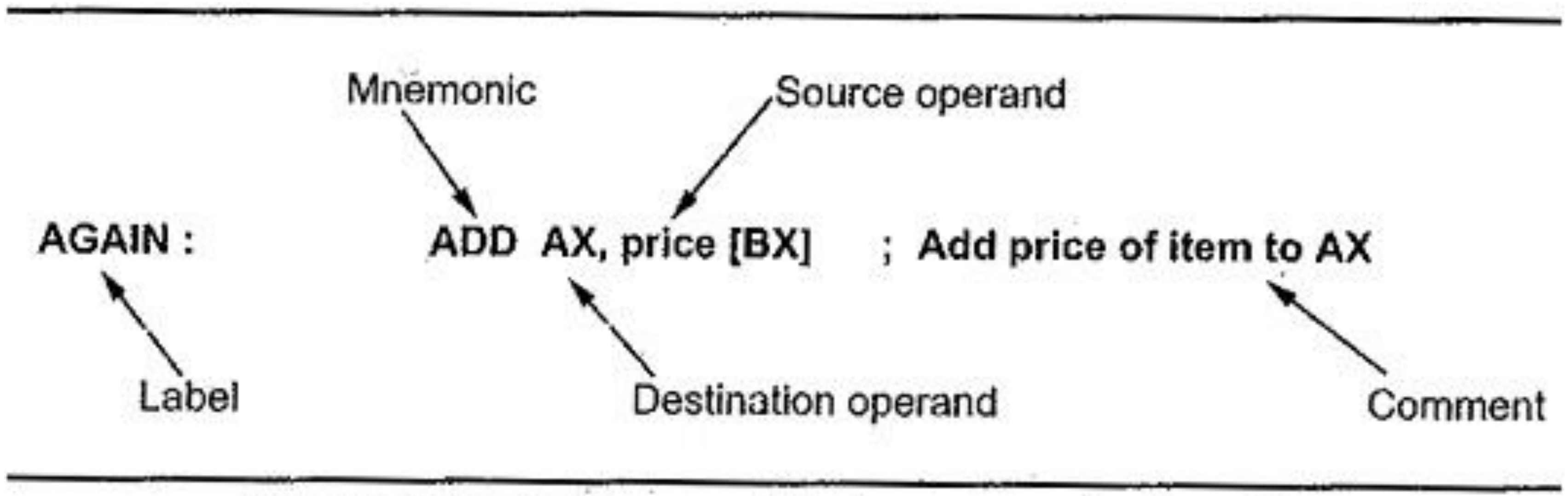
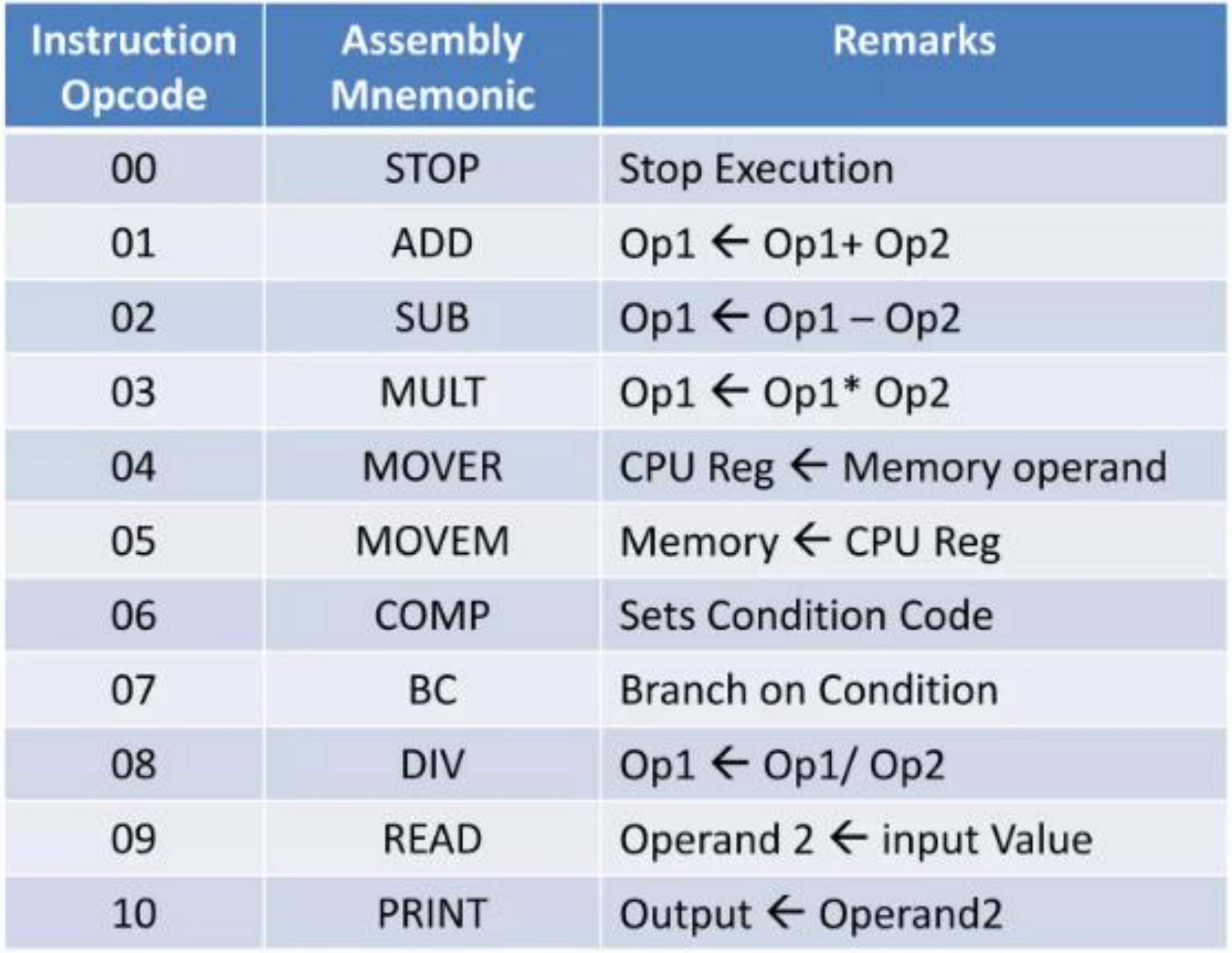


