SYLLABUS

SYSTEM SOFTWARE

Subject Code: 10CS52	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART – A

UNIT – 1 6 Hours

Machine Architecture: Introduction, System Software and Machine Architecture, Simplified Instructional Computer (SIC) - SIC Machine Architecture, SIC/XE Machine Architecture, SIC Programming Examples.

UNIT – 2 6 Hours

Assemblers -1: Basic Assembler Function - A Simple SIC Assembler, Assembler Algorithm and Data Structures, Machine Dependent Assembler Features - Instruction Formats & Addressing Modes, Program Relocation.

UNIT – 3 6 Hours

Assemblers -2: Machine Independent Assembler Features – Literals, Symbol-Definition Statements, Expression, Program Blocks, Control Sections and Programming Linking, Assembler Design Operations - One-Pass Assembler, Multi-Pass Assembler, Implementation Examples - MASM Assembler.

UNIT – 4 8 Hours

Loaders and Linkers: Basic Loader Functions - Design of an Absolute Loader, A Simple Bootstrap Loader, Machine-Dependent Loader Features – Relocation, Program Linking, Algorithm and Data Structures for a Linking Loader; Machine-Independent Loader Features - Automatic Library Search, Loader Options, Loader Design Options - Linkage Editor, Dynamic Linkage, Bootstrap Loaders, Implementation Examples - MS-DOS Linker.

PART – B

UNIT – 5 6 Hours

Editors and Debugging Systems: Text Editors - Overview of Editing Process, User Interface, Editor Structure, Interactive Debugging Systems - Debugging Functions and Capabilities, Relationship With Other Parts Of The System, User-Interface Criteria

UNIT – 6 8 Hours

Macro Processor: Basic Macro Processor Functions - Macro Definitions and Expansion, Macro Processor Algorithm and Data Structures, Machine-Independent Macro Processor Features - Concatenation of Macro Parameters, Generation of Unique Labels, Conditional Macro Expansion, Keyword Macro Parameters, Macro Processor Design Options - Recursive Macro Expansion, General-Purpose Macro Processors, Macro Processing Within

Language Translators, Implementation Examples - MASM Macro Processor, ANSI C Macro Processor.

UNIT – 7 **6 Hours**

Lex and Yacc – 1: Lex and Yacc - The Simplest Lex Program, Recognizing Words With LEX, Symbol Tables, Grammars, Parser-Lexer Communication, The Parts of Speech Lexer, A YACC Parser, The Rules Section, Running LEX and YACC, LEX and Hand- Written Lexers, Using LEX - Regular Expression, Examples of Regular Expressions, A Word Counting Program, Parsing a Command Line.

UNIT – 8 **6 Hours**

Lex and Yacc - 2

Using YACC – Grammars, Recursive Rules, Shift/Reduce Parsing, What YACC Cannot Parse, A YACC Parser - The Definition Section, The Rules Section, Symbol Values and Actions, The LEXER, Compiling and Running a Simple Parser, Arithmetic Expressions and Ambiguity, Variables and Typed Tokens.

Text Books:

1. Leland.L.Beck: System Software, 3rd Edition, Addison-Wesley, 1997.
   (Chapters 1.1 to 1.3, 2 (except 2.5.2 and 2.5.3), 3 (except 3.5.2 and 3.5.3), 4 (except 4.4.3))
2. John.R.Levine, Tony Mason and Doug Brown: Lex and Yacc, O'Reilly, SPD, 1998.
   (Chapters 1, 2 (Page 2-42), 3 (Page 51-65))

Reference Books:

1. D.M.Dhamdhere: System Programming and Operating Systems, 2nd Edition, Tata McGraw - Hill, 1999.

TABLE OF CONTENTS

UNIT – 5

UNIT–6

UNIT – 7

UNIT – 8

UNIT-1

MACHINE ARCHITECTURE

1.1. Introduction:

The Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer.

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

1.2. System Software and Machine Architecture:

One characteristic in which most system software differs from application software is machine dependency.

-> System software – support operation and use of computer. Application software - solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly,

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

-> There are 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.

1.3. The Simplified Instructional Computer (SIC):

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. There are two versions of SIC, they are, standard model (SIC), and, extension version (SIC/XE) (extra equipment or extra expensive).

_SIC Machine Architecture:

We discuss here the SIC machine architecture with respect to its Memory and Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output

Memory :

There are 2^{15} bytes in the computer memory, that is 32,768 bytes. It uses Little Endian format to store the numbers, 3 consecutive bytes form a word , each location in memory contains 8-bit bytes.

Registers:

There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

Data Formats:

Integers are stored as 24-bit binary numbers. 2's complement representation is used for negative values, characters are stored using their 8-bit ASCII codes.No floating-point hardware on the standard version of SIC.

Instruction Formats:

Opcode(8)	x	Address (15)

All machine instructions on the standard version of SIC have the 24-bit format as shown above

Addressing Modes:

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

There are two addressing modes available, which are as shown in the above table. Parentheses are used to indicate the contents of a register or a memory location.

Instruction Set :

- SIC provides, load and store instructions (LDA, LDX, STA, STX, etc.). Integer arithmetic operations: (ADD, SUB, MUL, DIV, etc.).
- All arithmetic operations involve register A and a word in memory, with the result being left in the register. Two instructions are provided for subroutine linkage.
- COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to indicate the result. There are conditional jump instructions: (JLT, JEQ, JGT), these instructions test the setting of CC and jump accordingly.
- JSUB jumps to the subroutine placing the return address in register L, RSUB returns by jumping to the address contained in register L.

Input and Output:

Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

Data movement and Storage Definition

LDA, STA, LDL, STL, LDX, STX (A- Accumulator, L – Linkage Register, X – Index Register), all uses 3-byte word. LDCH, STCH associated with characters uses 1-byte. There are no memory-memory move instructions.

Storage definitions are

- WORD - ONE-WORD CONSTANT
- RESW - ONE-WORD VARIABLE
- BYTE - ONE-BYTE CONSTANT
- RESB - ONE-BYTE VARIABLE

Example Programs (SIC):

Example 1: Simple data and character movement operation

```
            LDA     FIVE
            STA     ALPHA
            LDCH    CHARZ
            STCH    C1
        .
ALPHA       RESW    1
FIVE        WORD    5
CHARZ       BYTE    C’Z’
C1          RESB    1
```

Example 2: Arithmetic operations

```
            LDA     ALPHA
            ADD     INCR
            SUB     ONE
            STA     BETA
            ……..
            ……..
            ……..
            ……..
ONE         WORD    1
ALPHA       RESW    1
BEETA       RESW    1
INCR        RESW    1
```

Example 3: Looping and Indexing operation

```
            LDX     ZERO            ;   X = 0
 MOVECH     LDCH    STR1, X         ;   LOAD A FROM STR1
            STCH    STR2, X         ;   STORE A TO STR2
            TIX     ELEVEN          ;   ADD 1 TO X, TEST
            JLT     MOVECH
        .
        .
        .
STR1        BYTE    C ‘HELLO WORLD’
STR2        RESB    11
ZERO        WORD    0
ELEVEN      WORD    11
```

Example 4: Input and Output operation

```
INLOOP   TD     INDEV        : TEST INPUT DEVICE
         JEQ    INLOOP       : LOOP UNTIL DEVICE IS READY
         RD     INDEV        : READ ONE BYTE INTO A
         STCH DATA           :  STORE A TO DATA
         .
         .
OUTLP    TD     OUTDEV       : TEST OUTPUT DEVICE
         JEQ    OUTLP        : LOOP UNTIL DEVICE IS READY
         LDCH   DATA         : LOAD DATA INTO A
         WD     OUTDEV       : WRITE A TO OUTPUT DEVICE
          .
          .
INDEV    BYTE    X 'F5'      : INPUT DEVICE NUMBER
OUTDEV   BYTE    X '08'      : OUTPUT DEVICE NUMBER
DATA     RESB    1           : ONE-BYTE VARIABLE
```

Example 5: To transfer two hundred bytes of data from input device to memory

```
         LDX    ZERO
CLOOP    TD     INDEV
         JEQ    CLOOP   RD
                INDEV
         STCH   RECORD, X
         TIX    B200
         JLT    CLOOP
         .
         .
INDEV    BYTE   X  'F5'
RECORD   RESB   200
ZERO     WORD   0
B200     WORD   200
```

1.4 SIC/XE Machine Architecture:

Memory

Maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} bytes).

Registers

Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC.

Floating-point data type:

There is a 48-bit floating-point data type, $F*2^{(e-1024)}$

Instruction Formats:

The new set of instruction formats fro SIC/XE machine architecture are as follows.
Format 1 (1 byte): contains only operation code (straight from table).

Format 2 (2 bytes): first eight bits for operation code, next four for register 1 and following four for register 2.
The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).

Format 3 (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).

Format 4 (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.

Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

- **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.

- **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)

- **Immediate**(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)

- **Indirect**(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

- **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings

e - > e = 0 means format 3, e = 1 means format 4

Bits x,b,p : Used to calculate the target address using relative, direct, and indexed addressing Modes

Bits i and n: Says, how to use the target address

b and p - both set to 0, disp field from format 3 instruction is taken to be the target address. For a format 4 bits b and p are normally set to 0, 20 bit address is the target address

x - x is set to 1, X register value is added for target address calculation
i=1, n=0 Immediate addressing, **TA**: TA is used as the operand value, no memory reference
i=0, n=1 Indirect addressing, **((TA))**: The word at the TA is fetched. Value of TA is taken as the address of the operand value

i=0, n=0 or i=1, n=1 Simple addressing, **(TA)**:TA is taken as the address of the operand value

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)

Instruction Set:

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating-point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction : RMO, Register-to-register arithmetic operations, ADDR, SUBR, MULR, DIVR and, Supervisor call instruction : SVC.

Input and Output:

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.

1.5. Example Programs (SIC/XE)

Example 1: Simple data and character movement operation

```
            LDA     #5
            STA     ALPHA
            LDA     #90
            STCH    C1
                    .
                    .
ALPHA       RESW    1
C1          RESB    1
```

Example 2: Arithmetic operations

```
        LDS    INCR
        LDA    ALPHA
        ADD    S,A
        SUB    #1
        STA    BETA
        ………….
        …………..
ALPHA   RESW   1
BETA    RESW   1
INCR    RESW   1
```

Example 3: Looping and Indexing operation

```
            LDT     #11
            LDX     #0          :  X = 0
MOVECH      LDCH    STR1, X     :  LOAD A FROM STR1
            STCH    STR2, X     :  STORE A TO STR2



            TIXR    T           :  ADD 1 TO X, TEST (T)
            JLT     MOVECH
            ……….
            ……….
            ………
STR1        BYTE    C ‘HELLO WORLD’
STR2        RESB    11
```

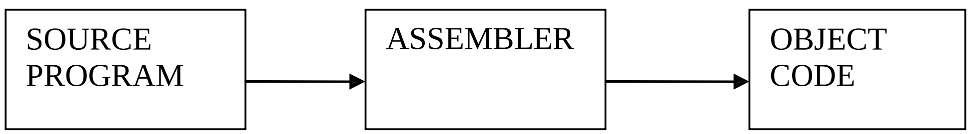
UNIT – 2

ASSEMBLERS – 1

2.1. Basic Assembler Functions:

The basic assembler functions are:

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.



• The design of assembler can be to perform the following:
- Scanning (tokenizing)
- Parsing (validating the instructions)
- Creating the symbol table
- Resolving the forward references
- Converting into the machine language

• The design of assembler in other words:
- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Decide the proper instruction format Convert the data constants to internal machine representations
- Write the object program and the assembly listing

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

• SIC Assembler Directive:
- START: Specify name & starting address.
- END: End of the program, specify the first execution instruction.
- BYTE, WORD, RESB, RESW
- End of record: a null char(00)
  End of file: a zero length record

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

2.2. Single-pass Assembler:

In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference. This is shown with an example below:

```
10      1000        FIRST       STL   RETADR          141033
--
--
--
--
95      1033        RETADR      RESW        1
```

In the above example in line number 10 the instruction STL will store the linkage register with the contents of RETADR. But during the processing of this instruction the value of this symbol is not known as it is defined at the line number 95. Since I single-pass assembler the scanning, parsing and object code conversion happens simultaneously. The instruction is fetched; it is scanned for tokens, parsed for syntax and semantic validity. If it valid then it has to be converted to its equivalent object code. For this the object code is generated for the opcode STL and the value for the symbol RETADR need to be added, which is not available.

Due to this reason usually the design is done in two passes. So a multi-pass assembler resolves the forward references and then converts into the object code. Hence the process of the multi-pass assembler can be as follows:

Pass-1

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

Pass-2

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

2.3 Assembler Design:

The most important things which need to be concentrated is the generation of Symbol table and resolving *forward references*.

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

Example Program:

The example program considered here has a main module, two subroutines

- Purpose of example program
  - Reads records from input device (code F1)
  - Copies them to output device (code 05)
  - At the end of the file, writes EOF on the output device, then RSUB to the operating system
- Data transfer (RD, WD)
  - A buffer is used to store record
  - Buffering is necessary for different I/O rates
  - The end of each record is marked with a null character (00)16
  - The end of the file is indicated by a zero-length record
- Subroutines (JSUB, RSUB)
  - RDREC, WRREC
  - Save link register first before nested jump

Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	2039	RDREC	LDX	ZERO	041030
130	203C		LDA	ZERO	001030
135	203F	RLOOP	TD	INPUT	E0205D
140	2042		JEQ	RLOOP	30203F
145	2045		RD	INPUT	D8205D
150	2048		COMP	ZERO	281030
155	204B		JEQ	EXIT	302057
160	204E		STCH	BUFFER,X	549039
165	2051		TIX	MAXLEN	2C205E
170	2054		JLT	RLOOP	38203F
175	2057	EXIT	STX	LENGTH	101036
180	205A		RSUB		4C0000
185	205D	INPUT	BYTE	X'F1'	F1
190	205E	MAXLEN	WORD	4096	001000
195					

195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	2061	WRREC	LDX	ZERO	041030
215	2064	WLOOP	TD	OUTPUT	E02079
220	2067		JEQ	WLOOP	302064
225	206A		LDCH	BUFFER,X	509039
230	206D		WD	OUTPUT	DC2079
235	2070		TIX	LENGTH	2C1036
240	2073		JLT	WLOOP	382064
245	2076		RSUB		4C0000
250	2079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

The first column shows the line number for that instruction, second column shows the addresses allocated to each instruction. The third column indicates the labels given to the statement, and is followed by the instruction consisting of opcode and operand. The last column gives the equivalent object code.

The *object code* later will be loaded into memory for execution. The simple object program we use contains three types of records:

- Header record
  - Col. 1 H
  - Col. 2~7 Program name
  - Col. 8~13 Starting address of object program (hex)
  - Col. 14~19 Length of object program in bytes (hex)
- Text record
  - Col. 1 T
  - Col. 2~7 Starting address for object code in this record (hex)
  - Col. 8~9 Length of object code in this record in bytes (hex)
  - Col. 10~69 Object code, represented in hex (2 col. per byte)
- End record
  - Col.1 E
  - Col.2~7 Address of first executable instruction in object program (hex) "^" is only for separation only

Object code for the example program:

1. Simple SIC Assembler

The program below is shown with the object code generated. The column named LOC gives the machine addresses of each part of the assembled program (assuming the program is starting at location 1000). The translation of the source program to the object program requires us to accomplish the following functions:

1. Convert the mnemonic operation codes to their machine language equivalent.
2. Convert symbolic operands to their equivalent machine addresses.
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into their internal machine representations in the proper format.
5. Write the object program and assembly listing.

All these steps except the second can be performed by sequential processing of the source program, one line at a time. Consider the instruction

10 1000 LDA ALPHA 00-----

This instruction contains the forward reference, i.e. the symbol ALPHA is used is not yet defined. If the program is processed (scanning and parsing and object code conversion) is done line-by-line, we will be unable to resolve the address of this symbol. Due to this problem most of the assemblers are designed to process the program in two passes.

In addition to the translation to object program, the assembler has to take care of handling assembler directive. These directives do not have object conversion but giv es direction to the assembler to perform some function. Examples of directives are the statements like BYTE and WORD, which directs the assembler to reserve memory locations without generating data values. The other directives are START which indicates the beginning of the program and END indicating the end of the program.

The assembled program will be loaded into memory for execution. The simple object program contains three types of records: Header record, Text record and end record. The header record contains the starting address and length. Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded. The end record marks the end of the object program and specifies the address where the execution is to begin.

The format of each record is as given below.

Header record:

Col 1	H
Col. 2-7	Program name
Col 8-13	Starting address of object program (hexadecimal)
Col 14-19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T
Col 2-7.	Starting address for object code in this record (hexadecimal)
Col 8-9	Length off object code in this record in bytes (hexadecimal)
Col 10-69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

Col. 1	E
Col 2-7	Address of first executable instruction in object program (hexadecimal)

The assembler can be designed either as a single pass assembler or as a two pass assembler. The general description of both passes is as given below:

- Pass 1 (define symbols)
  - Assign addresses to all statements in the program
  - Save the addresses assigned to all labels for use in Pass 2
  - Perform assembler directives, including those for address assignment, such as BYTE and RESW
- Pass 2 (assemble instructions and generate object program)
  - Assemble instructions (generate opcode and look up addresses)
  - Generate data values defined by BYTE, WORD
  - Perform processing of assembler directives not done during Pass 1
  - Write the object program and the assembly listing

2. 4. Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.
- Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2.
- A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

LOCCTR:

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

The Algorithm for Pass 1:

```
Begin
  read first input line
   if  OPCODE  =  'START'  then  begin
      save  #[Operand]  as  starting  addr
      initialize LOCCTR to starting address
      write line to intermediate file
       read  next  line
    end(  if  START)
    else
     initialize LOCCTR to 0
      While OPCODE != 'END' do
        begin
           if this is not a comment line then
               begin
                 if there is a symbol in the LABEL field then
                    begin
                      search SYMTAB for LABEL
                       if found then
                          set error flag (duplicate symbol)
                       else
                          (if symbol)
                  search OPTAB for OPCODE
                  if found then
                      add 3 (instr length) to LOCCTR
                  else if OPCODE = 'WORD' then
                       add 3 to LOCCTR
                  else if OPCODE = 'RESW' then
                        add 3 * #[OPERAND] to LOCCTR
                  else if OPCODE = 'RESB' then
                        add #[OPERAND] to LOCCTR
                  else if OPCODE = 'BYTE' then
                        begin
                                    find length of constant in bytes
                                    add length to LOCCTR
                         end
else
                        set error flag (invalid operation code)
            end (if not a comment)
            write line to intermediate file
             read next input line
        end { while not END}
        write last line to intermediate file
        Save (LOCCTR – starting address) as program length
End {pass 1}
```

- The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.

