CMR
INSTITUTE OF
TECHNOLOGY

USN

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 2 Answer Key– Nov. 2020								
Sub:	System Software				Sub Code:	18MCA34	Branch:	MCA
Date:	05/11/2020	Duration:	90 min's	Max Marks:	50	Sem	III	

Q1) Explain following machine independent features of assembler [10]

i) Program block

ii) Control Sections and Program linking

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.

Q2) Write One pass Assembler algorithm. [10]

```
1 while opcode != 'End' do
2 begin
3         if there is no conmment line then
4         begin
5                 if there is a symbol in the LABEL field then
6                 begin
7                         search SYMTAB for LABEL
8                         if found then
9                         begin
10                                if <symbol value> as null
11                                set <symbol value> as LOCCTR and search
12                                        the linked list with corresponding
13                                        operand
14                                PTR addresses and generate operand
15                                        addresses as corresponding symbol
16                                        values
17                                set symbol value as LOCCTR in symbol table
18                                        and delete  the linked list
19                        end
20                        else
21                                insert (LABEL,LOCCTR) into symtab
22
23                end
24                search OPTAB for OPCODE
25                if found then
26                begin
27                        search SYMTAB for OPERAND addresses
28                        if found then
29                                if symbol value not equal to null then
30                                        store symbol value as OPERAND address
31                                else
32                                        insert at the end of the linked list
33                                        with a node with address as LOCCTR
34                        else
35                                insert (symbol name,null)
36                        LOCCTR+=3
37                end
38                else if OPCODE='WORD' then
39                        add 3 to LOCCTR and convert comment to object code
40                else if OPCODE='RESW' then
41                        add 3 #[OPERAND] to LOCCTR
42                else if OPCODE='RESB' then
43                        add #[OPERAND] to LOCCTR
44                else if OPCODE='Byte' then
45                begin
46                        find the length of constant in bytes
47                        add length to LOCCTR
48                        convert constant to object code
49                end
50                if object code will not fit into current text record then
51                begin
52                        write text record to object program initialize new Text record
53                end
54                add object code to Text record
55        end
56        write listing line
57        read next input line
58 end
59 write last Text recordto object program
60 write End record to object program
61 write last listing line
62
```

Q3) Explain following machine dependent features of assembler [10]

i) **Instruction Formats and Addressing modes**

ii) **Program Relocation**

1) Instruction Formats and Addressing modes

The instruction formats depend on the memory organization and the size of the memory.
In SIC machine the memory size is 215 bytes. Accordingly it supports only one instruction format.
Whereas the memory of a SIC/XE machine is 220 bytes (1 MB).
This supports four different types of instruction types, they are:

- 1 byte instruction
- 2 byte instruction
- 3 byte instruction
- 4 byte instruction

Instructions can be

- Instructions involving register to register („register to register‟ instructions are faster than „register to memory‟ instruction because they do not require memory reference)
- Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
- Extended instruction format

Addressing Modes are:

- Index Addressing(SIC):
    Syntax Opcode m, x
    Example STCH BUFFER, X
- Indirect Addressing: prefixed with @
    Syntax Opcode @m
    Example J @RETADR
- Immediate addressing: prefixed with#
    Syntax Opcode #c
    Example LDA #3
- PC-relative:
    Syntax Opcode m
- Base relative:
    Syntax Opcode m

Instruction involving Register-Register:
During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes.
During pass2, these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.
Instruction involving Register-to-memory:
Most of the register-to-memory instructions are assembled using either program-counter relative or base relative addressing.
Program-Counter Relative: In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction.

TA = (PC) + displacement value

Base-Relative Addressing Mode: in this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

