MAHARASHTRA STATE BOARD OF TECHNICAL EDUCATION
(Autonomous)
(ISO/IEC - 27001 - 2005 Certified)

SUMMER-15 EXAMINATION
Model Answer

Subject Code: 17634 **Subject Name: System Programming**

Important Instructions to examiners:

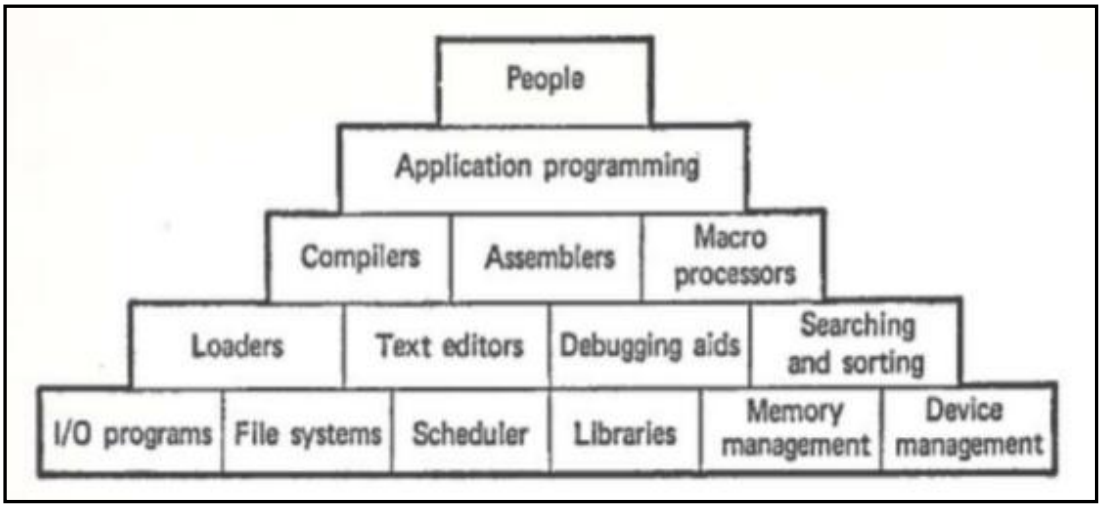
1) The answers should be examined by key words and not as word-to-word as given in themodel answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may try to assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given more Importance (Not applicable for subject English and Communication Skills).
4) While assessing figures, examiner may give credit for principal components indicated in the figure. The figures drawn by candidate and model answer may vary. The examiner may give credit for anyequivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constantvalues may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgment on part of examiner of relevant answer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalentconcept.

1. Attempt any FIVE of the following: **MARKS 20**

a) Explain and draw the neat labeled diagram of foundation of system programming.

(2 Marks – Diagram, 2 Marks Explanation)

Ans:



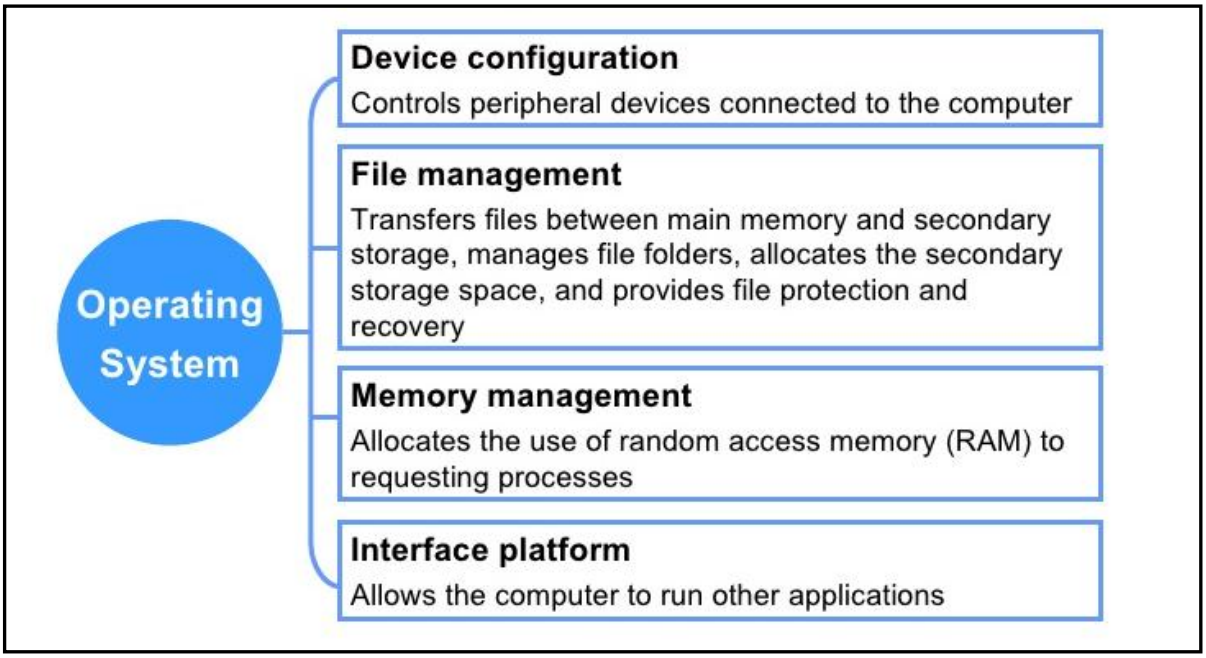
1. System programming is the important activity of programming system software.
2. It is necessary for the programmers to make assumptions for the software and hardware and other properties of the system that program runs on.
3. System programming is different from application programming.
4. Limited programming facilities are available with system programming.

b) Explain the function of Operating System.

(Diagram 2 Marks, Explanation 2 Marks)

Ans:

i. The operating system is the core software component of your computer.
ii. It performs many functions and is, in very basic terms, an interface between your computer and the outside world.
iii. In the section about hardware, a computer is described as consisting of several component parts including your monitor, keyboard, mouse, and other parts.
iv. The operating system provides an interface to these parts using what is referred to as "drivers".



c) Explain searching and describe any one method in detail.

