USN										

Internal Assessment Test 2Answer Key – April. 2018

Sub:	System Software					Sub Code:	17MCA25	Branch:	MCA
Date:	19/04/2018	Duration:	90 min’s	Max Marks:	50	Sem	II		OBE

1 (a) Discuss literals Symbol-defining Statements and Expressions with suitable examples? [06]

1) Literals

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

```
45          001A          ENDFIL          LDA          =C‟EOF‟
```

All the literal operands used in a program are gathered together into one or more literal pools. This is usually placed at the end of the program.

The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program.

An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program.

2) Symbol-Defining Statements

1) EQU

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this is EQU (Equate). The general form of the statement is

```
Symbol      EQU          value
```

This statement defines the given symbol (i.e., enter it into SYMTAB) and assigning to it the value specified.

3) Expressions

- Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address.
- Assemblers generally allow arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /.
- Individual terms in the expression may be constants, user-defined symbols, or special terms.
- The common special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement BUFFEND EQU *

1 (b) Explain Multi-pass Assembler [04]

Consider the following example

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

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined. As a result, ALPHA cannot be evaluated during second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definition.

Prohibiting forward references in symbol definition is not a serious inconvenience. Forward references tend to create difficulty for a person reading the program as well as for the assembler.

The general solution is multi pass assembler that can make has many passes are needed to process the definition of symbols.
It is not necessary for such an assembler to make more than two passes over the entire program. Instead, the portions of the program that involve forward references in symbol definition are saved during pass. Additional passes through these stored definitions are made as the assembly progresses.
There are several ways to accomplish the task outlined above.
- Store those symbol definitions that involve forward references in the symbol table.
- This table also indicates which symbols are dependent on the values of others, to facilitate symbol evaluation.

2 (a) Discuss Program Block and Control section & program linking with example for each. [04]
1)Program block
Program block refers to segment of code that are rearranged within a single object program unit and control section to refer to segments that are translated into independent object program units.
Assembler Directive USE indicate which portion of the source program belong to various blocks
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. Assemblers rearrange these segments to gather together the pieces of each block and assign address.

Pass1
A separate location counter for each program block is maintained. Save and restore LOCCTR when switching between blocks. At the beginning of a block, LOCCTR is set to 0. Assign each label an address relative to the start of the block. Store the block name or number in the SYMTAB along with the assigned relative address of the label Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1 Assign to each block a starting address in the object program by concatenating the program blocks in

a particular order
Pass2 : Calculate the address for each symbol relative to the start of the object program by adding: The location of the symbol relative to the start of its block. The starting address of this block

2) Control Sections and program linking
• A control section is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others.
• Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.
• Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called external references.
• The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive – assembler directive: CSECT The syntax secname CSECT
• separate location counter is maintained for each control section Control sections differ from program blocks in that they are handled separately by the assembler.

2 (b) Explain design of one pass assembler [04]

The main problem in designing the assembler using single pass was to resolve forward references.
We can avoid to some extent the forward references by:

• Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.

Unfortunately, forward reference to labels on the instructions cannot be eliminated as easily. Assembler provides some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:
1) One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
2) The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler

• Load-and-go assembler generates their object code in memory for immediate execution.
• No object program is written out, no loader is needed.
• It is useful in a system with frequent program development and testing. The efficiency of the assembly process is an important consideration.
• Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

3 Write short note on

i) MASM Assembler

ii) SPARC Assembler

i)MASM Assembler

The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows. Beginning with MASM 8.0 there are two versions of the assembler - one for 16-bit and 32-bit assembly sources, and another (ML64) for 64-bit sources only.

MASM is maintained by Microsoft, but since version 6.12 has not been sold as a separate product, it is instead supplied with various Microsoft SDKs and C compilers. Recent versions of MASM are included with Microsoft Visual Studio.

ii) SPARC Assembler

Sun OS SPARC assembler
Assembler language program is dived into units called sections.
Predefine section names

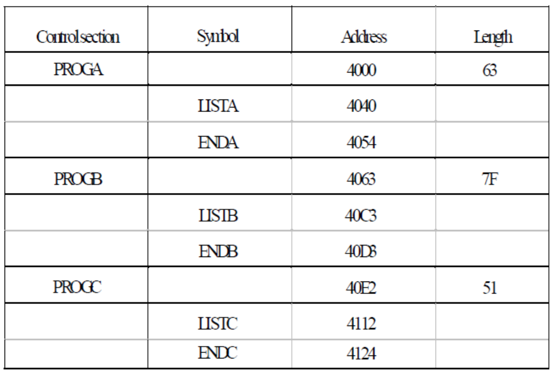
.TEXT – Executable instruction
.DATA- Initialized read/write data
.RODATA- Read only data
.BSS – uninitialized data areas

Programmer can switch between sections at any times using assembler directives. Separate location counter for each section.
When assembler switches to new section it also switches to location counter associated with that

4 (a) Describe all the data structures of linking loader [06]

1) External Symbol Table (ESTAB)

This table is analogous to SYMTAB ESTAB is used to stores the name and address of each external symbol in the set of control section being loaded. The table also often indicates in which control section the symbol is defined. A Hashed organization is typically used for this table.



Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

2) Program Load Address (PROGADDR)
PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the operating system.

3) Control Section Address (CSADDR)
CSADDR is the starting address assigned to the control section currently being scanned by the loader. This address is added to all relative address within the control section to convert them to actual address.

4 (b) Write short note on Bootstrap Loader [04]
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
```

5 Write the Algorithm for pass-1 and pass-2 of Linking Loader [10]

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}
```

6 Explain following loader design options [10]

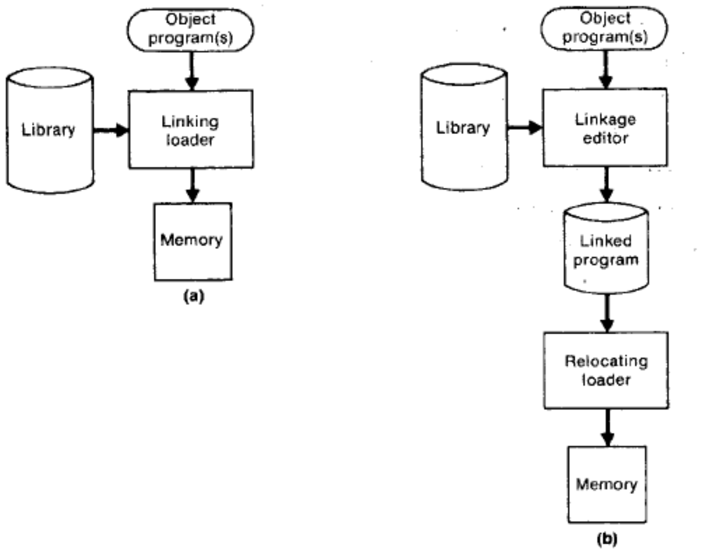
i) Linkage Editors

ii) Dynamic Linking

1. Linkage Editor

The figure below shows the processing of an object program using Linkage editor.

FIGURE 3.17 Processing of an object program using (a) linking loader and (b) 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.

Linkage editor can perform many useful functions besides simply preparing an object program for execution.

 produce core image if actual address is known in advance

 improve a subroutine (PROJECT) of a program (PLANNER) without going back to the original versions of all of the other subroutines

INCLUDE PLANNER(PROGLIB) DELETE PROJECT {delete from existing PLANNER} INCLUDE PROJECT(NEWLIB) {include new version} REPLACE PLANNER(PROGLIB) external references are retained in the linked 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. Compared to linking loader, Linkage editors in general tend to offer more flexibility and control, with a corresponding increase in complexity and overhead

2. Dynamic Linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called. This type of functions is usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs.

Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request. Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the OS. The OS examines its internal tables to determine whether or not the routine is already loaded. Control is then passed from the OS to routine being called. When the called subroutine completes its processing, it returns to its caller. OS then returns control to the program that issued the request.

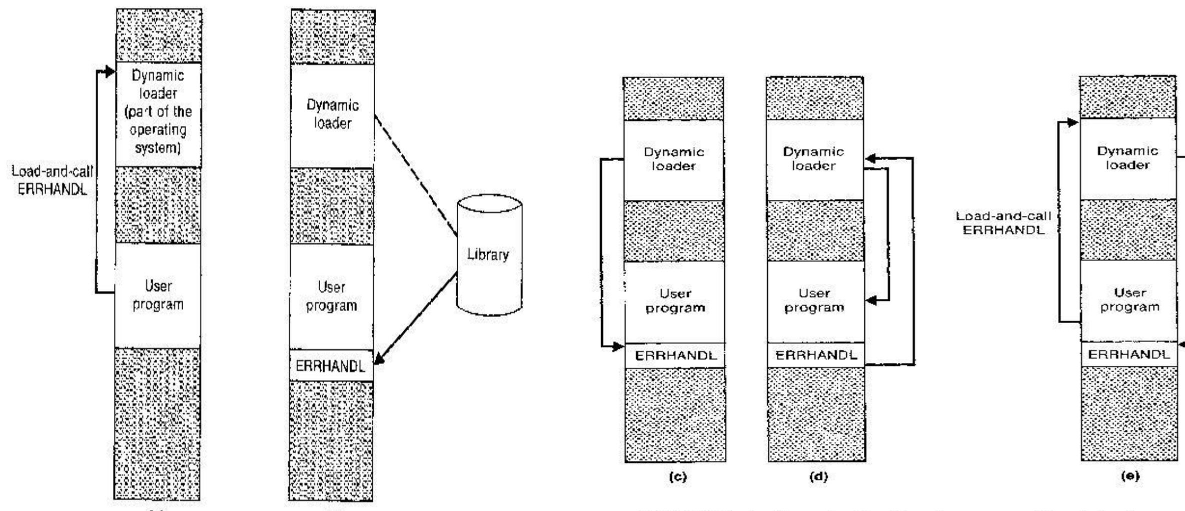


FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.

