1.1 Introduction

- ***System software***: Collection of variety of programs that support the operation of a computer which makes it possible for the user to focus on an application or a problem to be solved without the need to know the details of how the machine works internally.
- Systems software is divided into ***operating systems*** and ***utility software.***
- Programs in high level languages can be written and edited using a ***text editor***; they are translated in to machine language using a ***compiler.***
- The resulting machine language program is loaded into memory and prepared for execution by the ***loader/linker.***
- ***Debuggers*** are used to find errors in the program.
- In assembly language programs, ***macro instructions*** are used in order to read/write data and perform high level functions.
- An ***assembler*** which includes a ***macro processor*** is used to translate these programs into machine language.
- These translated programs are prepared for execution by the ***loader/linker*** and could be tested using the ***debugger***.
- All these processes are controlled by interacting with the ***operating system*** of the computer which takes care of the machine level details (storage devices used, networks connected to, performing input and output).

1.2 System Software and Machine Architecture

- The characteristic which differentiates system software from application software is ***machine dependency.***
- ***Application program*** uses computer as a tool to solve a problem; the focus is on the application and not on the computing system.
- ***System program*** is intended to support the operation and the use of computer rather than any particular application; it is related to the machine architecture.
- E.g. ***Assemblers***: Translate mnemonic instructions into machine code. ***Compilers***: Generate machine language code, taking into account the type of registers and machine instructions. ***Operating Systems***: Manage the resources associated with a computer system.
- Some aspects of system software that do not directly depend upon the type of computing system being supported are:
  i. The general ***design*** and ***logic*** of an assembler is basically the same on most computers.
  ii. ***Code optimization*** techniques used by compilers are independent of the target machines.
  iii. The process of *linking* together independently assembled subprograms does not usually depend on the computer being used.
- It can be difficult to distinguish between those features of the software that are truly fundamental and those that depend solely on a particular machine.
- ***Simplified Instructional Computer (SIC):*** It is a ***hypothetical*** computer that has been carefully designed to include hardware feature most often found on real machines, while avoiding irrelevant or unusual complexities.
- The central concepts of system software can be separated from the implementation details associated with a particular machine.

1.3 The Simplified Instructional Computer (SIC)

- The SIC comes in two versions:
  - ***i. SIC Standard model***
  - ***ii. SIC/XE (extra equipment)***
- The two versions are ***upward compatible*** i.e., an object program written for standard SIC machine will also execute on a SIC/XE machine.

1.3.1 SIC Machine Architecture

i. Memory

-Memory consists of 8-bit bytes; 3 bytes form a word (24 bits).

-All address on SIC are byte addresses.

-Words are addressed by the location of their lowest numbered byte.

-There are a total of 32,768 (2^{15}) bytes in the computer memory.

ii. Registers

-There are five registers for special use and are 24 bits in length.

Mnemonic	***Number***	***Special Use***
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including condition code (CC)

iii. Data Formats

- Integers are stored as 24-bit binary numbers; 2's complement representation is used for negative values.
- E.g.: Integer 2 is stored as 0000 0000 0000 0000 0000 0010. Integer -5 is stored as 1111 1111 1111 1111 1111 1011 (a. write the binary representation; b. flip all the bit c. add 1)
- Characters are stored using their 8 bit ASCII codes.
- E.g.: Character 'A' (ASCII 65) is stored as 01000001.
- There is no floating point hardware on the standard SIC.

iv. Instruction Formats

(04/08/15)

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

8	1	5
Opcode	x	address

v. Addressing Modes

- The two addressing modes are indicated by setting the value of *x* bit in the instruction.
- The target address is calculated based on the address given in the instruction.
- Parenthesis () are used to indicate the contents of a register or a memory location.

Mode	***Indication***	***Target address calculation***

Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (X)

- Ex.: LDA TEN Direct addressing mode (TA = address)
- Ex.: STCH BUFFER, X Indexed addressing mode (TA= address+(X))

vi. Instruction Set

- SIC provides a basic instruction set that includes the following types of instructions:
  1. ***Load* and *store*** instructions (LDA, LDX, STA, STX).
  2. ***Integer arithmetic*** operations (ADD, SUB, MUL, DIV) which involve register A and a word in memory with the result being stored in the register.
  3. ***Compare*** instruction (COMP) that compares the value in register A with a word in memory and sets a condition code CC to indicate the result (<, =, >).
  4. ***Conditional jump*** instructions (JLT, JEQ, JGT) can test the value of CC and jump accordingly.
  5. ***Subroutine linkage*** instructions to jump to the subroutine (JSUB), placing the return address in register L; and return (RSUB) by jumping to the address contained in register L.

vii. Input and Output

- Input and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of the register.
- Each device is assigned a unique 8 bit code.
- There are three I/O instructions:
  1. ***Test Device (TD)*** instruction tests whether the addressed device is ready to send or receive a byte of data. A setting of < means the device is ready to send or receive, and = means the device is not ready.
  2. A program needs to wait until the device is ready and then perform ***Read data (RD)***.
  3. A program needs to wait until the device is ready and then perform ***Write Data (WD).***

1.3.2 SIC/XE Machine Architecture

i. Memory

- The memory structure for SIC/XE is same as that of SIC but the maximum available memory is increased to 1 megabyte (2^{20} bytes).
- This leads to change in instruction formats and addressing modes.

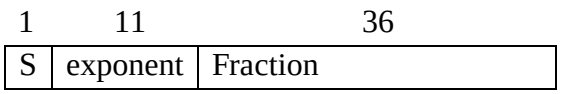
ii. Registers

-In addition to the SIC registers, the following registers have been provided by SIC/XE.

Mnemonic	***Number***	***Special use***
B	3	Base register; used for addressing
S	4	General working register – no special use
T	5	General working register – no special use
F	6	Floating – point accumulator (48 bits)

iii. Data formats

-SIC/XE provides the same data formats as the standard version; in addition there is a 48 – bit floating point data type.



-The fraction is a value between 0 and 1.

-The exponent is interpreted as an unsigned binary number between 0 and 2047.

-If the exponent has value *e* and the fraction has value *f* , the absolute value of the number represented is :

$$f * 2^{(e - 1024)}$$

-The sign of the floating point number is indicated by the value of *s* (0 = positive, 1 = negative).

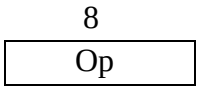
-A value of zero is represented by setting all bits to 0.

iv. Instruction formats

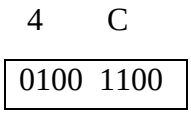
- There are two options: relative addressing and extension of the address field from 15 bits to 20 bits.
- Formats 1 and 2 are used for instructions that do not reference memory.

1. Format 1 (1 byte)

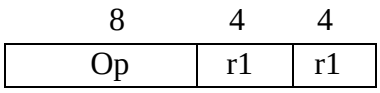
- It contains just the 8 bit opcode of the instruction



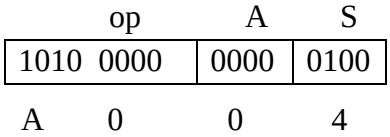
- Ex. RSUB instruction



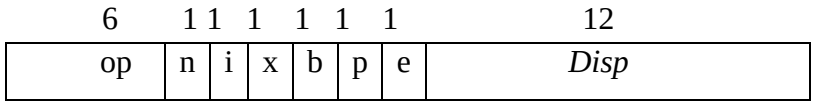
2. Format 2 (2 bytes)



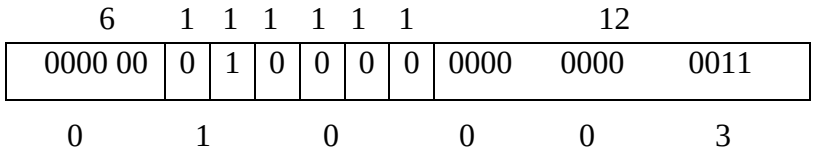
- Ex COMPR A,S (Compare the contents of registers A and S)



3. Format 3 (3 bytes)



- Ex.: LDA #3



4. Format 4 (4 bytes)

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

- Ex.: +JSUB RDREC (jump to address, 1036)

0100 10	1	1	0	0	0	1	0000 0001 0000 0011 0110

4 B 1 0 1 0 3 6

v. Addressing Modes

-Two 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* (-2049 <= *disp* <= 2047)

(B) → Contents of base register

(PC) → Contents of program counter

- For ***base relative*** addressing, the displacement field *disp* in format 3 is interpreted as a 12 bit unsigned integer.
- Ex.: 1056 STX LENGTH

6	1	1	1	1	1	1	12
0001 00	1	1	0	1	0	0	0000 0000 0000

Opcode n i x b p e *disp*

1 3 4 0 0 0 Object code

LENGTH = 0033

(B) = 0033

Disp = 0

TA = (B) + *disp* = 0033+0 = 0033

The content at address 0033 is loaded to the index register X.

