# APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

FIFH SEMESTER B.TECH DEGREE MODEL EXAMINATION, NOVEMBER 2017

Department: **Computer Science and Engineering**

**Subject: - CS303: System Software**

## PART A
**(Answer all questions)**

1. What is forward references? What is the difference between one pass and two pass assembler?
2. Write the sequence of instructions to perform the operation BETA = ALPHA + 1 using SIC instructions.
3. What is the need of SYMTAB(symbol table) & OPTAB in assembler
4. Explain immediate adressing ? Following is a memory configuration:

| Address | Value | Register R |
|---|---|---|
| 1 | 5 | 5 |
| 5 | 7 | |
| 6 | 5 | |

What is the result of the following statement:

ADD 6(immediate) to R (indirect)

**Total: (12marks)**

## PART B
**(Answer any two full questions)**

1. Describe the structure of a single pass assembler
2. Discuss in detail about the SIC & SIC/XE architecture.
3. What is meant by program relocation? Explain?

**Total: (18marks)**

**PART C**
**(Answer all questions)**

1. DefineMASM with features?What are the two different types of jump statements used in MASM assembler?
2. Define program block
3. What are the basic functions of loaders
4. Define absolute loader?What is meant by bootstrap loader?

**Total: (12marks)**

**PART D**
**(Answer any two questions)**

1. Discuss in detail about the machine-Independent Assembler features?
2. Explain about the Loader Design Options?
3. Explain in detail about MS-DOS Linker?

**Total: (18marks)**

**PART E**
**(Answer any Four questions)**

1. Explain in detail about the basic Macro Processor functions?
2. Explain about Macro Processor Design options? Explain in detail about MASM Macro Processor?
3. Explain in detail about the following i) Editing process ii) User Interface
4. Explain about various software tools?
5. Explain about the editor structure & discuss about debugging functions and capabilities?

**Total: (40marks)**

# APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

FIFH SEMESTER B.TECH DEGREE MODEL EXAMINATION, NOVEMBER 2017

Department: **Computer Science and Engineering**

**Subject: - CS303: System Software**

**PART A**
**(Answer all questions)**

1. What is forward references? What is the difference between one pass and two pass assembler?

The forward referencing problem arises in a very simple way: the expression evaluator attempts to evaluate an operand which is a symbol by searching the symbol table, and finds that the symbol is not in the symbol table. The symbol is not defined, the expression cannot be evaluated, and the assembly language statement cannot be assembled.It is a reference to a label that is defined later in a program. Different problems can be solved using One Pass or Two Pass forward referencing

Consider the statement

10 1000 STL RETADR

80 1036 RETADR RESW 1

The first instruction contains a forward reference RETADR.If we attempt to translate the

program line by line,we will unable to process the statement in line10 because we do not

know the address that will be assigned to RETADR .The address is assigned later(in line

80) in the program.

The difference between one pass and two pass assemblers is basically in the name. A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references and assemble code in one pass. A two pass assembler does two passes over the source file ( the second pass can be over a file generated in the first pass ). In the first pass all it does is looks for label definitions and introduces them in the symbol table. In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations and so on.

2. Write the sequence of instructions to perform the operation BETA = ALPHA + 1 using SIC

instructions.

```
LDA    ALPHA
```

```
              ADD    ONE
              STA    BETA
              ....   ....
ALPHA         RESW   1
BETA          RESW   1
ONE           RESW   1
```

3. What is the need of SYMTAB(symbol table) & OPTAB in assembler

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB)

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.
- In pass 1 the OPTAB is used to look up and validate the operation code in the source program.
- In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.
- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.
- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flagsnto indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization

**4.** Explain immediate adressing ? Following is a memory configuration:

| Address | Value | Register R |
|---|---|---|
| 1 | 5 | 5 |
| 5 | 7 | |
| 6 | 5 | |

What is the result of the following statement:

ADD 6(immediate) to R (indirect)

In this addressing mode the operand value is given directly. There is no need to refer memory. The immediate addressing is indicated by the prefix „#".

Eg: ADD #5

In this instruction one operand is in accumulator and the second operand is a immediate value the value 5 is directly added with the accumulator content and the result is stored in accumulator.

Here 6 is the immediate data and the next value is indirect data.ie the register contains the address of the operand. Here the address of the operand is 5 and its corresponding value is 7. 6

+ [R] = 6+ [5] = 6+ 7 =13

**PART B**
**(Answer any two full questions)**

1. Describe the structure of a single pass assembler

Assemblers a program that turns symbols into machine instructions. ISA-specific:close correspondence between symbols and instruction set mnemonics for opcodes, labels for memory locations, additional operations for allocating storage and initializing data

Each line of a program is one of the following:

•An instruction

•An assembler directive (or pseudo-op)

•A comment

Whitespace (between symbols) and case are ignored. Comments (beginning with ";") are also ignored

Assembler Design can be done in:

–Single pass –Two pass

•Single Pass Assembler:

–Does everything in single pass

–Cannot resolve the forward referencing

The ability to compile in a single pass is often seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Many languages were designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

2. Discuss in detail about the SIC & SIC/XE architecture.

SIC Machine Structure:

Memory:

• It consists of bytes(8 bits) ,words (24 bits which are consecutive 3 bytes) addressed by the location of their lowest numbered byte.

• There are totally 32,768 bytes in memory.

Registers:

There are 5 registers namely

1. Accumulator (A)
2. Index Register(X)
3. Linkage Register(L)
4. Program Counter(PC)
5. Status Word(SW).

• Accumulator is a special purpose register used for arithmetic operations.

• Index register is used for addressing.

• Linkage register stores the return address of the jump of subroutine instructions (JSUB).

• Program counter contains the address of the current instructions being executed.

• Status word contains a variety of information including the condition code.

Data formats:

• Integers are stored as 24-bit binary numbers: 2's complement representation is used for negative values characters are stored using their 8 bit ASCII codes.

• They do not support floating – point data items.

