Module 1 (Introduction)

System Software Vs. Application Software, Different System Software– Assembler, Linker, Loader, Macro Processor, Text Editor, Debugger, Device Driver, Compiler, Interpreter, Operating System(Basic Concepts only). SIC & SIC/XE Architecture, Addressing modes, SIC & SIC/XE Instruction set, Assembler Directives. **SIC/XE Programs**

Introduction

The computer system components can be **hardware and software**. A **program** is a collection of instructions that can be executed by a computer to perform a specific task. **Software** is a collection of programs, procedures, routines associated with the operation of the computer system.

Software can be classified as: **Application Software and System Software**. Application software performs specific task for the end user as per the requirement. eg: , Photoshop, Ms-Office, etc.

System Software consists of variety of program that supports the operation of a computer. This software makes 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.

[Just for general understanding.(*During the programming course, you had used different system software's, you wrote programs in high level language like c++, using the* ***text editor*** *to create and modify the program. You translated these programs into machine language programs using* ***compiler.*** *The resulting machine language program was loaded into memory and prepared for execution by a* ***loader and linker.*** *You may have used a* ***debugger*** *to help to detect errors in*

the program. Later when using assembly language you may come across macro instructions to read and write data, or to perform higher level functions. There you have used ***assembler*** *which probably includes a* ***macro processor*** *to translate these programs into machine language. The translated programs were prepared for execution by linker and loader may test with debugger.*)]

The central theme is to understand the relationship between machine architecture and the system software. The design of an assembler, operating system etc. is greatly influenced by the architecture of the machine on which it is to run.

System Software V/s. Application Software

System Software	Application Software
1. System software is the type of software which is the interface between application software and the system.	1. Application software is the type of software which runs as per user request – runs on the platform provided by the system software
2. It is intended to support the operation and make use of the computer system	2. It is concerned mainly with the solution of the problem and makes use of the computer as a software tool.
3. Are developed in low level language which is more compatible with the system hardware in order to interact with.	3. Developed in high level languages
4. Installed on the computer when the operating system is installed	4. Used by user to perform a specific task
5. Less or no user interaction	5. More user interaction
6. Depends on machine architecture	6. Independent on machine architecture
7. Eg: Compiler, assembler, etc.	7. Eg: web browser, Photoshop, ms-office

Different Types of System Software

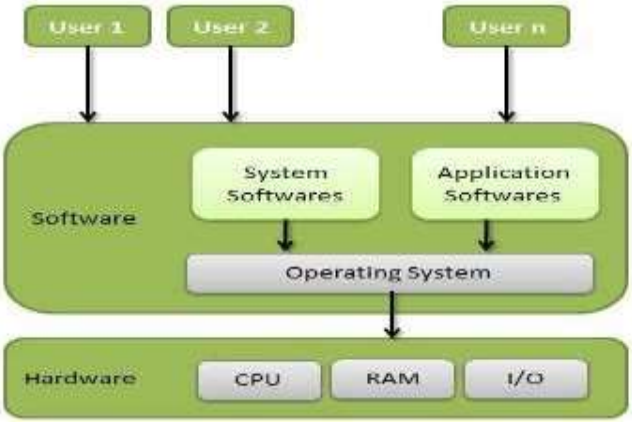
1. **Operating System**
2. **Device Drivers**
3. **Text Editor**

4. **Macro-processor**
5. **Language Translators**
    a. **Assembler**
    b. **Compiler**
    c. **Interpreter**
6. **Loader**
7. **Linker**
8. **Debugger**

(**marked in red should be studied in this semester in each module)

Operating System

Operating system provides a basis for application programs and acts as an intermediary between a user of a computer and the computer hardware by performing all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.



Examples of Operating System

Mac OS, Microsoft Windows *(you can add more examples)*

Functions of Operating System

1. **Memory Management**

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what parts are not in use?
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

2. Process Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
- Allocates/ Schedules the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

3. Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.

- Allocates the device in the efficient way.
- De-allocates devices.

4. File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

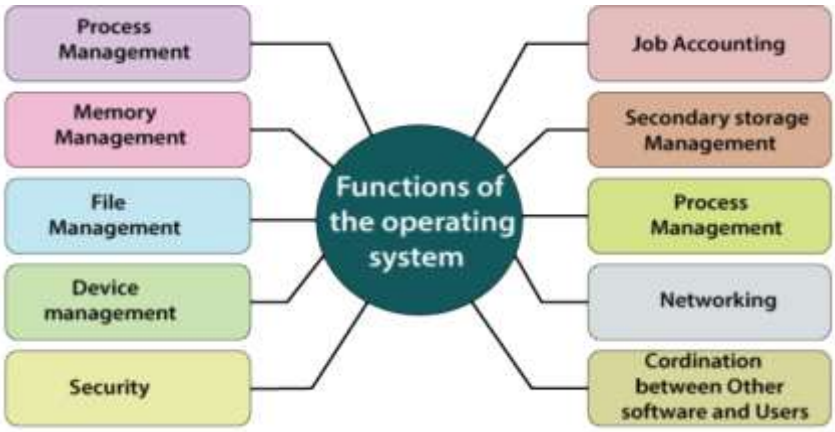
An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable(non-volatile) storage media

Following are some of the important activities that an Operating System performs −

1. **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
2. **Control over system performance** − Recording delays between request for a service and response from the system.
3. **Job accounting** − Keeping track of time and resources used by various jobs and users.
4. **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
5. **Coordination between other software's and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

6. **Secondary storage management** – is a task performed by an operating system in conjunction with the use of disks, tapes, and other secondary storage for a user's programs and data.



Types of Operating Systems

Operating systems are there from the very first computer generation and they keep evolving with time.

1. **Batch operating system**

   The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

   The problems with Batch Systems are as follows −

   - Lack of interaction between the user and the job.
   - CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
   - Difficult to provide the desired priority.

2. **Time-sharing operating systems**

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multi-programmed Batch Systems and Time-Sharing Systems is that in case of former, the objective is to maximize processor use, whereas in later, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if **n** users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most. The operating system perform CPU scheduling and multi-programming. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Time-sharing operating systems are as follows −

- Provides the advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows −

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

3. **Distributed operating System**

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows −

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

4. Network operating System

Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

- Centralized servers are highly stable.

- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

5. **Real Time operating System**

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

- Hard real-time systems

  Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

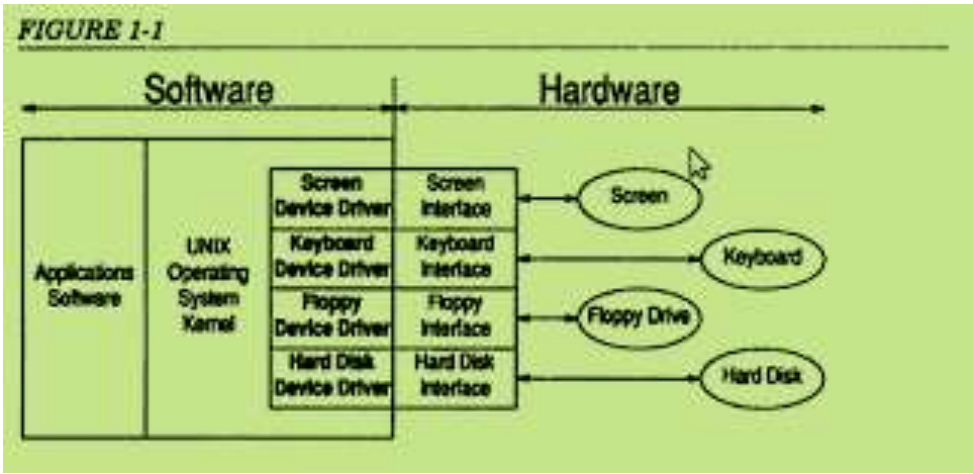
- Soft real-time systems

  Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects likes undersea exploration and planetary rovers, etc.

Device Drivers

A device driver is glue between OS and its I/O devices. Device Drivers act as translators which converts generic requests received from the operating system into commands that specific peripheral controllers can understand.

The application software makes system calls to the operating system requesting services. The operating system analyses these requests and when, necessary, issues requests to the appropriate device driver. The device driver in turn analyses the request from the operating system and when necessary, issues commands to the hardware interface to perform the operations needed to service the request.



Although the process may seem unnecessarily complex, the existence of device drivers actually simplifies the operating system considerably. Without device drivers operating system would be responsible for talking directly to the hardware. This would require the operating system's designer to include support for all of the devices that users might want to connect to their

computer. It would also mean that adding support for a new device would mean modifying the operating system itself.

By separating the device driver's functions from the operating system itself, the designer of the operating system can concern himself with issues that relate to the operation of the system as a whole. Furthermore, details related to individual hardware devices can be ignored and generic requests for I/O operations can be issued by the operating system to the device driver.

The device driver writer, on the other hand, does not have to worry about the many issues related to general I/O management, as these are handled by the operating system. All the device driver writer has to do is to take detailed device-independent requests from the operating system and to manipulate the hardware in order to fulfill the request.

Finally, the operating system can be written without any knowledge of the specific devices that will be connected. And the device driver writer can connect any I/O device to the system without having to modify the operating system. The operating system views all hard disks through the same interface, and the device driver writer can connect almost any type of disk to the system by providing a device driver so that the operating system is happy.

The result is a clean separation of responsibilities and the ability to add device drivers for new devices without changing the operating system. Device drivers provide the operating system with a standard interface to non-standard I/O devices.

(In detail we will deal in Module 5)

Text Editors

A text editor is a piece of computer software for editing plain text. Text editors are often provided with the operating system or software development packages, and are used to change configuration files and programming language source codes. Text editors are considered as the primary interface to the computer for all types of "knowledge workers" as they compose, organize, study and manipulate computer-based information.

The fundamental functions in editing are travelling, editing, viewing and display. **Travelling** implies movement of the editing context to a new position within the text. This may be done

explicitly by the user (eg: the line number command of a line editor) or may be implied in a user command (eg: the search command of a stream editor). **Viewing** implies formatting the text in a manner desired by the user. This is an abstract view, independent of the physical characteristics of an I/O device. The display component maps this view into the physical characteristics of the display device being used. This determines where a particular view may appear on the user's screen. The separation of viewing and display functions give rise to interesting possibilities like multiple windows on the same screen, concurrent edit operations using the same display terminal, etc. A simple text editor may choose to combine the viewing and display functions.

Text editors come in the following forms:

- **Line Editors**

  The scope of edit operations in a line editor is limited to a line of text. The line is designated *positionally,* eg: by specifying its serial number in the text, or *contextually*, eg: by specifying a context which uniquely identifies it. The primary advantage of line editors is their simplicity.
- **Stream Editors**

  Stream editors view the entire text as a stream of characters. This permit edits operations to cross line boundaries. Stream editors typically support character, line and context oriented commands based on the current editing context indicated by the position of a text pointer. The pointer can be manipulated using positioning and search commands

  - Line and Stream editors typically maintain multiple representations of text. One representation (the display form) shows the text as a sequence of lines. The editor maintains an internal form which is used to perform the edit operations. This form contains end-of-line characters and other edit characters. The editor ensures that these representations are compatible at every moment.
- **Screen Editors**

  A line or stream editors does not display the text in the manner it would appear if printed. A screen editor uses the what-you-see-is-what-you-get principle in editor design. The editor displays a screen full of text at a time. The user can move the cursor over the screen, position it at the point where he desires to perform some editing and proceed with the

editing directly. Thus it is possible to see the effect of an edit operation on the screen. This is very useful while formatting the text to produce printed documents.

- **Word Processor**

  Word Processors are basically document editors with additional features to produce well formatted hard copy output. Essential features of word processors are commands for moving sections of text from one place to another, merging of text, and searching and replacement of words. Many word processors support a spell-check option. With the advent of personal computer, word processors have seen widespread use amongst authors, office personnel and computer professionals. WordStar is a popular editor of this class.

- **Structure Editors**

  A structure editor incorporates an awareness of the structure of a document. This is useful in browsing through a document, eg: if a programmer wishes to edit a specific function in a program file. The structure is specified by the user while creating or modifying the document. Editing requirements are specified using the structure. A special class of structure editors, called syntax-directed editors, is used in programming environments.

  - Contemporary editors support a combination of line, string and screen editing functions. The Vi editors of Unix and the editors in desktop publishing systems are typical examples.

Macro Processor

To relieve programmers of the need to repeat identical parts of their program, operating systems provide a macro processing facility, which permits the programmer to define an abbreviation for a part of his program and to use the abbreviation in his program. Macro instructions (Macros) are single-line abbreviations for a group of instructions. The macro processor treats the identical parts of the program defined by the abbreviation as a macro definition and saves the definition. The macro processor substitutes the definition for all occurrences of the abbreviation (macro call) in the program

In addition to help programmers abbreviate their programs, macro facilities have been used as general text handlers and for specializing operating systems to individual computer installations. In specializing operating systems (systems generation), entire operating system is written as a

series of macro definitions. To specialize the operating system, a series of macro calls are written. These are processed by the macro processor by substituting the appropriate definitions, thereby producing all the programs for an operating system.

- Macros differ from subroutines in one fundamental respect. Use of a macro name in the mnemonic field of an assembly statement leads to its expansion, whereas use of a subroutine name in a call instruction leads to its execution.

