Module 1

Introduction to System Software

SYSTEM SOFTWARE

Ques 1) Define system software. How it is different from application software? Also list the various types of system software.

Or

Distinguish between Application software and System software. (2018 [03])

Or

How is system software different from application software? (2020[03])

Or

List out the differences between System Software and Application Software. (2019[04])

Ans: Computer Software

Computer software is set of instructions or programs written to carry out certain task on digital computers.

Types of Computer Software

Software can be classified into two major categories.

1) **System Software:** System software consists of a variety of programs that support the operation of a computer it helps run the computer hardware and computer system. System Software refers to the operating system and all utility programs that manage the computer resources at a low level.

System software is a term for the programs that handle the running of our computer's hardware. Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools.

These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally. Components of system software as shown in **figure 1.1**

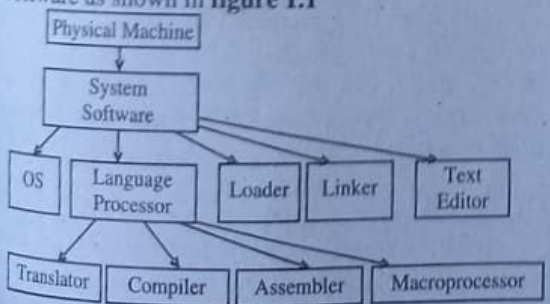


Figure 1.1: Components of System Software

2) **Application Software:** Application software allows end users to accomplish one or more specific tasks. Application software focuses on an application or problem to be solved. Typical applications include industrial automation, business software, educational software, medical software, databases, and computer games.

Difference between System Software and Application Software

The following **table** illustrates the key differences between system software and application software:

Table 1: Difference between System Software and Application Software

Basis for Comparison	System Software	Application Software
Basic	System Software manages system resources and provides a platform for application software to run.	Application Software, when run, perform specific tasks, they are designed for.
Language	System Software is written in a low-level language, i.e. assembly language.	Application Software is written in a high-level language like Java, C++, .net, VB, etc.
Run	System Software starts running when the system is turned on, and runs till the system is shut down.	Application Software runs as and when the user requests.
Requirement	A system is unable to run without system software.	Application software is even not required to run the system; it is user specific.
Purpose	System Software is general-purpose.	Application Software is specific-purpose.
Examples	Operating system.	Microsoft Office, Photoshop, Animation Software, etc.

Ques 2) Give the Classification of System Software in detail.

Ans: Classification of System Software

System software comes as three different sets of software as:

1) **Operating System:** An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application

programs require an operating system to function. Operating systems can be found on almost any device that contains a computer, from cellular phones and video game consoles to supercomputers and web servers.

2) **Language Processor:** It is an example of a "system program", class of tools designed to help software developers. It allows the development of applications in the language most appropriate to the task, removing the need for developing at the machine level.

Programmers can ignore the machine-dependent details of programming. Receives a textual representation of an algorithm in a source language, and produces as output a representation of the same algorithm in the object or target language.

A language processor defined as, **"special translator system software that is used to translate the program written in high-level language (or Assembly language) into machine code".**

3) **Utility Program:** A utility program, also called a utility, is type of system software that allows a user to perform maintenance – type tasks, usually related to managing a computer, its devices, or its programs.

Categories of Utility Programs

The utility programs are classified as into following categories:

i) **File Manager:** A file manager is a utility that performs functions related to file management. Some of the file management functions that a file manager performs are displaying a list of files on a storage medium; organizing files in folders; and copying, renaming, deleting, moving, and sorting files.

ii) **Search Utility:** A search utility is a program that attempts to locate a file on computer based on criteria you specify. The criteria could be a word or words contained in a file, date the file was created or modified, size of the file, location of the file, file name, author/artist, and other similar properties.

iii) **Image Viewer:** An image viewer is a utility that allows users to display, copy, and print the contents of a graphics file. With an image viewer, users can see images without having to open them in a paint or image editing program.

iv) **Personal Firewall:** A personal firewall is a utility that detects and protects a personal computer from unauthorized intrusions. Personal firewalls constantly monitor all transmissions to and from a computer.

v) **Uninstaller:** An uninstaller is a utility that removes a program, as well as any associated entries in the system files. When you install a program, the operating system records the information it uses to run the software in the system files. The uninstaller deletes files and folders from the hard disk, as well as removes program entries from the system files.

vi) **Disk Scanner:** A disk scanner is a utility searches for and removes unnecessary fi Windows Vista includes a disk scanner uti called Disk Cleanup.

vii) **Disk Defragmenter:** A disk defragmenter i utility that re-organizes the files and unused sp on a computer's hard disk, so that the operati system accesses data more quickly and progra run faster.

viii) **Diagnostic Utility:** A diagnostic utility compi technical information about your compute hardware and certain system software progra and then prepares a report outlining any identifi problems. Information in the report assi technical support staff in remedying any problem

ix) **Backup Utility:** A backup utility allows users copy, or backup, selected files or an entire ha disk to another storage medium such as C DVD, external hard disk, USB flash drive, tape. During the backup process, the backu utility monitors progress and alerts you if it nee additional media, such as another CD. Ma backup programs compress, or shrink the size files during the backup process.

Ques 3) 'System Software is machine dependent Justify the statement. **(2019[03]**

Ans: System software is strongly machine dependen Most of the system software differs from applicatio software in terms of machine dependency. Applicatio program is used to solve some problem using the compute as a tool. System program are intended to support th operation and use of the computer itself.

System software that do not directly depends upon th type of computing system being supported. **For example** the code optimization techniques used by compilers ar independent of the target machine.

DIFFERENT SYSTEM SOFTWARE

Ques 2) Define assembler. Write the steps of it working.

Or

Describe in detail about assembler system softwares. **(2019[01]**

Ans: Assemblers

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits (binary digits) that the computer's processor can use to perform its basic operations. These instructions are known as **assembly language**.

Or

An assembler is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader. Assembler program consists of macros, assembler directives, and instructions.

Working of Assembler

An assembler performs four basic **steps** which are as follows:

Step 1: Find the required information to perform a given task.

Step 2: Analyse and design suitable data structures to hold and manipulate the information.

Step 3: Find the processes or steps needed to gather information and also to maintain it.

Step 4: Determine the processing steps required to execute each identified task.

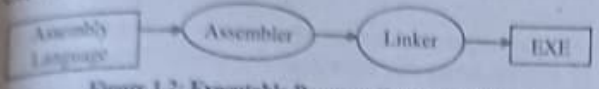


Figure 1.2: Executable Program Generation from Assembly Source Code

Ques 3) Give an introduction to linker and loader. Also define the various types of linkers.

Or

Describe in detail about linker and loader system softwares. (2019[01])

Ans: Linker

Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. Linking can be performed at compile time, when the source code is translated into machine code, at load time, when the program is loaded into memory and executed by the loader, and even at run time, by application programs. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called **linkers**.

Linkers play a crucial role in software development because they enable separate compilation. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and re-link the application, without having to recompile the other files.

Types of Linker

1) **Linking Loader:** the linking loader performs all linking and relocation operations and linked program directly into the main memory for execution.
2) **Linkage Editor:** a linkage editor produce a linked version of the program called as a load module or executable image. This load module is written onto a file or library for later execution.
3) **Dynamic Linker:** this linking postpones the linking function until execution time. Any sub-routine is loaded and linked to the rest of the program when it is first called. This is also called dynamic loading or load on call.