- For ***program counter relative*** addressing, the *disp* field is interpreted as a 12 bit signed integer, with negative values represented in 2's complement.
- Ex.: 0000 STL RETADR

6	1	1	1	1	1	1	12
0001 01	1	1	0	0	1	0	0000 0010 1101

Opcode n i x b p e disp

1 7 2 0 2 D Object code

PC = 0003 (next instruction address (0000+0003=0003))

disp = 002D

TA = (PC) + *disp* = 0003 + 002D = 0030

RETADR = 0030

- The content of RETADR 0030 is stored into the linkage register L.
- If bits *b* and *p* are both set to 0, the *disp* field from format 3 instructions is taken to be the target address.

- For a format 4 instruction, bits *b* and *p* are set to 0, and the target address is taken from the address field of the instruction; this is called ***direct addressing***.
- Ex.: LDA LENGTH

6	1	1	1	1	1	1	12
0001 00	1	1	0	0	0	0	0000 0011 0011

opcode	n	i	x	b	p	e	disp
1	3			0		0	3 3

- Accumulator contains the LENGTH 0033.
- Any of these addressing modes can be combined with ***indexed addressing*** – if bit *x* is set to 1, the term (X) is added in the target address calculation.
- Ex.: STCH BUFFER, X

6	1	1	1	1	1	1	12
0101 01	1	1	1	1	0	0	0000 0000 0011

opcode	n	i	x	b	p	e	disp
5	7			C		0	0 3

[X] = 0033
disp = 3
TA =BUFFER = 0036

- Accumulator A contains the content of BUFFER 0036.
- Bits *i* and *n* are used to specify the target address; If bit *i* = 1 and *n* = 0, the target address itself is used as the operand value; no memory reference is performed.
- This is called ***immediate addressing.***
- Ex.: LDA #9

6	1	1	1	1	1	1	12
0000 00	0	1	0	0	0	0	0000 0000 1001

opcode	n	i	x	b	p	e	disp	
0	1			0		0	0 9	object code

- Accumulator contains 9.
- If bit *i* = 0 and *n* = 1, the target address is fetched; the value contained in this word is taken as the address of the operand value.
- This is ***indirect addressing***.
- Ex.: 002A J @RETADR

6	1	1	1	1	1	1	12
0011 11	1	0	0	0	1	0	0000 0000 0011

opcode	n	i	x	b	p	e	disp	
0	1			0		0	0 3	object code

PC = 002D
disp = 003
RETADR = 002D + 003 = 0030

- If bits *i* and *n* are both set to 0 or 1, the target address is taken as the location of the operand; bits *b, p, e* are considered to be part of the address field of the instruction.
- This is referred as ***simple addressing.***
- *Indexing* cannot be used with ***immediate*** or ***indirect*** addressing modes.

vi. Instruction Set

- SIC/XE provides all of the instructions that are available on the standard SIC.
- In addition, there are instructions to ***load and store*** the new registers (LDB, STB), and to perform ***floating point arithmetic*** operations (ADDF, SUBF, MULF, DIVF).
- Instructions take their operands from registers that include ***RMO (register move)*** instruction, ***register-to-register arithmetic*** operations (ADDR, SUBR, MULR, and DIVR).
- A ***special supervisor call (SVC)*** instruction generates an interrupt that can be used for communication with the OS.

vii. Input and Output

- The I/O instructions on SIC are available on SIC/XE.
- In addition, there are ***I/O channels*** that can be used to perform input and output while the CPU is executing other instructions.
- This allows overlapping of computer instructions and I/O resulting in an efficient system operation.
- The instructions *SIO*, *TIO* and *HIO* are used to ***start test*** and ***halt*** the operation of I/O channels.

1.4 Assembler Functions

- The assembler performs functions such as ***translating*** mnemonic operation codes to their machine language equivalents and ***assigns machine addresses*** to symbolic labels used by a programmer.
- In an assembly language program – line numbers are for reference and not a part of the program. ***Indexed*** addressing is indicated by adding a modifier **",X"** following the operand. Lines beginning with **"."** contain ***comments*** only.
- The following are the assembler directives:

1. **START** Specify ***name and starting address*** of the program.
2. **END** Indicate the ***end of the source program*** and specify the ***first executable instruction*** in the program.
3. **BYTE** ***Generate character or hexadecimal constant***, occupying as many bytes as needed to represent the constant.
4. **WORD** ***Generate one-word integer constant***.
5. **RESB** ***Reserve*** the indicated number bytes for a data area.
6. **RESW** ***Reserve*** the indicated number of words for a data area.

- The program contains a main routine that ***reads records from an input device*** (identified with device code F1) and ***copies them to an output device*** (code 05).
- The main routine calls subroutine **RDREC** to read a record into a buffer and subroutine **WRREC** to write the record from the buffer to the output device.

- Each subroutine must transfer the record one character at a time using the I/O instructions **RD** and **WD.**
- The buffer is required since the I/O rate for the input and output devices are not the same.
- If the record is longer than the length of the buffer, only the first 4096 bytes are copied.
- The ***end of each record*** is marked with a null character (hex 00); when the end of file is detected, the program writes **EOF** on the output device and terminates by executing the RSUB instruction (returns the control to the OS).

1.4.1 A Simple SIC Assembler

- The column ***Loc*** gives the ***machine address*** for each part of the assembled program.
- Assume the program starts at address 1000.
  i. ***Convert*** the ***mnemonic*** operation codes to their ***machine language*** equivalents.
  ii. ***Convert symbolic operands*** to their equivalent ***machine addresses.***
  iii. ***Build the machine instructions*** in the proper format.
  iv. ***Convert the data constants*** specified in the source program into their ***internal machine representation.***
  v. Write the ***object program*** and ***assembly listing.***

Consider the statement in

```
10   1000   FIRST    STL     RETADR    141033
```

- This instruction contains a ***forward reference*** - a reference to a label (RETADR) that is defined later in the program.
- The program therefore cannot be translated line by line because the address assigned to each line is unknown and the line cannot be processed.
- To overcome this problem, the assembler makes ***two passes*** over the source program; the ***first pass scans the source program for label definitions and assigns addresses and the second pass performs the actual program translation***.
- The assembler processes the ***assembler directives*** (instructions to the assembler) which are not translated into machine instructions; instead they provide instructions to the assembler itself.
- Finally the assembler ***writes the object code*** onto the output device, which is later ***loaded into memory*** for execution.
- The object program format contains ***three types of records***:

1. Header record:

- It contains the ***program name***, ***starting address*** of the object program, and the ***length***.

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)

2. Text record:

- It contains the ***translated instructions*** and the ***data*** of the program, together with an indication of the ***addresses*** where these are to be loaded.

Col. 1:	T
Col. 2-7:	Starting address for object code in this record (hex)
Col. 8-9:	Length of the object code in this record in bytes (hex)

Col. 10-69: Object code represented in hex

3. End record:

- It marks the ***end of the object program*** and specifies the ***address*** in the program where ***execution*** is to begin.