Instruction formats:

All machine instructions are of 24-bits wide

• X-flag bit that is used to indicate indexed-addressing mode.

Addressing modes:

Opcode (8) X (1) Address (15)

• Two types of addressing are available namely,

1. Direct addressing mode

2. Indexed addressing mode or indirect addressing mode

Mode Indication Target Address calculation

Direct X=0 TA=Address

Indexe

d

X=1 TA=Address + (X)

• Where(x) represents the contents of the index register(x)

Instruction set:

It includes instructions like:

1. Data movement instruction

Ex: LDA, LDX, STA, STX.

2. Arithmetic operating instructions

Ex: ADD, SUB, MUL, DIB.

This involves register A and a word in memory, with the result being left in the

register.

3. Branching instructions

Ex: JLT, JEQ, TGT.

4. Subroutine linkage instructions

Ex: JSUB, RSUB.

Input and Output:

• I/O is performed by transferring one byte at a time to or from the rightmost 8 bits

of register A.

• Each device is assigned a unique 8-bit code.

• There are 3 I/O instructions,

1) The Test Device (TD) instructions tests whether the addressed device is

ready to send or receive a byte of data.

2) A program must wait until the device is ready, and then execute a Read

Data (RD) or Write Data (WD).

3) The sequence must be repeated for each byte of data to be read or written.

1.3 SIC/XE ARCHITECTURE & SYSTEM SPECIFICATION

6

Memory:

• 1 word = 24 bits (3 8-bit bytes)

• Total (SIC/XE) = 220 (1,048,576) bytes (1Mbyte)

Registers:

• 10 x 24 bit registers

| MNEMONIC | Register | Purpose |
|---|---|---|
| A | 0 | Accumulator |
| X | 1 | Index register |
| L | 2 | Linkage register (JSUB/RSUB) |
| B | 3 | Base register |
| S | 4 | General register |
| T | 5 | General register |
| F | 6 | Floating Point Accumulator (48 bits) |
| PC | 8 | Program Counter (PC) |
| SW | 9 | Status Word (includes Condition Code, CC) |

Data Format:

• Integers are stored in 24 bit, 2's complement format

• Characters are stored in 8-bit ASCII format

• Floating point is stored in 48 bit signed-exponent-fraction format:

• The fraction is represented as a 36 bit number and has value between 0 and 1.

• The exponent is represented as a 11 bit unsigned binary number between 0 and 2047.

• The sign of the floating point number is indicated by s : 0=positive, 1=negative.

• Therefore, the absolute floating point number value is: f*2(e-1024)

Instruction Format:

• There are 4 different instruction formats available:

Format 1 (1 byte):

op {8}

s exponent {11} fraction {36}

Format 2 (2 bytes): op {8} r1 {4} r2 {4}

Format 3 (3 bytes): op {6} n i x b p e displacement {12}

Format 4 (4 bytes):Formats 3 & 4 introduce addressing mode flag bits:

• n=0 & i=1

Immediate addressing - TA is used as an operand value (no memory reference)

• n=1 & i=0

Indirect addressing - word at TA (in memory) is fetched & used as an address to fetch the operand from

• n=0 & i=0

Simple addressing TA is the location of the operand

• n=1 & i=1

Simple addressing same as n=0 & i=0

Flag x:

x=1 Indexed addressing add contents of X register to TA calculation

Flag b & p (Format 3 only):

• b=0 & p=0

Direct addressing displacement/address field containsTA (Format 4 always uses

direct addressing)

• b=0 & p=1

PC relative addressing - TA=(PC)+disp (-2048<=disp<=2047)*

• b=1 & p=0

Base relative addressing - TA=(B)+disp (0<=disp<=4095)**

Flag e:

e=0 use Format 3

e=1 use Format 4

8

op {6} n i x b p e address {20}

Instructions:

SIC provides 26 instructions, SIC/XE provides an additional 33 instructions (59 total)

SIC/XE has 9 categories of instructions:

• Load/store registers (LDA, LDX, LDCH, STA, STX, STCH, etc.)

• integer arithmetic operations (ADD, SUB, MUL, DIV) these will use register A

and a word in memory, results are placed into register A

• compare (COMP) compares contents of register A with a word in memory and

sets CC (Condition Code) to <, >, or =

• conditional jumps (JLT, JEQ, JGT) - jumps according to setting of CC

• subroutine linkage (JSUB, RSUB) - jumps into/returns from subroutine using register L

• input & output control (RD, WD, TD) - see next section

• floating point arithmetic operations (ADDF, SUBF, MULF, DIVF)

• register manipulation, operands-from-registers, and register-to-register arithmetics

(RMO, RSUB, COMPR, SHIFTR, SHIFTL, ADDR, SUBR, MULR, DIVR, etc)

Input and Output (I/O):

• 28(256) I/O devices may be attached, each has its own unique 8-bit address

• 1 byte of data will be transferred to/from the rightmost 8 bits of register A

Three I/O instructions are provided:

• RD Read Data from I/O device into A

• WD Write data to I/O device from A

• TD Test Device determines if addressed I/O device is ready to send/receive a byte

of data. The CC (Condition Code) gets set with results from this test:

< device is ready to send/receive

= device isn't ready

SIC/XE Has capability for programmed I/O (I/O device may input/output data while CPU

does other work) - 3 additional instructions are provided:

• SIO Start I/O

• HIO Halt I/O

• TIO Test I/O

**3.** What is meant by program relocation? Explain?

The need for program relocation

• It is desirable to load and run several programs at the same time.

• The system must be able to load programs into memory wherever there is room.

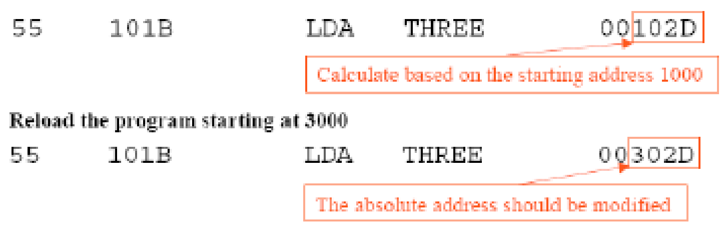
• The exact starting address of the program is not known until load time.