Fig. 8.2 Typical assembly language instruction

Mnemonic Operation Codes



Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 ← Op1+ Op2
02	SUB	Op1 ← Op1 – Op2
03	MULT	Op1 ← Op1* Op2
04	MOVER	CPU Reg ← Memory operand
05	MOVEM	Memory ← CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 ← Op1/ Op2
09	READ	Operand 2 ← input Value
10	PRINT	Output ← Operand2

Assembly scheme

Design Specification of an Assembler

The following 4 steps are used to develop a design specification for an Assembler

- Identify the information required to perform a task.
- Design a suitable data structure for recording the information.
- Determine the processing required for obtaining & maintaining the information.
- Determining the processing required for performing the task by using the recorded information.

Assembly scheme

Design Specification of an Assembler

The Assembler operates in two main phases: Analysis Phase and Synthesis Phase.

- The Analysis Phase breaks down the assembly language code into its constituent parts, such as symbols, labels, and instructions. It validates the syntax of the code, checks for errors, and creates a symbol table.

- The Synthesis Phase converts the assembly language instructions into machine code, using the information from the Analysis Phase.

- These two phases work together to produce the final machine code that can be executed by the computer.

- The combination of these two phases makes the Assembler an essential tool for transforming assembly language into machine code, ensuring high-quality and error-free software.

Assembly scheme - ANALYSIS PHASE

1. The primary function performed by the analysis phase **is the building of the symbol table.** It must determine the memory address with which each symbolic name used in a program are associated. **This function is called memory allocation.**

2. Memory allocation is performed by using a data structure called **Location Counter (LC).**

3. At the start of its processing, it **initializes the Location Counter to the constant specified in the START statement.**

4. While processing a statement, it checks whether the statement has a label. If so, it enters the label and the address contained in the location counter in a new entry of the symbol table. It then finds how many memory words are needed for the instruction or data represented by the assembly statement and updates the address in the location counter by that number.

Assembly scheme - ANALYSIS PHASE

5. The amount of memory needed for each assembly statement depends on the mnemonic of an assembly statement. It obtains this information from the length field in the mnemonics table.

6. **For DC and DS statements**, the memory requirement further depends on the constant appearing in the operand field, so the analysis phase should determine it appropriately.

7. **The Symbol table is constructed during analysis and used during synthesis.**

Define Storage (DS), Define Constant (DC)

The DS directive allocates space for a label;

The DC allocates space and assigns an initial value.

