CBCS SCHEME

USN | 1 | CA | 19 | MC | A | 59 | 18MCA34

Third Semester MCA Degree Examination, Dec.2019/Jan.2020
System Software

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Explain the architecture of SIC program. **(12 Marks)**

b. Explain the following with respect to SIC/XE machine Architecture with example.
(i) Instruction formats (ii) Addressing modes. **(08 Marks)**

2 a. What are the basic functions of an assembler? Explain Basic Assembler Directives with example. **(10 Marks)**

b. Write an algorithm for Pass-1 assembler. **(10 Marks)**

Module-2

3 a. Generate an object code for the following program using the OPCODES as given and also construct the symbol table.

```
Sample   START   1000
         LDS     #3
         LDT     #300
         LDX     #0
ADDLP    LDA     ALPHA, X
         ADD     BETA, X
         STA     GAMMA, X
         ADDR,   S, X
         COMPR   X, T
         JLT     ADDLP
ALPHA    RESW    100
BETA     RESW    100
GAMMA    RESW    100
```

OPCODE :

LDA – 00 LDS → 6C
STA – 0C
LDX – 04
ADD – 18
COMPR – A0
JLT – 38
LDT – 74
ADDR – 90 **(10 Marks)**

b. What is relocatable program? Explain the concept of Program relocation with an example and the means for implementing it. **(10 Marks)**

4 a. Write an algorithm for one-pass assembler. **(12 Marks)**

b. Write a note on MASM assembler. **(08 Marks)**

Module-3

5 a. What are the functions of a loader? (08 Marks)

b. Write the algorithm for pass -1 and pass – 2 of a linking loader. Also explain the data structures. (12 Marks)

6 a. Explain the following loader design options :
(i) Linkage Editor (ii) Dynamic Linking (12 Marks)

b. Write and explain an algorithm for absolute loader. (08 Marks)

Module-4

7 a. Explain Macro Definition and Expansion. (06 Marks)

b. Explain Macro processor algorithm and data structures. (14 Marks)

8 a. Explain the following :
(i) Concatenation of macro parameters
(ii) Keyword macro parameters. (12 Marks)

b. Explain Recursive Macro Expansion. (08 Marks)

Module-5

9 a. Explain recursion descent parsing. Write recursive descent Parse for 'READ' statement. (10 Marks)

b. Indicate whether the finite automation given in Fig.Q9(b), recognize the following strings:
(i) 9Alpha (ii) Num – 2 (iii) –Hello (iv) aaa – 8 –

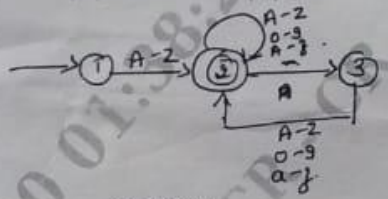


Fig.Q9(b) (04 Marks)

c. Define the following terms :
(i) Grammar (ii) Lexical analysis (06 Marks)

10 a. By using the BNF grammar below represent the syntax analysis of the PASCAL statement. VAR := SUMSQ DIV 100 – MEAN * MEAN in the parse tree.
<assign> :: = id := <exp>
<exp> :: = <term> | <exp> + <term> | <exp> – <term>
<term> :: = <factor> | <term> * <factor> | <term> DIV <factor>
<factor :: = id | int | (<exp>) (10 Marks)

b. Briefly discuss different machine dependent code optimization techniques. (10 Marks)

* * * * *

Q1a) Explain the architecture of SIC Program

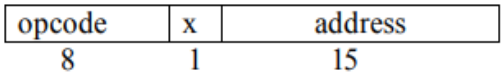
1) Memory
   - Memory consists of 8-bit bytes.
   - 3 consecutive bytes form a word (24 bits).
   - All the address in SIC are byte addresses.
   - Words are addressed by the location of their lowest numbered byte.
   - There are total of 32,768 bytes in the computer memory.
2) Registers

   There are five registers, each 24 bits in length.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register, the jump to subroutine instruction stores the return address in this register.
PC	8	Program counter, contains the address of the next instruction to be fetched for execution.
SW	9	Status word, contains a variety of information, including a Condition Code.