Absolute Program

• Program with starting address specified at assembly time

• The address may be invalid if the program is loaded into somewhere else.

• Example:



```
55      101B          LDA    THREE          00102D
```

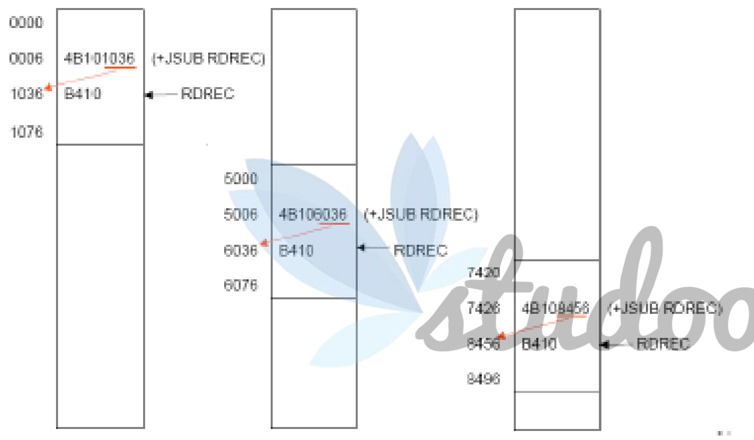
Calculate based on the starting address 1000

Reload the program starting at 3000

```
55      101B          LDA    THREE          00302D
```

The absolute address should be modified

**Example: Program Relocation**



Example: Program Relocation

• The only parts of the program that require modification at load time are those that specify direct addresses.

• The rest of the instructions need not be modified.

• Not a memory address (immediate addressing)

• PC-relative, Base-relative

• From the object program, it is not possible to distinguish the address and constant.

o The assembler must keep some information to tell the loader.

- The object program that contains the modification record is called a relocatable program.

The way to solve the relocation problem

• For an address label, its address is assigned relative to the start of the program(START 0)

• Produce a Modification record to store the starting location and the length of the address field to be modified.

• The command for the loader must also be a part of the object program.

Modification record

• One modification record for each address to be modified

• The length is stored in half-bytes (4 bits)

• The starting location is the location of the byte containing the leftmost bits of the address field to be modified.

• If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

## PART C
**(Answer all questions)**

- Define MASM with features ?What are the two different types of jump statements used in MASM assembler?

The Microsoft Macro Assembler (MASM) is an assembler for the x86 family of microprocessors, originally produced Microsoft MS-DOS operating system.

The features of MASM are listed below:

i. It supported a wide variety of macro facilities and structured programming idioms, including high-level constructions for looping, procedure calls and alternation (therefore, MASM is an example of a high-level assembler).

ii. MASM is one of the few Microsoft development tools for which there was no separate 16-bit and 32-bit version.

iii. Assembler affords the programmer looking for additional performance a three pronged approach to performance based solutions.

iv. MASM can build very small high performance executable files that are well suited where size and speed matter.

v. When additional performance is required for other languages, MASM can enhance the performance of these languages with small fast and powerful dynamic link libraries.

Near jump

A near jump is a jump to a target in the same segment and it is assembled by
using a current code segment CS.

Far jump

A far jump is a jump to a target in a different code segment and it is assembled
by using different segment registers.

**2**. Define program block

Allow the generated machine instructions and data to appear in the object program in a different order

• Separating blocks for storing code, data, stack, and larger data block

• Program blocks versus. Control sections

o Program blocks

Segments of code that are rearranged within a single object program unit.

o Control sections

Segments of code that are translated into independent object program units.

• Assembler rearranges these segments to gather together the pieces of each block and assign address.

• Separate the program into blocks in a particular order

• Large buffer area is moved to the end of the object program

• Program readability is better if data areas are placed in the source program close to the statements that reference them.

Assembler directive: USE

• USE [blockname]

• At the beginning, statements are assumed to be part of the unnamed (default) block

• If no USE statements are included, the entire program belongs to this single block

• Each program block may actually contain several separate segments of the source program

Example

Three blocks are used

• default: executable instructions.

• CDATA: all data areas that are less in length.

• CBLKS: all data areas that consists of larger blocks of memory

Assembler directive: USE

• USE [blockname]

• At the beginning, statements are assumed to be part of the unnamed (default)block

• If no USE statements are included, the entire program belongs to this single block

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

**3.** What are the basic functions of loaders

BASIC LOADER FUNCTIONS

Fundamental functions of a loader:

1. Bringing an object program into memory.
2. Starting its execution.

Loading – brings the object program into memory for execution

Bootstrapping

Actions taken when a computer is first powered on

The hardware logic reads a program from address 0 of ROM (Read Only Memory)

ROM is installed by the manufacturer ROM contains bootstrapping program and some other routines that controls hardware (e.g. BIOS)

Bootstrapping is a loader Loads OS from disk into memory and makes it run The location of OS on disk (or floppy) usually starts at the first sector Starting address in memory is usually fixed to 0 no need of relocation This kind of loader is simple no relocation no linking called "absolute loader"

2. Relocation – modifies the object program so that it can be loaded at an address different from the location originally specified

Assembler review

- Assembler generates an object code assuming that the program starts at memory address 0

- Loader decides the starting address of a program
- Assembler generates modification record
- Limits of modification record

Record format (address, length)

- It can be huge when direct addressing is frequently used
- If instruction format is fixed for absolute addressing, the length part can be removed
- Instead of address field, bit-vector can be used

1101100..... means instruction 1 2 4 5. .. need to be modified

Hardware support for relocation

Base register assembler and loader do not need to worry about relocation

3. Linking – combines two or more separate object programs and also supplies the information needed to reference them.

- Requirements for linking Each module defineswhich symbols are used by other modules symbols undefined in a module are assumed to be defined in other modules if these symbols are declared explicitly, it helps linker to resolve
- Principles Assembler evaluates as much as possible expressions If some cannot be resolved, provide the modification records

**4.**Define absolute loader?What is meant by bootstrap loader?

Absolute Loader :An absolute loader is the simplest of loaders. Its function is simply to take the output of the assembler and load it into memory. The output of the assembler can be stored on any machine-readable form of storage, but most commonly it is stored on punched cards or magnetic tape, disk, or drum

For a simple absolute loader, all functions are accomplished in a single pass as follows:

1) The Header record of object programs is checked to verify that the correct program has been presented for loading.

