CBCS Scheme

USN | | | | | | | | | |

16MCA25

Second Semester MCA Degree Examination, June/July 2017
System Software

Time: 3 hrs. Max. Marks: 80

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

Module-1

1 a. Compare system software and application software. Give examples for each. (04 Marks)

b. With reference to SIC/XE machine architecture, explain instruction formats and addressing modes clearly indicating the settings of different flag bits. (08 Marks)

c. Give the target address generated for the following instructions (hexadecimal), if (B) = 006000, (PC) = 003000, (X) = 000090 : i) 75101000, ii) 032026. (04 Marks)

OR

2 a. Explain the following with an example for each: i) WORD, ii) START. (04 Marks)

b. With reference to VAX machine architecture, explain memory, registers, data formats and instruction formats. (08 Marks)

c. ALPHA is an array of 100 words. Write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0. (04 Marks)

Module-2

3 a. Distinguish between a literal and an immediate operand with an example for each. (04 Marks)

b. Generate the complete object program for the following assembly language program, clearly showing the symbol table entries. All addresses are in hexadecimal.
Assume: LDT - 74, LDX - 04, LDCH - 50, STCH - 54, TIXR - B8, JLT - 38, X - 1, T - 5.
ASCII character codes (decimal): E - 69, O - 79, F – 70.

```
COPY     START   0
FIRST    LDT     #3
         LDX     #0
MOVECH   LDCH    STR1, X
         STCH    STR2, X
         TIXR    T
         JLT     MOVECH
STR1     BYTE    C ‘EOF’
STR2     RESB    3
         END     FIRST
```

(08 Marks)

c. Explain the features of MASM assembler. (04 Marks)

OR

4 a. What is a relocatable program? Explain the concept of program relocation with an example, and the means for implementing it. (04 Marks)

b. What are program blocks? Mention the relevant assembler directives used in writing SIC/XE source program involving program blocks and hence briefly discuss how are they handled by an assembler. Give example. (08 Marks)

c. Compare a two-pass assembler with a one-pass assembler. Bring out the difference involved in handling forward references. (04 Marks)

Module-3

5 a. Briefly explain a simple boot strap loader with an algorithm or a source program. **(08 Marks)**
b. Distinguish between a linkage editor and a linking loader. **(05 Marks)**
c. Enlist any three loader options specified using a command language. **(03 Marks)**

OR

6 a. Discuss the detailed design of a linking loader with an example. **(08 Marks)**
b. Explain dynamic linking with suitable diagrams. **(05 Marks)**
c. Enlist the different types of SunOS linkers and the associated output modules produced. **(03 Marks)**

Module-4

7 a. Mention the basic functions of a macroprocessor. **(03 Marks)**
b. Discuss with a suitable example, the usage of various data structures in handling an assembly language program involving macros. **(08 Marks)**
c. Explain the ANSI C macro language with examples. **(05 Marks)**

OR

8 a. With an example briefly explain keyword macro parameters. **(03 Marks)**
b. Expand the following macro invocation statements using the macro definition given below.
i) RDBUFF F1, BUFFER, (04, 12), LENGTH
ii) RDLOOP RDBUFF F2, BUFF, , LEN

```
RDBUFF  MACRO   &INDEV, &BUFADR, &EOR, &RECLTH
&EORCT  SET     %NITEMS (&EOR)
        CLEAR   X
        CLEAR   A
        +LDT    #4096
$LOOP   TD      =X '&INDEV'
        JEQ     $LOOP
        RD      =X '&INDEV'
&CTR    SET     1
        WHILE   (&CTR  LE   &EORCT)
        COMP    = X '0000&EOR[&CTR]'
        JEQ     $EXIT
&CTR    SET     &CTR + 1
        ENDW
        STCH    &BUFADR, X
        TIXR    T
        JLT     $LOOP
$EXIT   STX     &RECLTH
        MEND
```

(08 Marks)

c. Mention the advantages of general purpose macro processors. Discuss the details that must be considered while designing a general purpose macroprocessor. **(05 Marks)**

Module-5

9 a. What is a grammar? Using the BNF grammar below, represent the syntax analysis of the PASCAL statement VAR := SUMSQ DIV 100 – MEAN * MEAN in the form of a parse tree.

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

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

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

<factor> :: = id | int | (<exp>) **(07 Marks)**

b. Explain the various types of machine-independent code optimization techniques. **(05 Marks)**

c. Indicate whether the finite automation given in Fig.Q9(c) recognizes the following strings.

i) 9Alpha ii) Num_2 iii) Hello_ _ world iv) bbb_9_

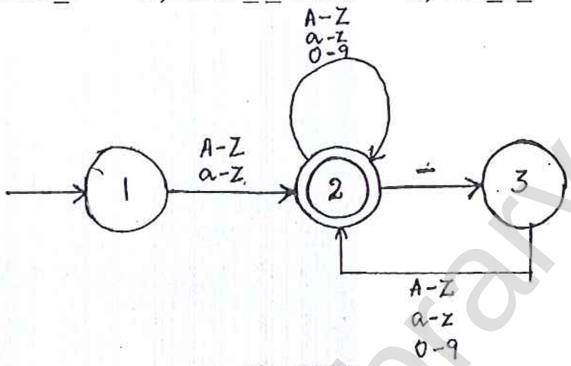


Fig.Q9(c) **(04 Marks)**

OR

10 a. Write the recursive-descent parse for a READ statement and show the corresponding syntax tree constructed for the statement READ (VALUE). The BNF grammar is given by the following:

<read> :: = READ (<id_list>)

<id_list> :: = id { , id} **(07 Marks)**

b. Explain P-code compilers with a neat diagram. **(05 Marks)**

c. Assume the array A is declared A:ARRAY{1..5, 1..6] OF INTEGER with each element occupying 3 bytes. Generate quadruples for the statement A[I, J] : = 0. **(04 Marks)**

* * * * *

Module 1

Q1

a. Compare system software and application software. Give example for each. (4 Marks)

<u>Answer:</u>

System Software	Application Software
Intended to support the operation and use of the computer	An application program is primarily concerned with the solution of some problem, using the computer as tool
Focus is on the Computer system and not on the application	The focus is on the application not on the computing system.
It depends on the structure of the machine on which it is executed.	It does not depend on the structure of the machine it works
Ex. Operating system, Loader, Linkers, assembler, compiler, text editors etc.	Ex. Banking system, Inventory system.

Q2 b. With reference to SIC/XE machine architecture, explain instruction format and addressing modes clearly indicating the settings of different flags.

<u>Answer:</u>

<u>Instruction Formats</u>

- 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)

8
op

Example RSUB (return to subroutine)

opcode
0100 1100
4 C

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)

0100 10	1	1	0	0	0	1	0000 0001 0000 0011 0110
	n	i	x	b	p	e	

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)

Parentheses are used to indicate the contents of a register or a memory location. For example, (X) represents the contents of register X.

Direct addressing mode

Example LDA TEN

0000 0000	1	001 0000 0000 0000
0 0		1 0 0 0
Opcode	x	TEN

Effective address (EA) = 1000

Content of the address 1000 is loaded to Accumulator.

Indexed addressing mode

Example STCH BUFFER, X

0101 0100	1	001 0000 0000 0000
5 4		1 0 0 0
Opcode	x	BUFFER

Effective address (EA) = 1000+[X]

= 1000+content of the index register X

The Accumulator content, the character is loaded to the effective address.

Q1 c. Give the target address generated for the following instructions (hexadecimal), if (B)=006000, (PC)=003000, (X)=000090 : i) 75101000, ii)032026.

<u>**Answer**</u>

i)75101000

0111 01	0	1	0	0	0	1	0000 0001 0000 0000 0000

n=0, i=1, x=0, b=0, p=0, e=0

since n=0 and i=1 its immediate addressing

TA= Operand Value
TA= $(01000)_{16}$
TA= $(4096)_{8}$

ii) 032026

000 00	1	1	0	0	1	0	0000 0010 0110

n=1, i=1, x=0, b=0, p=1, e=0

TA= (PC) + disp
TA=003000+026
TA=003026

Q2

a. Explain the following with an example for each: i) WORD, ii) START

Answer

i) WORD

Generate one-word integer constant
Example:
THREE WORD 3

ii) START

START specify the name and starting address of the program.
Example:
START 1000

Q2 b. With reference to VAX machine architecture, explain memory, register, data formats and instruction formats.

