Subject Code : 17MCA25

Second Semester MCA Degree Examination, June/July 2018

System Software Answer Key

Module 1

Q1 What is System software? Differentiate it from application software (6 Marks)

Answer:

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. On other hand there are some aspects of system software that do not directly depend upon the type of computing system, general design and logic of an assembler, general design and logic of a compiler and code optimization techniques, which are independent of target machines. Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.

Examples

- Assemblers translate mnemonic instructions into machine code, the instruction formats, addressing modes, etc., are of direct concern in assembler design.
- Compilers generate machine code, taking into account such hardware characteristics as the number and type of registers & machine instruction available.
- Operating system concerned with the management of nearly all resources of a computing system. Loader and linker load the machine language program into the memory and prepare for execution. Debugger detects the errors in the program.

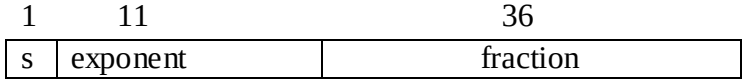
System Software	Application Software
Intended to support the operation and use of the computer	An application program is primarily concerned with the solution of some problem, using the computer as tool
Focus is on the Computer system and not on the application	The focus is on the application not on 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.

Q1 b. Describe the following with respect to SIC/XE machine (10 Marks)
i)Data Format ii) Addressing Mode iii)Instruction format.

Answer:

i) Data Format

- SIC/XE provides the same data formats as the standard version.
- In addition there is a 48 bit floating point data type with following format.



'S' indicate sign of floating point number (0= positive and 1= negative).
'fraction' is interpreted as value between 0 and 1.

'exponent' is interpreted as unsigned binary number between 0 and 2047.

If exponent as value e and fraction as value f then absolute value of the number is represented as $f*2^{(e-1024)}$.

ii) Addressing Modes

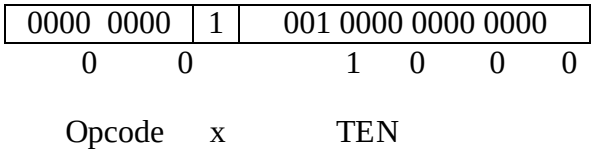
There are two addressing modes, indicated by the setting of the x bit in the instruction.

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

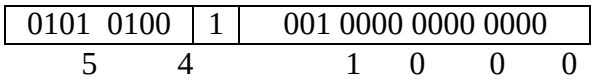


Effective address (EA) = 1000

Content of the address 1000 is loaded to Accumulator.

Indexed addressing mode

Example STCH BUFFER, X



Opcode x BUFFER

Effective address (EA) = 1000+[X]

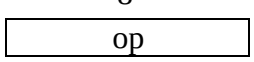
= 1000+content of the index register X

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

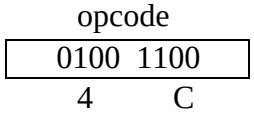
iii) Instruction Format

- SIC/XE has larger memory hence instruction format of standard SIC version is no longer suitable.
- SIC/XE provide two possible options; using relative addressing (Format 3) and extend the address field to 20 bit (Format 4).
- In addition SIC/XE provides some instructions that do not reference memory at all. (Format 1 and Format 2) .
- The new set of instruction format is as follows. Flag bit e is used to distinguish between format 3 and format 4. (e=0 means format 3, e=1 means format 4)

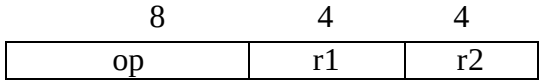
1. Format 1 (1 byte)



Example RSUB (return to subroutine)



2. Format 2 (2 bytes)



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	

Q2

a. What are the fundamental functions of any assembler? With an example, explain any four assembler directives.(10 marks)

Answer

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

5) RESB

Reserve the indicate number of bytes for a data area.

Example: BUFFER RESB 4096

6) RESW

Reserve the indicate number of words for a data area.

Example: LENGTH RESW 1

Q2 b. Explain the data structures used in assembler algorithm (06 Marks)

The simple assembler uses following internal data structures:

1) Operation Code Table (OPTAB)
2) Symbol Table (SYMTAB).
3) Location Counter (LOCCTR).

