



BACHELOR OF COMPUTER APPLICATIONS SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 3

Assemblers

Table of Contents

SL No	Topic	Fig No / Table / Graph	SAQ / Activity	Page No
1	Introduction	-	-	3
1.1	Learning Objectives	-	-	
2	Introduction to Assembler	1	-	4 -5
3	Assembler Directives	-	-	6 - 8
4	Forward Reference	-	1	9 - 10
5	Types of Assemblers	-	-	11 - 12
6	Data Structures of Assembler	2, 3, 4	2	13 - 16
7	Assembler Design – One Pass Assembler	-	-	17 - 18
8	Two Pass Assembler	-	3	19 - 20
9	Summary	-	-	21
10	Glossary	-	-	21
11	Terminal Questions	-	-	22
12	Answers	-	-	22 -23
13	Suggested Books and E-References	-	-	23

1. INTRODUCTION

An assembler is a program (system software) that accepts an assembly language program as input and produces its equivalent machine language program as output along with information for the loader. The input to the assembler program is called the source program and the output is called the object program.

This Unit discusses the functions of assemblers in detail. In this unit, we will see the design of an assembler, assembler directives, Data structures used for the construction of the assembler, and Types of assemblers. This unit also discusses single pass and two pass assembler designs.

1.1 Learning Objectives

After studying this unit, you should be able to:

- ❖ *Define an Assembler*
- ❖ *Describe forward references*
- ❖ *Describe Assembler directives*
- ❖ *Explain about Types of Assemblers*
- ❖ *Explain different Data structures of Assembler*
- ❖ *Explain about one pass and two pass Assembler design*

2. INTRODUCTION TO ASSEMBLER

Assembler is a program, which translates Assembly Language Program (ALP) into machine language program (object program). It places the object program in the secondary memory.

An **assembly language** is a machine-dependent, low-level programming language which is specific to a certain computer system (or a family of computer systems). Compared to the machine language of a computer system, it provides three basic features which simplify programming:

1. **Mnemonic operation codes:** The use of mnemonic operation codes (also called mnemonic opcodes) for machine instructions eliminates the need to memorize numeric operation codes. It also enables the assembler to provide helpful diagnostics, for example, the indication of a misspelling of operation codes.
2. **Symbolic operands:** Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in the assembly statement. The assembler performs memory bindings to these names; the programmer need not know any details of the memory bindings performed by the assembler.
3. **Data declarations:** Data can be declared in a variety of notations, including decimal notation. This avoids manual conversion of constants into their internal machine representation, for example, conversion of -5 into (11111010)₂ or 10.5 into (41A80000)₁₆.

An assembler translates a file of assembly language statements into a file of binary machine instructions. The translation process has two major parts. The first step is to find memory locations with labels so that the relationship between symbolic names and addresses is known when instructions are translated. The second step is to translate each assembly statement by combining the numeric equivalents of opcodes, register Specifies, and labels into a legal instruction. The assembler produces an output file, called an object file, which contains the machine instructions, data, and book keeping (reference) information.

The functions of the assemblers can be understood from Fig. 3.1. The assembler receives the assembly language program as input and converts it into the corresponding object program.

It also provides information to the loader. The loader places this object code into the main memory during execution.

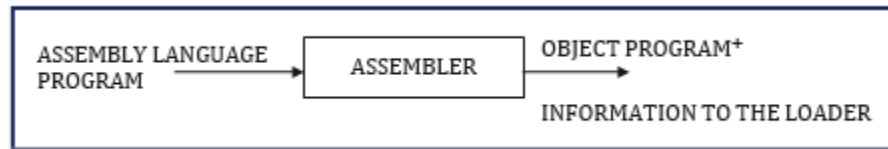


Fig 1: Assembler

The assembler does receive an assembly language program as its input. The output should be an object program. This will be placed generally in the secondary memory. This will be again loaded placed in the main memory for execution of the program by the loader. So the assembler has to also generate necessary information for the loader.