Col. 1: E

Col.2-7: Address of first executable instruction in object program (hex)

- A general description of the two passes of the assembler:

Pass 1 (define symbols):

a. Assign addresses to all statements in the program.
b. Save the values (addresses) assigned to all labels for use in Pass 2.
c. Perform some processing of assembler directives.

Pass 2 (assemble instructions and generate object program):

a. Assemble instructions (translating operation codes and looking up addresses).
b. Generate data values defined by BYTE, WORD, etc.
c. Perform processing of assembler directives not done during Pass 1.
d. Write the object program and the assembly listing.

```
HCOPY  001000001070
T001000E14103348203900103628103030101548206 13C100300102A0C103900102D
T00101E150C10364820610810334C0000454F46000003000000
T002039 1E041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T002057 1C1010364C0000F1001000041030E02079302064509039DC20792C1036
T00207307382064 4C000005
E001000
```

Figure 2.3 Object program corresponding to Fig. 2.2.

1.4.2 Assembler Algorithm and Data structures

- The assembler uses two major internal data structures: ***the Operation Code Table*** **(OPTAB)** and the ***Symbol Table*** **(SYMTAB).**
- OPTAB is used to ***look up the mnemonic codes and translate*** them to their machine language equivalents.
- SYMTAB is used to ***store values*** (addresses) assigned to labels.
- **LOCCTR** (location counter) is a variable used to ***assign addresses***; it is initialized to the beginning address specified in the START statement.
- After each source statement is processed, the length of the assembled instruction to be generated is added to LOCCTR.
- When any label is reached in the program, the current value of LOCCTR gives the address associated with that label.
- The OPTAB must contain at least the mnemonic operation codes, its machine equivalent and information about the instruction format and length (in complex assemblers).
- During ***Pass 1,*** the OPTAB is used to look up and validate operation codes in the source program; in ***Pass 2*** it is used to translate the operation codes to machine language.
- In a simple SIC assembler, both the processes can be done together in any one of the passes.

- For an SIC/XE machine that has instructions of different length, the OPTAB is searched in ***Pass 1*** to find the instruction length for incrementing LOCCTR.
- In ***Pass 2***, OPTAB is looked up to find out which instruction format to use in assembling the instruction.
- OPTAB is organized as a ***hash table*** with ***mnemonic operation code*** as the ***key*** which helps in providing fast retrieval with a minimum of searching.
- OPTAB is a ***static table*** – the entries are not normally added or deleted from it.
- The SYMTAB includes the ***name*** and ***value*** (addresses) for each label in the source program, together with ***flags*** to indicate ***error*** conditions and information about the data area or instruction labeled.
- During ***Pass 1*** of the assembler, labels are entered into SYMTAB as they are encountered in the source program along with their assigned addresses from LOCCTR.
- During ***Pass 2***, symbols used as operands are looked up in SYMTAB to obtain addresses to be inserted in the assembled instructions.
- SYMTAB is organized as a ***hash table*** for efficiency of insertion and retrieval.
- There is certain information that needs to be communicated between the two passes.
- For this reason, Pass 1 writes an ***intermediate file*** that contains each source statement together with its assigned address, error indicators etc. which is given as input Pass2.
- This working copy of the source program can be used to retain the results of certain operations that may be performed during Pass 1, so they need not be repeated in Pass 2

1.5 Machine–Dependent Assembler Features

- In the assembler language, ***prefix @*** is used to indicate ***indirect addressing***, ***prefix #*** is used to indicate ***immediate addressing.***
- Instructions that refer to memory are assembled using ***base relative*** or ***program counter relative*** mode.
- The assembler directive **BASE** is used in conjunction with ***base relative addressing***.
- If the required displacements are too large to into 3-byte format 3, the 4-byte ***extended format*** is used and is indicated by using the ***prefix +***.
- It uses ***register-to-register instructions*** wherever required which perform faster than register-memory instructions as they do not require a memory reference; thereby improving the execution speed of the program.
- Immediate addressing provides the operand as part of the instruction and does not require a memory reference.
- The use of indirect addressing avoids the need of another instruction such as a return.
- The larger memory of SIC/XE makes more space to load and run several programs at a time. This sharing of the machine between programs is called ***multiprogramming***.
- Multiprogramming results in productive use of hardware.

1.5.1 Instruction Formats and Addressing Modes

- **START** statement specifies a beginning program address of 0.
- The assembler must ***convert the mnemonic operation code*** to machine language and change each ***register mnemonic*** to its numerical equivalent (done in Pass 2).

- The conversion of register mnemonics to numbers can be done with a separate table; it is often convenient to use the ***symbol table*** **(SYMTAB)** which is preloaded with the register names and their values.
- Most of the ***register-to-memory*** instructions are assembled using either ***program relative or base relative addressing.***
- The assembler must in either case, calculate a displacement to be assembled as part of the object instruction which is added to the contents of the ***program counter (PC) or base register (B)*** which results in the target address.
- If neither program counter relative nor base relative addressing can be used because of large displacements, then the ***4 byte extended instruction*** format which has a 20 bit address field must be used.
- The programmer must specify the extended format by using the ***prefix +.***
- If the extended format is not specified, the assembler tries to translate the instruction using program counter relative addressing; if this is not possible the assembler then attempts to use base relative addressing.
- If neither of the relative addressing forms are not applicable, and the extended format is not specified, then the instruction cannot be properly assembled in which case an error message must be generated.
- During execution of instructions on SIC, the program counter is advanced after each instruction is fetched and before it is executed.
- Hence the content of the program counter will be known at execution time.
- The base register on the other hand is under the control of the programmer; hence the programmer must tell the assembler the contents of the base register.
- This is done using the assembler directive **BASE.**
- E.g. BASE LENGTH → informs the assembler that the base register contains the address of LENGTH.
- The assembler assumes that ***register*** contains this address until it encounters another BASE statement.
- If the base register is used for any other purpose later in the program, the programmer must use the assembler directive **NOBASE** in order to inform the assembler that the contents of base register can no longer be relied upon for addressing.
- BASE and NOBASE are assembler directives hence they do not produce any executable code.
- The programmer must provide instructions that load the proper value into the base register during execution which otherwise will produce incorrect target address calculation.
- The assembly of an instruction that specifies ***immediate addressing*** is simpler as it does not require any memory reference.
- The immediate operand is converted into its internal representation and inserted into the instruction.
- The assembly of instructions that specify ***indirect addressing mode set bit n to 1*** to indicate that the contents stored at this location represent the address of the operand, not the operand itself.

1.6 Program Relocation

- It is desirable to have more than one programs sharing the memory and other system resources and executing concurrently.
- If it was known in advance exactly which programs were to be executed concurrently, addresses would be assigned when the programs were assembled in way that they would fit together without overlap or wasted space.
- However it is not practical to plan execution this closely; hence it is desirable to be able to load a program into memory whenever there is room for it.

Ex. 55 101B LDA THREE 00102D

- In the object program, this statement is translated as 00102D, specifying that register A is to be loaded from memory address 102D.
- Suppose it is attempted to load the program and execute it at address 2000 instead of 1000; the address will not contain the value expected.
- There is a need to change the address portion of the instruction so the program can be loaded and executed at address 2000; at the same time there are portions of the program that need to remain the same regardless of where the program is loaded.
- Since the assembler does not know the actual location where the program to be loaded, it cannot make the necessary changes in the addresses used by the program.
- The assembler can identify for the loader those parts of the object program that need modification
- An object program that contains the information necessary to perform this kind of modification is called a ***relocatable program.***
- Program is loaded beginning at address 0000. The JSUB instruction is loaded at address 0006. The address of this instruction contains 01036 which is the address of the instruction labeled RDREC.
  - Suppose the program is loaded beginning at address 5000. The address of the instruction labeled RDREC is 6035.
  - The ***relocation problem*** can be solved in the following way:
    i. When the assembler generates the object code for the JSUB instruction; it will *insert the address* of RDREC *relative to the start of the program*.
    ii. The assembler will also *produce a command for the loader*, instructing it to *add* the beginning address of the program to the address field in the JSUB instruction at load time.
  - The command for the loader must be a part of the object program; which is accomplished making use of a ***modification record.***

