FIFTH SEMESTER BTECH DEGREE MODEL EXAMINATION

Computer Science and Engineering

CS303 System Software

Time: 3 Hours **Maximum Marks: 100**

PART A

Answer ALL questions. Each question carries 3 marks each

1.	What is system software? Give examples	3 Marks
2.	Write any three assembler directives in SIC\XE. Mention their use.	3 Marks
3.	Give the format of header, text and end records.	3 Marks
4.	What are the datastructures used in an assembler?	3 Marks

4*3=12 Marks

PART B

Answer ANY TWO questions. Each question carries 9 marks each

5.	Design a two pass assembler.	9 Marks
6.	Write a sequence of instructions for SIC/XE to divide BETA by GAMMA setting ALPHA to the integer portion of the quotient and DELTA to the remainder. Use register to register instructions to make the calculation as efficient as possible.	9 Marks
7.	a.)Give a brief idea about assemblers	4 ½ Marks
	b.)What is the purpose of pass one and two in two pass assembler?	4 ½ Marks

9*2=18 Marks

PART C

Answer ALL questions. Each question carries 3 marks each

8. How literals differ from immediate operand? 3 Marks
9. What is the difference between the following sequences of statements 3 Marks

```
i.          LDA #3
ii. THREE   EQU   3
       .
       :
       LDA       #THREE
```

10. What are the functions of a loader? 3 Marks
11. Give the algorithm of bootstrap loader. 3 Marks

4*3=12Marks

PART D

Answer ANY TWO questions. Each question carries 9 marks each

12.	Explain the algorithm of single pass assembler.	9 Marks
13.	Explicate the algorithm and datastructures for a linking loader	9 Marks
14.	a) What is the difference between the assembler directive EXTREF and EXTDEF?	4 ½ Marks
	b) Differentiate between linking loader and linkage editor?	4 ½ Marks

9*2=18 Marks

PART E

Answer ANY FOUR questions. Each question carries 10 marks each

15.	a.)Compare macro processor design options	5 Marks
	b.)What are the problems in recursive macro expansion?	5 Marks
16.	Write an algorithm for implementing one pass macro processor	10 Marks
17.	Explain about machine independent macro processor features	10 Marks
18.	Briefly explain the structure of an editor	10 Marks
19.	Explain the role of debuggers in program development	10 Marks
20.	Explain about character and block device drivers	10 Marks

4*10=40 Marks

FIFTH SEMESTER BTECH DEGREE MODEL EXAMINATION

Computer Science and Engineering

CS303 System Software

Time: 3 Hours **Maximum Marks: 100**

Answer key

PART A

1. System software consists of a variety of programs that support the operation of a computer. Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

Assembler translates mnemonic instructions into machine code. Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

Definition -1 marks

Any 2 example with definition- 2 marks

1+2=3 Marks

2. Assembler directives in SIC/XE

– START: Specify name & starting address.

– END: End of the program, specify the first execution instruction.

– BYTE: Generates character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
-WORD: Generates one word integer constant
-RESB: Reserves the indicated number of bytes for a data area
- RESW: Reserves the indicated number of words for a data area
Any 3 assembler directive with purpose- 1 mark each
1+1+1=3 Marks

3.

The simple object program we use contains three types of records:
• **Header record**

Col 1 H
Col. 2-7 Program name

Col 8-13 Starting address of object program (hexadecimal)
Col 14-19 Length of object program in bytes (hexadecimal)
Text record:
Col. 1 T
Col 2-7. Starting address for object code in this record (hexadecimal)
Col 8-9 Length off object code in this record in bytes (hexadecimal)
Col 10-69 Object code, represented in hexadecimal (2 columns per byte of object code)
End record:
Col. 1 E
Col 2-7 Address of first executable instruction in object program (hexadecimal)
Header Record-1 Mark
Text Record-1 Mark
End Record-1 Mark
1+1+1=3 Marks

4. **Datastructures used in assembler**

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.

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).

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.

OPTAB- 1 Mark

SYMTAB- 1Mark

LOCCTR- 1Mark

1+1+1=3 Marks

PART B

5. Design of two pass assembler

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 pass 1 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}
```

The Algorithm for Pass 2:

```
Begin
read 1st input line
if OPCODE = `START'  then
begin
write listing line
read next input line
end
```

```
write Header record to object program
initialize 1st Text record
while OPCODE != `END' do
begin
if this is not comment line then
      begin
search OPTAB for OPCODE
if found then
begin
if there is a symbol in OPERAND field then
begin
search SYMTAB for OPERAND field then
if found then
begin
store symbol value as operand address
else
begin
store 0 as operand address
set error flag (undefined symbol)
end
end (if symbol)
else store 0 as operand address
assemble the object code instruction
else if OPCODE = `BYTE' or `WORD" then
convert constant to object code
if object code doesn' t fit into current Text record then
      begin
Write text record to object code
initialize new Text record
end
add object code to Text record
end {if not comment}
write listing line
read next input line
end
write listing line
read next input line
write last listing line
      End {Pass 2}
```

Datastructures-3 marks

Pass 1 -3 marks

Pass2 – 3 marks

(3+3+3=9 Marks)

6.

LDA BETA

LDS GAMMA

