CMR INSTITUTE OF TECHNOLOGY	USN	

Internal Assesment Test – II Answer Key

Subject : System Software				Code : 16MCA25
Date : 10.05.2017	Duration : 90 mins	Max Marks : 50	Sem : II	Branch : MCA

		Marks	OBE	
	Answer Any FIVE FULL Questions		CO	RBT
1(a)	**What are the Basic Functions of Loader?**	[4]	CO3	L1

A loader is a system program that performs the loading function. The most fundamental function of loader is to brings object program into memory and starts its execution.

1) Design of Absolute Loader:
   The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

```
Begin
read Header record
verify program name and length
read first Text record
while record type is <> 'E' do
 begin
 {if object code is in character form, convert into internal  representation}
move object code to specified location in memory
read next object program record
end
jump to address specified in End record
end
```

2) Simple Bootstrap Loader
   When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

(b)	**Explain relocation with the Bit Mask.**	[6]	CO3	L2

If a machine primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using relocation bit

Each instruction is associated with one relocation bit. It Indicates that the corresponding word should be modified or not.

0: no modification is needed

1: modification is needed

This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record:

col 1: T

col 2-7: starting address

col 8-9: length (byte)

col 10-12: relocation bits

col 13-72: object code

These relocation bits in a Text record are gathered into bit masks.

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments.

E.g.   FFC=111111111100

E00=111000000000

```
000000 00107A
TΛ000000Λ1EΛFFCΛ140033Λ481039Λ000036Λ280030Λ300015Λ…Λ3C0003
TΛ00001EΛ15ΛE00Λ0C0036Λ481061Λ080033Λ4C0000Λ…Λ000003Λ000000
TΛ001039Λ1EΛFFCΛ040030Λ000030Λ…Λ30103FΛD8105DΛ280030Λ...
TΛ001057Λ0AΛ 800Λ100036Λ4C0000ΛF1Λ001000
TΛ001061Λ19ΛFE0Λ040030ΛE01079Λ…Λ508039ΛDC1079Λ2C0036Λ...
EΛ000000
```

2	**Write short note on**	[10]	CO3	L1

i) Automatic Library Search

ii) Loader options

Automatic Library Search

This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.

The routines are automatically retrieved from a library as they are needed during linking.

This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded.

The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program.

Loader Options

Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and an be specified using loader control statements in the source program. Here are the some examples of how option can be specified. INCLUDE program-name (library-name) - read the designated object program from a library DELETE csect-name – delete the named control section from the set pf programs being loaded CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

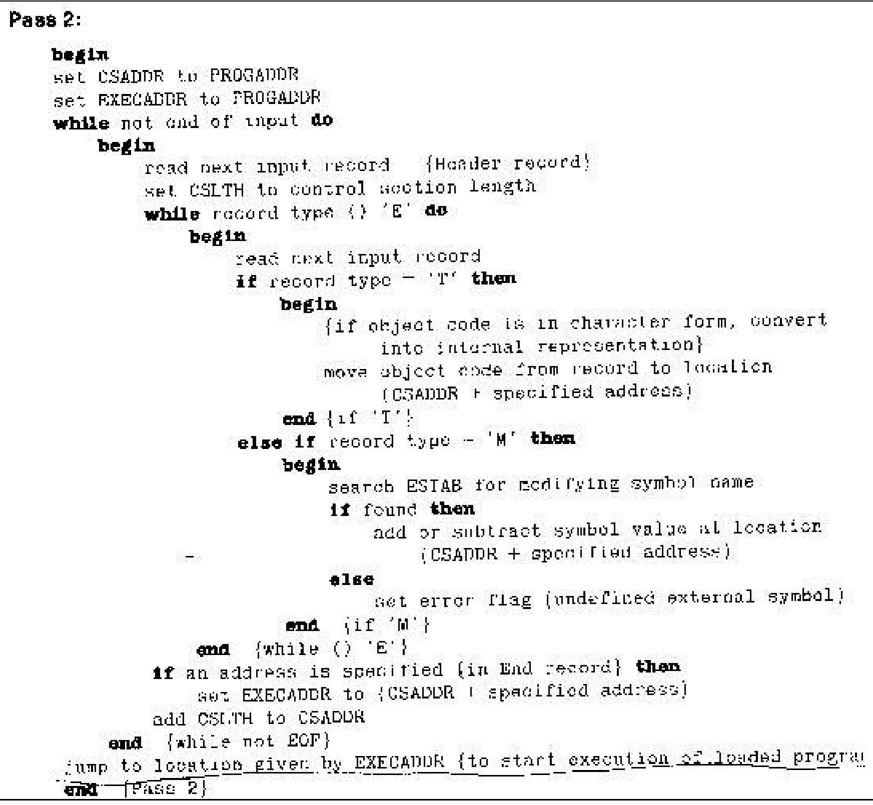
LIBRARY MYLIB – search MYLIB library before standard libraries NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

```
LIBRARY UTLIB

 INCLUDE READ (UTLIB)

INCLUDE WRITE (UTLIB)

DELETE RDREC,WRREC

CHANGE RDREC, READ

CHANGE WRREC, WRITE
```