VAX family of computers was introduce by Digital equipment corporation (DEC) in 1978. Memory : The VAX memory consists of 8-bit bytes. 2 consecutive bytes form word, 4 consecutive bytes form long word, 8 consecutive bytes form quad ward, and 16 consecutive bytes form an octaword. All VAX programs operate in a virtual address space of 232 bytes.

Registers: There are 16 general purpose registers on the VAX, denoted by Ro to R15, all are 32 bits in length. R15 is program counter, R14 is stack pointer, R13 is frame pointer, R12 is argument pointer, R11to R6 have no special functions and R0 to R5 are available for general use.

Data Formats: Integers are stored as binary numbers in byte, word, longword, quadword or octaword. 2's compliment representation is used for negative values. Characters are stored using their 8-bit ASCII codes. There are four different floating point data formats on the VAX, ranging in length from 4 to 16 bytes.

Instruction Format: VAX machine instruction uses a variable- length instruction format. Each instruction consist of an operation code (1 or 2 bytes) followed by up to six operand specifiers, depending on the type of instruction.

Addressing mode: VAX provide large number of addressing modes. register mode, register deferred mode, auto increment and auto decrement modes, several base relative addressing modes program-counter relative modes ,indirect addressing mode (called deferred modes) ,immediate operands

Instruction Set : Goal of the VAX designers was to produce an instruction set that is symmetric with respect to data type. The instruction mnemonics are formed by a prefix that specifies the type of operation ,a suffix that specifies the data type of the operands, a modifier that gives the number of operands involved

Input and Output: Input and output on the VAX are accomplished by I/O device controllers Each controller has a set of control/status and data registers, which are assigned locations in the physical address space (called I/O space)

Q2 c. ALPHA is an array of 100 words. Write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0.

Answer

```
            LDA         #0
            LDS         #3
            LDT         #300
            LDX         #0
LOOPA       STA         ALPHA,X
            ADDR        S,X
            COMP        X,T
            JLT         LOOPA

ALPHA       RESW        100
```

Module-2

Q3 a. Distinguish between a literal and an immediate operand with an example for each.

Answer

Literal

- with literals the assembler generates the specified value as the constant at some other memory location. The address of this generated constant is used as target address of machine instruction.

- Example:

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

Immediate operand

- With immediate addressing operand value is assembled as a part of instruction
- Example:

```
LDA   #9
```

Q3 b. Generate the complete object program for the following assembly language program, clearly showing the symbol table entries. All address are in hexadecimal.

Assume: LDT – 74, LDX- 04, LDCH-50, STCH-54, TIXR-B8, JLT-38, X-1, T-5. ASCII character codes(decimal):E-69,O-79, F-70

```
COPY      START     0
FIRST     LDT       #3
          LDX       #0
MOVECH    LDCH      STR1, X
          STCH      STR2,X
          TIXR      T
          JLT       MOVECH
STR1      BYTE      C ’EOF’
STR2      RESB      3
          END       FIRST
```

Answer

LOC	**Source Statement**			**Object Code**
0000	COPY	START	0	
0000	FIRST	LDT	#3	750003
0003		LDX	#0	050000
0006	MOVECH	LDCH	STR1, X	53A008
0009		STCH	STR2,X	57A010

000C		TIXR	T	B850
000E		JLT	MOVECH	3B2FF5
0011	STR1	BYTE	C ’EOF’	697970
001C	STR2	RESB	3	
0027		END	FIRST	

Q3 c. Explain the features of MASM assembler.

Answer

- It supports a wide variety of macro facilities and structured programming idioms, including high-level constructions for looping, procedure calls and alternation (therefore, MASM is an example of a high-level assembler).
- MASM is one of the few Microsoft development tools for which there was no separate 16-bit and 32-bit version.
- Assembler affords the programmer looking for additional performance a three pronged approach to performance based solutions.
- MASM can build very small high performance executable files that are well suited where size and speed matter.
- When additional performance is required for other languages, MASM can enhance the performance of these languages with small fast and powerful dynamic link libraries.
- For programmers who work in Microsoft Visual C/C++, MASM builds modules and libraries that are in the same format so the C/C++ programmer can build modules or libraries in MASM and directly link them into their own C/C++ programs. This allows the C/C++ programmer to target critical areas of their code in a very efficient and convenient manner, graphics manipulation, games, very high speed data manipulation and processing, parsing at speeds that most programmers have never seen, encryption, compression and any other form of information processing that is processor intensive.
- MASM32 has been designed to be familiar to programmers who have already written API based code in Windows. The invoke syntax of MASM allows functions to be called in much the same way as they are called in a high level compiler.

OR

Q4 a. What is a relocatable program? Explain the concept of program relocation with an example, and the means for implementing it.

Answer

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 havig following format:

Modification record

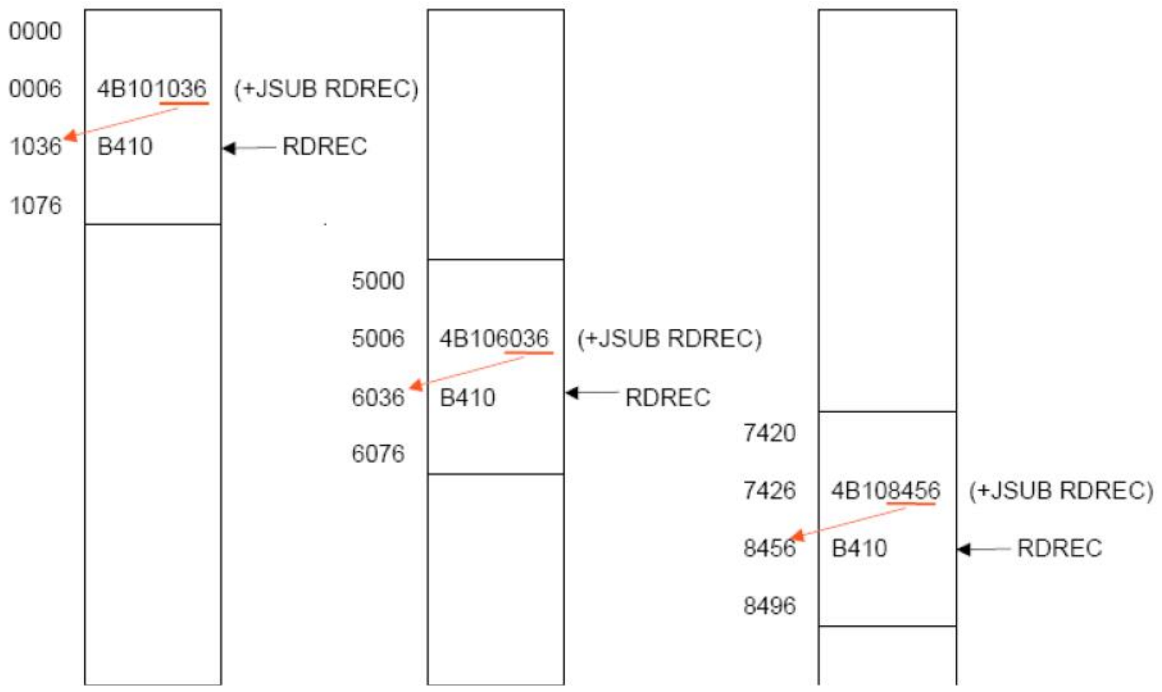
Col. 1 M

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)