```
DIVR S,A

STA ALPHA

MULR S,A

LDS BETA

SUBR A,S

STS DELTA

:

:

ALPHA RESW 1

BETA RESW 1

GAMMA RESW 1

DELTA RESW 1
```

Use of assembler directives resb/resw- 2 Marks

Logic-7 Marks (2+7=9 Marks)

7. **a)Assemblers**

A tool called an ***assembler*** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer' s 0s and 1s that simplifies writing and reading programs.

The basic assembler functions are:
- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.
- The design of assembler can be to perform the following:
  - – Scanning (tokenizing)
  - – Parsing (validating the instructions)
  - – Creating the symbol table
  - – Resolving the forward references
  - – Converting into the machine language
- The design of assembler in other words:
  - – Convert mnemonic operation codes to their machine language equivalents
  - – Convert symbolic operands to their equivalent machine addresses
  - – Decide the proper instruction format Convert the data constants to internal machine
    representations
  - – Write the object program and the assembly listing

So for the design of the assembler we need to concentrate on the machine architecture of the SIC/XE machine. We need to identify the algorithms and the various data structures to be used. According to the above required steps for assembling the assembler also has to handle *assembler directives*, these do not generate the object code but directs the assembler to perform certain operation. These directives are:

• SIC Assembler Directive:

– START: Specify name & starting address.

– END: End of the program, specify the first execution instruction. etc

Definition- 2 marks

Overall idea-2.5 marks

2+2.5=4.5 marks

7b)

Two-pass assembler resolves the forward references and then converts into the object code. Hence the process of the two-pass assembler can be as follows:

Pass-1

- Assign addresses to all the statements
- Save the addresses assigned to all labels to be used in *Pass-2*
- Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
- Defines the symbols in the symbol table(generate the symbol table)

Pass-2

- Assemble the instructions (translating operation codes and looking up addresses).
- Generate data values defined by BYTE, WORD etc.
- Perform the processing of the assembler directives not done during *pass-1*.
- Write the object program and assembler listing.

Pass1 -2 marks

Pass2-2.5 marks

2+2.5=4.5 marks

8. **Literals**

The assembler generates the specific value of the operand as a constant and stores it at some other memory location. The address of this generated constant is used as the target address for the machine instruction

Immediate operand

The operand value is assembled as a part of the machine instruction

Literals- 1.5 marks

Immediate operand-1.5 marks

1.5+1.5=3 marks

9. **LDA #3**

The value of the operand is specified as part of the instruction itself is called immediate addressing. Now the OPTAB will be as

A	3

```
. THREE    EQU   3
                   .
                   :
              LDA        #THREE
```

This statement, THREE EQU 3, defines the given symbol, THREE (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. Now the SYMTAB is as follows:

SYMBOLNAME	VALUE
THREE	3

Description about LDA #3---1.5 marks

Description about EQU---1.5 marks

1.5+1.5=3 marks

10. Functions of a loader

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)
Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)
Each one carries 1 mark.
1+1+1=3 marks

11. The algorithm of bootstrap loader.
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
```

12. Algorithm of single pass assembler

One-pass assemblers are used when it is necessary or desirable to avoid a second pass over the source program the external storage for the intermediate file between two passes is slow or is inconvenient to use

Main problem: forward references to both data and instructions

One simple way to eliminate this problem: require that all areas be defined before they are referenced. It is possible, although inconvenient, to do so for data items.

Forward jump to instruction items cannot be easily eliminated.

Data structures for assembler:

Op code table

Looked up for the translation of mnemonic code

key: mnemonic code

result: bits

Hashing is usually used once prepared, the table is not changed

efficient lookup is desired since mnemonic code is predefined, the hashing function can be tuned a priori The table may have the instruction format and length to decide where to put op code bits, operands bits, offset bits for variable instruction size used to calculate the address

Symbol table

Stored and looked up to assign address to labels efficient insertion and retrieval is needed deletion does not occur

Difficulties in hashing non random keys

Problem:

the size varies widely

pass 1: loop until the end of the program

1. Read in a line of assembly code

2. Assign an address to this line increment N (word addressing or byte addressing)

3. Save address values assigned to labels in symbol tables

4. Process assembler directives constant declaration space reservation

Algorithm for Pass 1 assembler:

```
begin
    if starting address is given
        LOCCTR = starting address;
    else
        LOCCTR = 0;
    while OPCODE != END do            ;; or EOF
        begin
        read a line from the code
        if there is a label
            if this label is in SYMTAB, then error
            else insert (label, LOCCTR) into SYMTAB
        search OPTAB for the op code
        if found
            LOCCTR += N        ;; N is the length of this instruction (4 for
MIPS)
        else if this is an assembly directive
            update LOCCTR as directed
        else error
        write line to intermediate file
        end
    program size =  LOCCTR - starting address;
end
```

Forward Reference:

Omits the operand address if the symbol has not yet been defined

Enters this undefined symbol into SYMTAB and indicates that it is undefined

Adds the address of this operand address to a list of forward references associated with the SYMTAB entry

Scans the reference list and inserts the address when the definition for the symbol is encountered.