3) Data Formats
   - Integers are stored as 24 bit binary numbers; 2‟s complement representation is used for negative values.
   - Characters are stored using their 8-bit ASCII codes.
   - There is no floating point hardware on the standard version of SIC.
4) Instruction Formats

   All machine instructions on the standard version of SIC have the following 24-bit format



   The flag bit x is used to indicate indexed addressing mode.
5) Addressing Modes

   There are two addressing modes, indicated by the setting of the x bit in the instruction

Mode	**Indication**	**Target address calculation**
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x)

6) Instruction Set

   SIC provides a basic set of instructions that are sufficient for most simple task.

   i) Data transfer instruction: This include instructions that load and store registers. Eg. LDA, LDX, STA, STX.

   ii) Arithmetic operation instruction: Basic arithmetic operations that involves register A Eg. ADD, SUB, MUL, DIV, COMP.

   iii) Conditional Branching: Conditional jump instructions test the settings of conditional code and jump accordingly. Eg. JLT, JGT, JEQ.

iv) Subroutine call Instructions: Perform subroutine linkage. Eg. JSUB, RSUB. Return address is stored in linkage(L) register.

7) Input and Output
   - Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator).
   - Each device is assigned a unique 8bit code.
   - There are 3 I/O instructions.
   - The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

Q1b) Explain the following with respect to SIC/XE machine architecture with example i)instruction format ii) Addressing modes

Instruction Formats

 SIC/XE has larger memory hence instruction format of standard SIC version

is no longer suitable.

 SIC/XE provide two possible options; using relative addressing (Format 3)

and extend the address field to 20 bit (Format 4).

 In addition SIC/XE provides some instructions that do not reference memory

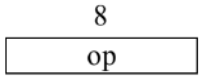
at all. (Format 1 and Format 2).

 The new set of instruction format is as follows. Flag bit e is used to

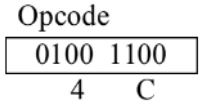
distinguish between format 3 and format 4. (e=0 means format 3, e=1 means

format 4)

1. Format 1 (1 byte)



Example RSUB (return to subroutine)



2. Format 2 (2 bytes)

8	4	4
op	r1	r2

Example COMPR A, S (Compare the contents of register A & S)

Opcode	A	S
1010 0000	0000	0100
A 0	0	4

3. Format 3 (3 bytes)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

Example LDA #3(Load 3 to Accumlator A)

0000 00	0	1	0	0	0	0	0000 0000 0011
0	n	i	x	b	p	e	0 0 3

4. Format 4 (4 bytes)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

Example JSUB RDREC(Jump to the address, 1036)

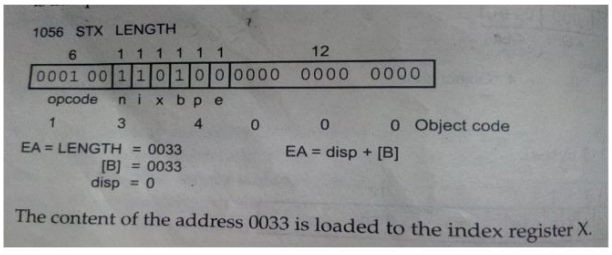
Addressing Modes

Two new relative addressing modes are available for use with instructions assembled using Format 3

Mode	Indication	Target address calculation
Base Relative	b=1, p=0	TA = (B) + disp (0≤ disp ≤ 4095)
Program-counter relative	b=0, p=1	TA = (PC)+disp (-2048 ≤ disp ≤ 2047)

b represents for base relative addressing where as p represents program counter relative addressing. If both the bits b and p are 0 then target address is taken form the address field of the instruction (i.e displacement)

Base relative addressing mode example:



Q2a) What are the basic functions of an assembler? Explain the basic assembler directives with example

In addition to the mnemonic machine instructions assembler uses following assembler directives. These statements are not translated into machine instructions. Instead they provide instructions to assembler itself.

