Third semester MCA Degree Examination, Jan/Feb 2021

System Software

Q1 a) Explain the instruction formats and addressing modes of SIC/XE machine architecture.

<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	

<u>Addressing Modes</u>

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)

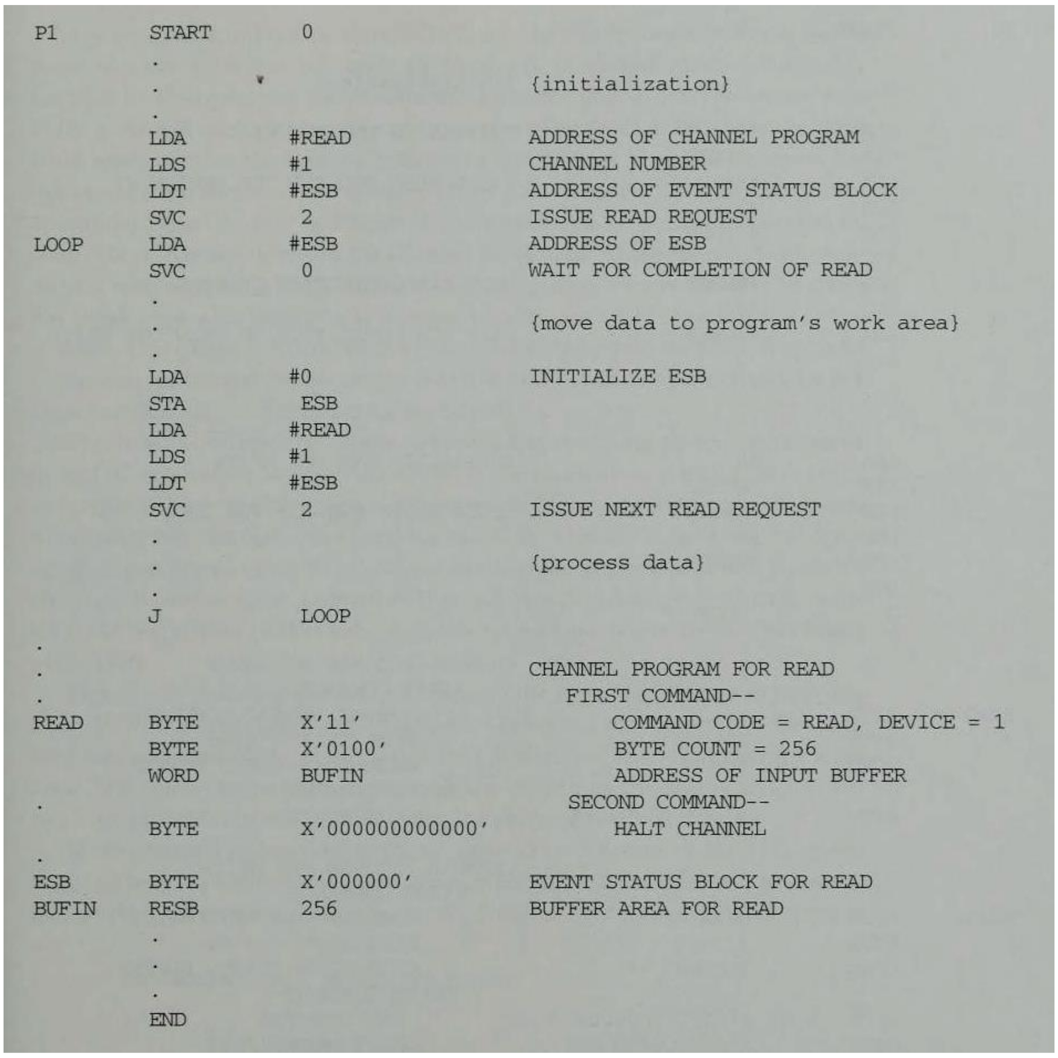
SIC/XE also support addressing modes that are assembled using Format 4.

Mode	**Indication**	**Target address calculation**
Direct	b=0, p=0, x=0	TA = disp
Indexed	x=1	TA = (x)+disp
Immediate	i=1, n=0	TA = operand value
Indirect	i=0, n=1	TA = address of operand value
simple	i=1, n=1 i=0, n=0	TA = location of the operand value

Q1 b) Explain with example simple input and output operations on SIC/XE machine architecture.

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions.

- Allows overlap of computing and I/O, resulting in more efficient system operation.
- The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.



```
P1        START      0
          .
          .                   {initialization}
          .
          LDA       #READ     ADDRESS OF CHANNEL PROGRAM
          LDS       #1        CHANNEL NUMBER
          LDT       #ESB      ADDRESS OF EVENT STATUS BLOCK
          SVC        2        ISSUE READ REQUEST
LOOP      LDA       #ESB      ADDRESS OF ESB
          SVC        0        WAIT FOR COMPLETION OF READ
          .
          .                   {move data to program's work area}
          .
          LDA       #0        INITIALIZE ESB
          STA        ESB
          LDA       #READ
          LDS       #1
          LDT       #ESB
          SVC        2        ISSUE NEXT READ REQUEST
          .
          .                   {process data}
          .
          J          LOOP
.
.                             CHANNEL PROGRAM FOR READ
.                                FIRST COMMAND--
READ      BYTE       X'11'          COMMAND CODE = READ, DEVICE = 1
          BYTE       X'0100'        BYTE COUNT = 256
          WORD       BUFIN          ADDRESS OF INPUT BUFFER
.                                SECOND COMMAND--
          BYTE       X'000000000000'    HALT CHANNEL
.
ESB       BYTE       X'000000'    EVENT STATUS BLOCK FOR READ
BUFIN     RESB       256          BUFFER AREA FOR READ
          .
          .
          .
          END
```

Q1 c) What is system software? Differentiate it from Application Software.