Loader

A loader is a utility of an operating system. It copies programs from a storage device to a computer's main memory, where the program can then be executed. Like linkers, loaders can also replace virtual addresses with real addresses. Most loaders function without user involvement. They are invisible to the user, but are a recognizable utility to the operating system.

Ques 4) Describe the debugger and device driver.

Or

Describe in detail about Debugger softwares. (2019[01])

Ans: Debugger

An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs. Many such systems are available during these days. Debugging means locating (and then removing) bugs i.e., faults in programs. In the entire process of program development errors may occur at various stages and efforts to detect and remove them may also be made at various stages. Debugging is a two-step process that begins when you find an error as a result of a successful test case.

1) Determination of the exact nature and location of the suspected error within the program.
2) Fixing the error.

Device Drivers

Programmers can manipulate I/O devices directly by reading or writing the memory-mapped I/O registers. However, it is better programming practice to call functions that access the memory-mapped I/O. These functions are called **device drivers**.

A device driver is software module which manages the communication with, and the control of, a specific I/O device, or type of device. It is the task of the device driver to convert the logical requests from the user into specific commands directed to the device itself.

For example, a user request to write a record to a floppy disk would be realised within the device driver as a series of actions, such as checking for the presence of a disk in the drive, locating the file via the disk directory, positioning the heads etc.

Ques 5) What is macro processor and text editor?

Ans: Macro Processors

A macro is a unit of specifications for program generation through expansion. Macros are special code fragments that are defined once in the program and are used repetitively by calling them from various places within the program. It is similar to the subprogram in the sense that both can be used to organize the program better by separating-out the frequently used fragment into a different block. The main program calls this block of code as and when needed. Both of them can have an associated list of parameters.

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. Macro processors are often embedded in other programs, such as assemblers and compilers. Sometimes they are standalone programs that can be used to process any kind of text.

Macro processors have been used for language expansion (defining new language constructs that can be expressed in terms of existing language components), for systematic text replacements that require decision making, and for text reformatting (e.g. conditional extraction of material from an HTML file).

Text Editors

A text editor is a program, which allows the user to create the source program in the form of text into the main memory. Generally the user prepares HLL program or any source program. The creation, editing modification, deletion, updating of document or files can be done with the help of the text editor. The scope of the editing is limited to the text only.

Ques 6) Define compiler and interpreter. Bring out the differences between them.

Or

Distinguish between an Assembler and a Compiler. Which are the different types of compilers? (2019[03])

Or

Describe in detail about compiler system softwares. (2019[01])

Ans: Compiler

Compiler translates a source program that is usually written in a high level language by a programmer into machine language. The compiler is capable of replacing single source program statement with a series of machine-language instructions or with a subroutine. For each high level language the machine requires a separate compiler.

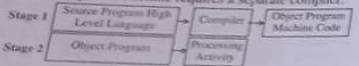


Figure 1.3: Compiler

The main **phases** involved in the compilation process are as given:

1) Syntax analysis,
2) Semantic analysis,
3) Lexical checking,
4) Parsing,
5) Intermediate code generation, and
6) Code optimization.

Types of Compilers

1) **Incremental Compiler:** Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.
2) **Cross Compiler:** A compiler which runs on one machine and produces the target code for another machine. Such compiler is called Cross Compiler. Compiler runs on platform X and target code runs on platform Y.

Interpreter

Interpreter is another example for a language processor. An interpreter is software which converts a program that is written in a high level language into a machine level code line by line. It is the simplest form of translation. The input file or text of the interpreters is called a source program that contains a program written in a high level language. The output of the interpreter is known as an object program **(figure 1.4)**.

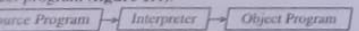


Figure 1.4: Interpreter

Difference between Interpreter and Compiler

The difference between an interpreter and a compiler is given below:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Difference between Assembler and Compiler

Table 1.1: shows the difference between assembler and compiler:

Table 1.1: Difference between Assembler and Compiler

Compiler	Assembler
Translates high-level language into machine code	Translates assembly language into machine code
Translates all code at the same time	Uses the processors instruction set to convert

Ques 7) Explain operating system and language processor.

Or

What are the functions of Operating System? (2018[03])

Ans: Operating System

An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs require an operating system to function.

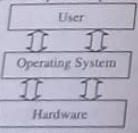


Figure 1.5: OS as Interface between User & Hardware

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware **(figure 1.5)**, although the application code is usually executed directly by the hardware and will frequently make a system call to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a computer, from cellular phones and video game consoles to supercomputers and web servers.

The operating system performs several key **functions**:

1) **Interface:** It provides a user interface so it is easy to interact with the computer.
2) **Manages the CPU:** It runs applications and executes and cancels processes.
3) **Multi-Tasks:** It allows multiple applications to run at the same time.
4) **Manages Memory:** It transfers programs into and out of memory, allocates free space between programs, and keeps track of memory usage.
5) **Manages Peripherals:** It opens, closes and writes to peripheral devices such as storage attached to the computer.
6) **Organises:** It creates a file system to organise files and directories.
7) **Security:** It provides security through user accounts and passwords.
8) **Utilities:** It provides tools for managing and organising hardware.

Language Processor

It is an example of a "system program", class of tools designed to help software developers. It allows the development of applications in the language most appropriate to the task, removing the need for developing at the machine level.

Programmers can ignore the machine-dependent details of programming. Receives a textual representation of an algorithm in a source language, and produces as output a representation of the same algorithm in the object or target language. A language processor defined as, "special translator system software that is used to translate the program

SIC & SIC/XE

Ques 8) List the machine dependent and independent features in system software.

Ans: Machine Dependent and Independent Features in System Software

Most of the features in system Software are Machine Dependent, but some may be **machine independent**. Some of the machine dependent features of some system software are:

1) **Assembler:** instruction format, addressing modes
2) **Compiler:** registers (number, type), machine instructions
3) **OS:** all of the resources of a computing system.

Some aspects of system software are **machine-independent**, they are

1) General design and logic of a assembler,
2) Code optimization in a compiler, and
3) Linking of independently assembled subprograms.

So to study system software it is necessary to know the machine architecture which includes:

1) Memory and Registers
2) Data Formats
3) Instruction Formats
4) Addressing Modes
5) Instruction Set
6) Input and Output

Since system software is machine dependent, there is a need for real machine. However, most real machines have certain characteristics which are unusual or unique. It is very difficult to study fundamental features using a real machine, so we take a hypothetical or a model machine similar to a real machine. **Simplified Instructional Computer (SIC)** is one such hypothetical computer that includes the hardware features most often found on real machines. SIC comes in two versions:

1) SIC (Standard model)
2) XE ("extra equipment")

The two versions have been designed to be upward compatible, i.e., an object program for the standard SIC machine will also execute properly on a SIC/XE system.

Ques 9) Describe SIC machine architecture with all options.

Or

Explain registers, data and instruction format for SIC architectures.

Or

Explain the instruction format and addressing modes of SIC. (2017 [03])

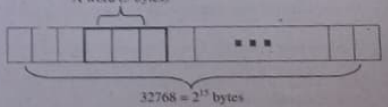
Or

List out the various registers used in SIC along with their purpose. (2018[03])

Ans: SIC Machine Architecture

The SIC machine has basic addressing, storing most memory addresses hexadecimal integer format. Similar to most modern computing systems, the SIC architecture stores all data in binary and uses the two's complement to represent negative values at the machine level. The following is the SIC machine architecture with respect to its Memory, Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output.