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)

An assembler does the following:

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

3. ASSEMBLER DIRECTIVES

These are the instructions present in the source program used to instruct the assembler to perform certain actions during the translation of a program. These are not translated into machine language instructions. The assembler directives are also called '*pseudo instructions*'. These are not the instructions for the processors, but for assemblers. The assembler directives discussed are from MASM assembler for the Intel family of architectures. These assembler directives are used in Intel 8086 macro assembler, Borland Turbo Assembler, and IBM macro assembler (MASM).

E.g.: ORG 8000H

-----END

An *assembler directive* is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process; for example, an assemble directive tells the assembler where a program is to be located in memory. We are going to use the following directives:

<label>	EQU	<value>	//Equate
	ORG	<value>	//Origin
<label>	DC	<value>	//Define constant
<label>	DS	<value>	//Define storage
	END	<value>	//End of assembly language program and "starting address" for execution

In each case, the term <label> indicates a user-defined label (i.e., symbolic name) that must start in column 1 of the program, and <value> indicates a value that must be supplied by the programmer (this may be a number or a symbolic name that has a value).

EQU

The **EQU** assembler directive simply equates a symbolic name to a numeric value. Consider:

Sunday	EQU	1
Monday	EQU	2

The assembler substitutes the equated value for the symbolic name; for example, if you write the instruction **ADD.B #Sunday, D2**, the assembler treats it as if it were **ADD.B #1, D2**.

You could also write

Sunday	EQU	1
Monday	EQU	Sunday + 1

In this case, the assembler evaluates “Sunday + 1” as 1 + 1 and assigns the value 2 to the symbolic name “Monday”.

Do not think that the EQU directive creates variables or constants. It doesn’t and it does not affect the code generated by the program. This directive simply allows you to make a name equivalent to its value (i.e., it’s a form of shorthand).

Origin

The *origin directive* tells the assembler where to load instructions and data into memory. The first 1024 bytes of memory are reserved for exception vectors. So, Your programs will start at location 1024; that is, you should begin your program with **ORG 1024** or **ORG \$400** (remember that $1024 = 400_{16}$).

Define Constant

The *defined constant* assembler directive allows you to put a data value in *memory at the time that the program is first loaded*. The **DC** directive takes the suffix. **B**, **W**, or **.L**. You can put several values on one line (each value is separated by a comma). The optional *label field* is given the address of the first location in memory allocated to the DC function. Consider the example:

	ORG	\$2000	Locate data here
Val1	DC.B	20,34	Store 20 and 34 in consecutive bytes

Define Storage

The *defined storage* directive is used to reserve one or more memory locations. This directive is similar to the Pascal **type** declaration. Consider:

Result	DS.B 1	Save a byte for Result
Table	DS.W 10	Save 10 words (20 bytes) for Table
Point	DS.L 1	Save 1 long word (4 bytes) for Point



4. FORWARD REFERENCE

It is a reference of a label (variable name), which is defined later in the program. Now consider the simple program in 8086 given in example 3.1.:

Example 3.1

CODE SEGMENT

ASSUME CS: CODE JMP **XXYY**

NOP NOP

XXYY: MOV AX, 0000H

NOP

CODE ENDS

END

This is a simple program written for Intel 8086 processor. This is a delay program. In the above program, there is a JMP statement to statement labeled 'XXYY'. In this case, there is a reference to a label named 'XXYY', which is defined later in the program. This is called forward reference. This is also called forward jump. This causes some problems in the translation of the assembly program into the object program. There can be a backward reference or backward jump possible in the program. In this case, reference to a label or symbol, which is defined already in the program.

The forward reference causes problems during assembly. During the assembly process, the assembler reads the instruction from the source program and translates them one after another. The assembler integrates the opcode and address of each instruction. In the case of an instruction having forward reference, the address will not be readily available. The address is actually defined later, after the particular instruction. See the instruction 'JMP **XXYY**' in example 3.1. The label is defined later in the program. This causes a problem in the assembly process. The assembly cannot be done till the address is known. This is called the *forward reference problem*. This problem is handled by some suitable mechanisms in the assemblers.

