Total Pages:3

Reg. No._________ Name:________________________

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

FIFTH SEMESTER B.TECH MODEL EXAMINATION, NOVEMBER 2017

Course Code: CS303

Course Name: SYSTEM SOFTWARE

Max. Marks: 100 **Duration: 3 Hours**

PART A

Answer all questions, each carries 3 marks.

1. What are assembler directives? Explain any two assembler directives.
2. What are the various addressing modes available in SIC?
3. Generate the target address for the following object codes.
   i) 032600 ii) 03C300
   Given the content of X=000090
   the content of PC=003000
   the content of B=006000
4. What are the five fundamental functions of an SIC Assembler?

PART B

Answer any two full questions, each carries 9 marks

5. Write a program in both SIC and SIC/XE to copy a character string `System Software' to another string. (9)
6. a) Write the algorithm for Pass-1 of a 2 pass assembler. (5)

   b) What is upward compatibility? How is it ensured between SIC and SIC/XE? (4)

7. a) Generate the machine codes for the following SIC/XE program.

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

The opcodes of the mnemonics are LDS=6C, LDT=74, LDX=04, LDA=00, STA=0C, ADDR=90, COMPR=A0, JLT=38. **(6)**

b) Explain the assembler data structure SYMTAB. **(3)**

PART C

Answer all questions, each carries 3 marks.

8. Write the formats for refer record and define record.
9. What are literals? Explain the use of LTORG assembler directive while using literals.
10. Write the algorithm for an absolute loader.
11. Explain automatic library search.

PART D

Answer any two full questions, each carries 9 marks.

12. Explain how a multipass assembler handles the following forward references:

```
1    HALFSZ    EQU   MAXLEN/2
2    MAXLEN    EQU   BUFFEND-BUFFER
3    PREVBT    EQU   BUFFER-1
4    BUFFER    RESB  4096
5    BUFFEND   EQU   *
```

Assume that, when the assembler goes to line 4, location counter contains 1034(Hex). **(9)**

13. With source code, explain the working of boot strap loader. **(9)**

14. a) What is dynamic linking? Mention one advantage of using it. **(5)**

b) Assume the following symbol table definitions:

Symbol	**Type**
BUFFER	Relative
LENGTH	Relative
FIRST	Relative
MAXLEN	Absolute
BUFEND	Relative

Classify the following into absolute, relative or neither absolute nor relative expressions.

(i)	BUFFER-FIRST	(v) MAXLEN-1
(ii)	2* LENGTH	(vi) BUFFER+4095
(iii)	MAXLEN-BUFFER	(vii) BUFFER-MAXLEN
(iv)	FIRST-BUFFER+BUFEND	(viii) FIRST+BUFFER

(4)

PART E

Answer any four full questions, each carries 10 marks.

15. a) With an example, explain generation of unique labels in macros. **(5)**

b) Explain the different data structures used in the implementation of a macro processor. **(5)**

16. a) Explain keyword macro parameters. **(5)**
    b) Explain concatenation of macro parameters **(5)**

17. Write the algorithm for a one pass macro processor. **(10)**

18. With a neat diagram explain the working of a typical editor structure. **(10)**

19. Explain the different debugging functions and capabilities. **(10)**

20. Explain debugging i) by Induction ii) by deduction **(10)**

FIFTH SEMESTER MODEL EXAMINATION, NOV 2014

CS 303 SYSTEM SOFTWARE

ANSWER KEY

Part A

1. Assembler directives are pseudo instructions. They dont have any machine codes. They are used to give directions to the assembler.

   eg:- START - specify name & starting address for the program

   RESW - Reserve the indicated no of words for a data area.

   COPY START 1000

   LENGTH RESW 3

2. There are 2 addressing modes available in SIC, indicated by the setting of the x bit in the instruction

8	1	15
opcode	x	address

   1. Direct Addressing mode. $x=0$ $TA = address$
   2. Indexed " $x=1$ $TA = address + (X)$

   - In indexed addressing mode TA is the sum of address and the content of the index register.

     eg:- LDA ALPHA, X

   - Direct addressing eg:- LDA ZERO.

3. i)

6	n	i	x	b	p	e	12
0000 00	1	1	0	0	1	0	600

$TA = disp + (PC)$

$= 600 + 003000 = 003600\,H$

ii)

6	n	i	x	b	p	e	12
0000 00	1	1	1	1	0	0	300

$TA = (X) + (B) + disp$

$= 000090 + 006000 + 300$

$= 006390\,H$

4 1) Convert mnemonic opcodes to their machine language equivalent

eg:- STL to 14

2) Convert symbolic operands to their equivalent m/c address

3) Build the m/c instructions in the proper format.