7 What are the Basic Functions of Loader? Explain design of absolute loader and Bootstrap loader [10]

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. Translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. The translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader.

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.

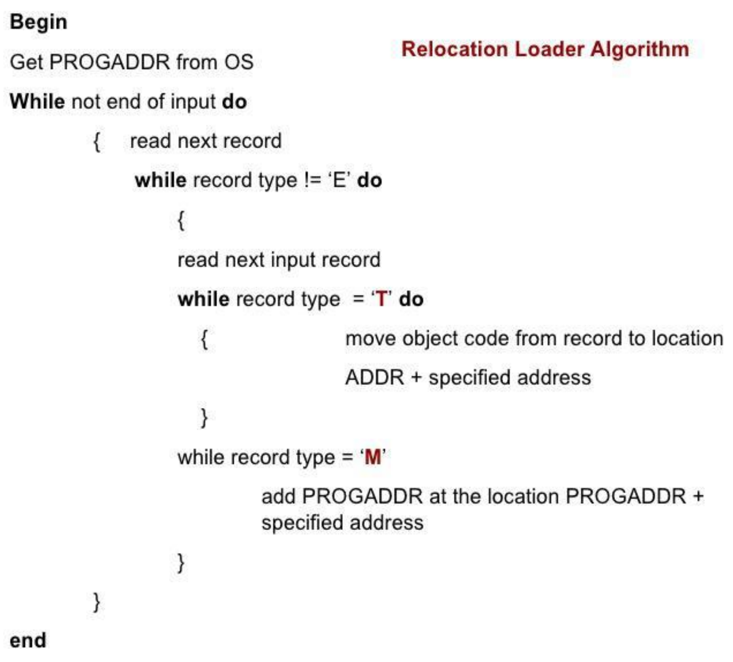
```
Begin
read Header record
verify program name and length
read first Text record
while record type is <> 'E' do
 begin
 {if object code is in character form, convert into internal  representation}   move object code to specified location in memory
read next object program record
end
jump to address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

8 Discuss Relocation using Modification record and Bit Mask (Relocation Bit) [10]

Loaders that allow for program relocation are called relocating loaders or relative loaders There are two methods for specifying relocation as part of the object program.

i) Modification record

A Modification record is used to describe each part of the object code that must be when program is relocated. There is one modification record for each value that must be changed during relocation. Each modification record specifies the starting address and length of the field whose value is to be altered. It then describes modification to be performed.



```
Begin
Get PROGADDR from OS                    Relocation Loader Algorithm
While not end of input do
        {   read next record
            while record type != 'E' do
               {
               read next input record
               while record type  = 'T' do
                  {            move object code from record to location
                               ADDR + specified address
                  }
               while record type = 'M'
                        add PROGADDR at the location PROGADDR +
                        specified address
               }
        }
end
```

ii) Relocation bit (Bit Mask)

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

9 Write short note on [10]

i) Automatic Library Search

ii) Loader options

i) 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.

ii) 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

10 (a) Write a short note on Ms-DOS Linker [05]

This explains some of the features of Microsoft MS-DOS linker, which is a linker for Pentium and other x86 systems. Most MS-DOS compilers and assemblers (MASM) produce object modules, and they are stored in .OBJ files. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file; this file is later executed for results.

The following table illustrates the typical MS-DOS object module

Record Types	Description
THEADR	Translator Header
TYPDEF,PUBDEF, EXTDEF	External symbols and references
LNAMES, SEGDEF, GRPDEF	Segment definition and grouping
LEDATA, LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information

MODEND End of object module
THEADR specifies the name of the object module. MODEND specifies the end of the module. PUBDEF contains list of the external symbols (called public names). EXTDEF contains list of external symbols referred in this module, but defined elsewhere. TYPDEF the data types are defined here. SEGDEF describes segments in the object module (includes name, length, and alignment). GRPDEF includes how segments are combined into groups. LNAMES contains all segment and class names. LEDATA contains translated instructions and data. LIDATA has above in repeating pattern. Finally, FIXUPP is used to resolve external references.

10 (b) Write a short note on Cray MPP Linker [05]
Cray MPP (massively parallel processing) Linker is developed for Cray T3E systems. A T3E system contains large number of parallel processing elements (PEs) – Each PE has local memory and has access to remote memory (memory of other PEs). The processing is divided among PEs - contains shared data and private data.

The loaded program gets copy of the executable code, its private data and its portion of the shared data. The MPP linker organizes blocks containing executable code, private data and shared data. The linker then writes an executable file that contains the relocated and linked blocks. The executable file also specifies the number of PEs required and other control information. The linker can create an executable file that is targeted for a fixed number of PEs, or one that allows the partition size to be chosen at run time. Latter type is called plastic executable.