Self-Assessment Questions -1

1. A System Software that accepts assembly language program as input and produces its equivalent machine language program as output is called _____.
2. The System Software which places the object code into the main memory during execution is called _____.
3. The instructions present in the source program used to instruct the assembler to perform certain actions during the translation of a program are called _____.
4. The assembler directive simply equates a symbolic name to a numeric value is called _____.
5. A reference of a label, which is defined later in the program is called _____.



5. TYPES OF ASSEMBLERS

The assemblers can be of any one of the following types:

- 1) Single pass assemblers
- 2) Two pass assemblers and
- 3) Multi pass assemblers

Pass is a terminology used in system software. Each reading of a program can be called a **pass**. The assembly process can be done within a pass, if so, such assemblers are single pass assemblers. If the assembly is done in two passes, then those assemblers are called two pass assemblers. The translation of the assembly language program can be done in several passes. Such assemblers are called multi pass assemblers.

Single Pass Assemblers

In single pass assemblers, the entire translation of the assembly language program into an object program is done in only one pass. The source program is read only once. These are also called 'one pass assembler'. These assemblers suffer the problem of Forward references. Handling the forward reference in a single pass assembler is difficult.

The object code can be produced in the single pass assemblers in two different ways. In a first way, the object code is directly loaded into the main memory for execution. Here, no loader is required. This type of loading scheme is called the 'compile and loading scheme'. In a second way, the object program will be loaded into the main memory for execution later as the necessity arises. Here, a separate loader program is necessary.

The forward reference problem is also handled to some extent in single pass assemblers. The assembler scans the source assembly language program, instruction by instruction and translates them into machine codes. If there is an instruction, which has a forward reference, the assembler leaves the address of that particular symbol (operand) and proceeds further. This symbol is entered in the symbol table and marked as undefined. There is another list of instructions that uses this symbol, which is also maintained. In that list, this particular instruction is also included. When the symbol definition is encountered the assembler inserts the address in the symbol table. The assembler also inserts the address to the

instructions, which use this symbol in the list of Forward references. This is how the forward reference problem is handled in the single pass assembler in a single pass. The forward reference problem is conveniently handled in the two pass assemblers. If undefined symbols are encountered after reaching the end of the statement, then the assembler generates the error messages. After translating all the instructions, the object program is used with and without the use of loaders. The control is transferred to the start of the object program for execution.

An assembler, which goes through an assembly language program only once, is known as One-pass assembler. It suffers from a forward reference problem. This is faster because they scan the program only once. This does not have many features supported by the two pass assemblers.

Two Pass Assemblers

The two pass assemblers are widely used and the translation process is done in two passes. The two pass assemblers resolve the problem of Forward references conveniently. An assembler, which goes through an assembly language program twice, is called a two **pass assembler**. During the first pass, it collects all labels. During the second pass, it produces the machine code for each instruction and assigns the address to each of them. It assigns addresses to labels by counting their position from the starting address. It provides more features than the single pass assembler. It is widely used.

Multi Pass Assemblers

If the assembly process is done more than two passes then those assemblers are called **multi pass assemblers**. One reading of a program (source program or any program in concern) can be called a pass. In assemblers, the passes are necessary to solve the problem of Forward references. In order to process the entire symbol definitions, several passes are made over the source program.

6. DATA STRUCTURES OF ASSEMBLER

This section discusses the data structures used for the construction of an assembler. The data structures used in the assembler are given below:

- 1) Mnemonics Table (MT)
- 2) Symbol Table (ST)
- 3) Pseudo Instruction Table (PIT)
- 4) Location Counter (LC)
- 5) Literal Table (LT)
- 1) Mnemonics Table (OPTAB)**

Mnemonics Table also called 'OPTAB' is a data structure used for maintaining the details (attributes) about the instructions of an instruction set of any machine. It will have mnemonics of the instructions, corresponding hexadecimal opcode, and the length of the instruction. The structure of the mnemonic table is shown in below figure 3.2

Mnemonics	Opcode	Length
SUB B	97	1
ADD B	80	1
...
HLT	76	1

Fig 2: Mnemonic Table Organization

This table can also be called a *machine code table* or *Machine opcode table* or *machine table*.

