Module 3

Assembler Features and Design Options

MACHINE DEPENDENT ASSEMBLER FEATURES

Ques 1) What is Machine Dependent Assembler Features? What are the advantages of SIC/XE?

Ans: Machine Dependent Assembler Features

There are certain features that are dependent on the architecture of the underlying machine. A few of the machine-dependent features for SIC version of assembler, are as follows:

1) Instruction formats and addressing modes, and
2) Program relocation.

In SIC/XE version of assembler:

1) Immediate operands denoted with prefix #.
2) Indirect addressing – indicated by adding prefix @.
3) Instructions that refer memory are assembled using Program counter or base relative addressing mode.
4) The assembler directive BASE is used in conjunction with base relative addressing mode.

Advantages of SIC/XE

1) Register to register instruction used to improve execution speed of program.
2) It does not require another memory reference and hence it is faster.
3) No need to fetch immediate operand & it is part of instruction.
4) Provide room to load and run several programs at same time. This kind of sharing of the machine between programs is called "multiprogramming".
5) If the displacements required for both PC & BASE relative addressing are too large to fit into a 3-byte instruction, then the 4-byte extended format (format 4) must be used.

Ques 2) Explain the Instruction Formats and Addressing Modes with examples.

Or

Discuss the Program-Counter Relative and Base-Relative Addressing in detail.

Or

Write short notes on the following:

i) Immediate Addressing Mode

ii) Indirect Address Translation

Or

Give the difference between PC Relative Addressing and Base Relative Addressing.

Ans: Instruction Formats and Addressing Modes

Assembler must:

1) Convert mnemonic operation code to machine language (using OPTAB) and change each register mnemonic to its numeric equivalent (done during pass 2).
2) The conversion of register mnemonics to numbers can be done with the separate table. To do this, SYMTAB would be preloaded with the register names (A, x, etc.,) and their values (0, 1, etc.).

The instruction formats depend on the memory organisation and the size of the memory.

Instruction Format for SIC

In SIC machine the memory is byte addressable. Word size is 3 bytes, so the size of the memory 2^{12} bytes. Accordingly it supports only one instruction format. It has only two registers – **register A** and **Index register**. Therefore the addressing modes supported by this architecture are **direct**, **indirect** and **indexed**.

Instruction Format for SIC/XE

The memory of a SIC/XE machine is 2^{20} bytes (1MB). This supports four different types of instruction types, they are:

1) **Format 1 (1 Byte):** Contains only operation code.
2) **Format 2 (2 Bytes):** First eight bits for operation code, next four for register 1 and following four for register 2.

   The numbers for the registers go according to the numbers indicated at the registers section (i.e., register T is replaced by hex 5, F is replaced by hex 6).
3) **Format 3 (3 Bytes):** First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i).

   The flags in order are – n, i, x, b, p, and e.

   The last flag e indicates the instruction format:

   e = 0 for Format 3

   e = 1 for Format 4
4) **Format 4 (4 Bytes):** Same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.

Instructions can be:

1) **Translations for the Instruction involving Register-Register Addressing Mode:** During **pass 1** the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes.

   During **pass 2**, these values are assembled alongwith the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2) **Translation involving Register-Memory Instructions:** In SIC/XE machine there are four instruction formats and five addressing modes.

   Among the instruction formats, format-3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other operand is in the memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

   This **addressing mode** can be represented by either using format-3 type or format-4 type of instruction format. In format-3, the instruction has the opcode followed by a 12-bit displacement value in the address field, whereas in format-4 the instruction contains the mnemonic code followed by a 20-bit displacement value in the address field.

Calculation of Displacement Value for Target Address

Assembler must calculate displacement value to get target address for both addressing modes. Displacement must be small enough to fit in the 12-bit field in the instruction:

i) If the displacements too large, 4 byte extended instruction format (Format 4) must be used. It contains a 20-bit address field. No need to calculate displacement.

Format 4 (Extended Format Instruction)

6 bits	1	1	1	1	1	1	20 bits
OP code	n	i	x	b	p	e	address

ii) '+' specifies extended format.

For example, 0006 CLOOP + JSUB RDREC 4B101036

There are two ways for translation involving Register-Memory Instructions:

i) **Program-Counter Relative:** In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. The range of displacement values are from 0-2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction. This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The **following example** shows how the address is calculated:

For PC relative P = 1 TA = (PC) + Displacement

⇒ Displacement = TA – PC

```
2   0000  FIRST  STL     RETADR   17202D
3   0003         LDB     #LENGTH  69202D
          --------
          --------
18  0030         RETADR  RESW1
```

TA = 0030 PC = 0003

Displacement = 0030 – 0003

i.e., 0000 0000 0011 0000 (0030)
0000 0000 0000 0011 (0003)
0000 0000 0000 0011

Displacement = 0 0 2 D

ii) **Base-Relative Addressing Mode:** In this mode the base register is used to mention the displacement value. Therefore the target address is:

TA = (base) + displacement value

This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive **BASE**. The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.

When **NOBASE** directive is used then it indicates the base register is no more used to calculate the target address of the operand.

Assembler first chooses PC-relative, when the **displacement field** is not enough it uses Base-relative. **For example,**

```
LDB #LENGTH (instruction)
BASE LENGTH (directive)
:
NOBASE
```

For Example

```
12   0003  LDB     #LENGTH        69202D
13         BASE    LENGTH
: :
100  0033  LENGTH  RESW    1
105  0036  BUFFER  RESB    4096
: :
160  104E  STCH    BUFFER, X      57C003
165  1051  TIXR    T              B850
```

In the **above example** the use of directive BASE indicates that Base-relative addressing mode is to be used to calculate the target address. PC-relative is no longer used. The value of the LENGTH is stored in the base register. If PC-relative is used then the target address calculated is:

$(0036)_{16} - (1051)_{16}$

$= (-1015)_{16} < (-0800)_{16}$

$= (-2048)_{10}$

The LDB instruction loads the value of length in the base register which 0033. BASE directive explicitly tells the assembler that it has the value of LENGTH.

BUFFER is at location $(0036)_{16}$
$(B) = (0033)_{16}$

Displacement = $0036 - 0033 = (0003)_{16}$

op	n	i	x	b	p	e	disp	
010101	1	1	1	1	0	0	003	⇒ 57C003

20 000A LDA LENGTH 032026
: :
175 1056 EXIT STX LENGTH 134000

Line 20 using PC-relative addressing mode.
Consider Line 175. If we use PC-relative:
Displacement = TA – (PC) = 0033 – 1059 = EFDA
PC relative is no longer applicable, so we try to use BASE relative addressing mode.

Consider another **example**,

For Base Relative b = 1

i) Specify content of base register.
ii) Use "**BASE**" assembler directive to specify address, i.e., **BASE LENGTH**

```
        Copy      START   1000
1000    FIRST     STL     RETADR
        ------    -----
        ------    -----
1033    LENGTH    RESB    1
        ------    -----
        ------    -----
1036    BUFFER    RESB    4096
        ------    -----
        ------    -----
```

Base register = 1033 (Address of LENGTH)
TA = 1036

Displacement Address = TA – B
= 1036 – 1033 = 0003

Difference between PC Relative Addressing and Base Relative Addressing

PC Relative Addressing	Base Relative Addressing
Assembler knows content of PC during execution time.	Programmer tells assembler what the base register contains during program execution. Assembler will calculate displacement.

Immediate Addressing Mode

1) No memory reference is involved
2) If immediate mode is specified, the target address becomes the operand
3) The immediate operand is the value of the symbol LENGTH, which is its address (not its content)
4) If immediate mode is specified, the target address becomes the operand.

12 0003 LDB #LENGTH

LENGTH is at address 0033
TA = (PC) + disp ⇒ disp = 0033 – 0006 = $(002D)_{16}$

OP	n	i	x	b	p	e	disp	
011010	0	1	0	0	1	0	02D	⇒ 69202D

Indirect Address Translation
Target addressing is computed as usual (PC-relative or BASE-relative) only the n bit is set to 1

70 002A J @RETADR
: :
95 0030 RETADR RESW 1

RETADR is at address 0033
TA = (PC) + disp ⇒ disp = 0033 – 002D = $(0003)_{16}$

OP	n	i	x	b	p	e	disp	
001111	0	1	0	0	1	0	003	⇒ 3E2003

Ques 3) Why is the displacement field of PC related addressing mode interpreted as 12 bit signed integer? (2020[03])

Ans: Two new relative addressing modes are available for use with instructions assembled using Format 3. These are described in the following table:

Mode	Instruction	Target Address Calculation	
Base relative	b =1, p = 0	TA = (B) + disp	(0 ≤ disp ≤ 4095)
Program-counter relative	b = 0, p = 1	TA = (PC) + disp	(–2048 ≤ disp ≤ 2047)

For base relative addressing, the displacement field disp in a Format 3 instruction is interpreted as a 12-bit unsigned integer.

1056 STX LENGTH

6	1	1	1	1	1	1	12	
0001 00	1	1	0	1	0	0	0000 0000 0000	
opcode	n	i	x	b	p	e		
1	3		4	0		0		0 Object Code

EA= LENGTH =0033 EA = disp + [B]
[B] = 0033
disp = 0

The content of the address 0033 is loaded to the index register X.

For program-counter relative addressing, this field is interpreted as a 12-bit signed integer, with negative values represented in 2's complement notation.

Ques 4) Explain program relocation with an example. (2017, 2019[03])

Or

Write the short notes on Format for Modification Record.

Or

Assemble the following instruction indicating the instruction formats used: (2020[03])
a) RMO S, A
b) +JSUB RDREC
c) LDA #1

Assume that the value of RDREC is 1036.
OPTAB

Opcode	Machine Code
RMO	AC
JSUB	48
LDA	00

REGISTER

A	0
S	4

Ans: Program Relocation

It is desirable to load a program into memory wherever there is space for it. In such a case, the actual starting address is not known until the load time.

The assembler cannot make changes over the addresses used by the program. But it could identify, the parts of the object program that requires modification.

An object program that contains necessary information to perform this kind of modification is called relocatable program. **For example,** consider a program with a starting address 0000.

Object code

0006 JSUB RDREC 4B1 01036

The address field for this instruction is 01036.

