1. Why an Assembly Language is needed?

Programming in machine code, by supplying the computer with the numbers of the operations it must perform, can be quite a burden, because for every operation the corresponding number must be looked up or remembered. Looking up all numbers takes a lot of time, and mis-remembering a number may introduce computer bugs.

So Assembly Languages are evolved which contains mnemonic instructions corresponding to the Machine codes using which the program can be written easily.

Therefore a set of mnemonics was devised. Each number was represented by an alphabetic code. So instead of entering the number corresponding to addition to add two numbers one can enter "add".

Although mnemonics differ between different CPU designs some are common, for instance: "sub" (subtract), "div" (divide), "add" (add) and "mul" (multiply).

2. What is an Assembler?

An assembler is system software which is used to convert an assembly language program to its equivalent obect code. The input to an assembler is a source code written in assembly language (using mnemonics) and the output is the object code.

3. Explain the terms a)Label, b)Opcode, c)Operand, and d)Comment
(What is the format in which the assembly language program is written?).

- **Label field**.
  - The label is a symbolic name that represents the memory address of an executable statement or a variable.
- **Opcode/directive fields**.
  - The opcode (e.g. operation code) specifies the symbolic name for a machine instruction.
  - The directive specifies commands to the assembler about the way to assemble the program.
- **Operand field.**
  - The operand specifies the data that is needed by a statement.
- **Comment field.**
  - The comment provides clear explanation for a statement.

4) What are the basic functions of an assembler?

The basic assembler functions are

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

- Convert *mnemonic operation codes* to their *machine language equivalents*
  E.g. STL -> 14 (line 10)
- Convert *symbolic operands* to their equivalent *machine addresses*
  E.g. RETADR -> 1033 (line 10)
- Build the machine instructions in the proper format
- Convert the *data constants* to *internal machine representations*
  E.g. EOF -> 454F46 (line 80)
- Write the *object program* and the *assembly listing*

5. What are assembler Directives?

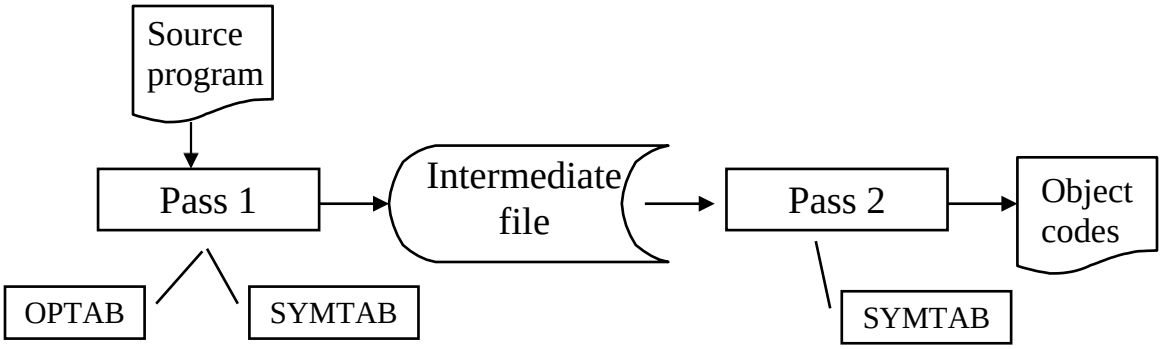
Assembler directives are Pseudo-instructions that are not translated into machine instructions and they provide instructions to the assembler itself.

- **The SIC assembler directives.**
  - START
    - Specification of the name and start address of the program.
  - END
    - Indication of the end of the program and optionally the address of the first executable instruction.
  - BYTE
    - Generate character or hexadecimal constant occupying as many as needed to represent the constant.
  - WORD
    - Generate one word constant.
  - RESB
    - Reserve the indicated number of bytes for a data area.
  - RESW
    - Reserve the indicated number of words for a data area.

