Module 2

Assembly Language Programming and Assemblers

SIC AND SIC/XE PROGRAMMING EXAMPLES

Ques 1) Assume that 100 words of data are stored from LOC1. Write a SIC program to copy these words to another location in memory starting from LOC2.

Ans:

```
        LDT   #31          initialise register T to 31
        LDX   #0           initialise index register to 0
MOVECH  LDCH  LOC1, X      load char from LOC1 to reg A
        STCH  LOC2, X      store char into LOC2
        TIXR  T            add 1 to index, compare to 31
        JLT   MOVECH       loop if "less than" 31
        :
LOC1    BYTE  C '100 Words Data'
LOC2    RESB  31
```

Ques 2) Write sequence of instruction for SIC to copy from STR1 to STR2.

Ans: Let the string is "TEST STRING". The length of this string is 11 including blank character between the two words "TEST" and "STRING". The string alongwith length can be defined as shown below:

```
STR1    BYTE    C'TEST STRING'
LEN     WORD    11
```

The above string should be copied into another string say STR2. So, size of string STR2 must be 11 and can be declared as shown below:

```
STR2    RESB  11
COPY    START 1000
FIRST   STL   RETADR  Store the return address which is in L
        LDX   ZERO    X = 0
L1      LDCH  STR1,X  L1:    A = STR1[X]
        STCH  STR2, X STR2[X] = A
        TIX   LEN     X = X+ 1
        JLT   L1      IF (X < LENG) goto L1
        LDL   RETADR  Get the return address into L
        RSUB          Return to OS
STR1    BYTE  C'TEST  STRING'
LEN     WORD  11
STR2    RESB  11
ZERO    WORD  0
RETADR  RESW  1       To store the return address
        END   FIRST
```

Ques 3) Write a program for a SIC/XE machine to copy a string "master of computer applications" from LOC1 and to LOC2.

Ans:

```
        LDT   #31          initialise register T to 31
        LDX   #0           initialise index register to 0
MOVECH  LDCH  LOC1, X      load char from LOC1 to reg A
        STCH  LOC2, X      store char into LOC2
        TIXR  T            add 1 to index, compare to 31
        JLT   MOVECH       loop if "less than" 31
        :
LOC1    BYTE  C 'master of computer applications'
LOC2    RESB  31
```

Ques 4) Suppose that ALPHA is an array of 100 words. Write a sequence of instruction for SIC/XE to set all 100 elements of the array to 1. Use immediate addressing and register to register instructions to make process as efficient as possible.

Or

Let NUMBER be an array of 100 words. Write a sequence of instructions for SIC to set all 100 elements of the array to 1 **(2017 [05])**

Or

Write an SIC/XE program to add the elements of an array ALPHA of 100 words and store the result in GAMMA. **(2020[04])**

Ans:

```
        LDS   # 3
        LDT   # 300
        LDX   # 1
ADDLP   LDA   ALPHA,X
        ADD   BETA,X
        STA   GAMMA,X
        ADDR  S,X
        COMPR X,T
        JLT   ADDLP
        :
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100
```

Ques 5) Write a sequence of instructions for SIC to ALPHA equal to the product of BETA and GAMMA. Assume that ALPHA, BETA and GAMMA.

Ans: Assembly Code

```
        LDA   BETA
        MUL   GAMMA
        STA   ALPHA
        :
        :
ALPHA   RESW  1
BETA    RESW  1
GAMMA   RESW  1
```

Ques 6) Write a sequence of instructions for SIC/XE to set ALPHA equal to 4 * BETA – 9. Assume that ALPHA and BETA are defined as in program. Use immediate addressing for the constants.

Ans: Assembly Code

```
        LDA     BETA
        LDS     #4
        MULR    S,A
        SUB     #9
        STA     ALPHA
        :
        :
ALPHA   RESW    1
```

Ques 7) Write SIC instructions to swap the values of ALPHA and BETA.

Ans: Assembly Code

```
        LDA     ALPHA
        STA     GAMMA
        LDA     BETA
        STA     ALPHA
        LDA     GAMMA
        STA     BETA
        :
        :
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
```

Ques 8) Write a sequence of instructions for SIC to set ALPHA equal to the integer portion of BETA ÷ GAMMA. Assume that ALPHA and BETA.

Ans: Assembly Code

```
        LDA     BETA
        DIV     GAMMA
        STA     ALPHA
        :
        :
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
```

Ques 9) Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the integer portion of the quotient and DELTA to the remainder. Use register-to-register instructions to make the calculation as efficient as possible.

Or

Write a sequence of instruction for SIC/XE to divide BETA by GAMMA and to store list integer quotient in ALPHA and remainder in DELTA.

(2017 [03])

Ans: Assembly Code

```
        LDA     BETA
        LDS     GAMMA
        DIVR    S, A
        STA     ALPHA
        MULR    S, A
        LDS     BETA
        SUBR    A, S
        STS     DELTA
        :
        :
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
```

Ques 10) Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the value of the quotient, rounded to the nearest integer. Use register-to-register instructions to make the calculation as efficient as possible.