Reports the error if there are still SYMTAB entries indicated undefined symbols at the end of the program

Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

Algorithm-7 marks

Forward reference-2 marks

7+2=9 marks

13.

Algorithm and Data structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program, which is already given. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism.

Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Pass1 algorithm

```
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}
```

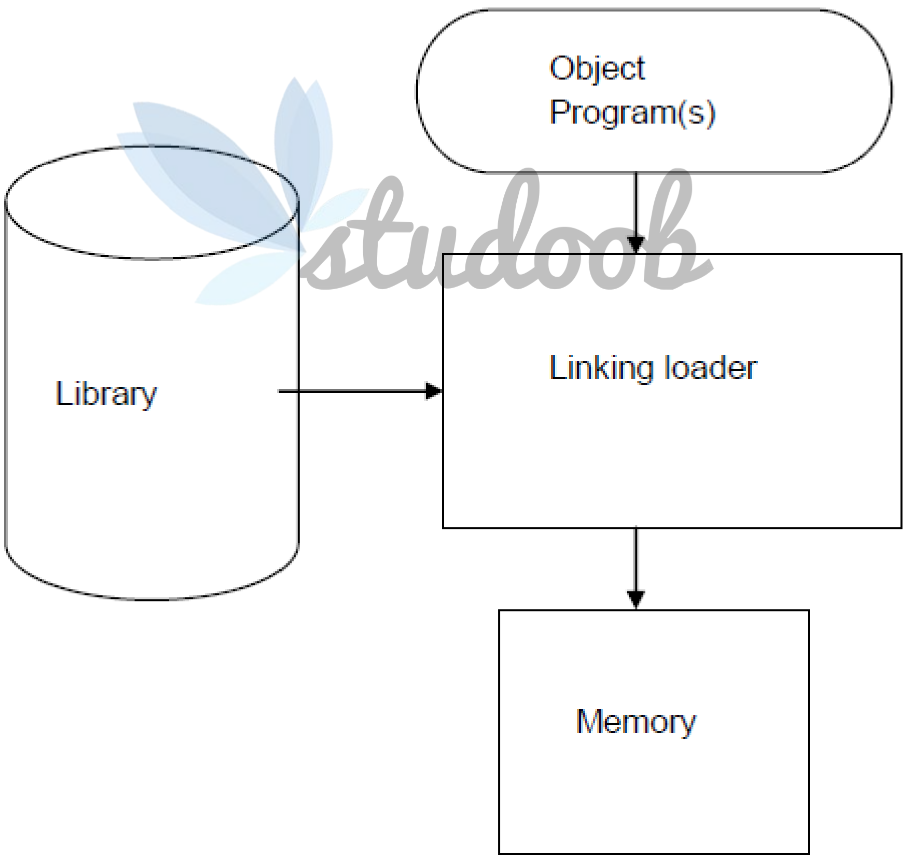
Pass2 algorithm

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}
```

Algorithm- pass1 and pass2 (4+4=8 marks)
Datastructures-1 marks
8+1=9 marks
14.
a)
Define Record(EXTDEF)
A Define Record gives information about external symbols that are defined in the same control section. (These symbols can be used by other sections)
Refer Record:(EXTREF)
A Refer Record gives information about external reference ie. That symbols are defined in another sections.
EXTDEF- 2.5 MARKS
EXTREF-2 MARKS
14b)
A linking loader performs all linking and relocation operations, including automatic library search, and loads the linked program into memory for execution. • A linkage editor produces a linked version of the program, called a load module or an executable image, which is normally written to a file for later execution.

Linking Loaders



LINKING LOADER- 2 .5MARKS
LINKAGE EDITOR-2 MARKS
(2.5+2=4.5 MARKS)

15.