4) Convert the data constants specified in the source program into their internal m/c representations..

eg:- translate EOF to 454F46

5) Write the object program & assembly listing.

Part B

5. SIC Program

```
        LDX    ZERO
LOOP    LDCH   STR1,X
        STCH   STR2,X
        TIX    LEN
        JLT    LOOP
          :
STR1    BYTE   C'SYSTEM PROGRAMMING'
STR2    RESB   18
ZERO    WORD   0
LEN     WORD   18
```

SIC\XE

```
        LDT    #18
        LDX    #0
LOOP    LDCH   STR1,X
        STCH   STR2,X
        TIXR   T
        JLT    LOOP
          :
STR1    BYTE   C'SYSTEM PROGRAMMING'
STR2    RESB   18
```

6 a) Algorithm for pass 1 of 2 pass assembler.

6 b) Algorithm for Pass 2 of a 2 pass assembler.

7a)

Line No	Address	Label	Mnemonic	Operand	m/c code
		PROG	START	0	
2	0000		LDS	#3	6D0003
3	0003		LDT	#300	75012C
4	0006		LDX	#0	050000
5	0009	LOOP	LDA	#0	010000
6	000C		STA	ALPHA, X	0FA007
7	000F		ADDR	S, X	9041
8	0011		COMPR	X, T	A015
9	0013		JLT	LOOP	3B2FF3
10	0016	ALPHA	RESW	100	

→ Line no: 2

6 bit	n	i	x	b	p	e	12 bit
0110 11	0	1	0	0	0	0	003

→ 6D0003

→ Line no: 3

	n	i	x	b	p	e	
0111 01	0	1	0	0	0	0	12C

→ 75012C

$(300)_{10} = (12C)_{16}$

→ Line no: 4

	n	i	x	b	p	e	
0000 01	0	1	0	0	0	0	000

→ 050000

→ Line no: 5

	n	i	x	b	p	e	
0000 00	0	1	0	0	0	0	000

→ 010000

→ Line no: 6

	n	i	x	b	p	e	
0000 11	1	1	1	0	1	0	007

→ 0FA007

disp = TA − (PC) = 0016 − 000F = 007

→ Line no: 7 S=4 X=1 ADDR = 90 → 9041

→ Line no: 8 X = 1 T=5 COMPR = A0 → A015

→ Line no: 9

	n	i	x	b	p	e	
[illegible] 10	1	1	0	0	1	0	FF3

→ 3B2FF3

disp = 0009 − 0016 = FF3

7b) SYMTAB (Symbol Table) is used to ~~assign~~ store addresses ~~to~~ assigned to labels. It includes the name & address (value) for each label in the source program together with flags to indicate error conditions. It may also include other information like type of symbol or length of the ~~symbol~~ data area represented by the symbol. During Pass 1, labels are entered into SYMTAB along with their addresses. During Pass 2, symbol addresses are looked up from the symbol table to generate the actual machine code.

SYMTAB is organized as a hash table. It is dynamic in nature.

Part C

8. Refer Record:

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

Define Record:

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

Col 14-73 : Repeat information in Col 2-13 for other external symbols.

9. When the value of a constant operand is given as part of the instruction using = symbol, it is called a literal.

eg:- LDA =C'EOF'

Literals are normally placed in a literal pool at the end of the program, which shows assigned addresses and generated data values.

eg: * =C'EOF' 454F46

But when literals are placed at the end of the program, labels & its definitions will be far apart from each other in most cases and it will force us in using format 4 instructions as the displacement will be large. So in some cases, it is desirable to place literals into a ~~pool~~ pool at some other location in the object program using the assembler directive LTORG. Literals placed in the LTORG pool will not be repeated in the pool at the end of the program.