Suppose, user wants to load this program beginning with the address 5000 then the address of the instruction becomes 6036.

Similarly when the program starting address is 7420, the JSUB instruction comes with the new address 08456.

Wherever the program is loaded, JSUB instruction is always 1036 bytes past the starting address of the program. The **above examples** are diagrammatically represented as shown in **figure 3.1:**

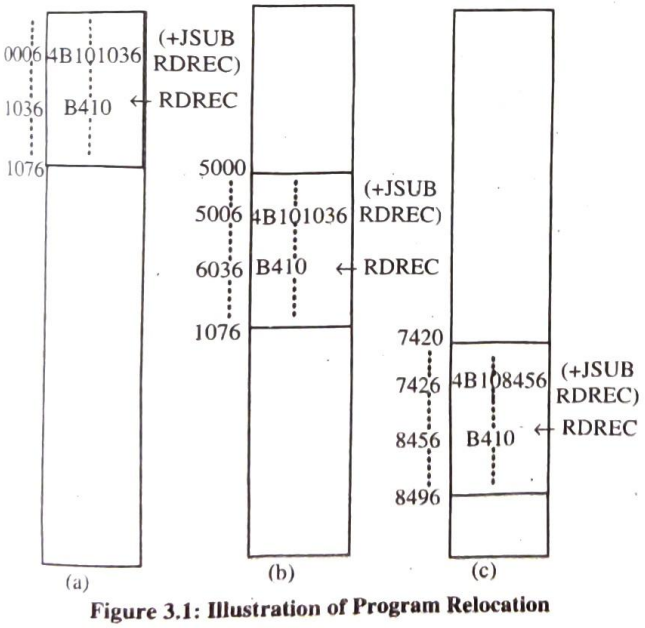


Figure 3.1: Illustration of Program Relocation

Note: The assembler must determine the portions of the code that change with the load address and make modifications accordingly. Other address remains the same.

The assembler still does not know where the program is actually loaded. Hence it cannot make any changes in the addresses used by the program but only identifies the parts and marks them, for the loader to take care of the rest.

The **problem** of program relocation is **solved as**: consider a JUMP instruction such as JSUB as an **example**. The assembler generates the code for the jump instruction relative to the start address of the program and inserts a command for the loader instructing it to add the load address to the address field of the JUMP instruction. This command for the loader is also made part of the final object codes. This is done using a modification record.

Format for Modification Record

Col 1	Char 'M'
Col 2-7	Starting location of address field to be modified, relative to the load address of the program.
Col 8-9	Length of the address field to be modified in 'Half Bytes'. The length is in half bytes.

For example, a sample modification record
M00000603

Explanation: M represents the modification record 000006. It indicates that it represents the load address of the program must be added to an address field beginning at address 000006.03 represents the length in half bytes of the field which is to be added to the load address.

Each address field that requires change when the program is loaded has a corresponding modification record.

Ques 5) Is there a need to use modification records for the given SIC/XE program segment? Explain your answer. If yes, show the contents of modification record. (2019[03])

```
0000        COPY        START     0
.....
0006                    +JSUB     RDREC
000A                    LDA       LENGTH
.....
0033        LENGTH      RESW      1
.....
1036        RDREC       CLEAR     X
```

Ans: Modification Record

The command for the loader must also be a part of the object program. We can accomplish this with a modification record having the following format:

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
Col. 8-9	Length of the address field to be modified, in half bytes(hexadecimal)

One modification record for each address to be modified. The length is stored in half-bytes (20 bits = 5 half-bytes). The starting location is the location of the byte containing the leftmost bits of the address field to be modified.

If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte at the starting location. These conventions are, of course, closely related to the architecture of SIC/XE. For other types of machines, the half-byte approach might not be appropriate. For the JSUB instruction we are using as an example the modification record would be.

M00000705

This record specifies that the beginning address of the program is to be added to a field that begins at address 000007 (relative to the start of the program) and is 5 half-bytes in length. Thus in the assembled instruction 4B101036, the first 12 bits (4B1) will remain unchanged. The program load address will be added to the last 20 bits (01036) to produce the correct operand address.

MACHINE INDEPENDENT ASSEMBLER FEATURES

Ques 6) Define the machine independent assembler features.

Or

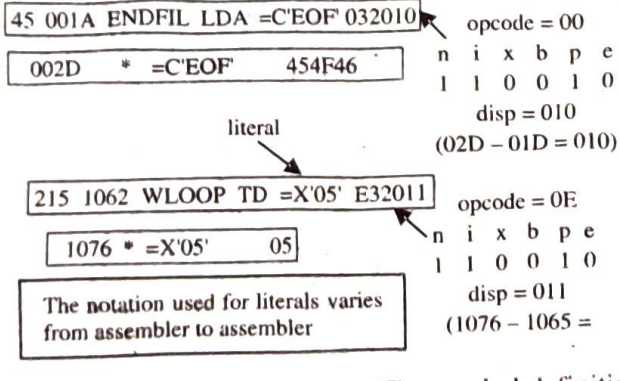
Explain the following machine independent features of assembler:

i) Literals
ii) Symbol Defining Statements
iii) Expressions

Ans: Machine Independent Assembler Features

There are some common assembler features that are not related to machine structure. The machine-independent features of assembler are listed as:

1) **Literals:** 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. In SIC/XE program, a literal is identified with the **prefix =**, followed by a specification of the literal value, using the same notation.



2) **Symbol-Defining Statements:** The symbol definition is as follows:
   i) **Label:** The value of labels on instructions or data areas is the address assigned to the statement on which it appears.
   ii) **EQU:** The **assembler directive EQU** allows the programmer to define symbols and specify their value. (Similar to #define or MACRO in C language).
   iii) **ORG:** The **assembler directive ORG** indirectly assigns values to symbols.

3) **Expressions:** Most assemblers allow the use of expressions whenever such a single operand is permitted.
   i) Each expression must be evaluated by the assembler to produce a single operand address or value.
   ii) Arithmetic expressions usually formed by using the operators +, –, *, and /.
   iii) The most common special term is the current value of the location counter (often designated by *).

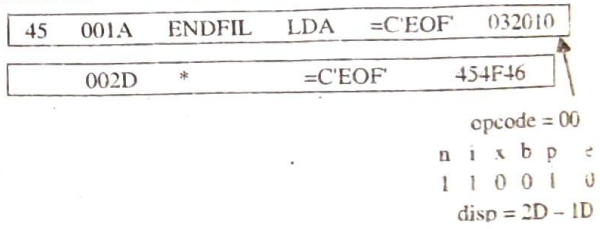
   The values of terms and expressions are either relative or absolute. **For example,**
   i) A constant is an absolute term.
   ii) Labels on instructions and data areas, and references to the location counter value are relative terms.
   iii) A symbol whose value is given by **EQU** (or some similar assembler directive) may be either an absolute term or a relative term depending on the expression used to define its value.

Ques 7) What is the difference between literal and immediate operand?

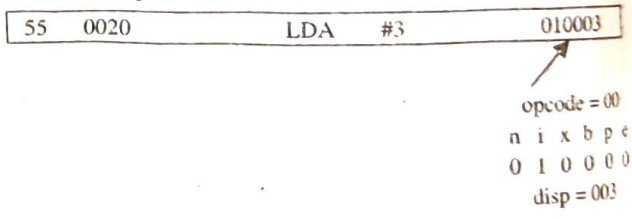
Ans: Difference between Literal and Immediate Operand

With a literal, the assembler generates the specified value as a constant at some other memory location.

The address of this generated constant is used as the target address for the machine instruction.



With immediate addressing, the operand value is assembled as part of the machine instruction.



Ques 8) Define duplicate literal.

Ans: Duplicate Literals

A duplicate literal is a literal used in more than one place in the program. **For example,** the literal =X '05' is used in following program on lines 215 and 230.

```
215   1062   WLOOP   TD = X'05'
230   106 B          WD = X'05'
```

The easiest way to recognise duplicate literals is by comparison of the character strings defining them. If one uses the character string defining a literal to recognise duplicates, he/she must be careful of literals whose value depends on their location in the program. **For example,** we allow literals that refer to **the current value of the location counter** (denoted by the symbol *).

13	0003	LDB	=* (* equals to 3)	
55	0020	LDA	=* (* equals to 20)	

Note: The same literal name * has different values.

Ques 9) Define literal pool and LTORG.

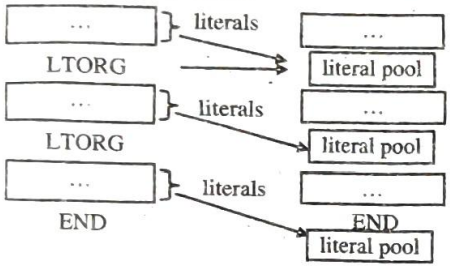
Or

Define LTORG directive and its advantages.

Ans: Literal Pool and LTORG

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.



At the end of the object program, a literal pool immediately following the END statement.

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.

For example, if we had not used the LTORG statement on line 93, the literal = 'EOF' would be placed in the pool at the end of the program. This literal pool would begin at address 1073. It is too far away from the instruction referencing it to allow PC relative addressing. The problem is the large amount of storage reserved for **BUFFER.**

Advantages of LTORG Directive

1) Literal pool placed near from the instruction when used since the previous LTORG.
2) LTORG keeps literal dose to the instruction that uses it.

Ques 10) Write a note on EQU statement.

Ans: EQU Statement

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this EQU (**Equate**). The **general form** of the statement is:

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. **For example,**

```
+LDT    #4096
```

This loads the register T with immediate value 4096, this does not clearly what exactly this value indicates. If a statement is included as:

```
MAXLEN  EQU    4096 and then
        +LDT   #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN alongwith its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction.

The object code generated is the same for both the options. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. One has to scan the whole program and make changes wherever 4096 is used. If one mentions this value in the instruction through the symbol defined by EQU, he/she may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

Another common usage of EQU statement is for defining values for the general-purpose registers. The assembler can use the mnemonics for register usage like a-register A, X – index register and so on.

But there are some instructions which require numbers in place of names in the instructions. **For example,** in the instruction RMO 0,1 instead of RMO A, X. The programmer can assign the numerical values to these registers using EQU directive.