System Software consists of a variety of programs that support the operation of a computer. It makes possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

They are usually related to the architecture of the machine on which they are to run.

Example: Assembler, Compiler, text editor, loader and linkers etc.

<u>Comparison between System software and application software</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 a) Explain the architectures of Complex Instruction set computer (CISC) machine.

Complex Instruction Set Computers (CISC) machines have complicated instruction set, different instruction formats and lengths,and many different addressing modes. Thus the implementation of such an architecture in hardware tends to be complex.

e.g. VAX

VAX Architecture

VAX family ofcomputers was introduce by Digital equipment corporation (DEC) in 1978. VAXarchitecture was designed for the compatibility with the earlier PDP-11 machine. A compatibility mode was provided at hardware level so PDP-11 programs could run unchanged on VAX.

Memory

 The VAX memory consist of 8-bit bytes

 2 consecutive bytes form word, 4 consecutive bytes form long word, 8 consecutive bytes form quadward, and 16 consecutive bytes form an octaword.

 All VAX programs operate in a virtual address space of 232 bytes.

 One half of the VAX virtual address space is called ystem space which contains operating system and is shared by all the programs.

 The other half of the address space is called process space and and is defined separately for each program.

Registers

 There are 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 machineinstruction use 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.

 Each operand specifier designates one of the VAX addressing modes and gives any additional information necessary to locate the operand.

Addressing mode

VAX provide large number of addressing modes.

 register mode

 register deferred mode

 autoincrement and autodecrement 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*)

 No special instructions are required to access registers in I/O space

 The association of an address in I/O space with a physical register in a device controller is handled by the memory management routines

Q2 b) Explain the basic assembler functions and any six assembler directives with examples.

There are certain fundamental functions that any assembler must perform, such as:

・Translating mnemonic language code to its equivalent object code.

・Assigning machine addresses to symbolic labels used by the programmer.

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

<u>5) RESB</u>

Reserve the indicate number of bytes for a data area.

Example: BUFFER RESB 4096

<u>6) RESW</u>

Reserve the indicate number of words for a data area.

Example: LENGTH RESW 1

Q3 a) Explain program relocation with suitable example and also explain how the relocation problem is solved.

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

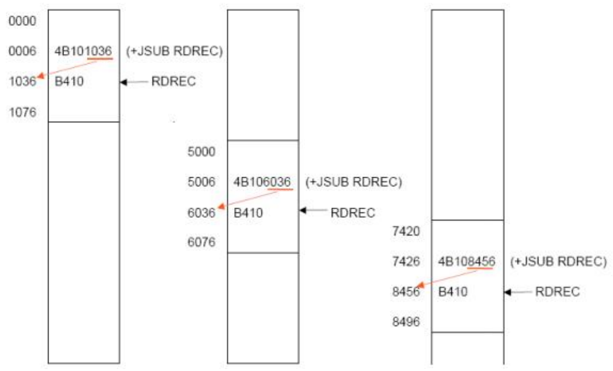
Col. 1 M

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.

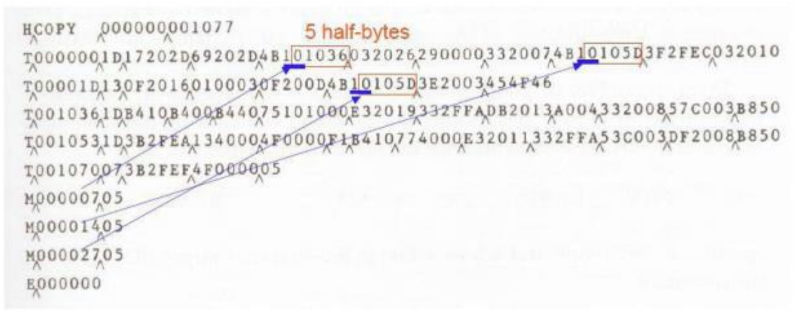
 The address field of this instruction contains 01036, which is the address of the instruction labelled RDREC. The second figure shows that if the program is to be loaded at new location 5000.