1) **Memory:** There are 2^{15} bytes in the computer memory that is 32,768 bytes. It uses **Little Endian** format to store the numbers, 3 consecutive bytes form a word, and each location in memory contains 8-bit bytes.



$32768 = 2^{15}$ bytes

Figure 1.6

2) **Register:** They are used as storage locations and also to perform certain special functions. There are 5 registers, each of which is 24 bits in length. **Table 1.2** given below indicates the numbers, mnemonics and uses of these registers:

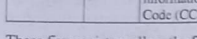
Table 1.2: Types of Registers in SIC

Mnemonic	Number	Special Use/Purpose
A	0	**Accumulator**: used for arithmetic operations.
X	1	**Index register**: used for addressing.
L	2	**Linkage register**: used in control transfer to store the return address of the current instruction.
PC	8	**Program Counter**: contains the address of the next instruction to be fetched for execution.
SW	9	**Status word**: contains various information, including a Condition Code (CC).

These five registers allow the SIC machine to perform most simple tasks in a customised assembly language. In the System Software, this is used with a theoretical series of operation codes to aid in the understanding of assemblers and linker-loaders required for the execution of assembly language code.

3) **Data Formats:** The standard SIC machine supports only the Integers and Character data formats. There is no hardware to support the floating-point numbers. Integers are stored as 24 bit binary numbers. Negative values are represented as 2's complement. Character data are stored using their 8 bit ASCII codes.

4) **Instruction Formats:** All machine instructions in the standard version of SIC have the following 24 bit format:



The flag bit x is used to indicate the indexed addressing mode.

5) **Addressing Modes:** SIC supports two types of addressing modes:

i) **Direct Addressing Mode:** In this mode the flag bit x = 0. Here the target address is specified by the actual address in the instruction itself.

ii) **Indexed Addressing Mode:** In this mode the flag bit x = 1. Here the target address is computed by adding the actual address specified in the instruction and the contents of the Index Register (X).

Table 1.3: Types of Addressing Modes in SIC

Addressing Mode	Flag Bit	Target Address
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (X)

Note: Parenthesis represents the contents of a register or memory.

6) **Instruction Set:** The most common simple tasks are accomplished using the instruction set provided by the SIC machine. The various types of instruction sets are summarised as follows:

i) **Data Transfer Instruction:** These include instructions that Load and Store registers. **For example,** LDA, LDX, STA, STX.

ii) **Arithmetic Operation Instruction:** The basic arithmetic operations can be done which involves register A. **For example,** ADD, SUB, MUL, DIV, COMP.

iii) **Conditional Branching Instruction:** Conditional Jump instructions test the setting of condition code and jumps accordingly. **For example,** JLT, JEQ, JGT.

iv) **Subroutine Call Instruction:** Two instructions are provided to perform subroutine linkage.

a) **JSUB:** To Jump

b) **RSUB:** To Return

In both cases the return address is stored in the register L.

v) **Input and Output Instruction:** Input and Output operations are executed by transferring a single byte each time. The target port is specified by the last 8 bits of the register A. Each device is assigned a unique 8 bit code to send and receive data and control signals.

The Test Device (TD) instruction tests or checks on whether a given I/O device is in ready state to send or receive data from the computer. Wait state must be introduced in the program till the device gets ready for actual data transfer. Read Data (RD) and Write Data (WD) are used to read and write data from or to the specified I/O device.

7) **Input and Output:** Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

8) **Data Movement and Storage Definition:** LDA, STA, LDL, STL, LDX, STX (A – Accumulator, L – Linkage Register, X – Index Register), all uses 3-byte word. LDCH, STCH associated with characters uses 1-byte. There are no memory-memory move instructions. Storage definitions are:

i) WORD – ONE-WORD CONSTANT
ii) RESW – ONE-WORD VARIABLE
iii) BYTE – ONE-BYTE CONSTANT
iv) RESB – ONE-BYTE VARIABLE

Ques 10) What will happen if a SIC program is loaded in a location different from the starting address specified in the program? Will the program work properly? Justify your answer. (2019[03])

Or

Explain with an example how relocation problem is handled by an assembler? (2020[05])

Ans: It is often desirable to have more than one program at a time sharing the memory and other resources of the machine. If we knew in advance exactly which programs were to be executed concurrently in this way, we could assign addresses when the programs were assembled so that they would fit together without overlap or wasted space.

Most of time, however, it is not practical to plan program execution this closely. We usually do not know exactly when jobs will be submitted, exactly how long they will run, etc.

Since the assembler does not know the actual location where the program will be loaded, it cannot make the necessary changes in the addresses used by the program. However, the assembler can identify for the loader those parts of the object program that need modification.

An object program that contains the information necessary to perform this kind of modification is called a relocatable program. Likewise, if we loaded the program beginning at address 7420, the JSUB instruction would need to be changed to 4B108456 to correspond to the new address of RDREC.

Note that no matter where the program is loaded, RDREC is always 1036 bytes past the starting address of the program. This means that **we can solve the relocation problem** in the following way:

1) When the assembler generates the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program. This is the reason we initialised the location counter to 0 for the assembly.

2) The assembler will also produce a command for the loader, instructing it to add the beginning address fo the program to the address field in the JSUB instruction at load time.

Ques 11) Discuss SIC/XE machine architecture in detail.

Or

What are the various addressing modes supported by SIC/XE? With the help of an example, explain how to find target address during assembling in each case. (2018 [09])

Or

With reference to the standard SIC/XE model, discuss the data formats, instruction formats and addressing modes.

Or

Write notes on the architecture of SIC/XE. (2017 [04])

Or

Explain with suitable examples, how the different instruction formats and addressing modes of SIC/XE are handled during assembling. (2017 [05])

Or

List and explain the different addressing modes and instruction formats used in SIC/XE architecture. (2019[05])

Or

Explain the architecture of an SIC machine. (2020[05])

Ans: SIC/XE Machine Architecture

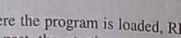
This is same as SIC with certain additional components and features. XE stands for Extra equipment.

1) **Memory:** Maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} bytes).

2) **Registers:** Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC:

Mnemonic	Number	Special Use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

3) **Floating-Point Data Type:** There is a 48-bit floating-point data type, $F=2^{(e-1024)}$.



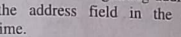
4) **Instruction Formats:** The new set of instruction formats for SIC/XE machine architecture are as follows:

i) **Format 1 (1 Byte):** It contains only operation code (straight from table).

For example, RSUB (Return to subroutines)

Opcode	
0100	0000
4	C

ii) **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).



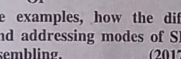
For example, COMPR A,S (Compare the contents of register A and S).

Opcode	A	S
1010 0000	0000	0100

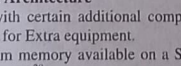
iii) **Format 3 (3 Bytes) e = 0:** First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand.

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 (0 for 3 and 1 for 4).

Bits i and n used to specify target address calculation.



For example, LDA #3 (Load 3 to Accumulator A)



There are some cases in format 3:

a) **Case (i):** If i = 1, n = 0 target address is used as operand value. No memory reference performed. This is called "Immediate Addressing". Immediate Addressing is indicated by prefix #.