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)

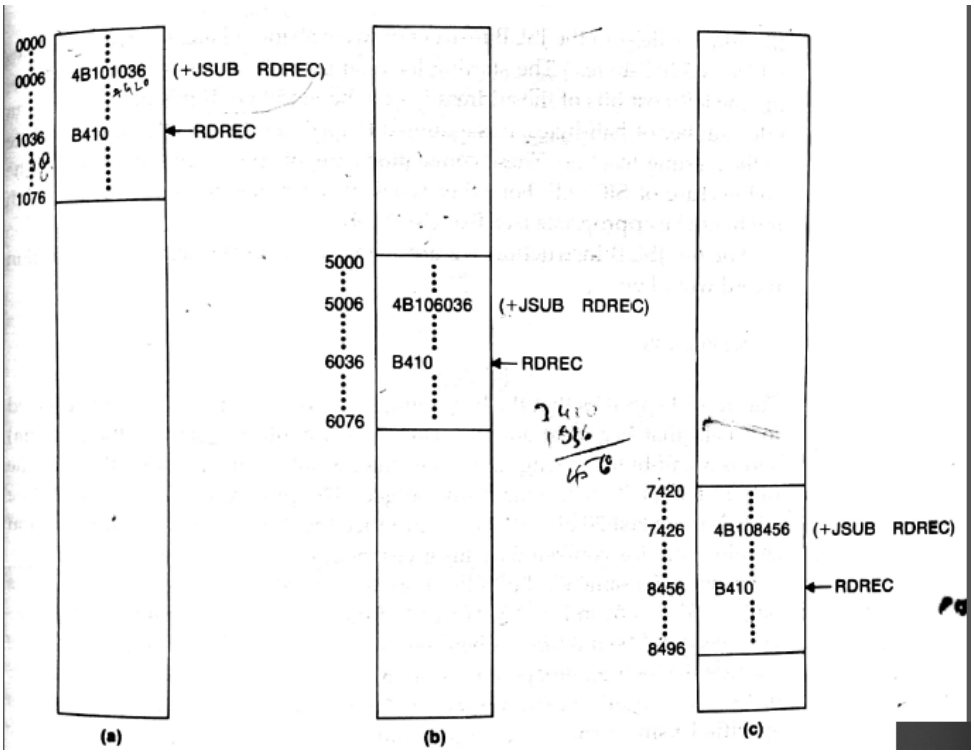
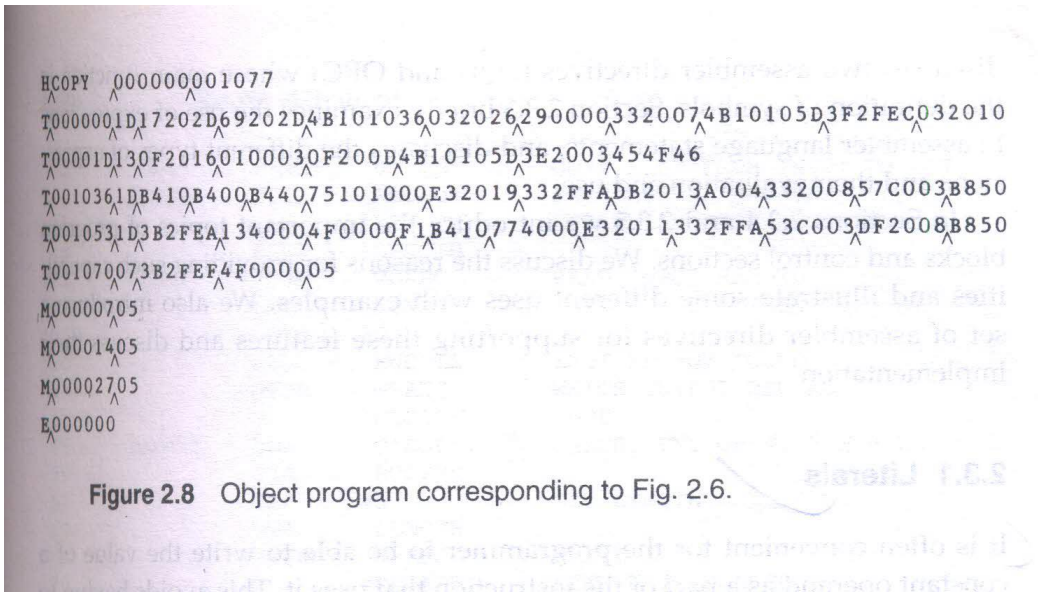


Figure 2.7 Program Relocation

- The length is stored in ***half bytes*** because the address field to be modified may not occupy an integral number of bytes.
- The starting location is the location of the byte containing the leftmost bits of the address field to be modified.
- Ex.: M00000705
- The above modification record specifies that the beginning address of the program is to be added to a field that begins at address 000007 (relative to the start of the program) and is 5 half bytes in length.
- Thus in the assembled instruction 4B101036 the first 12 bits (4B1) remain unchanged; and the program load address is added to the last 20 bits (01036) to produce the correct operand address.
- The same kind of relocation must be performed for the instructions on line 35 and 65 in fig. 2.6
- The rest of the instruction need not be modified when the program is loaded.



```
HCOPY  000000001077
T000000 1D 17202D 69202D 4B1010 360320 262900 003320 074B10 105D3F 2FEC03 2010
T00001D 13 0F2016 010003 0F200D 4B1010 5D3E20 03454F 46
T001036 1D B410 B400 B440 75101000 E32019 332FFA DB2013 A00433 200857 C003 B850
T001053 1D 3B2FEA 1340004 F0000F 1B4107 74000E 32011 332FFA 53C003 DF2008 B850
T001070 07 3B2FEF 4F000005
M00000705
M00001405
M00002705
E000000
```

Figure 2.8 Object program corresponding to Fig. 2.6.

1.7 MACHINE INDEPENDENT ASSEMBLER FEATURES

1.7.1 Literals

- The programmer may need to write the value of a constant operand as part of the instruction that uses it. This avoids having to define a constant elsewhere in the program and make up for a label for it.
- Such an operand is called a ***literal*** because the value is literally stated in the instruction.
- A literal is identified with a **prefix =**, which is followed by a specification of the literal value.
- Ex.: 45 001A ENDFIL LDA =C 'EOF' 032010
- Ex.: 215 1062 WLOOP TD =X '05' E32011
- ***Difference between literal and immediate value:***
  - With immediate addressing the operand value is ***assembled as part of the machine instruction.***
  - With a literal, the assembler generates the specified ***value as a constant at some other memory location.***
  - The address of the generated literal constant is used as the target address for the machine instruction.
- The effect of using a literal is exactly same as if the programmer had defined the constant explicitly and used the label assigned to the constant as the instruction operand. All the literal operands used in the program are gathered together into one or more ***literal pools*** which are placed at the end of the program.
- The assembly listing of a program using literals includes a listing of the literal pool which shows the assigned ***addresses*** and the generated ***data values***.
- This would sometimes place literal operands too far from the instructions referencing it to allow program counter relative addressing mode.
- This means a large amount of storage is reserved for the buffer which makes it desirable to place literals into a pool at some other location (other than the end) in the object program.