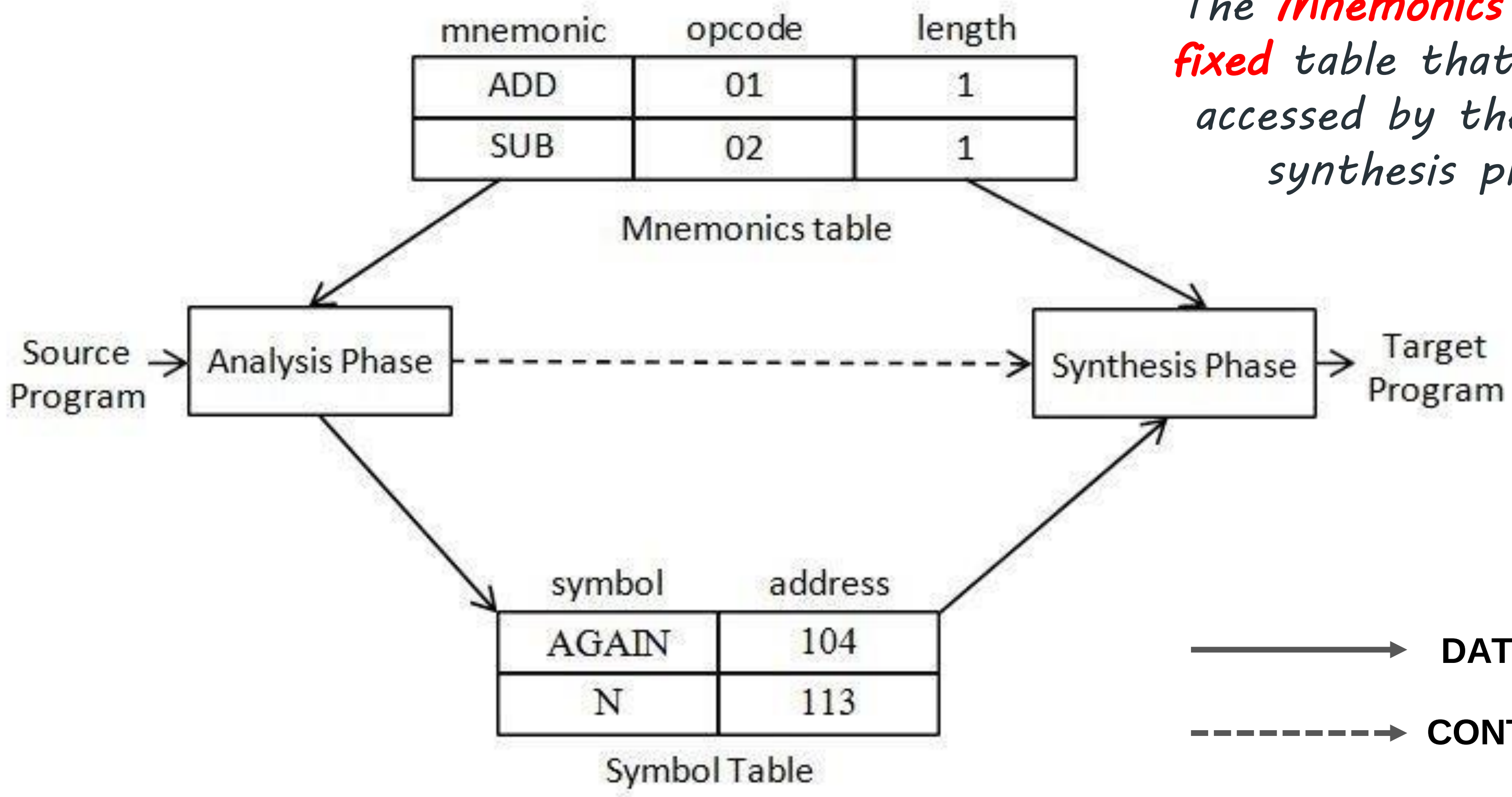
Assembly scheme - SYNTHESIS PHASE

- ✓ The Synthesis Phase is responsible for converting the assembly language instructions into machine code that can be executed by the computer. This phase uses the information from the Analysis Phase, such as the symbol table, to create the machine code and generate a listing file that shows the machine code and memory addresses for each instruction.

- ✓ The Synthesis Phase is critical for producing the final machine code that can be executed by the computer, making it the final step in the process of transforming assembly language code into machine code.

