CMR
INSTITUTE OF

TECHNOLOGY

Improvement Test 1 – November. 2016

Sub:	SYSTEM SOFTWARE						Code:	10CS52
Date:	18/11/2016	Duration:	90 mins	Max Marks:	50	Sem: 5	Branch:	CSE(B & C sec)

Note: Answer any five full questions. Each question carries 10 marks.

Scheme and Solution

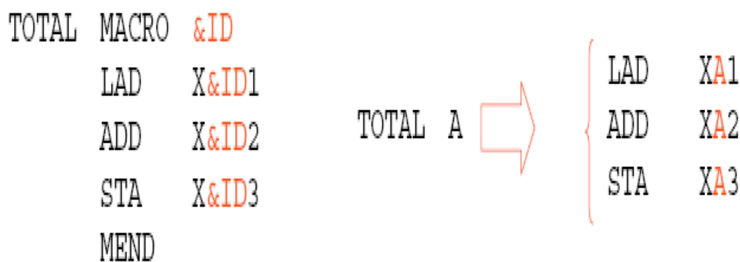
1. Explain any two machine independent macro processor features..

Mentioning 2 design options	1M
Explaining 2 features with example	9M

The design of macro processor doesn"t depend on the architecture of the machine.
We will be studying some extended feature for this macro processor. These features are:
• Concatenation of Macro Parameters
• Generation of unique labels
• Conditional Macro Expansion
• Keyword Macro Parameters

1.Concatenation of unique labels:

• Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction.
• The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, Xb1, etc.).
• Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like
 LDA X&ID



& is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended. • If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous. • Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as

```
1   SUM MACRO   &ID
2       LDA     X&ID→ 1
3       ADD     X&ID→ 2
4       ADD     X&ID→ 3
5       STA     X&ID→ S
6       MEND
```

```
SUM   A                SUM   BETA
 ↓                      ↓
LDA   XA1              LDA   XBEATA1
ADD   XA2              ADD   XBEATA2
ADD   XA3              ADD   XBEATA3
STA   XAS              STA   XBEATAS
```

```
ID123  MACRO  &ID
       LAD    X&ID→1
       ADD    X&ID→2
       STA    X&ID→3
       MEND
```

2.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 atwo-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. The following program shows the macro definition with labels to the instruction.