- If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.

- If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.

- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes.

- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

The Algorithm for Pass 2:

```
Begin
 read 1st input line
   if OPCODE = 'START' then
     begin
       write   listing   line
        read next input line
      end
   write Header record to object program
   initialize 1st Text record
while OPCODE != 'END' do
    begin
      if this is not comment line then
        begin
          search OPTAB for OPCODE
            if found then
              begin
                if there is a symbol in OPERAND field then
                    begin
                       search SYMTAB for OPERAND field then
                       if found then
                       begin
```

```
                                        store symbol value as operand address
else
                                    begin
                                        store 0 as operand address
                                        set error flag (undefined symbol)
                                    end
                                end (if symbol)
                              else store 0 as operand address
                                   assemble the object code instruction
                              else if OPCODE = 'BYTE' or 'WORD" then
                                  convert constant to object code
                               if object code doesn't fit into current Text record then
                                  begin
                                    Write text record to object code
                                    initialize new Text record

                                  end
                                  add object code to Text record
          end {if not comment}
         write listing line
        read next input line
        end
       write listing line
       read next input line
       write last listing line
     End {Pass 2}
```

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.

If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code(for example, for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written.

Design and Implementation Issues

Some of the features in the program depend on the architecture of the machine. If the program is for SIC machine, then we have only limited instruction formats and hence limited addressing modes. We have only single operand instructions. The operand is always a memory reference. Anything to be fetched from memory requires more time. Hence the improved version of SIC/XE machine provides more instruction formats and hence more addressing modes. The moment we change the machine architecture the availability of number of instruction formats and the addressing modes changes. Therefore the design usually requires considering two things: Machine-dependent features and Machine-independent features.

2.5. Machine-Dependent Features:

- Instruction formats and addressing modes
- Program relocation

1 .<u>Instruction formats and Addressing Modes</u>

The instruction formats depend on the memory organization and the size of the memory. In SIC machine the memory is byte addressable. Word size is 3 bytes. So the size of the memory is 2^{12} bytes. Accordingly it supports only one instruction format. It has only two registers: register A and Index register. Therefore the addressing modes supported by this architecture are direct, indirect, and indexed. Whereas the memory of a SIC/XE machine is 2^{20} bytes (1 MB). This supports four different types of instruction types, they are:

- 1 byte instruction
- 2 byte instruction
- 3 byte instruction
- 4 byte instruction

- Instructions can be:
  - Instructions involving register to register
  - Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
  - Extended instruction format
- Addressing Modes are:
  - Index Addressing(SIC): Opcode m, x
  - Indirect Addressing: Opcode @m
  - PC-relative: Opcode m
  - Base relative: Opcode m
  - Immediate addressing: Opcode #c

. *Translations for the Instruction involving Register-Register addressing mode:*

During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

Translation involving Register-Memory instructions:

In SIC/XE machine there are four instruction formats and five addressing modes. For formats and addressing modes

Among the instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other operand is in the memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

There are two ways: *Program-counter relative and Base-relative.* This addressing mode can be represented by either using format-3 type or format-4 type of instruction format. In format-3, the instruction has the opcode followed by a 12-bit displacement value in the address field. Where as in format-4 the instruction contains the mnemonic code followed by a 20-bit displacement value in the address field.

Program-Counter Relative:

In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value.

The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction.

This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:

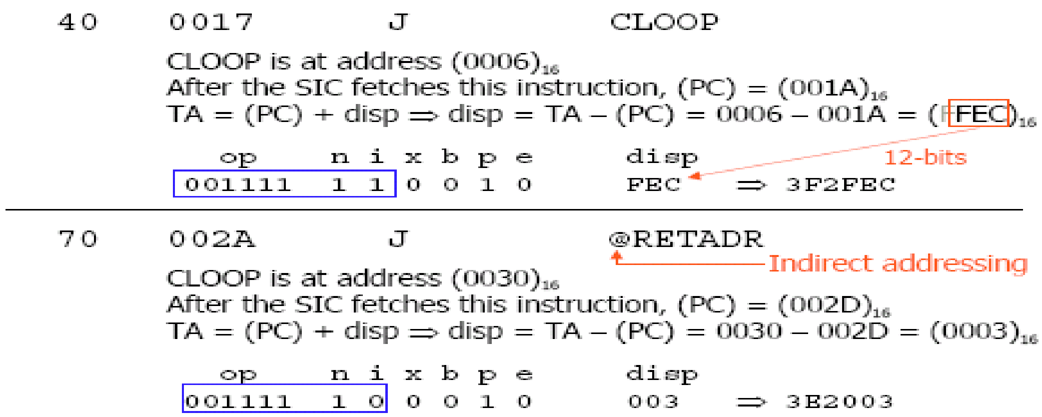
```
10    0000        FIRST      STL        RETADR
```

RETADR is at address $(0030)_{16}$

After the SIC fetches this instruction, $(PC) = (0003)_{16}$

$TA = (PC) + disp \Rightarrow disp = TA - (PC) = 0030 - 0003 = (02D)_{16}$

op	n	i	x	b	p	e	disp	
000101	1	1	0	0	1	0	02D	⇒ 17202D



Base-Relative Addressing Mode:

in this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

- This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE.
- The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.
- When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

```
LDB  #LENGTH (instruction)
BASE LENGTH (directive)
:
NOBASE
```

For example:

12	0003	LDB	#LENGTH		69202D
13		BASE	LENGTH		
: :					
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
: :					
160	104E	STCH	BUFFER,	X	57C003
165	1051	TIXR	T	B850	

In the above example the use of directive BASE indicates that Base-relative addressing mode is to be used to calculate the target address. PC-relative is no longer used. The value of the LENGTH is stored in the base register. If PC-relative is used then the target address calculated is:

- The LDB instruction loads the value of length in the base register which 0033. BASE directive explicitly tells the assembler that it has the value of LENGTH.

BUFFER is at location $(0036)_{16}$
$(B) = (0033)_{16}$
disp = 0036 – 0033 = $(0003)_{16}$

op	n i	x b p e	disp	
010101	1 1	1 1 0 0	003	⇒ 57C003

20	000A		LDA	LENGTH	032026
: :					
175	1056	EXIT	STX	LENGTH	134000

Consider Line 175. If we use PC-relative

Disp = TA – (PC) = 0033 –1059 = EFDA

PC relative is no longer applicable, so we try to use BASE relative addressing mode.

Immediate Addressing Mode

In this mode no memory reference is involved. If immediate mode is used the target address is the operand itself.

55 0020 LDA #3 ← Immediate operand

TA = $(0003)_{16}$

op	n i	x b p e	disp	
000000	0 1	0 0 0 0	003	⇒ 010003

133 103C +LDT #4096 ← Extended instruction format

TA = $(01000)_{16}$

op	n i	x b p e	disp(20 bits)	
011101	0 1	0 0 0 1	01000	⇒ 75101000

If the symbol is referred in the instruction as the immediate operand then it is immediate with PC-relative mode as shown in the example below:

```
12      0003          LDB          #LENGTH

        LENGTH is at address 0033
        TA = (PC) + disp ⇒ disp = 0033 – 0006 = (002D)16
           op      n i x b p e      disp
        011010    0 1  0 0 1 0      02D    ⇒  69202D
```

Indirect and PC-relative mode:

In this type of instruction the symbol used in the instruction is the address of the location which contains the address of the operand. The address of this is found using PC-relative addressing mode. For example:

```
70      002A                 J              @RETADR
                             :                 :
95      0030  RETADR         RESW           1
        RETADR is at address 0030
        TA = (PC) + disp ⇒ disp = 0030 – 002D = (0003)16
           op      n i x b p e      disp
        001111    1 0  0 0 1 0      003    ⇒  3E2003
```

The instruction jumps the control to the address location RETADR which in turn has the address of the operand. If address of RETADR is 0030, the target address is then 0003 as calculated above.

2.6. Program Relocation

Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

Absolute Program

In this the address is mentioned during assembling itself. This is called *Absolute Assembly*. Consider the instruction:

55   101B   LDA   THREE   00102D

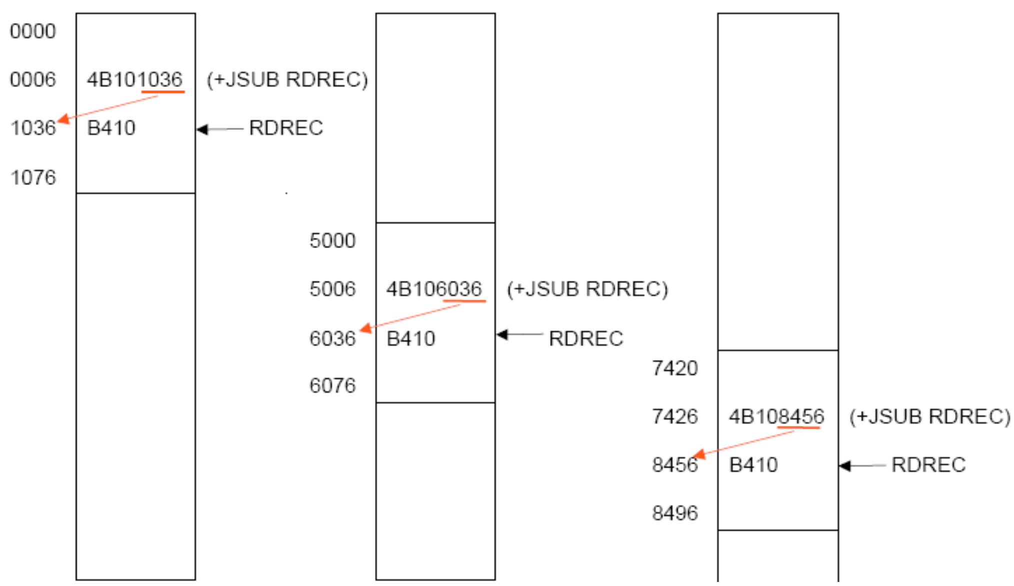
- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.

Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.

- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.

- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in 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.



- 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 labeled 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.

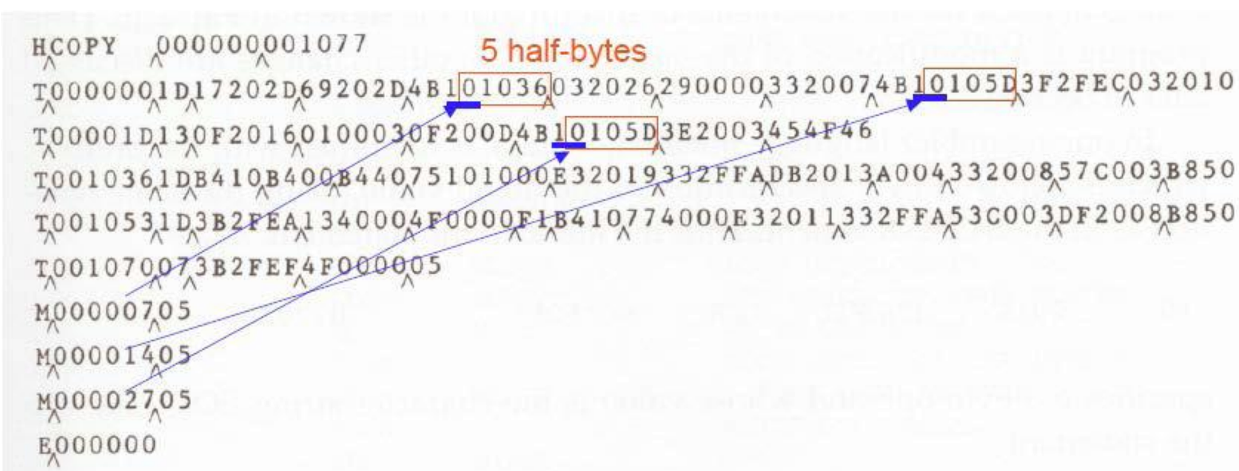
- From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.

- 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. The Modification has the 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 (4 bits) 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.



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.

UNIT – 3

ASSEMBLERS – 2

3.1. Machine-Independent features:

These are the features which do not depend on the architecture of the machine. These are:

- Literals
- Expressions
- Program blocks
- Control sections

Literals:

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

```
45      001A   ENDFIL       LDA    =C'EOF'       032010
-
-
93                          LTORG
        002D   *                   =C'EOF'       454F46
```

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

```
215     1062   WLOOP        TD     =X'05'        E32011
```

It is important to understand the difference between a constant defined as a literal and a constant defined as an immediate operand. In case of literals the assembler generates the specified value as a constant at some other memory location In immediate mode the operand value is assembled as part of the instruction itself. Example

```
55      0020                LDA    #03           010003
```

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

A 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.

Implementation of Literals:

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. The following figure shows the difference between the SYMTAB and LITTAB

SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062

LITTAB

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

3.2. Symbol-Defining Statements:

EQU Statement:

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

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants

and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example

```
+LDT        #4096
```

This loads the register T with immediate value 4096, this does not clearly what exactly this value indicates. If a statement is included as:

```
MAXLEN      EQU         4096 and then
            +LDT        #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

Another common usage of EQU statement is for defining values for the general-purpose registers. The assembler can use the mnemonics for register usage like a-register A , X – index register and so on. But there are some instructions which requires numbers in place of names in the instructions. For example in the instruction RMO 0,1 instead of RMO A,X. The programmer can assign the numerical values to these registers using EQU directive.

```
A           EQU         0
X           EQU         1 and so on
```

These statements will cause the symbols A, X, L… to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed. As another usage if in a machine that has many general purpose registers named as R1, R2,…, some may be used as base register, some may be used as accumulator. Their usage may change from one program to another. In this case we can define these requirement using EQU statements.

```
BASE        EQU         R1
INDEX       EQU         R2
```

COUNT EQU R3

One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined. For example, the following statement is not valid:

BETA EQU ALPHA
ALPHA RESW 1

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

ORG Statement:

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin). Its general format is:

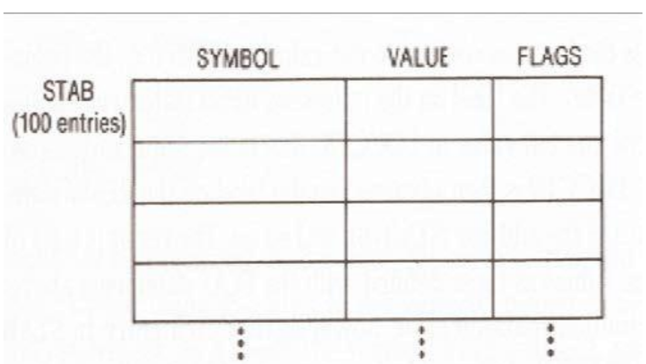
ORG value

Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

SYMBOL 6 Bytes
VALUE 3 Bytes
FLAG 2 Bytes

The table looks like the one given below.



The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the ttable can be reserved by the statement:

```
STAB      RESB      1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

```
SYMBOL    EQU       STAB
VALUE     EQU       STAB+6
FLAGS     EQU       STAB+9
```

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
LDA       VALUE, X
```

The same thing can also be done using ORG statement in the following way:

```
STAB      RESB      1100
          ORG       STAB
SYMBOL    RESB      6
VALUE     RESW      1
FLAG      RESB      2
          ORG       STAB+1100
```

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. For example for the sequence of statements below:

```
          ORG       ALPHA
BYTE1     RESB      1
BYTE2     RESB      1
BYTE3     RESB      1
          ORG
ALPHA     RESB      1
```

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

3.3 .Expressions:

Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /. Division is usually defined to produce an integer result. Individual terms may be constants, user-defined symbols, or special terms. The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement

```
BUFFEND   EQU       *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either absolute expression or relative expressions depending on the type of value they produce.