2) As each Text record is read, the object code it contains is moved to the indicated address in memory.

3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

BootStrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called a bootstrap loader, is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.

Working of a simple Bootstrap loader

• The bootstrap begins at address 0 in the memory of the machine.

• It loads the operating system at address 80.

• Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits just as it is in a Text record of a SIC object program.

• The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.

• After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

• Much of the work of the bootstrap loader is performed by the subroutine GETC.

• GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes = C "D8"à 4438 H converting to one byte 'D8'H.

• The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.

• The TIXR instruction is then used to add 1 to the value in X.

**PART D**

**(Answer any two questions)**

**1**.Discuss in detail about the machine-Independent Assembler features?

- Literals

• The programmer writes the value of a constant operand as a part of the instruction that uses it. This avoids having to define the constant elsewhere in the program and make a label for it.

• Such an operand is called a Literal because the value is literally in the instruction.

• Consider the following example

• It is convenient to write the value of a constant operand as a part of instruction.

• A literal is identified with the prefix =, followed by a specification of the literal value.

• Example:

Literals vs. Immediate Operands

• Literals The assembler generates the specified value as a constant at some other memory location.

• Immediate Operands The operand value is assembled as part of the machine instruction

• We can have literals in SIC, but immediate operand is only valid in SIC/XE.

Literal Pools

• Normally literals are placed into a pool at the end of the program

• In some cases, it is desirable to place literals into a pool at some other location in the object program

• Assembler directive LTORG

o When the assembler encounters a LTORG statement, it generates a literal pool (containing all literal operands used since previous LTORG)

• Reason: keep the literal operand close to the instruction

o Otherwise PC-relative addressing may not be allowed

Duplicate literals

• The same literal used more than once in the program

o Only one copy of the specified value needs to be stored

o For example, =X’05’

• Inorder to recognize the duplicate literals

o Compare the character strings defining them

- Easier to implement, but has potential problem
- e.g. =X 05

o Compare the generated data value

- Better, but will increase the complexity of the
- assembler
- e.g. =C EOF and =X 454F46

Problem of duplicate-literal recognition

• ‘*’ denotes a literal refer to the current value of program counter

o BUFEND EQU *

• There may be some literals that have the same name, but different values

o BASE *

o LDB =* (#LENGTH)

• The literal =* repeatedly used in the program has the same name, but different values

• The literal "=*" represents an "address" in the program, so the assembler must generate the appropriate "Modification records".

Literal table - LITTAB

23

• Content

o Literal name

o Operand value and length

o Address

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

Implementation of Literals

Pass 1

• Build LITTAB with literal name, operand value and length, leaving the address unassigned

• When LTORG or END statement is encountered, assign an address to each literal not yet assigned an address updated to reflect the number of bytes occupied by each literal

Pass 2

• Search LITTAB for each literal operand encountered

• Generate data values using BYTE or WORD statements

• Generate Modification record for literals that represent an address in the program

SYMTAB & LITTAB

Symbol-Defining Statements

• Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.

Assembler directive used is EQU.

• Syntax: symbol EQU value

• Used to improve the program readability, avoid using magic numbers, make it easier to find and change constant values

• Replace +LDT #4096 with

MAXLEN EQU 4096

+LDT #MAXLEN

• Define mnemonic names for registers.

A EQU 0 RMO A,X

X EQU 1

• Expression is allowed

MAXLEN EQU BUFEND-BUFFER

Assembler directive ORG

• Allow the assembler to reset the PC to values

o Syntax: ORG value

• When ORG is encountered, the assembler resets its LOCCTR to the specified value.

• ORG will affect the values of all labels defined until the next ORG.

• If the previous value of LOCCTR can be automatically remembered, we can return to the normal use of LOCCTR by simply writing

o ORG

Example: using ORG

• If ORG statements are used

• We can fetch the VALUE field by

LDA VALUE,X

X = 0, 11, 22, … for each entry

Forward-Reference Problem

• Forward reference is not allowed for either EQU or ORG.

• All terms in the value field must have been defined previously in the program.

• The reason is that all symbols must have been defined during Pass 1 in a two-pass assembler.

• Allowed:

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

• Not Allowed:

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

Expressions

• The assemblers allow “the use of expressions as operand”

• The assembler evaluates the expressions and produces a single operand address or value.

• Expressions consist of

Operator

o +,-,*,/ (division is usually defined to produce an integer result)

Individual terms

o Constants

o User-defined symbols

o Special terms, e.g., *, the current value of LOCCTR

• Examples

MAXLEN EQU BUFEND-BUFFER

STAB RESB (6+3+2)*MAXENTRIES

Relocation Problem in Expressions

• Values of terms can be

o Absolute (independent of program location)

- constants

o Relative (to the beginning of the program)

- Address labels
- (value of LOCCTR)

• Expressions can be

• Absolute

o Only absolute terms.

o MAXLEN EQU 1000

• Relative terms in pairs with opposite signs for each pair.

MAXLEN EQU BUFEND-BUFFER

• Relative

All the relative terms except one can be paired as described in “absolute”. The remaining unpaired relative term must have a positive sign. STAB EQU OPTAB + (BUFEND – BUFFER)

Restriction of Relative Expressions

• No relative terms may enter into a multiplication or division operation

o 3 * BUFFER

• Expressions that do not meet the conditions of either “absolute” or “relative”
should be flagged as errors.

o BUFEND + BUFFER

o 100 – BUFFER

Handling Relative Symbols in SYMTAB

• To determine the type of an expression, we must keep track of the types of all
symbols defined in the program.