2. Generate an algorithm for a one pass macro processor. 10M
Writing algorithm -10M

```
begin {macro processor}
        EXPANDINF := FALSE
        while OPCODE ≠ 'END' do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
end {macro processor}


Procedure PROCESSLINE
        begin
          search MAMTAB for OPCODE
          if found then
                EXPAND
          else if OPCODE = 'MACRO' then
                DEFINE
          else write source line to expanded file
        end {PRCOESSOR}

Procedure DEFINE
        begin
            enter macro name into NAMTAB
            enter macro prototype into DEFTAB
            LEVEL   :- 1
            while LEVEL > do
                begin
                    GETLINE
                    if this is not a comment line then
                      begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL +1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL – 1
                      end {if not comment}
                end {while}
            store in NAMTAB pointers to beginning and end of definition
        end {DEFINE}
```

```
Procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        while macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}

Procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

3. Expand the macro call statements for the following macro definition 10M

i) RDBUFF F1,BUFA,RLENG,04,1024

ii) RDBUFF F2,BUFB,RLENG

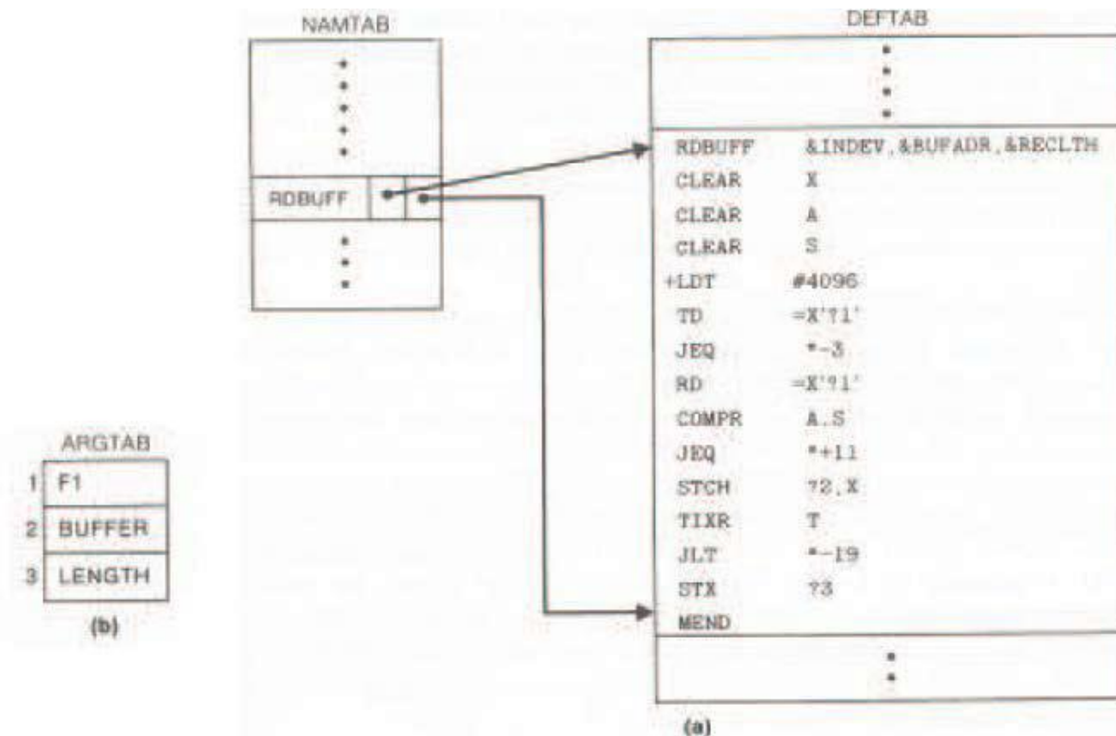
```
RDBUFF MACRO &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
        IF  (&EOR  NE  ‘  ‘ )
&EORCT  SET   1
        ENDIF
        CLEAR  X
        CLEAR  A
        IF  (&EORCT  EQ  1)
        LDCH  =X ‘&EOR’
        RMO  A,S
        ENDIF
        IF  (&MAXLTH  EQ ‘  ‘)
        +LDT  #4096
        ELSE
        +LDT  #&MAXLTH
        ENDIF
$LOOP    TD =X ‘&INDEV’
        JEQ  $LOOP
        RD =X ‘&INDEV’
        STCH  &BUFADR,X
        TIXR   T
        JLT     $LOOP
        STX   &RECLTH
        MEND
```

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. After *macro processing* the expanded file can become the input for the *Assembler*. The *Macro Invocation* statement is considered as comments and the statement generated from expansion is treated exactly as though they had been written directly by the programmer.

The difference between *Macros* and *Subroutines* is that the statement s from the body of the Macro is expanded the number of times the macro invocation is encountered, whereas the statement of the subroutine appears only once no matter how many times the subroutine is called. Macro instructions will be written so that the body of the macro contains no labels.