Ans: Assembly Code

```
        LDA     BETA
        DIVF    GAMMA
        FIX
        STA     ALPHA
        :
        :
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
```

Ques 11) Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks.

Ans: Assembly Code

```
        LDX     ZERO
LOOP    LDCH    BLANK
        STCH    STR1,X
        TIX     TWENTY
        JLT     LOOP
        :
        :
STR1    RESW    20
BLANK   BYTE    C' '
ZERO    WORD    0
TWENTY  WORD    20
```

Ques 12) 3Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

Ans: Assembly Code

```
        LDT     #20
        LDX     #0
LOOP    LDCH    #0
        STCH    STR1,X
        TIXR    T
        JLT     LOOP
        :
        :
STR1    RESW    20
```

Ques 13) Suppose that ALPHA is an array of 100 words, as defined in program 7. Write a sequence of instructions for SIC to set all 100 elements of the array to 0.

Ans: Assembly Code

```
        LDA     ZERO
        STA     INDEX
LOOP    LDX     INDEX
```

```
        LDA     ZERO
        STA     ALPHA, X
        LDA     INDEX
        ADD     THREE
        STA     INDEX
        COMP    K300
        TIX     TWENTY
        JLT     LOOP
        :
        :
INDEX   RESW    1
ALPHA   RESW    100
        :
ZERO    WORD    0
K300    WORD    100
THREE   WORD    3
```

Ques 14) Suppose that ALPHA is an array of 100 words, as defined in program 7. Write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

Ans: Assembly Code

```
        LDS     #3
        LDT     #300
        LDX     #0
LOOP    LDA     #0
        STA     ALPHA, X
        ADDR    S, X
        COMPR   X, T
        JLT     LOOP
        :
        :
ALPHA   RESW    100
```

Ques 15) Suppose that ALPHA is an array of 100 words. Write a sequence of instruction for SIC/XE to arrange the 100 words in ascending order and store result in an array BETA of 100 elements.

Ans: Assembly Code

```
SORT    START   0
OUTER   LDX     INDEX
        LDS     ARR1,X
        LDX     #0
INNER   LDT     ARR1,X
        COMPR   S,T
        JLT     LOOP
        JEQ     LOOP
        RMO     S,A
        RMO     T,S
        RMO     A,T
        RMO     X,A
        LDX     INDEX
        STS     ARR1,X
        RMO     A,X
        STT     ARR1,X
LOOP    RMO     X,A
        ADD     #3
        COMP    LENGTH
        RMO A,X RMO A,X
        JLT     INNER
        LDA     INDEX
        ADD     #3
        COMP    LENGTH
        STA     INDEX
        JLT     OUTER
ARR1    RESW    10
LENGTH  WORD    30
INDEX   WORD    0
```

Ques 16) Suppose that ALPHA and BETA are the two arrays of 100 words. Another array of GAMMA elements are obtained by multiplying the corresponding ALPHA element by 4 and adding the corresponding BETA elements.

Ans: Assembly Code

```
        LDS     #3
        LDT     #300
        LDX     #0
ADDLOOP LDA     ALPHA, X
        MUL     #4
        ADD     BETA, X
        STA     GAMMA, X
        ADDR    S, X
        COMPR   X, T
        JLT     ADDLOOP
        :
        :
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
        LDS     #3
        LDT     #300
        LDX     #0
ADDLOOP LDA     ALPHA, X
        MUL     #4
```

Ques 17) Suppose that ALPHA is an array of 100 words. Write a sequence of instructions for SIC/XE to find the maximum element in the array and store results in MAX.

Ans: Assembly Code

```
        LDS     #3
        LDT     #300
        LDX     #0
CLOOP   LDA     ALPHA, X
        COMP    MAX
        JLT     NOCH
        STA     MAX
NOCH    ADDR    S, X
        COMPR   X, T
        JLT     CLOOP
        :
        :
ALPHA   RESW    100
MAX     WORD    -32768
```

Ques 18) Suppose that RECORD contains a 100-byte record. Write a subroutine for SIC that will write this record on to device 05.

Ans: Assembly Code

```
        JSUB    WRREC
        :
```

```
            :
WRREC       LDX     ZERO
WLOOP       TD      OUTPUT
            JEQ     WLOOP
            LDCH    RECORD, X
            WD      OUTPUT
            TIX     LENGTH
            JLT     WLOOP
            RSUB
            :
            :
ZERO        WORD    0
LENGTH      WORD    1
OUTPUT      BYTE    X '05'
RECORD      RESB    100
```

Ques 19) Write a subroutine for SIC that will read a record into a buffer, as in program 10. The record may be any length from 1 to 100 bytes. The end of record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH.