Absolute Expressions: The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. Example:

```
MAXLEN    EQU       BUFEND-BUFFER
```

In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute immaterial o the relocation of the program. The expression can have only absolute terms. Example:

```
MAXLEN    EQU       1000
```

Relative Expressions: All the relative terms except one can be paired as described in “absolute”. The remaining unpaired relative term must have a positive sign. Example:

```
STAB      EQU       OPTAB + (BUFEND – BUFFER)
```

Handling the type of expressions: to find the type of expression, we must keep track the type of symbols used. This can be achieved by defining the type in the symbol table against each of the symbol as shown in the table below:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

3.4 Program Blocks:

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

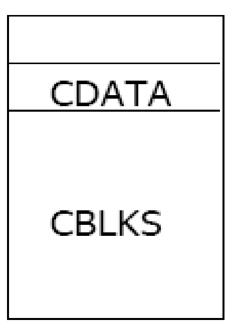
Assembler Directive USE:

USE [blockname]

At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions
CDATA: all data areas that are less in length
CBLKS: all data areas that consists of larger blocks of memory



Example Code

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

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

```
                                                    (default) block
004D    0                   USE
004D    0    WRREC          CLEAR     X             B410
004F    0                   LDT       LENGTH        772017
0052    0    WLOOP          TD        =X'05'        E3201B
0055    0                   JEQ       WLOOP         332FFA
0058    0                   LDCH      BUFFER,X      53A016
005B    0                   WD        =X'05'        DF2012
005E    0                   TIXR      T             B850
0060    0                   JLT       WLOOP         3B2FEF
0063    0                   RSUB                    4F0000
0007    1                   USE       CDATA    <--  CDATA block
                            LTORG
0007    1    *              =C'EOF'                 454F46
000A    1    *              =X'05'                  05
                            END       FIRST
```

Arranging code into program blocks:

Pass 1

- 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

Pass 2

- 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

3.5 Control Sections:

A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references.*

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive

- assembler directive: **CSECT**

The syntax

secname CSECT

- separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

Implicitly defined as an external symbol
first control section

```
COPY      START   0                        COPY FILE FROM INPUT TO OUTPUT
          EXTDEF  BUFFER,BUFEND,LENGTH
          EXTREF  RDREC,WRREC
FIRST     STL     RETADR                   SAVE RETURN ADDRESS
CLOOP     +JSUB   RDREC                    READ INPUT RECORD
          LDA     LENGTH                   TEST FOR EOF (LENGTH=0)
          COMP    #0
          JEQ     ENDFIL                   EXIT IF EOF FOUND
          +JSUB   WRREC                    WRITE OUTPUT RECORD
          J       CLOOP                    LOOP
ENDFIL    LDA     =C'EOF'                  INSERT END OF FILE MARKER
          STA     BUFFER
          LDA     #3                       SET LENGTH = 3
          STA     LENGTH
          +JSUB   WRREC                    WRITE EOF
          J       @RETADR                  RETURN TO CALLER
RETADR    RESW    1
LENGTH    RESW    1                        LENGTH OF RECORD
          LTORG
BUFFER    RESB    4096                     4096-BYTE BUFFER AREA
BUFEND    EQU     *
MAXLEN    EQU     BUFFEND-BUFFER
```

Implicitly defined as an external symbol
second control section

```
RDREC     CSECT
.
.         SUBROUTINE TO READ RECORD INTO BUFFER

          EXTREF  BUFFER,LENGTH,BUFFEND
          CLEAR   X                        CLEAR LOOP COUNTER
          CLEAR   A                        CLEAR A TO ZERO
          CLEAR   S                        CLEAR S TO ZERO
          LDT     MAXLEN
RLOOP     TD      INPUT                    TEST INPUT DEVICE
          JEQ     RLOOP                    LOOP UNTIL READY
          RD      INPUT                    READ CHARACTER INTO REGISTER A
          COMPR   A,S                      TEST FOR END OF RECORD (X'00')
          JEQ     EXIT                     EXIT LOOP IF EOR
          +STCH   BUFFER,X                 STORE CHARACTER IN BUFFER
          TIXR    T                        LOOP UNLESS MAX LENGTH HAS
          JLT     RLOOP                      BEEN REACHED
EXIT      +STX    LENGTH                   SAVE RECORD LENGTH
          RSUB                             RETURN TO CALLER
INPUT     BYTE    X'F1'                    CODE FOR INPUT DEVICE
MAXLEN    WORD    BUFFEND-BUFFER
```

Implicitly defined as an external symbol
third control section

```
WRREC       CSECT
.
.           SUBROUTINE TO WRITE RECORD FROM BUFFER

            EXTREF      LENGTH,BUFFER
            CLEAR       X                   CLEAR LOOP COUNTER
            +LDT        LENGTH
WLOOP       TD          =X'05'              TEST OUTPUT DEVICE
            JEQ         WLOOP               LOOP UNTIL READY
            +LDCH       BUFFER,X            GET CHARACTER FROM BUFFER
            WD          =X'05'              WRITE CHARACTER
            TIXR        T                   LOOP UNTIL ALL CHARACTERS HAVE
            JLT         WLOOP                  BEEN WRITTEN
            RSUB                            RETURN TO CALLER
            END         FIRST
```

Handling External Reference

Case 1

15 0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference.
  - The assembler has no idea where RDREC is
  - inserts an address of zero
  - can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

Case 2

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler inserts a value of zero
- passes information to the loader
- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

Case 3

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

107 1000 MAXLEN EQU BUFEND-BUFFER

Object Code for the example program:

```
0000  COPY     START   0
               EXTDEF  BUFFER,BUFFEND,LENGTH
               EXTREF  RDREC,WRREC
0000  FIRST    STL     RETADR                  172027
0003  CLOOP   +JSUB    RDREC                   4B100000   Case 1
0007           LDA     LENGTH                  032023
000A           COMP    #0                      290000
000D           JEQ     ENDFIL                  332007
0010          +JSUB    WRREC                   4B100000
0014           J       CLOOP                   3F2FEC
0017  ENDFIL   LDA     =C'EOF'                 032016
001A           STA     BUFFER                  0F2016
001D           LDA     #3                      010003
0020           STA     LENGTH                  0F200A
0023          +JSUB    WRREC                   4B100000
0027           J       @RETADR                 3E2000
002A  RETADR   RESW    1
002D  LENGTH   RESW    1
               LTORG
0030  *       =C'EOF'                          454F46
0033  BUFFER   RESB    4096
1033  BUFEND   EQU     *
1000  MAXLEN   EQU     BUFEND-BUFFER

0000  RDREC    CSECT
  .            SUBROUTINE TO READ RECORD INTO BUFFER
  .
               EXTREF  BUFFER,LENGTH,BUFEND
0000           CLEAR   X                       B410
0002           CLEAR   A                       B400
0004           CLEAR   S                       B440
0006           LDT     MAXLEN                  77201F
0009  RLOOP    TD      INPUT                   E3201B
000C           JEQ     RLOOP                   332FFA
000F           RD      INPUT                   DB2015
0012           COMPR   A,S                     A004
0014           JEQ     EXIT                    332009
0017          +STCH    BUFFER,X                57900000
001B           TIXR    T                       B850
001D           JLT     RLOOP                   3B2FE9
0020  EXIT    +STX     LENGTH                  13100000
0024           RSUB                            4F0000
0027  INPUT    BYTE    X'F1'                   F1
0028  MAXLEN   WORD    BUFFEND-BUFFER          000000     Case 2
```

```
0000     WRREC    CSECT

         .        SUBROUTINE TO WRITE RECORD FROM BUFFER
         .
                   EXTREF   LENGTH,BUFFER
0000               CLEAR    X                                    B410
0002              +LDT      LENGTH                               77100000
0006     WLOOP     TD      =X'05'                                E32012
0009               JEQ      WLOOP                                332FFA
000C              +LDCH     BUFFER,X                             53900000
0010               WD      =X'05'                                DF2008
0013               TIXR     T                                    B850
0015               JLT      WLOOP                                3B2FEE
0018               RSUB                                          4F0000
                   END      FIRST
001B     *        =X'05'                                         05
```

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

Define record (EXTDEF)

- Col. 1 D
- Col. 2-7 Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

Refer record (EXTREF)

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

Modification record

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified (hexadecimal)
- Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
- Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.

A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed:

adding or subtracting the value of some external symbol. The symbol used for modification my be defined either in this control section or in another section.

The object program is shown below. There is a separate object program for each of the control sections. In the *Define Record* and *refer record* the symbols named in EXTDEF and EXTREF are included.

In the case of *Define*, the record also indicates the relative address of each external symbol within the control section.

For EXTREF symbols, no address information is available. These symbols are simply named in the *Refer record.*

```
COPY
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000

RDREC
H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND  }
M^000028^06^-BUFFER  }  BUFEND - BUFFER
E

WRREC
H^WRREC ^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

Handling Expressions in Multiple Control Sections:

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).

When it comes in a program having multiple control sections then we have an extended restriction that:

- Both terms in each pair of an expression must be within the same control section
  - o If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located.
    - **Legal:** BUFEND-BUFFER (both are in the same control section)
  - o If the terms are located in different control sections, their difference has a value that is unpredictable.
    - **Illegal:** RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.
- **How to enforce this restriction**
  - o When an expression involves external references, the assembler cannot determine whether or not the expression is legal.
  - o The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
  - o The loader checks the expression for errors and finishes the evaluation.

3.6. ASSEMBLER DESIGN

Here we are discussing

- o The structure and logic of one-pass assembler. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program.
- o Notion of a multi-pass assembler, an extension of two-pass assembler that allows an assembler to handle forward references during symbol definition.

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 avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

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

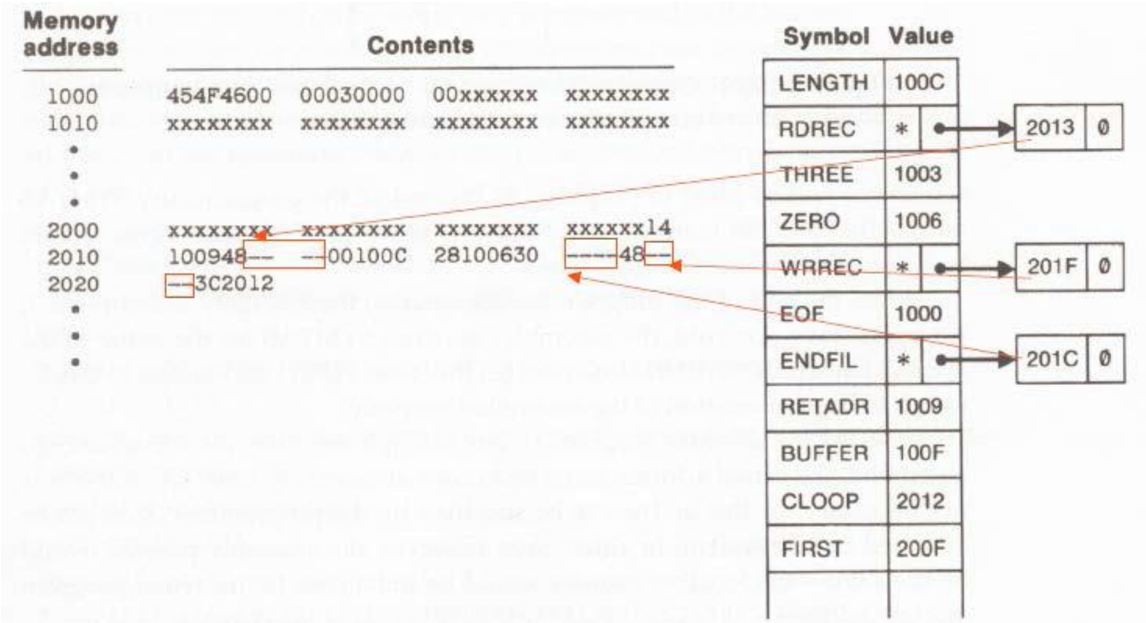
Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110		.			

Forward Reference in One-Pass Assemblers: In load-and-Go assemblers when a forward reference is encountered :

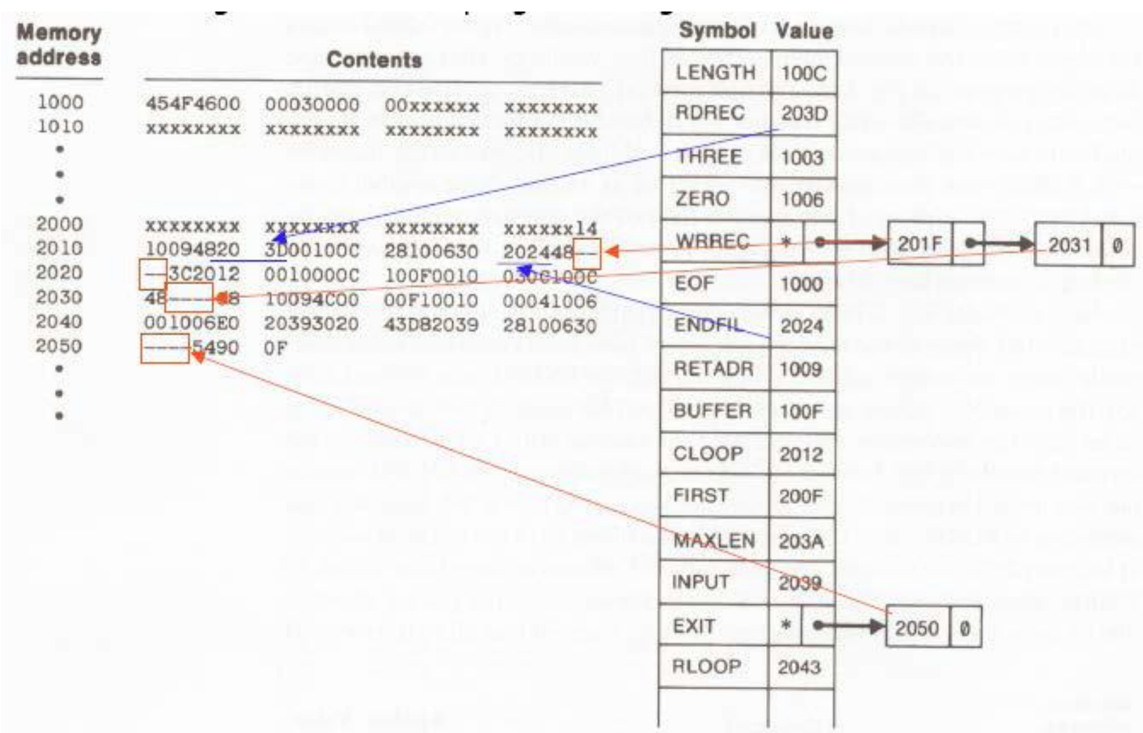
- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, 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

After Scanning line 40 of the program:
40 2021 J` CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:



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.

Object Code Generated by One-Pass Assembler:

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

Multi_Pass Assembler:

- For a two pass assembler, forward references in symbol definition are not allowed:

```
ALPHA    EQU   BETA
BETA     EQU   DELTA
DELTA    RESW  1
```

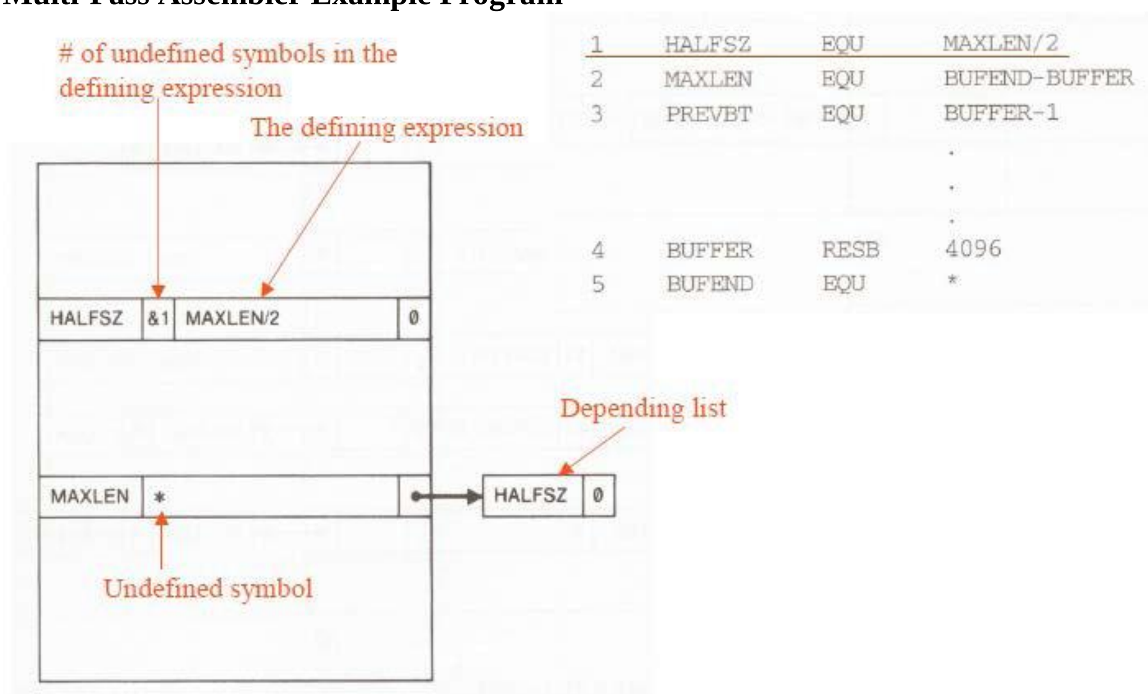
  - Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

Implementation Issues for Modified Two-Pass Assembler:

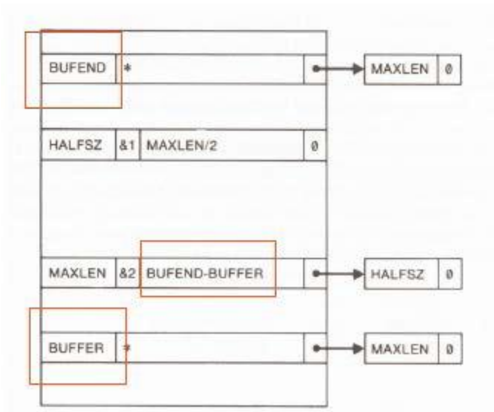
Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

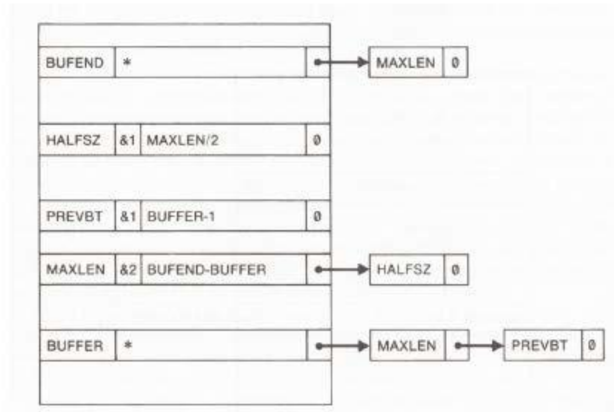
Multi-Pass Assembler Example Program



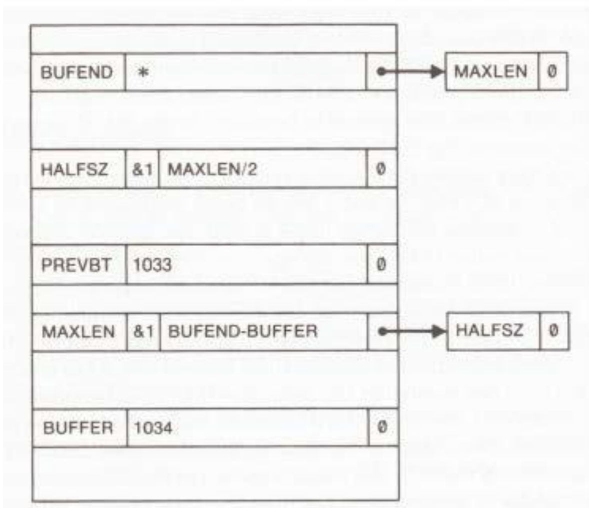
Multi-Pass Assembler (Figure 2.21 of Beck): Example for forward reference in Symbol Defining Statements:



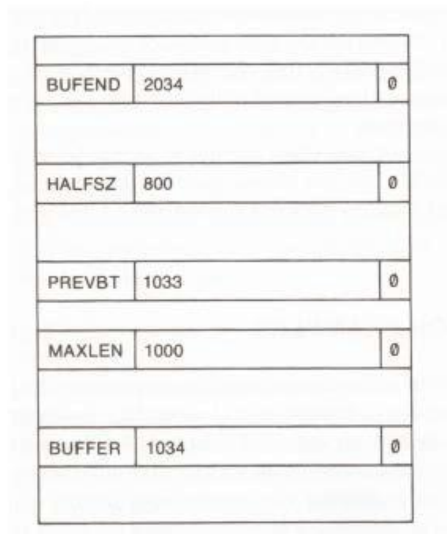
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

UNIT – 4

LOADERS AND LINKERS

4.1. Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

4.2. Basic Loader Functions :

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 4.1. Translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 4.2 the translator is specifically an assembler, which generates the object loaded, which becomes inpu t to the loader. The figure4.3 shows the role of both loader and linker.

Memory

Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

Design of Absolute Loader:

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. 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, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

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

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

Memory address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx ← COPY
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

(b) Program loaded in memory

4.3. Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. 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
```