Language Translators

Computers are basically machines that follow very specific and primitive instructions. Language translators translate the source program written in either high-level language or low-level language into machine understood object program. Various language translators are:

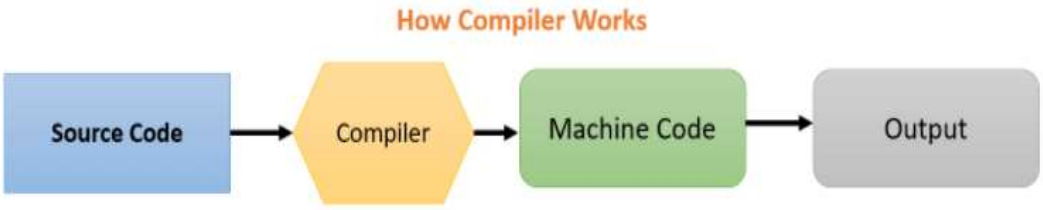
1. Assembler*(Detailed in Module 2 and 3)*

At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of 1's and 0's (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions.

Programmers found it difficult to write or read programs in machine language. In their quest for a more convenient language they began to use a mnemonic (symbol) for each machine instructions, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as Assemblers were written to automate the translation of assembly language into machine language. The input to an assembler program is called source program. The output is a machine language translation called object program.

2. Compiler

A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient which is optimized for execution time and memory space.



Characteristics of a Compiler

There are two characteristics features of the compiler. They are:

- Machine Dependent

  Each compiler has its own machine language. Compilers are machine dependent – means that you cannot use a compiler in CRAY, to translate a program that could be understood by the IBM-360.

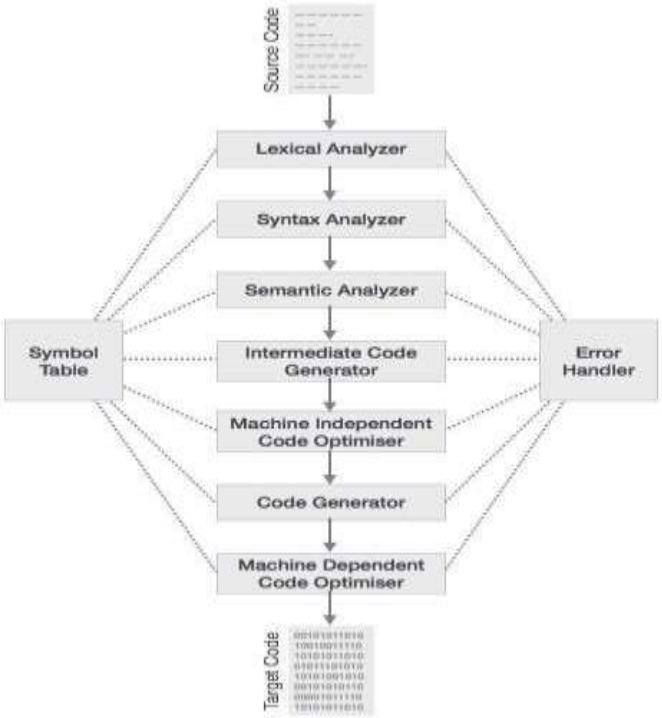
- Language Dependent

  A PASCAL compiler cannot be used to translate a program written in C language.

Phases of a Compiler

The compiling process includes basic translation mechanisms and error detection. Compiler process goes through lexical, syntax, and semantic analysis at the front end, and code generation

and optimization at a back-end. (***Just study the diagram…you will study the explanation as subject in higher semester)***



Types of Compilers (*mentioned because question asked in previous yr university exam)*

- Incremental compiler
- Cross Compiler
- Load & Go Compiler
- Threaded Code compiler
- Stage Compiler
- Just-in-time Compiler
- Parallelizing Compiler

According to passes, Compilers can be:

- Single pass compiler
- Multi-pass compiler

Key Differences between Compiler and Assembler

1. The **compiler** generates assembly code and some compilers can also directly generate executable code whereas, the **assembler** generates reloadable machine code.

2. The compiler takes as input the **preprocessed code** generated by preprocessor. On the other hands, the assembler takes **assembly code** as input.

3. The compilation takes place in two phases that are **analysis phase** and **synthesis phase**. In analysis phase, the input goes through **lexical analyzer, syntax analyzer, semantic analyzer** whereas; the synthesis analysis takes place via **intermediate code generator, code optimizer, code generator**. On the other hands, assembler passes the input through **two phases**. The first phase detects the identifiers and allots addresses to them in the second phase the assembly code is translated to binary code.

4. The assembly code generated by the compiler is a **mnemonic version** of machine code. However, the reloadable machine code generated by assembler is a **binary reloadable code**.

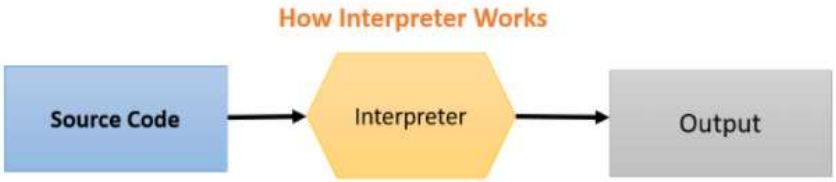
3. Interpreter

Interpreter is a type of language translator that does not necessarily have an output interface and hence its language processing activities are closely bound with the program execution. Use of interpreter is motivated by two reasons – efficiency in certain environments and simplicity. It is simpler to develop an interpreter than to develop a compiler because interpretation does not involve code generation. This makes interpretation more attractive in situations where programs or commands are not executed repeatedly. Hence interpretation is a popular choice for commands to an operating system or an editor. User interfaces of many software packages prefer interpretation.

The interpreter consists of three main components:

- Symbol Table: The symbol table holds the information concerning entities in the source program.
- Data Store: The data store contains values of the data items declared in the program being interpreted.

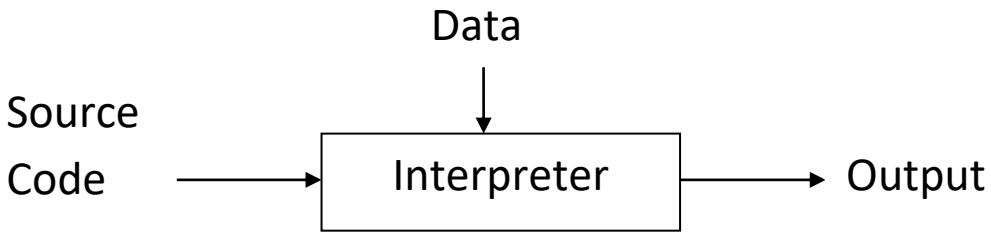
- Data Manipulation Routine: This set contains a routine for every legal data manipulation action in the source language.