```
A    EQU    0
X    EQU    1 and so on
```

These statements will cause the symbols A, X, L... to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed.

As another usage if in a machine that has many general purpose registers named as R1, R2,..., some may be used as base register, some may be used as accumulator. Their usage may change from one program to another. In this case, one can define these requirement using EQU statements.

```
BASE    EQU    R1
INDEX   EQU    R2
COUNT   EQU    R3
```

One **limitation** with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be pre-defined. **For example,** the following statement is not valid:

```
BETA    EQU     ALPHA
ALPHA   RESW    1
```

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

Ques 11) Briefly discuss the ORG statement. Also list the restrictions of ORG and EQU.

Ans: ORG Statement

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG **(for origin)**. Its **general form** of statement is:

```
ORG         value
```

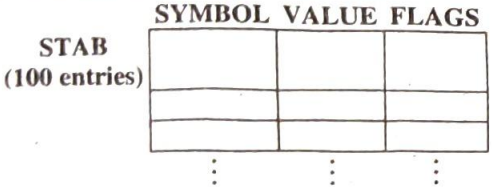
Where, **value** is a constant or an expression involving constants and previously defined symbols.

When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose one needs to define a symbol table with the following structure:

SYMBOL	6 Bytes
VALUE	3 Bytes
FLAG	2 Bytes

The table looks like the one given below:



The **SYMBOL** field contains a 6-byte user-defined symbol; **VALUE** is a one-word representation of the value assigned to the symbol; **FLAGS** is a 2-byte field specifies symbol type and other information. The space for the table can be reserved by the statement:

```
STAB        RESB        1100
```

If one wants to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, he/she needs to assign the values first as **shown below**:

SYMBOL	EQU	STAB
VALUE	EQU	STAB+6
FLAGS	EQU	STAB+9

To retrieve the VALUE field from the table indicated by register X, one can write a statement:

```
LDA         VALUE, X
```

The same thing can also be done using ORG statement in the following way:

STAB	RESB	1100
	ORG	STAB
SYMBOL	RESB	6
VALUE	RESW	1
FLAG	RESB	2
	ORG	STAB+1100

The **first statement** allocates 1100 bytes of memory assigned to label STAB.

In the **second statement** the ORG statement initialises the location counter to the value of STAB. Now the LOCCTR points to STAB.

The **next three lines** assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols.

The **last ORG statement** re-initialises the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. **For example,** for the sequence of statements below:

	ORG	ALPHA
BYTE1	RESB	1
BYTE2	RESB	1
BYTE3	RESB	1
	ORG	
ALPHA	RESB	1

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In the first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

Restrictions of EQU and ORG

1) All symbols used on the right-hand side of the EQU statement must have been defined previously in the program.

ALPHA	RESW	1	⇒	Allowed
BETA	EQU	ALPHA		
BETA	EQU	ALPHA	⇒	Not allowed
ALPHA	RESW	1		

2) ORG requires that all symbols used to specify the new location counter value must have been previously defined.

	ORG	ALPHA		
BYTE1	RESB	1		
BYTE2	RESB	1		
BYTE3	RESB	1	⇒	Not allowed
	ORG			
ALPHA	RESB	1		

Ques 12) Define various types of expressions.

Or

Explain the absolute and relative expressions.

Ans: Types of Expressions

By the type of value produced, expressions can be classified as:

1) **Absolute Expressions:** The value of an absolute expression is independent of the program location. The absolute expression may contains relative terms provided:
   i) Absolute expressions may also contain relative terms:
      a) Relative terms occur in **pairs**. Bother of the terms are positive.
      b) A relative term or expression represents some value that may be written as **(S + r)**, where:
         - **S** is the starting address of the program, and
         - **r** is the value of the term or expression relative to the starting address.
      c) A relative term usually represents some location within the program.

   ii) Both BUFFEND and BUFFER are relative terms, each representing an address within the program, such that the expression represents an **absolute value**. The difference between the addresses is the length of the buffer area in bytes.

   No relative term can enter multiplication or division operation. **For example,**

   BUFEND+BUFEND, 100-BUFFER, or 3*BUFFER are considered errors.

2) **Relative Expressions:** The value of a relative expression is relative the beginning address of the object program. A relative expression is one in which all of the relative terms except one can be paired. The remaining unpaired term must have a positive sign. No relative term can enter multiplication or division operation.

Expressions that are neither relative nor absolute should be flagged by the assembler as **errors**.

To determine the type of an expression, one must keep track of the types of all symbols defined in the program. For this, one needs a flag in the symbol table to indicate type of value (absolute or relative) in addition to the value itself. some of the symbol table entries might be:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

With the "Type" information, the assembler can easily determine the type of each expression to generate **Modification records** in the object program for relative values.

Ques 13) Explain Program block.

Or

Explain program block with an example, a machine independent assembler feature.

Or

Distinguish between Program Blocks and Control Section. **(2018[02])**

Or

Describe the concept of program blocks with a proper example. **(2019[04])**

Or

Using the given information, generate the machine instruction for the instruction at location 0006 and 003F. Assume that program blocks are used in the program, the machine code for LDA is 00 and STCH is 54 and the block table is as follows. **(2020[04])**

Block Name	Block Number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

Loc	Block Number	Label	Opcode	Operand
0006	0		LDA	LENGTH
003F	0		STCH	BUFFER,X
0003	1	LENGTH	RESW	1
0000	2	BUFFER	RESB	4096

Ans: Program Blocks

Many assemblers provide features that allow more flexible handling of the source and object program. Some allows the generated machine instructions and data to appear in different order from the corresponding source statements (**Program Blocks**.) and some results in the creation of several independent parts of the object program (**Control Sections**).

The program parts maintain their identity and are handled separately by the loader.

1) **Program Blocks:** Segments of code those are re-arranged within a single object program unit.
2) **Control Sections:** Segments that are translated into independent object program units.

Figure 3.2 shows the example of a program with multiple programs blocks:

```
 5 COPY    START   0           COPY FILE FROM
                               INPUT TO OUTPUT
10 FIRST   STL     RETADR      SAVE RETURN
                               ADDRESS
15 CLOOP   JSUB    RDREC       READ INPUT
                               RECORD
20         LDA     LENGTH      TEST FOR EOF
                               (LENGTH = 0)
25         COMP    #0
30         JEQ     ENDFIL      EXIT IF EOF
                               FOUND
35         JSUB    WRREC       WRITE OUTPUT
                               RECORD
```