1) START

START specify the name and starting address of the program.

Example: START 1000

2) END

Indicate the end of the source program and (optionally) specify the first executable instruction in the program.

Example: END FIRST

3) BYTE

Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

Example: BYTE X'F1'

4) WORD

Generate one-word integer constant

Example: THREE WORD 3

5) RESB

Reserve the indicate number of bytes for a data area.

Example: BUFFER RESB 4096

6) RESW

Reserve the indicate number of words for a data area.

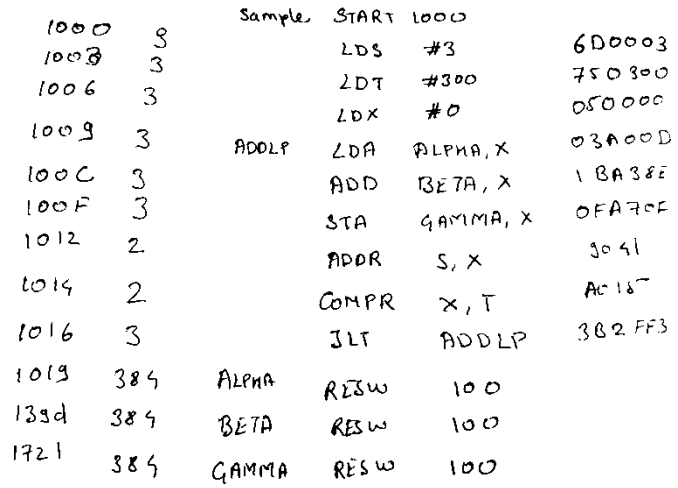
Example: LENGTH RESW 1

Q2b) Write an algorithm for pass1 assembler

Assembler Pass 1:

```
begin
    read first input line
        if OPCODE ='START' then
                begin
                save #[OPERAND] as starting address
                initialize LOCCTR to starting address
                write line to intermediate file
                read next input line
                end {if START}
        else
                initialize LOCCTR to 0
        while OCODE != '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
                                          insert (LABEL,LOCCTR) into SYMTAB
                                  end {if symbol}
                                search OPTAB for OPCODE
                                if found then
                                  add 3 {instruction 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 {if BYTE}
                                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}
```

Q3a) Generate the object code for following program



Q3b) What is relocatable program? Explain the concept of program relocation with an example and means for implementing it.

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

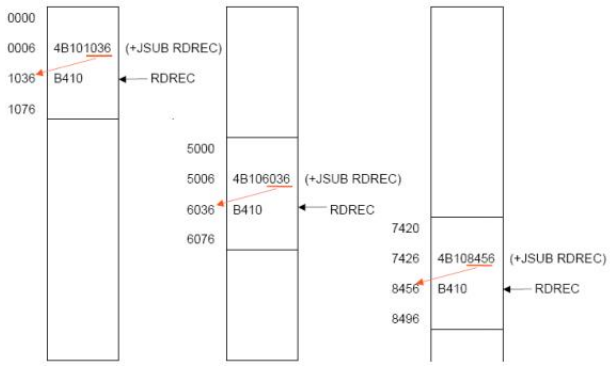
Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program (Hex)

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.

Example of Program Relocation



 The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.

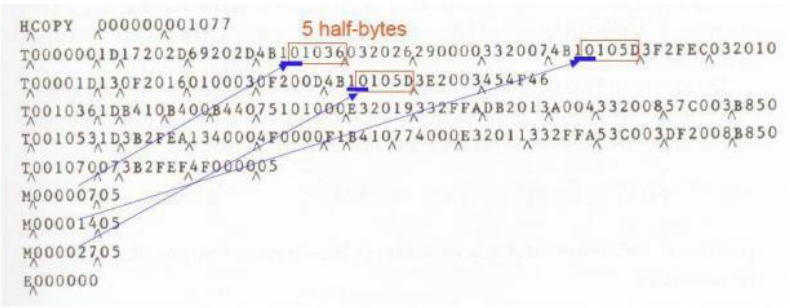
 The address field of this instruction contains 01036, which is the address of the instruction labelled RDREC. The second figure shows that if the program is to be loaded at new location 5000.