Types of Interpreter

- **Pure Interpreter**

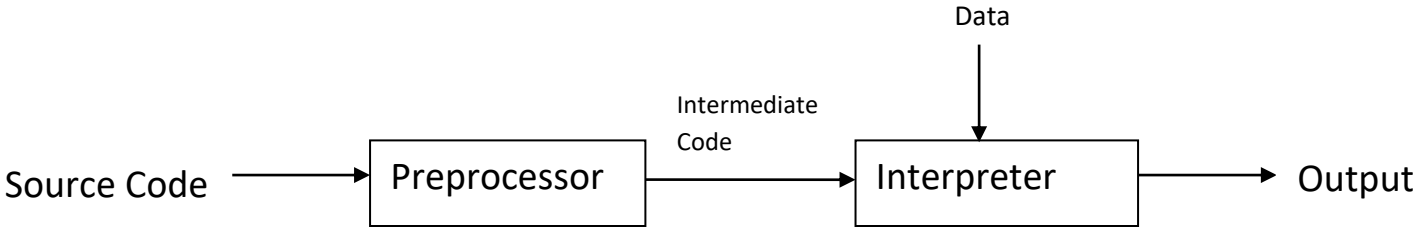
  Pure Interpreters do not change the source code and retains the source form throughout the interpretation process. This requires more overhead and the process becomes a bit complex.



- **Impure Interpreter**

  The source code is subjected to initial preprocessing before the code is eventually interpreted. The actual analysis overhead is now reduced

  and the process speeds up and efficient interpretation.



Key Difference between Compiler and Interpreter

1. The compiler takes a program as a whole and translates it, but interpreter translates a program statement by statement.
2. Intermediate code or target code is generated in case of a compiler. As against interpreter doesn't create intermediate code.
3. A compiler is comparatively faster than Interpreter as the compiler take the whole program at one go whereas interpreters compile each line of code after the other.
4. The compiler requires more memory than interpreter because of the generation of object code.
5. Compiler presents all errors concurrently, and it's difficult to detect the errors in contrast interpreter display errors of each statement one by one, and it's easier to detect errors.
6. In compiler when an error occurs in the program, it stops its translation and after removing error whole program is translated again. On the contrary, when an error takes place in the interpreter, it prevents its translation and after removing the error, translation resumes.
7. In a compiler, the process requires two steps in which firstly source code is translated to target program then executed. While in Interpreter It's a one-step process in which Source code is compiled and executed at the same time.
8. The compiler is used in programming languages like C, C++, C#, Scala, etc. On the other Interpreter is employed in languages like PHP, Ruby, Python, etc.

Linker (*detail in mod.5)*

The Assembler generates the object code of a source program and hands it over to the linker. The linker takes this object code and generates the **executable code** for the program, and hand it over to the Loader.The high-level language, programs have some **built-in libraries** and **header files**. The source program may contain some library functions whose definitions are stored in the built-

in libraries. The linker links these function to the built-in libraries. In case the built-in libraries are not found it informs to the compiler, and the compiler then generates the error. Sometimes the large programs are divided into the subprograms which are called **modules**. Now when these modules are compiled and assembled, the object modules of the source program are generated. The linker has the responsibility of combining/linking all the object modules to generate a single executable file of the source program. Two types of linkers:

- **Linkage Editor**: It is a linker that generates the reloadable, executable module.
- **Dynamic Linker**: It defers/postpones the linkage of some external modules until the load module/executable module is generated. Here, linking is done during load time or run time.

Loader (*detail in mod.5)*

The program that has to be executed currently must reside in the main memory of the computer. It is the responsibility of the **loader,** a program in an operating system, to load the executable file/module of a program, generated by the linker, to the main memory for execution. It allocates the memory space to the executable module in main memory. There are three kinds of loading approaches:

- Absolute loading
- Relocatable loading
- Dynamic run-time loading

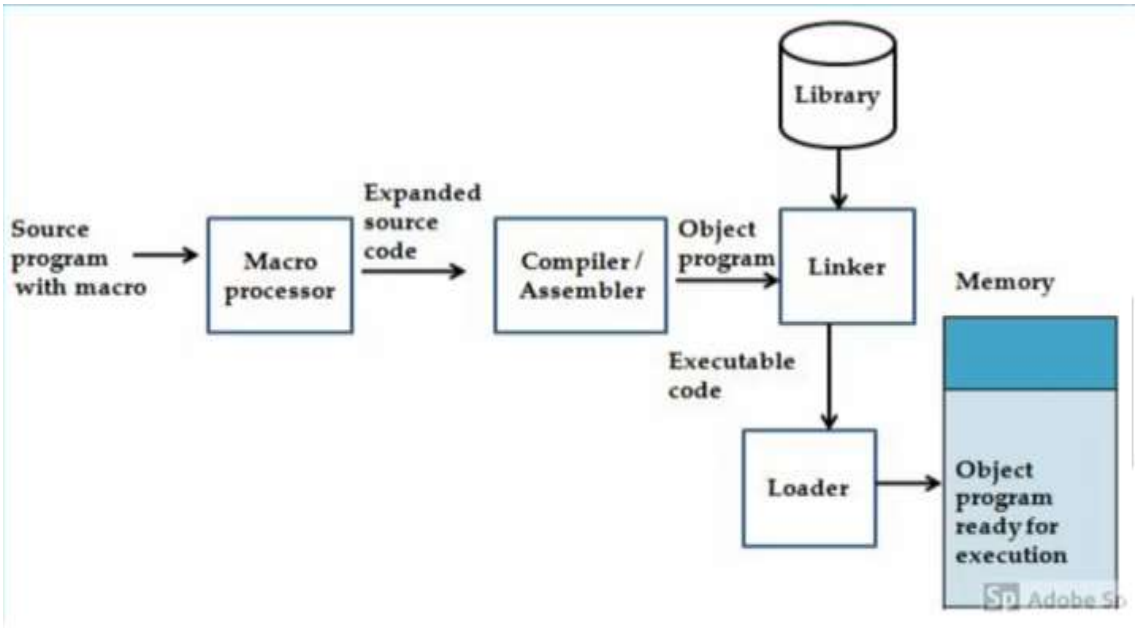
Absolute loading: This approach loads the executable file of a program into a **same main memory location** each time. But it has some **disadvantages** like a programmer must be aware of the assignment strategy for loading the modules to main memory. In case, the program is to be modified involving some insertion and deletion in the program, then all the addresses of the program have to be altered.

Relocatable loading: In this approach, the compiler or assembler does **not produce actual main memory address**. It produces the relative addresses.