	NOCALL SQRT, PLOT The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ, similarly references to WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.			
3	**Write the Algorithm for pass-1 and pass-2 of Linking Loader**	[10]	CO3	L2

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type () 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while () 'E'}
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```



Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record   {Header record}
        set CSLTH to control section length
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        {if object code is in character form, convert
                            into internal representation}
                        move object code from record to location
                            (CSADDR + specified address)
                    end {if 'T'}
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end  {if 'M'}
            end  {while <> 'E'}
        if an address is specified {in End record} then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded progra
end  {Pass 2}
```

4(a)	**Discuss Macro Definition and Expansion with suitable Example**	[5]	CO3	L1

<u>Macro Definition and Expansion:</u> The figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.

M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

Source

```
M1      MACRO    &D1, &D2
        STA      &D1
        STB      &D2
        MEND
  .
M1 DATA1, DATA2
  .
M1 DATA4, DATA3
```

Expanded source

```
  .
  .
  .
        STA    DATA1
        STB    DATA2
  .
        STA    DATA4
        STB    DATA3
  .
```

	The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation. The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.			
(b)	**Describe MASM Macro Processor** MASM The macro processor of MASM is integrated with Pass 1 of the assembler MASM generates the unique names of local labels in the form ??n, where n is a hexadecimal number in the range 0000 to FFFF .ERR: signals to MASM that an error has been detected EXITM: directs MASM to terminate the expansion of the macro &: is a concatenation operator ;; is a macro comment, serves only as documentation for the macro definition ; is an ordinary assembler language comment, included as part of the macro expansion IRP: sets the macro-time variable to a sequence of values specified in <…> The statements between the TRP and the matching ENDM are generated once for each value of the variable	[5]	CO3	L1
5	**Explain the different data structures used by Macro Processor** The data structures required are: DEFTAB (Definition Table) • Stores the macro definition including macro prototype and macro body • Comment lines are omitted. • References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.	[10]	CO3	L2

NAMTAB (Name Table)

- Stores macro names
- Serves as an index to DEFTAB
- Pointers to the beginning and the end of the macro definition (DEFTAB)

ARGTAB (Argument Table)

- Stores the arguments according to their positions in the argument list.
- As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.
- The figure below shows the different data structures described and their relationship.

6 **Explain the Following :** [10] CO3 L2

i) Generation of unique labels
ii) Concatenation of Macro Parameters

Generation of Unique Labels

- It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler.
- This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion.
- During macro expansion each $ will be replaced with $XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion. For example,
- XX = AA, AB, AC… This allows 1296 macro expansions in a single program.

Concatenation of Macro Parameters

There are applications of macro processors that are not related to assemblers or assembler programming.
Conditional assembly depends on parameters provides