```
40          J       CLOOP       LOOP
45 ENDFIL   LDA     =C 'EOF'    INSERT END OF
                                FILE MARKER
50          STA     BUFFER
55          LDA     #3          SET LENGTH = 3
60          STA     LENGTH
65          JSUB    WRREC       WRITE EOF
70          J       @RETADR     RETURN TO
                                CALLER
92          USE     CDATA
95 RETADR   RESW    1
100 LENGTH  RESW    1           LENGTH OF
                                RECORD
103         USE     CBLKS
105 BUFFER  RESB    4096        4096-BYTE
                                BUFFER AREA
106 BUFEND  EQU     *           FIRST LOCATION
                                AFTER BUFFER
107 MAXLEN  EQU     BUFEND-     MAXIMUM
                    BUFFER      RECORD LENGTH
110 .
115 .       SUBROUTINE TO READ RECORD INTO
            BUFFER
120 .
123         USE
125 RDREC   CLEAR   X           CLEAR LOOP
                                COUNTER
130         CLEAR   A           CLEAR A TO ZERO
132         CLEAR   S           CLEAR S TO ZERO
133         +LDT    #MAXLEN
135 RLOOP   TD      INPUT       TEST INPUT
                                DEVICE
140         JEQ     RLOOP       LOOP UNTIL
                                READY
145         RD      INPUT       READ
                                CHARACTER INTO
                                REGISTER A
150         COMPR   A, S        TEST FOR END OF
                                RECORD (X '00')
155         JEQ     EXIT        EXIT LOOP IF EOR
160         STCH    BUFFER, X   STORE
                                CHARACTER IN
                                BUFFER
165         TIXR    T           LOOP UNLESS
                                MAX LENGTH
170         JLT     RLOOP       HAS BEEN
                                REACHED
175 EXIT    STX     LENGTH      SAVE RECORD
                                LENGTH
180         RSUB                RETURN TO
                                CALLER
183         USE     CDATA
185 INPUT   BYTE    X 'F1'      CODE FOR INPUT
                                DEVICE
195 .
200 .       SUBROUTINE TO WRITE RECORD FROM
            BUFFER
205 .
208         USE
210 WRREC   CLEAR   X           CLEAR LOOP
                                COUNTER
212         LDT     LENGTH
215 WLOOP   TD      =X '05'     TEST OUTPUT
                                DEVICE
220         JEQ     WLOOP       LOOP UNTIL
                                READY
225         LDCH    BUFFER, X   GET CHARACTER
                                FROM BUFFER
230         WD      =X '05'     WRITE
                                CHARACTER
235         TIXR    T           LOOP UNTIL ALL
                                CHARACTERS
240         JLT     WLOOP       HAVE BEEN
                                WRITTEN
245         RSUB                RETURN TO
                                CALLER
252         USE     CDATA
253         LTORG
255         END     FIRST
```

Figure 3.2: Example of Multiple Program Blocks

Explanation

1) In this example, three program blocks are used:
   i) First **(unnamed) program block** – The executable instructions of the program.
   ii) The second **(named CDATA) program block** – All data areas that are a few words or less in length.
   iii) The third **(named CBLKS) program block** – All data areas that consist of larger blocks of memory.

2) The assembler directive **USE** indicates which portions of the source program belong to the various blocks.
   i) At the beginning of the program, statements are assumed to be part of the unnamed (default block).
   ii) If no USE statements are included, the entire program belongs to this single block.
      a) Line 123, 208 – resume default block.
      b) Line 92, 183, 252 – CDATA block.
      c) Line 103 – CBLKS block.

3) Each program block may actually contain several separate segments of the source program.

4) The assembler will logically **re-arrange these segments to gather together** the pieces of each block.

5) These blocks are assigned its addresses in the object program, with the blocks appearing in the same order in which they were first begun in the source program.

6) The result is the same as if the programmer had physically re-arranged the source statements to group together all the source lines belonging to each block.

7) The assembler accomplishes the program blocks by maintaining a **separate location** for each program block. (In **Pass 1**):
   i) Each label in the program is assigned an address that is relative to the start of the block that contains it.
   ii) When labels enter the symbol table, the block name or number is stored alongwith the assigned relative address.

Created at the end of Pass 1

Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

8) For code generation during **Pass 2**, the assembler needs to revise the address for each symbol relative to the start

of the object program (not the start of an individual program block). The assembler simply adds the location of the symbol, relative to the start of its block.

The **figure 3.3** demonstrates this process applied to program shown in **figure 3.2:**

Line	Loc/Block		Label	Instr	Operand	Object Code
5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C 'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	
110			.			
115			.	SUBROUTINE TO READ RECORD INTO BUFFER		
120			.			
123	0027	0		USE		
125	0027	0	RDREC	CLEAR	X	B410
130	0029	0		CLEAR	A	B400
132	002B	0		CLEAR	S	B440
133	002D	0		+LDT	#MAXLEN	75101000
135	0031	0	RLOOP	TD	INPUT	E32038
140	0034	0		JEQ	RLOOP	332FFA
145	0037	0		RD	INPUT	DB2032
150	003A	0		COMPR	A, S	A004
155	003C	0		JEQ	EXIT	332008
160	003F	0		STCH	BUFFER, X	57A02F
165	0042	0		TIXR	T	B850
170	0044	0		JLT	RLOOP	3B2FEA
175	0047	0	EXIT	STX	LENGTH	13201F
180	004A	0		RSUB		4F0000
183	0006	1		USE	CDATA	
185	0006	1	INPUT	BYTE	X 'F1'	F1
195			.			
200			.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205			.			
208	004D	0		USE		
210	004D	0	WRREC	CLEAR	X	B410
212	004F	0		LDT	LENGTH	772017
215	0052	0	WLOOP	TD	=X '05'	E3201B
220	0055	0		JEQ	WLOOP	332FFA
225	0058	0		LDCH	BUFFER, X	53A016
230	005B	0		WD	=X '05'	DF2012
235	005E	0		TIXR	T	B850
240	0060	0		JLT	WLOOP	3B2FEF
245	0063	0		RSUB		4F0000
252	0007	1		USE	CDATA	
253				LTORG		
	0007	1	*	=C 'EOF'		454F46
	000A	1	*	=X '05'		05
255				END	FIRST	

Figure 3.3: **Example of Multiple Program Blocks with Object Code**

Ques 14) How multiple program blocks are handled by assemblers?

Or

How the assembler handles multiple program blocks? (2018[07])

Or

How are program blocks handled by the assembler? (2020[05])

Ans: Multiple Program Blocks Explanation

The value of the symbol MAXLEN (line 107 in **figure 3.2**) is shown **without a block number** because MAXLEN is an absolute symbol (not relative to the start of any program block).

The separation of the program into blocks has considerably reduced the addressing problem. The large buffer area is moved to the end of the object program, so that:

i) Extended format instructions are no longer needed. (Lines 15, 35, 65)

ii) The base register is no longer necessary. (Lines 13 and 14 are deleted – LDB and BASE instructions).

iii) Literal placement and references are simplified. (include a LTORG in CDATA block to ensure literals are placed ahead of large data areas.)

The use of program blocks could satisfy both machine and human factors:

i) Machine considerations suggest that the parts of the object program appear in memory in a particular order.

ii) Human factors suggest that the source program should be in a different order.

The assembler can simply write the object code as it is generated during Pass 2 and insert the proper load address in each Text record. These load addresses will reflect the starting address of the block as well as the relative location of the code within the block. This process is illustrated in **figure 3.4**.

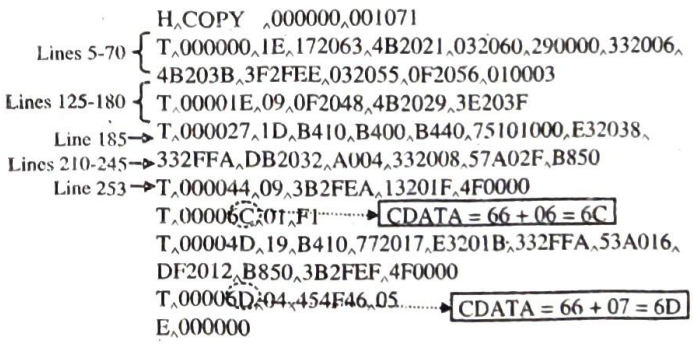


Figure 3.4: **Object Program corresponding to figure 3.3**

The **first two text records** are generated from the source program lines 5 through 70. When the USE statement on line 92 is recognised, the assembler writes out the current Text record. The assembler then prepares to begin a new Text record for the new program block. The statements on lines 95 through 105 result in no generated code, so no new Text records are created.

The **next two Text records** come from lines 125 through 180. This time the statements that belong to the next program block do result in the generation of object code.

The **fifth Text record** contains the single byte of the data from line 185. The **sixth Text record** resumes the default program block and the rest of the object program continues in similar way.

It does not matter that the Text records of the object program are not in sequence by address:

1) The loader will simply load the object code from each record at the indicated address.

2) When this loading is completed:
   i) The generated code from the default block will occupy relative locations 0000 through 0065.
   ii) The generated code and reserved storage for CDATA will occupy locations 0066 through 0070.
   iii) The storage reserved for CBLKS will occupy locations 0071 through 1070.

Figure 3.5 traces the blocks of the example program through this process of assembly and loading.

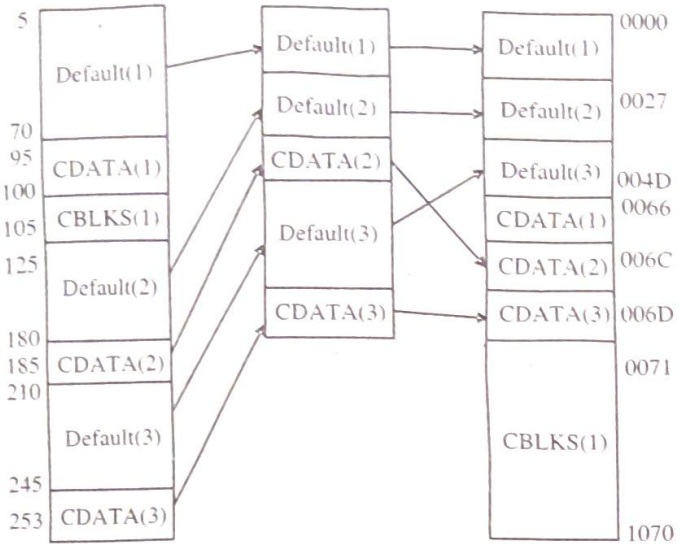


Figure 3.5: Multiple Program Blocks and Loading Process

Ques 15) Describe the Control Sections and Program Linking with Control Section example.

Or

How are control sections different from program blocks? Explain, with proper examples. The purpose of EXTREF and EXTDEF assembler directives. (2017 [04])

Or

What are Control Sections? What is the advantage of using them? (2018[03])

Or

What are the uses of assembler directive EXTDEF and EXTREF? (2018[03])

Or

What are control sections? Illustrate with an example, how control sections are used and linked in an assembly language program.

Ans: Control Sections and Program Linking

A control section is a part of the program that maintains its identity after assembly. Each control section can be loaded and relocated independently of the others.

Different control sections are most often used for subroutines or other logical subdivisions of a program. Control section allows programmers to assemble, load, and manipulate separately to enhance flexibility. Control sections need to provide some means for linking them together.

Some external references that reference instruction or data among control sections need extra information during linking. Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. The assembler generates information for each external reference that will allow the loader to perform the required linking.

The major **benefit** of using control sections is to increase flexibility.

Control Section Example

The **figure 3.6** shows example program of control section and program linking:

```
 5 COPY     START   0           COPY FILE FROM
                                INPUT TO OUTPUT
 6          EXTDEF  BUFFER,
                    BUFEND,
                    LENGTH
 7          EXTREF  RDREC,
                    WRREC
10 FIRST    STL     RETADR      SAVE RETURN
                                ADDRESS
15 CLOOP    +JSUB   RDREC       READ INPUT
                                RECORD
20          LDA     LENGTH      TEST FOR EOF
                                (LENGTH = 0)
25          COMP    #0
30          JEQ     ENDFIL      EXIT IF EOF FOUND
35          +JSUB   WRREC       WRITE OUTPUT
                                RECORD
40          J       CLOOP       LOOP
45 ENDFIL   LDA     =C'EOF'     INSERT END OF
                                FILE MARKER
50          STA     BUFFER
55          LDA     #3          SET LENGTH = 3
60          STA     LENGTH
65          +JSUB   WRREC       WRITE EOF
70          J       @RETADR     RETURN TO
                                CALLER
95 RETADR   RESW    1
100 LENGTH  RESW    1           LENGTH OF
                                RECORD
103         LTORG
105 BUFFER  RESB    4096        4096-BYTE BUFFER
                                AREA
106 BUFEND  EQU     *
107 MAXLEN  EQU     BUFEND-
                    BUFFER
109 RDREC   CSECT
110 .
115 .       SUBROUTINE TO READ RECORD INTO
            BUFFER
120 .
122         EXTREF  BUFFER,
                    LENGTH,
                    BUFEND
125         CLEAR   X           CLEAR LOOP
                                COUNTER
130         CLEAR   A           CLEAR A TO ZERO
132         CLEAR   S           CLEAR S TO ZERO
133         LDT     MAXLEN
```