 The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420 the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.

 The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn‟t require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.

 From the object program, it is not possible to distinguish the address and constant. The assembler must keep some information to tell the loader.

 For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a Modification record to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program.



In the above object code the red boxes indicate the addresses that need modifications.

The object code lines at the end are the descriptions of the modification records for those instructions which need change if relocation occurs. M00000705 is the modification suggested for the statement at location 0007 and requires modification 5-half bytes.

Similarly the remaining instructions indicate.

Q4a) write an algorithm for one pass assembler

```
while opcode != 'End' do
begin
        if there is no conmment line then
        begin
                if there is a symbol in the LABEL field then
                begin
                        search SYMTAB for LABEL
                        if found then
                        begin
                                if <symbol value> as null
                                set <symbol value> as LOCCTR and search
                                        the linked list with corresponding
                                        operand
                                PTR addresses and generate operand
                                        addresses as corresponding symbol
                                        values
                                set symbol value as LOCCTR in symbol table
                                        and delete  the linked list
                        end
                        else
                                insert (LABEL,LOCCTR) into symtab

                end
                search OPTAB for OPCODE
                if found then
                begin
                        search SYMTAB for OPERAND addresses
                        if found then
                                if symbol value not equal to null then
                                        store symbol value as OPERAND address
                                else
                                        insert at the end of the linked list
                                        with a node with address as LOCCTR
                        else
                                insert (symbol name,null)
                        LOCCTR+=3
                end
                else if OPCODE='WORD' then
                        add 3 to LOCCTR and convert comment to object code
                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 the length of constant in bytes
                        add length to LOCCTR
                        convert constant to object code
                end
                if object code will not fit into current text record then
                begin
                        write text record to object program initialize new Text record
                end
                add object code to Text record
        end
        write listing line
        read next input line
end
write last Text recordto object program
write End record to object program
write last listing line

```

Q4b) Write a note on MASM assembler

2.5.1 MASM Assembler

This section describes some of the features of the Microsoft MASM assembler for Pentium and other x86 systems. Further information about MASM can be found in Barkakati (1992).

As we discussed in Section 1.4.2, the programmer of an x86 system views memory as a collection of segments. An MASM assembler language program is written as a collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST, and STACK.

During program execution, segments are addressed via the x86 segment registers. In most cases, code segments are addressed using register CS, and stack segments are addressed using register SS. These segment registers are automatically set by the system loader when a program is loaded for execution. Register CS is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is set to indicate the last stack segment processed by the loader.

Data segments (including constant segments) are normally addressed using DS, ES, FS, or GS. The segment register to be used can be specified explicitly by the programmer (by writing it as part of the assembler language instruction). If the programmer does not specify a segment register, one is selected by the assembler.

By default, the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME. For example, the directive

```
ASSUME  ES:DATASEG2
```

Q5a) What are the functions of a loader?

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 an Absolute Loader

This loader does not need to perform linking and relocation, its operation is very simple.

The Header Record is checked to verify that the correct program has been

presented for loading (and that it will fit into the available memory).

As each Text Record is read, the object code it contains is moved to the

indicated address in the memory.

When the End Record is encountered the loader jumps to the specified

address to begin execution of the loaded program.

Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form.

Algorithm for an absolute loader

Begin

read Header record

verify program name and length

read first Text record

while record type is ≠ 'E' do

begin

{if object code is in character form, convert into internal

representation}

move object code to specified location in memory

read next object program record

end

jump to address specified in End record

end

2. A simple Bootstrap Loader.

When a computer is first turned on or restarted, a special type of absolute

loader, called bootstrap loader is executed. This bootstrap loads the first

program to be run by the computer -- usually an operating system.