eg:- LTORG

* =X'05' 05

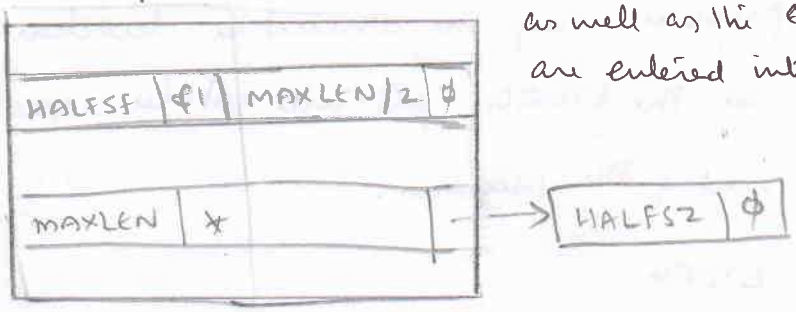
10. Algorithm for absolute loader

11. Linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. This allows programmer to use subroutines from one or more libraries as if they were part of the programming language. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The programmer just have to mention the subroutine as external reference in the source program. This feature is called automatic library search. This is implemented by entering the symbols from each Refer record into ESTAB, if not already present. When the symbol definition is encountered, it is also entered into ESTAB. At the end of Pass1, symbols in ESTAB that are still undefined will be searched in libraries & processes the subroutines found by this search as if they had been part of the primary input stream.

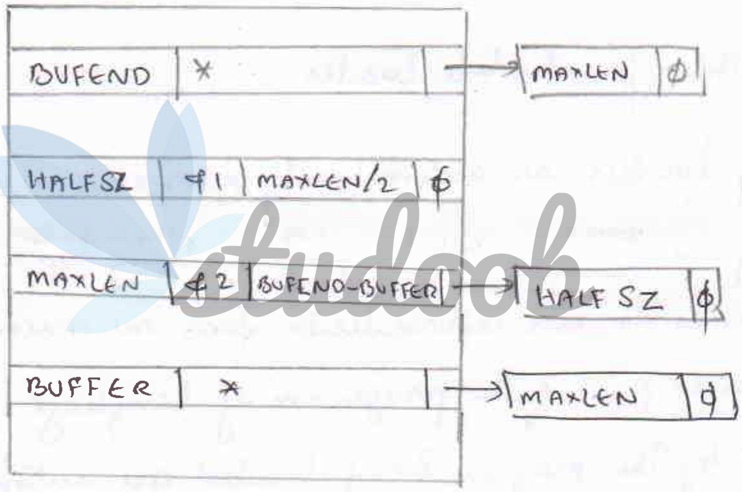
Part D

12. ① HALFSZ EQU MAXLEN/2

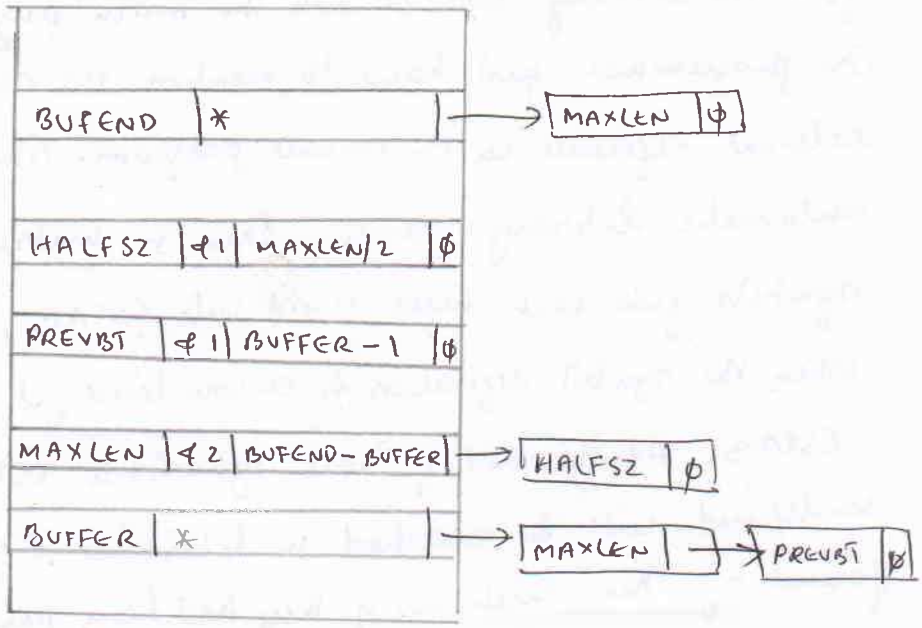
- MAXLEN not defined so value of HALFSZ cannot be computed
- So symbol table entry corresponding to HALFSZ & MAXLEN is made as follows. No of undefined symbols (&1) in HALFSZ defining as well as the expression for HALFSZ are entered into the SYMTAB.