a)There are two design options

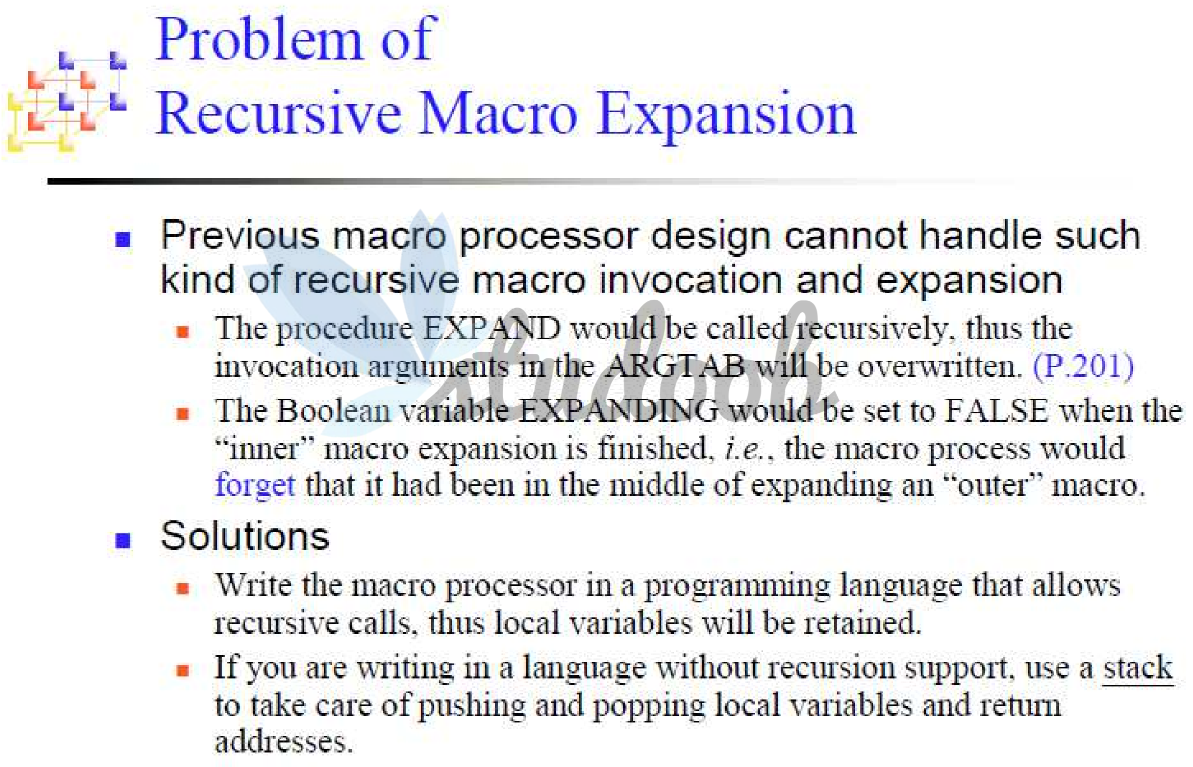
Single pass

- every macro must be defined before it is called
- one-pass processor can alternate between macro definition and macro expansion
- nested macro definitions may be allowed but nested calls are not

Two pass algorithm

- Pass1: Recognize macro definitions
- Pass2: Recognize macro calls
- nested macro definitions are not allowed

single pass-2.5 marks
two pass -2.5 marks
(2.5+2.5=5 marks)
15 b)

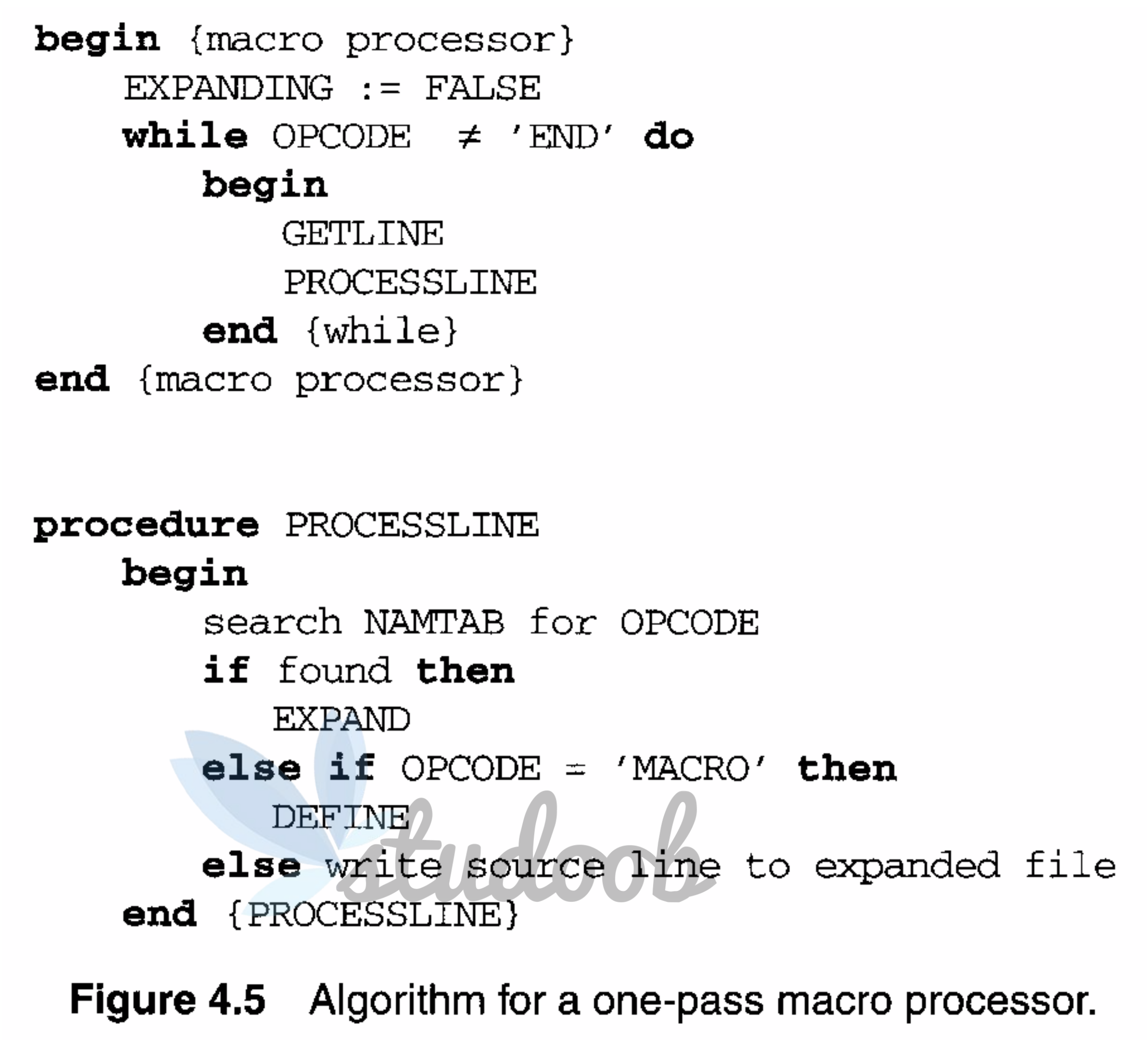


Problem of Recursive Macro Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion
  - The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten. (P.201)
  - The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, *i.e.*, the macro process would forget that it had been in the middle of expanding an "outer" macro.
- Solutions
  - Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  - If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

Problem-2.5 marks
Solution-2.5 marks
(2.5+2.5=5marks)
16.
One pass macro processor algorithm

- **Prerequisite**
  - **every macro must be defined before it is called**
- **Sub-procedures**
  - **macro definition: DEFINE**
  - **macro invocation: EXPAND**



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


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
           EXPAND
        else if OPCODE = 'MACRO' then
           DEFINE
        else write source line to expanded file
    end {PROCESSLINE}
```