(Searching 1 Mark, any one Search with example 3 Marks)

Ans:

i. Searching for data is one of the fundamental fields of computing.
ii. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set.
iii. The use of a hash table or binary search tree will result in more efficient searching, but more often than not an array or linked list will be used.
iv. The various searches are linear search, binary search

Linear Search:The most obvious algorithm is to start at the beginning and walk to the end, testing for a match at each item:

```
booljw_search ( int *list, int size, int key, int*& rec )
{
  // Basic sequential search
bool found = false;
int i;

for ( i = 0; i < size; i++ ) {
if ( key == list[i] )
break;
  }
if ( i < size ) {
found = true;
rec = &list[i];
  }

return found;
}
```

This algorithm has the benefit of simplicity; it is difficult to get wrong, unlike other more sophisticated solutions. The above code follows the convention of this article, they are as follows:

- All search routines return a true/false boolean value for success or failure.
- The list will be either an array of integers or a linked list of integers with a key.
- The found item will be saved in a reference to a pointer for use in client code.

Binary Search:In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater than the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our FindInCollection function can now be implemented:

```
static void *bin_search( collection c, int low, int high, void *key ) {
        int mid;
        /* Termination check */
        if (low > high) return NULL;
        mid = (high+low)/2;
        switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
                /* Match, return item found */
                case 0: return c->items[mid];
                /* key is less than mid, search lower half */
                case -1: return bin_search( c, low, mid-1, key);
                /* key is greater than mid, search upper half */
                case 1: return bin_search( c, mid+1, high, key );
                default : return NULL;
                }
        }
void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
  Pre-condition:
        c is a collection created by ConsCollection
        c is sorted in ascending order of the key
        key != NULL
  Post-condition: returns an item identified by key if
one exists, otherwise returns NULL
*/
        int low, high;
        low = 0; high = c->item_cnt-1;
        returnbin_search( c, low, high, key );
}
```

- bin_search is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
- There is a termination condition, If low > high then the partition to be searched has no elements in it and If there is a match with the element in the middle of the current partition, then we can return immediately.

- AddToCollection will need to be modified to ensure that each item added is placed in its correct place in the array.

- The procedure is simple: Search the array until the correct spot to insert the new item is found,Move all the following items up one position and Insert the new item into the empty position thus created.bin_search is declared static. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.

d) Explain various types of assembly statements.
(List 1 Mark; Explanation 1 Mark each)

Ans:

The various types of assemble statements are:

1. **Imperative Statement**: The imperative statements indicate action to be performed during execution of program. Each imperative statement is translated into one machine instruction.

2. **Declaration Statement**: Two declaration statements are used
   - [Label] DS<constant>: Declare Storage (DS) reserves memory area and associates name with the memory area.
   - [Label] DC<value>: Declare Constant (DC) constructs a memory word containing constant.

3. **Assembler Directives**: These directives are used to guide the assembler to perform certain actions during assembly of program.
   - START<constant>: This directive indicates the first word of the target program generated by the assembler to be placed in the memory word whose address is specified in the <constant>
   - END [<operand spec>]: This directive indicates the end of the source program.

e) Explain dynamic loading mechanism in loader.

(1Mark each point)

Ans:

i. Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory.
ii. It is one of the 3 mechanisms by which a computer program can use some other software; the other two are static linking and dynamic linking.
iii. Unlike static linking and dynamic linking, dynamic loading allows a computer program to start up in the absence of these libraries.
iv. To discover available libraries, and to potentially gain additional functionality.

f) Describe radix sort with suitable example.

(Radix sort 2 Marks, Example 2 Marks)

Ans:

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the digits of the numbers.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the one's digits:

Digit Sublist

0	340 710
1	
2	812 582
3	493
4	

5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The order in which we divide and reassemble the list is extremely important, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the ten's digit:

Digit Sublist

0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the hundred's digit:

Digit Sublist

0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

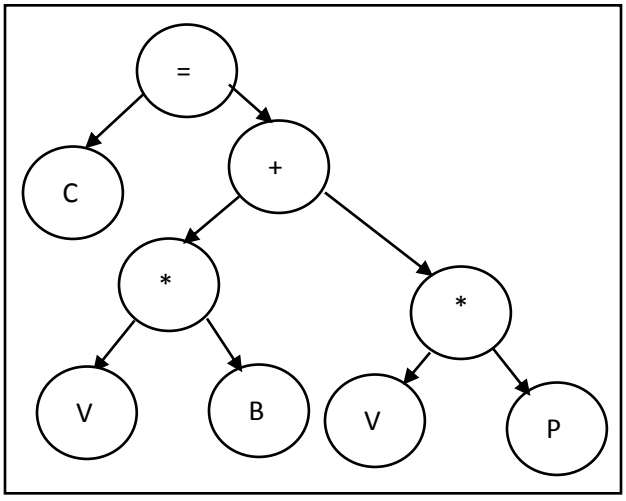
195 340 385 437 493 582 710 715 812

g) Draw the parse tree for the following expression.

C = V * B + V * P

(Diagram- 4 Marks)

Ans:



2. Attempt any TWO of the following: MARKS 16

a) Explain four components of system programming.

(2 Marks Each)

Ans:

The components of system software are

- **Assembler**: The program known as assembler is written to automate the translation of assembly language to machine language. Input to the language is called as source program and output of assembler is machine language translation called as object program.

- **Loader**: Loader is a system program which places program into the memory and prepares for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