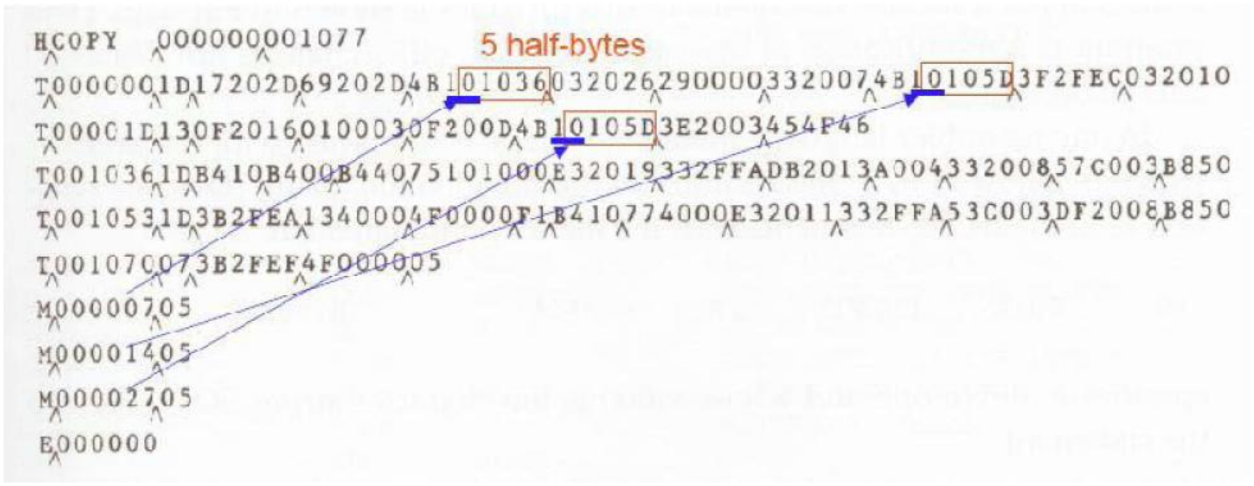
One modification record is created for each address to be modified The length is stored in half-bytes. The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in 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.


5 half-bytes

```
H^COPY ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

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.

Q4 b. What are program blocks? Mention the relevant assembler directives used in writing SIC/XE source program involving program blocks and hence briefly discuss how are they handled by an assembler. Give example.

Answer

Program being assembled was treated as one single unit and instructions appeared in same way as they were written.

Most assembler provides features that allow machine instruction and data to appear in a different order from the corresponding source program.

Other features create several independent part of the object program. These pats maintain their identity and are handled separately by the loader.

Program block refers to segment of code that are rearranged within a single object program unit and control section to refer to segments that are translated into independent object program units.

Assembler Directive USE indicate which portion of the source program belong to various blocks

```
        USE             [blockname]
```

At the beginning, statements are assumed to be part of the unnamed (default) block.

If no USE statements are included, the entire program belongs to this single block.

Each program block may actually contain several separate segments of the source program.

Assemblers rearrange these segments to gather together the pieces of each block and assign address.

Separate the program into blocks in a particular order.

Large buffer area is moved to the end of the object program.

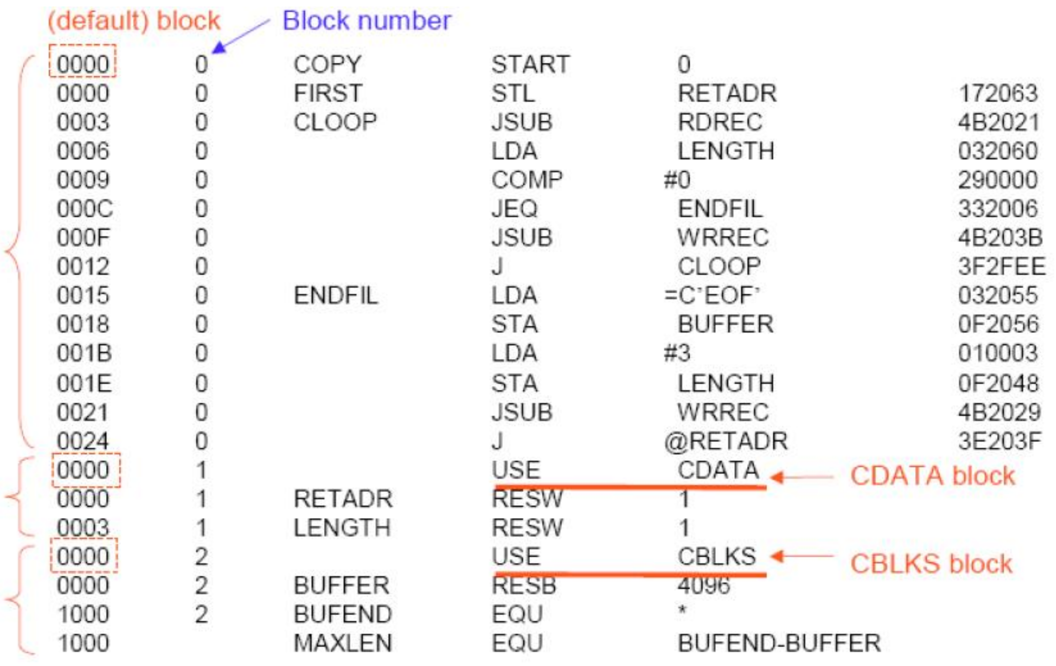
Program readability is better if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

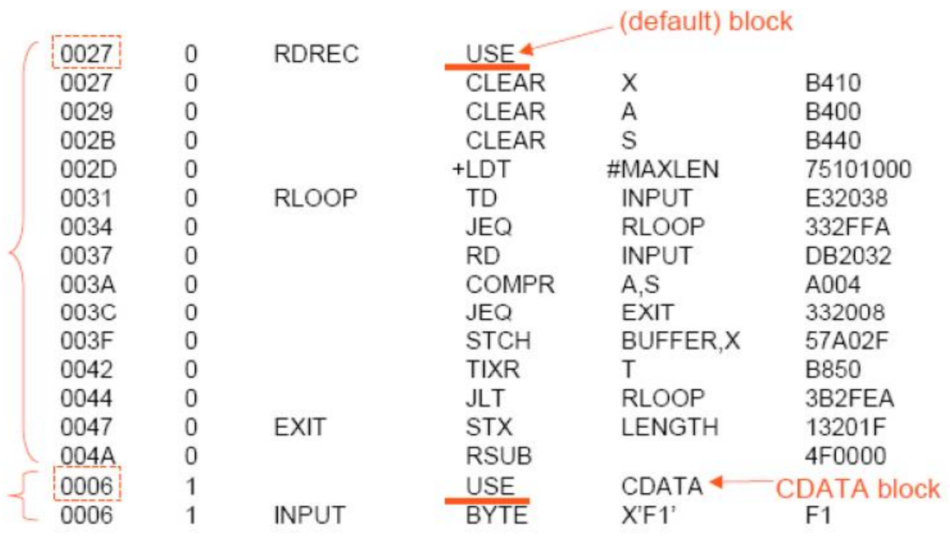
Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory.


(default) block
Block number
CDATA block
CBLKS block

```
0000   0   COPY     START    0
0000   0   FIRST    STL       RETADR       172063
0003   0   CLOOP    JSUB      RDREC        4B2021
0006   0            LDA       LENGTH       032060
0009   0            COMP     #0            290000
000C   0            JEQ       ENDFIL       332006
000F   0            JSUB      WRREC        4B203B
0012   0            J         CLOOP        3F2FEE
0015   0   ENDFIL   LDA      =C'EOF'       032055
0018   0            STA       BUFFER       0F2056
001B   0            LDA      #3            010003
001E   0            STA       LENGTH       0F2048
0021   0            JSUB      WRREC        4B2029
0024   0            J        @RETADR       3E203F
0000   1            USE       CDATA
0000   1   RETADR   RESW      1
0003   1   LENGTH   RESW      1
0000   2            USE       CBLKS
0000   2   BUFFER   RESB      4096
1000   2   BUFEND   EQU       *
1000       MAXLEN   EQU       BUFEND-BUFFER
```


(default) block
CDATA block