Figure 4.5 Algorithm for a one-pass macro processor.

•

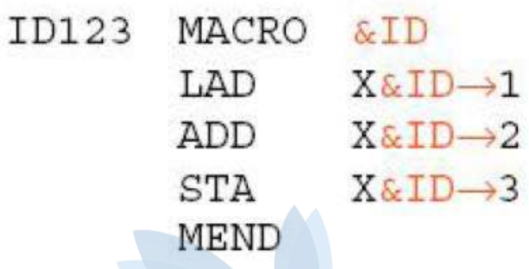
Algorithm- 10 marks

17. Machine independent macro processor features

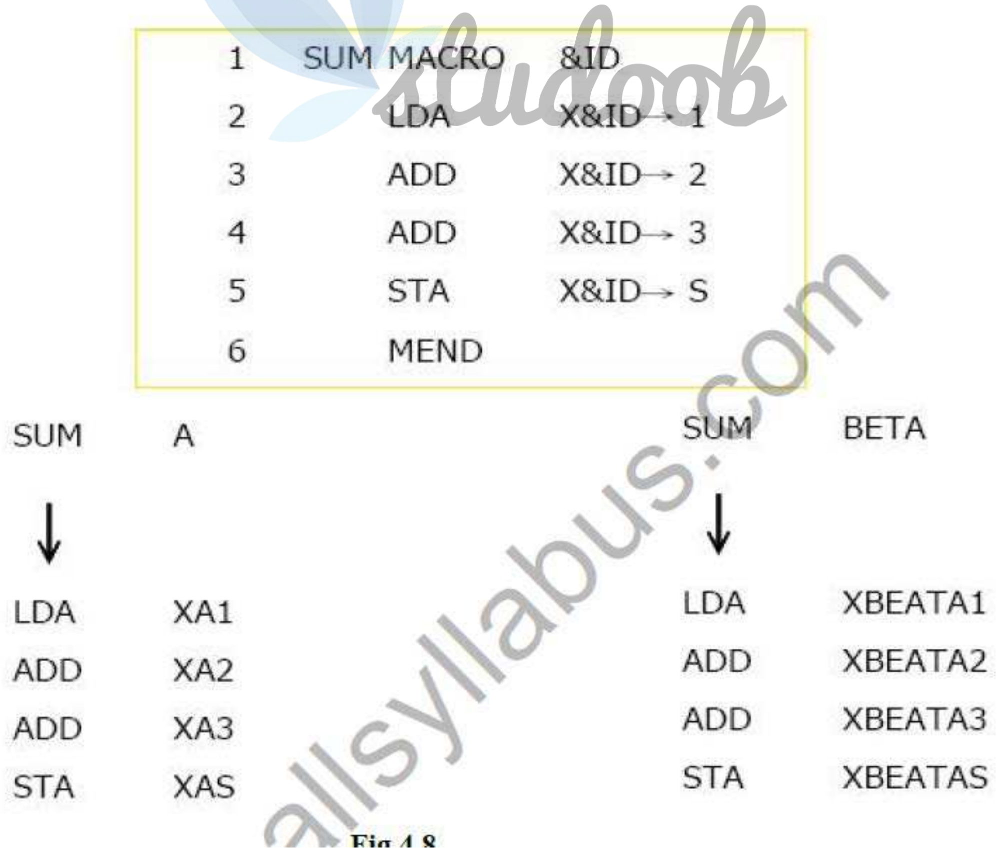
Machine-independent Macro-Processor Features.
The design of macro processor doesn' t depend on the architecture of the machine.
These features are:
- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

4.2.1 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&ID1
& 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
LDA X&ID→



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



```
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
```

Fig 4.8

The above figure shows a macro definition that uses the concatenation operator as previously described. The statement SUM A and SUM BETA shows the invocation statements and the corresponding macro expansion.