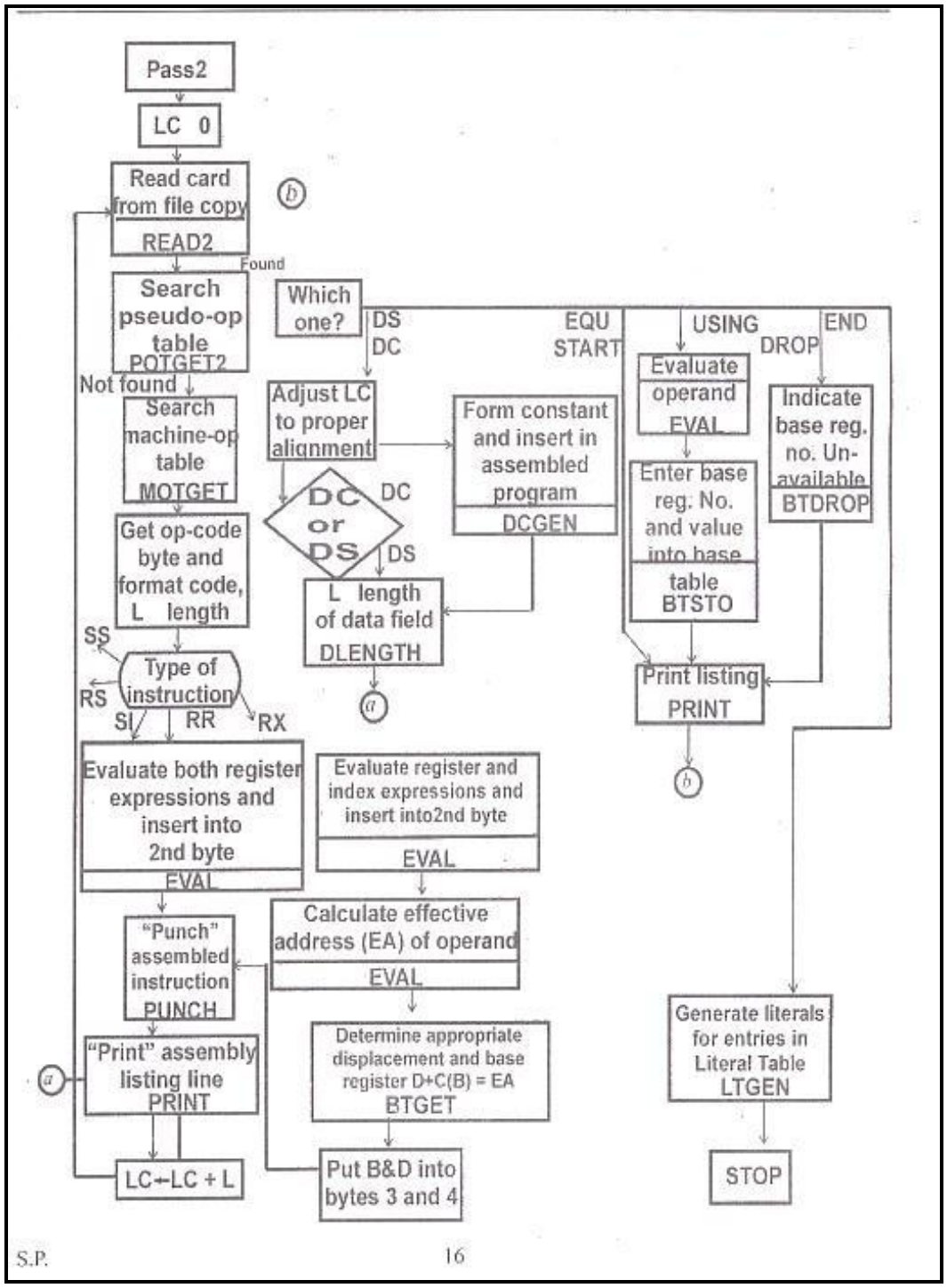
- **Macro**: A macro is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to a replacement output sequence (also often a sequence of characters) according to a defined procedure. The mapping process that instantiates (transforms) a macro use into a specific sequence is known as macro expansion. A facility for writing macros may be provided as part of a software application or as a part of a programming language. In the former case, macros are used to make tasks using the application less repetitive. In the latter case, they are a tool that allows a programmer to enable code reuse or even to design domain-specific languages.

- **Compiler**: A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).The most common reason for converting a source code is to create an executable program.

b) Draw the flowchart for the pass 2 of a two pass assembler.

(8 Marks Diagram)

Ans:

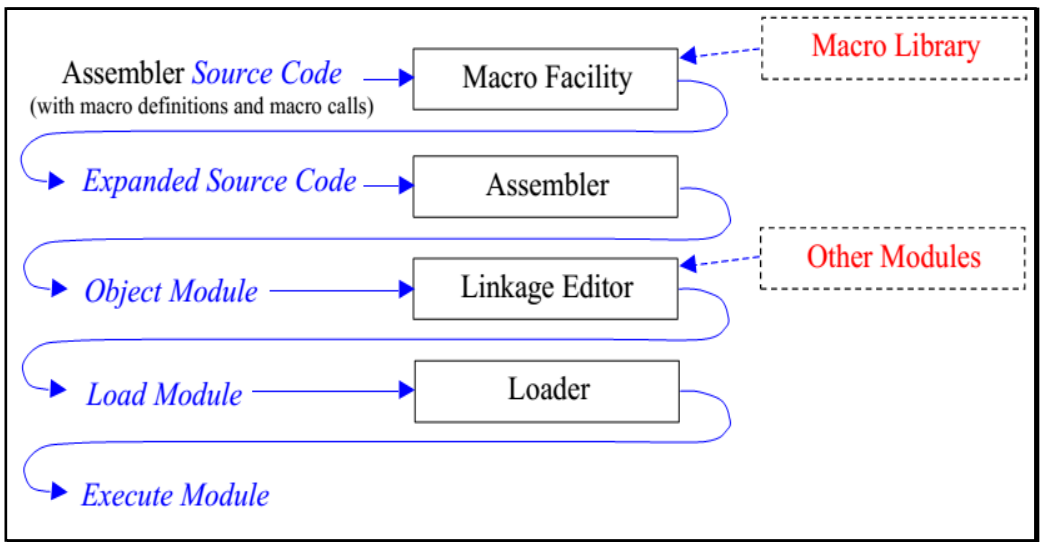

Pass2
LC 0
Read card from file copy
READ2
Found
Search pseudo-op table
POTGET2
Not found
Search machine-op table
MOTGET
Get op-code byte and format code, L length
SS
Type of instruction
RS
SI
RR
RX
Evaluate both register expressions and insert into 2nd byte
EVAL
"Punch" assembled instruction
PUNCH
"Print" assembly listing line
PRINT
LC←LC + L
Which one?
DS
DC
Adjust LC to proper alignment
DC or DS
DC
DS
L length of data field
DLENGTH
Form constant and insert in assembled program
DCGEN
EQU
START
USING
DROP
END
Evaluate operand
EVAL
Enter base reg. No. and value into base table
BTSTO
Indicate base reg. no. Un-available
BTDROP
Print listing
PRINT
Evaluate register and index expressions and insert into2nd byte
EVAL
Calculate effective address (EA) of operand
EVAL
Determine appropriate displacement and base register D+C(B) = EA
BTGET
Put B&D into bytes 3 and 4
Generate literals for entries in Literal Table
LTGEN
STOP
S.P.
16

c) Explain macroprocessor along with conditional macro with example.