```
0027   0   RDREC    USE
0027   0            CLEAR    X           B410
0029   0            CLEAR    A           B400
002B   0            CLEAR    S           B440
002D   0           +LDT     #MAXLEN      75101000
0031   0   RLOOP    TD       INPUT       E32038
0034   0            JEQ      RLOOP       332FFA
0037   0            RD       INPUT       DB2032
003A   0            COMPR    A,S         A004
003C   0            JEQ      EXIT        332008
003F   0            STCH     BUFFER,X    57A02F
0042   0            TIXR     T           B850
0044   0            JLT      RLOOP       3B2FEA
0047   0   EXIT     STX      LENGTH      13201F
004A   0            RSUB                 4F0000
0006   1            USE      CDATA
0006   1   INPUT    BYTE     X'F1'       F1
```

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

- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

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

Q4 c. Compare a two pass assembler with a one-pass assembler. Bring out the differences involved in handling forward references.

Answer

The translation of the source program to the object program requires us to accomplish the following functions:

1) Convert the mnemonic operation codes to their machine language equivalent.eg translate SLT to 14.
2) Convert symbolic operands to their equivalent machine addresses. Eg translate RETADR to 1033.
3) Build the machine instructions in the proper format.
4) Convert the data constants specified in the source program into their internal machine representations in the proper format. Eg. Translate EOF to 454F46
5) Write the object program and assembly listing.

All these steps except the second can be performed by sequential processing of the

source program, one line at a time. Consider the instruction

10 1000 SRL RETADR 141033

This instruction contains the **forward reference** that is reference to a label

RETADR is defined later in the program.

If we attempt to translate the program line-by-line, we will be unable to process

this statement because we do not know the address that will be assigned to

RETADR.

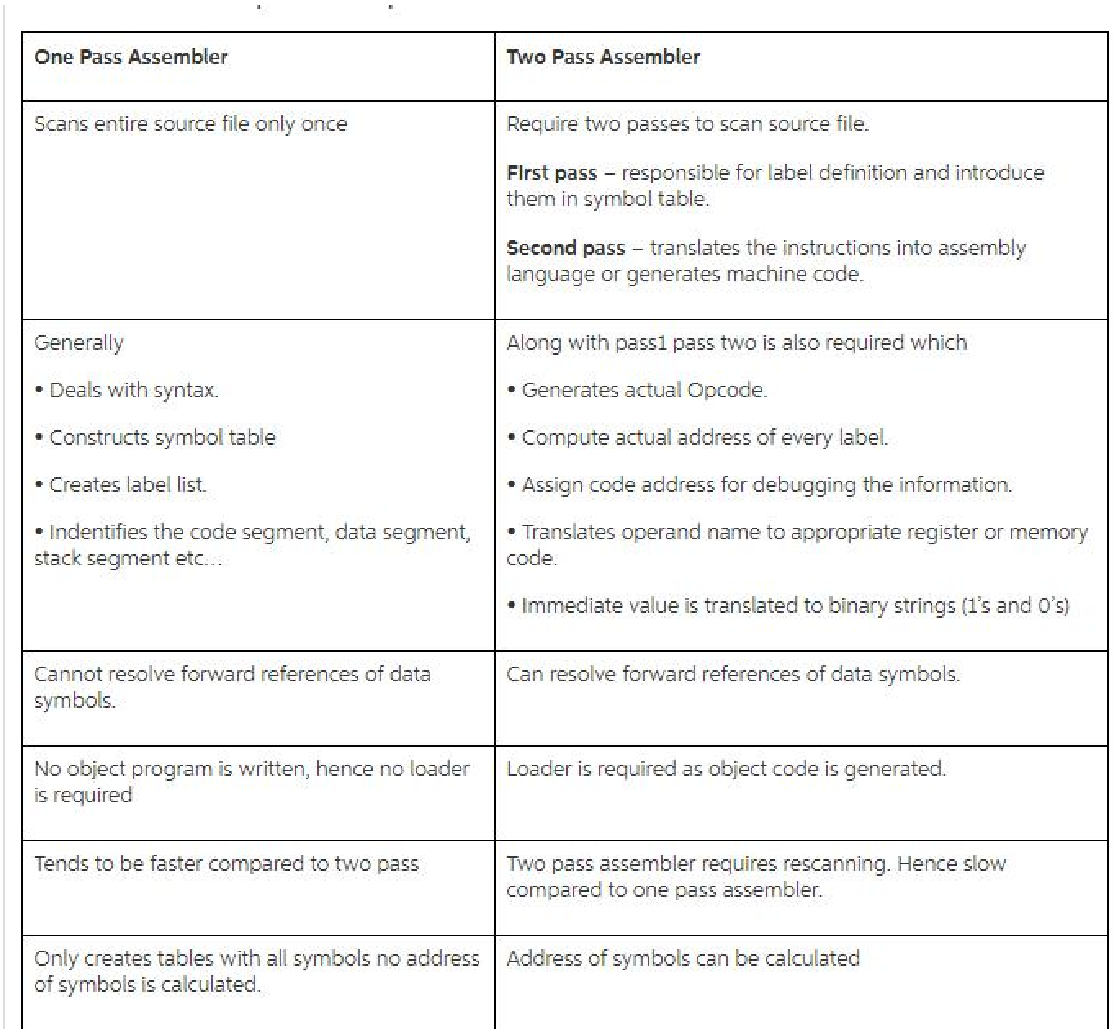
Due to this problem most of the assemblers are designed to process the program in

two passes.

First pass does little more that scan the source program for label definition and

assign addresses.

The second pas performs most of the actual translation previously described.



One Pass Assembler	Two Pass Assembler
Scans entire source file only once	Require two passes to scan source file. **First pass** – responsible for label definition and introduce them in symbol table. **Second pass** – translates the instructions into assembly language or generates machine code.
Generally • Deals with syntax. • Constructs symbol table • Creates label list. • Indentifies the code segment, data segment, stack segment etc…	Along with pass1 pass two is also required which • Generates actual Opcode. • Compute actual address of every label. • Assign code address for debugging the information. • Translates operand name to appropriate register or memory code. • Immediate value is translated to binary strings (1's and 0's)
Cannot resolve forward references of data symbols.	Can resolve forward references of data symbols.
No object program is written, hence no loader is required	Loader is required as object code is generated.
Tends to be faster compared to two pass	Two pass assembler requires rescanning. Hence slow compared to one pass assembler.
Only creates tables with all symbols no address of symbols is calculated.	Address of symbols can be calculated

Module-3

Q5 a. Briefly explain a simple bootstrap loader with an algorithm or a source program

<u>Answer</u>

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

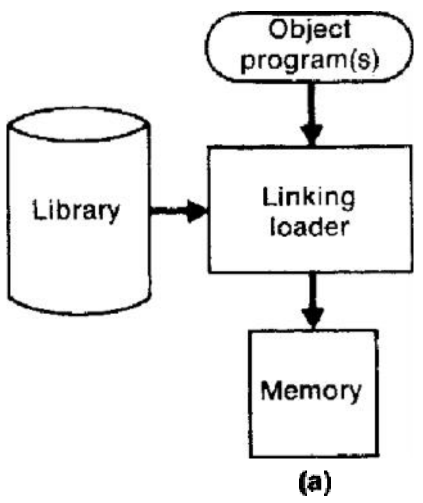
```
Begin

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
```

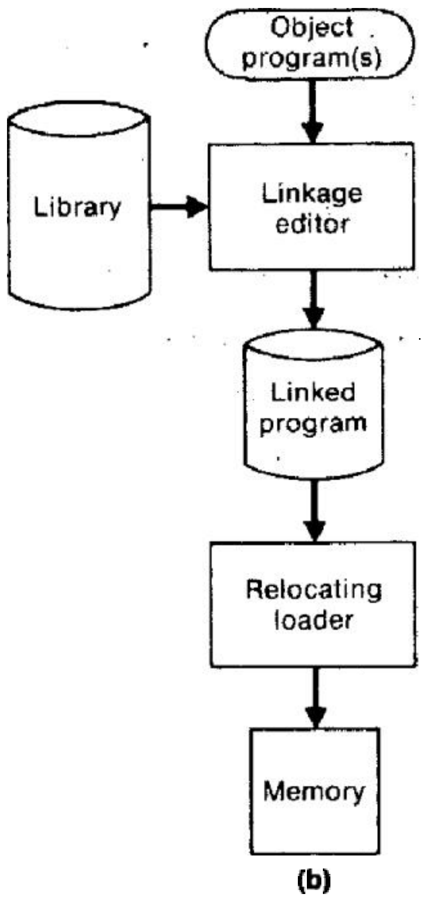
Much of the work of the bootstrap loader is performed by the subroutine GETC. This subroutine read one character from device F1 and converts it from the ASCII character code to the value of the hexadecimal digit that is represented by that character