• We need a “flag” in the SYMTAB for indication.

Program Blocks

• Allow the generated machine instructions and data to appear in the object
program in a different order

• Separating blocks for storing code, data, stack, and larger data block

• Program blocks versus. Control sections

o Program blocks

- Segments of code that are rearranged within a single object program unit.

o Control sections

- Segments of code that are translated into independent object program units.

• Assembler rearranges these segments to gather together the pieces of each block and assign address.

• Separate the program into blocks in a particular order

• Large buffer area is moved to the end of the object program

• Program readability is better if data areas are placed in the source program close to the statements that reference them.

Assembler directive: USE

• USE [blockname]

• At the beginning, statements are assumed to be part of the unnamed (default) block

• If no USE statements are included, the entire program belongs to this single block

• Each program block may actually contain several separate segments of the source program

Example

Three blocks are used

• default: executable instructions.

• CDATA: all data areas that are less in length.

• CBLKS: all data areas that consists of larger blocks of memory.

**3**.Explain about the Loader Design Options?

Linking loaders perform all linking and relocation at load time.

• There are two alternatives:

1. Linkage editors, which perform linking prior to load time.
2. Dynamic linking, in which the linking function is performed at execution time.

• Precondition: The source program is first assembled or compiled, producing an object program.

• A linking loader performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.

• A linkage editor produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

Linkage Editors

• The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.

• This means that the loading can be accomplished in one pass with no external symbol table required.

• If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.

• Linkage editors can perform many useful functions besides simply preparing an object program for execution. Ex., a typical sequence of linkage editor commands used:

INCLUDE PLANNER (PROGLIB)

DELETE PROJECT {delete from existing PLANNER}
INCLUDE PROJECT (NEWLIB) {include new version}
REPLACE PLANNER (PROGLIB)

• Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages.

• Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control editor

Dynamic Linking

• Linkage editors perform linking operations before the program is loaded for execution.

• Linking loaders perform these same operations at load time.

• Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program
when it is first called.

• Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C.

• With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.

• Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.

**3**.Explain in detail about MS-DOS Linker?

MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program. This executable program has the file name extension. EXE. LINK can also combine the translated programs with other modules from object code libraries.

MS-DOS compilers and assemblers produce object modules(.OBJ).

Each .OBJ contains a binary image of the translated instructions and data of the program.

MS-DOS LINK is a linkage editor that combines one-or more modules to produce a complete executable program (.exe).

MS-DOS object module.

Pass 1:

Computes a starting address for each segment in the program.

(i) Segments are placed in the same order as given in SEGDEF records.

(ii) Segments from different object modules that have the same segment name and class are combined.

(iii) Segments with the same class, but different names are concatenated.

(iv) Segments starting address is updated as these combinations and concatenation are performed.

Pass 2:

Extracts the translated instructions from the object modules.

(i) Process each LEDATA and LIDATA record along with FIXUPP

(ii) Relocation operation's that involve the starting address of the segment are added to a table of Segment Fixupps.

(iii) Build an image of the executable program in memory.

(iv) Write it to the executable (.EXE.) file.

(v) This file includes a header that contains table of fixupps,information about memory requirements .

**PART E**
**(Answer any *Four*f ull questions)**

**1.**Explain in detail about the basic Macro Processor functions?

The fundamental functions common to all macro processors are:

1. Macro Definition

2. Macro Invocation

3. Macro Expansion

Macro Definition and Expansion

• Two new assembler directives are used in macro definition:

o MACRO: identify the beginning of a macro definition

o MEND: identify the end of a macro definition

• Prototype for the macro:

o Each parameter begins with '&' label op operands name MACRO parameters

```
 :
 body
 :
 MEND
```

• Body: The statements that will be generated as the expansion of the macro.

It shows an example of a SIC/XE program using macro Instructions.

• This program defines and uses two macro instructions, RDBUFF and WRDUFF .

• The functions and logic of RDBUFF macro are similar to those of the RDBUFF subroutine.

• The WRBUFF macro is similar to WRREC subroutine.

• Two Assembler directives (MACRO and MEND) are used in macro definitions.

• The first MACRO statement identifies the beginning of macro definition.

• The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field identify the parameters of macro instruction.

• In our macro language, each parameter begins with character &, which facilitates the substitution of parameters during macro expansion.

• The macro name and parameters define the pattern or prototype for the macro instruction used by the programmer. The macro instruction definition has been deleted since they have been no longer needed after macros are expanded.

• Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.

• The arguments and parameters are associated with one another according to their positions.

Macro Invocation

• A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.

• The processes of macro invocation and subroutine call are quite different.

o Statements of the macro body are expanded each time the macro is invoked.

o Statements of the subroutine appear only one; regardless of how many times the subroutine is called.

• The macro invocation statements treated as comments and the statements generated from macro expansion will be assembled as though they had been written by the programmer.

Recursive Macro Expansion

• RDCHAR:

o read one character from a specified device into register A

o should be defined beforehand (i.e., before RDBUFF)

Implementation of Recursive Macro Expansion

• Previous macro processor design cannot handle such kind of recursive macro invocation and expansion, e.g., RDBUFF BUFFER, LENGTH, F1

• Reasons:

1) The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.

2) The Boolean variable EXPANDING would be set to FALSE when the"inner" macro expansion is finished, that is, the macro process would forget that it had been in the middle of expanding an "outer" macro.

3) A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.

• Solutions:

1) Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.

2) Use a stack to take care of pushing and popping local variables and return addresses.

• Another problem: can a macro invoke itself recursively?

One-Pass Macro Processor

• A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition.

• Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro.

Handling Recursive Macro Definition

• In DEFINE procedure

o When a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached.

o This would not work for recursive macro definition because the first MEND encountered in the inner macro will terminate the whole macro definition process.

o To solve this problem, a counter LEVEL is used to keep track of the level of macro definitions.

▪ Increase LEVEL by 1 each time a MACRO directive is read.

▪ Decrease LEVEL by 1 each time a MEND directive is read.