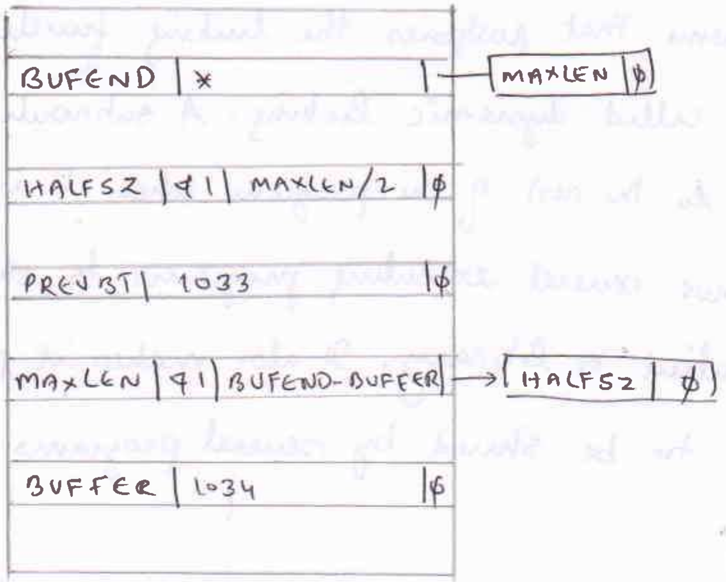
② MAXLEN EQU BUFFEND - BUFFER



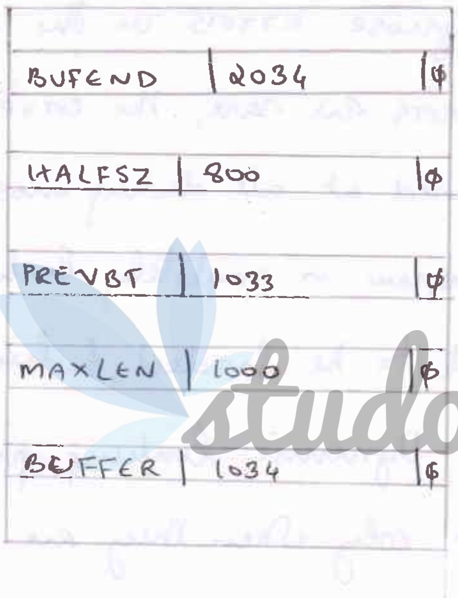
③



4



5



13. Write the source code for Bootstrap ~~algorithm~~ loader.

Bootstrap loader is the first program that executes when the system is switched on. It loads the OS to memory. It's an absolute loader. It resides at location 0 & loads the OS at loc 80. This source code for bootstrap loader contains a subroutine getc which reads the OS one character at a time. Each character read from the disk storage accomodates one byte. But in a machine code, each character is half byte in size. So to represent machine code 14, instead of 2 bytes, it should use one byte. for that SHIFTL operation & followed by ADDR operation is used.

14 a) A scheme that postpones the linking function until execution time is called dynamic linking. A subroutine is loaded & linked to the rest of the program when it is first called. It allows several executing programs to share one copy of a subroutine or library. It also makes it possible for one object to be shared by several programs.

Advantage :

Suppose, for eg, that a program contains subroutines that correct or clearly diagnose errors in the input data during execution. If such errors are rare, the correction & diagnostic routines may not be used at all during most executions of the program. If this program is completely linked before execution, these subroutines need to be loaded & linked every time the program is run. Dynamic linking provides the ability to load the routines only when they are needed.

14 b) i) Absolute expression
ii) Neither absolute nor relative
iii) Neither absolute nor relative
iv) Relative
v) Absolute
vi) Relative
vii) Relative
viii) Neither absolute nor relative.

Part E

15 a) Labels used within the macro body begin with the special character \$. Each symbol beginning with \$ will be replaced with \$xx, where xx is a character alphanumeric counter of the number of macro instructions expanded. For first macro expansion xx will have value AA, followed by AB, AC etc. Such a 2 character counter provides as many as 1296 macro expansions in a single program. This results in generation of unique labels for each expression of a macro instruction.