 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.



```
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.

Similarly the remaining instructions indicate.

Q3 b) Explain the following machine independent assembler features:

i) Literals ii) Symbol defining statements iii) Program blocks.

Literals

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

```
001A        ENDFIL  LDA        =C‟EOF‟
```

All the literal operands used in a program are gathered together into one or more *literal pool*s. This is usually placed at the end of the program.

The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program.

An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program.

Symbol-Defining Statements

EQU

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this is **EQU** (Equate). The general form of the statement is

Symbol EQU value

This statement defines the given symbol (i.e., enter it into SYMTAB) and assigning to it the value specified.

<u>Program blocks</u>

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

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

USE [blockname]

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

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

Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address.

Pass1

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

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

Q4 a) Explain the types of one pass assembler

The main problem in designing the assembler using single pass was to resolve forward references.

We can avoid to some extent the forward references by:

• Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.

Unfortunately, forward reference to labels on the instructions cannot be eliminated as easily. Assembler provides some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

1) One that produces object code directly in memory for immediate execution (Load-and-go assemblers).

2) The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler

• Load-and-go assembler generates their object code in memory for immediate execution.

• No object program is written out, no loader is needed.

• It is useful in a system with frequent program development and testing. The efficiency of the assembly process is an important consideration.

• Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

In load-and-Go assemblers when a forward reference is encountered:

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

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

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

• When the definition for the symbol is encountered, scans the reference list and inserts the address.

• At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.

• For Load-and-Go assembler o Search SYMTAB for the symbol named in the

END statement and jumps to this location to begin execution if there is no error

If One-Pass needs to generate object code:

• If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.

• Forward references are entered into lists as in the load-and-go assembler.

• When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.

• When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

Q4b) Write short note on the following:

i)MASM assembler   ii)Multipass assembler

ii)Multipass assembler

Consider the following example

ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined. As a result, ALPHA cannot be evaluated during second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definition.