The bootstrap itself begins at address 0. It loads the OS starting address 80

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

Much of the work of the bootstrap loader is performed by the subroutine

GETC. This subroutine read one character from device F1 and converts it

from the ASCII character code to the value of the hexadecimal digit that is

represented by that character

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

Q5b) write the algorithm for pass one and pass2 of a linking loader. Also explain the data structures.

Data Structures

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.

Controlsection	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D8	
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.

Pass 1: (only ***Header*** and ***Define*** records are concerned)

```
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)
```

Figure 3.11(a) Algorithm for Pass 1 of a linking loader.

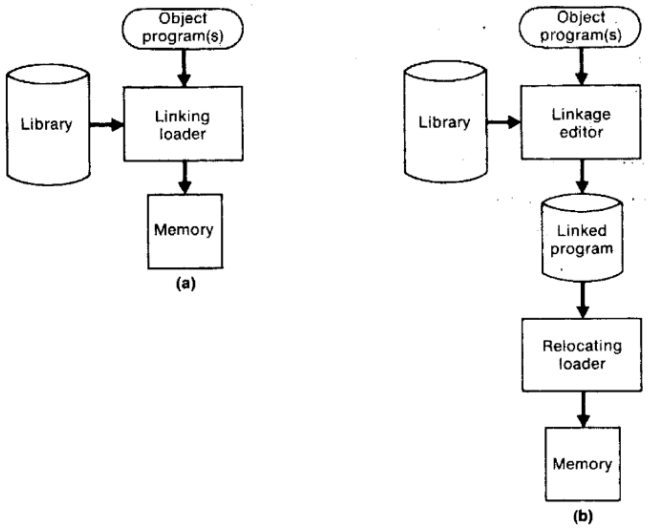
Pass 2:

```
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 // the next control section
    end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

Q6a) Explain the following loader design options i)Linkage Editor ii)Dynamic Linking

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



Linking Loaders – Perform all linking and relocation at load time.

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.

Q6b) Write and explain an algorithm for absolute loader

1. Design of an Absolute Loader

This loader does not need to perform linking and relocation, its operation is very simple.

The Header Record is checked to verify that the correct program has been

presented for loading (and that it will fit into the available memory).

As each Text Record is read, the object code it contains is moved to the

indicated address in the memory.

When the End Record is encountered the loader jumps to the specified

address to begin execution of the loaded program.

Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form.

Algorithm for an absolute loader

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

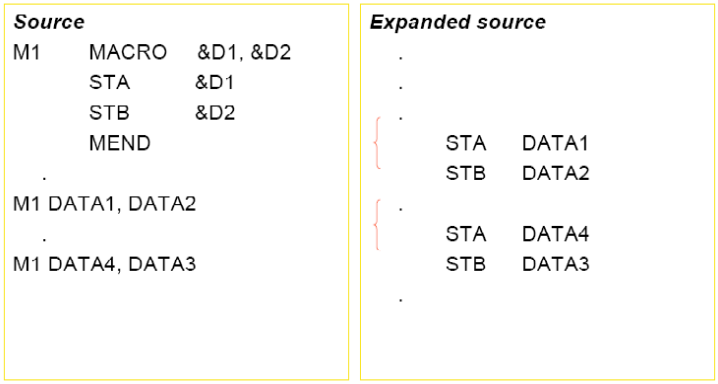
end

jump to address specified in End record

end

Q7a) Explain Macro Definition and Expansion.

Macro Definition and Expansion: The figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.
M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.



The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.
The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

Q7b) Explain Macro Processor algorithm and data structure

Data Structures

DEFTAB (Definition Table)

- Stores the macro definition including macro prototype and macro body
- Comment lines are omitted.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

NAMTAB (Name Table)

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

ARGTAB (Argument Table)

 Stores the arguments according to their positions in the argument list.

 As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.

 The figure below shows the different data structures described and their relationship.

