PAGE NO.
DATE : / /

Qu. 1

Assembly language statements.
1) Impeuative statements
2) Declauation statements
3) Assemblee diuective.

1) Impeuative statements

It indicates an action to be peufoumed duuing the execution of the assembly peoguam. Each impeuative statements typically tuanslates into one machine instuuction.

2) Declauation statements

Syntax of declauation statements :

[label] DS <constant>
[label DC '<value>'

It is used foe declauation puupose.

→ The DS (sauft foe declaue stauage) statement uesemues aueas of memoey and associates names with them.

→ The DC (sauft of declaue constant) statement constauds memoey wouds contauning constauints.

The statement :-

ONE DC '1'

3) Assemblee diuective

Assemblee diuective instauct the assemblee to peufoum ceutain actions duuing the assembly of a peoguam.

PAGE NO.:
DATE: / /

some assembler directive are described as follows,

START <constant>

This directive 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>]

This directive indicates the end of the source program. The optional <operandspec> indicates the address of the instruction where the execution of program should begin.

Qu 2 Explain data structure used by assembler.

Qu. 3 Discuss the pass structure of assemblers. How is the problem of forward reference resolved in single pass and two pass translations?

→

There are two types of assembly scheme to design an assembler

1) Single pass assembler
2) Two pass assembler

(1) Single Pass assembler

+) Single pass assembler scans the input file only once. During that single pass, the assembler handles both labels definitions and assembly.

2) It is generally faster than two pass assembler

3) Single pass assembler performs constructs

PAGE NO.:
DATE: / /

symbol table, literal table and also used mnemonic table and operation table.

4) The main problem encountered in one pass assembler is that of a forward reference.

5) The problem of forward reference is handled by single pass assembler by using a process called backpatching.

6) Backpatching is a process in which the operand field of an instruction containing a forward reference is left blank initially.

7) The address of the forward referenced symbol is put into this field when its definition is encountered in program.

8) In order to perform backpatching, single pass assembler requires an additional data structure called table of incomplete instructions (TII)

Syntax: (<Instruction address>, <symbol>)

(2) Two pass assembler

1) The two pass assembler scans the input assembly language program twice.

2) These scans are normally called Pass I and Pass II.

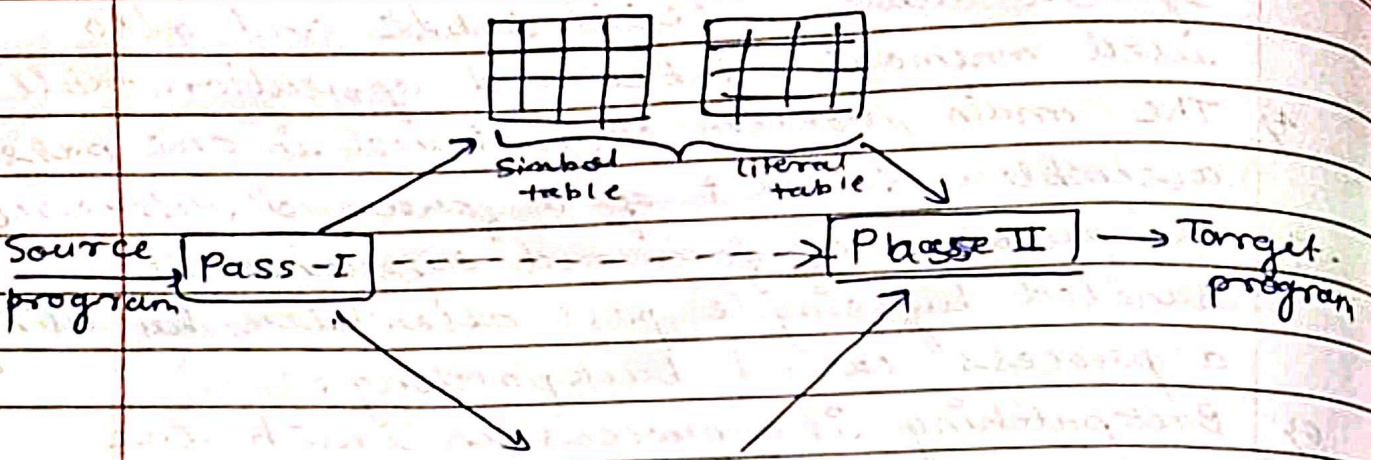
3) Pass I performs analysis of source program and Pass II assembler performs synthesis of target program.