1. Uses data structures generated by Analysis phase
2. To build Machine Instructions for every assembly statements as per Mnemonic Code & their address allocation
3. Create machine instruction as per source code

Design of Assembler



mnemonic	opcode	length
ADD	01	1
SUB	02	1

Mnemonics table

symbol	address
AGAIN	104
N	113

Symbol Table

The *Mnemonics table is a fixed* table that is merely accessed by the analysis synthesis phases

⟶ **DATA ACCESS**

- - - -⟶ **CONTROL ACCESS**

Design of Assembler - SUMMARY

The tasks performed by the analysis and synthesis phases can be summarized followed as

Analysis phase:

1. Separate contents of the label, mnemonic opcode and operand fields of a statement.
2. If a symbol is present in the label field, enter the pair (symbol, <LC>) in a new entry of the symbol table.
3. Check validity of the mnemonic opcode through a look-up in the Mnemonics table.
4. Perform LC processing, i.e., update the address contained in the location counter by considering the opcode and operands of the statement.

Synthesis phase:

1. Obtain the machine opcode corresponding to the mnemonic from the Mnemonics table.
2. Obtain the address of each memory operand from the Symbol table.
3. Synthesize a machine instruction or the correct representation of a constant, as the case may be.

Source Programme

START 100
MOVER AREG, A
PRINT B
ADD BREG, ='9'
SUB BREG, D
COMP CREG, ='23'
LTORG
A DS 3
LABEL: EQU A
ORIGIN 500
L1: MULT CREG, ='7'
B DC 10
MOVEM CREG, ='7'
D DC 8
END

Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 ← Op1+ Op2
02	SUB	Op1 ← Op1 – Op2
03	MULT	Op1 ← Op1* Op2
04	MOVER	CPU Reg ← Memory operand
05	MOVEM	Memory ← CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 ← Op1/ Op2
09	READ	Operand 2 ← input Value
10	PRINT	Output ← Operand2

REGISTERS	
Reg_No	Name
01	AREG
02	BREG
03	CREG
04	DREG

Data Structures Used by Assembler

1. **Symbol Table (SYMTAB)**
2. **Literal Table (LITTAB)**
3. **Mnemonics Table or Machine Operation Table (MOT)**
4. **Pseudo – Opcode / Operation Table (OPTAB or POT)**
5. **Location Counter (LC)**
6. **Pool Table (POOLTAB)**

Data Structures Used by Assembler

1. **Symbol Table (SYMTAB):** Is used to store Values (Address) assigned to labels. It includes Name of the Symbol, Address for each label in the source program along with the length of the symbol

INDEX	SYMBOL	LC
1	LOOP	202
2	NEXT	214

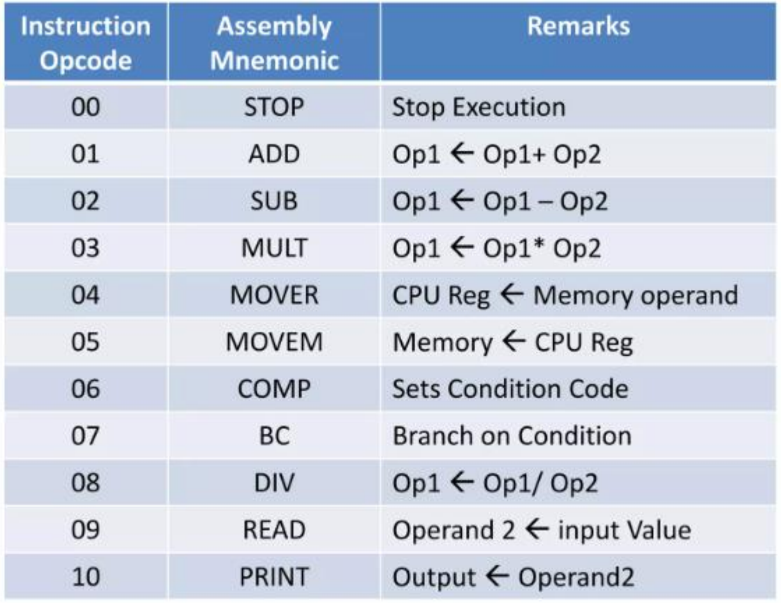
2. **Literal Table (LITTAB):** Stores each literal or constants with its location

Index	Literal	LC
1	=‘3’	200
2	=‘2’	202

Data Structures Used by Assembler

3. Mnemonics Table or Machine Operation Table (MOT)

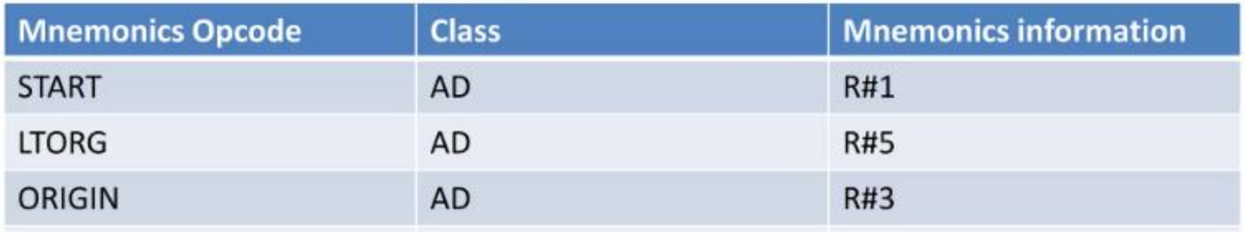
MOT is a static or fixed table in nature, in the sense that its contents does not change during the lifetime of the assembler. The mnemonic is the key for searching MOT and its value is binary code that is used for generation of machine code. This table includes all Imperative Statements (IS)



Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 ← Op1+ Op2
02	SUB	Op1 ← Op1 – Op2
03	MULT	Op1 ← Op1* Op2
04	MOVER	CPU Reg ← Memory operand
05	MOVEM	Memory ← CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 ← Op1/ Op2
09	READ	Operand 2 ← input Value
10	PRINT	Output ← Operand2

Data Structures Used by Assembler

4. Pseudo – Opcode / Operation Table (OPTAB): Contains Mnemonic opcodes, class and mnemonic information. This table includes all Assembler Directives(AD)



Mnemonics Opcode	Class	Mnemonics information
START	AD	R#1
LTORG	AD	R#5
ORIGIN	AD	R#3

5. Location Counter (LC) : Keeps the track of each instruction’s location. It is used to perform memory allocation

6. Pool Table (POOLTAB): Awareness of different literal pools is maintained using an Auxiliary table POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, current literal pool is the last pool in LITTAB

DESIGNING OF ASSEMBLER

- An Assembler can be designed in 2 ways;
    1. Single Pass
    2. Two Pass

➢ Single Pass Assembler

a) Output is given in one pass

b) Cannot solve Forward Reference Problem

➢ Two Pass Assembler

a) Output is given in TWO passes

b) Solves Forward Reference Problem

WORKING OF ONE PASS ASSEMBLER

- One pass Assembler is also called as SINGLE PASS ASSEMBLER.
- Analysis and Synthesis phase are done in the same pass.
- Location Counter Processing and symbol table is constructed
- Target Program is constructed in the same pass
- Forward Reference problem is associated with SINGLE PASS ASSEMBLER
- Forward References are handled by technique called BACKPATCHING
- There are 2 types of ONE-PASS ASSEMBLER

    1. LOAD – and – GO Assembler
    2. Object Program Output Assembler

SINGLE PASS ASSEMBLER

- Data Structures Used are:-
  1. **MOT**
  2. **POT**
  3. **DL TABLE**
  4. **REGISTERS TABLE**

No intermediate code will be generated, directly Target code will be generated

The **Table of Incomplete Instruction** is required to store unavailable information of different symbols or literals

It converts mnemonic code into machine code and put the machine code instructions straight into the computer memory to be executed.

FORWARD REFERENCE PROBLEM

- Rules of ALP state that the symbol can be defined anywhere in the program. Hence there can be cases in which **symbol is used before it is defined** and such a case is called **Forward Reference**

- Due to Forward Reference, Assembler cannot find the address of the symbol and hence not able to assemble the instructions. Such a problem is called **Forward Reference Problem**

- Solution to Forward Reference Problem
    1. Backpatching
    2. Using TWO PASS Assembler

FORWARD REFERENCE PROBLEM - Solution

BACKPATCHING

1. Assembler leaves the operand field of instruction blank if it contains a forward reference to that symbol
2. Address of symbol is put in operand field sometime after the definition of symbol is encountered in the program
3. Builds a Table of Incomplete Instructions. This table has entries for SYMBOL and its corresponding INSTRUCTION ADDRESS
4. By the time the END statement is processed, symbol table contains all addresses of symbols and each entry in table of incomplete instructions is processed by assembler

FORWARD REFERENCE PROBLEM - SOLUTION

- **Make two passes over the input program**

- Purpose of PASS 1 is to define the symbols and the literals encountered in the program

- Purpose of PASS 2 is to assemble the instructions. Thus, In PASS 2 Input file is read again along with Symbol Table and an assembled object code file is generated in the output.

TWO PASS ASSEMBLER

- The two pass assembler performs **Two Passes** over the source program.