6. What are the functions of two pass assembler?

Functions of Two Pass Assembler

- **Pass 1 - Define symbols (assign addresses)**
  - Assign addresses to all statements in the program
  - Save the values assigned to all labels for use in Pass 2
  - Process some assembler directives
- **Pass 2 - Assemble instructions and generate object program**
  Assemble instructions
  - Generate data values defined by BYTE, WORD, etc.
  - Process the assembler directives not done in Pass 1
  - Write the object program and the assembly listing



7. Write an algorithm for pass1 of an assembler.

```
Pass 1:

begin
   read first input line
   if OPCODE = 'START' then
      begin
         save #[OPERAND] as starting address
         initialize LOCCTR to starting address
         write line to intermediate file
         read next input 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
                        insert (LABEL,LOCCTR) into SYMTAB
                  end {if symbol}
               search OPTAB for OPCODE
               if found then
                  add 3 (instruction 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 {if BYTE}
               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}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

- 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 3 * value of operand to the LOCCTR.
- If it is BYTE it adds length of constant in bytes to the LOCCTR,
- if RESB it adds value of operand to the LOCCTR.
- 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.

8. Write an algorithm for Pass 2 of an assembler.

Pass 2:

```
begin
   read first input line (from intermediate file)
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end (if START)
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
      begin
         if this is not a 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
                           if found then
                              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
                  end (if opcode found)
               else if OPCODE = 'BYTE' or 'WORD' then
                  convert constant to object code
               if object code will not fit into the current Text record then
                  begin
                     write Text record to object program
                     initialize new Text record
                  end
               add object code to Text record
            end (if not comment)
         write listing line
         read next input line
      end (while not END)
   write last Text record to object program
   write End record to object program
   write last listing line
end (Pass 2)
```

Figure 2.4(b) Algorithm for Pass 2 of assembler.

- 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.

9. What is the disadvantage of single pass assembler
Or
what is forward reference?

Line	Loc	Source statement			Object code
		Label	**opcode**	**operand**	
5		**COPY**	**START**	**0**	
10	**0000**	**FIRST**	**STL**	**RETADR**	**17202D**
12	**0003**		**LDB**	**#LENGTH**	**69202D**
15	**0006**	**CLOOP**	**+JSUB**	**RDREC**	**4B101036**
20	**000A**		**LDA**	**LENGTH**	**032026**
25	**000D**		**COMP**	**#0**	**290000**
125	**1036**	**RDREC**	**CLEAR**	**X**	**B410**

Fig 8.1 Program With Object Code

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.

Forward reference *is a* reference to a label that is defined later in the program.
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 in 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.

10. What is the format of the Object Program generated by the Assembler?

OR

Explain the format of Header record, Text record and End record.

Contains 3 types of records:

Header record:

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

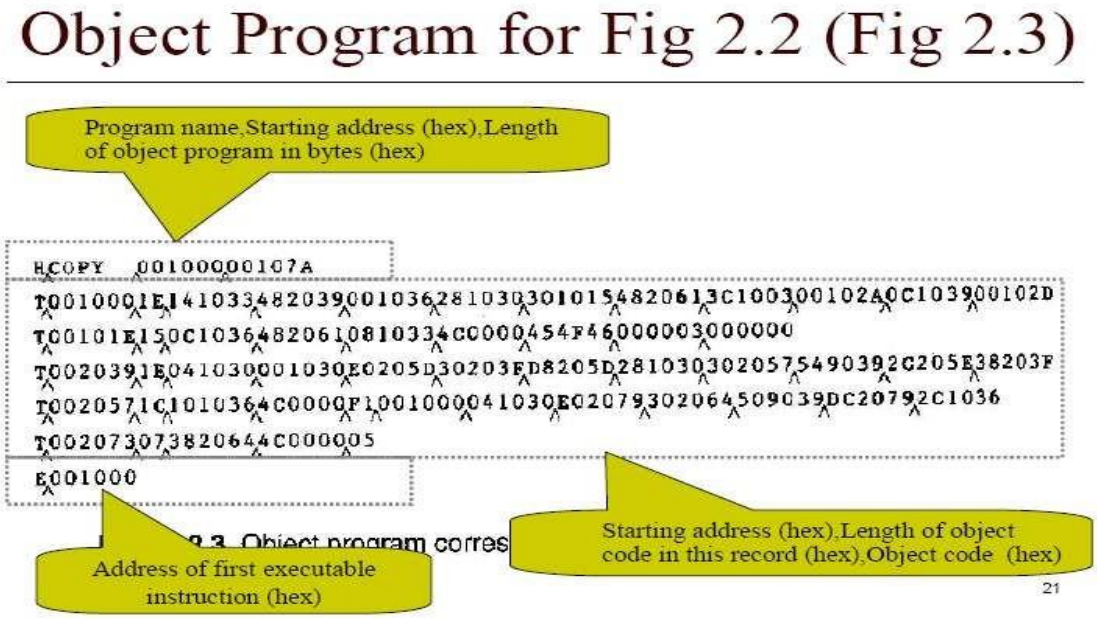
Text record

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

End record

Col.1 E
Col.27 Address of first executable instruction (hex)
(END program_name)

11. Give an example of object program generated by an Assembler.



12. What are the Data Structures used in an Assembler?

Data Structures:

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

1. OPTAB(Operation Code table)

- **Content**