4.2.2 Generation of Unique Labels

As discussed 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.

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

```
25      RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
30               CLEAR    X                        CLEAR LOOP COUNTER
35               CLEAR    A
40               CLEAR    S
45              +LDT      #4096                    SET MAXIMUM RECORD LENG
50      $LOOP    TD       =X'&INDEV'               TEST INPUT DEVICE
55               JEQ      $LOOP                    LOOP UNTIL READY
60               RD       =X'&INDEV'               READ CHARACTER INTI REG A
65               COMPR    A, S                     TEST FOR END OF RECORD
70               JEQ      $EXIT                    EXIT LOOP IF EOR
75               STCH     &BUFADR, X               STORE CHARACTER IN BUFFEI
80               TIXR     $LOOP                    HAS BEEN REACHED
90      $EXIT    STX      &RECLTH                  SAVE RECORD LENGTH
                 MEND
```

The following figure shows the macro invocation and expansion first time.

```
        .        RDBUFF   F1, BUFFER, LENGTH


30               CLEAR    X                        CLEAR LOOP COUNTER
35               CLEAR    A
40               CLEAR    S
45              +LDT      #4096                    SET MAXIMUM RECORD LENGTH
50      $AALOOP  TD       =X'F1'                   TEST INPUT DEVICE
55               JEQ      $AALOOP                  LOOP UNTIL READY
60               RD       =X'F1'                   READ CHARACTER INTI REG A
65               COMPR    A, S                     TEST FOR END OF RECORD
70               JEQ      $AAEXIT                  EXIT LOOP IF EOR
75               STCH     BUFFER, X                STORE CHARACTER IN BUFFER
80               TIXR     T                        LOOP UNLESS MAXIMUM LENGTH
85               JLT      $AALOOP                  HAS BEEN REACHED
90      $AAEXIT  STX      LENGTH                   SAVE RECORD LENGTH
```

If the macro is invoked second time the labels may be expanded as $ABLOOP $ABEXIT.

4.2.3 Conditional Macro Expansion

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.

4.2.4 Keyword Macro Parameters

All the macro instruction definitions used positional parameters. Parameters an arguments are matched according to their positions in the macro prototype and the macr invocation statement. The programmer needs to be careful while specifying th arguments. If an argument is to be omitted the macro invocation statement must contain null argument mentioned with two commas.

Positional parameters are suitable for the macro invocation. But if the macr invocation has large number of parameters, and if only few of the values need to be use in a typical invocation, a different type of parameter specification is required (fo example, in many cases most of the parameters may have default values, and th invocation may mention only the changes from the default values).

```
Ex:   XXX MACRO &P1, &P2, …., &P20, ….
      XXX A1, A2,,,,,,,,,,…,,A20,…..
                Null arguments
```