Prohibiting forward references in symbol definition is not a serious inconvenience. Forward references tend to create difficulty for a person reading the program as well as for the assembler.

The general solution is multi pass assembler that can make has many passes are needed to process the definition of symbols.

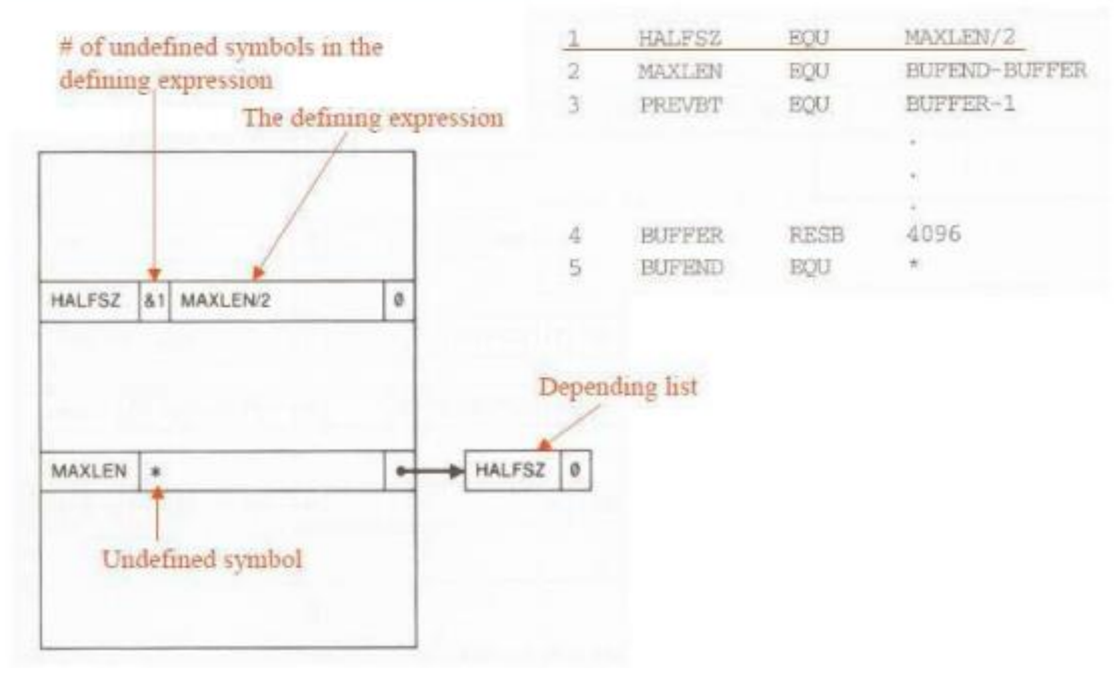
It is not necessary for such an assembler to make more than two passes over the entire program. Instead, the portions of the program that involve forward references in symbol definition are saved during pass. Additional passes through these stored definitions are made as the assembly progresses.

There are several ways to accomplish the task outlined above.

 Store those symbol definitions that involve forward references in the symbol table.

 This table also indicates which symbols are dependent on the values of others, to facilitate symbol evaluation.

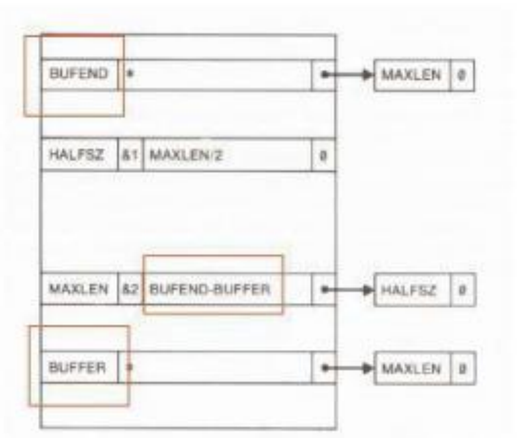
Multi-Pass Assembler Example Program



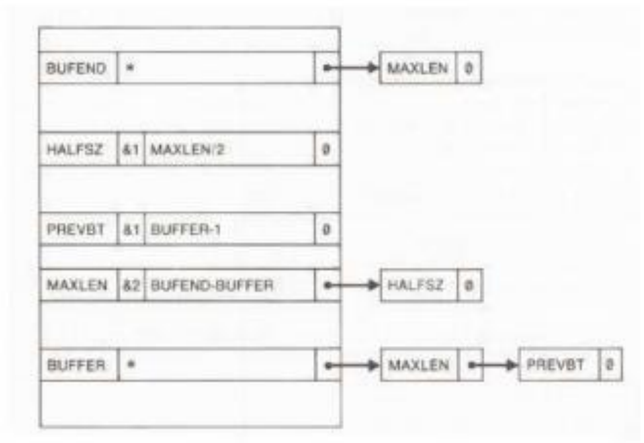
```
1   HALFSZ    EQU    MAXLEN/2
2   MAXLEN    EQU    BUFEND-BUFFER
3   PREVBT    EQU    BUFFER-1
                     .
                     .
                     .
4   BUFFER    RESB   4096
5   BUFEND    EQU    *
```

MAXLEN has not yet been defined, so no value for HALFSZ can be computed. Defining expression for HALFSZ is stored in symbol table in place of its value. The entry „&‟ indicates that one symbol in the definition of expression is undefined. The symbol MAXLEN is also entered in the symbol table with the flag

„* „ identifying it as undefined.

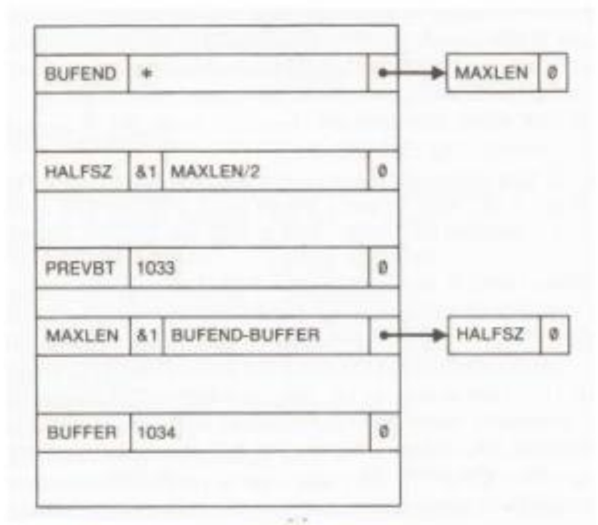


2 MAXLEN EQU BUFEND-BUFFER

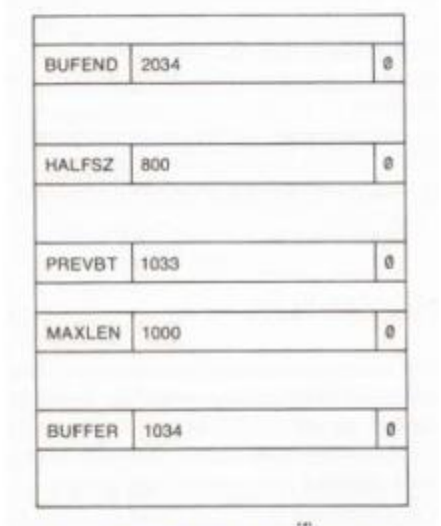


3 PREVBT EQU BUFFER-1

In this case there are two undefined symbols involved in the definition: BUFEND and BUFFER. Bothe of these are entered into SYMTAB with lists indicating the dependence of MAXLEN upon them. Similarly the definition of PREVBT causes this symbol to be added to the list of dependencies on BUFFER



4 BUFFER RESB 4096



5 BUFEND EQU *

The definition of BUFFER on line 4 let us begin evaluation of some of the symbols. Let us assume that when lin4 is read, the location counter contains the hexadecimal value 1034. This address is stored as the value of BUFFER. The assembler then examines the list of symbol that are dependent on BUFFER. The

symbol table entry for the first symbol in this list(MAXLEN) shows that it depends

on two currently undefined symbols; therefore, MAXLEN cannot be evaluated

immediately.

Instead, the &2 is changed to &1 to show that only one symbol in the definition

(BUFEND) remains undefined.

The other symbol in the list (PREVBT) can be evaluated because it only depends

on the BUFFER. The value of the defining expression for PREVBT is calculated

and stored in SYMTAB.

The remainder of the processing follows the same pattern. When BUFEND is

defined by line 5, its value is entered into the symbol table.

The list associated with BUFEND then directs the assembler to evaluate

MAXLEN, and entering value of MAXLEN causes the evaluation of the symbol in

its list (HALFSZ).

This completes the symbol definition process.

If any symbol remains undefined at the end of the program, assembler would flag

them as error.

Q5 a) Briefly explain absolute loader with the algorithm or source program

Absolute 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
```