This is a static table (no update) used for maintaining mnemonic, machine code (instruction format, length), etc. Contents of this table include:

- Mnemonic Codes of all instructions:
- Machine language opcode

- Other Information (for architectures with more than one length of instruction format) like Length of instruction and Instruction formats

Operations in pass 1:

- Validate opcodes
- Compute instruction length (in SIC/XE) In pass 2
- Translate opcodes to machine language

2) Symbol Table (SYMTAB)

A symbol table also called SYMTAB is also a data structure used for maintaining the details (attributes) of the symbols present in the assembly language program. The symbols may be labels like identifier names. The attributes, which can be maintained, are name and address. Other attributes like scope and value are also maintained if necessary. The structure of the symbol table is given in the following figure 3.3

Symbol name	Address
A	8080
B	8090
...	...
C	8100

Fig 3: Symbol table organization

The suggested data structure for both symbol table and Mnemonic table can be Hash Table, because of the efficiency in Insertion and retrieval.

Properties

This table is used for storing label name, value, flag, (type, length), etc.

- Dynamic table (insert, delete, search)
- Hash table, non-random keys, a hashing function
- Contents include
 - Name of symbol

- Address (value)
 - Error flags (from pass1)
 - Other information: attributes of data or instruction labeled
- Operations in pass 1
 - Enter labels as they are encountered in the source program
 - Enter addresses from LOCCTR
- Operations in pass 2
 - Look up operand labels for addresses
- Also organized as a hash table
 - Entries only (no deletions)
 - Non-random keys

3) Pseudo Instruction Table (PIT)

Sometimes the assemblers maintain a separate data structure called Pseudo Instruction Table (PIT). This table maintains the details of the Pseudo instructions or assembler directives like ORG, END, ASSUME, and others.

4) Location Counter (LOCCTR)

Location Counter also called 'LOCCTR' is another data structure used to keep the address of the next memory word. LC needs to know the length of different instructions (for increment-ing), hence MT (Mnemonic Table) can be included to have the length of the instruction. The process of maintaining the address of the next memory word can be called 'LC processing'. The location counter is also called an instruction counter. It is a variable, which keeps track of the execution-time address of the instruction being assembled.