```
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
```

Q5 b. Distinguish between a linkage editor and a linking loader.

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





Linking Loader

- Linking loader performs all linking and relocation operations including automatic library search if specified and loads the linked program directly into memory for execution
- Linking loader searched libraries and resolves external references every time the program is executed.

Linkage Editor

- A linkage editor produces a linked version of the program – often called a load module or an executable image, which is written to a file or library for later execution.

- Suitable when a program is to be executed many times without being reassembled because resolution of external references and library searching are only performed once.
- Compared to linking loader, Linkage editors in general tend to offer more flexibility and control, with a corresponding increase in complexity and overhead

Q5 c. Enlist any three options specified using a command language.

<u>Answer</u>

Here are the some examples of how option can be specified.

INCLUDE program-name (library-name) - read the designated object program from a library

DELETE csect-name – delete the named control section from the set pf programs being loaded

CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

LIBRARY MYLIB – search MYLIB library before standard libraries NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

OR

Q6 a. Discuss the detailed design of linking loader with an example.

Answer

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

Pass 1: Assign addresses to all external symbols

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

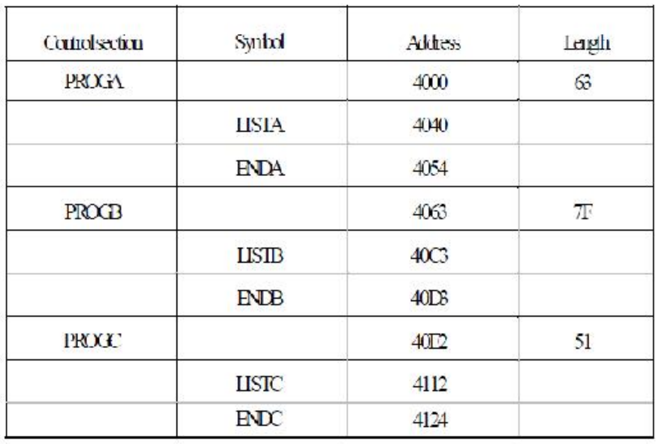
<u>Data Structures</u>

1) **External Symbol Table (ESTAB)**

This table is analogous to SYMTAB

ESTAB is used to stores the name and address of each external symbol in the set of control section being loaded.

The table also often indicates in which control section the symbol is defined. A Hashed organization is typically used for this table.



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

2) **Program Load Address (PROGADDR)**

PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the operating system.

3) **Control Section Address (CSADDR)**

CSADDR is the starting address assigned to the control section currently being scanned by the loader.

This address is added to all relative address within the control section to convert them to actual address.

<u>Algorithm</u>

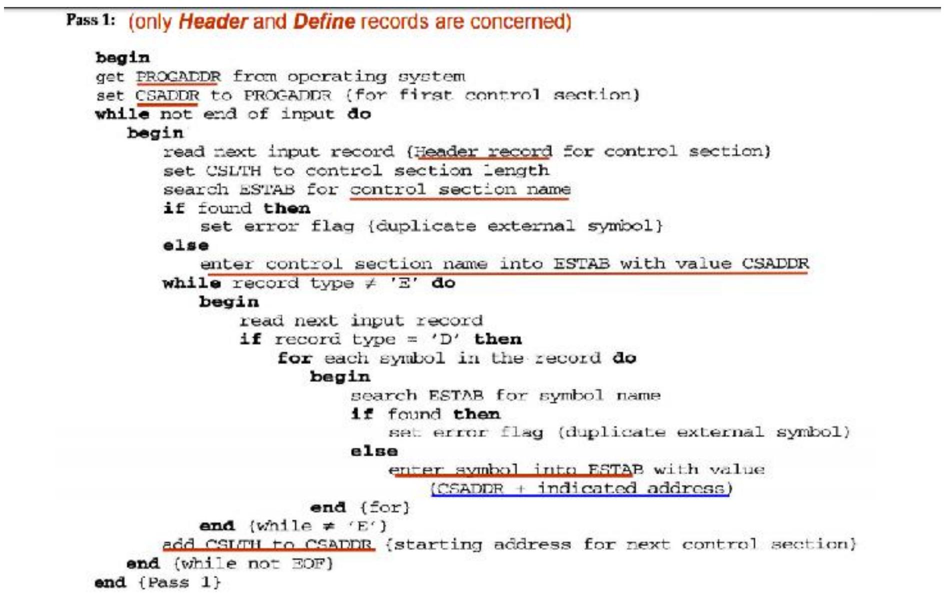
Pass 1

Pass 1 assign addresses to all external symbols.

The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB.

The pass 1 processes the Define Record.

The algorithm for Pass 1 of Linking Loader is given below.



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.

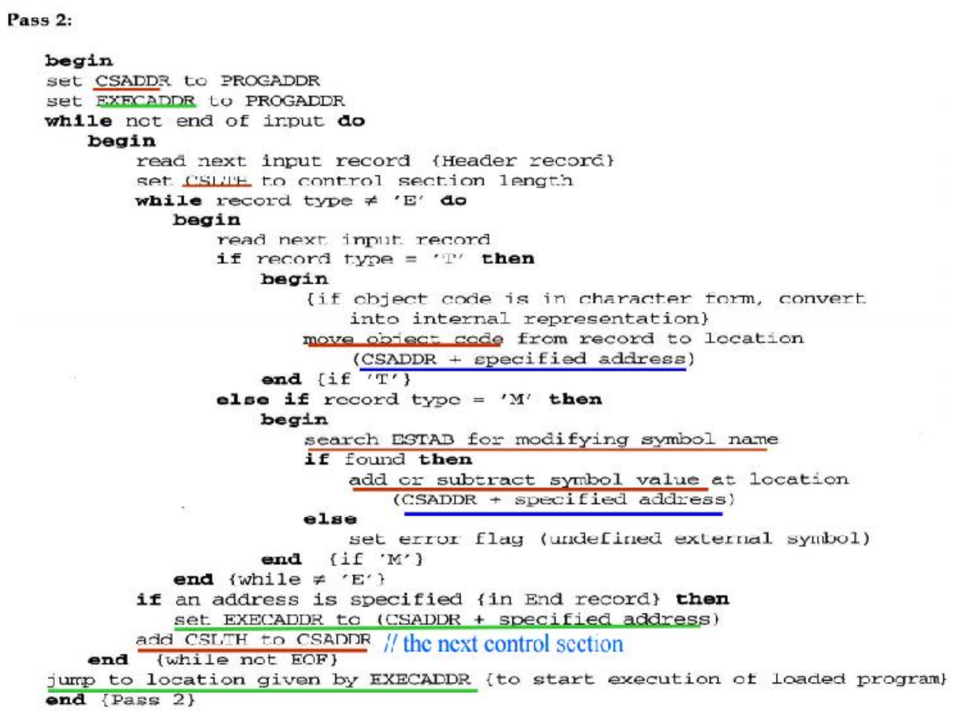
Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking.

It uses modification record and lookup the symbol in ESTAB to obtain its address.

Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program.

The pass 2 process Text record and Modification record of the object programs. The algorithm for Pass 2 of Linking Loader is given below.



```
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.

Q6 b. Explain dynamic linking with suitable diagrams.

Answer

The scheme that postpones the linking functions until execution.

A subroutine is loaded and linked to the rest of the program when it is first called.

This type of functions is usually called dynamic linking, dynamic loading or load on call.

The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library.

In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs.

Dynamic linking provides the ability to load the routines only when (and if) they are needed.

The actual loading and linking can be accomplished using operating system service request.

Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the OS.

The OS examines its internal tables to determine whether or not the routine is already loaded.

Control is then passed from the OS to routine being called.

When the called subroutine completes its processing, it returns to its caller. OS then returns control to the program that issued the request.