▪ A MEND can terminate the whole macro definition process only when LEVEL reaches 0.

▪ This process is very much like matching left and right parentheses
when scanning an arithmetic expression.

Two-Pass Macro Processor

• Two-pass macro processor

o Pass 1:

▪ Process macro definition

o Pass 2:

▪ Expand all macro invocation statements

• Problem

o This kind of macro processor cannot allow recursive macro definition, that is, the body of a macro contains definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

Example of Recursive Macro Definition

• MACROS (for SIC)

o Contains the definitions of RDBUFF and WRBUFF written in SIC
instructions.

• MACROX (for SIC/XE)

o Contains the definitions of RDBUFF and WRBUFF written in SIC/XE
instructions.

• A program that is to be run on SIC system could invoke MACROS whereas a program to be run on SIC/XE can invoke MACROX.

• Defining MACROS or MACROX does not define RDBUFF and WRBUFF.
These definitions are processed only when an invocation of MACROS or

MACROX is expanded.

An interactive editor is a computer program that allows a user to create and revise a target document. The term document includes objects such as computer programs, texts, equations, tables, diagrams, line art and photographs-anything that one might find on a printed page. Text editor is one in which the primary elements being edited are character strings of the target text. The document editing process is an interactive usercomputer dialogue designed to accomplish four tasks:

1) Select the part of the target document to be viewed and manipulated
2) Determine how to format this view on-line and how to display it.
3) Specify and execute operations that modify the target document.
4) Update the view appropriately.

Traveling – Selection of the part of the document to be viewed and edited. It involves first traveling through the document to locate the area of interest such as “next screenful”, ”bottom”,and “find pattern”. Traveling specifies where the area of interest is;

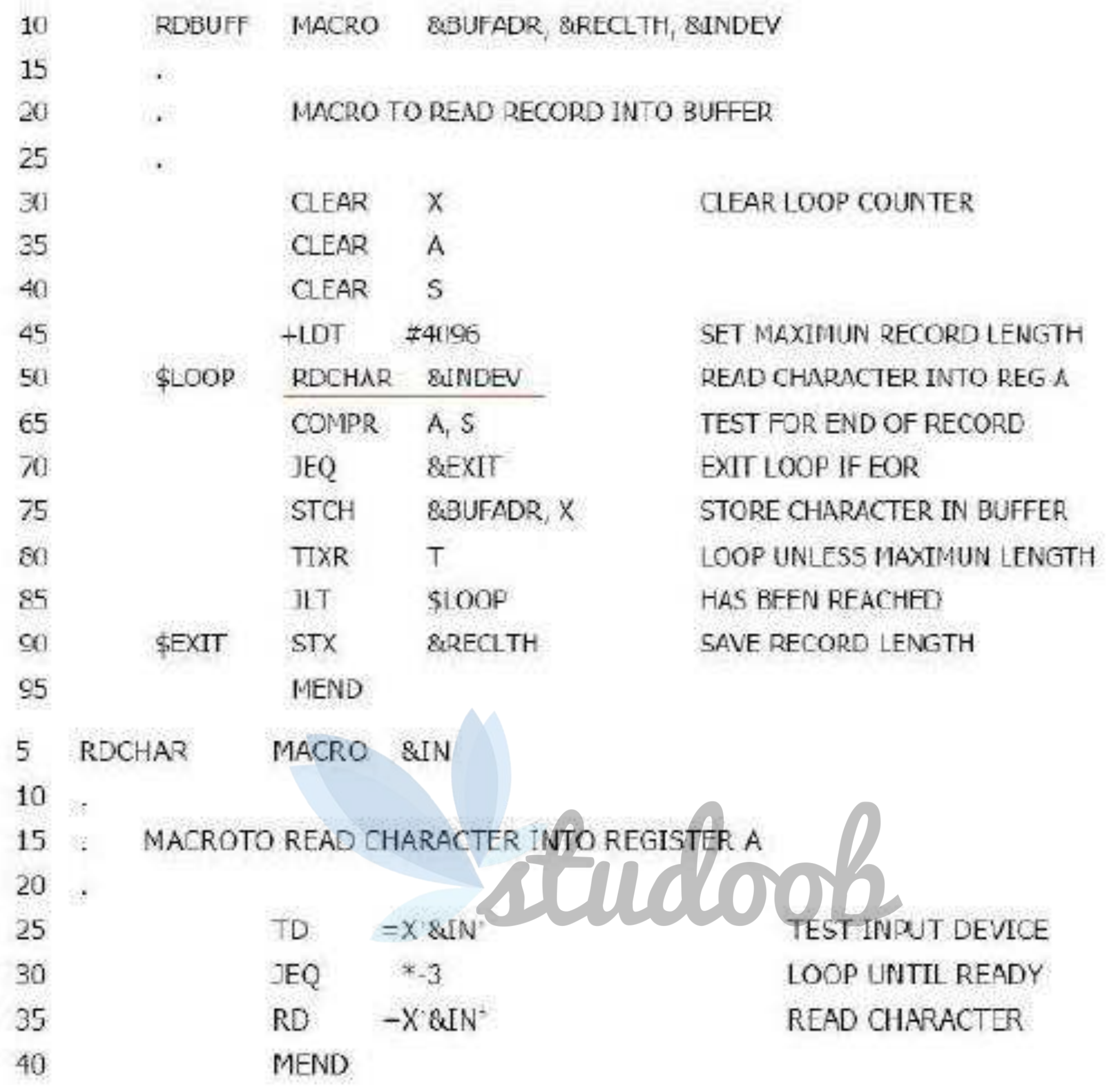
Filtering - The selection of what is to be viewed and manipulated is controlled by filtering. Filtering extracts the relevant subset of the target document at the point of interest such as next screenful of text or next statement.

Formatting: Formatting determines how the result of filtering will be seen as a visible representation (the view) on a display screen or other device.

Editing: In the actual editing phase, the target document is created or altered with a set of operations such as insert, delete, replace, move or copy.