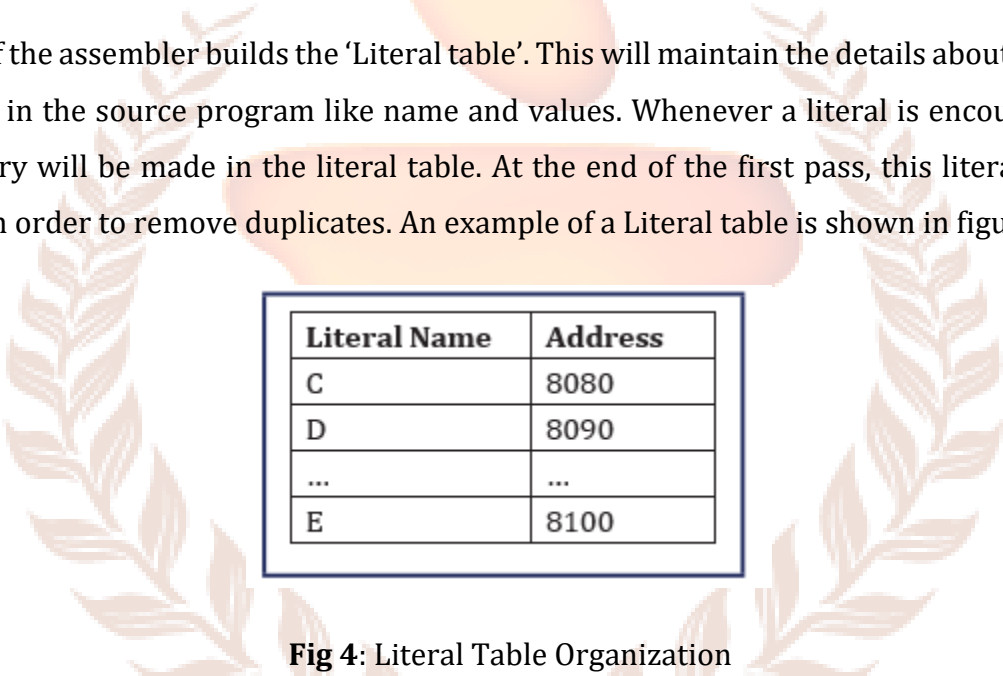
Properties

- i) Counted in bytes.
- ii) Initialized to address specified in START directive.
- iii) Length of each assembled instruction is added to LOCCTR.
- iv) LOCCTR points to starting address of each statement in the program.

Literal Table (LT)

Constants for which assembler automatically reserves memory is called Literals. A literal is an operand that states the required value directly in an instruction. It is an alternative to defining a constant elsewhere in the program and using a label to refer to it. The literals improve the readability of a program by making the value of the constant apparent in the source statement.

Pass 1 of the assembler builds the 'Literal table'. This will maintain the details about the literals used in the source program like name and values. Whenever a literal is encountered, a new entry will be made in the literal table. At the end of the first pass, this literal table is sorted in order to remove duplicates. An example of a Literal table is shown in figure 3.4.



Literal Name	Address
C	8080
D	8090
...	...
E	8100

Fig 4: Literal Table Organization

Self-Assessment Questions -2

- Each reading of a program can be called a _____.
- A data structure used for maintaining the details (attributes) about the instructions of instruction set of any machine is called ____.
- The table maintains the details of the Pseudo instructions or assembler directive is called ____.

7. ASSEMBLER DESIGN – ONE PASS ASSEMBLER

A one-pass assembler scans the program just once. One-pass assemblers are used when,

1. It is necessary or desirable to avoid a second pass over the source program.
2. The external storage for the intermediate file between two passes is slow or inconvenient to use.

Here, the Main problem is forward references to both data and instructions. One simple way to eliminate this problem is, it requires that all areas be defined before they are referenced. But, forward references to labels on instructions cannot be eliminated as easily. The one-pass assembler must make some special provision for handling forward references. There are two types of one-pass assemblers. One type of one-pass assembler produces object code directly in memory for immediate execution. Here, No object program is written out and no loader is needed. The other type of one-pass assembler produces the usual kind of object program for later execution.

The assembler that does not write the object program out and does not need a loader is called a **load-and-go** assembler. It avoids the overhead of writing the object program out and reading it back in. It is useful in a system that is oriented toward program development and testing. A load-and-go assembler can be a one pass assembler or a two-pass assembler.