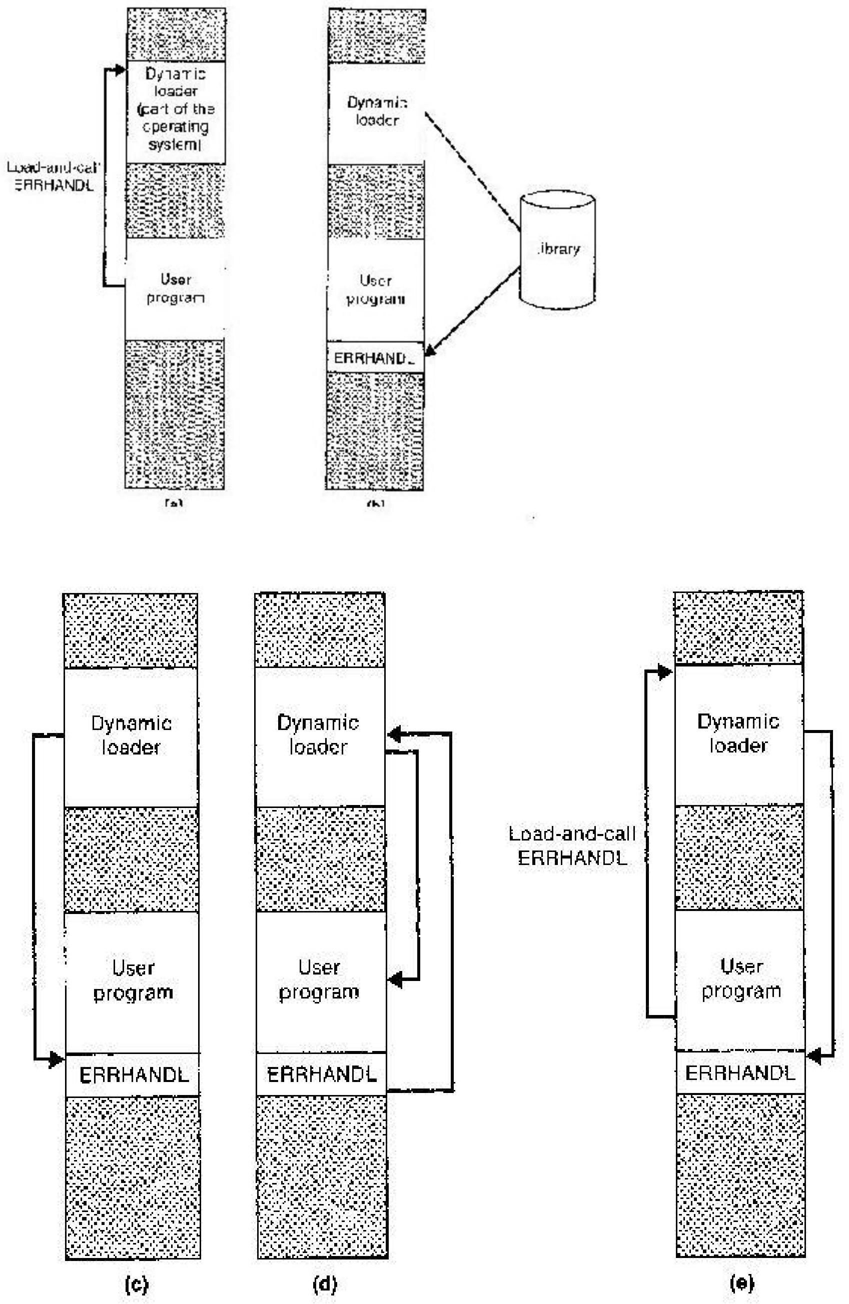


FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.

Q6 c. Enlist the different types of SunOs linkers and the associated output modules produced.

Answer

SunOS provide two different Linkers, called Link-editor and the run-time linker.

Link-editor:

Take one or more object modules (by assembler or compiler) and produce one single output module as

1) A relocatable object module, suitable for further link-editor
2) A static executable, all symbols are bound and ready to run
3) A dynamic executable, with some symbols being bounded at run time
4) A shared object, that can be bound at run time to other dynamic executables by run-time linker

Run-time linker:

Bind the shared objects with a dynamic executable, also check dependency among shared objects.

Load, re-location, and linking.

Lazing binding:

During link-editing, calls to global defined procedures are converted to references to a procedure linkage table

When a procedure is called first time at run time, it is passed to run-time linker via this table, the linker looks the actual address and place it in the call. Then in the future, the call directly goes to the procedure without via the table. Somehow similar to dynamic linking.

Module-4

Q7. a Mention the basic function of a macro processor.

Answer

Basic Macro Processor Functions

1. Macro Definition and Expansion

2. Macro Processor Algorithms and Data structures

Macro Definition and Expansion:

Two new assembler directives are used in macro definition

MACRO: identify the beginning of a macro definition

MEND: identify the end of a macro definition Prototype for the macro

Each parameter begins with ‘&’

```
Name          MACRO          parameters
                  :
```

```
            Body
               :
          MEND
```

Macro Expansion: The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statements 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.

Macro Processor Algorithm and Data structures

It is easy to design a two-pass macro processor

Pass 1: Process all macro definitions

Pass 2: Expand all macro invocation statements

However, one-pass may be enough Because all macros would have to be defined during the first pass before any macro invocations were expanded.

Q7 b. Discuss with a suitable example, the usage of various data structures in handling an assembly language program involving macros.

Answer

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.

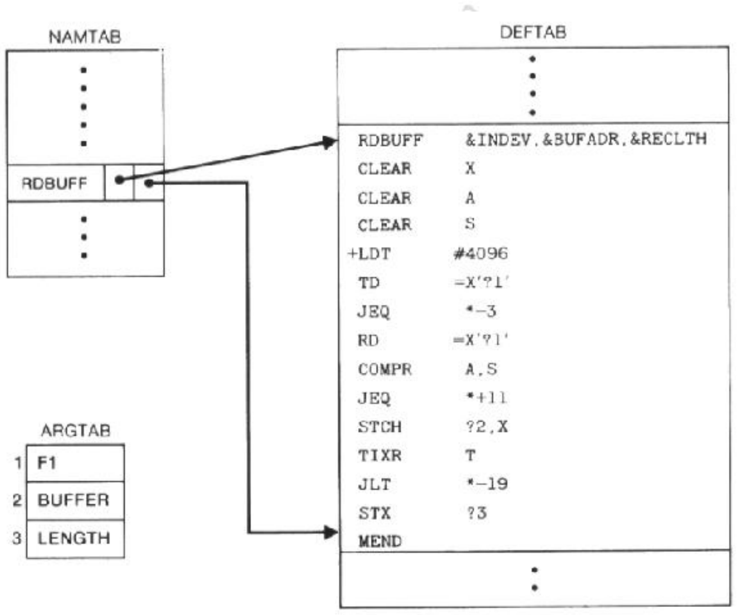
NAMTAB (Name Table)

- 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.



```
RDBUFF    &INDEV,&BUFADR,&RECLTH
CLEAR     X
CLEAR     A
CLEAR     S
+LDT     #4096
TD       =X'?1'
JEQ       *-3
RD       =X'?1'
COMPR     A,S
JEQ       *+11
STCH      ?2,X
TIXR      T
JLT       *-19
STX       ?3
MEND
```

1	F1
2	BUFFER
3	LENGTH

(b) (a)

Fig 4.4

In fig 4.4(a) definition of RDBUFF is stored in DEFTAB, with an entry in NAMTAB having the pointers to the beginning and the end of the definition. The arguments referred by the instructions are denoted by their positional notations. For example, TD =X'?1'

The above instruction is to test the availability of the device whose number is given by the parameter &INDEV. In the instruction this is replaced by its positional value? 1.

Figure 4.4(b) shows the ARTAB as it would appear during expansion of the RDBUFF statement as given below: CLOOP RDBUFF F1, BUFFER, LENGTH For the invocation of the macro RDBUFF, the first parameter is F1 (input device code), second is BUFFER (indicating the address where the characters read are stored), and the third is LENGTH (which indicates total length of the record to be read). When the ?n notation is encountered in a line fro DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

The algorithm of the Macro processor is given below. This has the procedure DEFINE to make the entry of macro name in the NAMTAB, Macro Prototype in DEFTAB. EXPAND is called to set up the argument values in ARGTAB and expand a Macro Invocation statement. Procedure GETLINE is called to get the next line to be processed either from the DEFTAB or from the file itself.

When a macro definition is encountered it is entered in the DEFTAB. The normal approach is to continue entering till MEND is encountered. If there is a program having a Macro defined within another Macro.

While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the definition of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL. Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

Q7 c. Explain the ANSI C macro language with examples.