Target address = Operand value

For example, If TA = 10, then operand value = 10

b) **Case (ii):** If i = 0, n = 1word given by target address location is fetched and value in word is taken as address of operand value. This is called "**Indirect Addressing**". Immediate Addressing is indicated by prefix #.

Value contained location in word = operand value

For example, ADD X, [500]

Word in memory location 500 is fetched. It gives address of operand. Second operand is given in indirect addressing mode.

c) **Case (iii):** If i = 0, n = 0 or i = 1, n = 1 target address is the location of operand. This is called "**Simple Addressing**".

TA = Location of operand

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

6 bit	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	20 bit
Opcode	n	i	x	b	p	e	address

Flag bits

For example, +JSUB RDREC (jump to the address 1036)

6 bit	1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	20 bit
0000 10	1	1	0	0	0	1	0000 0001 0000 0011 0110
Opcode	n	i	x	b	p	e	

4 B 1 0 1 0 3 6 Object Code

5) **Addressing Modes and Flag Bits:** Five possible addressing modes plus the combinations are as follows:

i) **Direct (x, b, and p All Set to 0):** Operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 one can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.

ii) **Relative (Either b or p Equal to 1 and the Other One to 0):** The address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1).

iii) **Immediate (i = 1, n = 0):** The operand value is already enclosed on the instruction (i.e., lies on the last 12/20 bits of the instruction).

iv) **Indirect (i = 0, n = 1):** The operand value points to an address that holds the address for the operand value.

v) **Indexed (x = 1):** Value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

Two relative addressing modes are available for format 3 instruction. Indexing cannot be used with immediate or indirect addressing modes. The two relative addressing modes are:

i) Base Relative addressing mode, and

ii) Program Counter Relative addressing mode.

Note: Target address is also called "**Effective address**".

Mode	Indication	Target Address Calculation
Base Relative Addressing Mode	b = 1 P = 0	**TA = Displacement + (B)** B – Base Register Displacement is 12 bit unsigned register. Displacement lies between 0 to 4095
Program Counter Relative Addressing Mode	b = 0 P = 1	**TA = Displacement + (PC)** PC – program counter Displacement is 12 bit signed integer. Displacement lies between – 2048 to 2047.
Direct Addressing Mode	b = 0, P = 0 (Format 4 instruction) b = 0, P = 0 (Format 3 instruction)	**TA = address field of format 4 instruction** **TA = Displacement field value of format 3 instruction**
Base Relative Indexed Addressing Mode	b = 1, P = 0 X = 1	**TA = Displacement + (B) + (X)** B – Base register X – Index register Displacement is 12 bit unsigned register. Displacement lies between 0 to 4095
Program Counter Relative Indexed Addressing Mode	b = 0, P = 1 X = 1	**TA = Displacement + (PC) + (X)** PC – program counter X – Index Register Displacement is 12 bit signed integer. Displacement lies between – 2048 to 2047

The given below shows the contents of registers B, PC, X and of selected memory location.

3030	003600
3600	103000
6390	00C303
C303	003030

(B) = 006000
(PC) = 003000
(X) = 000090

Note: All the Value of B, PC, and X and the contents of

Binary

Program Counter Relative Addressing: n i x b p e disp/address — 1 1 0 0 1 0 0110 0000 0000

Target Address

TA = (PC) + disp
= 003000 + 60C = 3600

With reference to the given above, in address 3600 the value loaded in the register is 103000.

Base Indexed Addressing: n i x b p e disp/address — 1 1 1 1 0 0 0011 0000 0000

Target Address

TA = (B) + (X) + disp
= 006000 +00090 +300 = 6390

With reference to the given, in address 6390 the value loaded in the register is 00C303.

Program Counter Relative Addressing: n i x b p e disp/address — 1 0 0 0 1 0 0000 0011 0000

Target Address

TA = (PC) + disp
= 003000 + 30 = 3030

With reference to given above, in address 3030 the value loaded in the register is 003600.

Direct Addressing: n i x b p e disp/address — 0 1 0 0 0 0 0000 0011 0000

Target Address

TA = displacement
= 30

Program Counter Relative Addressing: n i x b p e disp/address — 0 0 0 0 1 1 0110 0000 0000

Target Address

TA = (PC) + disp
= 003000 + 600
= 3600

With reference to the given above, in address 3600 the value loaded in the register is 103000.

Direct Addressing: n i x b p e disp/address — 1 1 0 0 0 1 0000 1100 0011 0000 0011

Target Address

TA = displacement
= 0C303

The value loaded in the register for the address 0C303

6) **Instruction Set:** Various instructions are available to load and store register. They can perform arithmetic operation between two registers. The instructions are as follows:

i) **Instruction that Load and Store New Register 'S'**

a) **LDB** – Load the register 'S' with some value. **For example,** LDBx – Load value of x into register B.

b) **STB** – Store the register 'S' content into some variable. **For example,** STBx – Store the register 'S' content into variable x.

ii) **Instruction those Perform Floating Point Arithmetic Operation**

a) **ADDF** b) **SUBF** c) **MULF** d) **DIVF**

Note: "F" represents floating point register.

For example, ADDFB register 'B' content is added with Accumulator content and result is left with accumulator.

iii) **Instruction that take Operand from Register:** RMO – Register move. **For example,** RMO S,B Register S content is moved to 'B' register.

iv) **Instruction which Perform Register to Register Arithmetic Operation**

a) ADDR b) SUBR c) MULR d) DIVR

Note: "R" represents register.

For example, ADDR S,B add value of register B with register S and store result in register B.

7) **Input and Output:** The SIC/XE supports all the I/O instructions in the standard version and follows the same mechanism for input and output. In addition to this, there are special I/O Channels which are utilised for data transfer when the CPU is involved in another process at the same time. Channels control the associated I/O devices. There can be a maximum of 16 I/O Channels each supporting a maximum of 16 devices. The sequence of operations to be performed by a channel is controlled by a Channel Program. Read Data (RD) and Write Data (WD) are used to read and write data from or to the specified I/O device.

SIO	Instruction is used to Start an I/O Channel number
TIO	Instruction is used to Test an I/O Channel number
HIO	Instruction is used to Halt an I/O Channel number

Ques 12) Differentiate SIC/XE with SIC machine.

Ans: Difference between SIC and SIC/XE

Basis	SIC	SIC/XE
Registers	Only 5 registers are used, which are A, X, L, SW and PC	There are 9 registers are used, which are A, X, L, SW, PC, B, T and F.
Floating Point Hardware	There is no floating point hardware.	Floating point hardware is used.
Instruction Format	Only one instruction format is used.	There are four different types of instructions format.
Addressing	There are two	There are many more

Ques 13) What are assembler directives? Describe any four assembler directives in detail.

Or

What are assembler directives? List any three assembler directives in SIC machine. (2017 [04])

Ans: Assembler Directives

Assembler directives are pseudo instructions. They provide instructions to the assembler itself. They are not translated into machine operation codes. The assembler can also process assembler directives. Assembler directives (or pseudo-instructions) provide instructions to the assembler itself. They are not translated into machine instructions. The SIC and SIC/XE assembler language has the following **assembler directives**:

Name of Directive	Function
START	Specify name and starting address for the program.
END	Indicate the end of the source program and (optionally) specify the first executable instruction in the program.
BYTE	Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
WORD	Generate one-word integer constant.
RESB	Reserve the indicate number of bytes for a data area.
RESW	Reserve the indicate number of words for a data area.