```
5       COPY    START   0                   COPY FILE FROM INPUT TO OUTPUT
180     FIRST   STL     RETADR              SAVE RETURN ADDRESS
190     .CLOOP  RDBUFF  F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
190a    CLOOP   CLEAR   X                   CLEAR LOOP COUNTER
190b            CLEAR   A
190c            CLEAR   S
190d            +LDT    #4096               SET MAXIMUN RECORD LENGTH
190e            TD      =X'F1'              TEST INPUT DEVICE
190f            JEQ     *-3                 LOOP UNTIL READY
190g            RD      =X'F1'              TEST FOR END OF RECORD
190h            COMPR   A, S                TEST FOR END OF RECORD
190i            JEQ     *+11                EXIT LOOP IF EOR
190j            STCH    BUFFER, X           STORE CHARACTER IN BUFFER
190k            TIXR    T                   LOOP UNLESS MAXIMUN LENGTH
190l            JLT     *-19                HAS BEEN REACHED
190M            STX     LENGTH              SAVE RECORD LENGTH
```



(b)

(a)

4. Explain the general purpose macro processors design options. 10M
   Definition-2M
   Expalnation-8M

Macro Processor Design Options
 Recursive Macro expression
 General-Purpose Macro Processors
 Macro Processing within Language Translators
General Purpose Macro Processor
 Advantages of general-purpose macro processors:
 The programmer does not need to learn about a different macro facility for each compiler or assembler language—the time and expense involved in training are eliminated
 The costs involved in producing a general-purpose macro processor are somewhat greater than those for developing a language-specific processor

 However, this expense does not need to be repeated for each language; the result is substantial overall saving in software development cost
 user to define the specific set of rules to be followed
 Comments should usually be ignored by a macro processor, however, each programming language has its own methods for identifying comments
 Each programming language has different facilities for grouping terms, expressions, or statements—a general-purpose macro processor needs to taking these grouping into account
 Languages differ substantially in their restrictions on the length of identifiers and the rules for the formation of constants
 Programming languages have different basic statement forms—syntax used for macro definitions and macro invocation statements

Macro definition

» header:
– a sequence of keywords and parameter markers (%)
– at least one of the first two tokens in a macro header must be a keyword, not a parameter marker

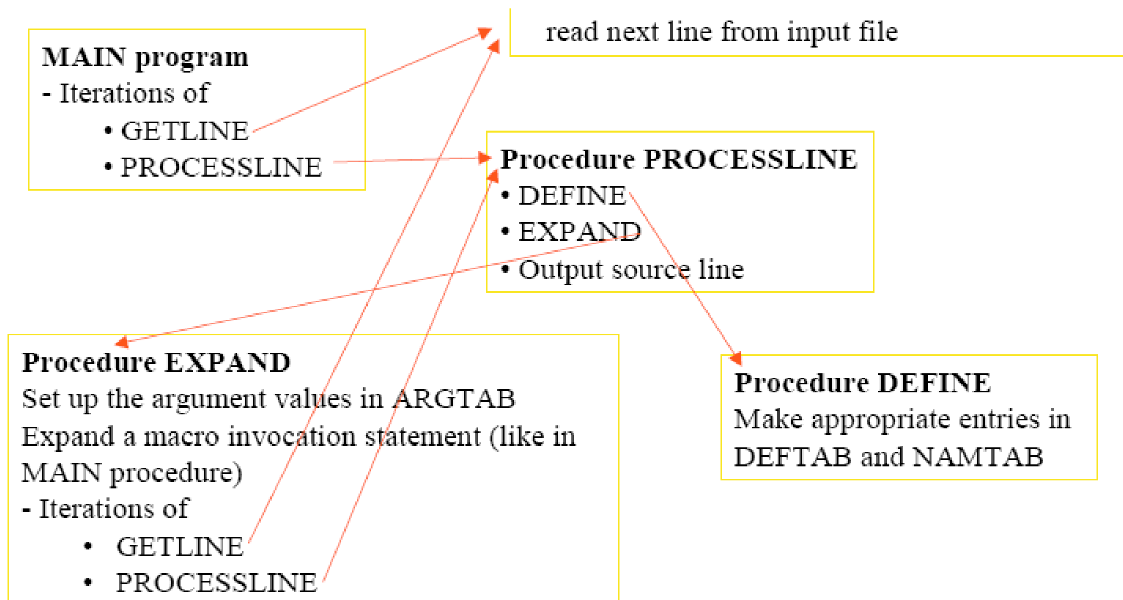
» body:
– the character & identifies a local label
– macro time instruction (.SET, .IF .JUMP, .E)
– macro time variables or labels (.)

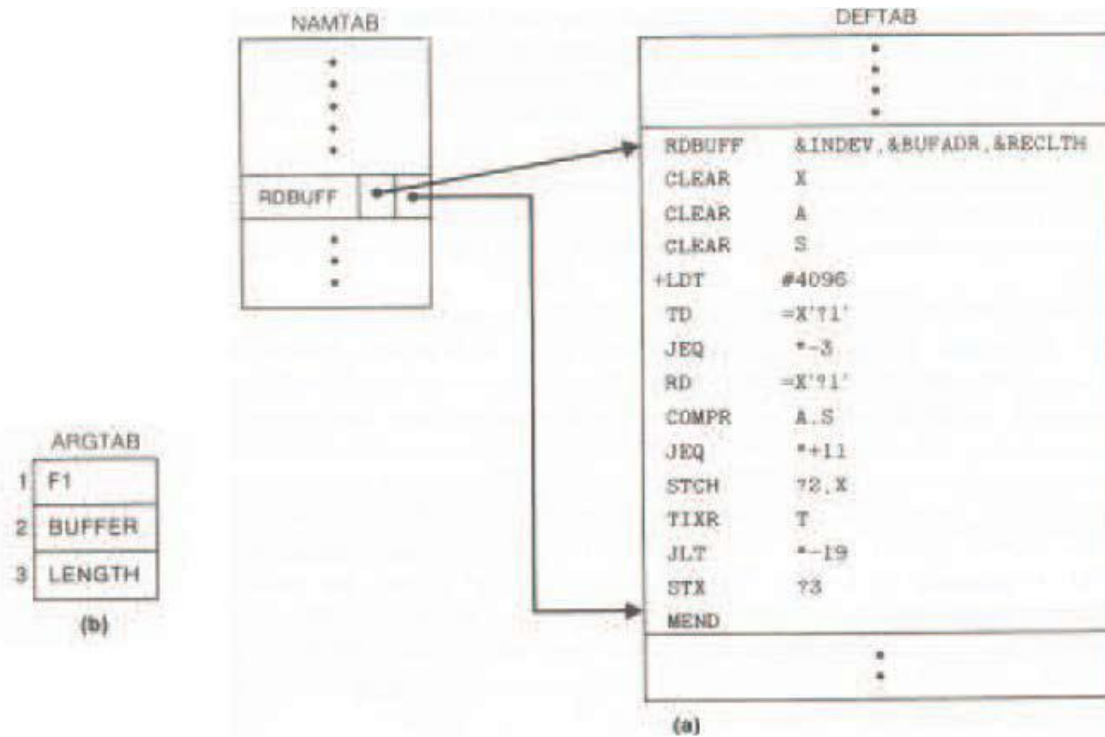
Macro invocation
» There is no single token that constitutes the macro ―name‖
» Constructing an index of all macro headers according to the keywords in the first two tokens of the header
» Example
– DEFINITION:
 ADD %1 TO %2
 ADD %1 TO THE FIRST ELEMENT OF %2
– INVOCATION:
 DISPLAY TABLE

5.Explain the various data structures used in the implementation of a macro processor. 10M
Data structures-5M
Implementation-5M

The design considered is for one-pass assembler. The data structures required are

• DEFTAB (Definition Table)
o Stores the macro definition including macro prototype and macro body
o Comment lines are omitted.
o References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