Handling of Forward references in one-pass load-and-go assembler

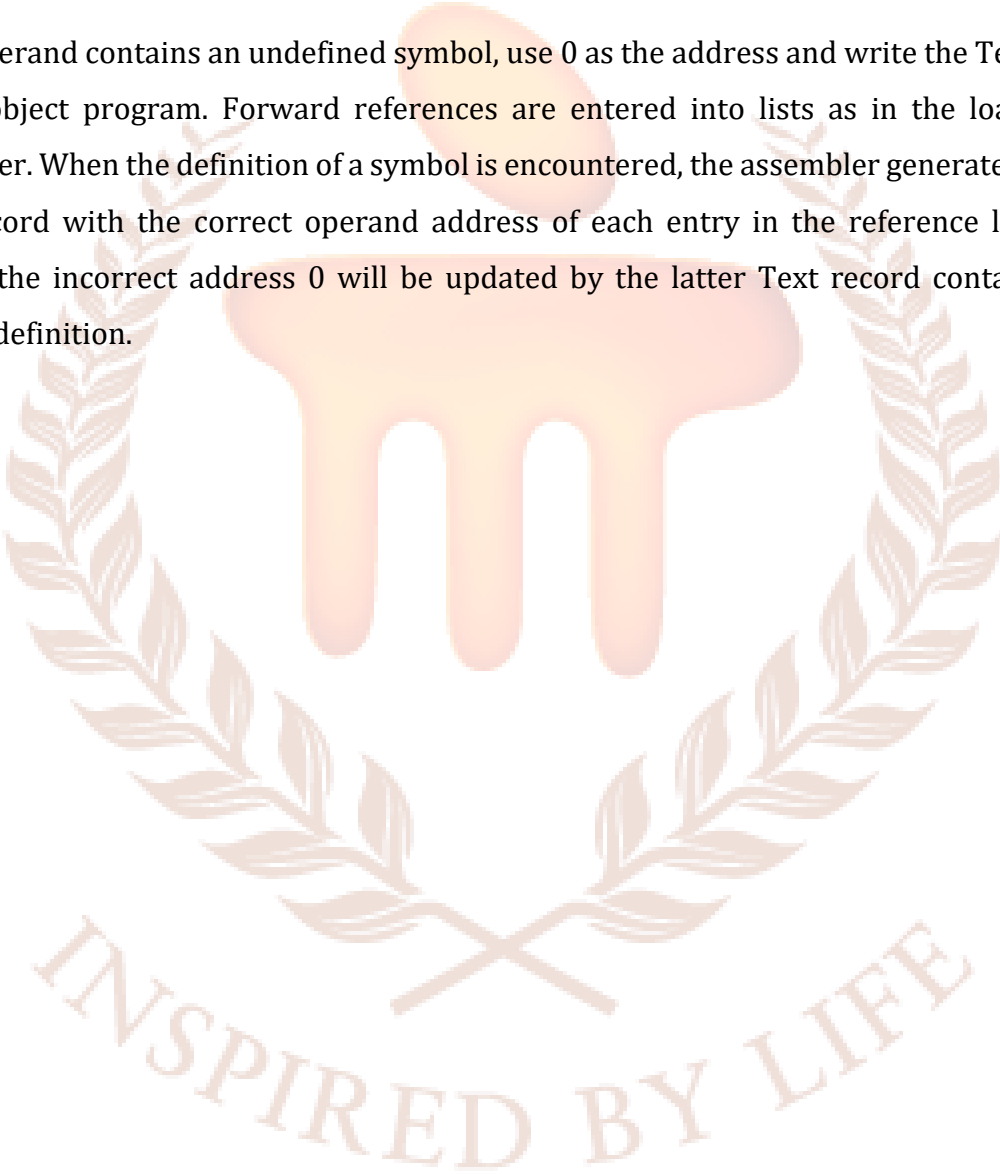
The assembler generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, then,

- The symbol is entered into the symbol table with a flag indicating that the symbol is undefined;
- The operand address is omitted when the instruction is assembled;
- The operand address is added to a list of Forward references associated with the symbol table entry.

When the definition for a symbol is encountered, the forward reference list for that symbol is scanned, and the proper address is inserted into any instructions previously generated. For a load-and-go assembler, the actual address must be known at assembly time.

One Pass Assembler that Produces Object Program

If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program. Forward references are entered into lists as in the load-and-go assembler. When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list. When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.



8. TWO PASS ASSEMBLER

The Design of Two pass assemblers is discussed in this section. The design of two pass assembler involves the design of Pass 1 and Pass 2.

Pass 1 of the Two Pass Assembler

It scans the assembly language program completely and analyses the same. This phase can be also called as *Analysis Phase*. The assembler analyses the assembly language program considering the following factors:

1. Symbols and labels.
2. Mnemonics and
3. Declarations.

The assembler uses the location counter and Mnemonics table for the pass1. It generates an intermediate representation of the source code (IR) and symbol table. The symbol table will maintain the details of the Symbols.