- This is done using the assembler directive **LTORG** which enables keeping the literal operands close to the instruction that uses it.
- When the assembler encounters LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG or the beginning of the program.
- This literal pool is placed in the object program at the location where LTORG directive was encountered.
- The literals placed in a pool will not be repeated in the pool at the end of the program.
- Assemblers recognize ***duplicate literals***– same literals used in more than one place in the program - and store only one copy of the specified data value.
- Only one data area with the literal value is generated hence all the instructions using that particular literal operand must refer to the same memory location.
- Ex.: The literal =X'05' is used on line 215 and 230 but is referenced from the same memory location
- The easiest way to recognize duplicate literals is by comparison of the character strings defining the literals. The assembler may also be designed to recognize equivalent literals.
- Ex. The literals = C 'EOF' and =X '454647' would specify identical operand values.
- Literals may also be used to refer to the current value of the location counter (denoted by symbol *) which helps in loading registers.
- Ex. BASE *
  LDB =*
  - When used as the first lines of the program would load the beginning address of the program into register B.
  - If the same statements are used in between the program, the literal values will change.
  - Hence these literals have identical strings but have different data values and both must appear separately in the LITTAB.
- The assembler handles literal operands making use of a data structure called the ***literal table*** **LITTAB**.
- For each literal used, this table contains the ***literal name***, the ***operand value*** and ***length*** and the ***address*** assigned to the operand when it is placed in a literal pool.
- LITTAB is organized as a ***hash table***, using the literal name or value as the key.
- In ***Pass 1***, the assembler ***searches the LITTAB*** for the specified literal name, if the literal is already present in the table no action is taken; if it is not present, the literal is ***added*** to LITTAB.
- When ***Pass 1*** encounters a ***LTORG*** statement or the end of the program, the assembler makes a ***scan of the LITTAB*** and each literal currently present in the table is assigned an address.
- The location counter is also updated to reflect the number of bytes occupied by each literal.
- During ***Pass 2***, the operand address for use in ***generating object code*** is obtained by searching LITTAB for each literal operand encountered.
- The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD statements.
- If the ***literal value*** represents an ***address*** in the program, the assembler must generate the appropriate ***modification record***.

1.7.2 Symbol Defining Statements

- Assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.
- The assembler directive **EQU** is used for this purpose and the general form is:

  symbol **EQU** ***value***
- The statement defines the given ***symbol***, enters it into **SYMTAB** and assigns to it the ***value*** specified, the *value* may be given as a ***constant*** or as an ***expression*** involving constants and previously defined symbols.
- ***Common uses of EQU***:
  i. To establish symbolic names that can be used to improve ***readability***.
  ii. To define mnemonic names for registers.
- Ex.(i): +LDT #4096 (loads the value 4096 into register T) can be written as

  MAXLEN EQU 4096
  +LDT #MAXLEN
- When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB with the value 4096.
- During assembly of the LDT instruction, the assembler searches SYMTAB for the symbol MAXLEN, using its value as the operand in the instruction.
- This improves the readability and makes it much easier to find and change the value of MAXLEN.
- Ex.(ii):

  A EQU 0
  X EQU 1
  L EQU 2
- This causes the symbols A, X, L to be entered into SYMTAB with their corresponding values 0, 1, 2.
- If the assembler expects register numbers instead of names (in an instruction like RMO), the programmer can write RMO A, X.
- The assembler will search SYMTAB, finds the values 0 and 1 for the symbol A and X and assemble the instruction.
- The assembler provides an assembler directive **ORG** that can be used to indirectly assign values to symbols.
- Format: **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*** to the specified ***value***.
- The ORG statement will affect the values of all labels defined until the next ORG.
- Ex.: Usage of ORG in label definition
- STAB (100 entries) SYMBOL VALUE FLAGS

- In this table, the SYMBOL field contains 6-byte user defined symbol, VALUE is a one word representation of the value assigned to the symbol, and FLAGS is a 2 byte field that specifies symbol type and other information.
- Space is reserved for this table with the statement STAB RESB 1100
- The entries in the table can be accessed using indexed addressing, one way of doing so is with EQU assembler directive:

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

- LDA VALUE, X to fetch the VALUE field from the table entry indicated by the contents of register X.
- The same can be accomplished using ***ORG***:

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

- The first ORG resets the *location counter* to the value of STAB; the label on the following RESB statement defines SYMBOL to have the current value in LOCCTR.
- LOCCTR is advanced so that label o the RESW statement assigns to VALUE the address and so on
- The last ORG statement sets the LOCCTR back to its previous value – the address of the next unassigned byte of memory
- ***Restrictions***:
- In the case of EQU, all symbols used on the right hand side of the statement must have been defined previously in the program
- In the case of ORG, all symbols used to specify the new location counter value must have been previously defined
- E.g.:

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

Would be allowed,

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

Will not be allowed

E.g.:

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

- Could not be processed as the value of the location counter of the ORG statement will not be known to the assembler

1.7.3 Expressions

- Expressions can be used in place of operands; where each expression must be evaluated by the assembler to produce a single operand address or value.
- Assemblers allow arithmetic expressions formed using normal rules and the operators +, -, *, /
- Division must produce an integer result.
- Individual terms in the expression maybe constants, user defined symbols or special terms (current value of location counter denoted *).
- The values of terms and expression are either ***relative*** or ***absolute.***
- A ***constant*** is an absolute term whereas ***labels on instructions and data areas***, references to the location counter are relative expressions.
- A symbol whose value is given by EQU may either relative or absolute.
- An expression can be classified as ***relative expression*** or ***absolute expression*** based on the type of value they produce.
- ***An expression that contains only absolute terms or pairs of relative terms in which the terms in each such pair have opposite signs is an absolute expression.***
- The terms may not be place adjacently in the expression and none of the relative terms may enter into multiplication or division operation.
- ***A relative expression is the one in which all terms except one can be paired; the remaining unpaired relative term must have a positive sign.***
- Expressions that do not meet the conditions given for either relative or absolute expressions should be flagged by the assembler as errors.
- ***A relative term or expression may be written (S + r), where S is the starting address of the program and r is the value of the term or expression relative to the starting address***.
- When the relative terms are paired with opposite signs, the dependency on the program starting address is cancelled out; the result is an absolute value.
- Ex.: MAXLEN EQU BUFEND-BUFFER
- Here both BUFEND and BUFFER are relative terms, each representing an address within the program.
- However, the expression represents an absolute value which is the length of the buffer area in bytes.
- The *Loc* value does not represent does not represent the address but the value assigned to the symbol MAXLEN (1000).
- Expressions which do not qualify as absolute or relative expressions are flagged as errors by the assembler.
- Ex.: 100-BUFFER, 3*BUFFER, BUFEND+BUFFER

SYMBOL	TYPE	VALUE
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

- To determine the type of expression, the types of all symbols defined in the program have to be tracked.
- For this purpose a ***flag*** is added to the symbol table to indicate the type of value (absolute or relative).

- With this information, the assembler can determine the type of each expression used as an operand and generate modification records in the object code for relative values.

1.7.4 Program Blocks

- The source program logically contains subroutines and data areas; but is handled by the assembler as one unit resulting in a single block of object code.
- Within this object code, the generated machine instructions and data appear in the same order as they are written in the source program.
- Assembler provide features that allow ***flexible handling of source and object code***:
  i. They allow generated machine instructions and data to appear in the object code in an order different from the source code.
  ii. They allow creation of several independent parts of the object program which maintain their identity and are handled separately by the loader.
- ***Program blocks*** refer to segments of code that are rearranged with a single object program unit.
- ***Control sections*** refer to segments that are translated into independent object program units.
- In the following program three blocks are used.
- The first unnamed program block contains the executable instructions of the program; the second block named CDATA contains all the data areas that are less in size and the third block named CBLKS consists of all data areas that consist of larger blocks of memory.
- The assembler directive ***USE*** indicates which portions of the source program belong to various blocks.
- At the beginning of the program no USE directive is included, hence the statements are assumed to belong to the default unnamed block
- The USE statement on *line 92*, signals the beginning of block named CDATA; source statements are associated with this block until *line 103*, which begins the block named CBLKS.
- The USE statement may also indicate a continuation of the previous block (*line 123* resumes the default block and *line 183* resumes black CDATA).
- Each program block may contain several separate segments of the source program; the assembler then logically rearranges these blocks by maintaining a separate location counter for each block in Pass 1.
- The ***location counter for a block is initialized to zero*** when the block is first begun and the current value of the location counter is saved when switching to another block.
- The saved value is restored when resuming a previous block.
- During Pass1 each label in the program is assigned an address that is relative to the start of the block that contains it.
- When labels are entered into the symbol table, the block name and number is stored along with the assigned relative address.
- At the end of Pass1, the latest value of the location counter for each block indicates the length of that block.
- The assembler then assigns to each block a starting address in the object program relative to the start of the program.