4) Two pass assembler can handle forward reference easily.

5) Pass I generates symbol table, literal table and location counter and Pass II uses all these datastructures to generate machine code

PAGE NO.:
DATE: / /

Datastructure



⟶ Datastructure access

- - - → Control transfer.

Tasks performed by two pass assembler.

Pass - I

1) Separate the symbol, mnemonic opcode and operand fields.
2) Build the symbol table and literal table.
3) Perform location counter processing.
4) Construct intermediate representation (IC)

Pass - II

1) Synthesize the target program
2) Evaluate fields and generate code.
3) Process pseudo-opcode.

PAGE NO.:
DATE: / /

Qu. 4 List and explain advanced assembler directives with examples.

Ans:- 1) ORIGIN
2) EQU
3) LTORG

(1) ORIGIN

Syntax :- ORIGIN <address spec>
where <address spec> is an <operand spec> or <constant>. This directive indicates that location counter should be set to the address given by <address spec>. The ability to use an <operand spec> in the origin statement provides the ability to perform LC processing in a relative rather than absolute manner.

Example :-

target program. The design details of assembler passes are discussed after introducing advanced assembler directives and their influence on LC processing.

4.4.1 Advanced Assembler Directives

ORIGIN

The syntax of this directive is

```
ORIGIN    <address spec>
```

where <*address spec*> is an <*operand spec*> or <*constant*>. This directive indicates that LC should be set to the address given by <*address spec*>. The ORIGIN statement is useful when the target program does not consist of consecutive memory words. The ability to use an <*operand spec*> in the ORIGIN statement provides the ability to perform LC processing in a *relative* rather than *absolute* manner. Example 4.1 illustrates the differences between the two.

Example 4.1 Statement number 18 of Fig. 4.8(a), viz. `ORIGIN LOOP+2`, sets LC to the value 204, since the symbol LOOP is associated with the address 202. The next statement, viz.

```
MULT      CREG, B
```

is therefore given the address 204. The statement `ORIGIN LAST+1` sets LC to address 217. Note that an equivalent effect could have been achieved by using the statements `ORIGIN 202` and `ORIGIN 217` at these two places in the program, however the absolute addresses used in these statements would need to be changed if the address specification in the START statement is changed.

EQU

The EQU statement has the syntax

```
<symbol>  EQU       <address spec>
```

where <*address spec*> is an <*operand spec*> or <*constant*>.

The EQU statement defines the symbol to represent <*address spec*>. This differs from the DC/DS statement as no LC processing is implied. Thus EQU simply associates the name <*symbol*> with <*address spec*>.

Example 4.2 Statement 22 of Fig. 4.8(a), viz. `BACK EQU LOOP` introduces the symbol BACK to represent the operand LOOP. This is how the 16^{th} statement, viz.

```
BC        LT, BACK
```

is assembled as '+ 07 1 202'.

```
1          START   200
2          MOVER   AREG, ='5'      200)  +04 1 211
3          MOVEM   AREG, A         201)  +05 1 217
4   LOOP   MOVER   AREG, A         202)  +04 1 217
5          MOVER   CREG, B         203)  +05 3 218
6          ADD     CREG, ='1'      204)  +01 3 212
7          ...

12         BC      ANY, NEXT       210)  +07 6 214
13         LTORG
                   ='5'            211)  +00 0 005
                   ='1'            212)  +00 0 001
14         ...
15  NEXT   SUB     AREG, ='1'      214)  +02 1 219
16         BC      LT, BACK        215)  +07 1 202
17  LAST   STOP                    216)  +00 0 000
18         ORIGIN  LOOP+2
19         MULT    CREG, B         204)  +03 3 218
20         ORIGIN  LAST+1
21  A      DS      1               217)
22  BACK   EQU     LOOP
23  B      DS      1               218)
24         END
25                 ='1'            219)  +00 0 001
```

