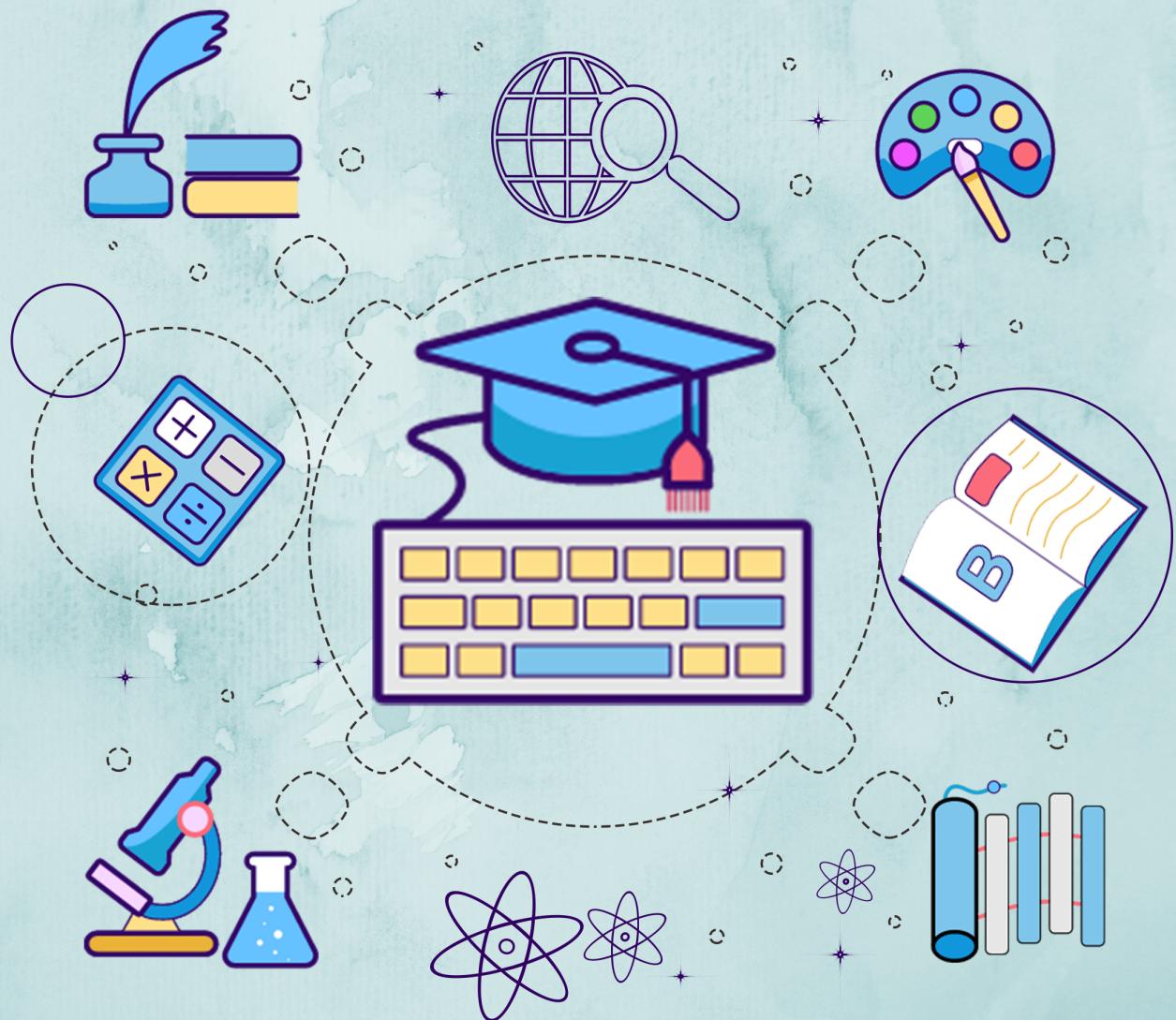


Kerala Notes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS



KTU STUDY MATERIALS

SYSTEM SOFTWARE

CST 305

Module 3

Related Link :

- KTU S5 STUDY MATERIALS
- KTU S5 NOTES
- KTU S5 SYLLABUS
- KTU S5 TEXTBOOK PDF
- KTU S5 PREVIOUS YEAR
SOLVED QUESTION PAPER

Module-3

Machine-Dependent Assembler Features:

In this section we consider the design and implementation of SIC/XE assembler.