  Mnemonic operation code and its machine language equivalent.
  In more complex assemblers this table also contain the information of instruction format and length.

- **Usage**

  Pass 1: OPTAB is used to look up and validate operation codes in the source program.
  Pass 2: It is used to translate the operation code to machine language equivalent.

Characteristic

Static table. Entries are not normally added to or deleted from it.

- **Implementation**

  Hash table for efficiency of *insertion* and *retrieval with mnemonic opcode as a key.*

OPTAB for the sample program (fig 8.1)

mnemonic	Opcode	format
STL	**14**	**3**
LDB	**68**	**3**
JSUB	**48**	**4**
LDA	**00**	**3**

2. SYMTAB (symbol table)

- **Content**
  Label name and its value (address)
  May also include flag (type, length) etc.
- **Usage**
  Pass 1: labels are entered into SYMTAB with their address (from LOCCTR) as they are encountered in the source program
  Pass 2: symbols used as operands are looked up in SYMTAB to obtain the address to be inserted in the assembled instruction
- **Characteristic**
  Dynamic table (insert, delete, search)
- **Implementation**
  Hash table for efficiency of *insertion* and *retrieval*

Label	Address
FIRST	0000
CLOOP	0006
RDREC	1036

SYMTAB for the sample program (fig 8.1)

3. Location Counter(LOCCTR)

- A variable used to help in *assignment of addresses*
- Initialized to the beginning address specified in the START statement
- Counted in bytes
- When a label in the source program is read, the current value of LOCCTR gives the address to be associated with that label.

13. What are the machine dependant features of a SIC/XE Assembler?

- **Instruction formats and addressing modes**
- **Program relocation**

Instruction format sand addressing modes.

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} (1 MB). This supports four different types of instruction types, they are:

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

Format 1 (1 byte)		
op(8)		
Format 2 (2 bytes)		
op(8)	r1(4)	r2(4)

Format 3(3 bytes)

op(6)	n	i	x	b	p	e	disp(12)
	1	1	1	1	1	1	

Format 4(4 bytes)

op(6)	n	i	x	b	p	e	address (20)
	1	1	1	1	1	1	

• 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 pass 2, 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 refer chapter 1. 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.

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.

Immediate Addressing Mode

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

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.

14) What is Program Relocation?

Program relocation

- **Principles.**
  - The load address of an object program is unknown at assembly time if the system implements the multiprogramming feature.
  - The assembler generates addresses relative to zero in the object program.
  - At load time, relocation is performed by adding the load address to the relative addresses.
  - Operands of instructions that use direct addressing must be relocated, and the assembler provides the relocation information in the object program.
  - Operands of instructions that use relative addressing do not need to be relocated.
  - Relocation can be processed by the loader or by the CPU using relocation registers.

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.

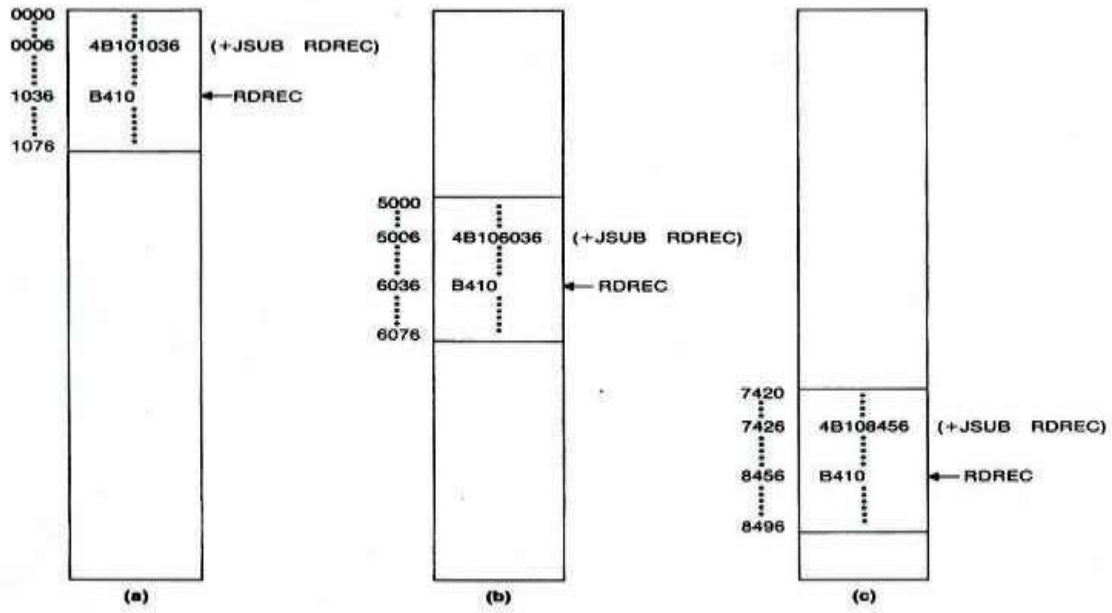


Figure 2.7 Examples of program relocation.

- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction 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.
- When assembler generates the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program.
- The assembler will 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.
- **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.**

Advantages of program relocation

- The larger main memory of SIC/XE
  - Several programs can be loaded and run at the same time.

- This kind of sharing of the machine between programs is called ***multiprogramming***

➢ To take full advantage
- Load programs into memory wherever there is space.
- Not specifying a fixed address at assembly time.

15. Explain Modification Record.

Col 1: M

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

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