→ example

~~15 b) NAMTAB~~

~~DEFTAB~~

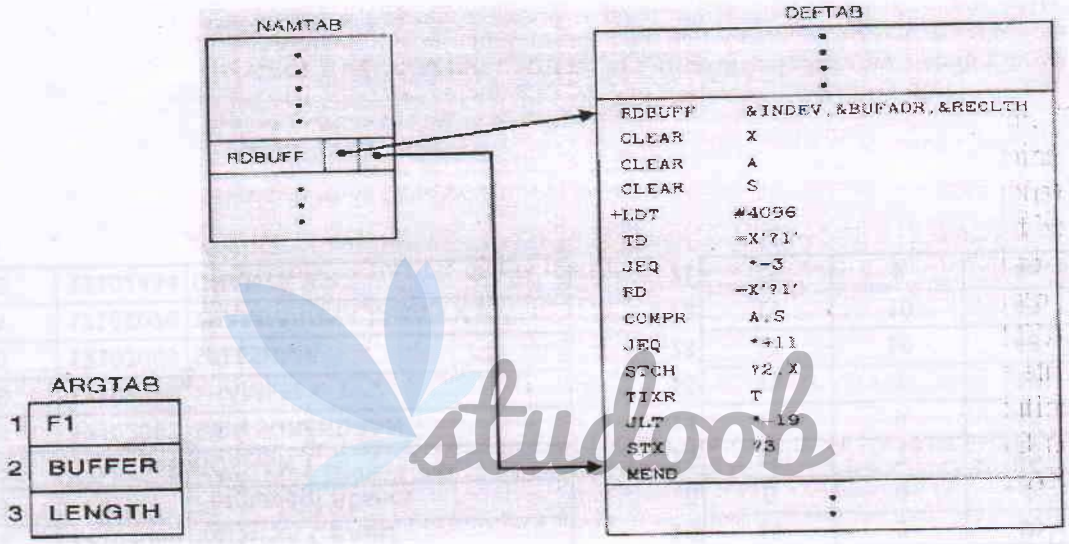
~~ARGTAB~~

~~– explain each with eg.~~

~~– draw example of tables.~~

16 a)

15 b) There are three main data structures involved in our macro processor. The macro definitions themselves are stored in a definition table (DEFTAB), which contains the macro prototype and the statements that make up the macro body (with a few modifications). Comment lines from the macro definition are not entered into DEFTAB because they will not be part of the macro expansion. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments. The macro names are entered into NAMTAB, which serves as an index to DEFTAB. For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in DEFTAB. z The third data structure is an argument table (ARGTAB), which is used during the expansion of macro invocations. When a macro invocation statement is recognized, the arguments are stored in ARGTAB according to their position in the argument list. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.



16 a) Keyword Macro Parameters : With positional parameters, the programmer must be careful to specify the arguments in the proper order. If an argument is to be omitted, the macro invocation statement must contain a null argument (two consecutive commas) to maintain the correct argument positions.

For example, a certain macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only 3rd and 9th parameters are to be specified. Then, the macro invocation might look like GENER , , DIRECT, , , , , , 3.

Using a different form of parameter specification, called keyword parameters, each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. For example, if 3rd parameter in the previous example is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation statement would be GENER TYPE=DIRECT, CHANNEL=3.

16 b) Concatenation of Macro Parameters z Suppose that a program contains one series of variables named by the symbols XA1, XA2, XA3, …, another series named by XB1, XB2, XB3, …, etc. If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction. The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.). z Most macro processors deal with this problem by providing a special concatenation operator. This operator is the character Æ. For example, the statement LDA X&IDÆ1 so that the end of the parameter &ID is clearly identified. The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so Æ will not appear in the macro expansion. z Fig 4.6(a) shows a macro definition that uses the concatenation operator as previously described. Fig shows macro invocation statements and the corresponding macro expansions.

```
1  SUM    MACRO   &ID
2         LDA     X&ID→1
3         ADD     X&ID→2
4         ADD     X&ID→3
5         STA     X&ID→S
6         MEND
```

(a)

```
SUM     A
   ↓
LDA     XA1
ADD     XA2
ADD     XA3
STA     XAS
```

(b)

```
SUM     BETA
   ↓
LDA     XBETA1
ADD     XBETA2
ADD     XBETA3
STA     XBETAS
```


17) Algorithm for one Pass Macro Processor.