```
135 RLOOP    TD       INPUT       TEST INPUT DEVICE
140          JEQ      RLOOP       LOOP UNTIL READY
145          RD       INPUT       READ CHARACTER INTO REGISTER A
150          COMPR    A, S        TEST FOR END OF RECORD (X'00')
155          JEQ      EXIT        EXIT LOOP IF EOR
160          +STCH    BUFFER, X   STORE CHARACTER IN BUFFER
165          TIXR     T           LOOP UNLESS MAX LENGTH
170          JLT      RLOOP       HAS BEEN REACHED
175 EXIT     +STX     LENGTH      SAVE RECORD LENGTH
180          RSUB                 RETURN TO CALLER
185 INPUT    BYTE     X'F1'       CODE FOR INPUT DEVICE
190 MAXLEN   WORD     BUFEND-BUFFER
193 WRREC    CSECT
195 .
200 .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205 .
207          EXTREF   LENGTH, BUFFER
210          CLEAR    X           CLEAR LOOP COUNTER
212          +LDT     LENGTH
215 WLOOP    TD       =X'05'      TEST OUTPUT DEVICE
220          JEQ      WLOOP       LOOP UNTIL READY
225          +LDCH    BUFFER, X   GET CHARACTER FROM BUFFER
230          WD       =X'05'      WRITE CHARACTER
235          TIXR     T           LOOP UNTIL ALL CHARACTERS
240          JLT      WLOOP       HAVE BEEN WRITTEN
245          RSUB                 RETURN TO CALLER
255          END      FIRST
```

Figure 3.6 Control Section Example

Explanation

1) In this example, there are three control sections:
   i) One for the main program – The **START statement** identifies the beginning of the assembly and gives a name (**COPY**) to the first control section.
   ii) One for the RDREC subroutine – The **CSECT statement** on line 109 begins the control section **RDREC**.
   iii) One for the WDREC subroutine – The **CSECT statement** on line 193 begins the control section **WDREC**.
2) Control sections are handled separately by the assembler. Symbols that are defined in one control section may not be used directly by another control section. They must be identified as **external reference** for the loader to handle.
3) Two assembler directives are adopted to solve external reference problem:
   i) **EXTDEF (External Definition)**
      a) Name symbols as external symbols that are defined in this control section and may be used by other sections.
      b) Control section names (e.g., COPY, RDREC, and WRREC) are automatically considered to be external symbols (Line 7).
   ii) **EXTREF (External Reference)**
      a) Name symbols that are used in this control section and are defined elsewhere.
      b) **For example,** BUFFER, BUFEND, and LENGTH are defined in the control section named COPY and made available to the other section (Lines 122, 193).

Ques 16) Draw the block diagram of SEGDEF, EXTDEF, LEDAT and FIXUP records of an object record. Explain the significance of each of them.

Ans: Significance and Block Diagram of SEGDEF

SEGDEF records describe the segments in the object module, including their name, length, and alignment. The record format is given below:

98H	length	Attributes(1-4)	Segment length	Name index	Check-sum

Significance and Block Diagram EXTDEF

EXTDEF record contains a list of the external symbols that are referred to in this object module. These records are similar in function to the SIC/XE Define and Refer records. Both PUBDEF and EXTDEF can contain information about the data type designated by an external name. These types are defined in the TYPDEF record. The record format is given below:

8CH	length	External reference list	Check-sum

Significance and Block Diagram LEDATA

LEDATA records contain translated instructions and data from the source program, similar to the SIC/XE Text record. The record format is given below:

A0H	length	Segment index(1-2)	Data offset(2)	data	Check-sum

Significance and Block Diagram FIXUPP

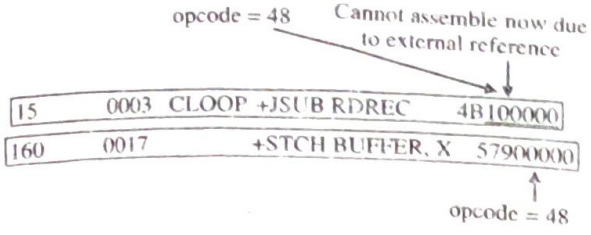
FIXUPP records are used to resolve external references, and to carry-out address modifications that are associated with relocation and grouping of segments within the program. This is similar to the function performed by the SIC/XE Modification records. However, FIXUPP records are substantially more complex, because of the more complicated object program structure. A FIXUPP record must immediately follow the LEDATA or LIDATA record to which it applies.

9CH	length	Locat (1)	Fix dat (1)	frame datum (1)	Target datum (1)	Target offset (2)	Check sum

Ques 17) How external references are used?

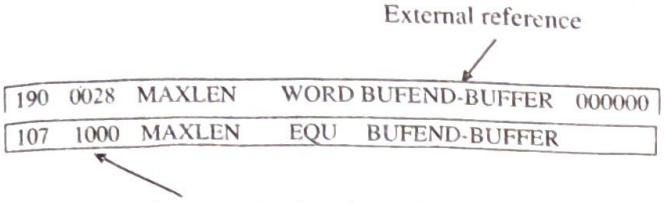
Ans: Using External References

The operand (RDREC) is named in the EXTREF statement for the control section, so this is an external reference.



The assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address for this instruction:

1) **Relative addressing** is not possible during assembling.
2) An **extended format instruction** must be used to provide room for the actual address to be inserted.



The assembler must also allow the same symbol to be used in different control sections. **For example,** the conflicting definitions (e.g., MAXLEN on lines 107 and 190) cause no problem:

i) A reference to MAXLEN in the control section COPY would see the definition on line 107.

ii) A reference to MAXLEN in the control section RDREC would see the definition on line 190.

3) We need two new record types in the object program and a change in a previously defined record type to include information for the loader to insert the proper values when required.

Ques 18) Explain the following records with respect to SIC/XE:
1) Define Record 2) Refer Record
3) Modification Record

Or

Explain the format and purpose of Define and Refer records in the object program. (2017 [03])

Or

Write down the format of Modification record. Describe each field with the help of an example. (2018 [03])

Or

Explain the format of Define and Refer Records. What are their uses? (2019[03])

Ans: Define Record

A Define record gives information about external symbols that are defined in this control section (named by **EXTDEF**).

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section.
Col. 8-13	Relative address of symbol within this control section (hexadecimal).
Col. 14-73	Repeat information in Col. 2-13 for other external symbols.

Refer Record

A Refer record lists symbols that are used as external references by the control section (named by **EXTREF**).

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section
Col. 8-73	Names of other external reference symbols.

The other information of program linking is added to the modification table.

Modification Record

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The format for modification record is as follows:

Col. 1	M
Col. 2-7	Starting address of the field to be modified, relative to the beginning of the control section (hexadecimal).
Col. 8-9	Length of the field to be modified, in half-bytes (hexadecimal).
Col. 10	Modification flag (+ or –).
Col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field.

For example, Object code with relocation by modification record will be

M^ 000007^ 05 + copy

where,

M = Modification Record
000007 = Starting address of the field whose value is to be modified relative to beginning of the program
05 = Length of the address field to be modified in half bytes
\+ = Modification flag
copy = External symbol copy indicates the starting address of the program which is to be added with the address in the object code

Since the modification record depend upon the architecture of machine, will be efficient to perform relocation by using any other method.

ASSEMBLER DESIGN OPTIONS

Ques 19) Explain single pass assemblers and its types.

Or

Define one pass assembler.

Or

Explain the working of any one type of one pass assemble.

Or

Explain the working of a single pass assembler with an example. (2019[05])

Ans: Single Pass Assemblers/One Pass Assemblers

It performs all operations in one go. One-pass assembler is faster as it goes through an assembly language program only once. Its **disadvantage** is that it does not provide as many features as a two-pass assembler. Main problem in trying to assemble a program in one pass involves **forward references.** Instruction operands often are symbols that have not yet been defined in the source program. Thus the assembler does not know what address to insert in the translated instruction. It is easy to **eliminate forward references** to data items; we can simply require that all such areas be defined in the source

program before they are referenced. This restriction is not too severe. The programmer merely places all storage reservation statements at the start of the program rather than at the end. Unfortunately, forward references to labels on instructions cannot be eliminated as easily. The logic of the program often requires a forward jump – **for example,** in escaping from a loop after testing some condition. Requiring that the programmer eliminate all such forward jumps would be much too restrictive and inconvenient. Therefore, the assembler must make some special provision for handling forward references. To reduce the size of the problem, many one-pass assemblers do, however, prohibit (or atleast discourage) forward references to data items.

Types of One-Pass Assembler

There are two main types of one-pass assembler:

1) One type produces object code directly in memory for immediate execution (Load-and-go assembler);
2) The other type produces the usual kind of object program for later execution.

One-pass assemblers that generate their object code in memory for immediate execution. No object program is written out, and no loader is needed. This kind of **load-and-go assembler** is useful in a system that is oriented toward program development and testing. In such a system, a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

A load-and-go assembler avoids the overhead of writing the object program out and reading it back in. This can be accomplished with either a one- or a two-pass assembler. However, a one-pass assembler also avoids the overhead of an additional pass over the source program.

Figure 3.7 illustrates the program of both types of one-pass assembler:

SIC