1) OPTAB:
   - It is used to lookup mnemonic operation codes and translates them to their machine language equivalents.
   - In more complex assemblers the table also contains information about instruction format and length
   - In pass 1 the OPTAB is used to look up and validate the operation code in the source program.
   - In pass 2, it is used to translate the operation codes to machine language.
   - In simple SIC machine this process can be performed in either in pass1 or in pass 2.
   - But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.
   - In pass 2 we take the information from OPTAB to tell us which
   - instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.
   - OPTAB is usually organized as a hash table, with mnemonic operation code as the key.
   - The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching.
   - Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

2) SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- During SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

3) LOCCTR:

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

Module-2

Q3 a. Compare a two-pass assembler with a one pass assembler. How forward references are handled in one-pass assemblers?

Answer

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

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.

Q3 b. Write note on 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

Q4a. Distinguish between a literal and an immediate operand. How does the assembler handle the literal operand?

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
```

All the literal operands used in a program are gathered together into one or

more literal pools. 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.

 The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

 Literal table is created for the literals which are used in the program. The literal table contains the literal name, operand value and length. The literal table is usually created as a hash table on the literal name

During Pass-1:

The literal encountered is searched in the literal table.

If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG for literal definition.

When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table.

At this time each literal currently in the table is assigned an address.

As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation.

LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

Q4 b. Explain the concept of program relocation with the help of neat figure.

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

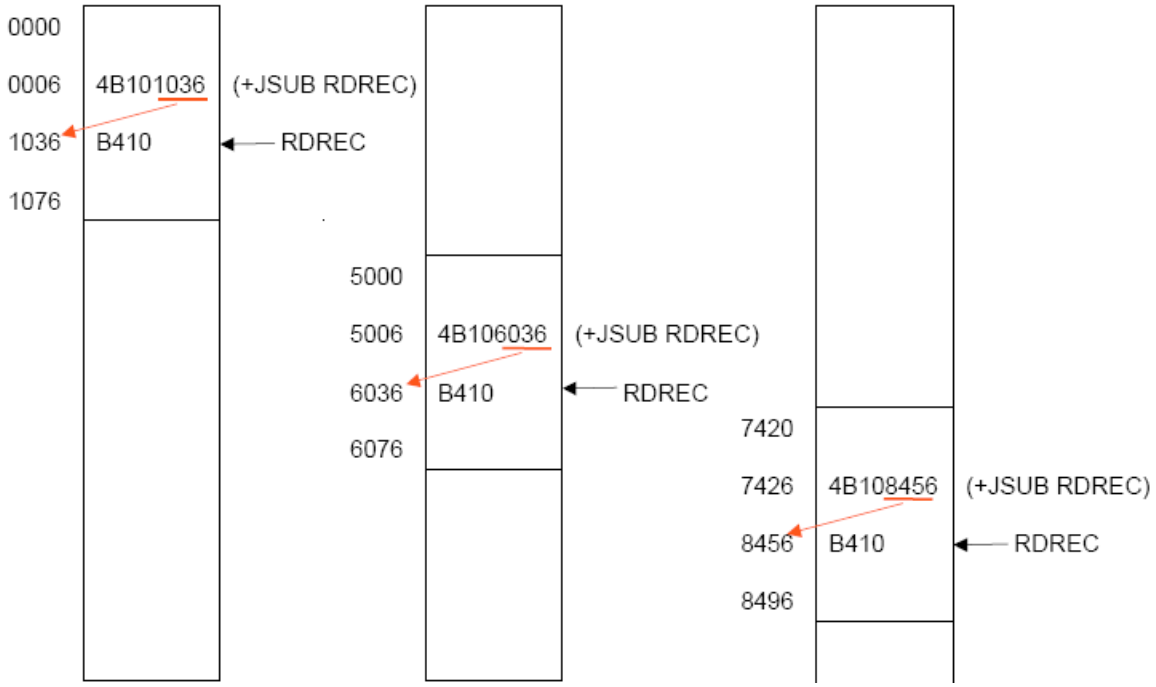
Col. 1 M

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

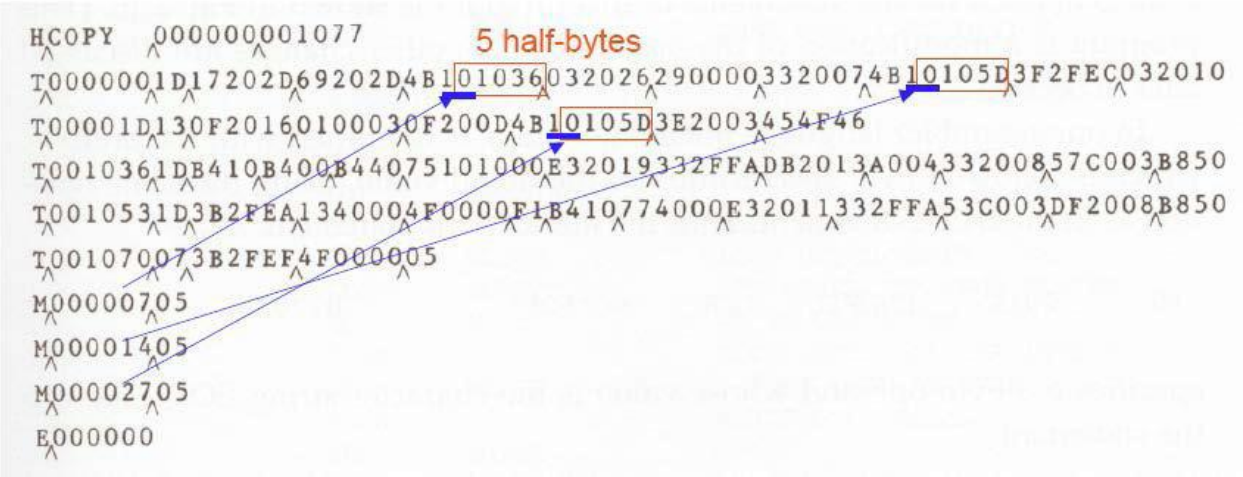
Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

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.



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.

Module-3

Q5 a. Briefly explain a simple bootstrap loader with an algorithm (10 marks)

Answer

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. 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.

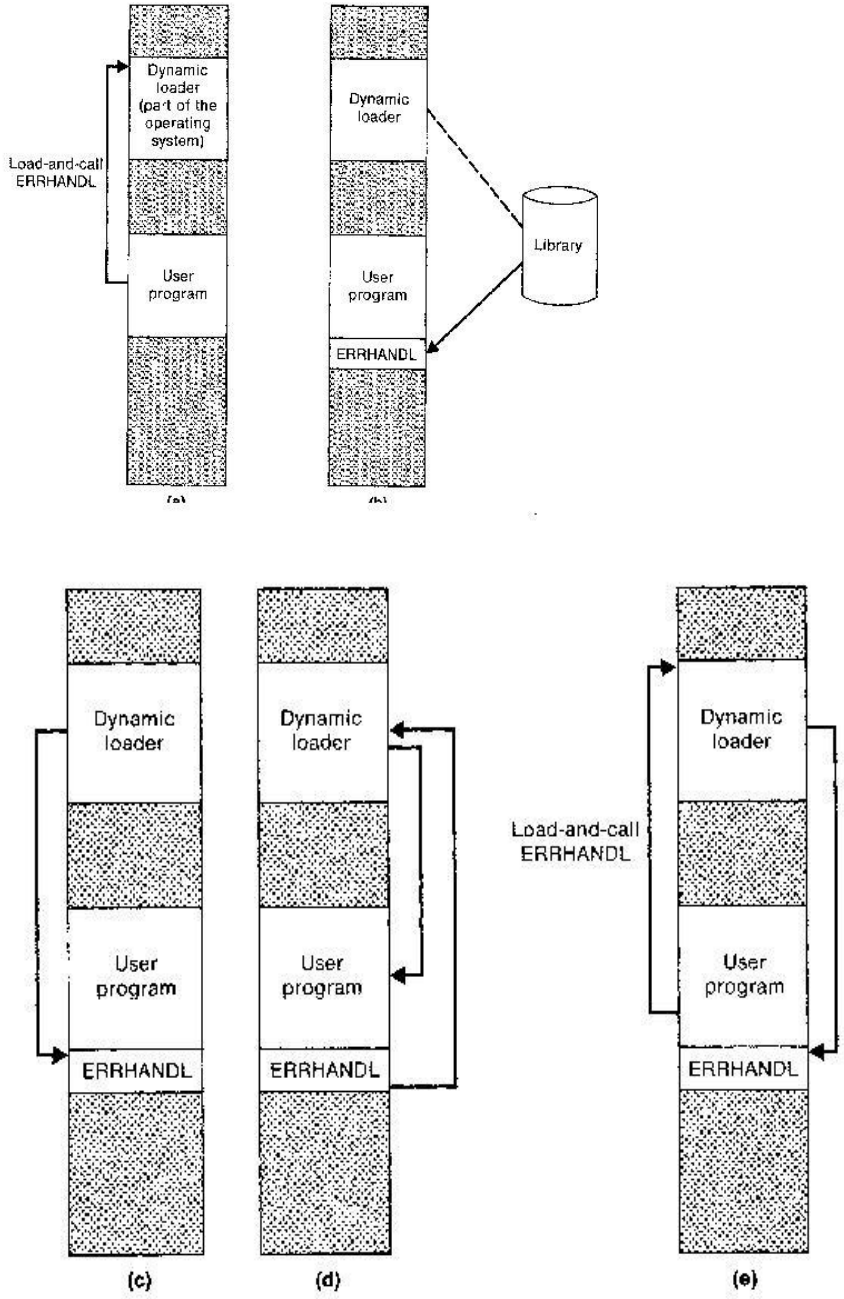