16. Define Literals. Explain literal table and literal pool.

Or

Describe how assembler handles literal operands.

Let programmers to be able to write the value of a constant operand as a part of the instruction that uses it. This avoids having to define the constant elsewhere in the program and make up a label for it. Such an operand is called a literal because the value is literally stated in the instruction.

Note that a literal is identified with the prefix = which followed by a specification of the literal value.

For ex:

45 001A ENDFIL LDA =C" EOF" 032010

Specifies a 3 byte operand with value "EOF".

Literals vs. Immediate Operands

➢ *Immediate Operands*

The operand value is assembled as part of the machine instruction

e.g. 55 0020 LDA #3 010003

➢ *Literals*

The assembler generates the specified value as a constant at some other memory location

e.g. 45 001A ENDFIL LDA = C"EOF" 032010

Literal pools

➢ All of the literal operands used in a program are gathered together into one or more literal pools.
➢ Normally literals are placed into a pool at the end of the program .
➢ In some cases, it is desirable to place literals into a pool at some other location in the object program .
➢ For this purpose **assembler directive LTORG** is used.
➢ Reason: keep the literal operand close to the instruction.

1. When the assembler encounters a LTORG statement , it creates a literal pool that contains all of the literal operands used since the previous LTORG.
2. This literal pool is placed in the object program at the location where the LTORG directive was encountered.
3. Literal placed in a pool by LTORG will not be repeated in the pool at the end of the program.

Duplicate literals:

e.g. 215 1062 WLOOP TD =X"05"
e.g. 230 106B WD =X"05"

- The assemblers should recognize duplicate literals and store only one copy of the specified data value .
- Only one data area with this value is generated. Both the instruction refer to the same address in the literal pool for their operand.

The easiest method to recognize duplicate literals is by:

- Comparison of the defining expression
  • Same literal name with different value, e.g. LOCCTR=*
- Comparison of the generated data value
  • The benefits of using generate data value are usually not great enough to justify the additional complexity in the assembler .

LITTAB:

The basic data structure that assembler handles literal operands is literal table. For each literal used, this table contains **the literal name, the operand value, the length and the address assigned to the operand** when its placed in the literal pool.

Pass 1

- Build LITTAB with literal name, operand value and length, leaving the address unassigned .
- When LTORG statement is encountered, assign an address to each literal not yet assigned an address

Pass 2

- Search LITTAB for each literal operand encountered .
- Generate the data values using BYTE or WORD statement.
- Generate modification record for literals that represent an address in the program.

Literal	hex value	length	address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

17. Explain symbol defining statements

Or

Explain the following : a. EQU Directive. B. ORG Directive.

The user defined symbols in the assembler language program appears as labels on instructions or data areas. The value of such a label is the address assigned to the statement on which it appears. Most

assemblers provide an assembler directive that allows the user to define the symbol and specify their values. **EQU is the assembler directive used.**
The general form of such a statement is
Symbol EQU Value
It defines the given symbol and assigns the specified value. The value may be a

a. Constant
b. Any expression involving constant.
c. Previously defined symbol.

EQU is used to establish symbolic names that can be used for improved readability in place of numeric values.

- In the last program, the statement used as

+LDT #4096 // to load the value 4096 into register T.

- If we include the statement

MAXLEN EQU 4096

- In the program then we can write it as

+LDT #MAXLEN

- When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB with the value 4096.

- Another common use of the EQU is defining mnemonic names for registers.

For ex