Ans: Assembly Code

```
            JSUB    RDREC
            :
            :
RDREC       LDX     ZERO
RLOOP       TD      INDEV
            JEQ     RLOOP
            RD      INDEV
            COMP    NULL
            JEQ     EXIT
            STCH    BUFFER, X
            TIX     K100
            JLT     RLOOP
EXIT        STX     LENGTH
            RSUB
            :
            :
ZERO        WORD    0
NULL        WORD    0
K100        WORD    1
INDEV       BYTE    X 'F1'
LENGTH      RESW    1
BUFFER      RESB    100
```

Ques 20) Write a subroutine for SIC/XE that will read a record into a buffer, as in program 10. The record may be any length from 1 to 100 bytes. The end of record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

Ans: Assembly Code

```
            JSUB    RDREC
            :
            :
RDREC       LDX     #0
            LDT     #100
            LDS     #0
RLOOP       TD      INDEV
            JEQ     RLOOP
            RD      INDEV
            COMPR   A, S
            JEQ     EXIT
            STCH    BUFFER, X
            TIXR    T
            JLT     RLOOP
EXIR STX    STX     LENGTH
            RSUB
            :
            :
INDEV       BYTE    X 'F1'
LENGTH      RESW    1
BUFFER      RESB    100
```

Ques 21) Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks.

Ans: Assembly Code

```
            LDX     ZERO
LOOP        LDCH    BLANK
            STCH    STR1,X
            TIX     TWENTY
            JLT     LOOP
STR1        RESW    20
BLANK       BYTE    C' '
ZERO        WORD    0
TWENTY      WORD    20
```

Ques 22) Write a sequence of instructions for SIC to set ALPHA = BETA*9+GAMMA. **(2018[03])**

Ans: Sequence of Instructions for SIC

```
LDA     BETA
LDS     #9
MUL     S,A
ADD     GAMMA
STA     ALPHA
  :
  :
ALPHA   RESWL
```

Ques 23) Let A,B and C are arrays of 10 word each. Write a SIC/XE program to add the corresponding elements of A and B and store the result in C.

(2018[06])

Ans:

```
LDS     #3
LDT     #30
LDX     #0
LDA     A, X
ADDLP
ADD     B, X
STA     C, X
ADDR    S, X
COMPR   S, T
ILT     ADDLP
RESW    10
RESW    10
RESW    10
```

ASSEMBLER

Ques 24) What is assembler? Define basic assembler functions and its phases.

Ans: Assembler

Assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to the assembler is a source code written in assembly language (using mnemonics) and the output is the object code. The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

Basic Functions of Assembler

1) Translate mnemonic opcodes to machine language.
2) Convert symbolic operands to their machine addresses.
3) Build machine instructions in the proper format
4) Convert data constants into machine representation.
5) Error checking is provided.
6) Changes can be quickly and easily incorporated with a reassembly.
7) Variables are represented by symbolic names, not as memory locations.

Phase in Assemblers

These steps are summarized in two different stages or phases, namely:

1) Analysis Phase
2) Synthesis Phase

The following **table 2.1** lists the details for each phase of an assembler:

Table 2.1: Phases of an Assembler

Analysis Phase	Synthesis Phase
Build the Symbol Table.	Look at the Mnemonics Table and get the opcode corresponding to the mnemonic.
Parse and separate labels, opcodes and operand fields in a statement.	Obtain the address of a memory operand from the Symbol Table.
Perform memory allocation.	Synthesize the machine instruction.
Check correctness of opcodes by looking at the contents of the mnemonics table.	
Update contents of Location Counter based on the length of each instruction.	

Figure 2.1 illustrates the usage during each phase of assembly:

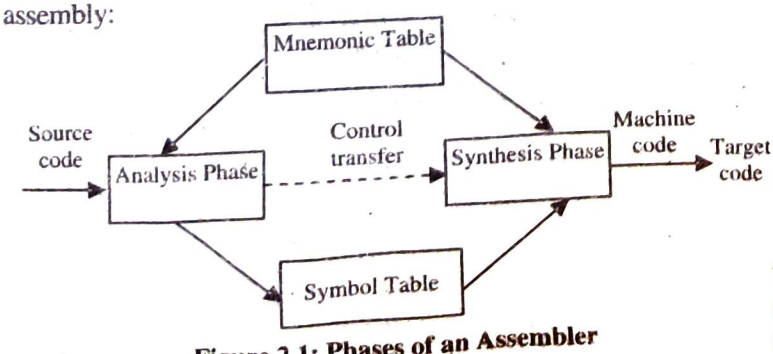


Figure 2.1: Phases of an Assembler

Ques 25) Describe the assembler output format. *Or* Explain the object program format of assembler. *Or* Briefly describe the format of object program generated by SIC assembler. (2019[03])

Ans: Assembler Output Format/Object Program Format

The simple object program format contains three types of records:

1) **Header Record:** The Header record contains the program name, starting address, and length.