```
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 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}
```

```
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}
```

Q8a) Explain the following: i)Concatenation of macro parameters ii) Keyword macro parameters

i)Concatenation of macro parameters

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→1

```
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
```

ii) Keyword Macro Parameters

Positional parameter:

 parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement

 if argument is to be omitted, null value should be used.

 Not suitable if a macro has a large number of parameters and only few of them has values

Keyword parameters:

 each argument value is written with a keyword that named the corresponding parameter

 Arguments may appear in any order.

 Null arguments no longer need to be used.

 It is easier to read and much less error-prone than the positional method.

Each parameter name is followed by an equal sign, which identifies a keyword parameter

The parameter is assumed to have the default value if its name does not appear in the macro invocation statement

Q8b) Explain the recursive Macro Expansion

We have seen an example of the definition of one macro instruction by another. But we have not dealt with the invocation of one macro by another.

The following example shows the invocation of one macro by another macro.

Problem of Recursive 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.

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.

The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows:

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin.

The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like at the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE.

Thus the macro processor would „forget“ that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

Q9a) Explain recursion descent parsing. Write recursive descent parse for 'READ' statement

A top-down method which is known as recursive descent is made up of procedures for each non-terminal symbol in the grammar.

When a procedure is called, it attempts to find a substring of the input, beginning with the current token that can be interpreted as the non-terminal with which the procedure is associated.

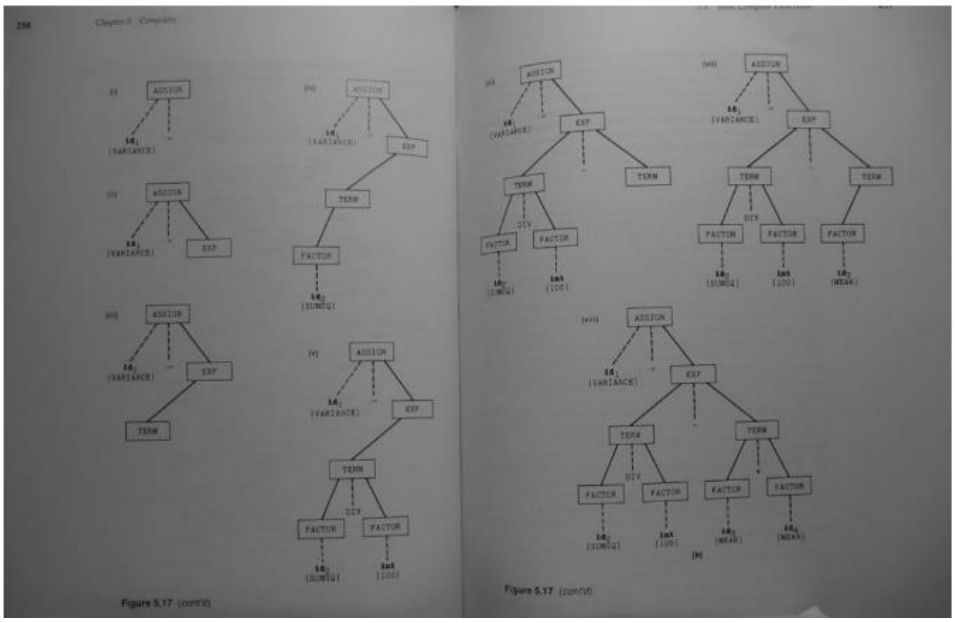
In the process of doing this, it may call other procedures or even call itself recursively, to search for other non-terminals. If a procedure finds the non-terminal that is its goal, it returns an indication of success to its caller. It also advances the current-token pointer past the substring it has just recognized.

If the procedure is unable to find a substring that can be interpreted as the desired non-terminal it returns and indication of failure or invokes an error diagnosis and recovery routine.

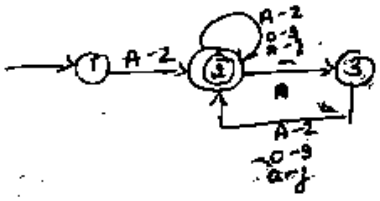
The procedure is only slightly more complicated when there are several alternatives defined by the grammar for a non-terminal. In that case, the procedure must decide which of the alternatives to try. For the recursive descent technique, it must be possible to decide which alternative to use by examining the next input token. There are other top-down methods that remove this requirement; however, they are not as efficient as recursive descent.