- During Pass 2, the assembler uses the address from the symbol table to assign addresses relative from the start of the program.
- Ex.: The header showing Loc/Block shows the relative address assigned to each source line and the block number involved (0=default; 1=CDATA; 2=CBLKS).
- The symbol MAXLEN is not assigned any block which indicates it is an absolute value and is not relative to the start of the program.
- At the end of Pass 1, the assembler constructs a table the starting address and the length for all blocks

BLOCK NAME	BLOCK NUMBER	ADDRESS	LENGTH
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

- Consider the instruction: 20 0006 0 LDA LENGTH 032060
- SYMTAB shows the value of the operand as relative location 0003 within program block 1.
- The starting address of block 1 (CDATA) is 0066 and thus the desired target address for the instruction is 0003 + 0066 = 0069.
- The instruction is assembled using program counter relative addressing mode and thus the required displacement would be 0069 – 0009 (Program Counter value) = 60.
- ***Advantages*** of using program blocks:
  i. The separation of program into program blocks has ***reduced the addressing problems***.
  ii. The large buffer area is moved to the end of the program which ***eliminates*** the need to use ***extended addressing mode*** on line 15, 35 and 65.
  iii. The ***base register is no longer needed***; statements LDB and BASE are deleted.
- The generated code in the object program need not be rearranged physically; the assembler writes the object code as it is generated during Pass 2 and inserts the correct load addresses in each Text record.
- These ***load addresses*** reflect the starting address of the block as well as the relative location of the code within the block
- Texts records are generated in the similar manner; except that the ***USE statement also begins a new Text record*** (even if there is space in the current Text record).
- The first two text records are generated from the source program lines 5 through 70; the next two text records are generated from the source program lines 125 through 180; the fifth text record contains a single byte from line 180 and the last text record is the continuation of the default program block (no Text record is required for lines 95 through 105 as there is no code generated).
- The Text records are not in sequence by address; but the loader simply loads the object code from each record at the indicated address.
- Once the loading is complete, the default block will occupy relative locations 0000 through 0065; the CDATA block will occupy relative locations 0066 through 0070 and the block CBLKS will occupy the locations 0071 through 1070.

1.7.5 Control Sections and Program Linking

- A ***control section*** is a part of the program that maintains its identity after assembly; each such control section can be ***assembled***, ***manipulated***, ***loaded*** and ***relocated*** independently of the others.
- Control sections form logically related parts of the program (subroutines or logical subdivisions of the program), and they have to be linked together.
- The instructions in one control section may need to refer to instructions or data located in another control section; such references are called ***external references***.
- The assembler generates information for each external reference that will allow the loader to perform the required ***linking***.
- There are three control sections in the program: one for main and one for each subroutine.
- The START statement identifies the beginning of the assembly and gives the name COPY to the first control section which continues until the **CSECT** statement on line 109.
- This assembler directive signals the start of the new control section named RDREC and the assembler directive **CSECT** on line 193 begins the control section WRREC.
- The assembler establishes a separate location counter for each of the control sections and they are handled separately by the assembler.
- Symbols that are defined in one control section may not be used directly by another and must be identified as ***external references*** for the loader to handle.
- There are two assembler directives for this purpose:
  i. **EXTDEF (external definition)**
  ii. **EXTREF (external reference)**
- The EXTDEF statement in a control section ***names*** symbols, called ***external symbols*** that are ***defined in this control section*** and may be ***used by other sections.***
- The EXTREF statement names symbols that are ***used in this control section but defined elsewhere***.
- Ex.: 15 0003 CLOOP +JSUB RDREC 4B100000
  - The operand RDREC is named in the EXTREF statement for the control section, and hence is an external reference.
  - The assembler cannot assemble the address for this instruction as it does not know where the control section containing the RDREC will be loaded.
  - The assembler inserts an address of zero and passes the information to the loader, which will cause the proper address to be inserted at load time.
  - Thus it requires an extended format in order to provide room for the actual address to be inserted (this is true for any instruction whose operand involves an external reference).
- Ex.: 160 0017 +STCH BUFFER, X 57900000
  - The above example makes an external reference to BUFFER and is assembled using the indexed addressing mode and the extended instruction format with an address of zero.
- Ex.: 190 0028 MAXLEN WORD BUFEND-BUFFER
  000000
  - The value of the data word to be generated is specified by an expression involving two external expressions: BUFEND, BUFFER.
  - The assembler stores this value as zero, as it does not know the exact address since BUFFER and BUFEND are not defined in the same control section.

- When the program is loaded the loader will add to this data area, the address of BUFFEND and subtract the address of BUFFER which results in the desired value.
- On line 107, the value assigned to MAXLEN will be known to the assembler since both BUFEND and BUFFER are defined in the same control section.

- The two new record types are:
  i. ***Define record***: Gives information about the external symbols that are defined in this control section (symbols named by EXTDEF).
  ii. ***Refer record***: Lists symbols that are used as external references by the control section (symbols named by EXTREF).
- ***Formats***:

Define record:

Col. 1 :	D
Col. 2-7:	Name of external symbol defined in this control section
Col. 8-13:	Relative addresses of symbol within this control section
Col. 14-73:	Repeat information in Col. 2-13 for the external symbols

Refer record:

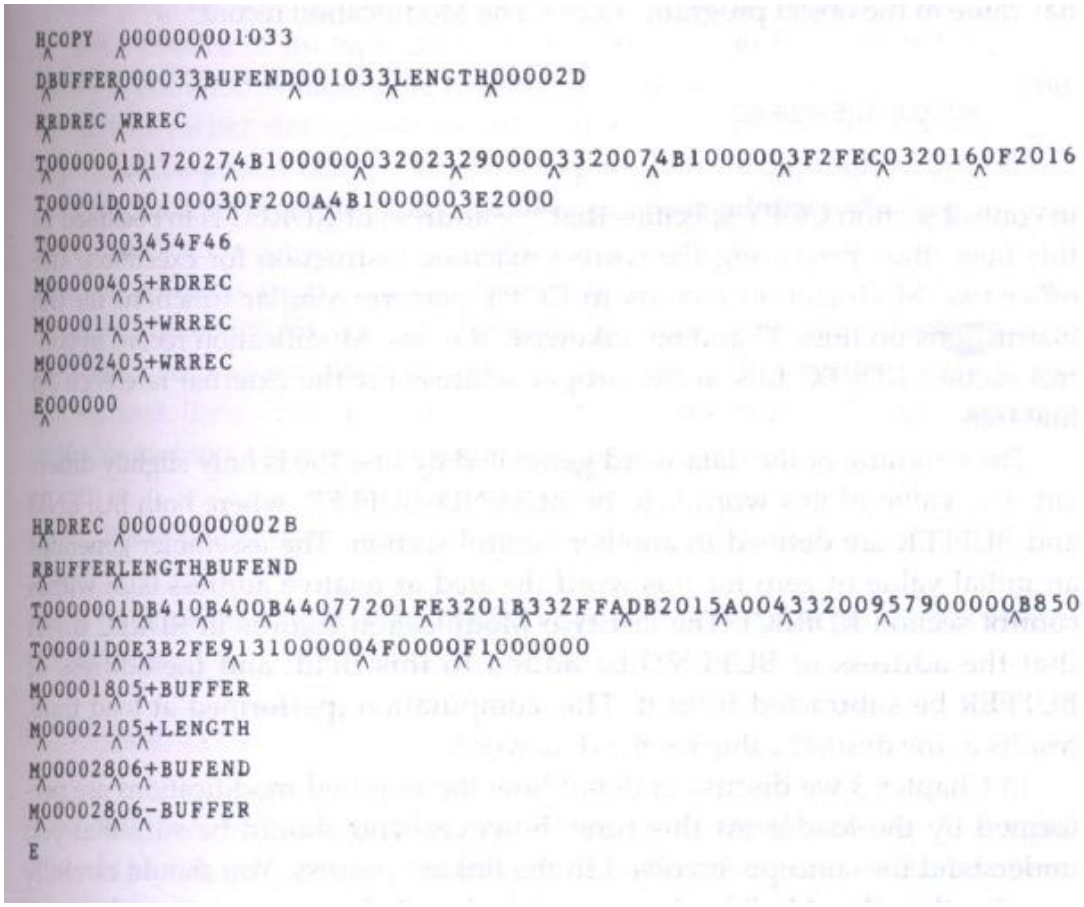
Col. 1 :	R
Col. 2-7:	Name of external symbol referred to in this control section
Col. 8-73:	Names of other external reference symbols