It uses a subroutine GETC, which is

```
GETC    A←read one character
        if A=0x04 then jump to 0x80
        if A<48 then GETC
        A ← A-48 (0x30)
        if A<10 then return
        A ← A-7
        return
```

4.4. Machine-Dependent Loader Features

Absolute loader is simple and efficient, but the scheme has potential disadvantages One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.

This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions.

Relocation

The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

Methods for specifying relocation

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record

- col 1: M
- col 2-7: relocation address
- col 8-9: length (halfbyte)
- col 10: flag (+/-)
- col 11-17: segment name

```
HΛCOPY Λ000000 001077
TΛ000000 Λ1DΛ17202DΛ69202DΛ48101036Λ…Λ4B105DΛ3F2FECΛ032010
TΛ00001DΛ13Λ0F2016Λ010003Λ0F200DΛ4B10105DΛ3E2003Λ454F46
TΛ001035 Λ1DΛB410ΛB400ΛB440Λ75101000Λ…Λ332008Λ57C003ΛB850
TΛ001053Λ1DΛ3B2FEAΛ134000Λ4F0000ΛF1Λ..Λ53C003ΛDF2008ΛB850
TΛ00070Λ07Λ3B2FEFΛ4F0000Λ05
MΛ000007Λ05+COPY
MΛ000014Λ05+COPY
MΛ000027Λ05+COPY
EΛ000000
```

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 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.

<u>Text record:</u>

col 1: T
col 2-7: starting address
col 8-9: length (byte)
col 10-12: relocation bits
col 13-72: object code

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. For absolute loader, there are no relocation bits column 10-69 contains object code. The object program with relocation by bit mask is as shown below. Observe FFC - means all ten words are to be modified and, E00 - means first three records are to be modified.

```
H^COPY                ^000000                00107A
T^000000^1E^FFC^140033^481039^000036^280030^300015^…^3C0003  ^ …
T^00001E^15^E00^0C0036^481061^080033^4C0000^…^000003^000000
T^001039^1E^FFC^040030^000030^…^30103F^D8105D^280030^...
T^001057^0A^ 800^100036^4C0000^F1^001000
T^001061^19^FE0^040030^E01079^…^508039^DC1079^2C0036^... E^000000
```

Program Linking

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH
EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC
EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here**.**

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

Example records

D LISTA 000040 ENDA 000054
D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here**.**

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section
Col. 8-73	Name of other external reference symbols

Example records

R LISTB ENDB LISTC ENDC
R LISTA ENDA LISTC ENDC
R LISTA ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation. The following figures give the sample programs and their corresponding object programs. Observe the object programs, which contain D and R records along with other records.

```
0000   PROGA      START          0
                  EXTDEF     LISTA, ENDA
                  EXTREF     LISTB, ENDB, LISTC, ENDC
                  ………..
                  ……….
0020   REF1       LDA        LISTA                             03201D
0023   REF2       +LDT       LISTB+4                         77100004
0027   REF3       LDX        #ENDA-LISTA                       050014
                     .
                     .
0040   LISTA      EQU        *

0054   ENDA       EQU        *
0054   REF4       WORD           ENDA-LISTA+LISTC              000014
0057   REF5       WORD           ENDC-LISTC-10                 FFFFF6
005A   REF6       WORD           ENDC-LISTC+LISTA-1            00003F
005D   REF7       WORD           ENDA-LISTA-(ENDB-LISTB)       000014
0060   REF8       WORD           LISTB-LISTA                   FFFFC0
                   END       REF1
```

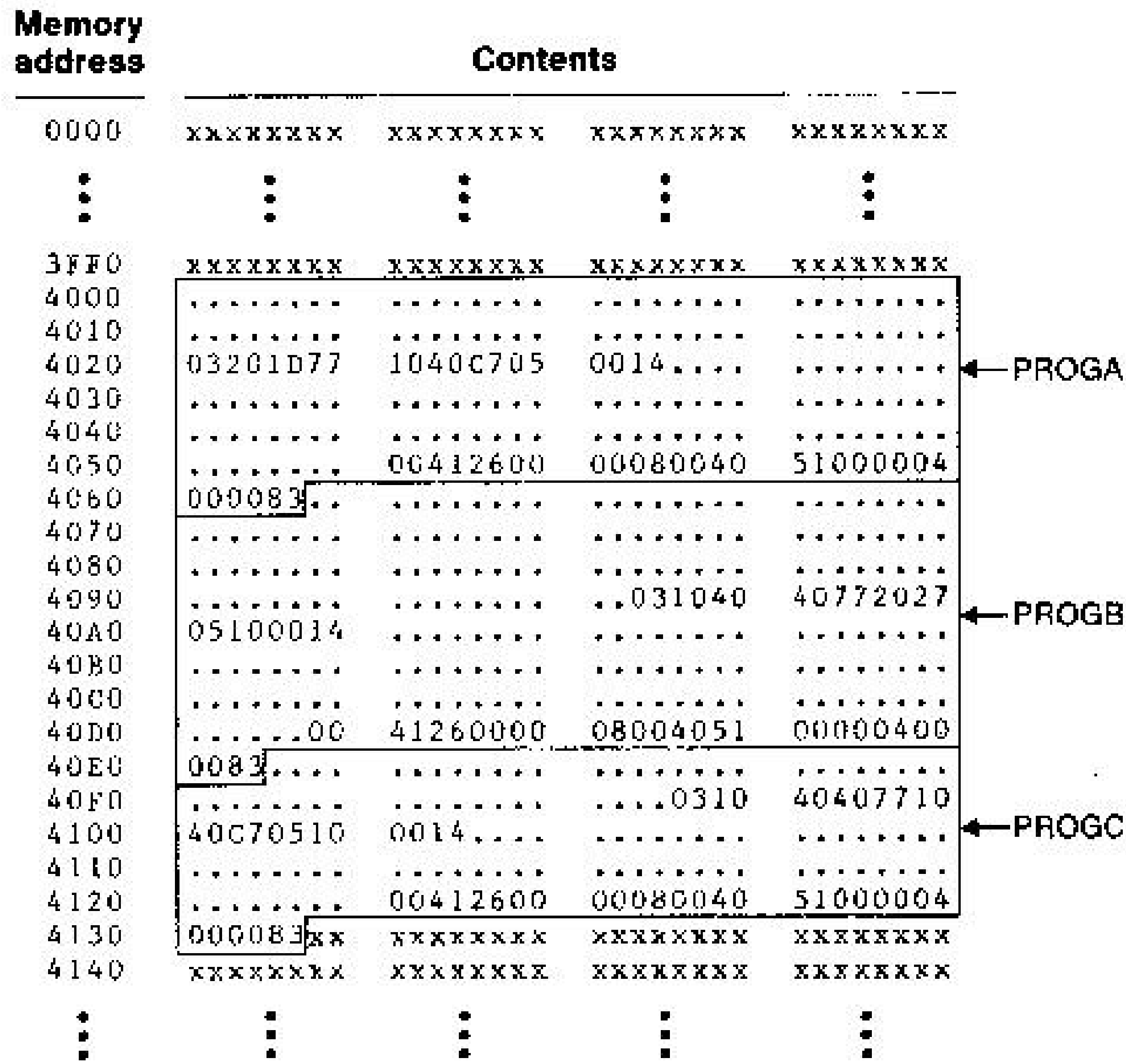
```
0000   PROGB      START          0
                  EXTDEF     LISTB, ENDB
                  EXTREF     LISTA, ENDA, LISTC, ENDC
                  ………..
                  ……….
0036   REF1       +LDA       LISTA                           03100000
003A   REF2       LDT        LISTB+4                           772027
003D   REF3       +LDX       #ENDA-LISTA                     05100000
                     .
                     .
0060   LISTB      EQU        *

0070   ENDB       EQU        *
0070   REF4       WORD           ENDA-LISTA+LISTC              000000
0073   REF5       WORD           ENDC-LISTC-10                 FFFFF6
0076   REF6       WORD           ENDC-LISTC+LISTA-1            FFFFFF
0079   REF7       WORD           ENDA-LISTA-(ENDB-LISTB)       FFFFF0
007C   REF8       WORD           LISTB-LISTA                   000060
                  END
```

```
0000   PROGC      START             0
                  EXTDEF    LISTC, ENDC
                  EXTREF    LISTA, ENDA, LISTB, ENDB
                  ………..
                  ………..
0018   REF1       +LDA      LISTA                              03100000
001C   REF2       +LDT      LISTB+4                            77100004
0020   REF3       +LDX      #ENDA-LISTA                        05100000
                      .
                      .
0030   LISTC      EQU       *

0042   ENDC       EQU       *
0042   REF4       WORD          ENDA-LISTA+LISTC               000030
0045   REF5       WORD          ENDC-LISTC-10                  000008
0045   REF6       WORD          ENDC-LISTC+LISTA-1             000011
004B   REF7       WORD          ENDA-LISTA-(ENDB-LISTB)        000000
004E   REF8       WORD          LISTB-LISTA                    000000
                   END
```

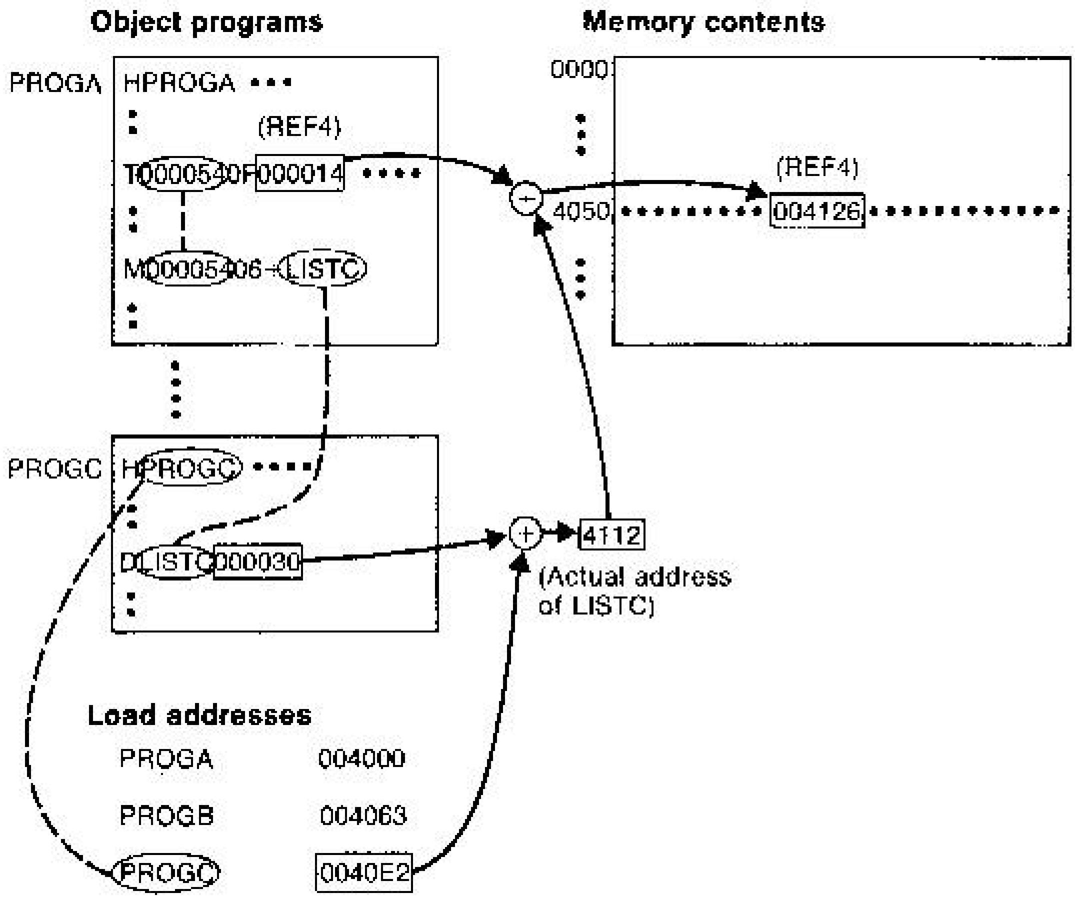
```
H PROGA 000000 000063
D LISTA    000040 ENDA    000054
R LISTB    ENDB   LISTC   ENDC
.
.
T 000020 0A 03201D 77100004 050014
.
.
T 000054 0F 000014 FFFF6 00003F 000014 FFFFC0
M000024    05+LISTB
M000054    06+LISTC
M000057    06+ENDC
M000057  06   -LISTC
M00005A06+ENDC
M00005A06    -LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

```
H PROGB 000000 00007F
D LISTB    000060 ENDB    000070
R LISTA    ENDA   LISTC   ENDC
.
T 000036 0B 03100000 772027 05100000
.
T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037    05+LISTA
M00003E    06+ENDA
M00003E  06  -LISTA
M000070  06  +ENDA
M000070  06  -LISTA
M000070  06  +LISTC
M000073  06  +ENDC
M000073  06  -LISTC
M000073  06  +ENDC
M000076  06  -LISTC
M000076    06+LISTA
M000079    06+ENDA
M000079  06  -LISTA
M00007C  06+PROGB
M00007C 06-LISTA
E

H PROGC 000000 000051
D LISTC    000030 ENDC    000042
R LISTA    ENDA   LISTB   ENDB
.
T 000018 0C 03100000 77100004 05100000
.
T 000042 0F 000030 000008 000011 000000 000000
M000019   05+LISTA
M00001D   06+LISTB
M000021   06+ENDA
M000021  06  -LISTA
M000042   06+ENDA
M000042  06  -LISTA
M000042  06+PROGC
M000048   06+LISTA
M00004B   06+ENDA
M00004B  006-LISTA
M00004B    06-ENDB
M00004B   06+LISTB
M00004E   06+LISTB
M00004E 06-LISTA
E
```

The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROG B and PROGC immediately following.



For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.



The initial value from the Text record T0000540F000014FFFFF600003F000014FFFFC0 is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126.

That is REF4 in PROGA is ENDA-LISTA+LISTC=4054-4040+4112=4126.

Similarly the load address for symbols LISTA: PROGA+0040=4040, LISTB: PROGB+0060=40C3 and LISTC: PROGC+0030=4112

Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

4.5. Algorithm and Data structures for a Linking Loader

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

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

Pass 1: Assign addresses to all external symbols
Pass 2: Perform the actual loading, relocation, and linking

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

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

Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record. The algorithm for Pass 1 of Linking Loader is given below.

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type () 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while () 'E'}
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its address. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs. The algorithm for Pass 2 of Linking Loader is given below.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record    {Header record}
        set CSLTH to control section length
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        {if object code is in character form, convert
                            into internal representation}
                        move object code from record to location
                            (CSADDR + specified address)
                    end {if 'T'}
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end  {if 'M'}
            end  {while <> 'E'}
        if an address is specified {in End record} then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded progra
end  {Pass 2}
```

Improve Efficiency, How?

The question here is can we improve the efficiency of the linking loader. Also observe that, even though we have defined Refer record (R), we haven't made use of it. The efficiency can be improved by the use of local searching instead of multiple searches of ESTAB for the same symbol. For implementing this we assign a reference number to each external symbol in the Refer record. Then this reference number is used in Modification records instead of external symbols. 01 is assigned to control section name, and other numbers for external reference symbols.