Example: consider following rule of grammar.

<assign>:= id:=<exp>



Q9b) Indicate whether the finite automation given recognize the following strings



i)9Alpha Not recognized

ii)Num-2 Recognized

iii)-Hello Not recognized

iv) aaa-8- Not recognized

Q9c) Define the following terms: i) Grammer ii)Lexical analysis

Grammar

A grammar for programming language is formal description of the syntax, or form, of programs and individual statements written in the language. The grammar does not describe the semantics or meaning of the various statements; consider the two statements.

I := J+K

And

X := Y+I

Where X and Y are REAL variables and I, J, K are INTEGER variables. These two statements have identical syntax. However, the semantics of the two statements are quite different. First one specifies integer arithmetic operation. Second one specifies floating point addition. However they will be described same way by the grammar.

A number of different notation can be used for writing grammars. One of the notation is BNF(for Backus-Naur Form). BNF is not the most powerful syntax description tool available, but have the advantage of being simple and widely used, and it provides capabilities that are sufficient for most purposes.

A BNF grammar contains set of rules each of which defines the syntax of some construct in the programming language.

Example:

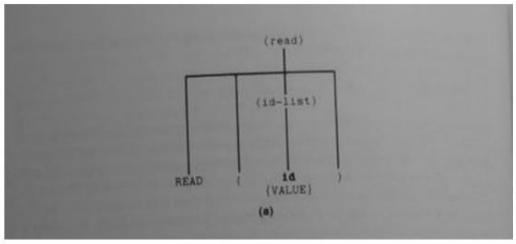
i) **<read> ::= READ(<id-list>)**

Symbol ::= can be read "is defined to be".

Character string enclosed between the angle brackets < and > are called nonterminal symbols.

Entries not enclosed in angle brackets are terminal symbols.

Blank spaces are included only to improve readability.



Lexical Analysis

Lexical analysis involves scanning the program to be compiled and recognizing the tokens that make up the source statements. Scanners are usually designed to recognize keywords, operators, and identifiers as well as integers, floating-point numbers, character strings, and other similar items that are written as part of the source program. The exact set of tokens to be recognized, of course, depends upon the programming language being compiled and the grammar being used to describe

it.

For example, an identifier might be defined by the rules.

<ident> ::= <letter>|<ident> <letter>|<ident><digit>

<letter> ::= A|B|C|D|....|Z

<digit> ::= 0|1|2|3|....|9

In such case the scanner would recognize as tokens the single characters A,B,0,1 and so on. The parser would interpret a sequence of such characters as the language construct <ident>. However, this approach would require the parser to recognize simple identifiers using general parsing techniques. A special purpouse routine can perform this same function much more efficiently.

The output of the scanner consists of a sequence of tokens, For efficiency of later use, each token is usually represented by some fixed-length code, such as integer, rather than a variable length character string.

When the token being scanned is a keyword or an operator, such coding scheme gives sufficient information. In the case of identifier, however, it is also necessary to specify the particular identifier name that was scanned. The same is accomplished by associating a token specifier with the type code for such tokens.

This specifier gives the identifier name, integer value etc., that was found by the scanner.