- The information related to program linking is added to the modification record

Modification record (revised):

Col. 1:	M
Col. 2-7:	Starting address of the field to be modified, relative to the beginning of the control section
Col. 8-9:	Length of the field to be modified, in half bytes
Col. 10:	Modification flag (+ or -)
Col. 11-16:	External symbol whose value is to added or to be subtracted from the indicated field

- The Define and Refer records for each control section include the symbols named in the EXTDEF and EXTREF statements.
- The Define record indicates the relative address of each external symbol within the control section.
- For EXTREF symbols, no address information is available and these are simply named in the Refer record.
- Ex.: M00000405+RDREC
  - The above Modification record in control section COPY specifies that the address of RDREC is to be added to this field, thus producing the correct machine instruction for execution.



```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D172027 4B10000032023290000332007 4B1000003F2FEC032016 0F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000


HRDREC 000000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A004332009579000000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E
```

1.8 Assembler Design Options

1.8.1 One – Pass Assembler

- The main problem in trying to assemble a program in one pass involves ***forward references***.
- Instruction operands are symbols that have not yet been defined in the source program, hence the assembler does not know what address is to be inserted in the translated instruction.
- It is easy to eliminate ***forward references to data items*** by defining them in the source program before they are referenced.
- This requires the programmer to place the storage reservation statements at the start of the program rather than at the end.
- ***Forward references to labels on instructions*** cannot be eliminated easily as they often require *forward jumps.*
- A special provision is made by the assembler to handle forward references.
- There are two types of one pass assembler.
  1. ***Load and go assembler***
     - It produces object code directly in memory for ***immediate execution***.
     - ***No object program is written out*** which simplifies forward references; hence no loader is required.
     - It is very useful in systems that are oriented towards ***program development and testing***.

- A load and go assembler avoids the overhead of writing the object program out and reading it back, which can be accomplished with either a one pass or a two pass assembler.
- One pass assembler further ***avoids the overhead*** of another pass over the source program.
- The assembler simply generates the object code instructions as it scans the source program.
- If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled.
- The symbol used as an operand is entered into the symbol operand and the entry is flagged to indicate that the symbol is undefined.
- The address of the operand field of the instruction referring to the undefined symbol is added to a list of forward references associated with the symbol table entry.
- When the definition of the symbol is encountered, the forward reference list for that symbol is scanned and the proper address is inserted into any instructions previously generated.
- At the end of the program, if and SYMTAB entries are still flagged, they indicate undefined symbols and are marked as error by the assembler.
- When the end of the program is encountered, the assembly is complete.
- If no errors have occurred, the assembler searches SYMTAB for the value of the symbol named in the END statement and jumps to this location to begin execution.
- The first forward reference - the operand RDREC is not defined, the instruction is assembled with no value assigned to the operand address.
- RDREC is entered into the SYMTAB as an undefined symbol indicated by *.
- The address of the operand field of the instruction (2013) is inserted in a list associated with RDREC.
- The same procedure is followed on line 30 and 35.
- When the symbol ENDFIL is defined on line 45, the assembler places its value in the SYMTAB entry which is then inserted into the instruction operand field.
- The definition of RDREC on line 125 results in the filling of the operand address at location 2013.
- This process is continued till the end of program.

2. Assemblers that produce object program for later execution

- These are used on systems where external working storage devices are slow, inconvenient or unavailable.
- Forward references are entered into lists.
- When the symbol definitions are encountered, the instructions that made forward references to that symbol may no longer be available in memory for modification.
- But they are written out as Text records in the object program.
- In this case, the assembler must write another Text record with the correct operand address.
- When the program is loaded, this address will be inserted into the instruction by the action of the loader.

- The second Text record contains the object code generated from lines 10 through line 40. The operand addresses for the instructions on lines 15, 30, 35 have been generated as 0000
- When the definition of ENDFIL on line 45 is encountered, the assembler generates the third Text record.
- This record specifies that the value 2024 is to be loaded at location 201C.
- When the program is loaded, the value 2024 will replace 0000 previously loaded.

1.8.2 Multi – Pass Assembler

- Consider the following sequence:

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

  - The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined.
  - ALPHA cannot be evaluated during the second Pass; any assembler that makes only two passes over a source program cannot resolve such a sequence of definitions.
- Forward references make it difficult for a person to read the program as well as for the assembler.
- The general ***solution*** is to have a ***multi pass assembler*** that can make as many passes as are needed to process the symbol definitions.
- The portions of the program that involve forward references in symbol definition are saved during Pass 1.
- Additional passes through these stored definitions are made as the assembly progresses which are followed by a normal Pass2.
- The symbol definitions that involve forward references are stored in the ***symbol table***, which also indicates which symbols dependent on the values of others to evaluate symbols.
- MAXLEN has not been defined hence no value for HALFSZ can be computed; the defining expression for HALFSZ is stored in the symbol table in place of its value.
- The entry **&1** indicates that one symbol in the defining expression is undefined, whose definition maybe stored in another location.
- The symbol MAXLEN is entered into SYMTAB with a ***flag*** ***** identifying it as undefined. Associated with this entry is a list of symbols whose values depend on MAXLEN.
- The same procedure is repeated for the rest of the undefined symbols.
- The two undefined symbols BUFEND and BUFFER and MAXLEN is dependent upon them.
- The definition of BUFFER enables evaluation of some of the labels; the address calculated is stored as the value of BUFFER.
- The assembler now examines the list of symbols that are dependent on BUFFER and evaluates them in the next passes.
- The symbol entry for the first symbol in the list (MAXLEN) shows that it depends on two undefined symbols; hence MAXLEN cannot be evaluated immediately.
- Instead &2 is changed to &1 to show that only one symbol definition has been evaluated.
- The symbol PREVBT can be evaluated as it depends only on BUFFER.

- The remainder of the processing follows the same pattern. When BUFFEND is defined by line 5, its value is entered into the symbol table. This enables evaluation of MAXLEN which in turn enables evaluation of HALFSZ.
- If any symbol remains undefined at the end of the program, the assembler flags them as errors.

TIPS FOR OBJECT CODE GENERATION:

- ✓ Memorize the 2 byte instructions as they are few in number and the format will not be provided as part of data (ADDR, CLEAR, COMPR, DIVR, MULR, RMO, SHIFTL,SHIFTR, SUBR, SVC, TIXR)
- ✓ Memorize the 1 byte instructions (FIX, FLOAT, HIO, NORM, SIO, TIO)
- ✓ Indirect addressing is indicated using @ (set n=1, i=0)
- ✓ Immediate addressing is indicated using #, do not calculate disp (set b,p=0), write the value of operand in the disp field. (set n=0, i=1)
- ✓ Indexing is indicated by the use of register X in the instruction. (set x =1)
- ✓ Always use Program Counter addressing by default, if the displacement calculated using PC is very large to fit in disp field, switch to Base Relative (remember to set b=1 for BR)
- ✓ Extended addressing is indicated by +(set b=0, p=0 and set e=1). Write the direct address of the operand in disp field. Don't calculate using PC or BR.
- ✓ Program counter, base relative and extended addressing are mutually exclusive, i.e. at a time only one among b,p,e bits can be set equal to 1.
- ✓ When indirect and immediate addressing are not used, set both bits n, i to 1 (not 0 even though it is specified as correct).
- ✓ Generation of object codes does not require to write the header text and end records (do it only when object program is asked).
- ✓ The address generation/ LOCCTR values are not written for START, END, BASE, NOBASE directives (However, you have to generate addresses for RESB, RESW, BYTE, WORD)
- ✓ Updating the LOCCTR depends on the length of the previous instruction, if the instruction is 1 byte add 1, if it is 2 byte add 2, if it is 3 byte add 3 to get the LOCCTR for the next instruction
- ✓ In case of RESB add the number of bytes after converting them to hex (ex. RESB 4096, convert 4096 to hex = 1000, add 1000 to LOCCTR)
- ✓ In case of RESW, convert the words to bytes and then add the hex equivalent. (ex. RESW 2000, convert to bytes -> 2000 x 3= 6000 bytes; convert 6000 to hex = 1770; add 1770 to LOCCTR)
- ✓ A new text record need to be created only when the size of text record (calculated as last address – first address + size of the last instruction) does not fit into the 2-column size field.
- ✓ Solve more problems to gain confidence.