For example,

```
COPY    START   1000    ← Copy the file from input to output
        -   -   -
        -   -   -
        -   -   -
BUFFER  RESB    4096    ← 4096 – BYTE Buffer area
INPUT   BYTE    X'F1'   ← Code for input device
MAXLEN  WORD    4096    ← Specify maximum length
        END     FIRST   ← End of program
```

Ques 14) With the help of an example explain the use of BASE assembler directive. (2020[04])

Ans: BASE Assembler Directive

The assembler assumes for addressing purposes that register B contains this address until it encouters another BASE statement. Later in the program, it may be desirable to use register B for another purpose(for example, as temporary storage for a data value). In such a case, the programmer must use another assembler directive(perhaps NOBASE) to inform the assembler that the contents of the base register can no longer be relied upon for addressing. It is important to understand that BASE and NOBASE are assembler directives and produce no executable code. The programmer must provide instructions that load the proper value into the base register during execution. If this is not done properly, the target address calculation will not produce the correct operand address.

For example, the instruction

```
160 104E    STCH    BUFFER, X   57C003
```

is a typical example of base relative assembly. According to the BASE statement, register B will contain 0033(the address of LENGTH) during execution. The address of BUFFER is 0036. Thus the displacement in the instruction

Ques 15) Give the purpose of following assembler directives with examples: (2020[03])

1) USE 2) CSECT

Ans: USE

It is used to divide the program in to many blocks called program blocks. The assembler directive USE indicates which portions of the source program belong to the various blocks. At the beginning of the program, statements are assumed to the part of the unnamed (default) block; if no USE statements are included, the entire program belongs to this single block.

For example,

Line	Source Statement				
92	-	USE	CDATA		
95	RETADR	RESW	1		
100	LENGTH	RESW	1		LENGTH OF RECORD
103		USE	CBLKS		

The USE statement on line 92 signals the beginning of the block named CDATA. Source statements are associated with this block until the USE statement on line 103, which begins the block named CBLKS. The USE statement may also indicate a continuation of a previously begun block.

CSECT

It is used to divide the program into many control sections. **For example,** the first section continues until the CSECT statement on line 109. This assembler directive signals the start of a new control section named RDREC.

```
109 RDREC CSECT
```

Ques 16) Explain the data movement operations in SIC and SIC/XE machine.

Ans: Data Movement Operations

The two programs of data movement operations for SIC and SIC/XE are shown below. All the data movement done using registers; hence there are no memory-to-memory move instructions.

Program 1 shows two examples of data movement in SIC. In the first example, a 3-byte word is moved by loading it into register A and then storing the register at the required destination.

In the second example, a single byte of data is moved using the instructions LDCH and STCH, load and store character respectively. There is also four different methods of defining storage for data items in the SIC shown in program 1.

The statement WORD reserves one word of storage, which is initialized to a value defined in the field of the statement (here is 5). The statement RESW reserves one or more words of storage for use by the program (here is ALPHA).

The statements BYTE and RESB perform similar storage-definition functions for data items that are characters instead of words. Here the CHARZ is a 1-byte data item whose value is intialised to the character 'Z' and C1 is a 1-

Program 1

SIC: Sample Data Movement Operations

```
        LDA     FIVE    LOAD CONSTANT 5 INTO
                        REGISTER A
        STA     ALPHA   STORE IN ALPHA
        LDCH    CHARZ   LOAD CHARACTER 'Z'
                        INTO REGISTER A
        STCH    C1      STORE IN CHARACTER
                        VARIABLE C1
        .
        .
        .
ALPHA   RESW    1       ONE-WORD VARIABLE
FIVE    WORD    5       ONE-WORD CONSTANT
CHARZ   BYTE    C'Z'    ONE-BYTE CONSTANT
C1      RESB    1       ONE-BYTE VARIABLE
```

Program 2 shows two examples of data movement in SIC/XE. Here the scheme used for loading is immediate addressing. The operand field for this instruction contains the flag # and the data value to be loaded. Similarly, the character 'Z' is placed into register A by using immediate addressing to load the value 90, which is the decimal value of the ASCII code that is used internally to represent the character Z.

Program 2

SIC/XE: Sample Data Movement Operations

```
        LDA     #5      LOAD VALUE 5 INTO
                        REGISTER A
        STA     ALPHA   STORE IN ALPHA
        LDA     #90     LOAD ASCII CODE FOR 'Z'
                        INTO REG A
        STCH    C1      STORE IN CHARACTER
                        VARIABLE C1
        .
        .
        .
ALPHA   RESW    1       ONE-WORD VARIABLE
C1      RESB    1       ONE-BYTE VARIABLE
```

Ques 17) Explain the arithmetic operation in SIC and SIC/XE machine.

Ans: Arithmetic Operations

The two programs of arithmetic operations for SIC and SIC/XE are shown below.

The **program 3** shows the two examples of arithmetic instructions for SIC. All the arithmetic operations are performed using register A, with the result being left in the register A. Thus this sequence of instructions stores the value (ALPHA+INCR−1) in BETA and the value (GAMMA+INCR−1) in DELTA.

Program 3

SIC: Sample Arithmetic Operations

```
        LDA     ALPHA   LOAD ALPHA INTO
                        REGISTER A
        ADD     INCR    ADD THE VALUE OF
                        INCR
        SUB     ONE     SUBTRACT 1
        STA     BETA    STORE IN BETA
        LDA     GAMMA   LOAD GAMMA INTO
                        REGISTER A
        ADD     INCR    ADD THE VALUE OF
                        INCR
        SUB     ONE     SUBTRACT 1
        STA     DELTA   STORE IN DELTA
        .
        .
        .
ONE     WORD    1       ONE-WORD CONSTANT
                        ONE-WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

Program 4 shows the same calculation as above for the SIC/XE. The value of INCR is loaded into register S initially and the register-to-register instruction ADDR is used to add this value to the register A when it is needed. This avoids having to fetch INCR from memory each time it is used in a calculation, which may make the program more efficient. Immediate addressing is used for the constant 1 in the subtraction operations.

Program 4

SIC/XE: Sample Arithmetic Operations

```
        LDS     INCR    LOAD VALUE OF INCR INTO
                        REGISTER S
        LDA     ALPHA   LOAD ALPHA INTO REGISTER A
        ADDR    S, A    ADD THE VALUE OF INCR
        SUB     #1      SUBTRACT 1
        STA     BETA    STORE IN BETA
        LDA     GAMMA   LOAD GAMMA INTO REGISTER A
        ADDR    S, A    ADD THE VALUE OF INCR
        SUB     #1      SUBTRACT 1
        STA     DELTA   STORE IN DELTA
        .
        .
        .
                        ONE-WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

Ques 18) Explain the looping and indexing operation in SIC and SIC/XE machine.

Ans: Looping and Indexing Operations

Program 5 shows a loop that copies one 11-byte character string to another. The index register (X) is initiated to 0 before the loop begins. During the first execution of the loop, the target address for the LDCH instruction will be the address of the first byte of STR1. Similarly, the STCH instruction will store the character being copied into the first byte STR2. The next instruction, TIX, performs two functions.

1) First it adds 1 to the value in register X and then
2) It compares the new value of register X to the value of operand.