The object programs for PROGA, PROGB and PROGC are shown below, with above modification to Refer record (Observe R records).

```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
:
T0000360B0310000077202705100000
:
T0000700F000000FFFFF6FFFFFFFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M00007006+03
M00007006-02
M00007006+04
M00007306+05
M00007306-04
M00007606+05
M00007606-04
M00007606+02
M00007906+03
M00007906-02
M00007C06+01
M00007C06-02
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
:
T0000200A03201D77100004050014
:
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+02
M00005406+04
M00005706+05
M00005706-04
```

```
HPROGC ^000000^000051
D^LISTC ^000030^ENDC  ^000042
R^02LISTA ^03ENDA  ^04LISTB ^05ENDB
.
.
T^000018^0C^031000^000771000004^051000000
.
.
T^000042^0F^000030^000008^000011^000000^000000
M^000019^05^+02
M^00001D^05^+04
M^000021^05^+03
M^000021^05^-02
M^000042^06^+03
M^000042^06^-02
M^000042^06^+01
M^000048^06^+02
M^00004B^06^+03
M^00004B^06^-02
M^00004B^06^-05
M^00004B^06^+04
M^00004E^06^+04
M^00004E^06^-02
E
```

Symbol and Addresses in PROGA, PROGB and PROGC are as shown below. These are the entries of ESTAB. The main advantage of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section

Ref No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

4.6. Machine-independent Loader Features

Here we discuss some loader features that are not directly related to machine architecture and design. Automatic Library Search and Loader Options are such Machine-independent Loader Features.

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 subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program.

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 an 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 pf programs being loaded

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

LIBRARY MYLIB – search MYLIB library before standard libraries

NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines

Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

```
LIBRARY          UTLIB
INCLUDE  READ  (UTLIB)
INCLUDE  WRITE (UTLIB)
```

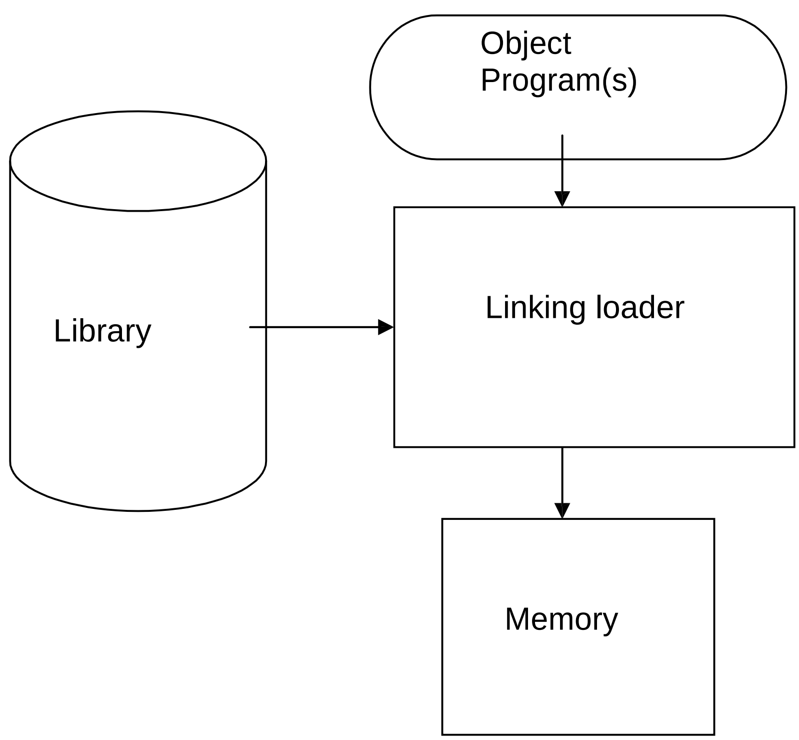
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
NOCALL SQRT, PLOT

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.

4.7 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

Linking Loaders

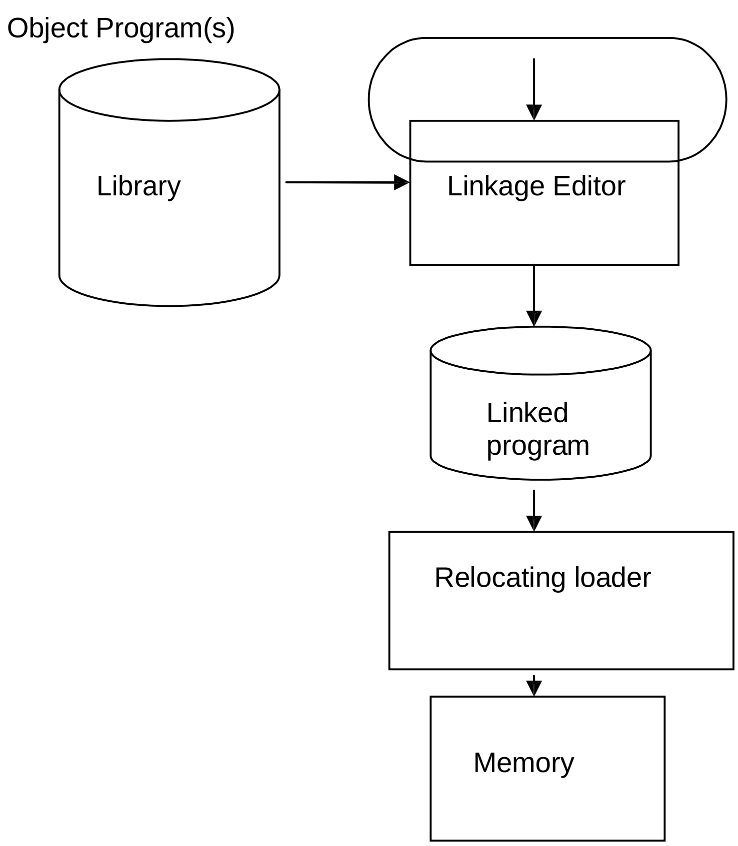


The above diagram shows the processing of an object program using Linking Loader. The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution.

Linkage Editors

The figure below shows the processing of an object program using 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. The linked program produced is generally in a form that is suitable for processing by a relocating loader.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source 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 – linking will be done later by linking loader – linkage editor + linking loader – savings in space



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 – 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.

Bootstrap Loaders

If the question, how is the loader itself loaded into the memory ? is asked, then the answer is, when computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution. The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

4.8. Implementation Examples

This section contains brief description of loaders and linkers for actual computers. They are, MS-DOS Linker - Pentium architecture, SunOS Linkers - SPARC architecture, and, Cray MPP Linkers – T3E architecture.

MS-DOS Linker

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.

UNIT – 5

EDITORS AND DEBUGGING SYSTEMS

5.1 Introduction

An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.

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

Text Editors:

- An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.

- A text editor allows you to edit a text file (create, modify etc…). For example the Interactive text editors on Windows OS - Notepad, WordPad, Microsoft Word, and text editors on UNIX OS - vi, emacs , jed, pico.

- Normally, the common editing features associated with text editors are, Moving the cursor, Deleting, Replacing, Pasting, Searching, Searching and replacing, Saving and loading, and, Miscellaneous(e.g. quitting).

5.2. Overview of the editing process

An interactive editor is a computer program that allows a user to create and revise a target document. Document includes objects such as computer diagrams, text, equations tables, diagrams, line art, and photographs. In text editors, character strings are the primary elements of the target text.

Document-editing process in an interactive user-computer dialogue has four tasks:

- Select the part of the target document to be viewed and manipulated
- Determine how to format this view on-line and how to display it
- Specify and execute operations that modify the target document
- Update the view appropriately

The above task involves traveling, filtering and formatting. Editing phase involves – insert, delete, replace, move, copy, cut, paste, etc…

- Traveling – locate the area of interest
- Filtering - extracting the relevant subset
- Formatting – visible representation on a display screen

There are two types of editors. Manuscript-oriented editor and program oriented editors. Manuscript-oriented editor is associated with characters, words, lines, sentences and paragraphs. Program-oriented editors are associated with identifiers, keywords, statements. User wish – what he wants – formatted.

5.3. User Interface:

Conceptual model of the editing system provides an easily understood abstraction of the target document and its elements. For example, Line editors – simulated the world of the key punch – 80 characters, single line or an integral number of lines, Screen editors – Document is represented as a quarter-plane of text lines, unbounded both down and to the right.

The user interface is concerned with, the input devices, the output devices and, the interaction language. The input devices are used to enter elements of text being edited, to enter commands. The output devices, lets the user view the elements being edited and the results of the editing operations and, the interaction language provides communication with the editor.

Input Devices are divided into three categories:

- text devices
- button devices
- locator devices.

1. Text Devices are keyboard. Button Devices are special function keys, symbols on the screen. Locator Devices are mouse, data tablet. There are voice input devices which translates spoken words to their textual equivalents.

2. Output Devices are Teletypewriters(first output devices), Glass teletypes (Cathode ray tube (CRT) technology), Advanced CRT terminals, TFT Monitors and Printers (Hard-copy).

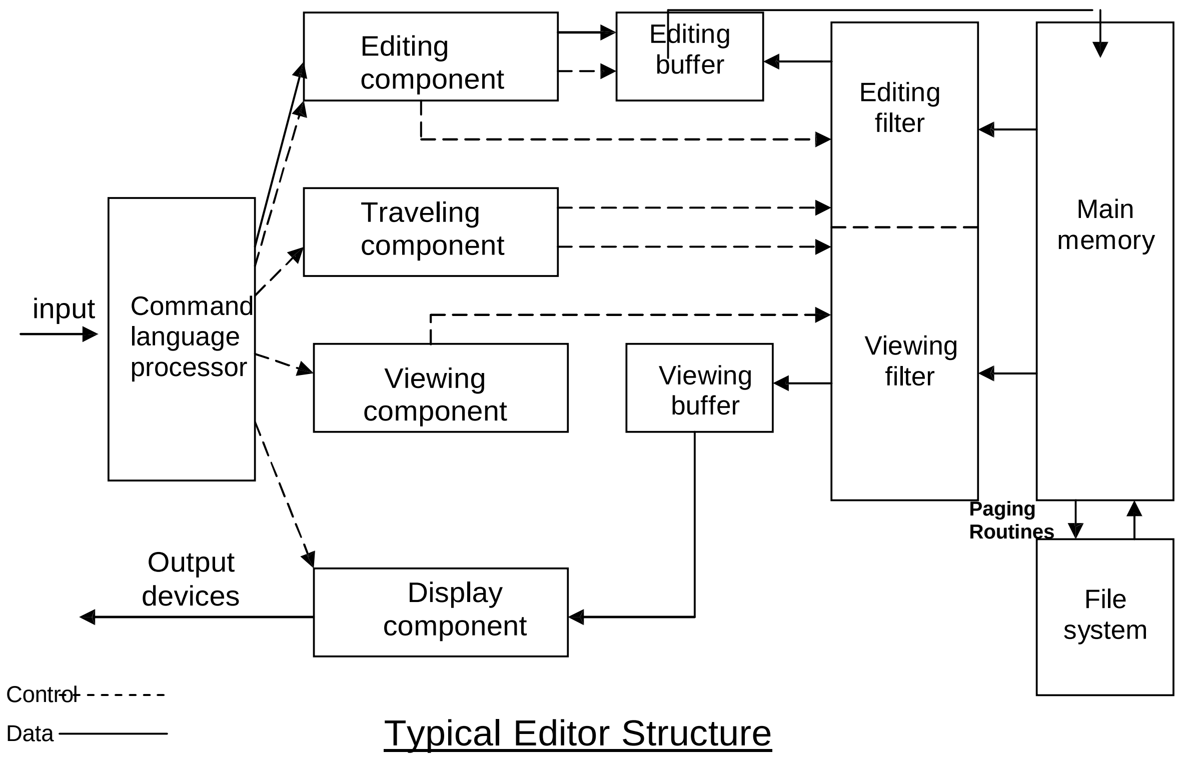
3. The interaction language could be, typing oriented or text command oriented and menu-oriented user interface. Typing oriented or text command oriented interaction was with oldest editors, in the form of use of commands, use of function keys, control keys etc.

4. <u>Menu-oriented</u> user interface has menu with a multiple choice set of text strings or icons. Display area for text is limited. Menus can be turned on or off.

5.4. Editor Structure:

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

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



Typical Editor Structure

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

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

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

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

- When display needs to be updated, viewing component invokes the viewing filter – generates a new viewing buffer – contains part of the document to be viewed from current viewing pointer. In case of line editors – viewing buffer may contain the current line, Screen editors - viewing buffer contains a rectangular cutout of the quarter plane of the text.

- Viewing buffer is then passed to the display component of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen – called a window. Identical – user edits the text directly on the screen. Disjoint – Find and Replace (For example, there are 150 lines of text, user is in 100th line, decides to change all occurrences of ‘text editor’ with ‘editor’).

- The editing and viewing buffers can also be partially overlap, or one may be completely contained in the other. Windows typically cover entire screen or a rectangular portion of it. May show different portions of the same file or portions of different file. Inter-file editing operations are possible.

- The components of the editor deal with a user document on two levels: In main memory and in the disk file system. Loading an entire document into main memory may be infeasible – only part is loaded – demand paging is used – uses editor paging routines.

- Documents may not be stored sequentially as a string of characters. Uses separate editor data structure that allows addition, deletion, and modification with a minimum of I/O and character movement.

Types of editors based on computing environment

Editors function in three basic types of computing environments:

1. Time sharing
2. Stand-alone
3. Distributed.

Each type of environment imposes some constraints on the design of an editor.

- In time sharing environment, editor must function swiftly within the context of the load on the computer's processor, memory and I/O devices.

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

Interactive Debugging Systems:

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

Here we discuss

- Introducing important functions and capabilities of IDS
- Relationship of IDS to other parts of the system
- The nature of the user interface for IDS

5.5.Debugging Functions and Capabilities:

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

A Debugging system should also provide functions such as tracing and traceback .

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

Program-Display capabilities

A debugger should have good program-display capabilities.

- Program being debugged should be displayed completely with statement numbers.
- The program may be displayed as originally written or with macro expansion.
- Keeping track of any changes made to the programs during the debugging session. Support for symbolically displaying or modifying the contents of

any of the variables and constants in the program. Resume execution – after these changes.

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

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

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

Examples of assignment statements

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

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

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

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

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

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

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

5.6. Relationship with Other Parts of the System:

- The important requirement for an interactive debugger is that it always be available. Must appear as part of the run-time environment and an integral part of the system.

- When an error is discovered, immediate debugging must be possible. The debugger must communicate and cooperate with other operating system components such as interactive subsystems.

- Debugging is more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops.

- The debugger must also exist in a way that is consistent with the security and integrity components of the system.

- The debugger must coordinate its activities with those of existing and future language compilers and interpreters.

5.7. User-Interface Criteria:

- Debugging systems should be simple in its organization and familiar in its language, closely reflect common user tasks.

- The simple organization contribute greatly to ease of training and ease of use.

- The user interaction should make use of full-screen displays and windowing-systems as much as possible.

- With menus and full-screen editors, the user has far less information to enter and remember. There should be complete functional equivalence between commands and menus – user where unable to use full-screen IDSs may use commands.

- The command language should have a clear, logical and simple syntax.

- command formats should be as flexible as possible.

- Any good IDSs should have an on-line HELP facility. HELP should be accessible from any state of the debugging session.

UNIT–6

MACRO PROCESSOR

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.
- The design of a macro processor generally is *machine independent!*
- 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

```
name   MACRO   parameters
       :
       body
       :
MEND
```

    - Body: the statements that will be generated as the expansion of the macro.

6.1. Basic Macro Processor Functions:

- *Macro Definition and Expansion*
- *Macro Processor Algorithms and Data structures*

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
   .
```

Fig 4.1

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 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.

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.

The difference between *Macros* and *Subroutines* is that the statement s from the body of the Macro is expanded the number of times the macro invocation is encountered, whereas the statement of the subroutine appears only once no matter how many times the subroutine is called. Macro instructions will be written so that the body of the macro contains no labels.

- Problem of the label in the body of macro:
  - If the same macro is expanded multiple times at different places in the program …
  - There will be *duplicate labels*, which will be treated as errors by the assembler.
- Solutions:

Do not use labels in the body of macro.
  - Explicitly use PC-relative addressing instead.
- Ex, in RDBUFF and WRBUFF macros,
  - JEQ *+11
  - JLT *-14
- It is inconvenient and error-prone.

The following program shows the concept of Macro Invocation and Macro Expansion.