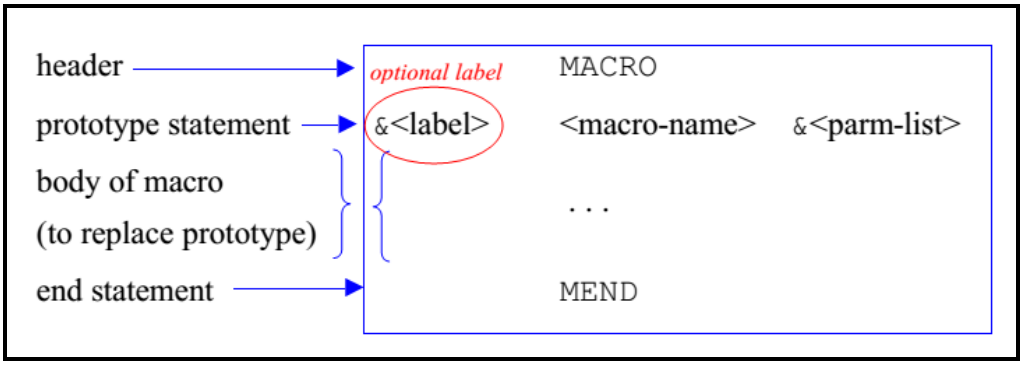
(2 Marks for Macro Processor Explanation; 3 Marks for AIF; 3 Marks for AGO)

Ans:

i. An assembly language macro is a template whose format represents a pattern of 0 or more assembly language statementsthat might be common to multiple programs.
ii. For this purpose, a macro language is used to provide a syntax fordefining macros.
iii. Where a sequence of assembly language statements can be represented by a macro, a macro call is inserted into the assembly program source code where the assembly code would otherwisego.
iv. A macro facility is used to interpret macro definitions and expand each macro call as it occurs with the requisite pattern of assembly language statements, providing expanded source code ready forthe assembler.
v. Assembly language macro definitions can be predefined and placed in a macro library, or can be included "in-line" with theassembly language program.



vi. Macro definition format



vii. Conditional Macro:Conditional Assembly in a macro facility refers to mechanisms for providing program control over the code generation process. Theserequire the addition of macro facility commands such as

- IF-ELSE-ENDIF
- WHILE-ENDW
- GOTO

along with branch point labels (e.g., !EXIT) and representationalforms for comparison; e.g.,
EQ for = NE for ¹
LT for < LTE for <=
GT for > GTE for >=
and logical operators AND, OR, NOT with the usual parenthesis grouping.

3. Attempt any TWO of the following: MARKS 16

a) Explain different types of Macro Arguments.
(Dummy Argument - 4 Marks, Keyword Arguments - 4 Marks)
Ans:-

Dummy argument - An identifier used to indicate where an actual argument is to be substituted in the macro, module, or directive.

Actual argument - The argument (value) used in the macro, directive or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL-HDL.

Dummy arguments are specified in macro declarations and in the bodies of macros, modules and directives. The dummy argument is preceded by a question mark in the places where an actual argument is to be substituted. The question mark distinguishes the dummy arguments from other ABEL-HDL identifiers occurring in the source file.

Take for example, the following macro declaration arguments:

OR_EM MACRO (a,b,c) { ?a # ?b # ?c };

This defines a macro named OR_EM that is the logical OR of three arguments. These arguments are represented in the definition of the macro by the dummy arguments, a, b, and c. In the body of the macro, which is surrounded by braces, the dummy arguments are preceded by question marks to indicate that an actual argument will be substituted.

The equation

D = OR_EM (x,y,z&1);

invokes the OR_EM macro with the actual arguments, x, y, and z&1. This results in the equation:

D = x # y # z&1;

Arguments are substituted into the source file before checking syntax and logic, so if an actual argument contains unsupported syntax or logic, the compiler detects and reports the error.

Keyword Parameters

names one or more macro parameters followed by equal signs. You can specify default values after the equal signs. If you omit a default value after an equal sign, the keyword parameter has a null value. Using default values enables you to write more flexible macro definitions and reduces the number of parameters that must be specified to invoke the macro. To override the default value, specify the macro variable name followed by an equal sign and the new value in the macro invocation.

Syntax:-

keyword-parameter=<**value**><. . . ,**keyword-parameter-n**=<**value**>>

b) Explain basic function of Macro Processor along with its database used.

(Each function 2 Marks)

Ans:

1. **Recognizing macro definitions:** A macro pre-processor must recognize macro definitions that are identified by the MACRO and MEND pseudo-ops. The macro definitions can be easily recognized, but this task is complicated in cases where the macro definitions appear within macros. In such situations, the macro pre-processor must recognize the nesting and correctly matches the last MEND with the first MACRO.

MDT

Macro Definition Table

& LAB	INCR	& ARG1,&ARG2,&ARG3
#0	A	1, #1
	A	2,#2
	A	3,#3
	MEND	

2. **Saving the definitions:** The pre-processor must save the macro instructions definitions that can be later required for expanding macro calls.

MNT

Macro Name Table

1	"INCRbbbb"	15
.	.	.
.	.	.
.	.	.
.	.	.

3. **Recognizing macro calls:** The pre-processor must recognize macro calls along with the macro definitions. The macro calls appear as operation mnemonics in a program.

ALA

Index	Argument
0	"bbbbbbbb" (all Blank)
1	"Data3bbb"
2	"Data2bbb"
3	"Data1bbb"

4. **Replacing macro definitions with macro calls:** The pre-processor needs to expand macro calls and substitute arguments when any macro call is encountered. The pre-processor must substitute macro definition arguments within a macro call.

c) Explain Pass I Algorithm of two pass macroprocessor.