The condition code is set to indicate the result of this comparison. The JLT instruction jumps if the condition code is set to "less than". Thus, the JLT causes a jump back to the beginning of the loop if te new value in register X is less than 11.

During the second execution of the loop, register X will contain the value 1. Thus, the target address for the LDCH

instruction will be the second byte of SRT1 and the target address for the STCH instruction will be the second byte of STR2.

The TIX instruction will again add 1to the value in register X and the loop will continue in this way until all 11 bytes have been copied from STR1 to STR2.

Program 5

SIC: Sample Looping and Indexing Operations

```
        LDX   ZERO      INITIALIZE INDEX REGISTER TO 0
MOVECH  LDCH  STR1, X   LOAD CHARACTER FROM STR1 INTO REG A
        STCH  STR2, X   STORE CHARACTER INTO STR2
        TIX   ELEVEN    ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT   MOVECH    LOOP IF INDEX IS LESS THAN 11
        .
        .
STR1    BYTE  C 'TEST   11-BYTE STRING CONSTANT
              STRING'
STR2    RESB  11        11-BYTE VARIABLE
        .               ONE-WORD CONSTANTS
ZERO    WORD  0
ELEVEN  WORD  11
```

Program 6 shows the same loop as above for SIC/XE. The main difference is that the instruction TIXR is used in place of TIX.

Program 6

SIC/XE: Sample Looping and Indexing Operations

```
        LDT   #11       INITIALIZE REGISTER T TO 11
        LDX   #0        INITIALIZE INDEX REGISTER TO 0
MOVECH  LDCH  STR1, X   LOAD CHARACTER FROM STR1 INTO REG A
        STCH  STR2, X   STORE CHARACTER INTO STR2
        TIXR  T         ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT   MOVEC     LOOP IF INDEX IS LESS THAN 11
              H
        .
STR1    BYTE  C 'TEST   11-BYTE STRING CONSTANT
              STRING'
STR2    RESB  11        11-BYTE VARIABLE
```

Programs 7 and 8 show the examples of looping and indexing operations.

Program 7

SIC: Sample Looping and Indexing Operations

```
        LDA   ZERO      INITIALIZE INDEX VALUE TO 0
        STA   INDEX
ADDLP   LDX   INDEX     LOAD INDEX VALUE INTO REGISTER X
        LDA   ALPHA, X  LOAD WORD FROM ALPHA INTO REGISTER A
        ADD   BETA, X   ADD WORD FROM BETA
        STA   GAMMA,    STORE THE RESULT IN A WORD
              X         IN GAMMA
        LDA   INDEX     ADD 3 TO INDEX VALUE
        ADD   THREE
        COMP  K300      COMPARE NEW INDEX VALUE TO 300
        JLT   ADDLP     LOOP IF INDEX IS LESS THAN 300
        .
INDEX   RESW  1         ONE-WORD VARIABLE FOR INDEX VALUE
        .               ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100
        .               ONE-WORD CONSTANTS
ZERO    WORD  0
K300    WORD  300
THREE   WORD  3
```

Program 8

SIC/XE: Sample Looping and Indexing Operations

```
        LDS   #3        INITIALIZE REGISTER S TO 3
        LDT   #300      INITIALIZE REGISTER T TO 300
        LDX   #0        INITIALIZE INDEX REGISTER TO 0
ADDLP   LDA   ALPHA,    LOAD WORD FROM ALPHA
              X         INTO REGISTER A
        ADD   BETA, X   ADD WORD FROM BETA
        STA   GAMMA     STORE THE RESULT IN A
              ,X        WORD IN GAMMA
        ADDR  S, X      ADD 3 TO INDEX VALUE
        COMPR X, T      COMPARE NEW INDEX VALUE TO 300
        JLT   ADDLP     LOOP IF INDEX VALUE IS LESS THAN 300
        .
        .               ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW  100
BETA    RESW  100
GAMMA   RESW  100
```

Ques 19) Explain the I/O operations in SIC and SIC/XE machine.

Ans: Input and Output Operations

Program 9 shows the example of input and output on SIC. This program fragment reads 1 byte of data from device F1 and copies it to device 05. The actual input of data is performed using the RD instruction. Executing the RD instruction transfer 1 byte of data from this device into the rightmost byte of register A. Before the RD can be executed the input device must be ready to transmit the data.

Output is performed in the same way. First the program uses TD to check whether the output device is ready to receive a byte of data. Then the byte to be written is loaded into the rightmost byte of register A and the WD instruction is used to transmit it to the device. The same procedure is used in SIC/XE.

Program 9

SIC: Sample Input and Output Operations

```
INLOOP  TD    INDEV     TEST INPUT DEVICE
        JEQ   INLOOP    LOOP UNTIL DEVICE IS READY
        RD    INDEV     READ ONE BYTE INTO REGISTER
```



```
        STCH  DATA      STORE BYTE THAT WAS READ
        .
OUTLP   TD    OUTDEV    TEST OUTPUT DEVICE
        JEQ   OUTLP     LOOP UNTIL DEVICE IS READY
        LDCH  DATA      LOAD DATA BYTE INTO REGISTER A
        .
        WD    OUTDEV    WRITE ONE BYTE TO OUTPUT DEVICE
        .
INDEV   BYTE  X'F1'     INPUT DEVICE NUMBER
OUTDEV  BYTE  X'05'     OUTPUT DEVICE NUMBER
DATA    RESB  1         ONE-BYTE VARIABLE
```

Ques 20) Explain the subroutine call and record input operations in SIC and SIC/XE machine.

Ans: Subroutine Call and Record Input Operations

The following programs shows the, how the instructions can be used to read a 100-byte record from an input device into memory. The read operation in this example is placed in a subroutine. This subroutine is called from the main program using the JSUB instruction. At the end of the subroutine there is an RSUB instruction, which returns control to the instruction that follows the JSUB.

The READ subroutine itself consists of a loop. Each execution f this loop reads 1 byte of data from the input device. The bytes of data that are read are stored in a 100-byte buffer area labelled RECORD.

Program 10

SIC: Sample Subroutine Call and Record Input Operations

```
        JSUB  READ      CALL READ SUBROUTINE
        .
        .
        .               SUBROUTINE TO READ 100-BYTE RECORD
READ    LDX   ZERO      INITIALIZE INDEX REGISTER TO 0
RLOOP   TD    INDEV     TEST INPUT DEVICE
        JEQ   RLOOP     LOOP IF DEVICE IS BUSY
        RD    INDEV     READ ONE BYTE INTO REGISTER A
        STCH  RECORD,   STORE DATA BYTE INTO RECORD
              X
        TIX   K100      ADD 1 TO INDEX AND COMPARE TO 100
        JLT   RLOOP     LOOP IF INDEX IS LESS THAN 100
        RSUB            EXIT FROM SUBROUTINE
        .
        .
INDEV   BYTE  X'F1'     INPUT DEVICE NUMBER
RECORD  RESB  100       100-BYTE BUFFER FOR INPUT RECORD
        .               ONE-WORD CONSTANTS
ZERO    WORD  0
K100    WORD  100
```

Program 11 shows the same technique as above for SIC/XE.