- In the first pass, it reads the entire source program, looking only for label definitions. All the labels are collected, assigned address, and placed in the symbol table in this pass, no instructions are assembled and at the end the symbol table should contain all the labels defined in the program. To assign address to labels, the assembler maintains a **Location Counter (LC).**

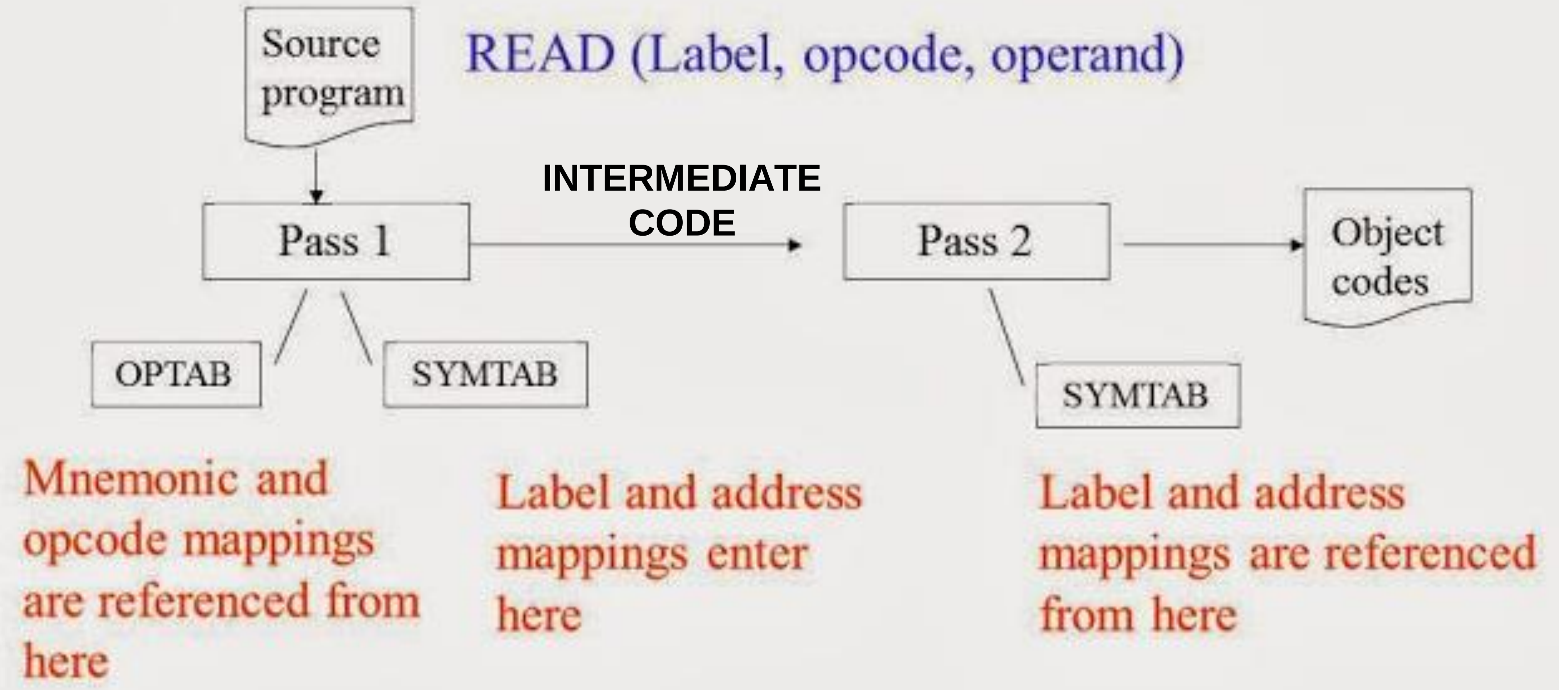
SUMMARY

1. Read the assembly language program one line at a time
2. Ignore anything that is not required e.g. COMMENTS
3. Create Symbol Table to enter Symbolic Address and Labels into specific address

TWO PASS ASSEMBLER

- In the second pass the instructions are again read and are assembled using the symbol table. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.

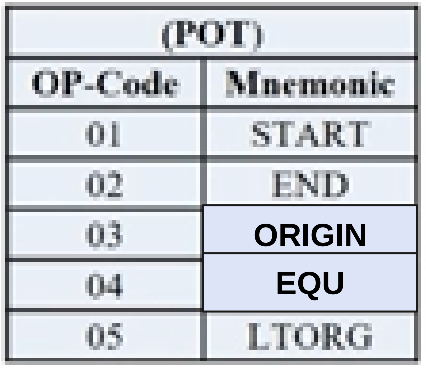
A Simple Two Pass Assembler Implementation


Source program
READ (Label, opcode, operand)
INTERMEDIATE CODE
Pass 1
Pass 2
Object codes
OPTAB
SYMTAB
SYMTAB
Mnemonic and opcode mappings are referenced from here
Label and address mappings enter here
Label and address mappings are referenced from here