Ans: (Explanation 4 Marks; Flowchart 4 Marks)

The first pass processes the definition of the macro by checking each operation code of the macro. In first pass, each operation code is saved in a table called Macro Definition Table (MDT). Another table is also maintained in first pass called Macro Name Table (MNT). First pass uses various other databases such as Macro Name Table Counter (MNTC) and Macro Name Table Counter (MDTC). The various databases used by first pass are:

1. The input macro source deck.
2. The output macro source deck copy that can be used by pass 2.
3. The Macro Definition Table (MDT), which can be used to store the body of the macro definitions. MDT contains text lines and every line of each macro definition, except the MACRO line gets stored in this table. For example, consider the code described in macro expansion section where macro INC used the macro definition of INC in MDT. Table 2.1 shows the MDT entry for INC macro:

4. The Macro Name Table (MNT), which can be used to store the names of defined macros. Each MNT entry consists of a character string such as the macro name and a pointer such as index to the entry in MDT that corresponds to the beginning of the macro definition. Table 2.2 shows the MNT entry for INCR macro:

5. The Macro Definition Table Counter (MDTC) that indicates the next available entry in the MDT.

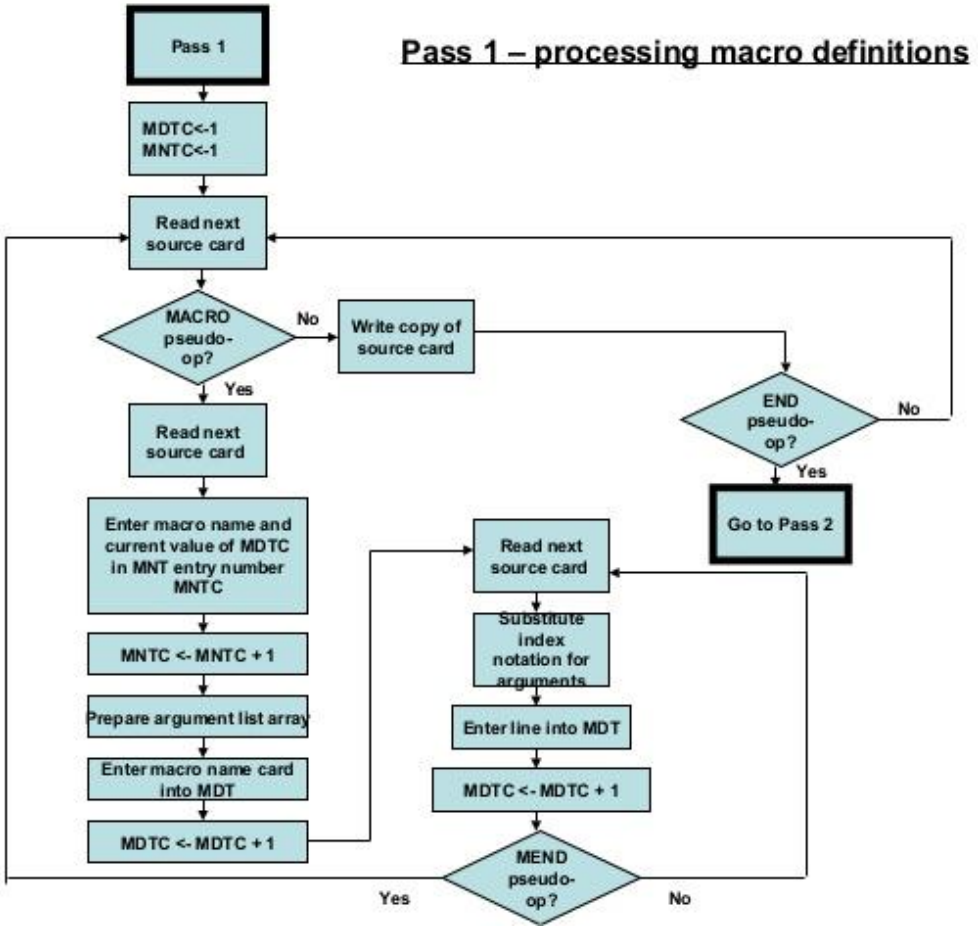
6. The Macro Name Table Counter (MNTC) that indicates the next available entry in the MNT.

7. The Argument List Array (ALA) that can be used to substitute index markers for dummy arguments prior to store a macro definition. ALA is used during both the passes of the macro pre-processor. During Pass 1, dummy arguments in the macro definition are replaced with positional indicators when the macro definition is stored. These positional indicators are used to refer to the memory address in the macro expansion. It is done in order to simplify the later argument replacement during macro expansion. The ith dummy argument on the macro name card is represented in the body of the macro by the index marker symbol #. The # symbol is a symbol reserved for the use of macro pre-processor.

Pass 1 algorithm examines each line of the input data for macro pseudo opcode. Following are the steps that are performed during Pass 1 algorithm:
1. Initialize MDTC and MNTC with value one, so that previous value of MDTC and MNTC is set to value one.
2. Read the first input data.
3. If this data contains MACRO pseudo opcode then

- Read the next data input.
- Enter the name of the macro and current value of MDTC in MNT.
- Increase the counter value of MNT by value one.
- Prepare that argument list array respective to the macro found.
- Enter the macro definition into MDT. Increase the counter of MDT by value one.
- Read next line of the input data.
- Substitute the index notations for dummy arguments passed in macro.
- Increase the counter of the MDT by value one.
- If mend pseudo opcode is encountered then next source of input data is read.
- Else expands data input.

4. If macro pseudo opcode is not encountered in data input then A. A copy of input data is created. B. If end pseudo opcode is found then go to Pass 2. C. Otherwise read next source of input data.



4. Attempt any TWO of the following: **MARKS 16**

a) Explain top down passes with example.

***[Note: Instead of passes, parser is consider]**

(Recursive Descent Parsing 3 Marks; Backtracking 3 Marks; example 1 Mark each)

Ans:

Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

Recursive descent parsing: It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking. Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.
This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.
Predictive parsers are much faster than backtracking ones Predictive parsers operate in linear time. Backtracking parsers operate in exponential time.

Back tracking: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S →rXd|rZd
X →oa|ea
Z →ai
For an input string: read, a top-down parser will behave like this:
It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).
Now the parser matches all the input letters in an ordered manner. The string is accepted.

b) Explain phases of a compiles according to the problem statement.

(List of Phases 2 Marks; Explanation 1 Mark for each phase)
Ans:

1. Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:
<token-name, attribute-value>

2. **Syntax Analysis**
The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

3. **Semantic Analysis**
Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

4. **Intermediate Code Generation**
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5. **Code Optimization**
The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

6. **Code Generation**
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

c) Explain main functions of storage Assignment along with database used.