```
A       EQU     0
X       EQU     1
L       EQU     2
```

These statement cause the symbols A,X,L… to be entered into SYMBOL with their corresponding values 0.1,2…

ORG DIRECTIVE:

- Indirectly assigns values to symbols. This is called Origin directive.
- Resets the location counter value to the specified value.
- Since the values of the symbols are taken from the LOCCTR, the ORG statement will affect the values of all the labels defined until the next ORG.
- EX:
  To define a symbol table with the following structure:
  a. SYMBOL field: 6 bytes
  b. VALUE field: one word.
  c. FLAGS field : 2 byte

	SYMBOL	VALUE	FLAGS
STAB (100 entries)			

Symbol field contains user defined symbol: Value field represents the value assigned to the symbol and the Flags field specifies symbol type and other information.
To reserve the space we write

```
STAB RESB  1100
```

We also define the labels as SYMBOL,VALUE, FLAGSS using with EQU statements:

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

To fetch the value field

```
LDA  VALUE,X
```

We can accomplish the same symbol definition using ORG in the following way.

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

a. The first ORG resets the LOCCTR to the value of STAB.
b. RESB statement defines the SYMBOL to have the current value in LOCCTR.
c. LOCCTR is then advanced so that the label on the RESW statement assigns to VALUE the address(STAB+6) and so on..
d. So that each entry in STAB consists of 6 byte SYMBOL, followed by one word VALUE, followed by a 2 byte FLAGS.
e. The last ORG statement set LOCCTR back to its previous value.

Notice that the two pass assembler design requires that all symbols be defined during Pass1.
For ex:
The sequence

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

would be allowed, whereas the sequence

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

would not be allowed.
Reason for this is symbol definition process.

ORG has the same type of restriction.
Consider the following example

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

The above sequence cannot be processed because assembler directive would not know the value

assigned to the location counter in response to the first ORG statement. The symbols BYTE1, BYTE2, BYTE3 cant be assigned addresses during PASS1.

18. Explain absolute and relative expressions. How these are processed by an assembler.