The format for header record is as follows:

Col 1	H
Col 2-7	Program name
Col 8-13	Starting address of object program (hexadecimal)
Col 14-19	Length of object program in bytes (hexadecimal)

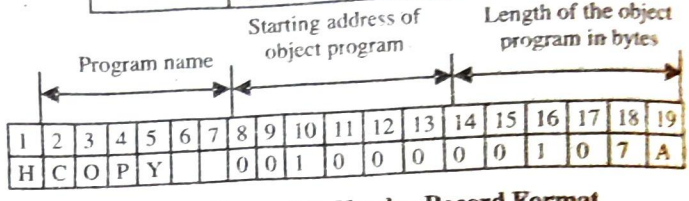


Figure 2.2: Header Record Format

2) **Text Record:** Text records contain the translated (i.e., machine code) instructions and data of the program, together with an indication of the addresses where these are to be loaded.

The format for text record is as follows:

Col 1	T
Col 2-7	Starting address of object code in record.
Col 8-9	Length of object code in bytes (hexadecimal).
Col 10-69	Object code representation in hexadecimal.

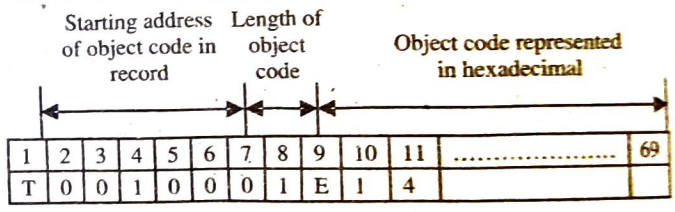


Figure 2.3: Text Record Format

3) **End Record:** The End record marks the end of the object program and specifies the address in the program where execution is to begin. (This is taken from the operand of the program's END statement. If no operand is specified, the address of the first executable instruction is used.) The format for end record is as follows:

Col 1	E
Col 2-7	Address of first executable instruction in object program (hexadecimal)

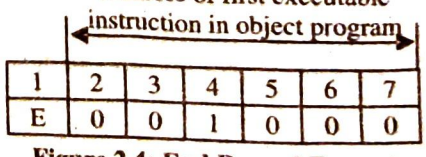


Figure 2.4: End Record Format

The details of the formats (column numbers, etc.) are arbitrary; however, the information contained in these records must be present (in some form) in the object program.

Ques 26) Describe the various types of assemblers.

Or

Define single-pass and two-pass assembler.

Or

Explain the concept of single pass assembler with a suitable example - (2017 [05])

Or

Describe the functions of two passes of a simple two pass assembler. (2020[03])

Ans: Types of Assemblers

Two types of assemblers are:

1) **Single-Pass Assembler:** As its name implies, this assembler reads the source file once. During that single pass, the assembler handles both label definitions and assembly. In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference.

   One-pass assemblers are used when:
   i) It is necessary or desirable to avoid a second pass over the source program
   ii) The external storage for the intermediate file between two passes is slow or is inconvenient to use

2) **Two Pass Assembler:** A two-pass assembler resolves the forward references and then converts into the object code. For a two pass assembler, forward references in symbol definition are not allowed. Symbol definition must be completed in pass 1.

   Such an assembler performs two passes over the source file. In the first pass it reads the entire source file, looking only for label definitions. All labels are collected, assigned values, and placed in the symbol table in this pass. No instructions are assembled and, at the end of the pass, the symbol table should contain all the labels defined in the program. In the second pass, the instructions are again read and are assembled, using the symbol table.

   Hence the **process/functions** of the two-pass assembler can be as follows:

   Pass 1: (Define Symbols)
   i) Assign addresses to all statements in program.
   ii) Save the values assigned to all labels for use in pass 2.
   iii) Perform some processing of assembler directives.

   Note: Perform analysis of assembly language program.

   Pass 2: (Assemble Instructions and Generate Object Code)
   i) Assembler instructions.
   ii) Generate data values defined by BYTE, WORD, etc.
   iii) Perform processing of assembler directives not done during pass 1.
   iv) Write object program and assembly listing.

   Note: Processes intermediate representation and produces target program.

Ques 27) Describe the assembler data structure.

Or

What are the uses of OPTAB and SYMTAB during the assembling process? Specify the uses of each during pass 1 and pass 2 of a two pass assembler. (2019[03])

Ans: Assembler Data Structures

The data structures used by an assembler are as follows:

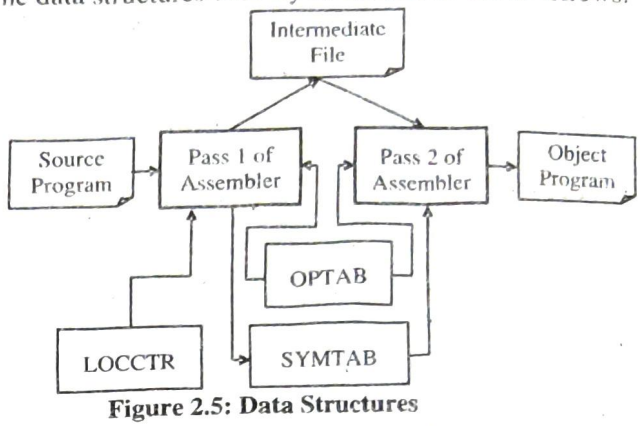


Figure 2.5: Data Structures

1) **Operation Code Table (OPTAB):** It is used to look up mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

   In **pass 1**, the OPTAB is used to look up and validate the operation code in the source program.