Fig. 4.8 An assembly program illustrating ORIGIN

LTORG

Fig. 4.4 has shown how literals can be handled in two steps. First, the literal is treated as if it is a <*value*> in a DC statement, i.e. a memory word containing the value of the literal is formed. Second, this memory word is used as the operand in place of the literal. Where should the assembler place the word corresponding to the literal? Obviously, it should be placed such that control never reaches it during the execution of a program. The LTORG statement permits a programmer to specify where literals should be placed. By default, assembler places the literals after the END statement.

At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a *literal pool*. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

Example 4.3 In Fig. 4.8, the literals ='5' and ='1' are added to the literal pool in statements 2 and 6, respectively. the first LTORG statement (statement number 13) allocates the addresses 211 and 212 to the values '5' and '1'. A new literal pool is now started. The value '1' is put into this pool in statement 15. This value is allocated the address 219 while processing the END statement. The literal ='1' used in statement 15 therefore refers to location 219 of the second pool of literals rather than location 212 of the first pool. Thus, all references to literals are forward references by definition.

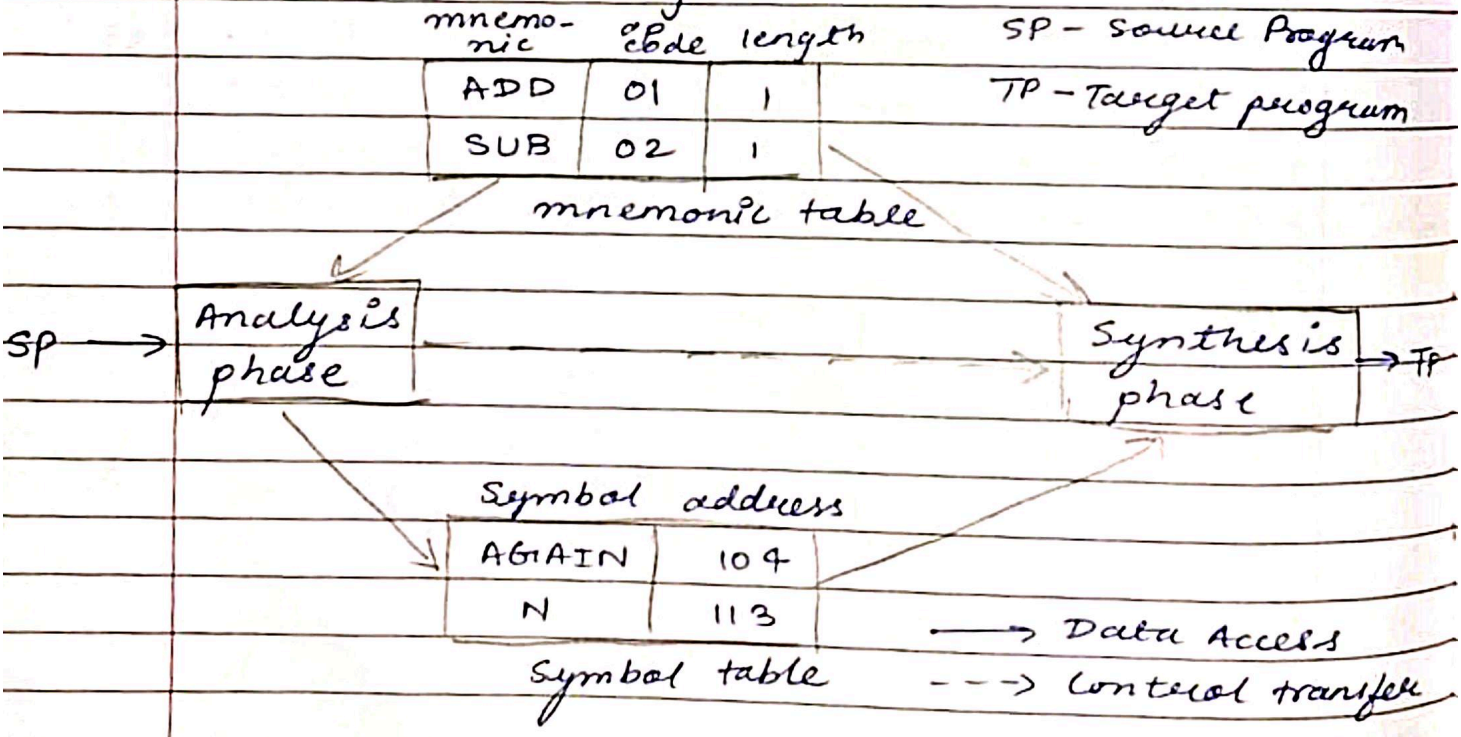
DATE: / /

Qu. 2 Explain the datastructures used by assembler?

Ans:- Data Structures :-

1) Symbol table
2) Mnemonic table.

Each entry of symbol table has two primary fields - name and address. The table is build by the analysis phase.
An entry in the mnemonics table has two primary fields - mnemonic and opcode. The synthesis ~~tab~~ phase uses these tables to obtain the machine address with which a name is associated, and the machine opcode corresponding to a mnemonic, respectively. Hence, the table have to be searched with the symbol name and the mnemonic as key.



~~Analysis phase :-~~

1) ~~Isolat~~

Qu. 5 Explain Pass-I of two pass assembler along with data structures used.

Ans :- Pass - I

1) Separate the symbol, mnemonic opcode and operand field.
2) Build the symbol table.
3) Perform location counter (LC) processing.
4) Construct intermediate representation.

Pass I performs analysis of the source program and synthesis of the intermediate representation while Pass II processes the intermediate representation to synthesize the target program