Functions of Pass 1 of the Two Pass Assembler

1. Tracks the location counter
2. Generate the symbol table by determining the symbols, labels, and their addresses
3. Generates the Literal Table (LT)
4. Finds the length of assembly level instructions from the mnemonics table.
5. Processes the assembler directives
6. Finally generates the intermediate representation of the Source Program.

Pass 2 of the Two Pass Assembler

This phase only generates the object code of the source program. It uses the intermediate representation from the pass 1 and symbol table. It actually synthesizes the outputs of the Pass 1 and opcodes to generate the object code. This pass can also be called as *Synthesis Phase*. It just integrates the Opcode and addresses. It uses the symbol table, Literal table, mnemonic Table, and Intermediate representation of the ALP from Pass1. The function of the pass2 is to generate the assembly listing. It should also generate some information

needed by the loader and linkers for linking up procedures assembled at different times into a single executable file and loading into the main memory.

Functions of Pass 2 of the Two Pass Assembler

1. Finds out the machine codes (Opcodes) for the assembly level instructions from the mnemonics table.
2. Finds out the addresses of the symbols from the symbol table and substitutes them in the assembly sequence.
3. Processes the assembler directives
4. Generate the object code sequence
5. Generates necessary information to the loader

Self-Assessment Questions -3

9. The assembler that does not write object program out and does not need a loader is called_____.
10. The phase which scans the assembly language program completely and analyses the same can be also called as_____.
11. The assembler uses the_____, _____for the pass1.

9. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- An assembler is a program (system software) that accepts assembly language program as input and produces its equivalent machine language program as output along with information for the loader.
- An assembly language is a machine-dependent, low-level programming language which is specific to a certain computer system.
- An *assembler directive* is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process.
- Each reading of a program can be called a pass. The assembly process can be done within a pass, if so, such assemblers are single pass assemblers. If the assembly is done in two passes, then those assemblers are called two pass assemblers.
- Constants for which assembler automatically reserves memory is called Literals.

10. GLOSSARY

Assembler: A computer program that translates programs [source code files] written in assembly language into their machine language equivalents [object code files].

Forward Reference: Is a label of the variable name

Literals: Constants for which assembler automatically reserves memory

Location Counter: 'LOCCTR' is the data structure used to keep the address of the next memory word.

Opcode: Another name for Mnemonic operation codes

Pseudo instructions: Instructions in the source program are used to instruct the assembler to perform certain actions during the translation of a program.

Single Pass Assemblers: Translation of assembly language program into object program is done in only one pass. The source program is read only once. Also called as 'one pass assembler'

11. TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Q1. Explain briefly about Assembler directives. Q2. Describe Forward Reference.

Q3. Explain about different types of Assemblers.

Q4. Explain about different data structures of Assembler.

Q5. Describe the design of One Pass and Two Pass Assembler.

12. ANSWERS

SELF ASSESSMENT QUESTIONS

1. Assembler
2. Loader
3. Assembler directives
4. EQU
5. Forward Reference
6. Pass
7. Mnemonic Table
8. Pseudo Instruction Table
9. Load and Go Assembler
10. Analysis phase
11. Location counter, mnemonics table

TERMINAL QUESTIONS

SHORT ANSWER QUESTIONS

Answer 1: These are the instructions present in the source program used to instruct the assembler to perform certain actions during the translation of a program. These are not translated into machine language instructions. (Refer section 3 for detail)

Answer 2: Forward Reference is a reference of a label (variable name), which is defined later in the program. (Refer section 4)

Answer 3: There are three types of Assemblers, single pass, two pass and multi pass. (Refer section 5)

Answer 4: Different data structures are used in the passes of the Assembler. (Refer section 6)

Answer 5: A one pass assembler scans the program just once. (Refer section 7) and the design of two pass assemblers involves the design of Pass 1 and Pass 2. (Refer section 8)

13. SUGGESTED BOOKS AND E-REFERENCES

- Dhamdhare (2002). Systems programming and operating systems Tata McGraw-Hill.
- M. Joseph (2007). System software, Firewall Media.