```
.                               MAIN PROGRAM
.
          FIRST     STL        RETADR              SAVE RETURN ADDRESS
          CLOOP     RDBUFF     F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
                    LDA        LENGTH              TEST FOR END OF FILE
                    COMP       #0
                    JEQ        ENDFIL              EXIT IF EOF FOUND
                    WRBUFF     05,BUFFER,LENGTH    WRITE OUTPUT RECORD
                    J          CLOOP               LOOP
          ENDFIL    WRBUFF     05,EOF,THREE        INSERT EOF MARKER
                    J          @RETADR
          EOF       BYTE       C'EOF'
          THREE     WORD       3
          RETADR    RESW       1
          LENGTH    RESW       1                   LENGTH OF RECORD
          BUFFER    RESB       4096                4096-BYTE BUFFER AREA
                    END        FIRST
```

5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
180	FIRST	STL	RETADR	SAVE RETURN ADDRESS
190	.CLOOP	RDBUFF	F1,BUFFER,LENGTH	READ RECORD INTO BUFFER
190a	CLOOP	CLEAR	X	CLEAR LOOP COUNTER
190b		CLEAR	A	
190c		CLEAR	S	
190d		+LDT	#4096	SET MAXIMUN RECORD LENGTH
190e		TD	=X'F1'	TEST INPUT DEVICE
190f		JEQ	*-3	LOOP UNTIL READY
190g		RD	=X'F1'	TEST FOR END OF RECORD
190h		COMPR	A, S	TEST FOR END OF RECORD
190i		JEQ	*+11	EXIT LOOP IF EOR
190j		STCH	BUFFER, X	STORE CHARACTER IN BUFFER
190k		TIXR	T	LOOP UNLESS MAXIMUN LENGTH
190l		JLT	*-19	HAS BEEN REACHED
190M		STX	LENGTH	SAVE RECORD LENGTH

Fig 4.2

6.2 Macro Processor Algorithm and Data Structure:

Design can be done as two-pass or a one-pass macro. In case of two-pass assembler.

Two-pass macro processor

- You may design a two-pass macro processor
  - Pass 1:
    - Process all macro definitions
  - Pass 2:
    - Expand all macro invocation statements
- However, one-pass may be enough
  - Because all macros would have to be defined during the first pass before any macro invocations were expanded.
    - The definition of a macro must appear before any statements that invoke that macro.
- Moreover, the body of one macro can contain definitions of the other macro
- Consider the example of a Macro defining another Macro.
- In the example below, the body of the first Macro (MACROS) contains statement that define RDBUFF, WRBUFF and other macro instructions for SIC machine.
- The body of the second Macro (MACROX) defines the se same macros for SIC/XE machine.
- A proper invocation would make the same program to perform macro invocation to run on either SIC or SIC/XEmachine.

MACROS for SIC machine

```
1       MACROS    MACOR     {Defines SIC standard version macros}
2       RDBUFF    MACRO     &INDEV,&BUFADR,&RECLTH
                  .
                  .         {SIC standard version}
                  .
3                 MEND      {End of RDBUFF}
4       WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
                  .
                  .         {SIC standard version}
5                 MEND      {End of WRBUFF}
                  .
                  .
                  .
6                 MEND      {End of MACROS}
```

Fig 4.3(a)

MACROX for SIC/XE Machine

```
1       MACROX    MACRO     {Defines SIC/XE macros}
2       RDBUFF    MACRO     &INDEV,&BUFADR,&RECLTH
                  .
                  .         {SIC/XE version}
                  .
3                 MEND      {End of RDBUFF}
4       WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
                  .
                  .         {SIC/XE version}
                  .
5                 MEND      {End of WRBUFF}
                  .
                  .
6                 MEND      {End of MACROX}
```

Fig 4.3(b)

- A program that is to be run on SIC system could invoke MACROS whereas a program to be run on SIC/XE can invoke MACROX.
- However, defining MACROS or MACROX does not define RDBUFF and WRBUFF.
- These definitions are processed only when an invocation of MACROS or MACROX is expanded.

One-Pass Macro Processor:

- A one-pass macro processor that alternate between *macro definition* and *macro expansion* in a recursive way is able to handle recursive macro definition.
- Restriction
    - The definition of a macro must appear in the source program before any statements that invoke that macro.
    - This restriction does not create any real inconvenience.

The design considered is for one-pass assembler. The data structures required are:

- DEFTAB (Definition Table)
    - Stores the macro definition including *macro prototype* and *macro body*
    - Comment lines are omitted.
    - References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
- NAMTAB (Name Table)
    - Stores macro names
    - Serves as an index to DEFTAB
        - Pointers to the beginning and the end of the macro definition (DEFTAB)

- ARGTAB (Argument Table)
    - Stores the arguments according to their positions in the argument list.
    - As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.
    - The figure below shows the different data structures described and their relationship.

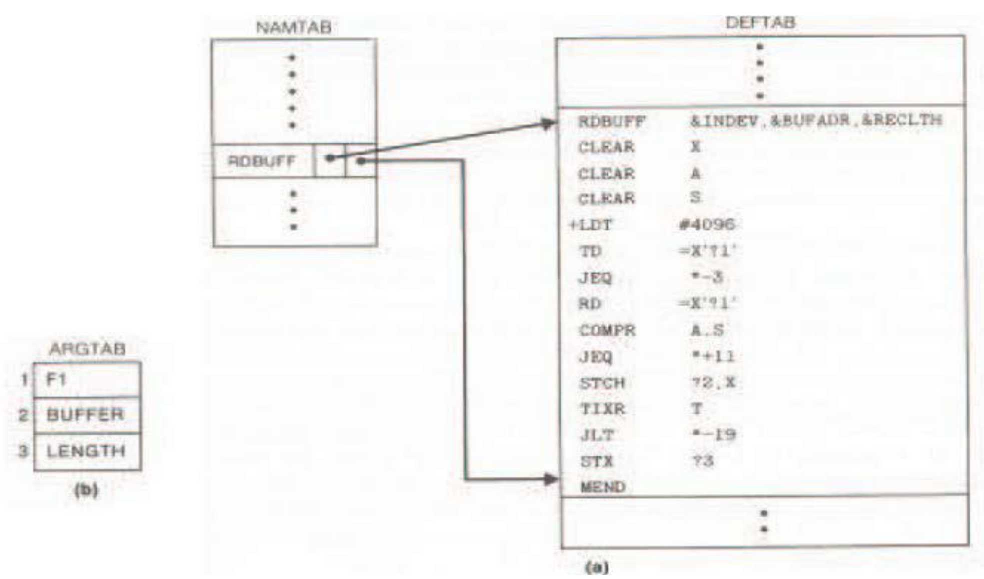


Fig 4.4

- The above figure shows the portion of the contents of the table during the processing of the program in page no. 3. In fig 4.4(a) definition of RDBUFF is stored in DEFTAB, with an entry in NAMTAB having the pointers to the beginning and the end of the definition. The arguments referred by the instructions are denoted by the their positional notations. For example,

  TD =X'?1'

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

- Figure 4.4(b) shows the ARTAB as it would appear during expansion of the RDBUFF statement as given below:

  CLOOP RDBUFF F1, BUFFER, LENGTH

- For the invocation of the macro RDBUFF, the first parameter is F1 (input device code), second is BUFFER (indicating the address where the characters read are stored), and the third is LENGTH (which indicates total length of the record to be read). When the ?n notation is encountered in a line fro DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

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

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

- While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the difintion of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL.

Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

Most macro processors allow thr definitions of the commonly used instructions to appear in a standard system library, rather than in the source program. This makes the use of macros convenient; definitions are retrieved from the library as they are needed during macro processing.

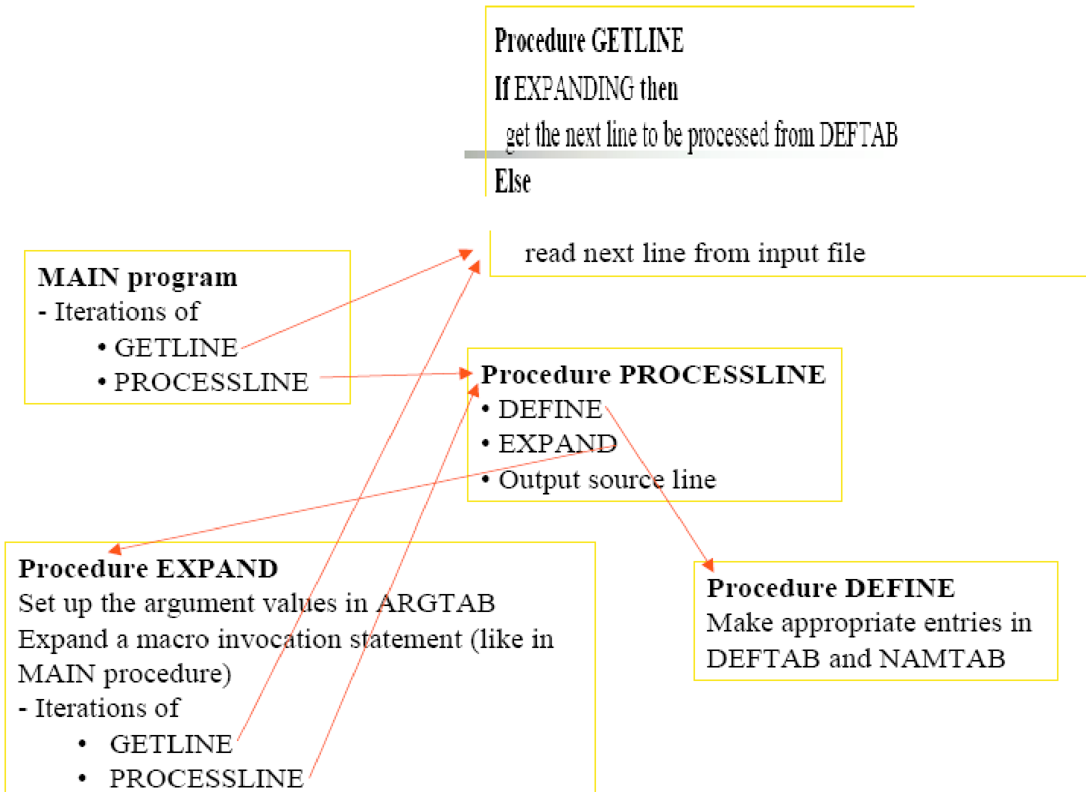

Procedure GETLINE
If EXPANDING then
get the next line to be processed from DEFTAB
Else
read next line from input file
MAIN program
- Iterations of
• GETLINE
• PROCESSLINE
Procedure PROCESSLINE
• DEFINE
• EXPAND
• Output source line
Procedure EXPAND
Set up the argument values in ARGTAB
Expand a macro invocation statement (like in MAIN procedure)
- Iterations of
• GETLINE
• PROCESSLINE
Procedure DEFINE
Make appropriate entries in DEFTAB and NAMTAB

Fig 4.5

Algorithms

```
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}
```

Fig 4.6

6.3.Comparison of Macro Processor Design

- *One-pass algorithm*
    - Every macro must be defined before it is called
    - One-pass processor can alternate between macro definition and macro expansion
    - Nested macro definitions are allowed but nested calls are not allowed.
- *Two-pass algorithm*
    - Pass1: Recognize macro definitions
    - Pass2: Recognize macro calls
    - Nested macro definitions are not allowed

6.4. Machine-independent Macro-Processor Features.

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor. These features are:

- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

Concatenation of unique labels:

- Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction.

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

- Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like
  - LDA X&ID1

Fig 4.7

& is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended.

- If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.

- Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as

```
LDA        X&ID→
```

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

```
SUM  MACRO   &ID
     LDA     X&ID→ 1
     ADD     X&ID→ 2
     ADD     X&ID→ 3
     STA     X&ID→ S
     MEND
```

```
SUM    A                      SUM    BETA
  ↓                             ↓
LDA    XA1                    LDA    XBEATA1
ADD    XA2                    ADD    XBEATA2
ADD    XA3                    ADD    XBEATA3
STA    XAS                    STA    XBEATAS
```

Fig 4.8

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

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…

This allows 1296 macro expansions in a single program.

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

```
RDBUFF  MACRO   &INDEV, &BUFADR, &RECLTH
        CLEAR   X               CLEAR LOOP COUNTER
        CLEAR   A
        CLEAR   S
        +LDT    #4096           SET MAXIMUM RECORD LENGTH
$LOOP   TD      =X'&INDEV'      TEST INPUT DEVICE
        JEQ     $LOOP           LOOP UNTIL READY
        RD      =X'&INDEV'      READ CHARACTER INTI REG A
        COMPR   A, S            TEST FOR END OF RECORD
        JEQ     $EXIT           EXIT LOOP IF EOR
        STCH    &BUFADR, X      STORE CHARACTER IN BUFFER
        TIXR    $LOOP           HAS BEEN REACHED
$EXIT   STX     &RECLTH         SAVE RECORD LENGTH
        MEND
```

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

```
.               RDBUFF    F1, BUFFER, LENGTH

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

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

Conditional Macro Expansion

There are applications of macro processors that are not related to assemblers or assembler programming.

Conditional assembly depends on parameters provides

```
MACRO &COND
……..
  IF (&COND NE '')
        part  I
  ELSE
        part  II
  ENDIF
………
ENDM
```

Part I is expanded if condition part is true, otherwise part II is expanded. Compare operators: NE, EQ, LE, GT.

Macro-Time Variables:

Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion. Any symbol that begins with symbol & and not a macro instruction parameter is considered as *macro-time variable.* All such variables are initialized to zero.

```
RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH
         IF       (&EOR NE ' ')
&EORCK   SET      1
         ENDIF
         CLEAR    X                   CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'            SET EOR COUNTER
         RMO      A, S
         ENDIF
         IF       (&MAXLTH EQ ' ')
        +LDT      #4096               SET MAX LENGTH = 4096
         ELSE
        +LDT      #&MAXLTH            SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD       =X'&INDEV'          TEST INPUT DEVICE
         JEQ      $LOOP               LOOP UNTIL READY
         RD       =X'&INDEV'          READ CHARACTER INTI REG A
         IF       (&EORCK EQ 1)
         COMPR    A, S                TEST FOR END OF RECORD
         JEQ      $EXIT               EXIT LOOP IF EOR
         ENDIF
         STCH     &BUFADR, X          STORE CHARACTER IN BUFFER
         TIXR     T                   LOOP UNLESS MAXIMUN LENGTH
         JLT      $LOOP               HAS BEEN REACHED
$EXIT    STX      &RECLTH             SAVE RECORD LENGTH
         MEND
```

Macro-time variable

Fig 4.9(a)

Figure 4.5(a) gives the definition of the macro RDBUFF with the parameters &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH. According to the above program if &EOR has any value, then &EORCK is set to 1 by using the directive SET, otherwise it retains its default value 0.

```
.               RDBUFF    F31 BUF, RECL, 04, 2048

                CLEAR     X              CLEAR LOOP COUNTER
                CLEAR     A
                LDCH      =X'04'         SET EOR CHARACTER
                RMO       A, S
               +LDT       #2048          SET MAXIMUM RECORD LENGTH
      $AALOOP   TD        =X'F3'         TEST INPUT DEVICE
                JEQ       $AALOOP        LOOP UNTIL READY
                RD        =X'F3'         READ CHARACTER INTI REG A
                COMPR     A, S           TEST FOR END OF RECORD
                JEQ       $AAEXIT        EXIT LOOP IF EOR
                STCH      BUF, X         STORE CHARACTE IN BUFFER
                TIXR      T              LOOP UNLESS MAXIMUM LENGTH
                JLT       $AALOOP        HAS BEEN REACHED
      $AAEXIT   STX       RECL           SAVE RECORD LENGTH
```

Fig 4.9(b) Use of Macro-Time Variable with EOF being NOT NULL

```
.               RDBUFF    OE, BUFFER, LENGTH, , 80

                CLEAR     X              CLEAR LOOP COUNTER
                CLEAR     A
               +LDT       #80            SET MAXIMUM RECORD LENGTH
      $ABLOOP   TD        =X'0E'         TEST INPUT DEVICE
                JEQ       $ABLOOP        LOOP UNTIL READY
                RD        =X'0E'         READ CHARACTER IN REG A
                STCH      BUFFER, X      STORE CHARACTER IN BUFFER
                TIXR      T              LOOP UNLESS MAXIMUM LENGTH
                JLT       $ABLOOP        HAS BEEN REACHED
      $ABEXIT   STX       LENGTH         SAVE RECORD LENGTH
```

Fig 4.9(c) Use of Macro-Time conditional statement with EOF being NULL

```
     .          RDBUFF    F1. BUFF, ELENG, 04

30              CLEAR     X           CLEAR LOOP COUNTER
35              CLEAR     A
40              LDCH      =X'04'      SET EOR CHARACTER
42              RMO       A, S
45             +LDT       #4096       SET MAX LENGTH = 4096
50   $ACLOOP    TD        =X'F1'      TEST INPUT DEVICE
55              JEQ       $ACLOOP     LOOP UNTIL READY
60              RD        =X'F1'      READ CHARACTER INTI REG A
65              COMPR     A.S         TEST FOR END OF RECORD
70              JEQ       $ACEXIT     EXIT LOOP IF EOR
75              STCH      BUFF,X      STORE CHARACTER IN BUFFER
80              TIXR      T           LOOP UNLESS MAXIMUM LENGTH
85              JLT       $ACLOOP     HAS LOOP REACHED
90   $ACEXIT    STX       RLENG       SAVE RECORD LENGTH
```

Fig 4.9(d) Use of Time-variable with EOF NOT NULL and MAXLENGTH being NULL

The above programs show the expansion of Macro invocation statements with different values for the time variables. In figure 4.9(b) the &EOF value is NULL. When the macro invocation is done, IF statement is executed, if it is true EORCK is set to 1, otherwise normal execution of the other part of the program is continued.

The macro processor must maintain a symbol table that contains the value of all macro-time variables used. Entries in this table are modified when SET statements are processed. The table is used to look up the current value of the macro-time variable whenever it is required.

When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.

If the value of this expression TRUE,

- The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement.
- If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF.
- Once it reaches ENDIF, it resumes expanding the macro in the usual way.

If the value of the expression is FALSE,

- The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement.
- The macro processor then resumes normal macro expansion.

The *macro-time* IF-ELSE-ENDIF structure provides a mechanism for either generating(once) or skipping selected statements in the macro body. There is another construct WHILE statement which specifies that the following line until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. The testing of this condition, and the looping are done during the macro is under expansion. The example shown below shows the usage of Macro-Time Looping statement.