Pass - I uses the following data structures :-

1) OPTAB - A table of mnemonic opcodes and related information.
2) SYMTAB - Symbol Table
3) LITTAB - A Table of literals used in program.

OPTAB contains the field mnemonic opcode, class and mnemonic info. The class field indicates whether the opcode corresponds to an imperative statement, (IS) declaration statement (DL) or an assembler directive (AD)

The LTORG directive has very little relevance for the simple assembly language we have assumed so far. The need to allocate literals at intermediate points in the program rather than at the end is critically felt in a computer using a base displacement mode of addressing, e.g. computers of the IBM 360/370 family.

EXERCISE 4.4.1

1. An assembly program contains the statement

```
X       EQU     Y+25
```

   Indicate how the EQU statement can be processed if
   (a) Y is a back reference,
   (b) Y is a forward reference.
2. Can the operand expression in an ORIGIN statement contain forward references? If so, outline how the statement can be processed in a two pass assembly scheme.

4.4.2 Pass I of the Assembler

Pass I uses the following data structures:

OPTAB	A table of mnemonic opcodes and related information
SYMTAB	Symbol table
LITTAB	A table of literals used in the program

Figure 4.9 illustrates sample contents of these tables while processing the program of Fig. 4.8. OPTAB contains the fields *mnemonic opcode*, *class* and *mnemonic info*. The *class* field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD). If an imperative, the *mnemonic info* field contains the pair (*machine opcode*, *instruction length*), else it contains the id of a routine to handle the declaration or directive statement. A SYMTAB entry contains the fields *address* and *length*. A LITTAB entry contains the fields *literal* and *address*.

Processing of an assembly statement begins with the processing of its label field. If it contains a symbol, the symbol and the value in LC is copied into a new entry of SYMTAB. Thereafter, the functioning of Pass I centers around the interpretation of the OPTAB entry for the mnemonic. The *class* field of the entry is examined to determine whether the mnemonic belongs to the class of imperative, declaration or assembler directive statements. In the case of an imperative statement, the length of the machine instruction is simply added to the LC. The length is also entered in the SYMTAB entry of the symbol (if any) defined in the statement. This completes the processing of the statement.