Program 11

SIC/XE: Sample Subroutine Call and Record Input Operations

```
        JSUB  READ      CALL READ SUBROUTINE
        .
        .
        .               SUBROUTINE TO READ 100-BYTE RECORD
READ    LDX   #0        INITIALIZE INDEX REGISTER TO 0
        LDT   #100      INITIALIZE REGISTER T TO 100
RLOOP   TD    INDEV     TEST INPUT DEVICE
        JEQ   RLOOP     LOOP IF DEVICE IS BUSY
        RD    INDEV     READ ONE BYTE INTO REGISTER A
        STCH  RECORD,   STORE DATA BYTE INTO RECORD
              X
        TIXR  T         ADD 1 TO INDEX AND COMPARE TO 100
        JLT   RLOOP     LOOP IF INDEX IS LESS THAN 100
        RSUB            EXIT FROM SUBROUTINE
        .
        .
INDEV   BYTE  X'F1'     INPUT DEVICE NUMBER
RECORD  RESB  100       100-BYTE BUFFER FOR INPUT RECORD
```

Ques 21) Consider the memory contents shown in the following figure: (2020[04])

.	.	(X)	000090
3030	003600	(PC)	003000
.	.	(B)	006000
3600	103000		
.	.		
6390	00C303		
.	.		
C303	003030		

What would be loaded to register A with the following instructions?
i) 03C300
ii) 022030

Ans:

Machine Instruction										Value Loaded into Register A
Hex	Binary								Target address	
	op	n	i	x	b	p	e	disp/address		
03C300	000000	1	1	1	1	0	0	0011 0000 0000	6390	00C303
022030	000000	1	0	0	0	1	0	0000 0011 0000	3030	103000

Assembly Language Programming and Assemblers

SIC AND SIC/XE PROGRAMMING EXAMPLES

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

Ans:

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

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

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

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

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

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

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

Ans:

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

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

Or

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

Or

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

Ans:

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

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

Ans: Assembly Code

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


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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Or

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

(2017 [03])

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

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

Ans: Assembly Coue

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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


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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Assembly Code

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

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

Ans: Sequence of Instructions for SIC

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

Ques 23) Let A,B and C are arrays of 10 word each. Write a SIC/XE program to add the corresponding elements of A and B and store the result in C. (2018[06])

Ans:

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

ASSEMBLER

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

Ans: Assembler

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

Basic Functions of Assembler

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

Phase in Assemblers

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

1) Analysis Phase
2) Synthesis Phase

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

Table 2.1: Phases of an Assembler

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

Figure 2.1 illustrates the usage during each phase of assembly:

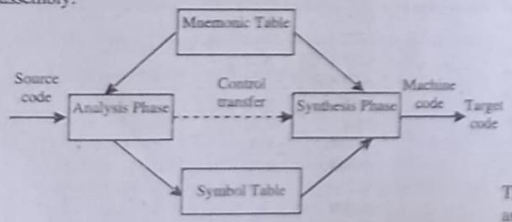


Figure 2.1: Phases of an Assembler

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

Ans: Assembler Output Format/Object Program Format

The simple object program format contains three types of records:

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

The format for header record is as follows:

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

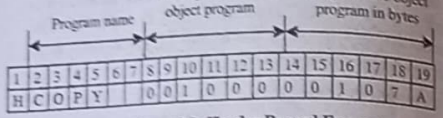


Figure 2.2: Header Record Format

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

The format for text record is as follows:

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

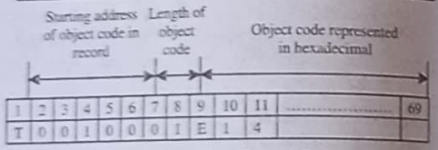


Figure 2.3: Text Record Format

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

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

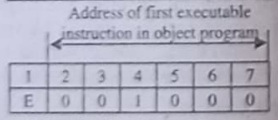


Figure 2.4: End Record Format

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

Ques 26) Describe the various types of assemblers.

Or

Define single-pass and two-pass assembler.

Or

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

Or

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

Ans: Types of Assemblers

Two types of assemblers are:

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

One-pass assemblers are used when:

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

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

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

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

Pass 1: (Define Symbols)

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

Note: Perform analysis of assembly language program.

Pass 2: (Assemble Instructions and Generate Object Code)

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

Note: Processes intermediate representation and produces target program.

Ques 27) Describe the assembler data structure.

Or

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

Ans: Assembler Data Structures

The data structures used by an assembler are as follows:

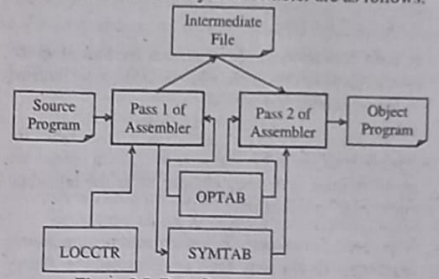


Figure 2.5: Data Structures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Or

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

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

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

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

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

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

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

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

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

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

```
1          START   200
2          MOVER   AREG, ='5'     200)  +04 1 211
3          MOVEM   AREG, A        201)  +05 1 217
4   LOOP   MOVER   AREG, A        202)  +04 1 217
5          MOVER   CREG, B        203)  +05 3 218
6          ADD     CREG, ='1'     204)  +01 3 212
7          ...
12         BC      ANY, NEXT      210)  +07 6 214
```



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

Figure 2.6: Assembly Program Illustrating the ORIGIN Directive

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

Table 2.2-a: OPTAB

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

Table 2.2-b: SYMTAB

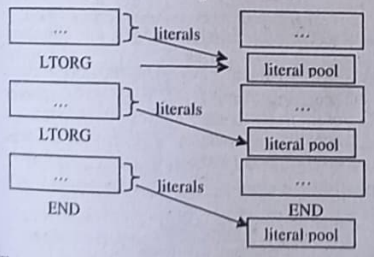
symbol	address	length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

Table 2.2-c: LITTAB

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

Ques 30) How does the assembler handle literals?
Or
Define LITTAB. Also write the difference between the SYMTAB and LITTAB

Ans: Handling of Literals by Assemblers



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

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

Literal Table (LITTAB)
For each used literal, LITTAB contains:
1) The literal name,
2) The operand value and length,
3) The address assigned to the operand when it is placed in a literal pool.

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

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

In Pass 2
1) The **operand address** for use in generating object code is obtained by searching LITTAB for each literal operand encountered.

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

SYMTAB

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

LITTAB

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

2) The data values specified by the literal in each literal pool are inserted at the appropriate places in the object program.
3) If a literal value represents an address in the program (e.g., location counter value), the assembler must also generate the appropriate **Modification record**.

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

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

Ans:

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

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

OPTAB

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

SYMTAB

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

LITTAB

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

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

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

Ans:

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

Generating object code for each instructions:

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

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

Symbol Table

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

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

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

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

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

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

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

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

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

Ques 34) Describe the assembly language statement format.