Q5 b) Explain the machine independent loader features:

i)Automatic library search ii) Loader option

1. Automatic Library Search

This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.

The routines are automatically retrieved from a library as they are needed during linking. This allows programmer to use subroutines from one or more libraries.

The programmer does not need to take any action beyond mentioning the subroutine names as external references.

This feature is called Automatic library call or Automatic library search

 Enter the symbols from each Refer record into ESTAB

 When the definition is encountered (Define record), the address is assigned

 At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references

 The loader searches the libraries specified (or standard) for undefined symbols or subroutines

 If all libraries are searched and some undefined symbol still remains, output error a message

The presented linking loader allows the programmer to override the standard subroutines by supplying his own routines.

The loader searches for the subroutines by scanning the Define records for all of the object programs which is inefficient.

Assembled or compiled versions of the subroutines in a library can be structured using a directory that gives the name of each routine and a pointer to its address within the library.

The same techniques applies equally well to the resolution of external references to data items.

2. Loader Options

Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways.

They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and can be specified using loader control statements in the source program.

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

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

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

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

LIBRARY MYLIB – search MYLIB library before standard libraries NOCALL

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

LIBRARY UTLIB

INCLUDE READ (UTLIB)

INCLUDE WRITE (UTLIB)

```
DELETE RDREC, WRREC

CHANGE RDREC, READ

CHANGE WRREC, WRITE

NOCALL SQRT, PLOT
```

The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol

RDREC to be changed to the symbol READ, similarly references to WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.

Q6 a) Explain Loader design options.

There are some common alternatives for organizing the loading functions, including relocation and linking. Linking Loaders – Perform all linking and relocation at load time.

The Other Alternatives are Linkage editors, which perform linking prior to load time and, Dynamic linking, in which linking function is performed at execution time

1. Linkage Editor

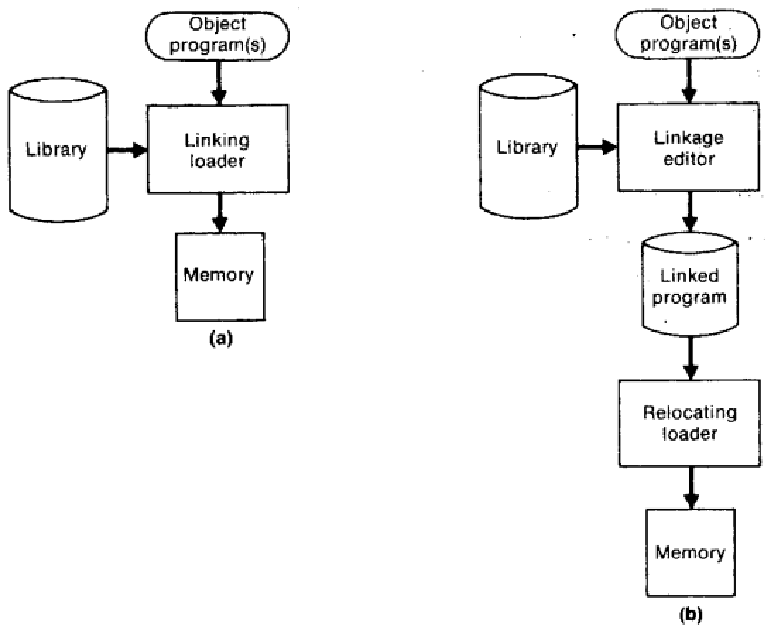
2. Dynamic Linking

3. Bootstrap Loaders

1. Linkage Editor

The figure below shows the processing of an object program using Linkage editor.

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



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.

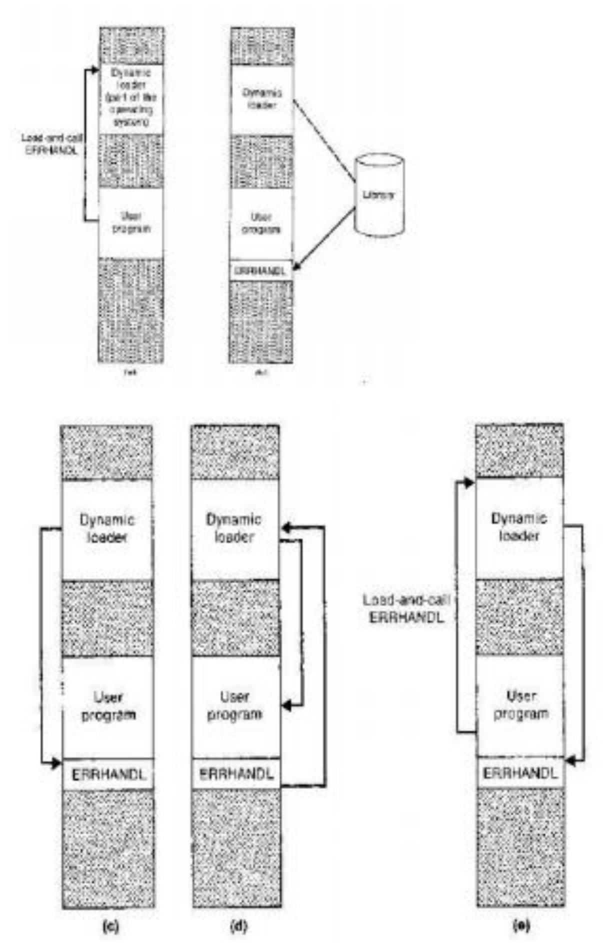


FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.