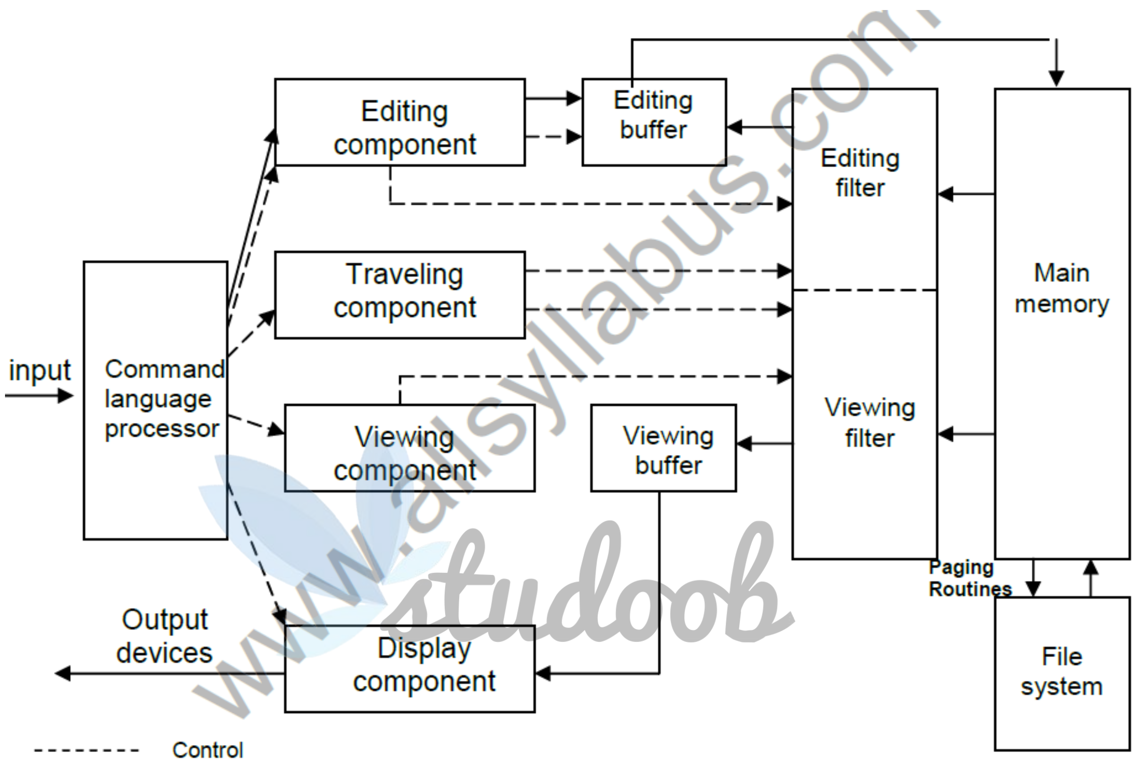
Keyword parameters

- Each argument value is written with a keyword that names the corresponding parameter.
- Arguments may appear in any order.
- Null arguments no longer need to be used.
- Ex: XXX P1=A1, P2=A2, P20=A20.
- It is easier to read and much less error-prone than the positional method.

18. the structure of an editor

Most text editors have a structure similar to that shown in the following figure. That is most text editors have a structure similar to shown in the figure regardless of features and the computers

Command language Processor accepts command, uses semantic routines – performs functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions.



Typical Editor Structure

Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.

In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste paragraph etc..,.

When editing command is issued, editing component invokes the editing filter – generates a new editing buffer – contains part of the document to be edited from current editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer being created.

In viewing a document, the start of the area to be viewed is determined by the current viewing pointer maintained by the viewing component. Viewing component is a collection of modules responsible for determining the next view. Current viewing pointer can be set or reset as a result of previous editing operation.

When display needs to be updated, viewing component invokes the viewing filter – generates a new viewing buffer – contains part of the document to be viewed from current viewing pointer. In case of line editors – viewing buffer may contain the current line, Screen editors - viewing buffer contains a rectangular cutout of the quarter plane of the text. Viewing buffer is then passed to the display component of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen – called a window. The editing and viewing buffers may be identical or may be completely disjoint. Identical – user edits the text directly on the screen. Disjoint – Find and Replace (For example, there are 150 lines of text, user is in 100th line, decides to change all occurrences of 'text editor' with 'editor'). The editing and viewing buffers can also be partially overlap, or one may be completely contained in the other. Windows typically cover entire screen or a rectangular portion of it. May show different portions of the same file or portions of different file. Inter-file editing operations are possible.

The components of the editor deal with a user document on two levels: In main memory and in the disk file system. Loading an entire document into main memory may be infeasible – only part is loaded – demand paging is used – uses editor paging routines. Documents may not be stored sequentially as a string of characters. Uses separate editor data structure that allows addition, deletion, and modification with a minimum of I/O and character movement.

Editors function in three basic types of computing environments: Time sharing, Stand-alone, and Distributed. Each type of environment imposes some constraints on the design of an editor.

In time sharing environment, editor must function swiftly within the context of the load on the computer's processor, memory and I/O devices. In stand-alone environment, editors on stand-alone system are built with all the functions to carry out editing and viewing operations – The help of the OS may also be taken to carry out some tasks like demand paging. In distributed environment, editor has both functions of stand-alone editor, to run independently on each user's machine and like a time sharing editor, contend for shared resources such as files.

Block diagram-4 marks

Explanation-6 marks (4+6=10 marks)

19. The Role Of Debuggers In Program Development

An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs. Many such systems are available during these days. Our discussion is broad in scope, giving the overview of interactive debugging systems – not specific to any particular existing system.

1. **Debugging Functions and Capabilities**

Program-Display capabilities

2.

Relationship with Other Parts of the System

3.

User-Interface Criteria

4.

1.

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as tracing and traceback. Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on… Traceback can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

2.

A debugger should have good program-display capabilities. Program being debugged should be displayed completely with statement numbers. The program may be displayed as originally written or with macro expansion. Keeping track of any changes made to the programs during the debugging session. Support for symbolically displaying or modifying the contents of any of the variables and constants in the program. Resume execution – after these changes.

To provide these functions, a debugger should consider the language in which the program being debugged is written. A single debugger – many programming languages – language independent. The debugger - a specific programming language – language dependent. The debugger must be sensitive to the specific language being debugged.

The context being used has many different effects on the debugging interaction. The statements are different depending on the language

```
Cobol -   MOVE 6.5 TO X
Fortran -  X = 6.5
C       -  X = 6.5
```

Examples of assignment statements

Similarly, the condition that X be unequal to Z may be expressed as

```
Cobol   - IF X NOT EQUAL TO Z
Fortran - IF ( X.NE.Z)
C       - IF ( X <> Z)
```

Similar differences exist with respect to the form of statement labels, keywords and so on…

The notation used to specify certain debugging functions varies according to the language of the program being debugged. Sometimes the language translator itself has debugger interface modules that can respond to the request for debugging by the user. The source code may be displayed by the debugger in the standard form or as specified by the user or translator.