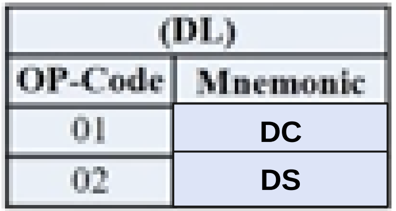
PASS 1 OF TWO PASS ASSEMBLER

- Data structures used in PASS 1 of TWO PASS Assembler are:-
  1. MOT
  2. POT
  3. DL TABLE
  4. REGISTERS TABLE

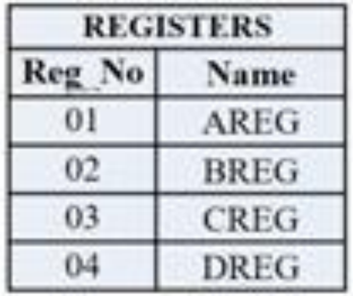
MOT		
Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 ← Op1+ Op2
02	SUB	Op1 ← Op1 – Op2
03	MULT	Op1 ← Op1* Op2
04	MOVER	CPU Reg ← Memory operand
05	MOVEM	Memory ← CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 ← Op1/ Op2
09	READ	Operand 2 ← input Value
10	PRINT	Output ← Operand2



(POT)	
OP-Code	Mnemonic
01	START
02	END
03	ORIGIN
04	EQU
05	LTORG



(DL)	
OP-Code	Mnemonic
01	DC
02	DS



REGISTERS	
Reg_No	Name
01	AREG
02	BREG
03	CREG
04	DREG

TYPES OF ASSEMBLY STATEMENTS

Assembly Instructions are categorized into THREE:-

1. **Imperative Statements (IS):-** These indicate an action to be performed during execution of the assembled program. Each imperative statement typically translates into one machine instruction.

Ex:- SUB BREG,X

(Subtract the content of X from the content of BREG)

2. **Assembler Directives (AD) -** These instruct the assembler to perform certain actions during the assembly of a program. For example: **START <constant>** *directive* indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <constant>. No memory allocation is done for these statements, it just translates assembly instructions to Intermediate code. The representation that is used in Intermediate code is ***(instruction_type, opcode)***

TYPES OF ASSEMBLY STATEMENTS

3. Declarative Statements (DL): The syntax of declaration statements is asfollows:

[Label] DS<constant>

OR

[Label] DC '<value>' .

The DS statement reserves areas of memory and associates names with them. The DC statement constructs memory words containing constants.

Declare Storage (DS) declares the storage area or declares the constant in program. The DS statement reserves areas of memory and associates names with them.

Syntax:- <Symbol> DS <constant>

Eg:- ONE DS 1

The 1 word memory location is reserved and it associate name ONE

Declare Constant: The DC statement is used to construct memory words containing constants.

Syntax:- <Symbol> DC '<value>'

Eg. ONE DC 1

The memory location containing the value 1 and is associates name ONE.

ASSEMBLER DIRECTIVES

- ✓ These statements instructs the assembler to perform certain action during the assembly of a program.
- ✓ These are **pseudo instructions and are not translated to machine code.**

START: This directive indicates that the first word of the target program generated by the assembler should be placed in the memory with address <constant>

Syntax : START <constant>

Example: START 113

ORIGIN: Sets the **Location Counter** Value to the specified address

Syntax : ORIGIN <address specification>

Example: ORIGIN 113

ASSEMBLER DIRECTIVES

EQU: Stands for EQUATES. Gives a symbolic name to a numeric constant. It can also assign address to some constant.

Syntax : <symbol> EQU <address specification>

Example: BACK EQU 113

OR

D EQU A -- D is set to address of Y

LTORG: Allocates addresses for literals. It also assembles literals into a literal pool