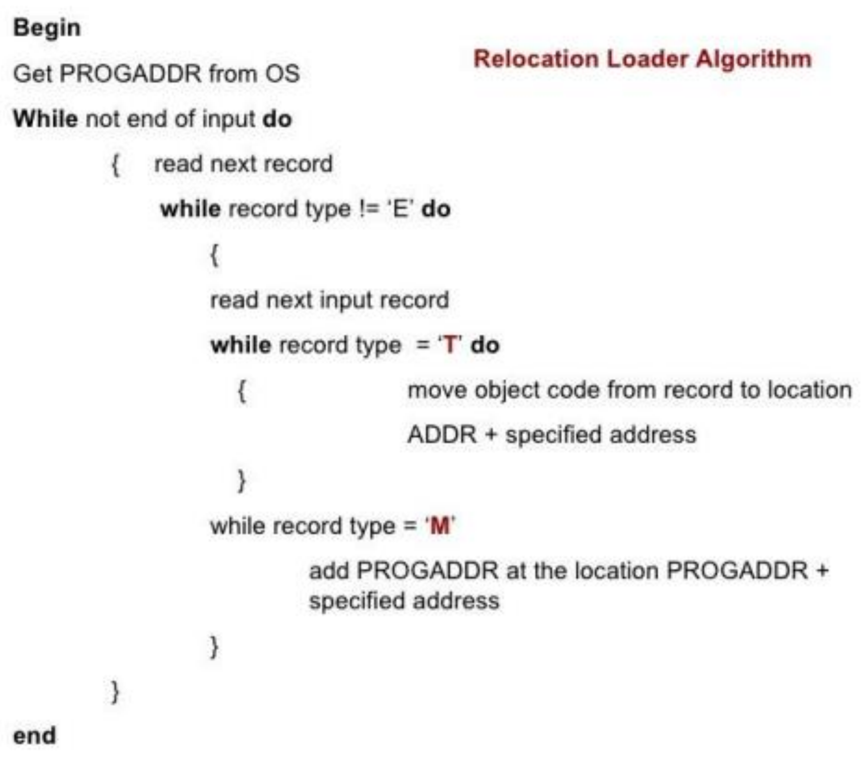
Q6 b) Explain relocation loader algorithm

Loaders that allow for program relocation are called relocating loaders or relative loaders

There are two methods for specifying relocation as part of the object program.

i) Modification record

A Modification record is used to describe each part of the object code that must be when program is relocated. There is one modification record for each value that must be changed during relocation. Each modification record specifies the starting address and length of the field whose value is to be altered. It then describes modification to be performed.



Relocation Loader Algorithm

```
Begin
Get PROGADDR from OS
While not end of input do
        {   read next record
            while record type != 'E' do
                {
                read next input record
                while record type  = 'T' do
                  {              move object code from record to location
                                 ADDR + specified address
                  }
                while record type = 'M'
                        add PROGADDR at the location PROGADDR +
                        specified address
                }
        }
end
```

ii) Relocation bit (Bit Mask)

If a machine primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using relocation bit

Each instruction is associated with one relocation bit. It Indicates that the corresponding word should be modified or not.

0: no modification is needed

1: modification is needed

This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record:

col 1: T

col 2-7: starting address

col 8-9: length (byte)

col 10-12: relocation bits

col 13-72: object code

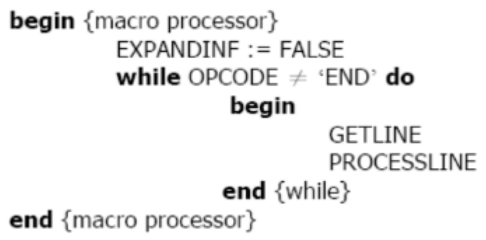
These relocation bits in a Text record are gathered into bit masks.

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments.