(1 Mark for each function, 1 Mark for each database)

The purpose of these phases is to:

1. assign storage to all variables referenced in the source program

2. Assign storage to all temporary locations that are necessary for intermediate result, e.g. the result of matrix lines. These storage references were reserved by the interpretation phase and dis not appear in the source code.
3. Assign storage to literals.
4. Ensure that the storage is allocated and appropriate locations are initialized (literals and any variables with the initial attribute).

To do this it must insert information for the code generation phase into the identifier table, the literal table , and the matrix.

DATA BASES

Identifier table – created by lexical analysis and furnished with all attribute entries by interpretation phase. Storage assignment designates locations to all identifier that denote data. The next two phases use this information to generate accesses to the identifiers.

Temporary storage table –(part of identifier table) created by the interpretation phase to describe the temporary results of computations in the matrix. This table may be implemented as part of the identifier table since much of the information is of the same format. Further, note that all storage of this type all storage of this type is implemented as automatic, e.g., it is not physically allocated until the procedure is activated, This is satisfactory since no other procedure ever references it, nor is the space needed unless the procedure that contains the storage is activated.

Name	Base	Scale	Precision	Storage class	Array bounds	Structure info	Initial value	Address

MI	BASE	SCALE	PRECISION	AUTOMATIC	OTHER		ADDRESS

Storage class	Size	

Operator Operand Operand

Initialize	Identifier or literal	

Literal table – The storage assignment phase assigns al literal address and places an entry in the matrix to denote that code generation should allocate this storage.

Matrix – storage assignment places entries ito the matrix to ensure that code generation creates enough storage for identifiers, temporary storage, and literals. It also ensures that the appropriate ones are initialized.

Matrix entries by storage allocation

5. Attempt any TWO of the following: MARKS 16

a) Explain with matrix optimization for the expression and draw a tree diagram.

Cost = a * (b-c) + 2 * a* (b-c)

(4 Marks for explanation, 4 Marks for tree diagram)

Ans:

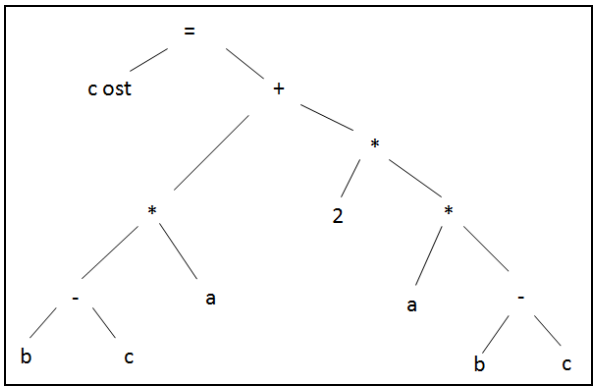
- When a subexpression occurs in a same statement more than once, we can delete all duplicate matrix entries and modify all references to the deleted entry so that they refer to the remaining copy of that subexpression as shown in following figure.
- Compile time computation of operations, both of whose operands are constants
- Movement of computations involving operands out of loops
- Use of the properties of boolean expressions to minimize their computation
- Machine independent optimization of matrix should occur before we use the matrix as a basis for code generation.

	Operator	Operand1	Operand2	Matrix entries
1	-	b	C	M1
2	*	a	M1	M2
3	*	2	A	M3
4	-	b	C	M4
5	*	M3	M4	M5
6	+	M2	M5	M6
7	=	COST	M6	

	operator	Operand1	Operand2	Matrix entries
1	-	b	C	M1
2	*	A	M1	M2
3	*	2	M2	M3
4	+	M3	**M2**	M4
5	=	COST	M4	

Matrix with common subexpressions Matrix after elimination of common subexpressions

Tree Diagram:



b) For the given expression generate the machine dependent and independent code
cost = Rate * (start - finish) + 2 * Rate * (start - finish)

(4 Marks for machine dependent code, 4 Marks for machine independent code)

Ans:

- Machine dependent optimization .

Optimized Matrix				First try		Improved code				
				L	1, START	L	1, START			
1	-	START	FINISH	S	1, FINISH	S	1, FINISH	M1	→	R1
				ST	1, M1					
				L	1, RATE	L	3, RATE			
2	*	RATE	M1	M	0, M1	MR	2, 1	M2	→	R3
				ST	1, M2					
				L	1, =F'2'	L	5, = F'2'			
3	*	2	M2	M	0, M2	M	4, M2	M3	→	R5
				ST	1, M3					
				L	1, M2					
4	+	M2	M3	A	1, M3	AR	3, 7	M4	→	R7
				ST	1, M4	ST	3, COST			
5	=	M4	COST	ST	1, COST					

- Machine independent optimization of matrix for code generation basis for code generation

	Operator	Operand 1	Operand 2	Matrix entries
1	-	START	FINISH	M1
2	*	RATE	M1	M2
3	*	2	RATE	M3
4	-	START	FINISH	M4
5	*	M3	M4	M5
6	+	M2	M5	M6
7	=	COST	M6	

	operator	Operand 1	Operand 2	Matrix entries
1	-	START	FINISH	M1
2	*	RATE	M1	M2
3	*	2	M2	M3
4	+	M3	**M2**	M4
5	=	COST	M4	

Matrix with common subexpressions Matrix after elimination of common subexpressions

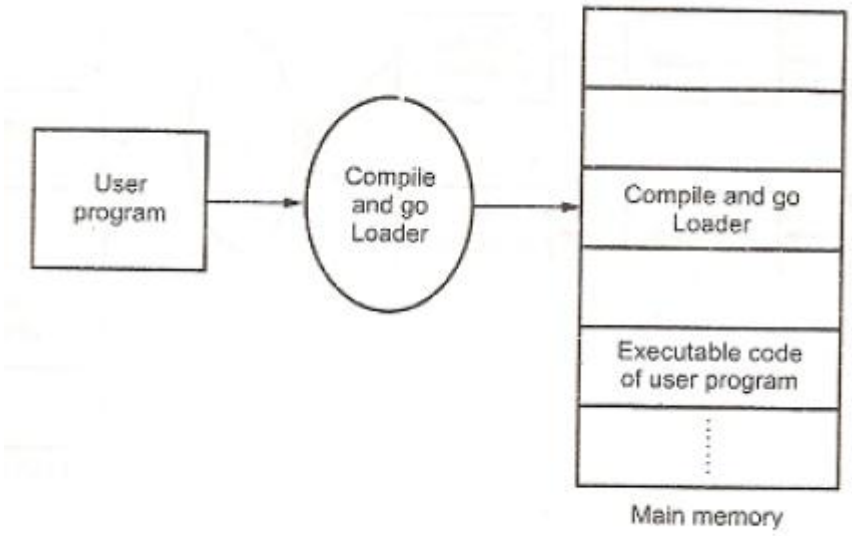
c) Explain any two loaders schemes.