1.9 Macro processors

- 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*)
- Two new assembler directives are used in macro definition:
  MACRO: identify the beginning of a macro definition
  MEND: identify the end of a macro definition
- Prototype for the macro:

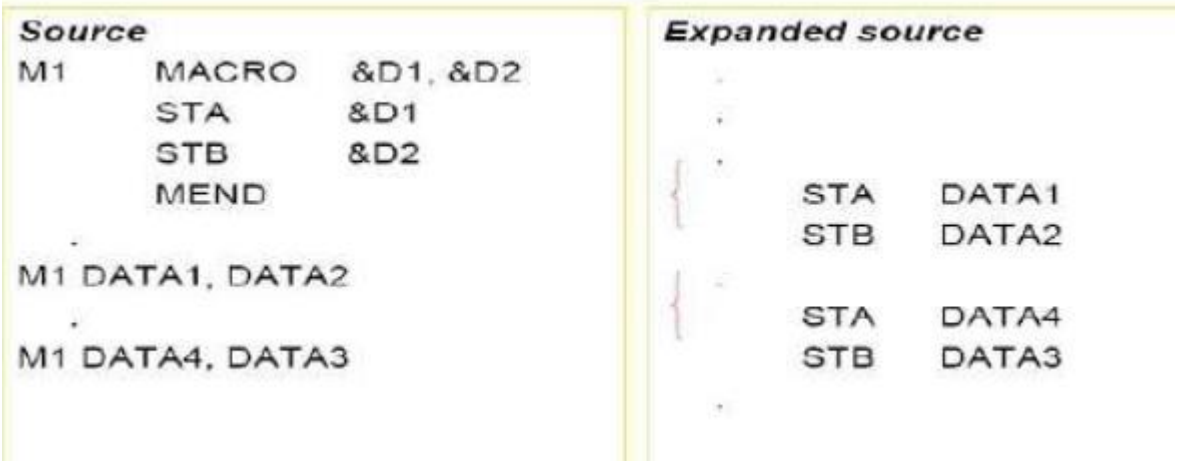
- Each parameter begins with &

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

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

1.9.1 Basic Macro Processor Functions:

- *Macro Definition and Expansion*



- *Macro Processor Algorithms and Data structures*
- 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 expanded with the executable statements.
- The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.
- The program with macros is supplied to the macro processor.
- Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype.

- During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.
- 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 statements 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:
  1. 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:
  1. Do not use labels in the body of macro.
  2. Explicitly use PC-relative addressing instead.

```
170  .                          MAIN PROGRAM
175  .
180         FIRST   STL     RETADR          SAVE RETURN ADDRESS
190         CLOOP   RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
195                 LDA     LENGTH          TEST FOR END OF FILE
200                 COMP    #0
205                 JEQ     ENDFIL          EXIT IF EOF FOUND
210                 WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
215                 J       CLOOP           LOOP
220         ENDFIL  WRBUFF  05,EOF,THREE    INSERT EOF MARKER
225                 J       @RETADR
230         EOF     BYTE    C'EOF'
235         THREE   WORD    3
240         RETADR  RESW    1
245         LENGTH  RESW    1               LENGTH OF RECORD
250         BUFFER  RESB    4096            4096-BYTE BUFFER AREA
255                 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
```

1.9.2 Macro Processor Algorithm and Data Structure

- Design can be done as two-pass or a one-pass macro. In case of two-pass assembler.
- 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.
- The body of one macro can contain definitions of the other macro
- Consider the example of a Macro defining another Macro.
- 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 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/XE machine.

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}
```

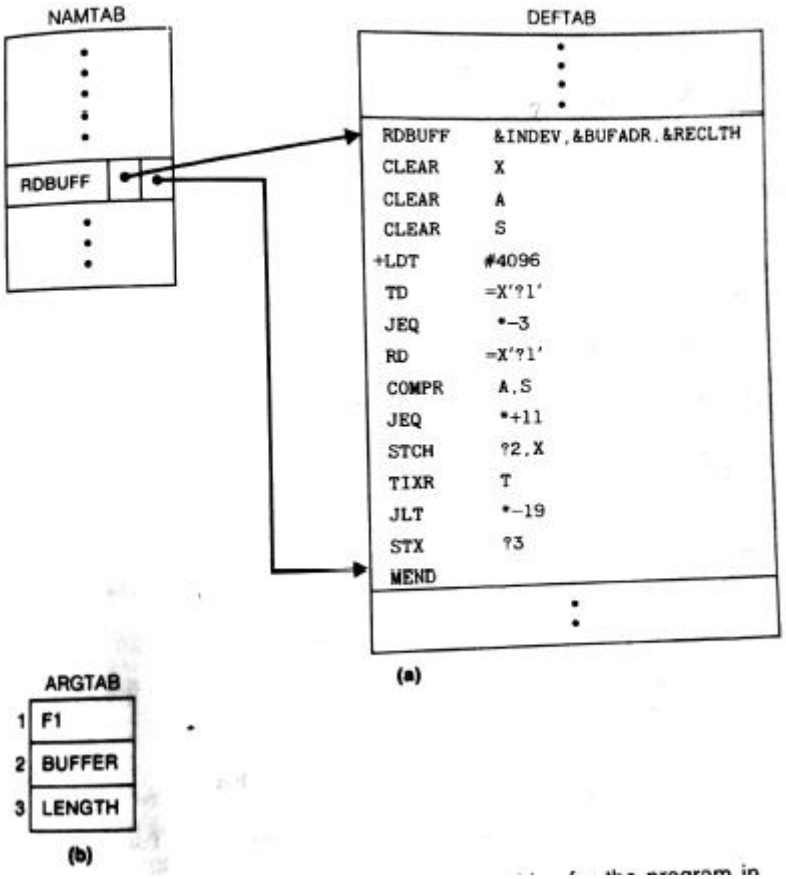
MACROX for SIC/XE Machine

```
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}
```

- 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.



- The above figure shows the portion of the contents of the table during the processing of the program.
- The definition of RDBUFF is stored in DEFTAB, with an entry in NAMTAB having the pointers to the beginning and the end of the definition.
- The arguments referred by the instructions are denoted by their positional notations. For example, TD =X‟?1‟
- The above instruction is to test the availability of the device whose number is given by the parameter &INDEV.
- In the instruction this is replaced by its positional value? 1. Figure 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 from DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.
- The algorithm of the Macro processor has the procedure DEFINE to make the entry of *macro name* in the NAMTAB, *Macro Prototype* in DEFTAB. EXPAND is called to set up the argument values in ARGTAB and expand a *Macro Invocation* statement.
- Procedure GETLINE is called to get the next line to be processed either from the DEFTAB or from the file itself.
- When a macro definition is encountered it is entered in the DEFTAB.
- The normal approach is to continue entering till MEND is encountered.
- If there is a program having a Macro defined within another Macro.
- While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition.

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