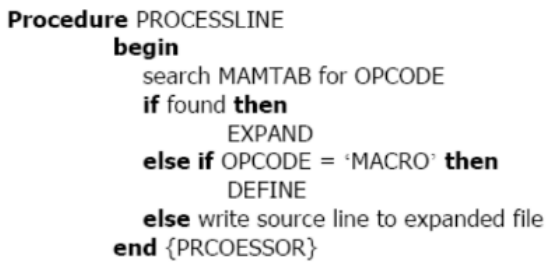
E.g. FFC=111111111100

E00=111000000000

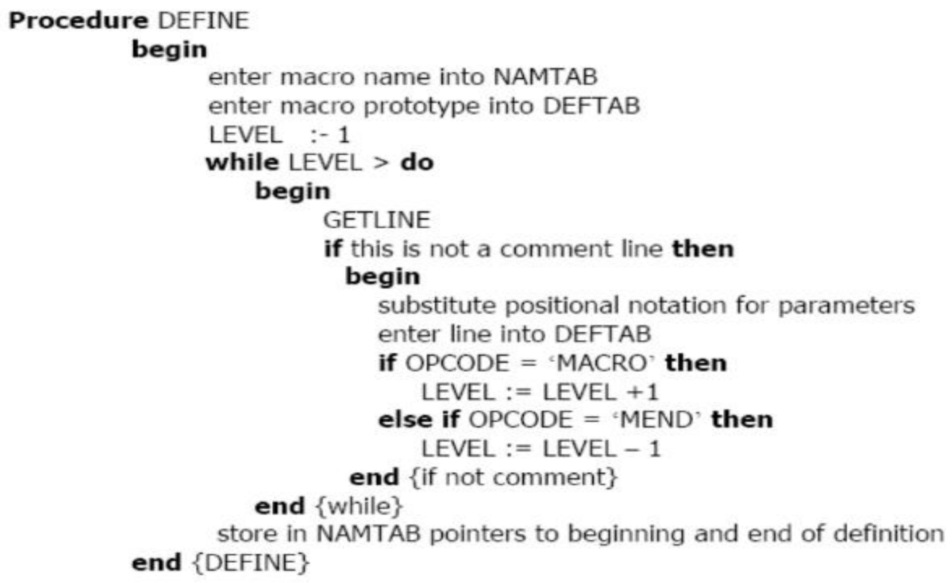
Q7 a) Explain macro processor algorithm



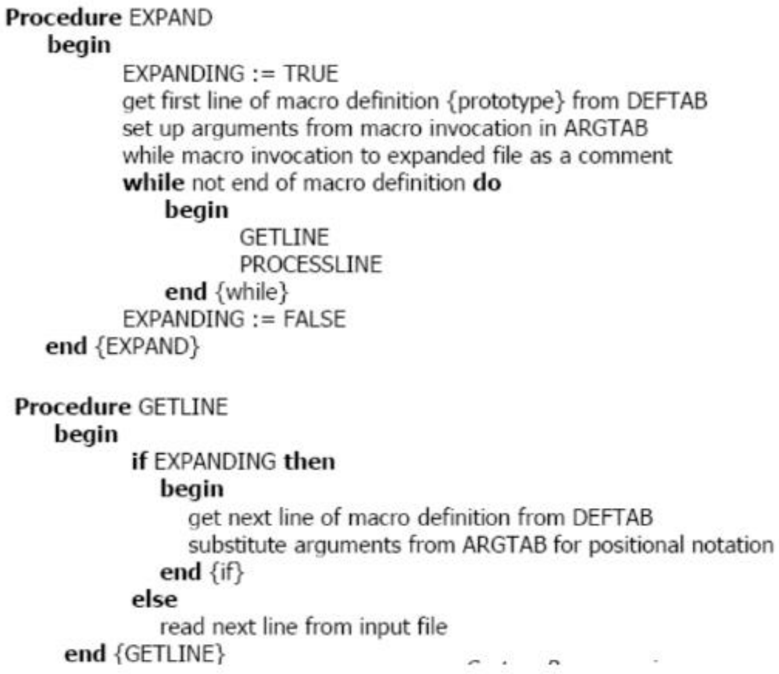
```
begin {macro processor}
        EXPANDINF := FALSE
        while OPCODE ≠ 'END' do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
end {macro processor}
```



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



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



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

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

Q7 b) Explain the following machine independent macro processor features:

i)Concatenation of macro parameters ii) Generation of Unique labels

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.

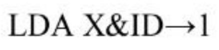
The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, Xb1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

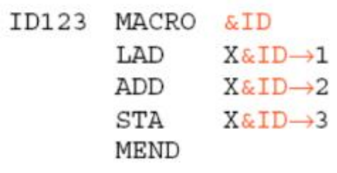
LDA X&ID1

& 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.

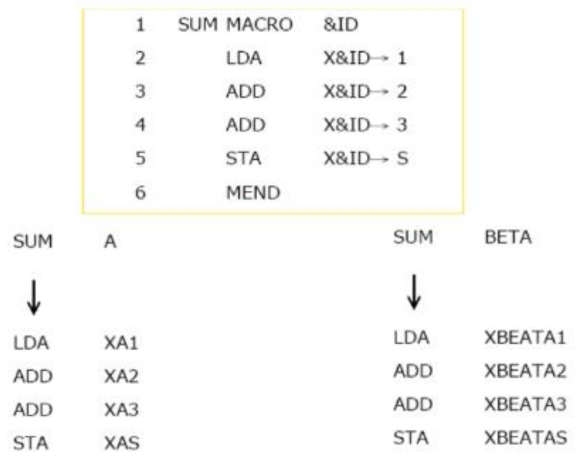
If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous. Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 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) **Generation of Unique Labels**

It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler.

This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion.

During macro expansion each $ will be replaced with $XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion. For example, XX = AA, AB, AC…

Q8 a) Define MACRO. Explain macro definition and expansion with suitable example.

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.

```
Source
M1      MACRO    &D1, &D2
        STA      &D1
        STB      &D2
        MEND
    .
M1 DATA1, DATA2
    .
M1 DATA4, DATA3
```

```
Expanded source
    .
    .
    .
        STA    DATA1
        STB    DATA2
    .
        STA    DATA4
        STB    DATA3
    .
```

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.

Q8 b) Explain the advantages and disadvantages of 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

Q9 a) Explain the following: i) interpreter ii) P-code compilers

Interpreters

 An interpreter processes a source program written in a high-level language, just as a compiler does.

 The main difference is that interpreters execute a version of the source program directly, instead of translating it into machine code.

 An interpreter usually performs lexical and syntactic analysis functions like those we have described for a compiler and then translates the source program into an internal form.