   In **pass 2**, it is used to translate the operation codes to machine language.

   In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, one must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR. In pass 2 one takes the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction. OPTAB is usually organised as a hash table, with mnemonic operation code as the key. The hash table organisation is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table – i.e., entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored. The following table shows the example of OPTAB.

Mnemonic Opcode	Machine Opcode	Class
READ	09	I (Imperative)
ADD	01	I (Imperative)
LOAD	04	I (Imperative)
SUB	02	I (Imperative)
START	-	III (Assembler Directive)
END	-	III (Assembler Directive)
DS	-	II (Declarative)
DC	-	II (Declarative)

2) **Symbol Table (SYMTAB):** This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).

In **pass 1**, labels are entered into the symbol table alongwith their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

In **pass 2**, symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.

SYMTAB is usually organised as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimisation.

Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2. A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2.

Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used.

This avoids need to repeat many of the table-searching operations. The example of SYMTAB is shown **table** below.

Symbol Name	Address	Length	Other Information
A	100	1	-
B	101	2	-

3) **LOCCTR:** Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses.

LOCCTR is initialised to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

Ques 28) Give the data structures with their purpose used in the SIC assembler. ***Or***
Describe the data structures used in the two pass SIC assembler algorithm. (2017 [03])

Or

Describe the data structures used by a simple two pass assembler. (2020[05])

Ans: Data Structures Used in the SIC Assembler
Pass I
Data structure used in pass I are as follows:

Data structure	Purpose
User Program	The source.
Location Counter	To hold the address of the next instruction to be executed.
Operation Code Table	Holds the mnemonic and machine opcode for each instruction along with length and type.
Pseudo Optable	Hold the pseudo instruction and the corresponding action to be taken.
Symbol Table	Stores all symbols encountered in the program along with memory address.
Literal Table	Stores all literals in the program along with memory address.

Pass II
Data structures used in pass II are as follows:

Data structure	Purpose
Intermediate Code	For further processing and assembly code.
Location Counter	To hold address of next instruction
Operation Code Table	Holds the mnemonic machine opcode for each instruction along with length and type.
Symbol Table	Create by pass I to store all symbols in the program and its associated value.
Base Table	To indicate which registers are specified as base registers by the USE directive and their content.

Ques 29) Explain the use of OPTAB, SYMTAB and LITTAB in the Assembler Pass I using suitable example.

Ans: Use of OPTAB
OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalents.

Use of SYMTAB
SYMTAB is used to store values assigned to labels.

Use of LITTAB
The basic data structure needed is a literal table LITTAB. For each literal used, this table contains the literal name, the operand value and length, and the address assigned to the operand when it is placed in a literal pool. LITTAB is often organised as a hash table, using the literal name or value as the key.

Example: As each literal operand is recognized during Pass 1, the assembler searches LITTAB for the specified literal name (or value). If the literal is already present in the table, no action is needed; if it is not present, the literal is added to LITTAB (leaving the address unassigned).

```
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
```

Figure 2.6: Assembly Program Illustrating the ORIGIN Directive

Table 2.2 illustrates contents of these tables after processing the END statement of the program of **figure 2.6**:

Table 2.2-a: OPTAB

mnemonic opcode	class	mnemonic info
MOVER	IS	(04, 1)
DS	DL	R#7
START	AD	R#11
	:	

Table 2.2-b: SYMTAB

symbol	address	length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

Table 2.2-c: LITTAB

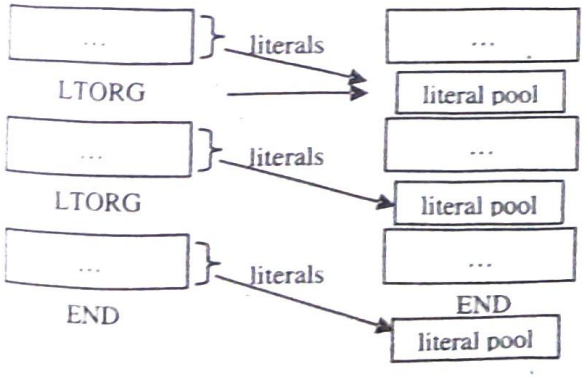
	value	address
1	='5'	
2	='1'	
3	='1'	

Ques 30) **How does the assembler handle literals?**

Or

Define LITTAB. Also write the difference between the SYMTAB and LITTAB

Ans: **Handling of Literals by Assemblers**



The assembler handles literals in such a way that a Literal Table (LITTAB) is used. Literals are constants written in the operand of instructions. This avoids having to define the constant elsewhere in the program and make-up a label for it.

All of the literal operands used in a program are gathered together into one or more **literal pools**. Normally, literals are placed into a pool **at the end of the program**. The assembly listing of a program containing literals usually includes a listing of this literal pool. The **assembler directive LTORG** tells the assembler generate a literal pool here. LTORG allows to place literals into a pool at some other location in the object program. When the assembler encounters an LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG. This literal pool is placed in the object program at the location where the LTORG directive was encountered.