Dynamic Run-Time loading: In this approach, the absolute address for a program is generated when an instruction of an executable module is actually executed. It is very flexible, the loadable module/executable module can be loaded into **any region of main memory**. The executing program can be interrupted in between and can be swapped out to the disk and back to main memory this time at a different main memory address.

Key Differences between Linker and Loader

1. The linker generates the **executable** file of a program whereas, the loader loads the executable file obtained from the linker into **main memory for execution**.
2. The linker intakes the **object module** of a program generated by the assembler. However, the loader intakes the **executable module** generated by the linker.
3. The linker combines all object module of a program to generate **executable modules** it also links the **library function** in the object module to **built-in libraries** of the high-level programming language. On the other hands, loader **allocates space to an executable** module in main memory.
4. The linker can be classified as **linkage editor** and **dynamic linker** whereas loader can be classified as **absolute loader, relocatable loader** and **dynamic run-time loader**.



Debugger (in detail mod.6)

A debugger is a software program used to test and find bugs(errors) in other programs. The tool is known as debugging tool.

Two modes of operation of debugging tools are:

- Full Simulation
- Partial Simulation

There are two types of debuggers:

- CorDBG (command-line debugger)
- DbgCLR (graphic debugger)

Machine Architecture and System Software

One characteristic in which most system software differs from application software is **machine dependency.** But most system software is machine-dependent.

The Machine Dependent features are:

- Assembler translate mnemonic instructions into machine code, the instruction formats, addressing modes, etc. are of direct concern in assembler design.
- Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available.
- Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

The Machine Independent features are:

- The general design and logic of an assembler is basically same on most of the computers.
- Some of the code optimization techniques used by compilers (but some are machine dependent)

- The process of linking independently assembled subprograms does not usually depend on the computer being used.

General approach to a new machine

There are a series of questions that we ask if we wish to program the machine.

1. *MEMORY*
   What are the memory's basic unit, size and addressing scheme?
2. *REGISTERS*
   How many registers are there? What are their size, function and interrelationship?
3. *DATA*
   What types of data can be handled by the computer? Can it handle characters, numbers, and logical data? How is this data stored?
4. *INSTRUCTIONS*
   What are the classes of instructions on the machine? Are there arithematic instructions, logical instructions, symbol-manipulation instructions? What are their formats? How are they stored in memory?
5. *SPECIAL FEATURES*
   What is the interrupt structure of the machine? What sort of protection mechanism is available to the user?

Simplified Instructional Computer (SIC)

SIC is a hypothetical model particularly designed to include the hardware features which are most often found on real time machines. The hardware components in such a system are basic to all real machines thereby eliminating irrelevant complexities. Since the design is very simple, the software concepts are clearly separated from implementation details. In many ways SIC is similar to microcomputer. Like many other products, SIC comes in two versions:

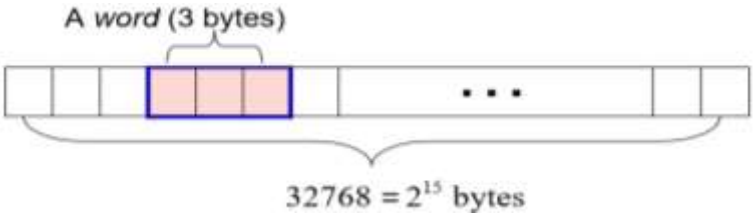
1. SIC standard model
2. SIC/XE (XE stands for eXtra Expensive or eXtra Equipment)

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

SIC Machine Architecture

Memory

Memory consists of 8-bit bytes. Any 3 consecutive bytes form a word (24 bits).All addresses on SIC is byte addresses. Words are addressed by the location of their lowest numbered byte. There are a total of 2^{15} (32768) bytes in the computer memory.



Registers

There are 5 registers and all of them have special uses. Each register is 24- bits in length.

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

Data Formats

Integers are stored as 24-bit binary numbers. Negative numbers are stored as 2's complement representation. Characters are stored using their 8-bit ASCII codes. There is no floating-point hardware on the standard version of SIC.

Instruction Format

All machine instructions on standard version of SIC are 24-bit format. The following is the representation of instruction format.

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

Addressing Modes

There are two addressing modes available, indicated by the setting of the x bit in the instruction. In direct addressing mode the flag bit x=0. Here the target address is specified by the actual address in the instruction itself. In indexed addressing 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. The table following describes how the target address is calculated from the address given in instruction. Parentheses are used to indicate the contents of a register or a memory location. For example, (X) represents the contents of register X.

Mode	Indication	Target address calculation
Direct	x=0	TA = address
Indexed	x=1	TA = address + (X)

Instruction Set

- Instructions for load and store registers
  - LDA, STA, LDX, STX, LDL, STL
- Instructions for integer arithmetic operations
  - ADD, SUB, MUL, DIV
  - All arithmetic operations involve register A and a word in memory with result being left in the register
- Instruction COMP

- Compares the value in register A with a word in memory
- This sets a condition code, CC to indicate the result (<,=,>)

- Conditional Jump Instructions
  - JLT, JEQ, JGT
  - Can test the setting of CC and jump accordingly.
- Instructions for Subroutine linkages
  - JSUB – jumps to the subroutine, placing the return address in register L
  - RSUB – returns by jumping to the address contained in the register L
- Input/ Output Instructions
  - Input and output are performed by transferring 1 byte at a time to or from the rightmost 8-bits of register A
  - Each device is assigned a unique 8-bit code.
  - TD (Test Device): this instruction tests whether the addressed device is ready to send or receive a byte of data. Condition code is set to indicate the result of this test.
    - If < means the device is ready to send or receive data
    - If = means the device is not ready
  - A program needing to transfer data must wait until the device is ready, then execute RD and WD.
  - RD (Read Data)
  - WD(Write Data)
  - This sequence must be repeated for each byte of data to be read or written.

Programming Examples –SIC

We will not write machine language in ones and zeros, nor will we use hexa-decimal numbers. Rather we will use a mnemonic form of machine language called an assembly language. Programs known as assemblers were written to automate the translation of assembly language into machine language.

Assembler Directives

Assembler directives are known as Pseudo instruction. They tell the assembler to establish start address for your program, to reserve space for variables, to include additional source files. START, END, BYTE, WORD, RESW, RESB are some of the assembler directives.