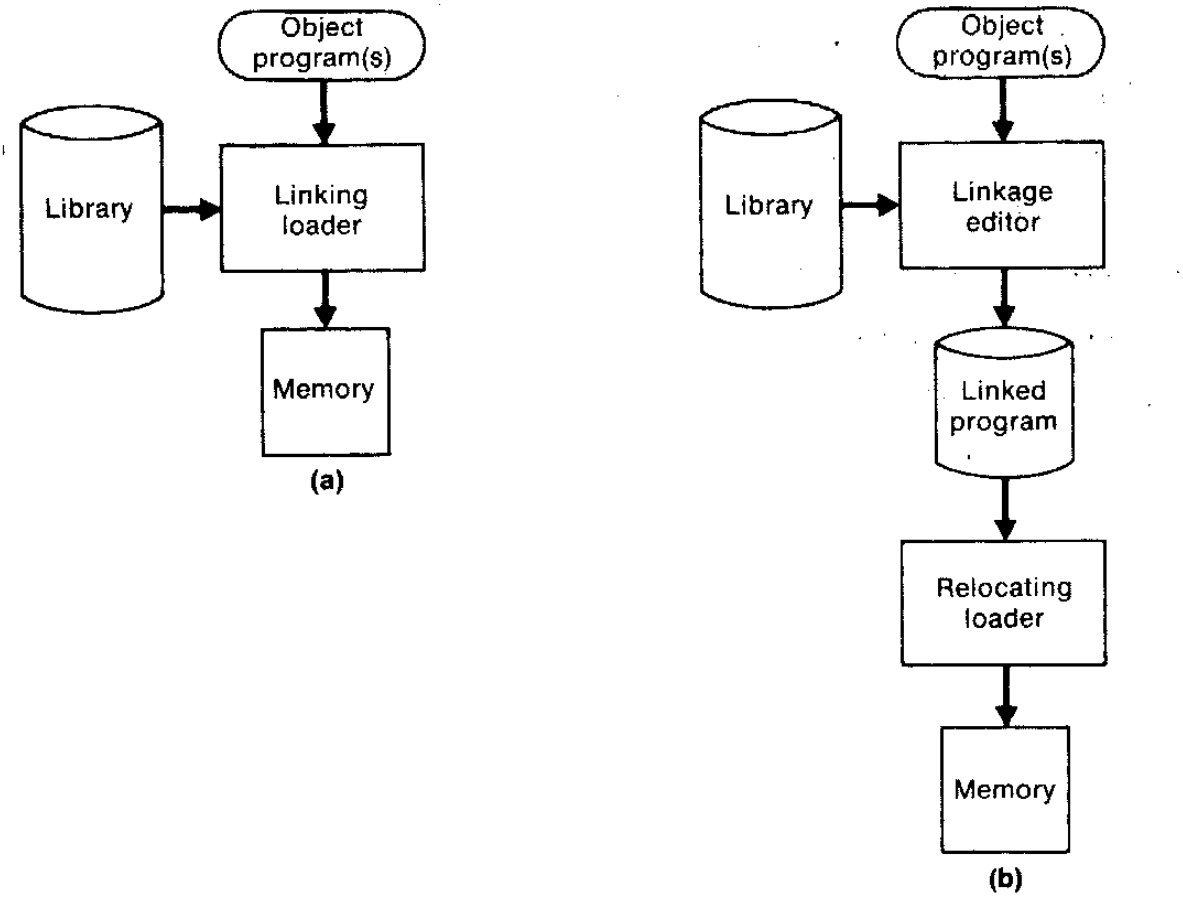


FIGURE 3.18 Loading and calling of a subroutine using dynamic linking.

OR

Q6 a. 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

Q6 b. Explain the features of MS-DOS linker

Answer

This explains some of the features of Microsoft MS-DOS linker, which is a linker for Pentium and other x86 systems. Most MS-DOS compilers and assemblers (MASM) produce object modules, and they are stored in .OBJ files. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file; this file is later executed for results.

The following table illustrates the typical MS-DOS object module

Record Types Description

THEADR Translator Header

TYPDEF,PUBDEF, EXTDEF External symbols and references

LNAMES, SEGDEF, GRPDEF Segment definition and grouping

LEDATA, LIDATA Translated instructions and data

FIXUPP Relocation and linking information

MODEND End of object module

THEADR specifies the name of the object module. MODEND specifies the end of the module. PUBDEF contains list of the external symbols (called public names).

EXTDEF contains list of external symbols referred in this module, but defined elsewhere.

TYPDEF the data types are defined here. SEGDEF describes segments in the object module (includes name, length, and alignment). GRPDEF includes how segments are combined into groups. LNAMES contains all segment and class names. LEDATA contains translated instructions and data. LIDATA has above in repeating pattern. Finally,

FIXUPP is used to resolve external references.

Module-4

Q7. a Write an algorithm for one-pass macro processor (8 marks)

