Module 1, INTERPRETERS it processes a source program written in a high-level language. The main difference between compiler and interpreter is that interpreters execute a version of the source program directly, instead of translating it into machine code. An interpreter performs lexical and syntactic analysis functions just like compiler and then translates the source program into an internal form. The internal form may also be a sequence of quadruples

Compiler: It is a program which translates a high level language program into a machine language program. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program run time is more and occupies a larger part of the memory. It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes. If a compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler. On the other hand, if a compiler runs on a computer and produces the machine codes for other computer then it is known as a cross compiler

Linker: In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program

Loader: is a program that loads machine codes of a program into the system memory. In Computing, a loader is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All OS that support program loading have loaders. In many operating systems the loader is permanently resident in memory

Text editor is a tool that allows a user to create and revise documents in a computer. Though this task can be carried out in other modes, the word text editor commonly refers to the tool that does this interactively. Earlier computer documents used to be primarily plain text documents, but nowadays due to improved input-output mechanisms and file formats, a document frequently contains pictures along with texts whose appearance (script, size, colour and style) can be varied within the document. Apart from producing output of such wide variety, text editors today provide many advanced features of interactiveness and output

Types of Text Editors Depending on how editing is performed, and the type of output that can be generated, editors can be broadly classified as -

1. Line Editors - During original creation lines of text are recognised and delimited by end-of-line markers, and during subsequent revision, the line must be explicitly specified by line number or by some pattern context. eg. edlin editor in early MS-DOS systems.

2. Stream Editors - The idea here is similar to line editor, but the entire text is treated as a single stream of characters. Hence the location for revision cannot be specified using line numbers. Locations for revision are either specified by explicit positioning or by using pattern context. eg. sed in Unix/Linux. Line editors and stream editors are suitable for text-only documents.

3. Screen Editors - These allow the document to be viewed and operated upon as a two dimensional plane, of which a portion may be displayed at a time. Any portion may be specified for display and location for revision can be specified anywhere within the displayed portion. eg. vi, emacs, etc.

4. Word Processors - Provides additional features to basic screen editors. Usually support non-textual contents and choice of fonts, style, etc.

5. Structure Editors - These are editors for specific types of documents, so that the editor recognises the structure/syntax of the document being prepared and helps in maintaining that structure/syntax.

Module 1, Debugger 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. However, the word debugging is usually in context of errors that manifest while running the program during testing or during actual use. The most common steps taken in debugging are to examine the flow of control during execution of the program, examine values of variables at different points in the program, examine the values of parameters passed to functions and values returned by the functions, examine the function call sequence, etc. In the absence of other mechanisms, one usually inserts statements in the program at various carefully chosen points, that prints values of significant variables or parameters, or some message that indicates the flow of control (or function call sequence). When such a modified version of the program is run, the information output by the extra statements gives clue to the errors.

Interpreters 1.The process of translating a source program into some internal form is simpler and faster. 2.Execution of the translated program is much slower.
3.Debugging facilities can be easily provided.
4.During execution the interpreter produce symbolic dumps of data values, trace of program execution related to the source statement.
5.Program testing can be done effectively using interpreter as the operation on different data can be traced. 6.Easy to handle dynamic scoping

Compilers

1.The process of translating a source program into some internal form is slower than interpreter.
2.Executing machine code is much faster.
3.Provision of bugging facilities are difficult and complicated.
4.The compiler does not produce symbolic dumps of date value. Debugging tools are required for trace the program.
5.It is difficult to test as the compiler execution file gives the final result.
6.Difficult to handle dynamic scooping

Operating System An OS is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. Important functions OS
*Memory Management *Processor Management
*Device Management *File Management
*Security *Control over system performance
*Job accounting *Error detecting aids
*Coordination between other software and users

Instruction Formats – SIC

*24 bit : 8bit opcode- 1bit Ad mode- 15bit adres
*Load and Store Registers
– LDA, LDX, STA, STX, etc.
*Integer Arithmetic (all involve register A)
– Add, SUB, MUL, DIV
*Compare COMP – compares A with a word in memory, Sets the CC in the SW
*Jump instructions
– JLT, JEQ, JGT – based on the CC as set by COMP
*Subroutine Linkage
– JSUB – jumps to subroutine, places return address in L
– RSUB – returns, using the address in L

Input/Output - SIC

*TD – test device is ready to send/receive data
– CC of < means device is ready
– CC of = means device is not ready
*RD – read data, when the device is ready
*WD – write data
*Transfers 1 byte at a time to or from the rightmost 8 bits of register A.
*Each device has a unique 8-bit code as an operand.

System software support operation and use of computer. Application software - solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly, 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.

Module 1, Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

***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 we 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

***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)

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

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

***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.

Instruction formats and Addressing Modes

The instruction formats depend on the memory organization and the size of the memory. In SIC machine the memory is byte addressable. Word size is 3 bytes. So the size of the memory is 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. Whereas the memory of a SIC/XE machine is 2^20 bytes (1 MB). This supports four different types of instruction types, they are:

*1 byte instruction
*2 byte instruction
*3 byte instruction
*4 byte instruction

Instructions can be:

– Instructions involving register to register
– Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
– Extended instruction format

• Addressing Modes are:

– Index Addressing(SIC): Opcode m, x
– Indirect Addressing: Opcode @m
– PC-relative: Opcode m
– Base relative: Opcode m
– Immediate addressing: Opcode #c

Device Driver An I/O device driver issues commands to the device controller by storing values into the appropriate registers, exactly as if they were physical memory locations. Likewise, software routines may read these registers to obtain status information. The association of an address in I/O space with a physical register in a device controller is handled by the memory management routines.

Assembler Directives In addition to translating the instructions of the source program, the assembler must process statements called assembler directives (or pseudo-instructions). These statements are not translated into machine instructions (although thev may have an effect on the object program). Instead, they provide instructions to the assembler itself. Examples of assembler directives are statements like BYTE and WORD, which direct the assembler to generate constants as part of the object program, and RESB and RESW, which instruct the assembler to reserve memory locations without generating data values. The other assembler directives in our sample program are START, which specifies the starting memory address for the object program, and END, which marks the end of the program.

Header record:

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)

Text record:

Col. 1 T
Col. 2-7 Starting address for object code in this record(hexadecimal)
Col. 8-9 Length of object code in this record in bytes (hexadecimal)
Col. 10-69 Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

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