Assembler Directive	**Description**
START	Specify name and starting address for the program. eg: START 100 Statement indicates that the first word of the object program generated by the assembler should be placed with the address 100.
END	Indicate the end of the source program means no more assembly statements remain to be processed. 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 eg: CHARZ BYTE C'Z'
WORD	Generate one-word integer constant eg: FIVE WORD 5
RESB	Reserve the indicated number of bytes for a data area eg: C1 RESB 1
RESW	Reserve the indicated number of words for a data area eg: ALPHA RESW 1

Program 1: Data Movement

```
        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
```

Explanation

- There is no memory – memory move instruction
- So all data movement must be done using registers
- Similar to LDA and STA we have for X register we have (LDX and STX), for L register we have (LDL and STL).
- In the example, Firstly, 3-byte word is moved by loading into register A and then storing the register at the desired location
- Secondly, a single byte of data is moved using the instructions LDCH (Load Character) and STCH (Store Character)
- These instructions operate by loading or storing the rightmost 8-bit byte of register A; the other bits in register A are not affected.
- **WORD** reserves one word of storage, which is initialized to a value defined in the operand field of the statement.
- Here in this example the WORD statement defines a data word labeled FIVE whose value is initialized to 5
- **RESW** reserves one or more words of storage for use by the program
- In this example one word storage labeled ALPHA will be used to hold a value generated by the program.
- The statements **BYTE, RESB** perform similar storage definition functions for the data items that are characters instead of words.
- In this example, CHARZ is a 1-byte data item whose data item whose value is initialized to the character “Z”

- And C1 is a 1-byte variable with no initial value.

Program 2: Arithmetic Operation

```
        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
```

Explanation

- All arithmetic operations are performed using register A. With the result being left in register A.

- LDA ALPHA

// Here we do not know the content of ALPHA. So we can write as ALPHA itself. Now value of A register = ALPHA

- ADD INCR

// Add operation is performed with the contents of A register and the content in INCR and the result is stored in A register. Now, A= (ALPHA +INCR)

- SUB ONE

// Next the content in the A register is subtracted by 1 here we have given the one word constant with a value 1 and result is stored in A register. Now, A= (ALPHA +INCR – 1)

- STA BETA

// Now we are moving the result from A register to BETA because if we use the A register for the next instruction the content get over- written. So after performing this instruction BETA = (ALPHA+INCR-1)

- LDA GAMMA

//Again we are loading the content in the GAMMA to A register. Now, A= GAMMA

- ADD INCR

//Add operation is performed with the contents of A register and the content in INCR and the result is stored in A register. Now, A= (GAMMA+INCR)

- SUB ONE

// Next the content in the A register is subtracted by 1 here we have given the one word constant with a value 1 and result is stored in A register. Now, A= (GAMMA +INCR – 1)

- STA DELTA

// Now we are moving the result from A register to DELTA because if we use the A register for the next instruction the content get over- written. So after performing this instruction DELTA = (GAMMA+INCR-1)

- So finally we can say that, the sequence of instructions stores the value as :
  - BETA = (ALPHA + INCR - 1)
  - DELTA = (GAMMA + INCR -1)

Program 3: Input-Output Operation

```
INLOOP   TD      INDEV    TEST INPUT DEVICE   cc : = denotes device busy
         JEQ     INLOOP   LOOP UNTIL DEVICE IS READY
         RD      INDEV    READ ONE BYTE INTO REGISTER A
         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'    OUTP  UT DEVICE NUMBER
DATA     RESB    1        ONE-BYTE VARIABLE
```

Explanation

- 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
- The operand for the RD is a byte in memory that contains the hexadecimal code for the input device (here it is F1)
- Executing RD instruction transfers 1 byte of data from this device into the rightmost byte of A register

- If the input device is character oriented (if keyboard), the value placed in register A is ASCII code for the character that was read
- Before RD can be executed, input device must be ready to transmit the data.
- For example, if the input device is a keyboard, the operator must have typed a character.
- The program checks for this by using the TD instruction.
- When TD is executed, the status of the addressed device is tested and condition code is set to indicate the result of this test.
- If the device is ready to transmit data, the condition code is set to "less than".
- If the device is not ready, the condition code is set to "equal"
- The program must execute the TD instruction and then check the condition code by using a conditional jump
- If condition code is "equal"(device not ready), the program jumps back to the TD instruction.
- The two instruction loop will continue until the device becomes ready, then RD will be executed.
- Output is performed in the same way.
- The program uses TD to check whether the output device (here, 05) is ready to receive a byte of data.
- Then the byte to be written is loaded into the rightmost byte of register A, and WD instruction is used to transmit it to the device.

Program 4: Looping and Indexing

```
        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 STRING'  11-BYTE STRING CONSTANT
STR2    RESB    11              11-BYTE VARIABLE
.                               ONE-WORD CONSTANTS
ZERO    WORD    0
ELEVEN  WORD    11
```

Explanation

- A loop that copies one 11-byte character string to another
- The index register (X) is initialized to zero before the loop begins
- During the first execution of loop, the target address for the LDCH instruction will be the address of the first byte of STR1
- Similarly, the STCH instruction will copy the character being copied into the first byte of STR2.
- Next instruction TIX performs two functions:
- It adds 1 to the value in X register
- It compares new value in the X register with the operand (here 11)
- Condition code is set to indicate the result of the 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 the 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 STR1 and the target address for the STCH instruction will be the second byte of STR2.
- The TIX instruction will again add 1 to the value in register X, and the loop will continue in this way until all 11 bytes have been copied from STR1 to STR2.
- Notice that after the TIX instruction is executed, the value in register X is equal to the number of bytes that have already been copied.

Program 5: Subroutine call and record Input Operations

```
        JSUB    READ        CALL READ SUBROUTINE
        .
.                           SUBROUTINE TO READ 100-BYTE RCORD
READ    LDX     ZERO        INITAILIZE 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,X    STORE DATA BYTE INTO RECORD
        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
```

Explanation

- These 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 by using JSUB (Jump to Subroutine) instruction.
- At the end of the subroutine there is an RSUB (Return from Subroutine) instruction, which returns control to the instruction that follows the JSUB.
- The READ subroutine itself consists of a loop.
- Each execution of this loop reads 1 byte of data from the input device, using the same concept used in input –output operations(ref. program: 3)
- The bytes of data that are read are stored in a 100-byte buffer area labelled RECORD.
- The indexing and looping techniques that are used in storing characters in this buffer are essentially the same as those illustrated (in program: 4)

SIC/XE Machine Architecture

Memory

Memory consists of 8-bit bytes. Any 3 consecutive bytes form a word (24 bits).All addresses on SIC/XE is byte addresses. Words are addressed by the location of their lowest numbered byte. The maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} bytes). This increase leads to a change in instruction formats and addressing modes.

Registers

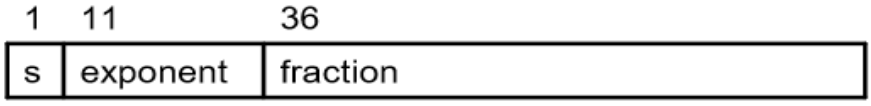
There are 9 registers and all of them have special uses. Each register is 24- bits in length, except floating point accumulator (F) which is 48- bit length.