Most assemblers allow the use of the expressions whenever a single operand is permitted. Each such expression has to be evaluated by the assembler to produce a single operand address or value.
Expressions can be classified as

- ABSOLUTE EXPRESSIONS
- RELATIVE EXPRESSIONS.

Depending upon the type of the value they produce.

RELATIVE EXPRESSIONS:

Relative means relative to the beginning of the program. labels on the instructions and data areas and references to the location counter values are relative terms. No relative term into multiplication or division operation.

ABSOLUTE EXPRESSIONS.

Absolute means independent of program location. A constant is an absolute term. Absolute expressions may also contain relative terms provided they occur in pair and the terms in each pair have opposite sign.
A relative term or expression represents some value that may be written as (S+r) where
S= Starting address of the program
R= value of the term or expression relative to the starting address.

Example: 107 MAXLEN EQU BUFFEND-BUFFER
Both BUFFEND and BUFFER are relative terms, each representing an address within the program. However the expression BUFFEND-BUFFER represents the absolute term.
To determine the type of an expression we must keep track of the type of the symbols defined in the program.
Following table shows the symbol table entries.

Symbol	Type	Value
RETADR	R	0000
BUFFER	R	0036
BUFFEND	R	1036
MAXLEN	A	1000

19. Explain how the assembler handles multiple 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]

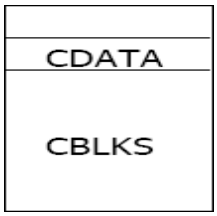
At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather 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 endof the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

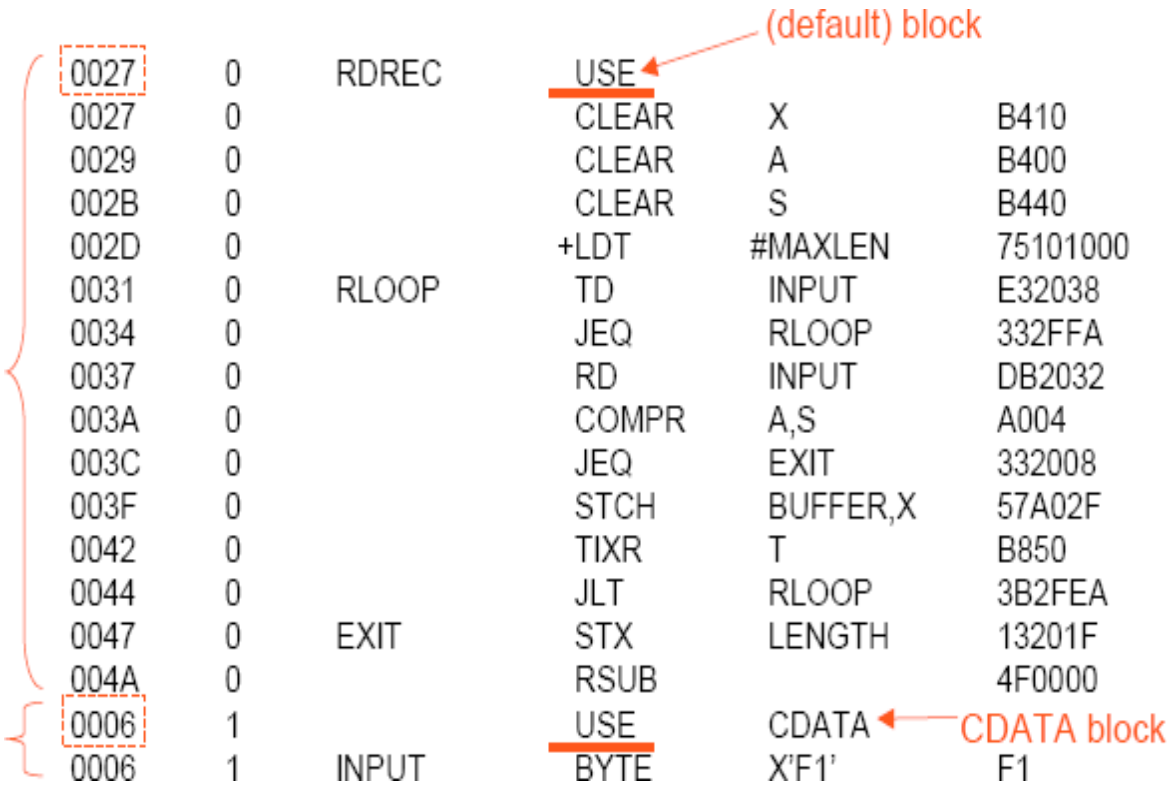
CBLKS: all data areas that consists of larger blocks of memory



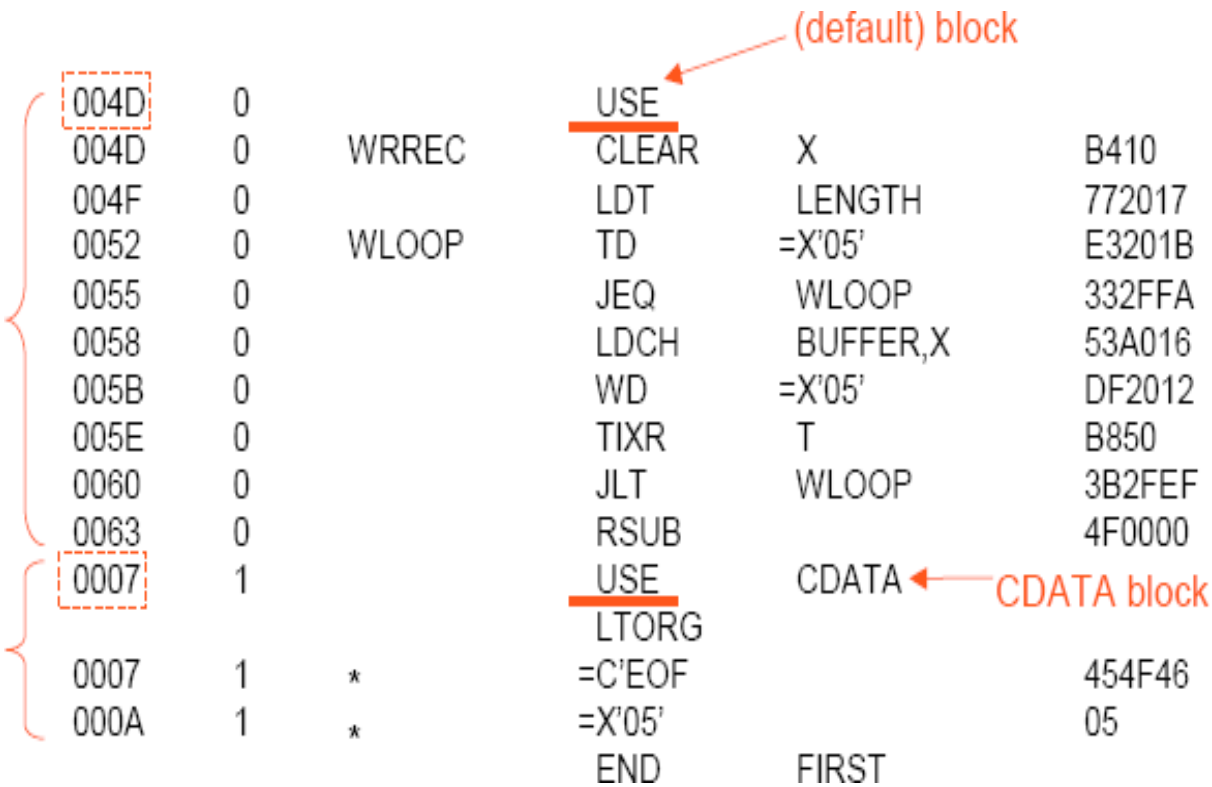
(default) block — Block number

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	

Example Code



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



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	
			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

20. Explain control sections and program linking in assemblers.

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 assembleris 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

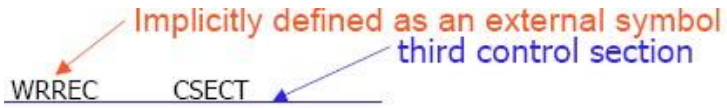
first control section