Literal Table (LITTAB)

For each used literal, LITTAB contains:

1) The literal name,
2) The operand value and length,
3) The address assigned to the operand when it is placed in a literal pool.

LITTAB is often organised as a hash table, using the literal name or value as the key.

Creation of LITTAB

In Pass 1

1) The assembler searches LITTAB for the specified literal name (or value) because each literal operand is recognised during Pass 1.
2) If the literal is already present in the table, no action is needed.
3) If the literal is not present, the literal is added to LITTAB (leaving the address unassigned).
4) When Pass 1 encounters an LTORG statement or the end of the program, the assembler makes a scan of the literal table. Each literal currently in the table is assigned an address (unless such an address has already been filled in).
5) As these addresses are assigned, the **location counter** is updated to reflect the number of bytes occupied by each literal.

In Pass 2

1) The **operand address** for use in generating object code is obtained by searching LITTAB for each literal operand encountered.

The following **figure** shows the **difference between the SYMTAB and LITTAB**:

SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062

LITTAB

Literal	Hex Value	Length	Address
C 'EOF'	454F46	3	002D
X '05'	05	1	1076

2) The data values specified by the literal in each literal pool are inserted at the appropriate places in the object program.

3) If a literal value represents an address in the program (e.g., location counter value), the assembler must also generate the appropriate **Modification record**.

Ques 31) Given the following assembly program, show the contents of all the tables generated after pass I.

```
        START   300
B       DC      3
        MOVER   AREG, ='5'
        MOVEM   AREG, A
L1      MOVER   CREG, B
        ADD     CREG, ='1'
        COMP CREG,  AREG
        BC      NE, NX
        LTORG
                ='5'
                ='1'
NX      SUB     AREG, ='1'
        COMP CREG,  AREG
        BC      LT, LS
LS      STOP
        ORIGIN  L1 + 2
        MULT CREG,  B
A       DS      1
        END
```

Ans:

1		START	300			
2	B	DC	3	300)		
3		MOVER	AREG, = '5'	301)	+04	307
4		MOVEM	AREG, A	302)	+05	313
5	L1	MOVER	CREG, B	303)	+04	304
6		ADD	CREG, = '1'	304)	+01	308
7		COMP CREG,	AREG	305)	+06	
8		BC	NE, NX	306)	+07	309
9		LTORG				
			= '5'	307)	+00	005
			= '1'	308)	+00	001
10	NX	SUB	AREG, = '1'	309)	+02	308
11		COMP CREG,	AREG	310)	+06	
12		BC	LT, LS	311)	+07	312
13	LS	STOP		312)	+00	000
14		ORIGIN	L1 + 2	–	–	–
15		MULT CREG,	B	304)	+03	304
16	A	DS	1	313)	–	–
17		END		–	–	–

Mnemonic Opcode	Class	Mnemonic Info
MOVER	IS	(04, 1)
DS	DL	R#7
START	AD	R#11

OPTAB

Symbol	Address
B	300
L1	303
NX	309
LS	312
A	313

SYMTAB

	Literal	Address
1	= '5'	
2	= '1'	
3		

LITTAB

Ques 32) Generate the object code for the below SIC/XE. Also show the contents of symbol table.

```
SUM     START   4000
        LDX     ≠0
        LDA     ≠0
        +LDT    ≠4000
        BASE    COUNT
LOOP    ADD     TABLE, X
        TIXR    T
        JLT     LOOP
        STA     TOTAL
        RSUB
TOTAL   RESW    1
TABLE   RESW    4000
COUNT   RESW    1
        END
```

OP codes
LDX – 04, LDA – 00, LDT – 74, ADD – 18, TIXR – B8, JLT – 38, STA – OC, RSUB – 4C.

Ans:

Loc Counter	Length	Label	Mnemonic	Operand	Opcode
		SUM	START	4000	
4000	3		LDX	≠0	04
4003	3		LDA	≠0	00
4006	3		+LDT	≠4000	74
			BASE	COUNT	
4009	2	LOOP	ADD	TABLE, X	18
400C	3		TIXR	T	B8
400F	3		JLT	LOOP	38
4012	3		STA	TOTAL	OC
4015	3		RSUB		4C
4018	3	TOTAL	RESW	1	
401B	3	TABLE	RESW	4000	
7788	3	COUNT	RESW	1	
778B	LOCCTR		END		

Generating object code for each instructions:

Line No.		
1		
2	11010100/00011000(clear=b4, x=10)	B410
3	000000/010000/0000 0000 0000	010000
4	011010/010001/05788	69105788
5		
6	000110/111010/00C	1BA00C
7	001011/110010/006	2F2006
8	001110/110010/FF8	3B2FF8
9	000011/110100/000	0F4000

The symbol table created by pass 1 of the assmber are as follows:

Symbol Table

Symbol	Address
LOOP	4009
TOTAL	4018
TABLE	401B
COUNT	7788
EXIT	778B

Ques 33) Define the various elements of assembly language programming.