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
B	3	Base register; used for addressing
S	4	General working register-no special use
T	5	General working register-no special use
F	6	Floating-point accumulator (48bits)
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

Data Formats

Integers are stored as 24-bit binary numbers. Negative numbers are stored as 2's complement representation. Characters are stored using their 8-bit ASCII codes. There is no floating-point hardware on the standard version of SIC.

Additionally, there is a 48-bit floating-point data type



- s – indicates sign of floating – point number
  - If s= 0, then it is a positive number
  - If s= 1, then it is a negative number
- A value of zero is represented by setting all bits (including sign, exponent and fraction) as 0

- Fraction(f) is the value between 0 and 1
- Exponent (e) part is an unsigned binary number between 0 and 2047 $\rightarrow(2^{11}=2048)$
- Absolute value $f* 2^{(e-1024)}$

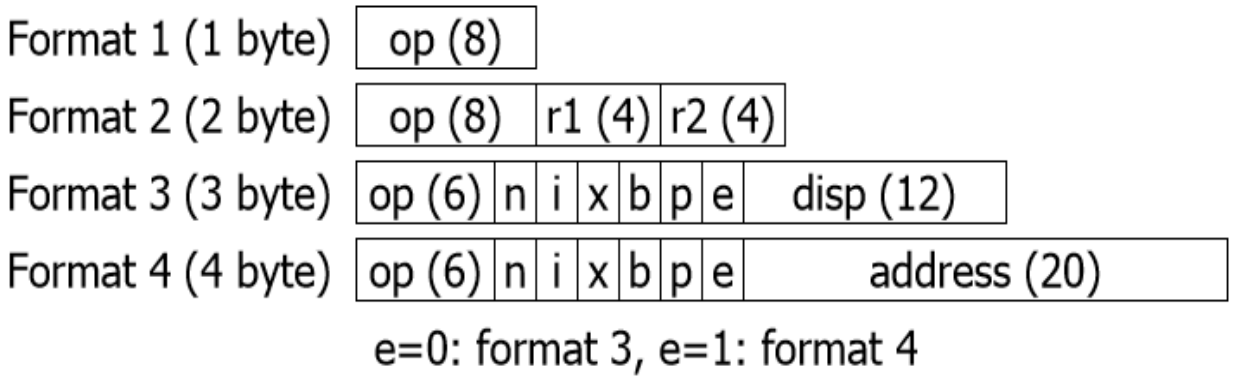
Instruction Format

In SIC, there is only 1 format (24-bit). But in **SIC/XE**, as the memory size is large, this instruction format is not enough to address the whole memory There are two possible options:

1. Use some form of relative addressing
2. Extend the address field to 20 bits

There are **4 formats** of instructions available in SIC/XE

SIC/XE provides some instructions that do not reference memory at all (as in format 1 &2)



- In format 1, 8bits for opcode only eg: RSUB
- In format 2, 8 bits for opcode and 2 registers each having 4bits eg: COMPR A,S
- In format 3, 6 bits for opcode, 6bits of flags specifies the addressing modes, and 12 bit displacement. Eg: LDA #3
- In format 4, 6 bits for opcode, 6 bits of flags specifies the addressing modes and 20 bits of address. Eg: JSUB READ

Addressing Modes

(Specifies where the operand of the particular instruction available)

In SIC, only 2 addressing modes: direct addressing with or without indexing

(ie., 1. Direct addressing 2. Indexed addressing)

Consider the flag bits discussed in instruction format

n	**i**	**x**	**b**	**p**	**e**

- n → indirect
- i → immediate
- x → indexed
- b → base relative
- p → program counter relative
- e → indicate the format (either 3 or 4)
- **e bit** :
    - If e=0, then instruction is of format 3.
    - If e=1, then instruction is of format 4.

Already, we have a chart indicates the format to be used with each machine instruction(I have given the pdf of the appendix of the text in your lms)

- **b and p bit**: before that first understood what is relative addressing
    - **Relative Addressing**
        - Means we are specifying some address of the operand relating to some registers (normally using base register or program counter)
        - If specifying the address in relative to base register (B) , then it is called **base relative addressing**

- If specifying the address in relative to program counter(PC), then it is called **program counter relative addressing**

Mode	Indication	Target address calculation
Base relative	b=1,p=0	TA=(B)+disp (0 ≤ disp ≤ 4095)
Program-counter relative	b=0,p=1	TA=(PC)+disp (-2048 ≤ disp ≤ 2047)

If b = 0, p = 0 TA = disp

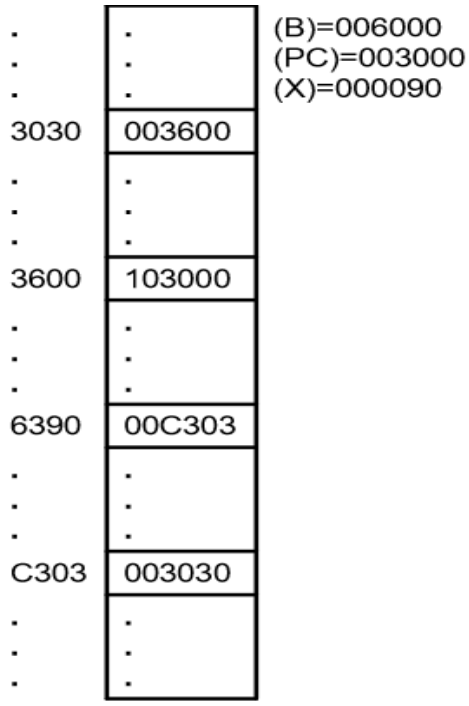
- Displacement field (***disp):*** It is an 8-bit or 16-bit immediate value given in the instruction
- In Format 3 instruction:
  - For **base relative addressing**, the displacement field (disp) is interpreted as 12-bit unsigned integer.
  - For **program counter relative addressing**, the displacement field(disp) is interpreted as 12-bit signed interger, with negative values in 2's complement representation
- Both the relative addressing can also be combined indexed addressing by using the x bit flag field
- **x bit** : indexed addressing
  - If x =1, TA can be calculated by adding the content of the X register also with the already calculated the TA for the relative addressing
- In Format 4 instruction the bits b and p are normally set to 0 → then the target address is taken from the address field of the instruction. This is known as **Direct Addressing**
- **i and n bits**: specify how target address is used.