18)

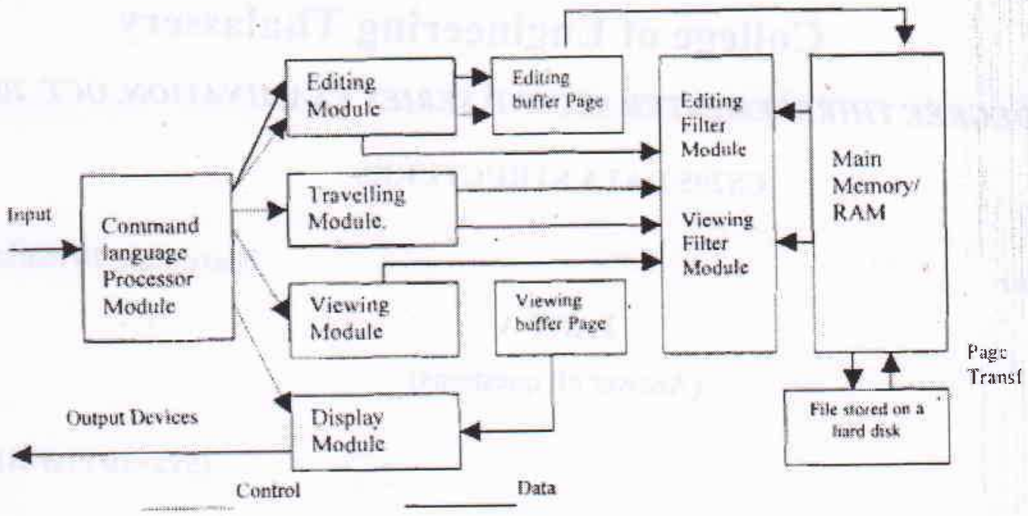


Figure: Typical editor structure

Most text editors have a structure similar to the above figure. The command language processor accepts input from the user's input devices, and analyses the tokens and syntactic structure of the commands. In this sense, the command language processor functions much like the lexical and syntactic phases of a compiler.

In a text editor, these semantic routines perform functions such as editing and viewing. Alternatively, the command language processor may produce an intermediate representation of the desired editing operations. This intermediate representation is then decoded by an interpreter that invokes the appropriate semantic routines. The use of an intermediate representation allows the editor to provide a variety of user-interaction languages with a single set of semantic routines that are driven from a common intermediate representation.

The semantic routines involve travelling, editing, viewing, and display functions. In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing module, which is the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user with travelling commands, such as next paragraph and next screen, or implicitly by the system as a side effect of the previous editing operation, such as delete paragraph.

The travelling module of the editor actually performs the setting of the current editing and viewing pointers. When the user issues an editing command, the editing module invokes the editing filter.

This component filters the document to generate a new editing buffer based on the current editing pointer' as well as on the editing filter parameters. These parameters, which are specified both by the user and the system, provide such information as the range of text that can be affected by an operation. Filtering may simply consist of the selection of contiguous characters beginning at the current point. The semantic routines of the editing component then operate on the editing buffer, which is essentially a filtered subset of the document data structure. Similarly, in viewing a document, the start of the area to be viewed is determined by the current viewing pointer.

This pointer is maintained by the viewing module of the editor, which is a collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user with a travelling command or implicitly by the system as a result of the previous editing operation. When the display needs to be updated, the viewing component invokes the viewing module. This component filters the document to generate a new viewing buffer based on the current viewing pointer as well as on the viewing filter parameters. I11 line editors, the viewing buffer can contain the current line; in screen editors, this buffer may contain a rectangular cutout of the quarter-plane of text. This viewing buffer is then passed to the display module of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen, called a window or viewport.

19) Debugging Functions and Capabilities: The most obvious requirement is for a set of unit test functions that can be specified by the programmer. One important group of such functions deals with execution sequencing, which is the observation and control of the flow of program execution. After execution is suspended, other debugging commands can be used to analyse the progress of the progrm and to diagnose error s detected; then execution of the program can be resumed. Given a good graphic representation of program progress, it may even be useful to enable the program to run at various speeds called gaits.

A debugging system should also provide function such as tracing and trace back. Tracing can be ~~TOOIS~~ used to track the flow of execution logic and data modifications. The control flow can be traced at different Ievels of detail namely module, subroutine, branch instruction and so on. Trace back can show the path by which the current statement is reached. For a given variable or parameter, traceback can show which statements modified it. Such information should be displayed symbolically.

It is also important for a debugging system to have good program display capabilities. It must be possible to display the program being debugged, complete with statement numbers. The system should save all the debugging specifications for a recompilation, so the programmer does not need to reissue all of these debugging commands.