• NAMTAB (Name Table)
o Stores macro names
o Serves as an index to DEFTAB
 Pointers to the beginning and the end of the macro definition (DEFTAB)
• ARGTAB (Argument Table)
o Stores the arguments according to their positions in the argument list.
o As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.
o The figure below shows the different data structures described and their relationship.





6.Explain the data structures and pass 1 algorithm of SIC assembler. 10M

Explanation of datastructures-5M
Algorithm-5M

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

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate

the error conditions (e.g., if a symbol is defined in two different places).

• During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
• During Pass 2: Symbols used as operands are looked up the symbol table to obtain
the address value to be inserted in the assembled instructions.

• SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

• Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the
pass 2.
• A copy of the source program is also an input to the pass 2, which is used to retain
the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

LOCCTR:
Apart from the SYMTAB and OPTAB, this is another important variable which helps in
the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

The Algorithm for Pass 1:
Begin
read first input line
if OPCODE = „START" then begin
save #[Operand] as starting addr
initialize LOCCTR to starting address
write line to intermediate file
read next line
end(if START)
else
initialize LOCCTR to 0
While OPCODE != „END" do
begin
if this is not a comment line then
begin
if there is a symbol in the LABEL field then
begin
search SYMTAB for LABEL
if found then
set error flag (duplicate symbol)
else
(if symbol)
search OPTAB for OPCODE
if found then
add 3 (instr length) to LOCCTR
else if OPCODE = „WORD" then
add 3 to LOCCTR
else if OPCODE = „RESW" then

add 3 * #[OPERAND] to LOCCTR
else if OPCODE = „RESB‟ then
add #[OPERAND] to LOCCTR
else if OPCODE = „BYTE‟ then
begin
find length of constant in bytes
add length to LOCCTR
end
else
set error flag (invalid operation code)
end (if not a comment)
write line to intermediate file
read next input line
end { while not END}
write last line to intermediate file
Save (LOCCTR – starting address) as program length
End {pass 1}

7a . Generate an algorithm for absolute loader. 5M

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

7b . What is loader? Develop an algorithm for a bootstrap loader. 5M

Definition of loader-1M
Algorithm-4m

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
It uses a subroutine GETC, which is
GETC A←read one character
if A=0x04 then jump to 0x80
if A<48 then GETC
A ← A-48 (0x30)
if A<10 then return
A ← A-7
return

8. Generate the complete object program for the following SIC/XE program .
Generation of address 3M
Calculation of object codes 4M
Complete object program 3M

LOC				object code
	COPY	START	1000	
1000	CLOOP	+JSUB	RDREC	A3101157
1004		LDA	LENGTH	83214A
1007		COMP	ZERO	932141
100A		JEQ	EXIT	B32003
100D		J	CLOOP	BB2FFC
1010	EXIT	STA	BUFFER	532009
1013		LDA	THREE	832138
1016		STA	TOTAL_LENGTH	53213B
1019		RSUB		
101C	BUFFER	RESW	100	4C0000
1148	EOF	BYTE	C ’EOF’	454F46
114B	ZERO	WORD	0	
114E	THREE	WORD	3	
1151	LENGTH	RESW	1	
1154	TOTAL_LENGTH	RESW	1	
1157	RDREC	LDX	ZERO	632001

MNEMONICS:

JSUB=A0, LDA=80, LDX=60, STA=50, COMP=90, RSUB=4C, JEQ=B0, J=B8

.