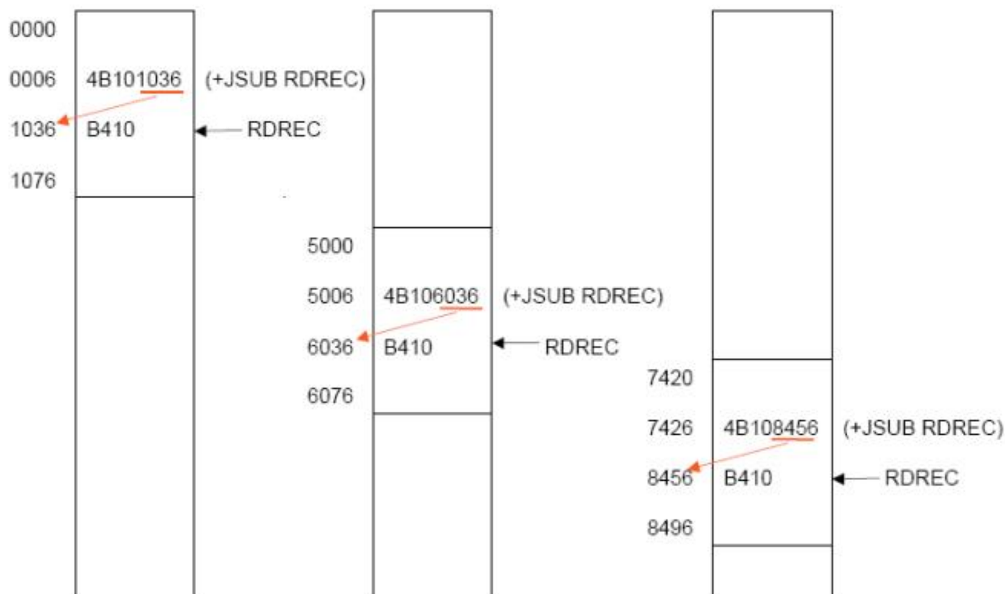
2) Program Relocation.

It is often desirable to have more than one program at a time sharing the memory and other resources of the machine.
In such a situation the actual starting address of the program is not known until the load time. Program in which the address is mentioned during assembling itself. This is called Absolute Assembly or Absolute Program.

Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used by the program. However, the assembler identifies for the loader those parts of the program which need modification.
An object program that has the information necessary to perform this kind of modification is called the relocatable program.

This can be accomplished with a Modification record having following format:
Modification record
Col. 1 M
Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes. 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.



Q4 a) Explain absolute loader with a algorithm [5]

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

Q4 b) Explain bootstrap loader with a algorithm [5]

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

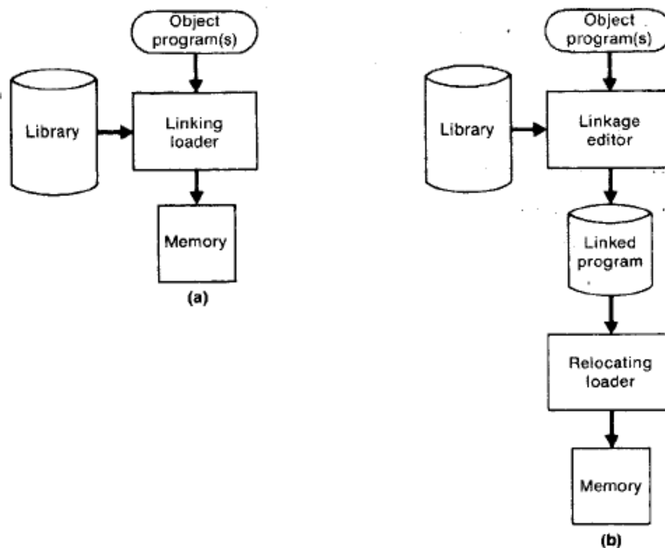
Begin
X=0x80 (the address of the next memory location to be loaded
Loop
A←GETC (and convert it from the ASCII character code to the value of the hexadecimal digit) save the value in the high-order 4 bits of S
A←GETC combine the value to form one byte A← (A+S) store the value (in A) to the address in register X
X←X+1
End

Q5) 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.