Ans: (8 Marks for any two)

Loader Schemes:
Based on the various functionalities of loader, there are various types of loaders:
1) **"compile and go" loader:** in this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it's a FORTRAN compiler which uses such "load and go" scheme. This loading scheme is also called as "assemble and go".

Advantages:
• This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.
Disadvantages:
• In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.

• There is no production of .objfile, the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.
• It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.
• For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.
• The execution time will be more in this scheme as every time program is assembled and then executed.

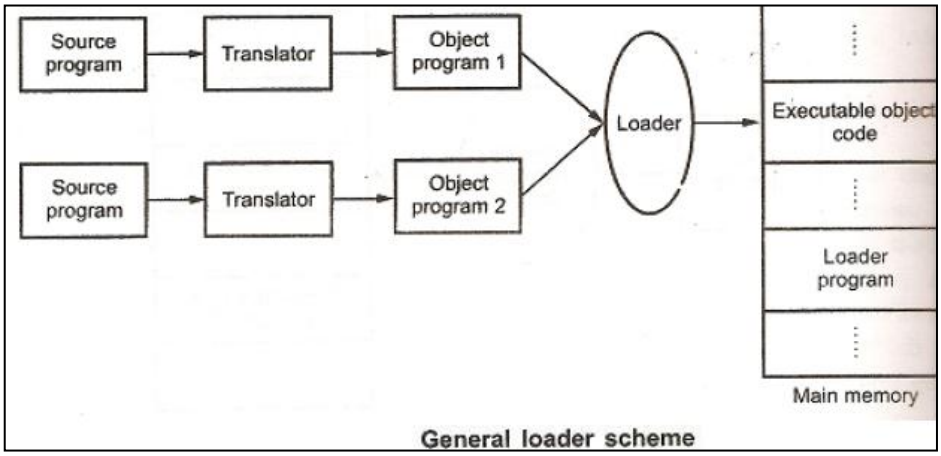


Compile and go loading scheme

2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

Advantages:

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.
• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.
• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

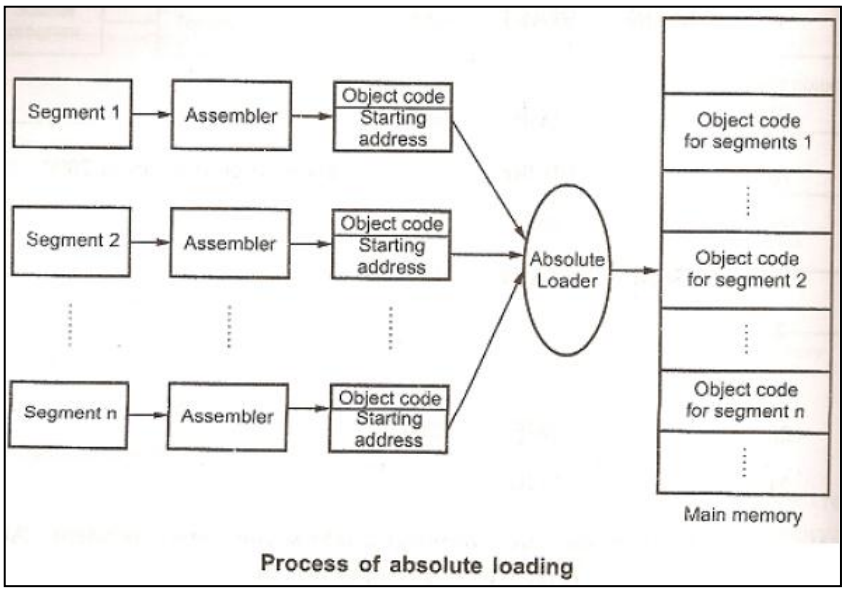


General loader scheme

3) **Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next .modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

```
Line number
1          MAIN      START      1000
.                    .
.                    .
.                    .
15                   JMP        5000
16                   STORE      ;instruction at location 2000
                     END
1          SUM       START      5000
2
20                   JMP        2000
21                   END
```

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN. Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependantdresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codesfor the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.
The entire process is modeled in the following figure.



Process of absolute loading

Thus the absolute loader is simple to implement in this scheme

l) Allocation is done by either programmer or assembler
2)Linkingis done by the programmer or assembler
3)Resolutionis done by assembler
4)Simplyloading is done by the loader
As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

1. It is simple to implement
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient.

Disadvantages:

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.

4) Direct Linking Loaders

- It is a general relocatable loader, and is perhaps the most popular loading scheme presently used.
-
- There are four sections of the object deck for a direct linking loader

  1. External symbol dictionary card (ESD)
  2. Text (TXT)
  3. Relocation and linkage Directory card (RLD)
  4. End card (END)

- **The ESD card** contain the information necessary to build the external symbol. The external symbol are symbols that can be referred beyond the subroutine level. The normal label in the source program are used only by the assembler.

- **The TXT card** contain the blocks of data and the relative address at which data is to be placed. Once the loader has decided where to load the program, it adds the Program Load Address (PLA) to relative address. The data on the TXT card may be instruction, non related data or initial values of address constants.

- **The RLD cards** contain the following information
  1. The location and length of each address constant that needs to be changed for relocation or linking.
  2. The external symbol by which the address constant should be modified.
  3. The operation to be performed.

- **The END card** specifies the end of the object deck.

Advantages of DLL:
The DLL has the advantage of allowing the programmer multiple procedure segment and multiple data segment and of giving him complete freedom in referring data or instruction contained in other segment. This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translation of programs.

6. Attempt any TWO of the following: MARKS 16

a) Explain Bindes and dynamic Bindes.