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

An assembly language statement has the following format:

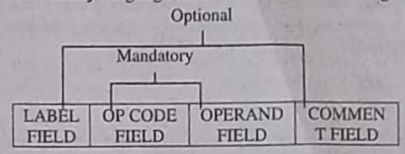


Figure 2.7: Assembly Language Program Statement Format

1) **Label Field:** It's placed at the beginning of the line. And assigns a symbolic name to the address corresponding to line. **For example,**
```
LOOP  ADD   R1,R1,#–1
      BRp   LOOP
```

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

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

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

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

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

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

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

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

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

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

Note: DS stands for Declare Storage and

For **declaring a constant**, Declare Constant (DC) statement can be used. **For example,** consider the following statement

```
ONE DC '1'
```

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

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

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

For example, consider the following statement

```
START 202
```

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

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

Ans: Hand Assembly of SIC/XE Program

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

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

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

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

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

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

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



```
             END    FIRST
```

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

Machine instruction

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

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

```
Loc      Source   Statement    Object code
     STRCP2  START  1000
     FIRST   LDT    #11
             LDX    #0
     MOVECH  LDCH   STR1,X
             STCH   STR2,X
             TIXR   T
```

```
HSTRCP2001000000027
T001000117500OB05000053A00857A010B8503B2FF5
T0010110B544553542053545249
E001000
```

```
             JLT    MOVECH
     STR1    BYTE   C'TEST STRING'
     STR2    RESB   11
             END    FIRST
```

Ans: Assembly Program to SIC/XE Object Code

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

Machine instruction

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

```
HSTRCPY00100000002B
T00100OF041025050900F54901A2C1028381003
T00100F0B5445535420535453494E47
T0010250600000000000B
E001000
```

Figure 2.8

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:

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

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

$= (-2048)_{10}$

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:**

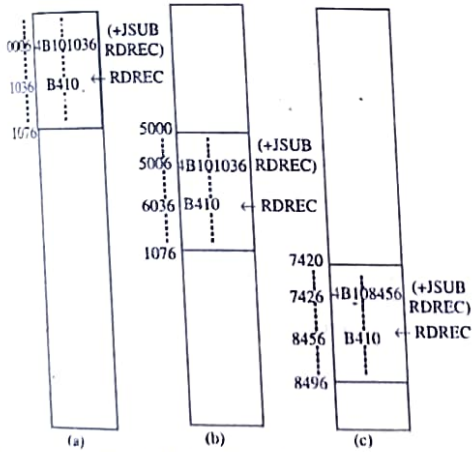


Figure 3.1: Illustration of Program Relocation

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

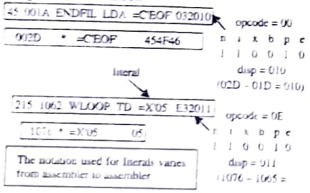
Explain the following machine independent features of assembler:

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.

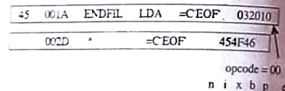
iii) A symbol whose value is given by **EQU** (or some similar assembler directive) may be either an absolute term or a relative term depending on the expression used to define its value.

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

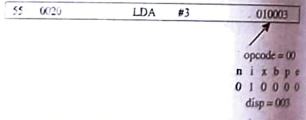
Ans: Difference between Literal and Immediate Operand

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   106B           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.

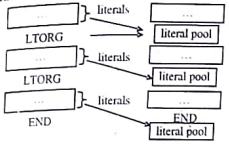
Or

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 predefined. 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

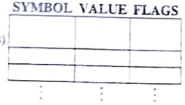
Where, **value** is a constant or an expression involving constants and previously defined symbols.

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

Ques 12) Define various types of expressions.

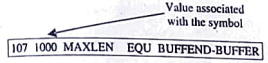
Or

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.**

```
                H,COPY  ,000000,001071
Lines 5-70      T,000000,1E,172063,4B2021,032060,290000,332006,
                4B203B,3F2FEE,032055,0F2056,010003
Lines 125-180   T,00001E,09,0F2048,4B2029,3E203F
Line 185        T,000027,1D,B410,B400,B440,75101000,E32038,
Lines 210-245   332FFA,DB2032,A004,332008,57A02F,B850
Line 253        T,000044,09,3B2FEA,13201F,4F0000
                T,000060,01,F1          CDATA = 66 + 06 = 6C
                T,00004D,19,B410,772017,E3201B,332FFA,53A016,
                DF2012,B850,3B2FEF,4F0000
                T,000063,04,454F46,05          CDATA = 66 + 07 = 6D
                E,000000
```

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.

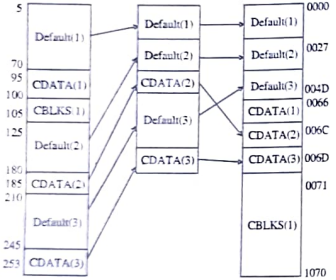


Figure 3.5: Multiple Program Blocks and Loading Process

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 th t 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

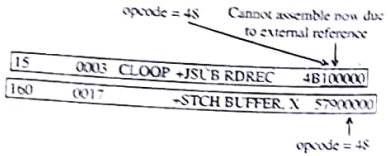
FIXUPP records are used to resolve external references, and to carry-out address modifications that are associated with relocation and grouping of segments within the program. This is similar to the function performed by the SIC/XE Modification records. However, FIXUPP records are substantially more 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?

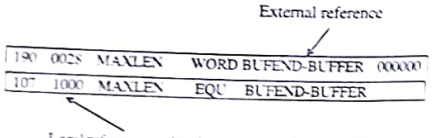
Ans: Using External References

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 reference
9          .
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**.

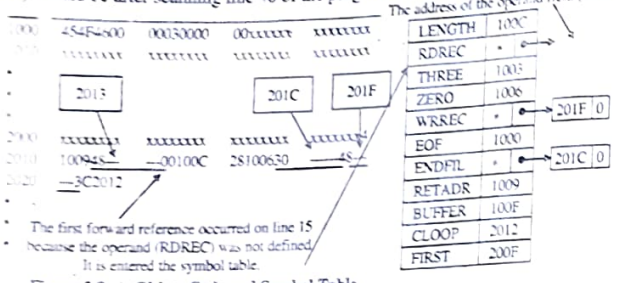


Figure 3.8(a): Object Code and Symbol Table

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.

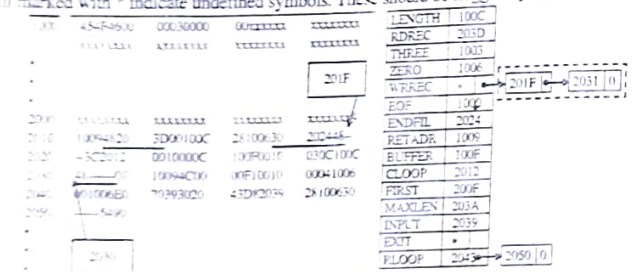


Figure 3.8 (b): After Scanning Line 60

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 assembler, 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.

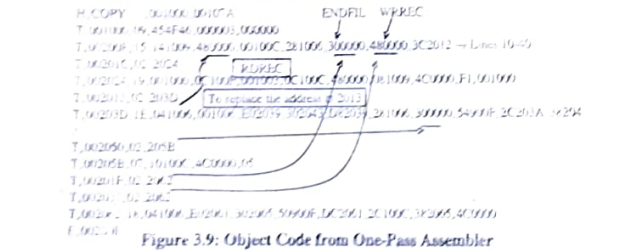


Figure 3.9: Object Code from One-Pass Assembler

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

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**.

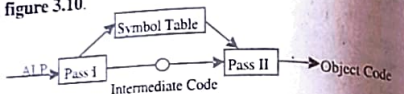


Figure 3.10: Multi-Pass Assembler

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.

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. It is possible for data items, though inconvenient.	Though forward referencing is much widely used forward references in symbol definition are not allowed. **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+1]
      b) srtab_no : = srtab_no+1;
      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 appearing 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

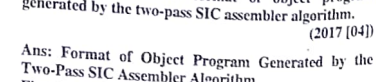


Figure 3.13: Object program

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.)