```
0    1000  COPY     START   1000
1    1000  EOF      BYTE    C'EOF'          454F46
2    1003  THREE    WORD    3               000003
3    1006  ZERO     WORD    0               000000
4    1009  RETADR   RESW    1
5    100C  LENGTH   RESW    1
6    100F  BUFFER   RESB    4096                          Forward
9                   .                                     reference
10   200F  FIRST    STL     RETADR          141009
15   2012  CLOOP    JSUB    RDREC           48203D
20   2015           LDA     LENGTH          00100C
25   2018           COMP    ZERO            281006
30   201B           JEQ     ENDFIL          302024
35   201E           JSUB    WRREC           482062
40   2021           J       CLOOP           302012
45   2024  ENDFIL   LDA     EOF             001000
50   2027           STA     BUFFER          0C100F
55   202A           LDA     THREE           001003
60   202D           STA     LENGTH          0C100C
65   2030           JSUB    WRREC           482062
70   2033           LDL     RETADR          081009
75   2036           RSUB                    4C0000
115                 .   SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
121  2039  INPUT    BYTE    X'F1'           F1
122  203A  MAXLEN   WORD    4096            001000
124                 .
125  203D  RDREC    LDX     ZERO            041006
130  2040           LDA     ZERO            001006
135  2043  RLOOP    TD      INPUT           E02039
140  2046           JEQ     RLOOP           302043
145  2049           RD      INPUT           D82039
150  204C           COMP    ZERO            281006
155  204F           JEQ     EXIT            30205B
160  2052           STCH    BUFFER,X        54900F
165  2055           TIX     MAXLEN          2C203A
170  2058           JLT     RLOOP           382043
175  205B  EXIT     STX     LENGTH          10100C
180  205E           RSUB                    4C0000
195                 .
200                 .   SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
206  2061  OUTPUT   BYTE    X'05'           05
207                 .
210  2062  WRREC    LDX     ZERO            041006
215  2065  WLOOP    TD      OUTPUT          E02061
220  2068           JEQ     WLOOP           302065
225  206B           LDCH    BUFFER,X        50900F
230  206E           WD      OUTPUT          DC2061
235  2071           TIX     LENGTH          2C100C
240  2074           JLT     WLOOP           382065
245  2077           RSUB                    4C0000
255                 END     FIRST
```

Figure 3.7: One Pass Assembler Example

Because the object program is produced in memory rather than being written-out on secondary storage, the handling of forward references becomes less difficult. The assembler simply generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled. The symbol used as an operand is entered into the symbol table (unless such an entry is already present). This entry is flagged to indicate that the symbol is undefined. The address of the operand field of the instruction that refers to the undefined symbol 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 (if one exists), and the proper address is inserted into any instructions previously generated. The **figure 3.8(a)** shows the object code and symbol table entries as they would be after scanning line 40 of the program in **figure 3.7**.

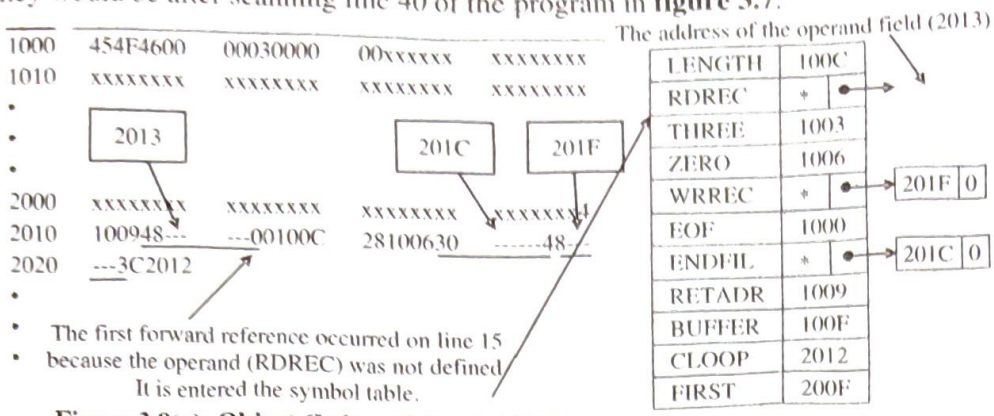


Figure 3.8(a): Object Code and Symbol Table

Now consider the **figure 3.8(b)**, which corresponds to the situation after scanning line 60. Some of the forward references have been resolved by this time, while other has been added. When the symbol ENDFIL was defined the assembler place its value in the SYMTAB entry; it then inserted this value into the instruction operand field (at address 201C) as directed by the forward reference list. From this point on, any references to ENDFIL would not be forward reference and would not be entered into a list. Similarly the definition of RDREC resulted in the filling in of the operand address at location 2013. Meanwhile, two new forward references have been added – to WRREC and EXIT. At the end of the program, any SYMTEB entries that are still marked with * indicate undefined symbols. These should be flagged by the assembler as errors.

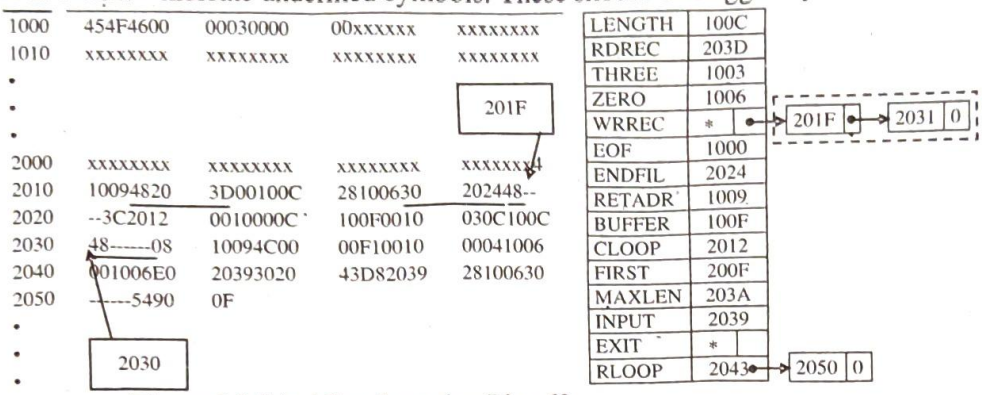


Figure 3.8 (b): After Scanning Line 60

One-pass assembler usually adopts absolute program. It produces **object programs** as output and is usually adopted on systems where external working-storage devices are not available or slow. If the actual address is assigned by the assembly, the **location counter** would be initialised to the actual program starting address at assembling time. When the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. In general, they have already been written out as part of a **Text record** in the object program. In this case, the assembler must generate another Text record with the correct operand address. When the program is loaded, this address will be inserted into the instruction by the action of the loader. **Figure 3.9** illustrate this process.

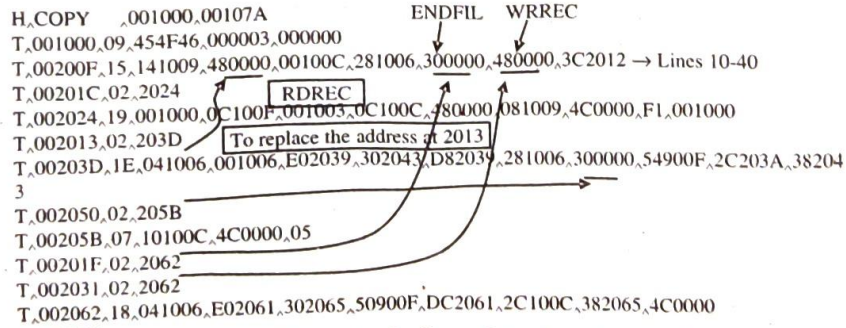


Figure 3.9: Object Code from One-Pass Assembler

Working of a Single Pass Assembler

A single-pass assembler scans the input file only once. This, it is generally faster than a two-pass assembler. It uses the same set of data structures as the two-pass assembler. However, it needs some extra data structures and processing to handle the references to symbols defined in the later part of the program (that is, references to symbols occurring before their definitions). **For example,** consider the following piece of code:

```
    ⋮
X : db 10
    ⋮
MOV AL, X
MOV AL, X
Y :  resb
```

Here, when the assembler reaches the line "X: db 10", it makes entry into the symbol table and simultaneously generates code to reserve one byte initialized to 10. Thus, when the assembler reaches the line "MOV AL, X", the address of X is already available in the symbol table which can be used to generate code. However, at the very next line, "MOV Y, AL" is not simple to process as the type and address of Y is not known at this point of time. These Information will only be available when the line "Y: resb" is seen. This type of situation is known as **forward reference**. Similar conditions will arise for branch targets, which are not yet seen by the assembler.

Ques 20) Explain the load-and-go assembler.

Ans: Load-and-Go Assembler

Load-and-go assembler generates their object code in memory for immediate execution. No program is written out, and no loader is needed. It is useful in a system that is oriented toward program development and testing. **For example,** University computing system for student's use.

Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

The handling of forward references becomes less difficult due to the in-memory process. An instruction operand is a symbol that has not yet been defined, the operand address is omitted when it is assembled.

1) The address of the operand field of the instruction that refers to the undefined symbol is added to a **forward reference list** associated with the symbol table entry.
2) 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.

At the end of assembling, the assembler searches SYMTAB for the value of the symbol named in the END statement (in this case – FIRST).

This is the simplest assembler program. It accepts as input a program whose instructions are essentially one to one correspondence with those of the machine language, but sybolic names used for op-code and operands. The output of the load and go assembler is the machine language, which is loaded directly in the memory and executed.

The resultant, machine language occupies storage locations which are fixed at the time of translation and cannot be changed subsequently.

The load and go assembler program can also called the library subroutines if required. This type of assembler program has the ability to design code and test different program components in parallel. Most of the assemblers are further divided as:

1) **One-Pass Module Assembler:** In this assembler, the translation of machine op-code for mnemonic, machine address for symbolic and machine encoding of character is performed in on a sequential pass over th esource text. These one pass assemblers are useful when secondary storage is either slow or missing entirely. These one pass assembler are suitable for small machines.
2) **Two-Pass Assembler:** These are developed for two-pass translation. During the first pass, the assembler examines the assembler language program and collects the symbolic names into a table. During the second pass, the assembler generates code which is not quite in machine language. It is rather in a similar form sometimes called "relocatable code" and also called as object code (or object module).

Ques 21) Discuss the problems encountered during the single pass assembly. Also write its solution.

Or

Define the forward references problem of single pass assembler. What are the solutions for this problem?

Or

What is meant by Forward Reference? How it is resolved by Two Pass Assembler? (2018[03])

Or

What do you mean by forward reference? How is forward reference handled by a One-Pass Assembler that generates object code? (2020[05])

Ans: Problems Encountered during Single Pass Assembly

A single pass assembler shares some problems as follows:

1) **Forward References:** A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction.

A strict 1-pass scanner cannot assemble source code which contains forward references. Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code. A symbolic name may be forward referenced in a variety of ways. When used as a data operand in a statement, its assembly is straightforward. An entry can be made in the **Table of Incomplete Instructions (TII)**. This entry would identify the bytes in code where the address of the referenced symbol should be put. When the symbol's definition is encountered, this entry would be analysed to complete the instruction. However, use of a symbolic name as the destination in a branch instruction gives rise to a peculiar problem. A more serious problem arises when the type of a forward referenced symbol is used in an instruction. The type may be used in a manner which influences the size/length of a declaration. Such usage will have to be disallowed to facilitate single pass assembly.

For example, consider the statements