It is also important that a debugging system be able to deal with optimized code. Many optimizations like

- Invariant expressions can be removed from loops
- Separate loops can be combined into a single loop
- Redundant expression may be eliminated
- Elimination of unnecessary branch instructions

Leads to rearrangement of segments of code in the program. All these optimizations create problems for the debugger, and should be handled carefully.

3.

The important requirement for an interactive debugger is that it always be available. Must appear as part of the run-time environment and an integral part of the system. When an error is discovered, immediate debugging must be possible. The debugger must communicate and cooperate with other operating system components such as interactive subsystems.

Debugging is more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops. The debugger must also exist in a way that is consistent with the security and integrity components of the system. The debugger must coordinate its activities with those of existing and future language compilers and interpreters.

4.

Debugging systems should be simple in its organization and familiar in its language, closely reflect common user tasks. The simple organization contribute greatly to ease of training and ease of use. The user interaction should make use of full-screen displays and windowing-systems as much as possible. With menus and full-screen editors, the user has far less information to enter and remember. There should be complete functional equivalence between commands and menus – user where unable to use full-screen IDSs may use commands. The command language should have a clear, logical and simple syntax; command formats should be as flexible as possible. Any good IDSs should have an on-line HELP facility. HELP should be accessible from any state of the debugging session.

Point1-3 marks

Point2-3 marks

Point3- 2 marks

Point4-2 marks

3+3+2+2=10 marks

20. character and block device drivers

Write a complete char device driver

scull

a. Simple Character Utility for Loading Localities

b. Not hardware dependent

c. Just acts on some memory allocated from the kernel

The Design of scull

- Implements various devices
- scull0 to scull3
    - Four device drivers, each consisting of a memory area
        - Global
            - Data contained within the device is shared by all the file descriptors that opened it
        - Persistent
            - If the device is closed and reopened, data isn' t lost
- The Design of scull
- scullpipe0 to scullpipe3
    - Four FIFO devices
    - Act like pipes
    - Show how blocking and nonblocking read and write can be implemented
        - Without resorting to interrupts
        - The Design of scull
- scullsingle
    - Similar to scull0
    - Allows only one process to use the driver at a time
- scullpriv
    - Private to each virtual console
        - The Design of scull
- sculluid
    - Can be opened multiple times by one user at a time
    - Returns “Device Busy” if another user is locking the device
- scullwuid
    - Blocks open if another user is locking the device
        - Major and Minor Numbers

- Char devices are accessed through names in the file system
    - Special files/nodes in /dev

```
>cd /dev
>ls –l
crw------- 1 root  root     5,    1 Apr 12 16:50 console
brw-rw---- 1 root  disk     8,    0 Apr 12 16:50 sda
brw-rw---- 1 root  disk     8,    1 Apr 12 16:50 sda1
```

    - Major and Minor Numbers

- Char devices are accessed through names in the file system
    - Special files/nodes in /dev

```
>cd /dev
>ls –l
crw------- 1 root  root     5,    1 Apr 12 16:50 console
brw-rw---- 1 root  disk     8,    0 Apr 12 16:50 sda
brw-rw---- 1 root  disk     8,    1 Apr 12 16:50 sda1
```

- Major and Minor Numbers
- Major number identifies the driver associated with the device
    - /dev/sda and /dev/sda1 are managed by driver 8
- Minor number is used by the kernel to determine which device is being referred to
- The Internal Representation of Device Numbers
- dev_t type, defined in <linux/types.h>
- Macros defined in <linux/kdev_t.h>
    - 12 bits for the major number
        - Use MAJOR(dev_t dev) to obtain the major number
    - 20 bits for the minor number
        - Use MINOR(dev_t dev) to obtain the minor number
    - Use MKDEV(int major, int minor) to turn them into a dev_t

- Allocating and Freeing Device Numbers
- To obtain one or more device numbers, use

  int register_chrdev_region(dev_t first, unsigned int count, char *name);

    - first
        - Beginning device number
        - Minor device number is often 0
    - count
        - Requested number of contiguous device numbers
    - name
        - Name of the device
- Allocating and Freeing Device Numbers
- To obtain one or more device numbers, use

  int register_chrdev_region(dev_t first, unsigned int count, char *name);

    - Returns 0 on success, error code on failure

Block device driver

- Block Drivers
- Provides access to devices that transfer randomly accessible data in ***blocks***, or fixed size chunks of data (e.g., 4KB)
    - Note that underlying HW uses ***sectors*** (e.g., 512B)
- Bridge core memory and secondary storage
    - Performance is essential
    - Or the system cannot perform well
- Block driver registration
- To register a block device, call

  int register_blkdev(unsigned int major,

  const char *name);

    - **major**: major device number
        - If 0, kernel will allocate and return a new major number

    - **name**: as displayed in **/proc/devices**

- To unregister, call

  int unregister_blkdev(unsigned int major,

  const char *name);

- Block driver registration
- To register a block device, call

  int register_blkdev(unsigned int major,

  const char *name);

    - **major**: major device number
        - If 0, kernel will allocate and return a new major number
    - **name**: as displayed in **/proc/devices**

- To unregister, call

  int unregister_blkdev(unsigned int major,

  const char *name);

character device driver-5 marks

block device driver- 5 marks

5+5=10 marks