<u>Answer</u>

- C high level language
- In ANSI C language, definations and invocations of macros are handled by a pre-processor
- For examples:
  - #define NULL 0, #define EOF (-1), #define EQ ==
  - Also called constant definition
- More complicate, macro with parameters:
  - #define ABSDIFF(x,y) ((x)>(y)? (x)-(y): (y)-(x))
  - So ABSDIFF (I+1,J-5), ABSDIFF(I,3.1415), ABSDIFF(‘D’,’A’).
  - #define ABSDIFF(x,y) x>y? x-y: y-x will result in problem.
  - Note: the importance of parentheses, due to the simple string substitution.
- Conditional compilation
  - Example1
    - #ifndef BUFFER_SIZE
    - #define BUFFER_SIZE 1024
    - #endif
  - Example 2

- #define DEBUG 1
- …
- #if DEBUG==1
- printf(….);
- #endif
- pintf(…) will appears in the program. If change #define DEBUG 0, then the printf(…) will not.
- Also #ifdef DEBUG
- printf(…);
- #endif
- In this case, #define DEBUG will have printf(…) in the program and printf(…) will not appear in program without #define DEBUG

OR

Q8 a. With an example briefly explain keyword macro parameters.

Answer

<u>Positional parameter:</u>

- 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

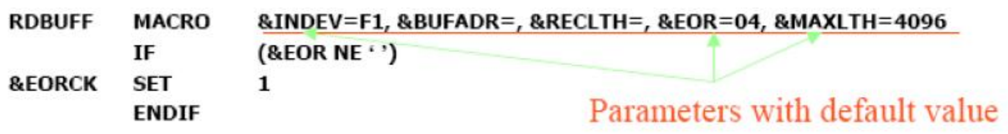
<u>Keyword parameters:</u>

- 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.

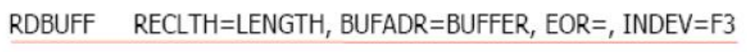
Each parameter name is followed by an equal sign, which identifies a keyword parameter

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

EXAMPLE



```
RDBUFF   MACRO   &INDEV=F1, &BUFADR=, &RECLTH=, &EOR=04, &MAXLTH=4096
         IF      (&EOR NE ' ')
&EORCK   SET     1
         ENDIF
```



```
RDBUFF   RECLTH=LENGTH, BUFADR=BUFFER, EOR=, INDEV=F3
```

Q8 b. Expand the following macro invocation statements suing the macro definition given below.

i) RDBUFF F1, BUFFER, (04, 12), LENGTH

ii) RDLOOP RDBUFF F2, BUFF, , LEN

```
RDBUFF    MACRO     &INDEV, &BUFADR,&EOR,&RECLTH
&EORCT    SET       %NITEMS(&EOR)
          CLEAR     X
          CLEAR     A
          +LDT      #4096
$LOOP     TD        =X '&INDEV'
          JEQ       $LOOP
          RD        =X '&INDEV'
&CTR      SET       1
          WHILE     (&CTR  LE  &EORCT)
          COMP      =X '0000&EOR[&CTR]'
          JEQ       $EXIT
&CTR      SET       &CTR + 1
          ENDW
          STCH      &BUFADR, X
          TIXR      T
          JLT       $LOOP
```

```
$EXIT       STX         &RECLTH
            MEND
```

Answer

1) RDBUFF F1, BUFFER, (04, 12), LENGTH

```
            CLEAR       X
            CLEAR       A
            +LDT        #4096
$AALOOP     TD          =X ‘F1’
            JEQ         $ AALOOP
            RD          =X ‘F1’
            COMP        =X ‘000004’
            JEQ         $AAEXIT
            COMP        =X ‘000012’
            JEQ         $AAEXIT
            STCH        BUFFER, X
            TIXR        T
            JLT         $AALOOP
$AAEXIT     STX         LENGTH
```

ii) RDLOOP RDBUFF F2, BUFF, , LEN

```
            CLEAR       X
            CLEAR       A
            +LDT        #4096
$ABLOOP     TD          =X ‘F2’
            JEQ         $ ABLOOP
            RD          =X ‘F2’
            STCH        BUFF, X
```

```
                TIXR        T
                JLT         $ABLOOP
$ABEXIT         STX         LEN
```

Q8 C. Mention the advantages of general purpose macro processor. Discuss the details that must be considered while designing a general purpose macro processor.

1) General-Purpose Macro Processors

Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages

Pros

- Programmers do not need to learn many macro languages.
- Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

Cons

- Large number of details must be dealt with in a real programming language Situations in which normal macro parameter substitution should not occur, e.g., comments.
- Facilities for grouping together terms, expressions, or statements Tokens, e.g., identifiers, constants, operators, keywords
- Syntax had better be consistent with the source programming language

Module-5

Q9 a. What is grammar? Using the BNF grammar below, represent the syntax analysis of the PASCAL statement VAR :=SUMSQ DIV 100 –MEAN * MEAN in the form of a parse tree.

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

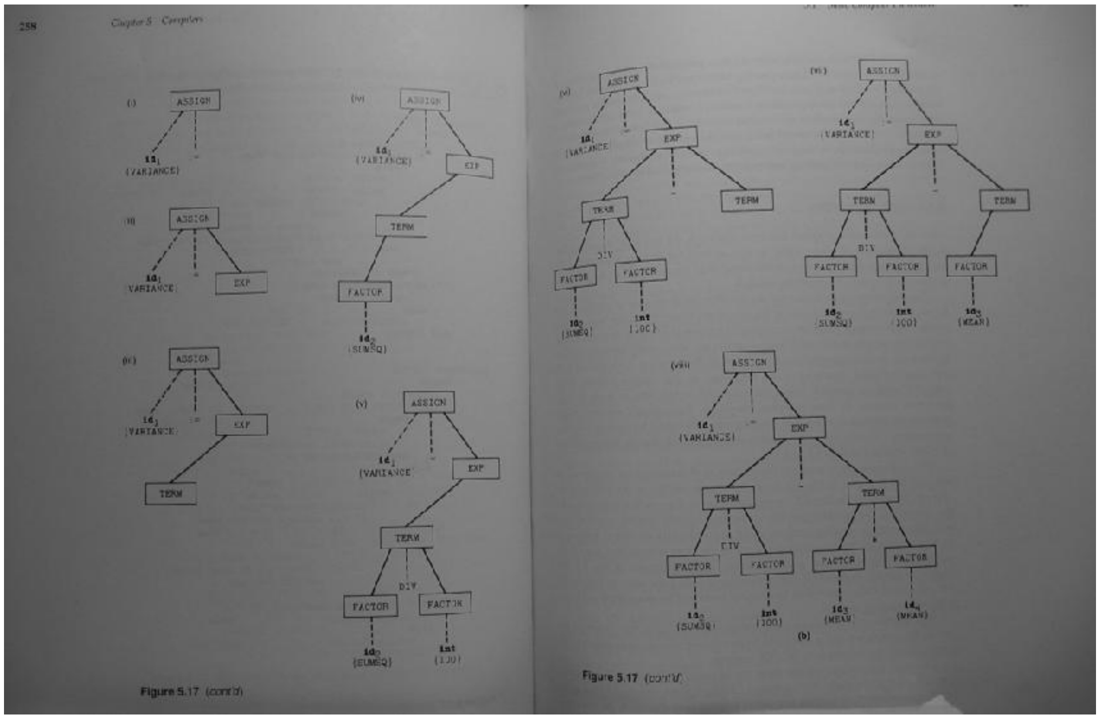
<exp> :: = <term> | <exp> + <term> | <exp> - <term>

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

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

Answer

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.



b. Explain the various types of machine-independent code optimization techniques.

Answer

<u>1) Elimination of common sub expressions.</u>

One important source code optimization is the elimination of common sub expressions. These sub expressions that appear at more than one point in the program and that compute the same value.

Common sub expressions are usually detected through the analysis of an intermediate form of the program

Example:

1)	:=	#1		I
5)	*	#2	J	i3
12)	*	#2	J	i10

We see that quadruples 5 and 12 are same except for the name of the intermediate result produced. Operand J does not change value between 5 and 12. It is not possible to reach quadruples 12 without passing through 5. This means we can delete quadruple 12 and replace any reference to its result (i10) with reference to i3, the result of quadruple 5.

This modification eliminates the duplicate calculation of 2*J, which we identified previously as a common sub expression in the source statement

2) Removal of loop variants.

These are the sub expressions within a loop whose values do not change from one iteration of the loop to the next. Thus their values can be computed once before loop is entered, rather than being recalculated for each iteration. Because most programs spend most of their running time in the execution of loops, the time saving from this sort of optimization can be highly significant.