```
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
```




```
.
.           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 ReferenceCase 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 formatto provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loaderto 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
```

21. Explain the format of Define and Refer Records. What are their uses?

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 may 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

```
HCOPY  000000001033
DBUFFER00033BUFEND001033LENGTH00002D
RRDREC WRREC
T000000 1D 1720274B100000 032023 290000 3320074B100000 3F2FEC 032016 0F2016
T00001D 0D 0100030F200A4B1000003E2000
T000030 03 454F46
M000004 05 +RDREC
M000011 05 +WRREC
M000024 05 +WRREC
E000000
```

RDREC

```
HRDREC  000000 00002B
RBUFFER LENGTH BUFEND
T000000 1D B410 B400 B440 77201F E3201B 332FFA DB2015 A00433 200957 900000 B850
T00001D 0E 3B2FE9 131000004F0000 F1 000000
M000018 05 +BUFFER
M000021 05 +LENGTH
M000028 06 +BUFEND
M000028 06 -BUFFER
E
```

BUFEND - BUFFER

WRREC

```
HWRREC  000000 00001C
RLENGTH BUFFER
T000000 1C B410 77100000E3201 232FFA 53900000DF2008 B850 3B2FEE 4F000005
M000003 05 +LENGTH
M00000D 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 anextended restriction that:

- Both terms in each pair of an expression must be within the same control section
    - If two terms represent relative locations within the same control section , theirdifference is an absolute value (regardless of where the control section is located.
        - **Legal:** BUFEND-BUFFER (both are in the same control section)

    - 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 ofany use.

- How to enforce this restriction
    - When an expression involves external references, the assembler cannotdetermine whether or not the expression is legal.
    - The assembler evaluates all of the terms it can, combines these to form aninitial expression value, and generates Modification records.
    - The loader checks the expression for errors and finishes the evaluation.