Manuscript oriented editors operate on elements such as single characters, words, lines, sentences and paragraphs; Program-oriented editors operates on elements such as identifiers, keywords and statements.

2.We have seen an example of the definition of one macro instruction by another. But we have not dealt with the invocation of one macro by another. The following example shows the invocation of one macro by another macro.



```
RDBUFF  MACRO   &BUFADR, &RECLTH, &INDEV
.
.       MACRO TO READ RECORD INTO BUFFER
.
        CLEAR   X                 CLEAR LOOP COUNTER
        CLEAR   A
        CLEAR   S
        +LDT    #4096             SET MAXIMUN RECORD LENGTH
$LOOP   RDCHAR  &INDEV            READ CHARACTER INTO REG A
        COMPR   A, S              TEST FOR END OF RECORD
        JEQ     &EXIT             EXIT LOOP IF EOR
        STCH    &BUFADR, X        STORE CHARACTER IN BUFFER
        TIXR    T                 LOOP UNLESS MAXIMUN LENGTH
        JLT     $LOOP             HAS BEEN REACHED
$EXIT   STX     &RECLTH           SAVE RECORD LENGTH
        MEND
```

```
RDCHAR  MACRO   &IN
.
.       MACROTO READ CHARACTER INTO REGISTER A
.
        TD      =X'&IN'           TEST INPUT DEVICE
        JEQ     *-3               LOOP UNTIL READY
        RD      =X'&IN'           READ CHARACTER
        MEND
```

**Problem of Recursive Expansion**

Macro Processor

Problem of Recursive Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion
- The procedure EXPAND would be called recursively, thus the invocationarguments in the ARGTAB will be overwritten. (P.201)
- The Boolean variable EXPANDING would be set to FALSE when the “inner” macro expansion is finished, i.e., the macro process would forget that it had been in the middle of expanding an “outer” macro.

Solutions

- Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
- If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

MASM86The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows. The earlier versions were released in the year 1981 .TurboAssembler (TASM) is also an assembler package developed by Borland which runs on andproduces code for 16- or 32-bit x86 MS-DOS or Microsoft Windows. These softwares providethe assembly language tools to program the 8086 processor based systems.To program the x86 processors using in MASM , certain concepts are very important. They arereserved words, identifiers, predefined symbols, constants, expressions, operators, data types, registers,and statements.Reserved Words : A reserved word has a special meaning fixed by the language. This must beused under certain special conditions. These reserved words are Instructions, which correspond to operations the processor can execute. Directives, which give commands to the assembler. Attributes, which provide a value for a field, such as segment alignment. Operators, which are used in expressions.MASM reserved words are not case sensitive except for predefined symbols. The assemblergenerates an error if you use a reserved word as a variable.

**3.** The document editing process is an interactive usercomputer dialogue designed to accomplish four tasks:

1) Select the part of the target document to be viewed and manipulated

2) Determine how to format this view on-line and how to display it.

3) Specify and execute operations that modify the target document.

4) Update the view appropriately.

Traveling – Selection of the part of the document to be viewed and edited. It involves first traveling through the document to locate the area of interest such as “next screenful”, ”bottom”,and “find pattern”. Traveling specifies where the area of interest is;
Filtering - The selection of what is to be viewed and manipulated is controlled by filtering. Filtering extracts the relevant subset of the target document at the point of

interest such as next screenful of text or next statement.
Formatting: Formatting determines how the result of filtering will be seen as a visible representation (the view) on a display screen or other device.
Editing: In the actual editing phase, the target document is created or altered with a set of operations such as insert, delete, replace, move or copy.
Manuscript oriented editors operate on elements such as single characters, words, lines, sentences and paragraphs; Program-oriented editors operates on elements such as identifiers, keywords and statements

**THE USER-INTERFACE OF AN EDITOR.**

The user of an interactive editor is presented with a conceptual model of the editing system. The model is an abstract framework on which the editor and the world on which the operations are based. The line editors simulated the world of the keypunch they allowed operations on numbered sequence of 80-character card image lines.
The Screen-editors define a world in which a document is represented as a quarter-plane of text lines, unbounded both down and to the right. The user sees, through a cutout, only a rectangular subset of this plane on a multi line display terminal. The cutout can be moved left or right, and up or down, to display other portions of the document. The user interface is also concerned with the input devices, the output devices, and the interaction language of the system.
INPUT DEVICES: The input devices are used to enter elements of text being edited, to enter commands, and to designate editable elements. Input devices are categorized as: 1) Text devices 2) Button devices 3) Locator devices
1) Text or string devices are typically typewriter like keyboards on which user presses and release keys, sending unique code for each key. Virtually all computer key boards are of the QWERTY type.
2) Button or Choice devices generate an interrupt or set a system flag, usually causing an invocation of an associated application program. Also special function keys are also available on the key board. Alternatively, buttons can be simulated in software by displaying text strings or symbols on the screen. The user chooses a string or symbol instead of pressing a button.
3) Locator devices: They are two-dimensional analog-to-digital converters that position

a cursor symbol on the screen by observing the user‟s movement of the device. The most common such devices are the mouse and the tablet.
The Data Tablet is a flat, rectangular, electromagnetically sensitive panel. Either the ballpoint pen like stylus or a puck, a small device similar to a mouse is moved over the surface. The tablet returns to a system program the co-ordinates of the position on the data tablet at which the stylus or puck is currently located. The program can then map these data-tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. Text devices with arrow (Cursor) keys can be used to simulate locator devices. Each of these keys shows an arrow that point up, down, left or right. Pressing an arrow key typically generates an appropriate character sequence; the program interprets this sequence and moves the cursor in the direction of the arrow on the key pressed.
VOICE-INPUT DEVICES: which translate spoken words to their textual equivalents, may prove to be the text input devices of the future. Voice recognizers are currently available for command input on some systems.
OUTPUT DEVICES The output devices let the user view the elements being edited and the result of the editing operations.