```
MACRO &COND
   ……..
   IF (&COND NE „‟)
         part I
   ELSE
         part II
  ENDIF
  ………
ENDM
```

Part I is expanded if condition part is true, otherwise part II is expanded. Compare operators: NE, EQ, LE, GT.

Macro-Time Variables:

Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion. Any symbol that begins with symbol & and not a macro instruction parameter is considered as *macro-time variable.* All such variables are initialized to zero.

If the value of this expression TRUE,

- The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement.
- If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF.
- Once it reaches ENDIF, it resumes expanding the macro in the usual way.

If the value of the expression is FALSE,

- The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement.
- The macro processor then resumes normal macro expansion.

WHILE-ENDW structure

- When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- TRUE −The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.
- When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.
- FALSE − The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.

7(a) **Write short note on** [8] CO4 L1

i) Linkage Editor

The figure below shows the processing of an object program using Linkage editor. A linkage editor produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution.

The linked program produced is generally in a form that is suitable for processing by a relocating loader. Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known.

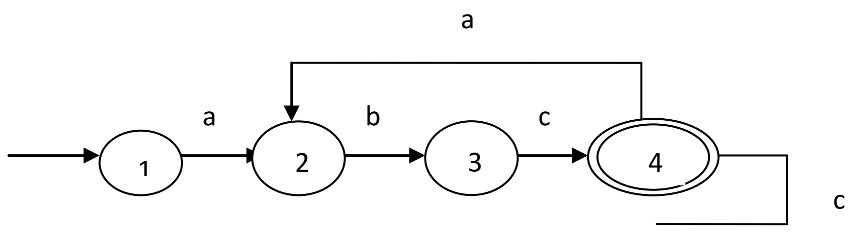
New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together.

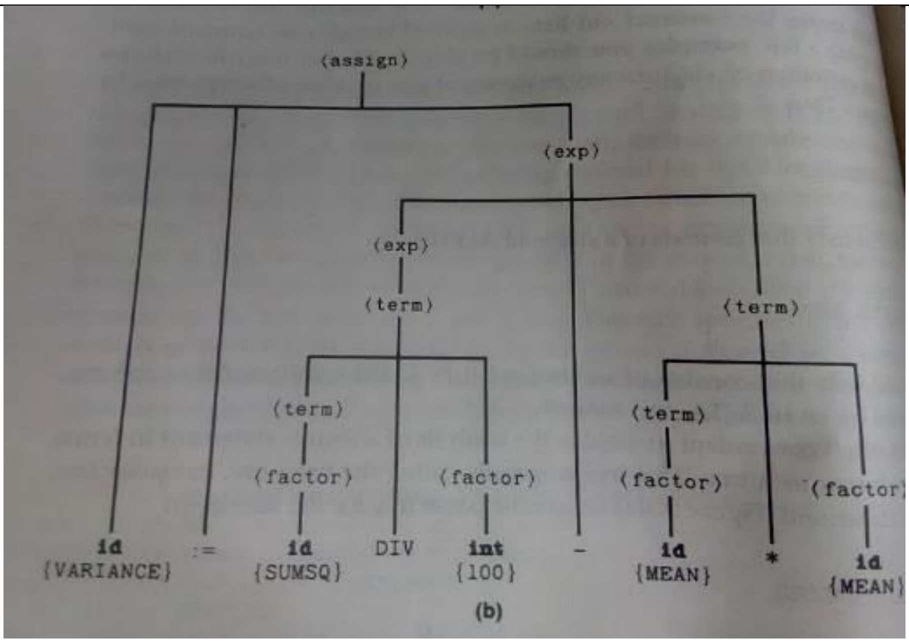
Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space

ii) Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

	Begin X=0x80 (the address of the next memory location to be loaded **Loop** A←GETC (and convert it from the ASCII character code to the value of the hexadecimal digit) save the value in the high-order 4 bits of S A←GETC combine the value to form one byte A← (A+S) store the value (in A) to the address in register X X←X+1 **End**			
(b)	**What is Recursive-Descent Parsing** Recursive-Descent parsing is a top down parsing method. For each non-terminal, there is a procedure which Begins from the current token, search the following tokens, and try to recognize the rule associated with the non-terminal. May call other procedures or even itself for non-terminals included in the rule of this non-terminal. When a match is recognized, the procedure returns an indication of success, otherwise, error.	[2]	CO4	L1
8(a)	**Consider the following finite automata and check whether the following strings are recognized or not** **i)abc ii) abccabc ii) ac iv)abcabc v)abcac**  i)abc - Recognized ii) abccabc - Recognized ii) ac - Not Recognized iv)abcabc - Recognized v)abcac- - Not Recognized	[4]	CO4	L3
(b)	**Construct Parsing Tree for following PASCAL statement** **<assign> :: = id : = <exp>**	[4]	CO4	L3

	 ⟨assign⟩ ⟨exp⟩ ⟨exp⟩ ⟨term⟩ ⟨term⟩ ⟨term⟩ ⟨factor⟩ ⟨factor⟩ ⟨term⟩ ⟨factor⟩ ⟨factor⟩ id {VARIANCE} := id {SUMSQ} DIV int {100} - id {MEAN} * id {MEAN} (b)			
(c)	**What are the Basic Functions of complier?** Basic functions of Compiler are Scanning, parsing, and (object) code generation. 1) Lexical analysis Scan the program to be compiled and recognize the tokens (from string of characters). 2) Syntactic analysis The source statements written by programmers are recognized as language constructs described by the grammar. This process is achieved by building the parse tree for the statements being translated. 3) Code Generation Most compilers create machine-language programs directly instead of producing a symbolic program for later translation by an assembler.	[2]	CO4	L1