In addition to its primary function of recognizing tokens, the scanner usually is responsible for reading the lines of the source program as needed, and possibly for printing the source listing.

Comments are ignored by the scanner, except for printing on the output listing so they are effectively removed from the source statements before parsing begins.

Q10 a) By using the BNF grammar below represent the syntax analysis of the PASCAL statement.

VAR := SUMSQ DIV 100 – MEAN * MEAN in the parse tree.

<assign> :: = id := <exp>

<exp> :: = <term> | <exp> + <term> | <exp> – <term>

<term> :: = <factor> | <term> * <factor> | <term> DIV <factor>

<factor :: = id | int | (<exp>)

<assign> ::=id := <exp>

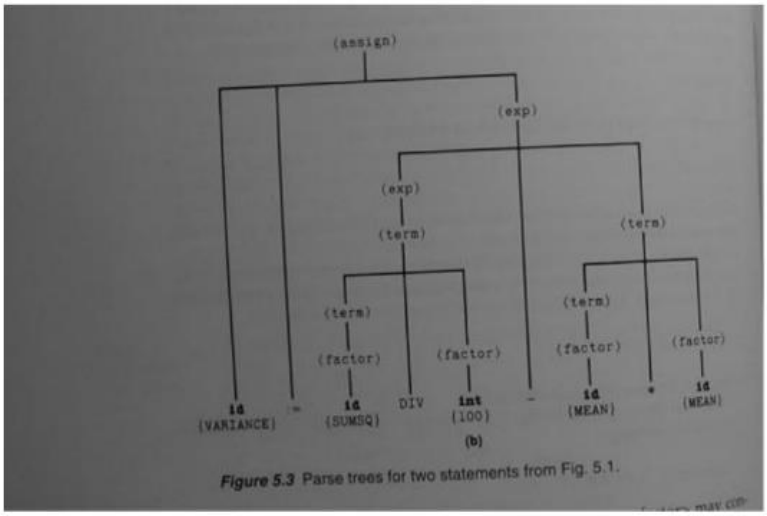


Figure 5.3 Parse trees for two statements from Fig. 5.1.

Q10b) Briefly discuss different machine dependent code optimization techniques.

1) Assignment and use of registers

General purpose register are used for various purpose like storing values or intermediate result or for addressing (base register, index register).

Registers are also used as instruction operands. Machine instructions that use registers as operands are usually faster than the corresponding instruction that refer to location in memory. Therefore it is preferable to store value or intermediate results in registers.

There are rarely as many registers available as we would like to use. The problem then becomes one of selecting which register value to replace when it is necessary to assign a register for some other purpose.

One approach is to scan the program and the value that is not needed for longest time will be replaced. If the register that is being reassigned contains the value of some variable already stored in memory, the can value can be simply discarded. Otherwise this value must be saved using temporary variable

Second approach is to divide the program into basic blocks. A basic block is a sequence of quadruples with one entry point, which is at the beginning of the block, one exit point, which is at the end of the block and no jumps within the block. When control passes from one block to another all the values are stored in temporary variables.

2) Rearranging quadruples before machine code is generated.

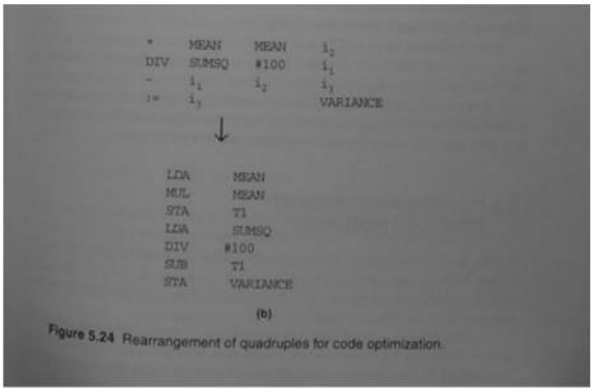


Figure 5.24 Rearrangement of quadruples for code optimization.

With a little analysis, an optimizing compiler could recognize this situation and rearrange the quadruples so the second operand of the subtraction is computed first. The resulting machine code requires two fewer instructions and uses only one temporary variable instead of two.

3) Taking advantage of specific characteristics and instructions of the target machine

For example there may be special loop-control instructions or addressing modes that can be used to create more efficient object code.

On some computers there are high level machines instructions that can perform complicated functions such as calling procedures and manipulating data structures in single operations.

Use of such feature can greatly improve the efficiency of the object program.

CPU is made of several functional units. On such system machine instruction order can affect speed of execution. Consecutive instructions that require different functional unit can be executed at the same time.