```
XYZ   DB    LENGTH ABC DUP(0)
- -
ABC   DD    ?
```

Here the forward reference to ABC makes it impossible to assemble the DB statement in a single pass.

2) **Segment Registers:** An ASSUME statement indicates that a segment register contains the base address of a segment. The assembler represents this information by a pair of the form (segment register, segment name). This information can be stored in a **segment registers table (SRTAB).** SRTAB is updated on processing an ASSUME statement. For processing the reference to a symbol **symb** in an assembly statement, the assembler accesses the symbol table entry of symb and finds (seg_{symb}, $offset_{symb}$) where seg_{symb}, is the name of the symbol containing the definition of symb. It uses the information in SRTAB to find the register which contains seg_{symb}. Let it be register r. It now synthesizes the pair (r, $offset_{symb}$). This pair is put in the address field of the target instruction.

Solutions for Single Pass Assembler

Two methods can be used:

1) **Eliminating Forward References:** We can resolve the problem in two ways:
   i) **Approach 1:** The assembler may read the source program twice – two passes. On pass one, the definitions of symbols, including statement labels, are collected and stored in a table. When pass two starts, the definitions of symbols are known.
   ii) **Approach 2:** On the first reading of the assembly program convert it to an intermediate form stored in memory. The second pass is made over this intermediate form. This saves on I/O time

2) **Generating the Object Code in Memory:** No object program is written out and no loader is needed. The program needs to be re-assembled every time.

Ques 22) Explain multi-pass assembler in detail.

Or

Explain, with examples, the working of a multi pass assembler. (2017 [05])

Or

Give an example of situation where the use of a multi pass assembler can be justified? (2020[03])

Or

What is a multi-pass assembler? Explain with the help of an example, a situation where we would need such an assembler. (2019[03])

Ans: Multi-Pass Assemblers

Multi pass assembler means more **than** one pass it used by assembler. Multi pass assembler is used to eliminate forward reference in symbol definition. It creates a number of passes that is necessary to process the definition of symbols. Multi pass assembler does the work in two pass the resoling the forward references. Translation of assembly language program into object code requiring many passes is termed as multi-pass assembler.

Note: The assembler converts the given assembly language program into intermediate code and then, converts into object code.

The schematic diagram of a multi-pass assembler is shown in **figure 3.10**.

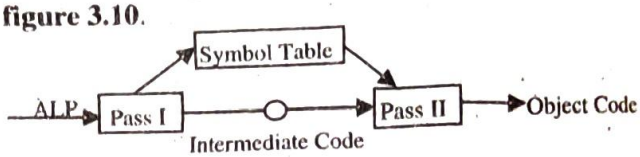


Figure 3.10: Multi-Pass Assembler

Constraints in two-pass assemblers:

1) Any symbol used in the right-hand side (i.e., in the expression giving the value of the new symbol) of assembly directives (e.g., **EQU** and **ORG**) be defined previously in the source program. **For example,**

```
ALPHA EQU   BETA
      BETA  EQU   DELTA
      DELTA RESW  1
```

   BETA cannot be assigned a value in the first pass because DELTA has not yet been defined.

2) Forward reference prohibition is not a serious inconvenience for the programmer.

Multi-pass assemblers can **solve the definitions of symbols**:

1) Forward references in symbol definition are saved during Pass 1.
2) Additional passes through these stored definitions are made as the assembly progresses.

Forward Reference

In an assembly language program, the symbols (variables) are defined elsewhere in the program. If the symbols are used, before they are defined it is considered as a forward reference variable.

Note: Only when forward reference variable are used in a program, it needs two-passes to generate the object code. If not, single-pass is sufficient to generate the object code.

Pass I

1) Accept Assembly language program as input and converts into an Intermediate Code (IC).
2) Separates the symbol. Mnemonic Opcode and operand field.
3) Build the symbol table.
4) Construct the intermediate code for every imperative statement of ALP.

Pass II

1) Synthesize the target code (Object Code) by processing the intermediate code generated during pass I.

Note: The output of pass I named Intermediate code is given as input to pass II where it is converted into object code.

Number of Passes Comparison

There are two of assemblers based on how many passes through the source are needed (how many times the assemblers reads the source) to produce the executable program.

1) One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
2) Multi-pass assemblers create with all symbols and their values in the first passes, then use the table in later passes to generate code.

Example: In the following code snippet a one-pass assembler would be able to determine the address of the backward reference BKWD when assembling statement S2, but would not be able to determine the address of the forward reference FWD when assembling the branch statement SI; indeed FWD may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known generating code in pass2,

```
S1 B FWD
...
FWD EQU*
BKWD EQU*
...
S2 B BKWD
```

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary pad it with one or more "no-operation" instruction in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use to one-pass assemblers was speed of assembly-often a second pass would require rewinding and rereading the program source on tape or rereading a deck of cards or punched paper tape. With modern computers this has ceased to be an issue. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.

Ques 23) How does single pass assembler differ from multipass assembler?

Ans: Difference between Single Pass and Two Pass Assemblers

Single Pass Assemblers	Two Pass Assemblers
Performs translation in one pass itself and hence known as **load-and-go assembler.**	Performs translation in 2 passes by using different organizing principles.
Speed of external/secondary storage never affects the working.	The process can be slow if the external storage used for IR is working slowly.
Forward Referencing can be troublesome though a process called back-patching can be used.	Forward referencing is easily as synthesizing is on pass 2 and every entry in symbol table will be complete before passing it to pass 2.
Back-patching – The address of undefined operand is omitted and is entered to symbol table and is noted on TII (Table of Incomplete Instructions) and when the definition is encountered TII is checked and values are added.	No need for back-patching.
The default address of an undefined symbol will be zero and is later updated with the correct address.	No such scenario will arise and hence no such conventions.
No need for an intermediate code to be generated.	An intermediate code is generated after pass 1 to be passes to pass 2.
All the tables should be available in memory during the translation process and hence more memory intensive.	Optab (Table containing mnemonics and op-codes) is needed only during 1st pass and hence can be eliminated in second pass.
Generally all areas are defined before they are used and back patching is left only for forward jumping.	Though forward referencing is much widely used forward references in symbol definition are not allowed.
It is possible for data items, though inconvenient.	**For example,** jack EQU jill is not allowed.
The object code is generated in memory but is not written out implicitly and hence no loader is needed.	The object code is written-out and a loader is needed.
Emphasize more on processing power and requires more memory So used where memory consumption is of no serious concern.	Used in areas where there is a limitation of available memory.

Ques 24) Write the algorithm for single pass assembler.

Ans: Algorithm for Single Pass Assembler (for Inter 8088)

1) code_area_address:=address of code_area;
   srtab_no: = 1;
   LC: = 0;
   stmt_no: = 1;
   SYMTAB_segment_entry: = 0;
   Clear ERRTAB, SRTAB_ARRAY.

2) While the next statement is not an END statement
   i) Clear machine_code_buffer.
   ii) If a symbol is present in the label field then this_label: = symbol in the label field;
   iii) If an EQU statement
      a) this_address:= value of <address specification>;
      b) Make an entry for this_label in SYMTAB with offset : = this_addr;
         Defined : = 'yes'
         owner_segment: =owner segment in SYMTAB entry of the symbol in the operand field.
         source_stmt_#:= stmt_no;
      c) Enter stmt_no in the CRT list of the label in the operand field.
      d) Process forward references to this_label;
      e) Size:=0;
   iv) If an ASSUME statement
      a) Copy the SRTAB in SRTAB_ARRAY[srtab_no] into SRTAB_ARRAY [srtab_no+l]
      b) srtab_no : = srtab_no+l;
      c) For each specification in the ASSUME statement
         - this_register:= register mentioned in the specification.
         - this_segment:= entry number of SYMTAB entry of the segment apperating in the specification.
         - Make the entry (this_register, this_segment) in SRTAB_ARRAY [srtab_no]. (It overwrites an existing entry for this_register.)
         - size:= 0.
   v) If a SEGMENT statement
      a) Make an entry for this_label in SYMTAB and note the entry number.
      b) Set segment name? := true;
      c) SYMTAB_segment_entry := entry no. in SYMTAB;
      d) LC :=0;
      e) size := 0;
   vi) If an ENDS statement then SYMTAB_segment_entry :=0;
   vii) If a declaration statement
      a) Align LC according to the specification in the operand field.
      b) Assemble the constant(s), if any, in the machine_code_buffer.
      c) size := size of memory area required;
   viii) If an imperative statement
      a) If the operand is a symbol symb then enter stmt_no in CRT list of symb.
      b) If the operand symbol is already defined then Check its alignment and addressability.
         Generate the address specification (segment register, offset) for the symbol using its SYMTAB entry and SRTAB_ARRAY[srtab_no).
         else
            Make an entry for symb in SYMTAB with defined :='no';
            Make the entry (srtab no_LC, usage code, stmt_no) in FRT of symb.
      c) Assemble instruction in machine_code_buffer.
      d) size := size of the instruction;
   ix) If size ≠ 0 then
      a) If label is present then
         Make an entry for this_label in SYMTAB with
         owner_segment := SYMTAB_segment_entry;
         Defined := 'yes';
         offset := LC;
         source_stmt_# := stmt_no;
      b) Move contents of machine_code_buffer to the address code_area_address + <LC>;
      c) LC := LC + size;
      d) Process forward references to the symbol. Check for alignment and addressability errors. Enter errors in the ERRTAB.
      e) List the statement along with errors pertaining to it found in the ERRTAB.
      f) Clear ERRTAB.
3) (Processing of END statement)
   i) Report undefined symbols from the SYMTAB.
   ii) Produce cross reference listing.
   iii) Write code_area into the output file.

Ques 25) Write an algorithm for pass 1 of two pass SIC assembler.

Or

Give the algorithm for pass 1 of two pass SIC assembler. (2017 [05])

Or

Design an algorithm for performing the pass 1 operations of a two pass assembler. (2019[05])

Ans: Algorithm for Pass 1 of Two Pass SIC Assembler

The algorithm for pass 1 assembler is shown in **figure 3.11**:

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR to starting address
      write line to intermediate file
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
    begin
      if this is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin
              search SYMTAB for LABEL
              if found then
                set error flag (duplicate symbol)
              else
                insert (LABEL, LOCCTR) into SYMTAB
            end {if symbol}
```

```
            search OPTAB for OPCODE
            if found then
               add 3 {instruction length} to LOCCTR
            else if OPCODE = 'WORD' then
               add 3 to LOCCTR
            else if OPCODE = 'RESW' then
               add 3 * #[OPERAND] to LOCCTR
            else if OPCODE = 'RESB' then
               add #[OPERAND] to LOCCTR
            else if OPCODE = 'BYTE' then
               begin
                  find length of constant in bytes
                  add length to LOCCTR
               end {if BYTE}
            else
               set error flag (invalid operation code)
         end {if not a comment}
         write line to intermediate file
         read next input line
      end {while not END}
      write last line to intermediate file
      save (LOCCTR – starting address) as program length
end {Pass 1}
```

Figure 3.11: Algorithm for Pass 1 of Assembler

The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initialises the LOCCTR value to this address. This line is written to the intermediate line. If no operand is mentioned the LOCCTR is initialised to zero.