For a declaration or assembler directive statement, the routine mentioned in the *mnemonic info* field is called to perform appropriate processing of the statement. For example, in the case of a DS statement, routine R#7 would be called. This routine

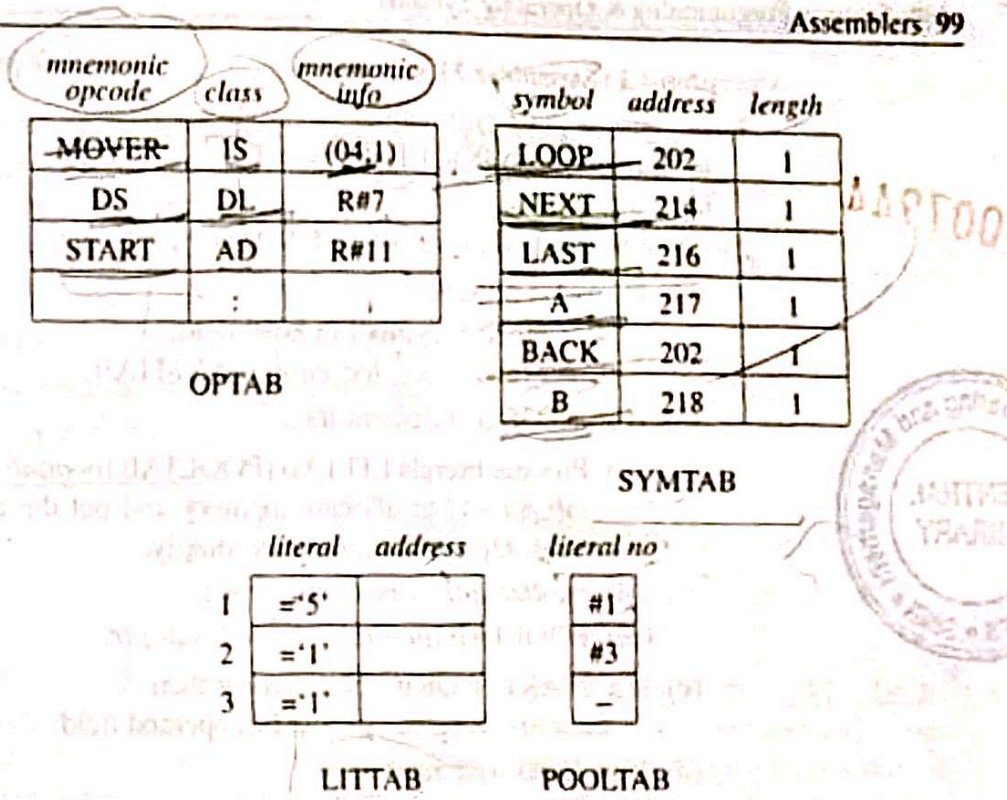


Fig. 4.9 Data structures of assembler Pass I

processes the operand field of the statement to determine the amount of memory required by this statement and appropriately updates the LC and the SYMTAB entry of the symbol (if any) defined in the statement. Similarly, for an assembler directive the called routine would perform appropriate processing, possibly affecting the value in LC.

The use of LITTAB needs some explanation. The first pass uses LITTAB to collect all literals used in a program. Awareness of different literal pools is maintained using the auxiliary table POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, the current literal pool is the last pool in LITTAB. On encountering an LTORG statement (or the END statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented. Thus, the literals of the program in Fig. 4.8(a) will be allocated memory in two steps. At the LTORG statement, the first two literals will be allocated the addresses 211 and 212. At the END statement, the third literal will be allocated address 219.

We now present the algorithm for the first pass of the assembler. Intermediate code forms for use in a two pass assembler are discussed in the next section.

Q6.

PAGE NO.:

DATE : / /

Qu-6 Discuss two variants of intermediate code in assemblers and compare them. 19-20