WHILE-ENDW structure

- When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- TRUE
  - The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.
  - When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.
- FALSE
  - The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.

```
RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH, &EOR
&EORCT   SET     %NITEMS (&EOR)   <-- Macro processor function
         CLEAR   X            CLEAR LOOP COUNTER
         CLEAR   A
        +LDT     #4096              SET MAX LENGTH = 4096
$LOOP    TD      =X'&INDEV'         TEST INPUT DEVICE
         JEQ     $LOOP              LOOP UNTIL READY
         RD      =X'&INDEV'         READ CHARACTER INTO REG A
&CTR     SET     1
         WHILE   (&CTR LE &EORCT)
         COMPR   =X'0000&EOR[&CTR]'   <-- List index
         JEQ     $EXIT
&CTR     SET     &CTR+1
         ENDW
         STCH    &BUFADR, X         STORE CHARACTER IN BUFFER
         TIXR    T                  LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP              HAS BEEN REACHED
$EXIT    STX     &RECLTH            SAVE RECORTD LENGTH
         MEND
```

```
.            RDBUFF    F2, BUFFER, LENGTH, (00, 03, 04)
                                           List

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
45             +LDT     #4096           SET MAX LENGTH = 4096
50    $AALOOP   TD      =X'F2'          TEST INPUT DEVICE
55              JEQ      $AALOOP        LOOP UNTIL READY
60              RD      =X'F2'          READ CHARACTER INTO REG A
65              COMP    =X'000000'
70              JEQ      $AAEXIT
65              COMP    =X'000003'
70              JEQ      $AAEXIT
65              COMP    =X'000004'
70              JEQ      $AAEXIT
75              STCH     BUFFER, X      STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $AALOOP        HAS BEEN REACHED
90    $AAEXIT   STX      LENGTH         SAVE RECORD LENGTH
```

Keyword Macro Parameters

- All the macro instruction definitions used positional parameters. Parameters and arguments are matched according to their positions in the macro prototype and the macro invocation statement.
- The programmer needs to be careful while specifying the arguments. If an argument is to be omitted the macro invocation statement must contain a null argument mentioned with two commas.
- Positional parameters are suitable for the macro invocation. But if the macro invocation has large number of parameters, and if only few of the values need to be used in a typical invocation, a different type of parameter specification is required

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

Keyword parameters

- Each argument value is written with a keyword that names the corresponding parameter.
- Arguments may appear in any order.

- Null arguments no longer need to be used.
- Ex: XXX P1=A1, P2=A2, P20=A20.
- It is easier to read and much less error-prone than the positional method.

```
RDBUFF   MACRO    &INDEV=F1, &BUFADR=, &RECLTH=, &EOR=04, &MAXLTH=4096
         IF       (&EOR NE ' ')
&EORCK   SET      1
         ENDIF
         CLEAR    X                 CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'          SET EOR CHARACTER
         RMO      A, S
         ENDIF
        +LDT      #MAXLTH           SET MAXIMUM RECORD LENGTH
$LOOP    TD       =X'&INDEV'        TEST INPUT DEVICE
         JEQ      $LOOP             LOOP UNTIL READY
         RD       =X'&INDEV'        READ CHARACTER INTI REG A
         IF       (&EORCK EQ 1)
         COMPR    A, S              TEST FOR END OF RECORD
         JEQ      $EXIT             EXIT LOOP IF EOR
         ENDIF
         STCH     $BUFADR, X        STORE CHARACTER IN BUFFER
         TIXR     T                 LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP             HAS BEEN REACHED
$EXIT    STX      &RECLTH           SAVE RECORD LENGTH
         MEND
```

Parameters with default value

```
         RDBUFF   BUFADR=BUFFER, RECLTH-LENGTH
```

```
         CLEAR    X                 CLEAR LOOP COUNTER
         CLEAR    A
         LDCH     =X'04'            SET EOR CHARACTER
         RMO      A, S
        +LDT      #4096             SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F1'            TEST INPUT DEVICE
         JEQ      $AALOOP           LOOP UNTIL READY
         RD       =X'F1'            READ CHARACTER INTI REG A
         COMPR    A, S              TEST FOR END OF RECORD
         JEQ      $AAEXIT           EXIT LOOP IF EOR
         STCH     BUFFER, X         STORE CHARACTER IN BUFFER
         TIXR     T                 LOOP UNLESS MAXIMUM LENGTH
         JLT      $AALOOP           HAS BEEN REACHED
$AAEXUT  STX      LENGTH            SAVE RECORD LENGTH
```

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


                CLEAR    X                CLEAR LOOP COUNTER
                CLEAR    A
               +LDT      #4096            SET MAXIMUM RECORD LENGTH
$ABLOOP         TD       =X'F3'           TEST INPUT DEVICE
                JEQ      $ABLOOP          LOOP UNTIL READY
                RD       =X'F3'           READ CHARACTER INTO REG A
                STCH     BUFFER, X        STORE CHARACTER IN BUFFER
                TIXR     T                LOOP UNLESS MAXIMUM LENGTH
                JLT      $ABLOOP          HAS BEEN REACHED
$ABEXIT         STX      LENGTH           SAVE RECORD LENGTH
```

Fig 4.10 Example showing the usage of Keyword Parameter

6.5. Macro Processor Design Options

Recursive Macro Expansion

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

```
RDBUFF  MACRO   &BUFADR, &RECLTH, &INDEV
        .
        .       MACRO TO READ RECORD INTO BUFFER
        .
                CLEAR   X               CLEAR LOOP COUNTER
                CLEAR   A
                CLEAR   S
               +LDT     #4096           SET MAXIMUN RECORD LENGTH
$LOOP           RDCHAR  &INDEV          READ CHARACTER INTO REG A
                COMPR   A, S            TEST FOR END OF RECORD
                JEQ     &EXIT           EXIT LOOP IF EOR
                STCH    &BUFADR, X      STORE CHARACTER IN BUFFER
                TIXR    T               LOOP UNLESS MAXIMUN LENGTH
                JLT     $LOOP           HAS BEEN REACHED
$EXIT           STX     &RECLTH         SAVE RECORD LENGTH
                MEND
```

```
RDCHAR  MACRO   &IN
.
.       MACROTO READ CHARACTER INTO REGISTER A
.
        TD      =X'&IN'                 TEST INPUT DEVICE
        JEQ     *-3                     LOOP UNTIL READY
        RD      =X'&IN'                 READ CHARACTER
        MEND
```

Problem of Recursive Expansion

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

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

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like

At the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE. Thus the macro processor would 'forget' that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

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**
  - o Programmers do not need to learn many macro languages.
  - o 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**
  - o 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

Macro Processing within Language Translators

- The macro processors we discussed are called "Preprocessors".
  - o Process macro definitions
  - o Expand macro invocations
  - o Produce an expanded version of the source program, which is then used as input to an assembler or compiler
- You may also combine the macro processing functions with the language translator:
  - o Line-by-line macro processor
  - o Integrated macro processor

Line-by-Line Macro Processor

- Used as a sort of input routine for the assembler or compiler
  - Read source program
  - Process macro definitions and expand macro invocations
  - Pass output lines to the assembler or compiler
- Benefits
  - Avoid making an extra pass over the source program.
  - Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)
  - Utility subroutines can be used by both macro processor and the language translator.
    - Scanning input lines
    - Searching tables
    - Data format conversion
  - It is easier to give diagnostic messages related to the source statements

i. Integrated Macro Processor

- An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.
  - Ex (blanks are not significant in FORTRAN)
    - DO 100 I = 1,20
      - a DO statement
    - DO 100 I = 1
      - An assignment statement
      - DO100I: variable (blanks are not significant in FORTRAN)
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.

UNIT – 7

LEX AND YACC – 1

7.1.INTRODUCTION:

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex.

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

```
                +-------+
Source ->  |  Lex   |  -> yylex
                +-------+

                +-------+
Input ->   | yylex | -> Output

                +-------+
```

7.2.SIMPLEST LEX PROGRAM AND LEX STRUCTURE:

The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%
Rules section
%%
C code section

- The **definition** section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Example:

```
/*** Definition section ***/

%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells lex to read only one input file */

%%
    /*** Rules section ***/

    /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }

.       {   /* Ignore all other characters. */   }

%%
/*** C Code section ***/

int main(void)
{
    /* Call the lexer, then quit. */
    yylex();
    return 0;
```

7.3. REGULAR EXPRESSIONS:

regular expression specifies a set of strings to be matched. It contains text characters and operator characters The letters of the alphabet and the digits are always text characters; thus the regular expression integer matches the string integer wherever it appears and the expression

```
a57D
```

looks for the string a57D.

Operators:

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters.
Thus

```
xyz"++"
```

matches the string xyz++ when it appears.

- Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

- An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions.
Another use of the quoting mechanism is to get a blank into an expression; blanks or tabs end a rule. Any blank character not contained within []must be quoted.

- Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.
- Character classes. Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges.

For example:

[a-z0-9<>_]indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order.

- Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. If it is desired to include the character - in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus , [^abc] matches all characters except a, b, or c, including all special or control characters

or [^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

- Optional expressions.: The operator ? indicates an optional element of an expression. Thus ab?c

matches either ac or abc.

- Repeated expressions: Repetitions of classes are indicated by the operators * and +.

Ex: a*
is any number of consecutive a characters, including zero, while a+ is one or more instances of a.

For example [a-z]+
is all strings of lower case letters.

A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches *any* character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.

|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

Regular expression	**Meaning**
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C. Every token has an associated expression. (Examples of tokens and expression are given in the following table.) Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

7.4. HOW TO RUN LEX PROGRAM:

If *lex.l* is the file containing the **lex** specification, the C source for the lexical analyzer is produced by running **lex** with the following command:

```
lex   lex.l
```

lex produces a C file called *lex.yy.c*.

Options

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument.

The **-t** option sends **lex**'s output to the standard output rather than to the file *lex.yy.c*.

The **-v** option prints out a small set of statistics describing the so-called finite automata that **lex** produces with the C program *lex.yy.c*.

In this section we can add C variable declarations. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declarations of Lex.

Declarations for the word-counting program

```
%{
int wordCount = 0;
%}
chars  [A-za-z\_\'\.\"]
numbers       ([0-9])+
delim      ["       "\n\t]
whitespace  {delim}+
words {chars}+
%%
```

The double percent sign implies the end of this section and the beginning of the second of the three sections in Lex programming.

Lex rules for matching patterns

Let's look at the Lex rules for describing the token that we want to match. (We'll use C to define what to do when a token is matched.) Continuing with our word-counting program, here are the rules for matching tokens.

Lex rules for the word-counting program

```
{words} { wordCount++; /*
increase the word count by one*/ }
{whitespace} { /* do
nothing*/ }
{numbers} { /* one may
want to add some processing here*/ }
%%
```

C code

The third and final section of programming in Lex covers C function declarations (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below.

C code section for the word-counting program

```
void main()
{
    yylex(); /* start the analysis*/
    printf(" No of words:
    %d\n", wordCount);
}
```

```
int yywrap()
{
    return 1;
}
```

7.5. LEXER

lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**. A lexer often exists as a single function which is called by a parser or another function

Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parenthesis as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

lexeme	**token type**
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

7.6 Examples:

1. Write a Lex source program to copy an input file while adding 3 to every positive number divisible by 7.

```
%%
        int  k;
    [0-9]+   {
        k = atoi(yytext);
        if (k%7 == 0)
            printf("%d", k+3);
        else
            printf("%d",k);
        }
```

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                int k;
-?[0-9]+            {
                k = atoi(yytext);
                printf("%d",
                  k%7 == 0 ? k+3 : k);
                }
-?[0-9.]+           ECHO;
[A-Za-z][A-Za-z0-9]+   ECHO;
```

Numerical strings containing a “." or preceded by a letter will be picked up by one of the last two rules, and not changed. The if-else has been replaced by a C conditional expression to save space; the form a?b:c means “if a then b else c".

2. Write a Lex program that histograms the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+   lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
```

```
{
int i;
printf("Length   No. words\n");
for(i=0; i<100; i++)
   if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

3.. Write a lex program to find the number of vowels and consonants.

```
%{
/*  to find vowels and consonents*/
   int vowels = 0;
   int consonents = 0;
%}
%%
[ \t\n]+
[aeiouAEIOU]   vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
consonents++;
.
%%
main()
{
   yylex();
   printf(" The number of vowels = %d\n", vowels);
   printf(" number of consonents = %d \n", consonents);
return(0);
}
```

The same program can be executed by giving alternative grammar. It is as follows: Here a file is opened which is given as a argument and reads to text and counts the number of vowels and consonants.

```
%{
   unsigned int vowelcount=0;
   unsigned int consocount=0;
%}
vowel [aeiouAEIOU]
consonant [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
eol \n

%%

{vowel} { vowelcount++;}
{consonant} { consocount++; }

%%
 main(int argc,char *argv[])
{
   if(argc > 1)
  {
    FILE              *fp;
    fp=fopen(argv[1],"r");
    if(!fp)
    {
```

```
      fprintf(stderr,"could not open %s\n",argv[1]);
      exit(1);
    }
   yyin=fp;
  }
  yylex();
 printf(" vowelcount=%u  consonantcount=%u\n ",vowelcount,consocount);
return(0);
}
```

4. Write a Lex program to count the number of words, characters, blanks and lines in a given text.

```
%{
      unsigned int charcount=0;
      int   wordcount=0;
      int   linecount=0;
      int blankcount =0;
%}
word[^ \t\n]+
eol \n
%%
[ ] blankcount++;
{word} { wordcount++; charcount+=yyleng;}
{eol} {charcount++; linecount++;}
. { ECHO; charcount++;}
%%
main(argc, argv)
int argc;
char **argv;
{
     if(argc > 1)
     {
      FILE *file;
      file = fopen(argv[1],"r");
      if(!file)
      {
        fprintf(stderr, "could not open %s\n", argv[1]);
        exit(1);
      }
     yyin = file;
        yylex();
        printf("\nThe number of characters = %u\n", charcount);
        printf("The number of wordcount  =  %u\n",  wordcount);
        printf("The number of linecount  =  %u\n",  linecount);
        printf("The number of blankcount = %u\n", blankcount);
        return(0);
      }
      else
        printf(" Enter the file name along with the program \n");
}
```

5. Write a lex program to find the number of positive integer,

negative integer, positive floating positive number and negative floating point number.

```
      int posnum = 0;
      int negnum = 0;
      int posflo = 0;
      int negflo = 0;
%}
%%
[\n\t ];
   ([0-9]+) {posnum++;}
   -?([0-9]+) {negnum++; }
   ([0-9]*\.[0-9]+)   { posflo++; }
   -?([0-9]*\.[0-9]+) { negflo++; }
. ECHO;
%%
 main()
 {
      yylex();
       printf("Number of positive numbers = %d\n", posnum);
      printf("number  of  negative  numbers  =  %d\n",  negnum);
      printf("number of floting positive number = %d\n", posflo);
      printf("number of floating negative number = %d\n", negflo);
 }
```

6. Write a lex program to find the given c program has right number of brackets. Count the number of comments. Check for while loop.

```
%{
   /* find main, comments, {, (, ), } */
  int comments=0;
  int opbr=0;
  int clbr=0;
  int opfl=0;
  int clfl=0;
  int j=0;
  int k=0;
%}
%%
"main()" j=1;
"/*"[ \t].*[ \t]"*/" comments++;
"while("[0-9a-zA-Z]*")"[ \t]*\n"{"[ \t]*.*"}"  k=1;
^[ \t]*"{"[ \t]*\n
^[ \t]*"}"                                   k=1;
"(" opbr++;
")" clbr++;
"{"  opfl++;
 "}" clfl++;
[^ \t\n]+
. ECHO;
%%
main(argc, argv)
int argc;
char *argv[];
{
```

```
    if (argc > 1)




    {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file)
        {
           printf("error opeing a file \n");
           exit(1);
        }
      yyin = file;
     }
     yylex();
     if(opbr != clbr)
         printf("open brackets is not equal to close brackets\n");
     if(opfl != clfl)
         printf("open flower brackets is not equal to close flower
                brackets\n");
         printf(" the number of comments = %d\n",comments);
     if (!j)
         printf("there is no main function \n");
     if (k)
         printf("there is loop\n");
     else printf("there is no valid for loop\n");
     return(0);
}
```

6. Write a lex program to replace scanf with READ and printf with WRITE statement also find the number of scanf and printf.

```
%{
int pc=0,sc=0;
%}
%%
printf fprintf(yyout,"WRITE");pc++;
scanf fprintf(yyout,"READ");sc++;
. ECHO;
%%
main(int argc,char* argv[])
{
  if(argc!=3)
  {
   printf("\nUsage: %s <src> <dest>\n",argv[0]);
   return;
  }
  yyin=fopen(argv[1],"r");
  yyout=fopen(argv[2],"w");
  yylex();
  printf("\nno. of printfs:%d\nno. of scanfs:%d\n",pc,sc);
}
```

7. Write a lex program to find whether the given expression is valid.

```
%{
   #include <stdio.h>
   int valid=0,ctr=0,oc = 0;
%}
NUM [0-9]+



OP  [+*/-]
%%
{NUM}({OP}{NUM})+ {
                   valid = 1;
                   for(ctr = 0;yytext[ctr];ctr++)
                   {
                     switch(yytext[ctr])
                       {
                          case  '+':
                          case  '-':
                          case '*':
                          case '/': oc++;
                       }
                    }
                   }
{NUM}\n {printf("\nOnly a number.");}
\n  { if(valid) printf("valid \n operatorcount = %d",oc);
      else  printf("Invalid");
      valid = oc = 0;ctr=0;
    }
%%
main()
{
  yylex();
}

/*    Another solution for the same problem     */

%{
int oprc=0,digc=0,top=-1,flag=0;
char stack[20];
%}
digit [0-9]+
opr [+*/-]
%%
[ \n\t]+
['(']  {stack[++top]='(';}
[')']  {flag=1;
        if(stack[top]=='('&&(top!=-1))
            top--;
          else
            {
            printf("\n Invalid expression\n");
            exit(0);
            }
         }
{digit}  {digc++;}
{opr}/['('] { oprc++; printf("%s",yytext);}
{opr}/{digit} {oprc++; printf("%s",yytext);}
```

```
. {printf("Invalid "); exit(0);}
%%
main()
{
yylex();
if((digc==oprc+1||digc==oprc) && top==-1)
{
printf("VALID");
```

```
printf("\n oprc=%d\n digc=%d\n",oprc,digc);
}
else
printf("INVALID");
}
```

8.Write a lex program to find the given sentence is simple or compound.

```
%{
int flag=0;
%}
%%
(" "[aA][nN][dD]" ")|(" "[oO][rR]" ")|(" "[bB][uU][tT]" ") flag=1;
.  ;
%%
main()
{yylex();
if (flag==1)
      printf("COMPOUND SENTENCE \n");
else
      printf("SIMPLE SENTENCE \n");
}
```

9. Write a lex program to find the number of valid identifiers.

```
%{
int count=0;
%}
%%
(" int ")|(" float ")|(" double ")|(" char ")

{
int ch; ch = input();
for(;;)
   {
       if (ch==',') {count++;}
       else
        if(ch==';') {break;}
           ch = input();
   }
count++;
}
%%
main(int argc,char *argv[])
```

```
{
yyin=fopen(argv[1],"r");
yylex();
printf("the no of identifiers used is %d\n",count);
}
```

UNIT - 8
LEX AND YACC – 2

8.1. Introduction

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. Yacc is written in portable C.