***[Note: description of statics binder and dynamic binder shall be consider]*

(4 Marks for each)

Ans:

Static binders:

- In static binders a specific core allocation of a program is performed at the time that the subroutines are bound together.
- Since this kind of module looks like an actual 'snapshot' or 'image' of a section of core, it is also called as 'core image module' and the corresponding binder is also called a core image builder.
- It does not keep track of relocation information
- The module loader performs allocation and loading.
- Relatively simple and fast

Dynamic binders:

- The dynamic binders is also called as 'linkage editor'.
- It keep track of the relocation information so that the resulting load module can be further relocated and loaded anywhere in the core.
- In this case the module loader must perform additional allocation and relocation as well as loading.
- It does not have to worry about the complex problems of linking.

In both cases, a program that is to be used repeatedly need only be bound once and then can be loaded whenever required.

b) Give the format of database used in loader.

(4 Marks for each)

Ans:

- **The ESD card** contain the information necessary to build the external symbol. The external symbol are symbols that can be referred beyond the subroutine level. The normal label in the source program are used only by the assembler.

ESD card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters ESD
5-14	Blank
15-16	ESD identifier (ID) for program name (SD) external symbol(ER) or blank for entry (LD)
17-24	Name, padded with blanks
25	ESD type code (TYPE)
26-28	Relative address or blank
29	Blank
30-32	Length of program otherwise blank
33-72	Blank
73-80	Card sequence number

- **The TXT card** contain the blocks of data and the relative address at which data is to be placed. Once the loader has decided where to load the program, it adds the Program Load Address (PLA) to relative address. The data on the TXT card may be instruction, non related data or initial values of address constants.

TXT card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte
9-10	Blanks
11-12	Byte Count (BC) = number of bytes of information in cc. 17-72
13-16	Blank
17-72	From 1 to 56 data bytes
73-80	Card sequence number

- **The RLD cards** contain the following information

1. The location and length of each address constant that needs to be changed for relocation or linking.
2. The external symbol by which the address constant should be modified.
3. The operation to be performed.

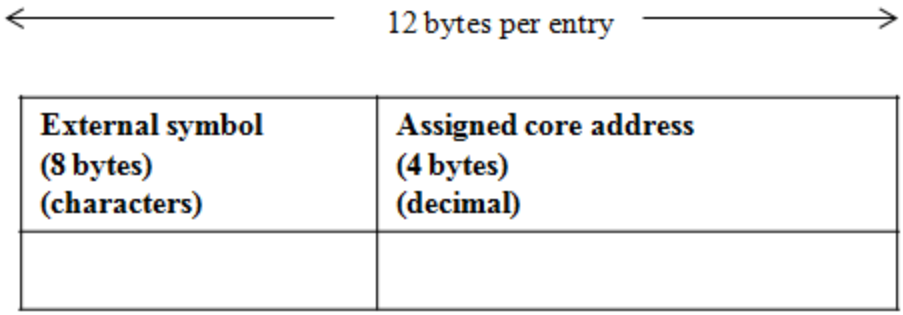
RLD card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	Relative address of first data byte
21	Blanks
22-24	Byte Count (BC) = number of bytes of information in cc. 17-72
25-72	Blank
73-80	Card sequence number

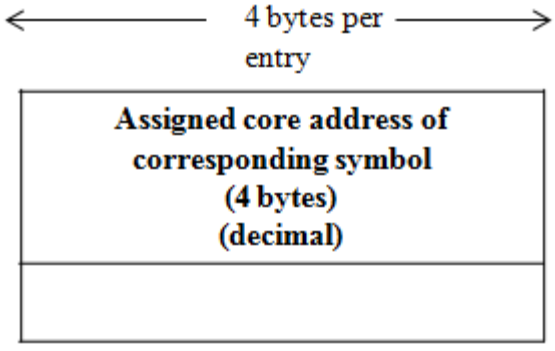
- **The END card** specifies the end of the object deck.
  END card format:

Columns	Contents
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program
9-72	Blanks
73-80	Card sequence number

- **GEST** specifies the Global External Symbol Table format.



- **The LESA** specifies the local External Symbol Array.



c) Explain address calculation sort assume suitable data and apply on it.

(4 Marks for explanation, 4 Marks for example)

Ans:

- This can be one of the fastest types of sorts if enough storage space is available.
- The sorting is done by transforming the key into an address in the table that "represents" the key.
- For example if the key were four characters long, one method of calculating the appropriate table address would be to divide the key by the table length in items, multiply by the length is a power of 2, then the division reduces to a shift. This sort would take only N* (time to calculate address) if it were known that no two keys would be assigned the same address.
- However, in general. This is not the case and several keys will be reduced to the same address.
- Therefore, before putting an item at the calculated address it is necessary to first check whether that location is already occupied. If so, the item is compared with the one that is already there, and a linear search in the correct direction is performed to find the correct place for the new item.
- If we are lucky, there will be n empty space in which to put the item in order. Otherwise, it will be necessary to move some previous entries to make room.

Example:

- Following table shows example of address calculation sort

Data number =	1	2	3	4	5	6	7	8	9	10	11	12
Data =	19	13	05	27	01	26	31	16	02	09	11	21
Calculated address =	6	4	1	9	0	8	10	5	0	3	3	7
Table =												
0	--	--	--	--	01	01	01	01	*01	01	01	01
1	--	--	05	05	05	05	05	05	↓02	02	02	02
2	--	--	--	--	--	--	--	--	↓05	*05	05	05
3	--	--	--	--	--	--	--	--	--	09	*09	09
4	--	13	13	13	13	13	13	13	13	13	↓11	11
5	--	--	--	--	--	--	--	16	16	16	↓13	13
6	19	19	19	19	19	19	19	19	19	19	↓16	16
7	--	--	--	--	--	--	--	--	--	--	↓19	*19
8	--	--	--	--	--	26	26	26	26	26	26	↓21
9	--	--	--	27	27	27	27	27	27	27	27	26
10	--	--	--	--	--	--	31	31	31	31	31	27
11	--	--	--	--	--	--	--	--	--	--	--	↓31

Figure: Example of address calculation sort

- The table is of size 12; since it is known that maximum key is less than 36, the address transformation is to divide the key by 3 and take the integer part. A '*' indicates a conflict between keys, and the arrow indicates when a move is necessary and in which direction. The associated addresses calculated are given in the second row.