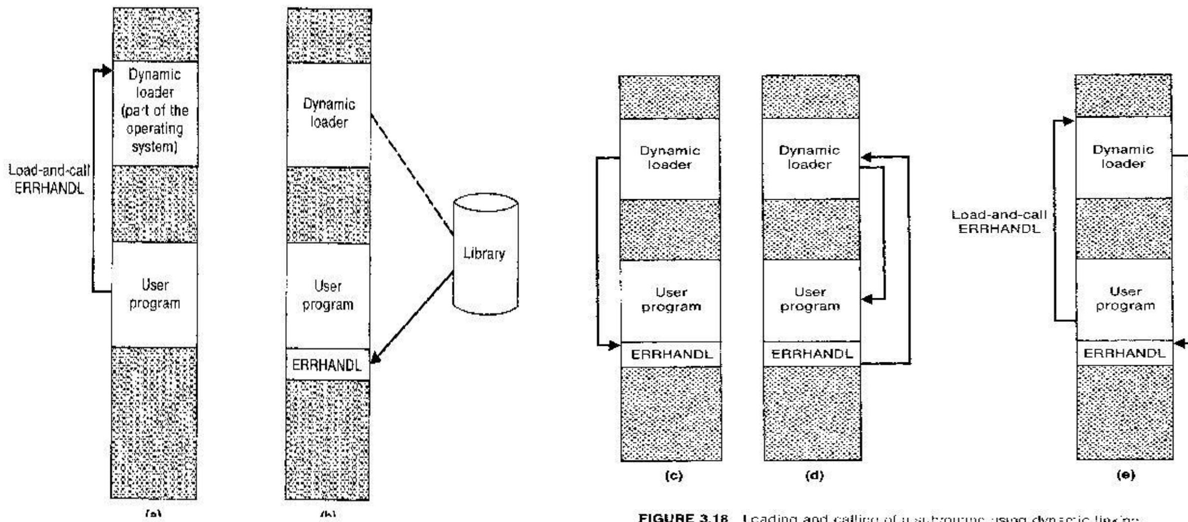


FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.

Q6) Explain Program Linking with neat diagram [10]

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here.

Col. 1 D

Col. 2-7 Name of external symbol defined in this control section

Col. 8-13 Relative address within this control section (hexadecimal)

Col.14-73 Repeat information in

Col. 2-13 for other external symbols

Example records D LISTA 000040 ENDA 000054 D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here.

Col. 1 R

Col. 2-7 Name of external symbol referred to in this control section

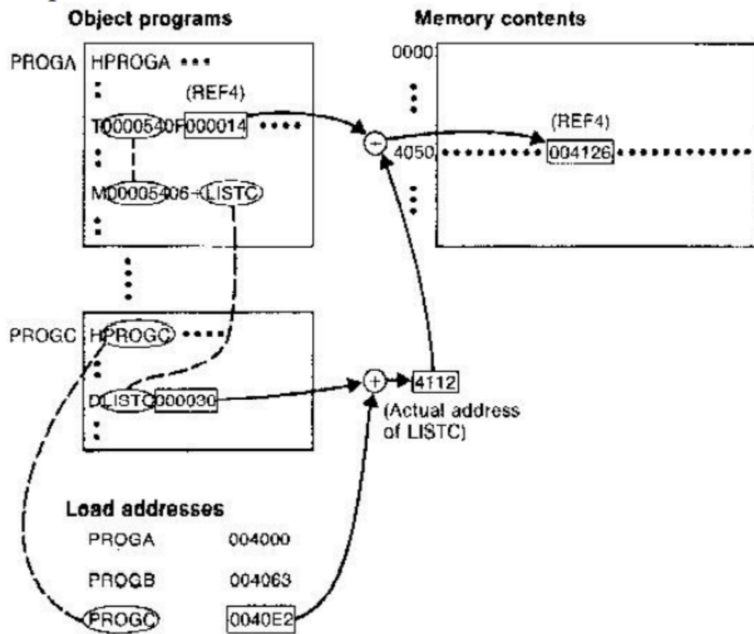
Col. 8-73 Name of other external reference symbols

Example records R LISTB ENDB LISTC ENDC R LISTA ENDA LISTC ENDC R LISTA ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs used to illustrate linking and relocation. The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROG B and PROGC immediately following.

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
3FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4000	....	....	....	....	
4010	....	....	....	....	
4020	03201D77	1040C705	0014....	....	←PROGA
4030	....	....	....	....	
4040	....	....	....	....	
4050	....	00412600	00080040	51000004	
4060	000083..	....	....	....	
4070	....	....	....	....	
4080	....	....	....	....	
4090	....	....	..031040	40772027	←PROGB
40A0	05100014	....	....	....	
40B0	....	....	....	....	
40C0	....	....	....	....	
40D0	00	41260000	08004051	00000400	
40E0	0083....	....	....	....	
40F0	....	....	0310	40407710	
4100	40C70510	0014....	....	....	←PROGC
4110	....	....	....	....	
4120	....	00412600	00080040	51000004	
4130	000083xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.



The initial value from the Text record T0000540F000014FFFFF600003F000014FFFFC0 is 000014.

To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126. That is REF4 in PROGA is ENDA-LISTA+LISTC=4054-4040+4112=4126. Similarly the load address for symbols LISTA: PROGA+0040=4040, LISTB: PROGB+0060=40C3 and LISTC: PROGC+0030=4112 Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

Q7) Explain all machine independent features of loader [10]

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

Q8) Write Pass1 algorithm for 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}
```