```
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. Write the note on the following machine independent features of macro processor (8 marks)

i) Concatenation of macro parameters ii) Keyword macro parameters

Answer

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

```
TOTAL  MACRO  &ID
       LAD    X&ID1
       ADD    X&ID2
       STA    X&ID3
       MEND
```

```
TOTAL  A   ⇨   LAD    XA1
               ADD    XA2
               STA    XA3
```

& 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
```

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

ii) Keyword macro parameters

Positional parameter:

- parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement
- if argument is to be omitted, null value should be used.
- Not suitable if a macro has a large number of parameters and only few of them has values

Keyword parameters:

- each argument value is written with a keyword that named the corresponding parameter
- Arguments may appear in any order.
- Null arguments no longer need to be used.
- It is easier to read and much less error-prone than the positional method.

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

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

EXAMPLE

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

Parameters with default value

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

OR

Q8 a. What do you mean by a MACRO? Explain macro definition and expansion with suitable example. (8 marks)

Answer

A Macro represents a commonly used group of statements in the source programming language.

A macro instruction (macro) is a notational convenience for the programmer. It allows the programmer to write shorthand version of a program (module programming)

The macro processor replaces each macro instruction with the corresponding group of source language statements (expanding)

Normally, it performs no analysis of the text it handles. It does not concern the meaning of the involved statements during macro expansion.

Macro Definition and Expansion:

Two new assembler directives are used in macro definition

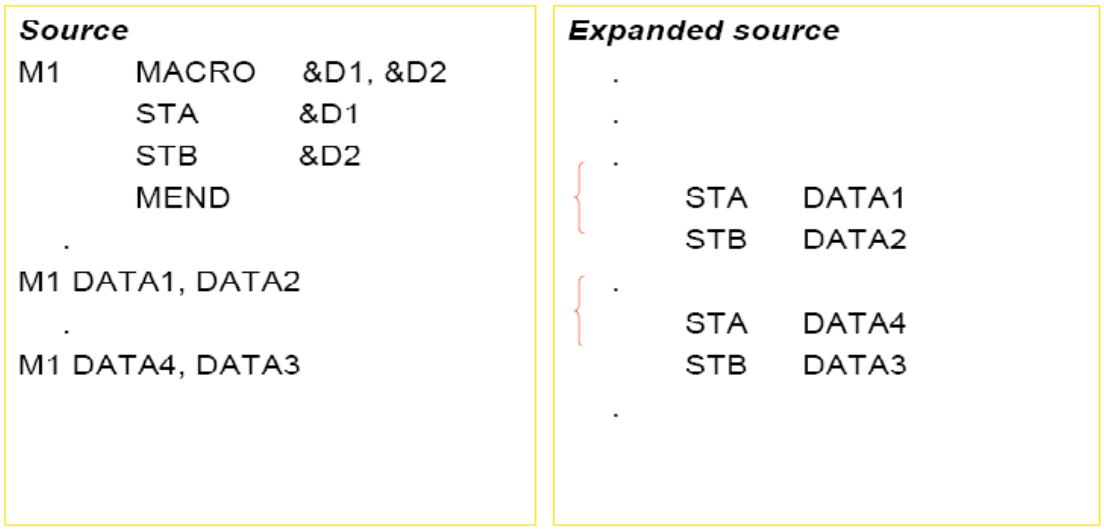
MACRO: identify the beginning of a macro definition

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

Each parameter begins with '&'