Yacc provides a general tool for imposing structure on the input to a computer program. User prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

8.2. Grammars:

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ',' year

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

8.3. Basic Specifications:

Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

declarations
%%
rules
%%
programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%
rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules.

A grammar rule has the form:

A : BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

8.4. SYMBOLS AND ACTIONS:

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline
'\r' return
'\'' single quote ``'''
'\\' backslash ``\''
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' ``xxx'' in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A   :   B C D ;
A   :   E F ;
A   :   G ;
```

can be given to Yacc as

```
A   :   B C D
    |   E F
    |   G
    ;
```

- It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.
- If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:
- **empty : ;**

- Names representing tokens must be declared; this is most simply done by writing
- **%token name1, name2 . . .**

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

- Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

- It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the % start keyword:

- **%start symbol**

- The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

- It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end-of-file'' or ``end-of-record''.

Actions:

- With each grammar rule, the user may associate actions to be Yacc: Yet Another Compiler-Compiler performed each time the rule is recognized in the input process.

- These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

- An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A :  '('  B  ')'
        {  hello( 1, "abc" );  }
```

and

```
XXX :   YYY  ZZZ
        { printf("a message\n");
          flag = 25;   }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr   :      '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr   :      '('  expr  ')'       {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B    ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

%{ int variable = 0; %}

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.
In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

8.5. Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the

token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
      extern int yylval;
      int c;
      . . .
      c = getchar();
      . . .
      switch( c ) {
            . . .
      case  '0':
      case '1':
       . . .
      case '9':
            yylval = c-'0';
            return( DIGIT );
            . . .
```

```
}
...
```

- The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

- This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser;

- For example, the use of token names 'if' or 'while' will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively.

- The token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc.

- The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

8.6. How the Parser Works :

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex. however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

. reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34. Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

8.7. Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr '-' expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

(expr - expr) - expr

or as

expr - (expr - expr)

(The first is called **left association**, the second **right association**).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

- expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr - expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

expr - expr - expr

It could then apply the rule to the rightmost three symbols, reducing them to expr and leaving

expr - expr

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

expr - expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a **shift / reduce conflict.**

It may also happen that the parser has a choice of two legal reductions; this is called a **reduce / reduce conflict**. Note that there are never any ``Shift/shift'' conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two **disambiguating** rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ``if-then-else'' construction:

```
stat   :    IF '(' cond ')' stat
       |    IF '(' cond ')' stat ELSE stat
       ;
```

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

EXAMPLE:

IF (C1) IF (C2) S1 ELSE S2

can be structured according to these rules in two ways:

```
IF ( C1 ) {
     IF ( C2 ) S1
 }   ELSE
S2
```

or

```
IF ( C1 ) {
     IF ( C2 ) S1
     ELSE S2
     }
```

- The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding ``un-ELSE'd" IF. In this example, consider the situation where the parser has seen

IF (C1) IF (C2) S1

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

IF (C1) stat

and then read the remaining input,

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

- On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

IF (C1) IF (C2) S1 ELSE S2

can be reduced by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule.

- Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

- This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

IF (C1) IF (C2) S1

- In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

stat : IF '(' cond ')' stat

- Once again, notice that the numbers following ``shift'' commands refer to other states, while the numbers following ``reduce'' commands refer to grammar rule numbers. In the y.output file, the rule numbers are printed after those rules which can be reduced.

8.8. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associatively. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars.

- The basic notion is to write grammar rules of the form

  expr : expr OP expr

  and

  expr : UNARY expr

  for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators.

- This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

- The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens.

- All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

  %left '+' '-'

  %left '*' '/'
- describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

- The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators

  %right '='

  %left '+' '-'
- **%left '*' '/'**
- **%%**
- **expr : expr '=' expr**
  - **| expr '+' expr**
  - **| expr '-' expr**
  - **| expr '*' expr**
  - **| expr '/' expr**
  - **| NAME**

might be used to structure the input

a = b = c*d - e - f*g

as follows

a = (b = (((c*d)-e) - (f*g)))

- When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences.
  - An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal.
  - It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
 %left '*' '/'
 %%
expr   :    expr '+' expr
       |    expr '-' expr
       |    expr '*' expr
       |    expr '/' expr
       |    '-' expr    %prec '*'
       |    NAME
       ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedence and associatively are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

3. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially ``cookbook'' fashion, until some experience has been gained. The y.output file is very useful in deciding whether the parser is actually doing what was intended.

8.9. Recursive rules:

The algorithm used by the Yacc parser encourages so called ``left recursive'' grammar rules: rules of the form

name : name rest_of_rule ;

These rules frequently arise when writing specifications of sequences and lists:

```
list  :   item
      |   list ',' item
```

```
          ;
and
     seq    :     item
          |     seq  item
          ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
     seq    :     item
          |     item  seq
          ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
     seq    :     /* empty */
          |     seq  item
          ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read

8.10. RUNNING BOTH LEXER AND PARSER:

The yacc program gets the tokens from the lex program. Hence a lex program has be written to pass the tokens to the yacc. That means we have to follow different procedure to get the executable file.

- **i.** The lex program <lexfile.l> is fist compiled using lex compiler to get **lex.yy.c.**
- **ii.** The yacc program <yaccfile.y> is compiled using yacc compiler to get **y.tab.c.**
- iii. Using c compiler b+oth the lex and yacc intermediate files are compiled with the lex library function. **cc y.tab.c lex.yy.c –ll.**
- iv. If necessary out file name can be included during compiling with –o option.

8.11. Examples

1. Write a Yacc program to test validity of a simple expression with +, - , /, and *.

```
/* Lex program that passes tokens */
%{
        #include     "y.tab.h"
        extern int yyparse();
%}
%%
[0-9]+ { return NUM;}
[a-zA-Z_][a-zA-Z_0-9]* { return IDENTIFIER;}
[+-] {return ADDORSUB;}
[*/] {return PROORDIV;}
[)(] {return yytext[0];}
[\n] {return '\n';}
%%
int main()
{
  yyparse();
}
/* Yacc program to check for valid expression */
%{
#include<stdlib.h>
extern int yyerror(char * s);
extern int yylex();
%}
%token NUM
%token ADDORSUB
%token PROORDIV
%token IDENTIFIER
%%
input :
    | input line
        ;
line    : '\n'


        | exp '\n' { printf("valid"); }
        | error '\n' { yyerrok; }
        ;
exp     : exp ADDORSUB term
        | term
        ;
term    : term PROORDIV factor
```

```
            |factor
            ;
factor : NUM
      |IDENTIFIER
      |'(' exp ')'
            ;
%%
int yyerror(char *s)
{
  printf("%s","INVALID\n");
}

/* yacc program that gets token from the c porogram */

%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUMBER LETTER
%left '+' '-'
%left '*' '/'
%%
line:line expr '\n' {printf("\nVALID\n");}
    |line '\n'
    |
    |error '\n' { yyerror ("\n INVALID"); yyerrok;}
    ;
expr:expr '+' expr
    |expr '-' expr
    |expr '*'expr
    |expr '/' expr
    |NUMBER
    |LETTER







   ;
%%
main()
{
yyparse();
}
yylex()
```

```
{
char c; while((c=getchar())=='
'); if(isdigit(c)) return
NUMBER; if(isalpha(c))
return LETTER; return c;
}
yyerror(char *s)
{
printf("%s",s);
}
```

2. Write a Yacc program to recognize validity of a nested 'IF' control statement and display levels of nesting in the nested if.

```
/* Lex program to pass tokens */
%{
        #include "y.tab.h"
%}
digit [0-9]
num {digit} + ("." {digit}+)?
binopr [+-/*%^=> <&|"= =”| “!=” | “>=” | “<=”
unopr [~!]
char [a-zA-Z_]
id {char}({digit} | {char})*
space [ \t]
%%
  {space} ;
  {num} return num;
  { binopr } return binopr;
  { unopr } return unopr;

  { id} return id
 "if" return if
 . return yytext[0];
%%
NUMBER {DIGIT}+
/* Yacc program to check for the valid expression */
%{
#include<stdio.h>
```

```
int cnt;
%}
%token binopr
%token unop
%token num
%token id
%token if
%%
foo: if_stat { printf("valid: count = %d\n", cnt); cnt = 0;
                    exit(0);
               }
          | error { printf("Invalid \n"); }
if_stat: token_if '(' cond ')' comp_stat {cnt++;}
cond: expr
          ;
expr:   sim_exp
          | '(' expr ')'
          | expr binop factor
          | unop factor
          ;
factor: sim_exp
          | '(' expr ')'
          ;
sim_exp:        num
          | id
          ;
sim_stat:       expr ';'
          | if
          ;
stat_list:      sim_stat
          | stat_list sim_stat
          ;

comp_stat:      sim_stat
          | '{' stat_list '}'
          ;
%%
main()
{
          yyparse();
}
yyerror(char *s)
{
          printf("%s\n", s);
          exit(0);
}
```

3. Write a Yacc program to recognize a valid arithmetic expression that uses +, - , / , *.

```
%{
        #include<stdio.h>
        #include <type.h>
%}

% token num
% left '+' '-'
% left '*' '/'
%%
st      :  st expn '\n'   {printf ("valid \n"); }
        |
        | st '\n'
        | error '\n' { yyerror ("Invalid \n"); }
        ;
%%
void main()
        {
                yyparse (); return 0 ;
        }
        yylex()
        {
                char c;
                while (c = getch () ) == ' ')

                        if (is digit (c))
                        return     num;
                        return c;
        }
        yyerror (char *s)
        {
                printf("%s", s);
        }
```

4. Write a yacc program to recognize an valid variable which starts with letter followed by a digit. The letter should be in lowercase only.

```
/*      Lex program to send tokens to the yacc program      */

%{
        #include "y.tab.h"
%}
%%
[0-9]     return  digit;
```

```
[a-z]      return  letter;
[\n]  return yytext[0];
.  return 0;
%%

/*        Yacc program to validate the given variable              */

%{
          #include<type.h>
%}
% token  digit letter ;
%%
ident    :  expn '\n' { printf ("valid\n"); exit (0); }
         ;
expn     : letter
         | expn letter
         | expn digit
         | error  { yyerror ("invalid \n");  exit (0); }
         ;
%%

main()
{
          yyparse();
}
yyerror (char *s)
{
          printf("%s", s);
}

/*        Yacc program which has c program to pass tokens                 */

%{
#include <stdio.h>
#include <ctype.h>
%}
%token LETTER DIGIT
%%
st:st LETTER DIGIT  '\n' {printf("\nVALID");}
  | st '\n'
  |
  | error '\n' {yyerror("\nINVALID");yyerrok;}
 ;
%%
 main()
{
yyparse();
```

```
}

yylex()
{
char  c;  while((c=getchar())=='
');     if(islower(c))          return
LETTER;  if(isdigit(c))   return
DIGIT; return c;
}
yyerror(char *s)
  {
  printf("%s",s);
  }
```

5.Write a yacc program to evaluate an expression (simple calculator program).

```
/*          Lex program to send tokens to the Yacc program     */
%{
                    #include"  y.tab.h"
                    expern int yylval;
%}
%%
[0-9]    digit
char[_a-zA-Z]
id         {char} ({ char } | {digit })*

%%
{digit}+ {yylval = atoi (yytext);
                    return num;
          }
{id}     return name
[ \t]      ;
\n         return 0;
.          return yytext [0];
%%

/*         Yacc Program to work as a calculator           */
%{
          #include<stdio.h>
          #include <string.h>
          #include <stdlib.h>
%}
% token num name
% left '+' '-'
% left '*' '/'
```

```
% left unaryminus
%%

st      :  name '=' expn
        | expn   { printf ("%d\n" $1);  }

expn    :  num   { $$ = $1 ; }
        | expn '+' num { $$ = $1 + $3; }
        | expn '-'  num  { $$ = $1 - $3; }
        | expn '*' num { $$ = $1 * $3; }
        | expn '/'  num  { if (num == 0)
                { printf ("div by zero \n");
                  exit (0);
                }
                else
                 { $$ = $1 / $3; }
        | '(' expn ')' { $$ = $2; }
        ;
%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}
```

Write a yacc program to recognize the grammar { $a^n b$ for n >= 0}.

```
/*      Lex program to pass tokens to yacc program */
%{
        #include "y.tab.h"
%}
[a] { return   a ;  printf("returning A to yacc \n"); }
[b]  return  b
[\n]  return yytex[0];
.  return error;
%%

/*      Yacc program to check the given expression              */

%{
```

```
        #include<stdio.h>
%}
% token a b error

%%
input   : line
        | error
        ;
line    : expn '\n' { printf(" valid new line char \n"); }
        ;
expn    : aa expn bb
        | aa
        ;
aa      : aa a
        | a
        ;
bb      : bb b
        | b
        ;
error   : error { yyerror ( " " ) ; }

%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}

/* Yacc to evaluate the expression and has c program for tokens */

%{
/* 6b.y   {A^NB  N >=0}     */

#include <stdio.h>
%}
%token A B
%%
st:st reca endb '\n'      {printf("String belongs to grammar\n");}
  | st endb  '\n'              {printf("String belongs to grammar\n");}
  | st '\n'
  | error '\n'        {yyerror ("\nDoes not belong to grammar\n");yyerrok;}

 ;
reca: reca enda | enda;
```

```
enda:A;
endb:B;
%%
main()
{
yyparse();
}
yylex()
{
char                    c;
while((c=getchar())==' ');
if(c=='a')
  return A;
if(c=='b')
  return B;
return c;
}
yyerror(char *s)
 {
fprintf(stdout,"%s",s);
}
```

7. Write a program to recognize the grammar { $a^n b^n$ | n >= 0 }

```
/*      Lex program to send tokens to yacc program          */

%{
        #include "y.tab.h"
%}
[a] {return   A ;  printf("returning A to yacc \n"); }
[b]  return B
[\n]  return yytex[0];
.  return error;
%%

/*      yacc program that evaluates the expression   */

%{
        #include<stdio.h>
%}
% token a b error

%%

input   : line
        | error
        ;
```

```
line    : expn '\n' { printf(" valid new line char \n"); }
        ;
expn    : aa expn bb
        |
        ;
error   : error { yyerror ( " " ) ; }

%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}

/*      Yacc program which has its own c program to send tokens  */
%{
/* 7b.y   {A^NB^N  N >=0}    */

#include <stdio.h>
%}
%token A B
%%
st:st reca endb '\n'      {printf("String belongs to grammar\n");}
  | st  '\n'              {printf("N value is 0,belongs to grammar\n");}
 |
  |error '\n'
                          {yyerror ("\nDoes not belong to grammar\n");yyerrok;}


reca: enda reca endb | enda;
enda:A;
endb:B;
%%
main()
{
yyparse();
}
yylex()
{
char                      c;
while((c=getchar())==' ');
if(c=='a')
  return A;
```

```
if(c=='b')
  return B;
return c;
}
yyerror(char *s)
 {
fprintf(stdout,"%s",s);
}
```

8. Write a Yacc program t identify a valid IF statement or IF-THEN-ELSE statement.

```
/*      Lex program to send tokens to yacc program          */

%{
#include "y.tab.h"
%}
CHAR [a-zA-Z0-9]
%x CONDSTART
%%
<*>[ ]  ;
<*>[ \t\n]+ ;
<*><<EOF>>  return 0;
if return(IF);
else return(ELSE);

then return(THEN);
\(  {BEGIN(CONDSTART);return('(');}
<CONDSTART>{CHAR}+ return COND;
<CONDSTART>\) {BEGIN(INITIAL);return(')');}
{CHAR}+ return(STAT) ;
%%

/* Yacc program to check for If and IF Then Else statement          */


%{
 #include<stdio.h>
%}
%token IF COND THEN STAT ELSE
%%
```

```
Stat:IF '(' COND ')' THEN STAT {printf("\n VALId Statement");}
    | IF '(' COND ')' THEN STAT ELSE STAT {printf("\n VALID Statement");}
    |
    ;
%%
main()
{
 printf("\n enter statement ");
 yyparse();
}
yyerror (char *s)
{
 printf("%s",s);
}

/*      Yacc program that has c program to send tokens      */

%{
        #include <stdio.h>
        #include <ctype.h>
%}
%token if simple
% noassoc reduce
% noassoc else
%%




start   : start st '\n'
        |
        ;
st      : simple
        | if_st
        ;
if_st   : if st %prec reduce   { printf ("simple\n"); }
        | if st else st         {printf ("if_else \n"); }
        ;
%%
int yylex()
{
  int c;
        c = getchar();
        switch ( c )
```

```
                {
                        case ‘i’ : return if;
                        case ‘s’ : return simple;
                        case  ‘e’  :  return  else;
                        default : return c;
    }
        }
        main ()
        {
          yy parse();
        }
        yyerror (char *s)
        {
                printf("%s", s);
        }
```