Ans: Elements of Assembly Language Programming

An assembly language is the lowest level programming language for a computer. It is specific to a particular computer system, and hence machine-dependent. It provides certain features which makes programming much easier than in the machine language.

1) **Mnemonic Operation Codes:** Instead of using numeric operation codes (Opcodes), mnemonics are used. This eliminates the need for the programmer to remember all numeric opcodes.

The following **table 2.3** shows a list of instruction operation codes for a computer in assembly language.

Table 2.3: Instruction Operation Codes and Assembly of Mnemonics for a Hypothetical Computer

Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop the process.
01	ADD	First operand is assumed to be an accumulator.
02	SUB	First operand is assumed to be an accumulator.
03	MULT	First operand is assumed to be an accumulator.
04	LOAD	Load into the accumulator.
05	STORE	Store accumulator into the storage location.
06	TRANS	Transfer control to the address mentioned.
07	TRIM	Transfer only when the accumulator is lesser than 0.
08	DIV	Divide the accumulator by storage location contents.
09	READ	Read a card into the storage location.
10	PRINT	Print the storage location contents.
11	LIR	Load index register with last 3 digits of storage operand.
12	IIR	Increment index register with last 3 digits of storage operand.
13	LOOP	Used for iteration.

2) **Symbolic Operand Specification:** Symbolic names are associated with data or instructions. Hence, instead of using the actual instruction or data, the programmer can substitute these symbols.

3) **Declaration of Data/Storage Areas:** Data can be declared using the decimal notation. This eliminates the manual conversion of constants into their internal machine representation.

Ques 34) Describe the assembly language statement format.

Ans: Statement Format

1) Each line of a program is one of the following:
   i) An instruction
   ii) An assembler directive (or pseudo-op)
   iii) A comment
2) Whitespace (between symbols) and case are ignored.
3) Comments (beginning with ";") are also ignored.

An assembly language statement has the following format:

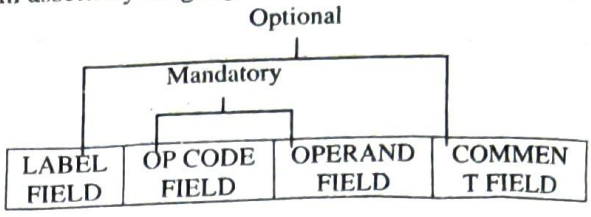


Figure 2.7: Assembly Language Program Statement Format

1) **Label Field:** It's placed at the beginning of the line. And assigns a symbolic name to the address corresponding to line. **For example,**

```
LOOP  ADD   R1,R1,#–1
BRp   LOOP
```

2) **Opcode Field:** Reserved symbols that correspond to LC-3 instructions. **For example,** ADD, AND, LD, LDR, ...

3) **Operand Field**
   i) **Registers:** Specified by Rn, where n is the register number.
   ii) **Numbers:** Indicated by # (decimal) or x (hex).
   iii) **Label:** Symbolic name of memory location.
   iv) Separated by comma.
   v) Number, order, and type correspond to instruction format.

For example,

```
ADD   R1,R1,R3
ADD   R1,R1,#3
LD    R6,NUMBER
BRz   LOOP
```

4) **Comment Field**
   i) Anything after a semicolon is a comment.
   ii) Ignored by assembler.
   iii) Used by humans to document/understand programs.
   iv) Tips for useful comments:
      a) Avoid restating the obvious, as "decrement R1"
      b) Provide additional insight, as in "accumulate product in R6"
      c) Use comments to separate pieces of program

Ques 35) Define the various types of assembly language statements.

Ans: Types of Assembly Language Statements

Assembly language supports the following three types of statements, which any assembly language program may contain:

1) **Imperative Statement:** An imperative assembly language statement indicates actions to be performed during the execution of the assembled program. Hence, each imperative statement translates into (generally one) machine instruction.

2) **Declarative Statements:** A declarative assembly language statement declares constants or storage areas in a program. **For example,** consider the following statements

```
A    DS    1
G    DS    200
```

The first statement declares a storage area of one word and gives it a name A. The second indicates that a block of two hundred words is to be reserved. When G is used as an operand, it stands for the first word of the specified block of storage of 200 words. The words of G can also be accessed through indexing, e.g., G(5) indicates that the content of index register 5 would determine which word of G is to be accessed.

Note: DS stands for **D**eclare **S**torage and

For **declaring a constant**, **D**eclare **C**onstant (DC) statement can be used. **For example,** consider the following statement

```
ONE  DC '1'
```

declares ONE to be the name associated with the constant 1. The programmer can declare constants in decimal, binary, hexadecimal, etc. Many assemblers permit the use of literals. These are essentially constants directly used in an operand field. Since the 'value' of a constant is known from the way it is written, the literals are also called self-defining terms. Use of a literal saves us from the difficulty of defining a constant through a DC statement.