Example

Loop-invariant computation is the term 2*j. the result of this computation depends only on the operand J, which does not change in value during the execution of the loop. Thus it can be moved to the point immediately before the loop is entered.

3) Substitution of more efficient operation for less efficient one.

Consider following example:

```
DO    10     I = 1,20
TABLE(I) = 2**I
```

This DO loop creates a table that contains the first 20 powers of 2. On closer examination, we can see that there is a more efficient way to perform the computation. For each iteration of the loop, the value of I increased by 1. Therefore, the value of 2**I for the current iteration can be found by multiplying the value for the previous iteration by 2. Clearly this method of computing 2**I is much more efficient than performing a series of multiplications or using a logarithmic technique. Such a transformation is called reduction in strength of an operation.

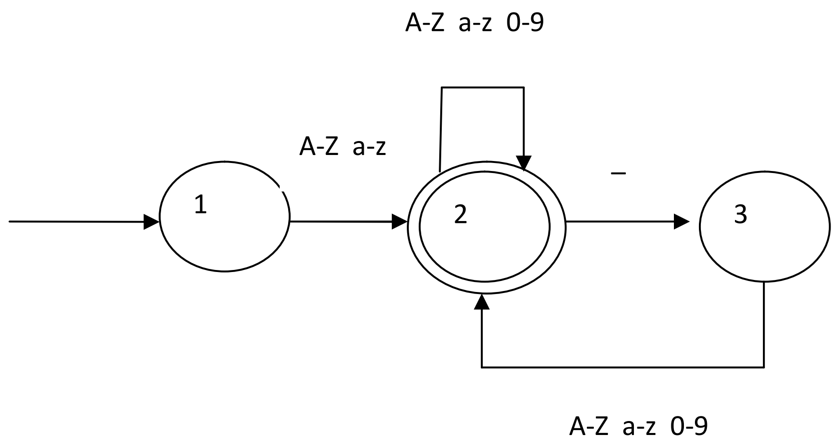
There are number of other possibilities for machine-impendent code optimization.

For example, computations whose operand values are known at compilation time can be performed by the compiler. This optimization is known as folding.

Other optimization include converting a loop into straight line code(loop unrolling) and margining of the bodies of loop (loop jamming))

c. Indicate whether the finite automation given in Fig. Q9(c) recognizes the following strings.

i) 9ALPHA ii) NUM_2 iii) HELLO_ _ word iv) bbb_9_



Answer

i) 9ALPHA - Not Recognized
ii) NUM_2 - Recognized
iii) HELLO_ _ word – Not Recognized
iv) bbb_9_ - Not Recognized

OR

Q10 a Write the recursive-decent parse for a READ statement and show the corresponding syntax tree constructed for the statement for the statement READ(VALUE). The BNF grammar is given by the following:

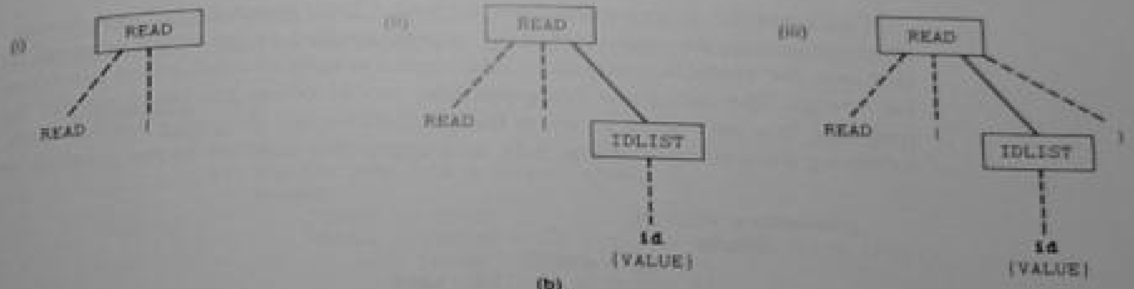
<read> ::= READ(<id_list>)

<id_list> ::= id { ,id}

```
procedure READ
    begin
        FOUND := FALSE
        if TOKEN = 8 {READ} then
            begin
                advance to next token
                if TOKEN = 20 { ( } then
                    begin
                        advance to next token
                        if IDLIST returns success then
                            if TOKEN = 21 { ) } then
                                begin
                                    FOUND := TRUE
                                    advance to next token
                                end {if ) }
                    end {if ( }
            end {if READ}
        if FOUND = TRUE then
            return success
        else
            return failure
    end {READ}

procedure IDLIST
    begin
        FOUND := FALSE
        if TOKEN = 22 {id} then
            begin
                FOUND := TRUE
                advance to next token
                while (TOKEN = 14 {,}) and (FOUND = TRUE) do
                    begin
                        advance to next token
                        if TOKEN = 22 {id} then
                            advance to next token
                        else
                            FOUND := FALSE
                    end {while}
            end {if id}
        if FOUND = TRUE then
            return success
        else
            return failure
    end {IDLIST}
```

(a)

Figure 5.16 Recursive-descent parse of a READ statement.



(b)

Figure 5.16 (*cont'd*)

b. Explain P-code compilers with a neat diagram.

Answer

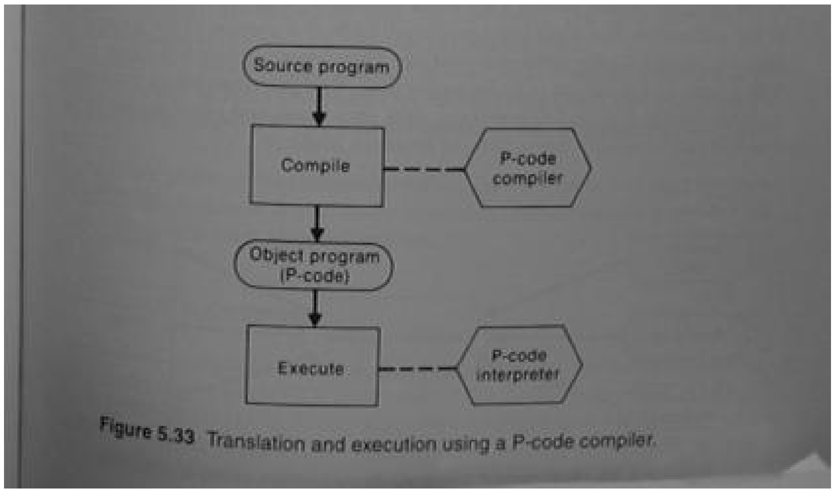


Figure 5.33 Translation and execution using a P-code compiler.

- P-code compilers (also called bytecode compilers) are very similar in concept to interpreters.
- In both cases, the source program is analyzed and converted into an intermediate form, which is then executed interpretively.
- With a P-code compiler, however, this intermediate form is the machine language for a hypothetical computer, often called pseudo-machine or P-machine.
- The source program is compiled, with the resulting object program being in P-code.
- This P-code program is then read and executed under the control of a P-code interpreter
- The main advantage of this approach is portability of software. It is not necessary for the compiler to generate different code for different computers, because the p-code object programs can be executed on any machine that has a p-code interpreter.
- Even the compiler itself can be transported if it is written in the language that it compiles. To accomplish this, the source version of the compiler is compiled into p-code; this p-code can then be interpreted on another computer.
- In this way a p-code compiler can be used without modification on a wide variety of system if a p-code interpreter is written for each different machine.
- The design of a P-machine and the associated P-code is often related to the requirements of the language being compiled.
- The interpretive execution of a p-code program may be much slower than the execution of the equivalent machine code.

- P-code object program is often much smaller than a corresponding machine-code program would be. This particularly useful n machines with severely limited memory size
- Many p-code compilers designed for a single user running on dedicated microcomputer system.
- If execution speed is important some P-code compilers support the use of machine language subroutines.
- By rewriting a small number of commonly used routines in machine language, rather than P-code, it is often possible to achieve substantial improvements in performance. But this approach scarifies some of the portability associated with the use of P-code compiler

c. Assume the array A is declared A:ARRAY{1..5, 1..6] of INTEGER with each element occupying 3 bytes. Generate quadruples for the statement A[I,J] :=0

Answer

-	I	#1	i1
*	i1	5	i2
-	J	#1	i3
+	i2	i3	i4
*	i4	#3	i5
:=	#0		A[i5]