Mode	Indication	Operand value
Immediate addressing	i=1, n=0	**TA**: TA is used as the operand value, no memory reference
Indirect addressing	i=0, n=1	**((TA))**: The word at the TA is fetched. Value of TA is taken as the address of the operand value
Simple addressing	i=0, n=0	Standard SIC
	i=1, n=1	**(TA)**:TA is taken as the address of the operand value

- Indexing **cannot** be used with immediate or indirect addressing modes

- Special Characters used:
  - # means immediate addressing
  - @ means indirect addressing

Examples of different addressing modes in SIC/XE (LDA instruction)



Hex	Machine instruction Binary							Target address	Value loaded into register A
	op	n	i	x	b	p	e	disp/address		
032600	000000	1	1	0	0	1	0	0110 0000 0000	3600	103000
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
010030	000000	0	1	0	0	0	0	0000 0011 0000	30	000030
003600	000000	0	0	0	0	1	**0**	0110 0000 0000	3600	103000
0310C303	000000	1	1	0	0	0	1	0000 1100 0011 0000 0011	C303	003030

Instruction Set

- Instructions for load and store registers
  - LDA, STA, LDX, STX, LDL, STL
  - LDB, STB

- Instructions for integer arithmetic operations
  - ADD, SUB, MUL, DIV
  - All arithmetic operations involve register A and a word in memory with result being left in the register
- Instructions for Floating point arithmetic operations
  - ADDF, SUBF, MULF, DIVF
- Instructions that take operands from registers
  - RMO (Register Move)
- Instruction for register – to - register arithmetic operations
  - ADDR, SUBR, MULR, DIVR
- Special instruction
  - SVC (Supervisor call) Executing this instruction generates an interrupt that can be used for communication with the operating system
  - LPS (Load Processor Status)→ This instruction stores the CPU status and register contents existing at any time of the interrupt. It transfers control to the instruction that follows the one that was being executed when the interrupt occurred.
- Instruction COMP, COMPR
  - Compares the value in register A with a word in memory and also between registers
  - This sets a condition code, CC to indicate the result (<,=,>)
- Conditional Jump Instructions
  - JLT, JEQ, JGT
  - Test the setting of CC and jump accordingly
- Instructions for Subroutine linkages
  - JSUB – jumps to the subroutine, placing the return address in register L
  - RSUB – returns by jumping to the address contained in the register L
- Input/ Output Instructions
  - Input and output are performed by transferring 1 byte at a time to or from the rightmost 8-bits of register A
  - Each device is assigned a unique 8-bit code.

- TD (Test Device): this instruction tests whether the addressed device is ready to send or receive a byte of data. Condition code is set to indicate the result of this test.
  - If < means the device is ready to send or receive data
  - If = means the device is not ready
- A program needing to transfer data must wait until the device is ready.
- RD (Read Data)
- WD(Write Data)
- This sequence must be repeated for each byte of data to be read or written

o SIC/XE supports I/O channels that can be used to perform input and output operation (data transfer) while the CPU is executing other instructions

o Channel program controls sequence of operation performed by a channel

o This allows overlapping of computing and I/O – results in more efficient system operation

- SIO Start an I/O channel number
- TIO Test an I/O channel number
- HIO Halt an I/O channel number

Programming Examples - SIC/XE

Program 1: Data Movement

```
        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
```

Explanation

- Shows two data movement operations
- Here value 5 is loaded into register A using immediate addressing
- The operand field for this instruction contains the flag #(which specifies the immediate addressing) and the data value to be loaded.

- 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: Arithmetic Operation

```
            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
```

Explanation

- Arithmetic operations performed on SIC/XE architecture
- The value of INCR is loaded into S register initially.
- The register-to-register instruction ADDR is used to add this value to register A when it is needed.
- This avoids having to fetch INCR from memory each time it is used in calculation, which may make the program more efficient.
- Immediate Addressing is used for the constant 1 in the subtracting operations
- So finally we can say that, the sequence of instructions stores the value as :
    - BETA = (ALPHA + INCR - 1)
    - DELTA = (GAMMA + INCR -1)

Program 3: Looping and Indexing

```
        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          SOTRE CHARACTER INTO STR2
        TIXR    T               ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT     MOVECH          LOOP IF INDEX IS LESS THAN 11
        .
        .
STR1    BYTE    C'TEST STRING'  11-BYTE STRING CONSTANT
STR2    RESB    11              11-BYTE VARIABLE
```

Explanation

- Here the example shows the same loop as in SIC architecture: a loop that copies one 11 byte character string to another.
- The main difference is that the instruction TIXR is used in place of TIX.
- TIXR works exactly like TIX, except that the value used for comparison is taken from another register (here taken in T register), not from memory.
- This makes the loop more efficient, because the value does not have to be fetched from memory each time the loop is executed.
- Immediate Addressing is used to initialize register T to the value 11 and to initialize register x to 0

Program 4: Indexing and Looping

```
        LDS     #3              INITIALIZE REGISTER S TO 3
        LDT     #300            INITIALIZE REGISTER T TO 300
        LDX     #0              INITIALIZE INDEX REGISTER TO 0
ADDLP   LDA     ALPHA,X         LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X          ADD WORD FROM BETA
        STA     GAMMA,X         STORE THE RESULT IN A 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
```

Explanation

- Another example of looping and indexing operation
- The variables ALPHA, BETA and GAMMA are arrays of 100 words each.
- In this case the task of the loop is to add together the corresponding elements of ALPHA and BETA, storing the results in the elements of GAMMA.
- The general principle of looping and indexing is same as the previous example.
- However the value in the index register must be incremented by 3 for each iteration of this loop, because each iteration processes a 3-byte (ie., one word) element of the arrays.
- The TIX instruction always add 1 to register x, so it is not suitable for this program fragment.
- Instead, we use arithmetic and comparison instructions to handle the index value. In this example, the index value is kept permanently in register X. The amount by which to increment the index value (3) is kept in register S, and the register –to - register ADDR instruction is used to add this increment to register X.
- Similarly, the value 300 is kept in register T, and the instruction COMPR is used to compare registers X and T in order to decide when to terminate the loop.

Program 5: Subroutine call and record input operation

```
        JSUB    READ        CALL READ SUBROUTINE
        .
        .
.                           SUBROUTINE TO READ 100-BYTE RECORD
READ    LDX     #0          INITIALIZE INDEX REGISTER TO 0
RLOOP   LDT     #100        INITIALIZE REGISTER T TO 100
        TD      INDEV       TEST INPUT DEVICE
        JEQ     RLOOP       LOOP IF DEVICE IS BUSY
        RD      INDEV       READ ONE BYTE INTO REGISTER A
RLOOP   STCH    RECORD,X    STORE DATA BYTE INTO RECORD
        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
```

Explanation

- The program shows the same READ subroutine in SIC architecture: to read a 100-byte record from an input device into memory. Main difference is the use of immediate addressing and TIXR instruction.