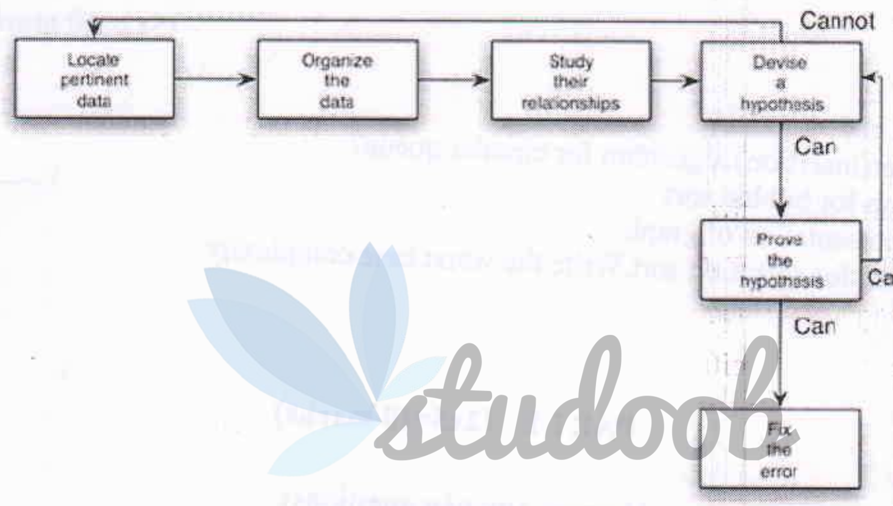
A debugging system should consider the language in which the program being debugged is written. Debugger commands that initiate actions and collect data about a program's execution should be common across languages. However, a debugging system must be sensitive to the specific language being debugged, so that procedural, arithmetic and conditional logic can be coded in the syntax of that language. These requirements have a number of consequences for the debugger and for the other software. When the debugger receives control, the execution of the program being debugged is temporarily suspended. The debugger must then be able to determine the language in which the program is written and set its context accordingly.

The notation used to specify certain debugging functions varies according to the language of the program & being debugged. The debugger must have access to information gathered by the language translator. The internal symbol dictionary formats vary widely between different language . translators. Future compilers and assemblers should aim toward a consistent interface with the debugging system. One approach is that the language translators to produce the needed information in a standard external form for the debugger regardless of the internal form used in the translator.

20) i) Debugging by Induction

It should be obvious that careful thought will find most errors without the debugger even going near the computer. One particular thought process is induction, where you move from the particulars of a situation to the whole. That is, start with the clues (the symptoms of the error, possibly the results of one or more test cases) and look for relationships among the clues. The induction process is illustrated in Figure 7.1

Figure 7.1. The inductive debugging process.

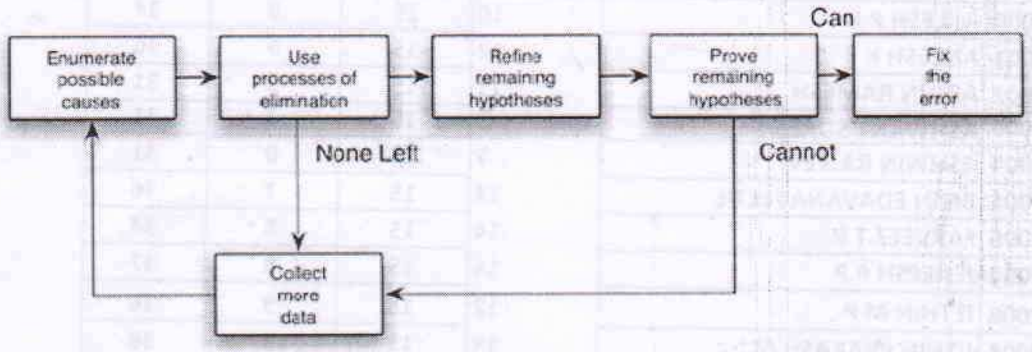


ii) Debugging by Deduction

The process of deduction proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error). See Figure 7.4.

As opposed to the process of induction in a murder case, for example, where you induce a suspect from the clues, you start with a set of suspects and, by the process of elimination (the gardener has a valid alibi) and refinement (it must be someone with red hair), decide that the butler must have done it. The steps are as follows:

Figure 7.4. The deductive debugging process.


Enumerate possible causes
Use processes of elimination
Refine remaining hypotheses
Prove remaining hypotheses
Can
Fix the error
Cannot
None Left
Collect more data