If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value. If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol. It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction. If the opcode is the directive WORD it adds a value 3 to the LOCCTR.

If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes. If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

Ques 26) **Write an algorithm for pass 2 for two pass SIC assembler.**

Or

With the aid of an algorithm explain the Second pass of a Two Pass Assembler. (2018[06])

Ans: **Algorithm for Pass 2 for Two Pass SIC Assembler**

Algorithm for pass 2 is shown in **figure 3.12.**

```
begin
   read first input line {from intermediate file}
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end {if START}
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               search OPTAB for OPCODE
               if found then
                  begin
                     if there is a symbol in OPERAND field then
                        begin
                           search SYMTAB for OPERAND
                           if found then
                              store symbol value as operand
                              address
                           else
                              begin
                                 store 0 as operand address
                                 set error flag(undefined symbol)
                              end
                        end {if symbol}
                     else
                        store 0 as operand address
                     assemble the object code instruction
                  end {if opcode found}
               else if OPCODE = 'BYTE' or 'WORD' then
                  convert constant to object code
               if object code will not fit into the current
                                          Text record then
                  begin
                     write Text record to object program
                     initialize new Text record
                  end
               add object code to Text record
            end {if not comment}
         write listing line
         read next input line
      end {while not END}
   write last Text record to object program
   write End record to object program
   write last listing line
end {Pass 2}
```

Figure 3.12: Algorithm for Pass 2 of Assembler

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialised. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code. If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode.

If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled. If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code (e.g., for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written.

Ques 27) Describe the format of object program generated by the two-pass SIC assembler algorithm. (2017 [04])

Ans: Format of Object Program Generated by the Two-Pass SIC Assembler Algorithm

Figure 3.12 shows the object program

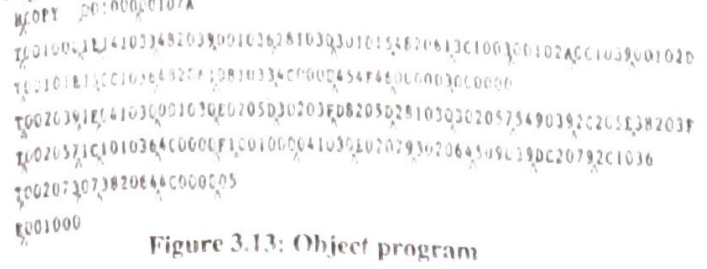


Figure 3.13: Object program

Ques 28) Write the assembly program to compute n! and also write its machine language program with the data structure used.

Ans: Assembly Program to compute N! and Equivalent Machine Language Program

					Opcode (2 digit)	Register operand (1 digit)	Memory operand (3 digit)
1		START	101				
2		READ	N	101)	09	0	113
3		MOVER	BREG, ONE	102)	04	2	115
4		MOVEM	BREG, TERM	103)	05	2	116
5	AGAIN	MULT	BREG, TERM	104)	03	2	116
6		MOVER	CREG, TERM	105)	04	3	116
7		ADD	CREG, ONE	106)	01	3	115
12		MOVEM	CREG, TERM	107)	05	3	116
13		COMP	CREG, N	108)	06	3	113
14		BC	LE, AGAIN	109)	07	2	104
15		MOVEM	BREG, RESULT	110)	05	2	114
16		PRINT	RESULT	111)	10	0	114
17		STOP		112)	00	0	000
18	N	DS	1	113)			
19	RESULT	DS	1	114)			
20	ONE	DC	'1'	115)	00	0	001
21	TERM	DS	1	116)			
22		END					

Data Structure of Pass-I of an assembler

OPTAB		
Mnemonic OPCODE	**Class**	**Mnemonic info**
START	AD	R#11
READ	IS	(09,1)
MOVER	IS	(04,1)
MOVEM	IS	(05,1)
ADD	IS	(01,1)
BC	IS	(07,1)
DC	DL	R#5
SUB	IS	(02,1)
STOP	IS	(00,1)
COMP	IS	(06,1)
DS	DL	R#7
PRINT	IS	(10,1)
END	AD	
MULT	IS	(03,1)

SYMTAB		
Symbol	**Address**	**Length**
AGAIN	104	1
N	113	1
RESULT	114	1
ONE	115	1
TERM	116	1

LITTAB and POOLTAB are empty as there are no Literals defined in the program.

IMPLEMENTATION EXAMPLE

Ques 29) Write short notes on MASM assembler. (2017 [03])

Ans: MASM Assembler

The **Microsoft Macro (MASM) Assembler** is an X86 architecture assembler for MS-DOS and Microsoft Windows. While the name MASM has earlier usage as the Unisys OS 1100 Meta-Assembler, it is commonly understood in more recent years to refer to the Microsoft Macro Assembler. It is an standard MACRO assembler for the x86 PC market that is owned and maintained by a major operating system vendor and since the introduction of MASM **version 6.0** in **1991** has had a powerful pre-processor that supports pseudo high level emulation of variety of high level constructions including loop code, conditional testing and has a semi-automated system of procedure creation and management available if required.

Version 6.11d was 32 bit object module capable of using a specialised linker available in the WinNT 3.5 SDK but with the introduction of binary patches that upgraded version 6.11d, all later versions were 32 bit Portable Executable console mode application that produced both OMF and COFF object modules for 32 bit code. Programmer of an x86 system views memory as a collection segments.

An MASM assembler language program is written as a **collection of segments**. Each segment is defined as belonging to a particular **class** (corresponding to its contents). Commonly used classes are **CODE**, **DATA**, **CONST** and **STACK**. During program execution, segments are addressed via the x86 **segment registers**.

1) **Code Segments:** These are addressed using register CS.
2) **Stack Segments:** These are addressed using register SS.
3) **Data Segments (including Constant Segments):** These are normally addressed DS, ES, FS, or GS.

Ques 30) Define the segments registers of MASM assembler.

Ans: MASM Assembler – Segments

Segment registers are automatically set by the **system loader** when a program is loaded for execution.

1) **Register CS:** This is set to indicate the segment that contains the starting label specified in the END statement of the program.

2) **Register SS:** This is set to indicate the last stack segment processed by the loader.

Segment registers can be specified by the programmer or auto-selected by the assembler. By default, the assembler assumes that all references to **data segments** use register DS. It is possible to collect several segments into a group and use ASSUME to associate a segment register with the group.

```
ASSUME
ES:DATASEG2
```
→ Tell the assembler to assume that register ES indicates the segment DATASEG2. Any references to labels that are defined in DATASEG2 will be assembled using register ES.

Registers DS, ES, FS, and GS must be loaded **by the program** before they can be used to address data segments.

For example,

```
MOV    AX, DATASEG2
MOV    ES, AX
```
→ Set ES to indicate the data segment DATASEG2.

ASSUME directive is similar to the BASE directive in SIC/XE.

1) The BASE directive tells an SIC/XE assembler the contents of register B; the programmer must provide executable instructions to load this value into the register.
2) ASSUME tells MASM the contents of a segment register; the programmer must provide instructions to load this register when the program is executed.

Ques 31) Describe the JMP instruction of MASM.

Ans: MASM Assembler – JMP Instruction

Forward references to labels could cause problems. **For example,**

```
JMP    TARGET
```

If the label TARGET occurs in the program before the JMP instruction, the assembler can tell whether this is a near jump or a far jump.

If this is a forward reference to TARGET, the assembler does not know how many bytes to reserve for the instruction.

By default, MASM assumes that a forward jump is a near jump. If the target of the jump is in another code segment, the programmer must warn the assembler by writing – **(Similar to extended format in SIC/XE)**

```
JMP    FAR    PTR    TARGET
```

If the jump address is with 128 bytes of the current instruction, the programmer can specify the shorter (2-byte) near jump by writing:

```
JMP    SHORT    TARGET
```

If the JMP to TARGET is a far jump and the programmer does not specify **FAR PTR**, a problem occurs. The later versions of MASM assembler can repeat Pass 1 to generate correct location counter values.

Ques 32) Explain the instructions of MASM assembler.

Ans: MASM Assembler – Instruction

There are many other instructions in which the length of an assembled instruction depends on the operands that are used.

1) The operands of an **ADD** instruction may be registers, memory locations, or immediate operands.
   i) Immediate operands may occupy from 1 to 4 bytes in the instruction.
   ii) An operand that specifies a memory location may take varying amounts of space in the instruction, depending on the location of the operand.
2) Pass 1 of an x86 assembler must be considerably ore complex than Pass 1 of an SIC assembler.
   i) The first pass of x86 assembler must analyse the operands of an instruction, in addition to looking at the operation code.
   ii) The operation code table must also be more complicated, since it must contain information on which addressing modes are valid for each operand.

Ques 33) Give a note on references with respect to MASM assembler.

Ans: MASM Assembler – References

Segments in an MASM source program can be written in more than one part. If a **SEGMENT** directive specifies the same name as a previously defined segment, it is considered to be a continuation of that segment.

All of the parts of a segment are gathered together by the assembly process. Segments can perform a similar function to the **program blocks** in SIC/XE.

References between segments that are assembled together are automatically handled by the assembler.

1) External references between separately assembly modules must be handled by the **linker**.
2) The MASM directive PUBLIC has approximately the same function as the SIC/XE directive EXTREF.
3) The MASM directive EXTERN has approximately the same function as the SIC/XE directive EXTREF.

The object program from MASM may be in several different formats to allow easy and efficient execution of the program in a variety of the operation environment.

MASM can also produce an **instruction timing listing** that shows the number of clock cycles required to execute each machine instruction. (This allows the programmer to exercise a great deal of control in optimising timing-critical sections of code.)