3) **Assembler Directive Statements:** Assembler directive statements neither represent machine instructions to be included in the object program nor indicate the allocation of storage for constants or variables. Instead, these statements direct the assembler to perform some specific actions during the assembling process of the program.

Assembler directive statements neither represent machine instruction to be included in the object program nor indicate the allocation of storage for constants or variable. Instead, these statements direct the assembler to perform some specific action during the assembling process of the program.

For example, consider the following statement

```
START 202
```

Explanation: The first word of the object program generated by the assembler has to be placed in the memory address 202.

Ques 36) Write a note on hand assembly of SIC/XE program.

Ans: Hand Assembly of SIC/XE Program

One have to translate assembly language program into the machine language, either into hexadecimal or into binary numbers. One can translate an assembly language program by hand, instruction by instruction. This is called hand assembly. The following **table** illustrates the hand assembly of the addition program:

Instruction Mnemonic	Roister/Memory Location	Hexadecimal Equivalent
LDR	R1, num1	E59F1010
LDR	R0, num2	E59F0008
ADD	R5, R1, R0	E0815000
STR	R5, num3	E58F5008

Hand assembly is a rote task which is uninteresting, repetitive, and subject to numerous minor errors. Picking the wrong line, transposing digits, omitting instructions, and misreading the codes are only a few of the mistakes that you may make. Most microprocessors complicate the task even further by having instructions with different lengths. Some instructions are one word long while others may be two or three. Some instructions require data in the second and third words; others require memory addresses, register numbers, or who knows what?

Assembly is a rote task that we can assign to the microcomputer. The microcomputer never makes any mistakes when translating codes; it always knows how many words and what format each instruction requires. The program that does this job is an "assembler". The assembler program translates a user program, or "source" program written with mnemonics, into a machine language program, or 'object program, which the microcomputer can execute. The assembler's input is a source program and its output is an object program.

Assemblers have their own rules that you must learn. These include tile use of certain markers (such as spaces, commas, semicolons, or colons) in appropriate places, correct spelling, tile proper control of information, and perhaps even the correct placement of names and numbers. These rules are usually simple and can be learned quickly.

Translate (by hand) the following assembly program to SIC object code. (The output format will look like **figure 2.8**, which contains H record, T record, and E record.)

```
Loc       Source statement                Object code
          STRCPY  START   1000
          FIRST   LDX     ZERO
          MOVECH  LDCH    STR1,X
                  STCH    STR2,X
                  TIX     ELEVEN
                  JLT     MOVECH
          STR1    BYTE    C'TEST STRING'
          STR2    RESB    11
          ZERO    WORD    0
          ELEVEN  WORD    11
```

```
                END     FIRST
```

Loc		Source statement		Object code
1000	STRCPY	START	1000	
1000	FIRST	LDX	ZERO	041025
1003	MOVECH	LDCH	STR1,X	50900F
1006		STCH	STR2,X	54901A
1009		TIX	ELEVEN	2C1028
100C		JLT	MOVECH	381003
100F	STR1	BYTE	CTEST STRING3	544553...
101A	STR2	RESB	11	
1025	ZERO	WORD	0	000000
1028	ELEVEN	WORD	11	00000B
102B		END	FIRST	

Machine instruction

Code		op		Binary disp/address	Hex
LDCH	STR1,X	01010000	1	001 0000	50900
				0000 1111	F
STCH	STR2,X	01010100	1	001 0000	54901
				0001 1010	A

Ques 37) Translate (by hand) the following assembly program to SIC/XE object code.

Loc		Source	Statement	Object code
	STRCP2	START	1000	
	FIRST	LDT	#11	
		LDX	#0	
	MOVECH	LDCH	STR1,X	
		STCH	STR2,X	
		TIXR	T	
		JLT	MOVECH	
	STR1	BYTE	C'TEST STRING'	
	STR2	RESB	11	
		END	FIRST	

```
HSTRCP2001000000027
T001000117500B05000053A00857A010B8503B2FF5
T0010110B544553542053545249
E001000
```

Ans: Assembly Program to SIC/XE Object Code

Loc		Source statement		Object code
1000	STRCP2	START	1000	
1000	FIRST	LDT	#11	75000B
1003		LDX	#0	050000
1006	MOVECH	LDCH	STR1,X	53A008
1009		STCH	STR2,X	57A010
100C		TIXR	T	B850
100E		JLT	MOVECH	3B2FF5
1011	STR1	BYTE	C'TEST STRING'	544553...
101C	STR2	RESB	11	
1027		END	FIRST	

Machine instruction

Code		op	n i x b p e	Binary disp/address	Hex
LDCH	STR1,X	0101 00	1 1 1 0 1 0	0000 0000 1000	53A008
STCH	STR2,X	0101 01	1 1 1 0 1 0	0000 0001 0000	57A010
JLT	MOVECH	0011 10	1 1 0 0 1 0	1111 1111 0101	3B2FF5

```
HSTRCPY001000000002B
T0010000F041025050900F054901A2C1028381003
T00100F0B5445535420535453494E47
T0010250600000000000B
E001000
```

Figure 2.8