```
Name        MACRO       parameters
              :
            Body
              :
            MEND
```

The figure shows the MACRO expansion.



The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.

M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

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.

The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

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

Q8 b. Mention the advantages and disadvantages of 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. Describe how finite automata is used in recognizing the tokens of a typical programming language.

Answer

The tokens of most programming languages can be recognized by a finite automaton. Mathematically, a finite automaton consists of finite set of states and a set of transitions form one state to another. One of the states is designated, as the starting state, and one or more states are designated as final states

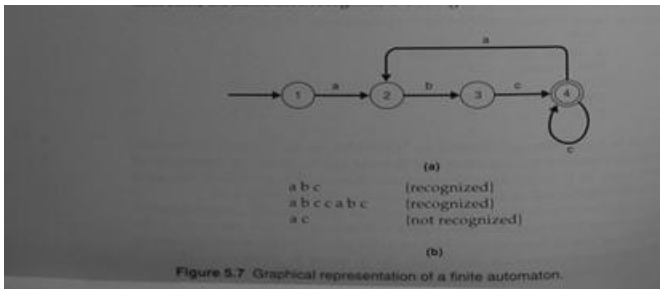


Figure 5.7 Graphical representation of a finite automaton.

States are represented by circles and transitions by arrow from one state to another. Each arrow is labeled with character or a set of characters that cause the special transition to occur. The starting state has an arrow entering it that is not connected to anything else. Final states are identified by double circles.

We can visualize the finite automaton as beginning in the starting state and moving from one state to another as it examines the characters being scanned. It stops when there is no transition from its current state that matches the next character to be scanned. If the automaton stops in a final state, we

say that it recognizes the string being scanned. If it stops in a non-final state it fails to recognize the string.

Finite automata provide an easy way to visualize the operation of a scanner. However, the real advantage of this kind of representation is in ease of implementation.

b. Describe process of Compiler-compilers.

Answer

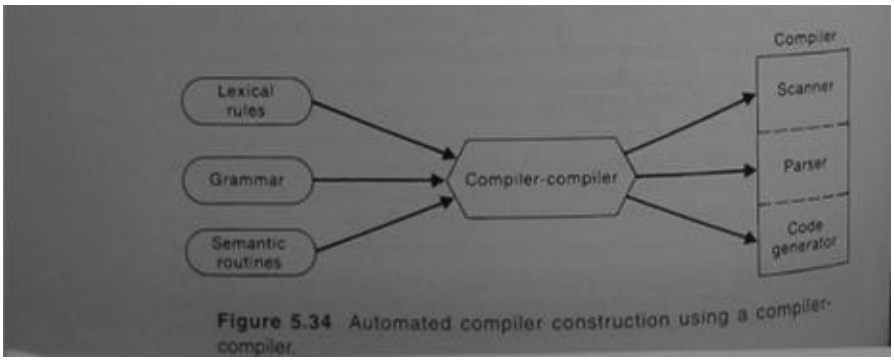
- A compiler-compiler is a software tool that can be used to help in the task of compiler construction.
- Such tools are often called compiler generators or translator-writing systems.