Ideally, as soon as LTORG encountered all the literals are assigned addresses

Syntax : LTORG

Example: LTORG

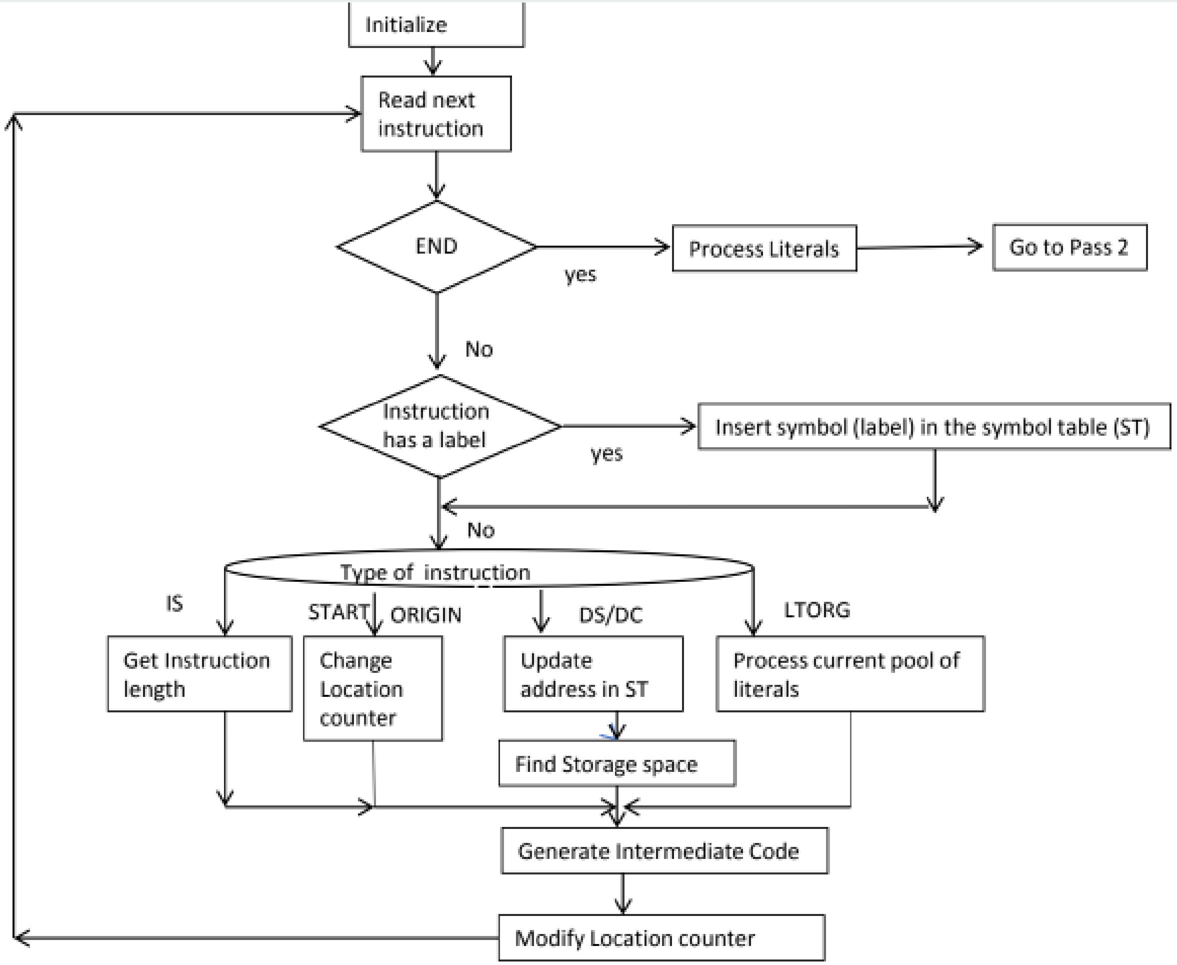
END: This directive indicates the end of the source program

Syntax : END

PASS 1 - EXAMPLE

START 200
MOVER AREG, ‘5’
MOVEM AREG , X
L1: MOVER BREG , ‘2’
ORIGIN L1 + 3
LTORG
NEXT: ADD AREG , ‘1’
SUB BREG , ‘2’
LTORG
BACK: EQU , L1
MULT CREG , ‘4’
STOP
X DS 1
END

Flow Chart of Pass-1 of Two pass Assembler



UNIVERISTY QUESTIONS

1. Draw flowchart and explain with databases the working pass 2 of assembler.
2. With reference to assembler, explain the following tables with suitable example.
   (i) POT (ii) MOT (iii) ST (iv) LT
3. Explain different types of errors that are handled by Pass 1 and Pass 2 of assembler
4. Explain the flowchart design of two pass assembler
5. Explain different assembler directives with example
6. Discuss with example Forward Reference
7. State and explain different types of statements used in assemblers w.r.t system programming
8. Draw and explain the flowchart of pass 1 of two pass assembler with example
9. Explain forward reference problem and how it is handled in assembler design

Consider the following Assembly Program:-

```
START    501
      A  DS 1
      B  DS 1
      C  DS 1
READ A
READ B
MOVER   AREG, A
ADD     AREG, B
MOVEM   AREG, C
PRINT C
END
```

Generate Pass-1 and Pass-2 and also show the content of Database table involved in it.