 After translating the source program into internal form, the interpreter executes the operations specified by the program.

 During this phase, an interpreter can be viewed as a set of subroutines.

The execution of these subroutines is driven by the internal form of the program.

 The process of translating a source program into some internal form is simpler and faster than compiling it into machine code. However, execution of the translated program by an interpreter is much slower than execution of the machine code produced by a compiler.

 Thus and interpreter would not normally be used if speed of execution is important.

 If speed of translation is of primary concern, and execution of the translated program will be short, then an interpreter may be a good choice.

 The real advantage of interpreter over a compiler, however, is in the debugging facilities that can easily be provided. The symbol table, source line numbers, and other information from the source program are usually retained by the interpreter.

 During execution, these can be used to produce symbolic dumps of data values, traces of program execution related to the source statements, etc

 Thus interpreters are especially attractive in educational environment where the emphasis is on learning and program testing.

 Most programming languages can either compiled or interpreted successfully. However, some languages are practically well suited to the use of an interpreter.

3) 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

Q9 b) Explain the tool compiler-compiler and also its advantages

Compiler-compiler

- A compiler-compiler is a software tool that can be used to help in the task of compiler construction.

- Such tools are often called compiler generators or translator-writing systems.

- The process of using a typical compiler-compiler is illustrated below.
- The user provides a description of the language to be translated. This description may consist of a set of lexical rules for defining tokens and a grammar for the source language.

- Some compiler-compilers use this information to generate a scanner and a parser directly.

- Others create tables for use by standard table driven scanning and parsing routines that are supplied by the compiler-compiler.

- In addition to the description of the source language, the user provides a set of semantic or code-generation routines.

 Compiler-compilers frequently provide special languages, notations, data structures and other similar facilities that can be used in the writing of semantic routines.

 The main advantage of using a compiler-compiler is of course ease of compiler construction and testing.

 The amount of work required from the user varies considerably form one compiler-compiler to another depending upon the degree of flexibility provided.

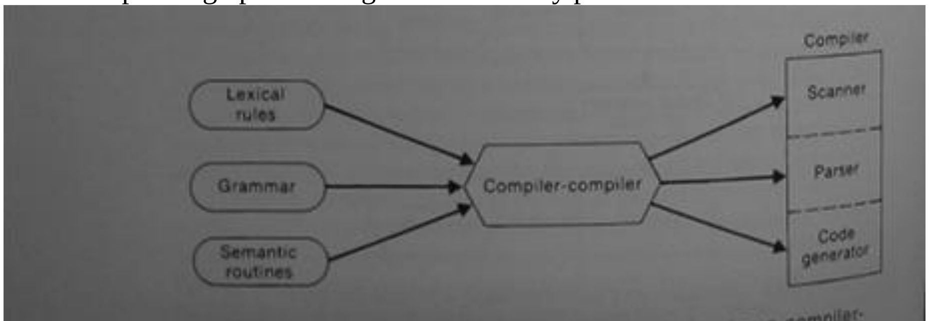


Figure 5.34 Automated compiler construction using a compiler-compiler.

Q10 a) Explain about code generation phase of compiler.

- After syntax of program has been analyzed, the task of compilation is the generation of object code.
- This technique creates object code for each part of the program as soon as its syntax has been recognized.
- It involves a set of routines, one for each rule or alternative rule in grammar.
- When the parser recognizes a portion of the source program according to some rule of the grammar the corresponding routine is executed. Such routine are often called as semantic routines or code generations routines.
- The specific code to be generated clearly depends upon the computer for which the program is being compiled.
- Code generation routines make use of two data structures for working storage: a list and a stack (list follows FIFO order and stack follows LIFO order)
- LISTCOUNT is used to keep a count of the number of items currently in the list.
- Code generation routines also make use of the token specifiers these specifiers are denoted by S(token).

Q10 b) Briefly discuss the 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).

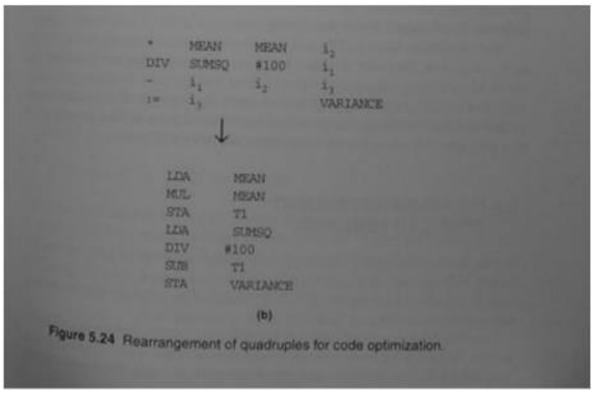
Registers are also used as instruction operands. Machine instructions that use registers as operands are usually faster than the corresponding instruction that refer to location in memory. Therefore it is preferable to store value or intermediate results in registers.

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.



```
*     MEAN    MEAN    i2
DIV   SUMSQ   #100    i1
-     i1      i2      i3
:=    i3              VARIANCE

          ↓

LDA     MEAN
MUL     MEAN
STA     T1
LDA     SUMSQ
DIV    #100
SUB     T1
STA     VARIANCE
```

(b)

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.