- The process of using a typical compiler-compiler is illustrated below.
- 

Figure 5.34 Automated compiler construction using a compiler-compiler.
- 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.
- Compilers that are generated in this way tend to require more memory and compile programs more slowly than hand written compilers.
- However, the object code generated by the compiler may actually be better when a compiler-compiler is used.
- Because of the automatic construction of scanners and parsers, and the special tools provided for writing semantic routines the compiler writer is freed from many of the mechanical details of compiler construction.
- The writer can therefore focus more attention on good code generation and optimization.

OR

Q10 a Explain the machine dependent code optimization of compiler with an example

There are several different possibilities for performing machine dependent code optimization.

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).

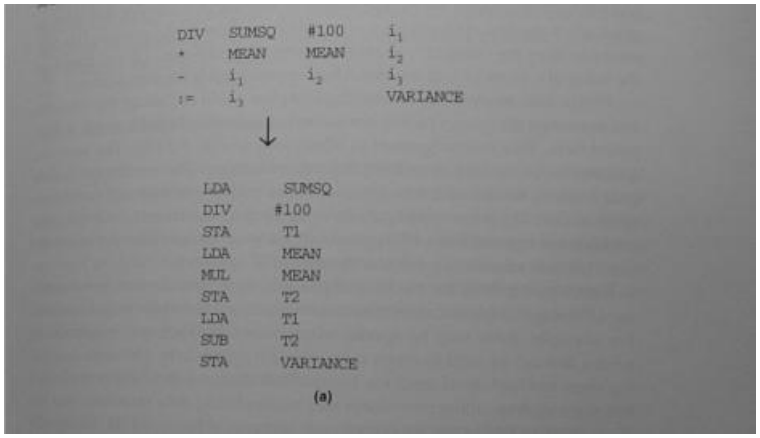
Registers are also used as instruction operands. Machine instructions that use registers as operands are usually faster than the corresponding instruction that refer to location in memory. Therefore it is preferable to store value or intermediate results in registers.

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.



Note that the value of the intermediate result i1 is calculated first and stored in temporary variable T1. Then the value of i2 is calculated. The third quadruple in this series calls for subtracting the value of i2 from i1. Since i2 had just been computed, its value is available registers A; however, this does no good, since the first operand for a – operation must be in register. It is necessary to store the value of i1 from T1 into register A before performing the subtraction**.**

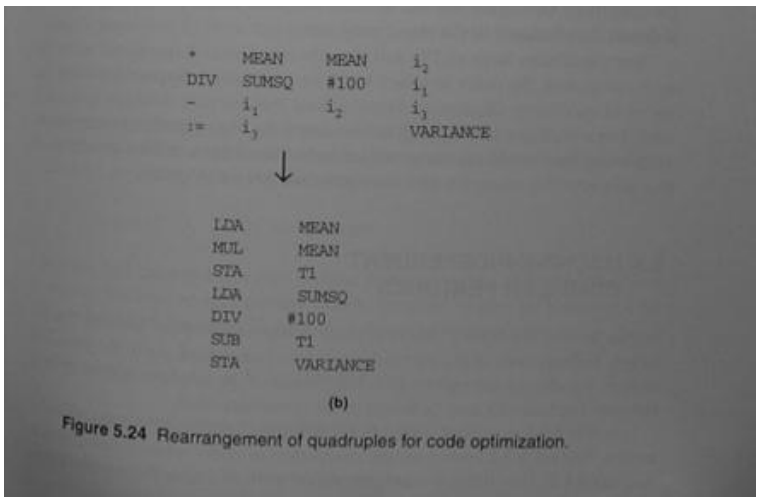


Figure 5.24 Rearrangement of quadruples for code optimization.

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.

b. What is parse tree ? Give an example.

Answer

A parse tree or parsing tree or derivation tree or concrete syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.