- The first output devices were teletypewriters and other character-printing terminals that generated output on paper.
- Next "glass teletypes" based on Cathode Ray Tube (CRT) technology which uses CRT screen essentially to simulate the hard-copy teletypewriter. Today‟s advanced CRT terminals use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and pages.
- The modern professional workstations are based on personal computers with high resolution displays; support multiple proportionally spaced character fonts to produce realistic facsimiles of hard copy documents

**4.**Explain about various software tools?
Text editor
An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of

"knowledge workers" as they compose, organize, study, and manipulate computer-based information. A text editor allows you to edit a text file (create, modify etc…). For example the Interactive text editors on Windows OS - Notepad, WordPad, Microsoft Word, and text editors on UNIX OS - vi, emacs, jed, pico. Normally, the common editing features associated with text editors are, Moving
the cursor, Deleting, Replacing, Pasting, Searching, Searching and replacing, Saving and loading, and, Miscellaneous(e.g. quitting).
These are text editors more sophisticated programs with text editor as front ends. Editor in two modes one is command mode and second is that data mode. In such editor edit are command and data like mix up edit. In the data mode the user keys in the text to be added to the file Failure to recognize the current mode of the editor can lead to mix up of command & data.
Program testing and debugging In selection of testing data for the program, analysis of test results to detect errors and debugging. Ex . localization and removal of errors. Software tools to assist the programmer in these steps come in the following forms. 1. Test data generators help the user in selecting test data for his program. It use through user get help and done thoroughly tested. 2. Automated test drivers help in regression testing.
10. Þ These are given as inputs to the test driver. The driver subject one at time organizes execution of the program on the data. Þ Fig of process Sets of test data Test driver
**5**.Explain about the text editor structure & discuss about debugging functions and capabilities?
Most text editors have a structure similar to that shown in the following figure.
That is most text editors have a structure similar to shown in the figure regardless of features and the computers
Command language Processor accepts command, uses semantic routines –
performs functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions.
Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.
In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or

reset due to next paragraph, next screen, cut paragraph, paste paragraph etc..,.
When editing command is issued, editing component invokes the editing filter –
generates a new editing buffer – contains part of the document to be edited from current
editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer
being created.
In viewing a document, the start of the area to be viewed is determined by the
current viewing pointer maintained by the viewing component. Viewing component is a
collection of modules responsible for determining the next view. Current viewing pointer
can be set or reset as a result of previous editing operation. When display needs to be updated, viewing component invokes the viewing filter – generates a new viewing buffer – contains part of the document to be viewed from
current viewing pointer. In case of line editors – viewing buffer may contain the current
line, Screen editors - viewing buffer contains a rectangular cutout of the quarter plane of
the text. Viewing buffer is then passed to the display component of the editor, which
produces a display by mapping the buffer to a rectangular subset of the screen – called a
window. The editing and viewing buffers may be identical or may be completely disjoint. Identical – user edits the text directly on the screen. Disjoint – Find and Replace (For
example, there are 150 lines of text, user is in 100th line, decides to change all
occurrences of 'text editor' with 'editor'). The editing and viewing buffers can also be
partially overlap, or one may be completely contained in the other. Windows typically
cover entire screen or a rectangular portion of it. May show different portions of the same
file or portions of different file. Inter-file editing operations are possible. The components of the editor deal with a user document on two levels: In main
memory and in the disk file system. Loading an entire document into main memory may
be infeasible – only part is loaded – demand paging is used – uses editor paging routines. Documents may not be stored sequentially as a string of characters. Uses separate editor
data structure that allows addition, deletion, and modification with a minimum of I/O and
character movement
An interactive debugging system provides programmers with facilities that aid in
testing and debugging of programs. Many such systems are available during these days. Our discussion is broad in scope, giving the overview of interactive debugging systems –

not specific to any particular existing system. Here we discuss
- Introducing important functions and capabilities of IDS
- Relationship of IDS to other parts of the system
- The nature of the user interface for IDS

• Debugging Functions and Capabilities

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.A Debugging system should also provide functions such as tracing and traceback. Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on… Traceback can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

Program-Display capabilities

A debugger should have good program-display capabilities. Program being debugged should be displayed completely with statement numbers. The program may be displayed as originally written or with macro expansion. Keeping track of any changes made to the programs during the debugging session. Support for symbolically displaying or modifying the contents of any of the variables and constants in the program. Resume execution – after these changes.

To provide these functions, a debugger should consider the language in which the program being debugged is written. A single debugger – many programming languages – language independent. The debugger - a specific programming language – language dependent. The debugger must be sensitive to the specific language being debugged. The context being used has many different effects on the debugging interaction. The statements are different depending on the language

Cobol - MOVE 6.5 TO X

Fortran - X = 6.5

C - X = 6.5

Examples of assignment statements

Similarly, the condition that X be unequal to Z may be expressed as

Cobol - IF X NOT EQUAL TO Z

Fortran - IF ( X.NE.Z)

C - IF ( X <> Z)

Similar differences exist with respect to the form of statement labels, keywords and so on…

The notation used to specify certain debugging functions varies according to the language of the program being debugged. Sometimes the language translator itself has debugger interface modules that can respond to the request for debugging by the user. The source code may be displayed by the debugger in the standard form or as specified by the user or translator.

It is also important that a debugging system be able to deal with optimized code. Many optimizations like

- Invariant expressions can be removed from loops
- Separate loops can be combined into a single loop
- Redundant expression may be eliminated
- Elimination of unnecessary branch instructions

Leads to rearrangement of segments of code in the program. All these optimizations create problems for the debugger, and should be handled carefully. 4.2.3 Relationship with Other Parts of the System

The important requirement for an interactive debugger is that it always be available. Must appear as part of the run-time environment and an integral part of the system. When an error is discovered, immediate debugging must be possible. The debugger must communicate and cooperate with other operating system components such as interactive subsystems. Debugging is more important at production time than it is at application- development time. When an application fails during a production run, work dependent on that application stops. The debugger must also exist in a way that is consistent with the security and integrity components of the system. The debugger must coordinate its activities with those of existing and future language compilers and interpreters.