- Instruction formats and addressing modes
- Program relocation.

Instruction formats and Addressing Modes

1. Translation of Register to Register instructions

In this the assembler must simply convert the opcode to machine language and change each register to its numeric value.

Eg:

COMPR A, S A004

(The opcode for COMPR is A0 , the number of register A is 0 and register S is 4.)

2. Translation of Format 4 instructions

This format contains 20 bit address field . No displacement is calculated.

Eg:

CLOOP +JSUB RDREC 4B101036

Here the opcode for JSUB instruction is 48 and the address of RDREC is 1036. Write the instruction format and set the bits n, i and e to 1.

(If neither immediate nor indirect mode is used set the bits n and i to 1. Format 4 is identified by the prefix + . If format 4 is not specified assembler first attempts to translate the instruction using program counter relative addressing. If this is not possible, (because the required displacement is out of range), the assembler then attempts to use base relative addressing. If neither form of relative addressing is applicable and the extended format is not specified then the instruction can not be properly assembled. In this case the assembler must generate an error message.)

3. Translation PC relative instructions

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In PC relative addressing made $TA = \text{disp} + [\text{PC}]$

$$\text{disp} = TA - [\text{PC}]$$

Eg:1

0000 FIRST STL RETADR 17202D

$(14)_{16}$ 1 1 0 0 1 0 $(02D)_{16}$

$\Rightarrow \text{displacement} = \text{RETADR} - \text{PC} = 30-3 = 2D$

Eg: 2

0017 J CLOOP 3F2FEC

$(3C)_{16}$ 1 1 0 0 1 0 $(FEC)_{16}$

displacement= CLOOP-PC= 6 - 1A= -14= FEC

4. Translation of Base relative instructions

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In Base relative addressing made TA = disp + [B]

$$\text{disp} = \text{TA} - [\text{B}]$$

The displacement calculation process for base relative addressing is much the same as for PC relative addressing. In this the programmer must tell the assembler what the base register will contain during execution of the program so that assembler can compute displacements. This is done with the assembler directive BASE. For example, the statement BASE LENGTH informs the assembler that the base register will contain the address of LENGTH. The register B will contain this address until another BASE statement is encountered.

If the base register has to be used for another purpose the programmer must use NOBASE directive to inform the assembler that the contents of the base register is not used for addressing.

104E	LDB #LENGTH BASE LENGTH STCH BUFFER, X	57C003
------	---	--------

$(54)_{16}$ 1 1 1 1 0 0 $(003)_{16}$

(54) 1 1 1 0 1 0 $0036-1051 = -101B_{16}$

displacement= BUFFER - B = 0036 - 0033 = 3

5. Translation of Immediate addressing

In this no memory reference is involved. Convert the immediate operand into its internal representation and insert it into its internal representation.

Eg:

◆ 0020 LDA #3 010003

$(00)_{16}$ 0 1 0 0 0 0 $(003)_{16}$

◆ 103C +LDT #4096 75101000

$(74)_{16}$ 0 1 0 0 1 $(01000)_{16}$

6. Translation involving indirect addressing

In this the displacement is computed to produce the target address.. Then bit n is set to 1. The example given below is indirect and PC relative.

Eg:

002A J @RETADR 3E2003

(3C)₁₆ 1 0 0 0 1 0 (003)₁₆

- ⇒ TA=RETADR=0030
- ⇒ TA=(PC)+disp=002D+0003

Program Relocation

- Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.
- Absolute Program- In this the address is mentioned during assembling itself. This is called *Absolute Assembly*.

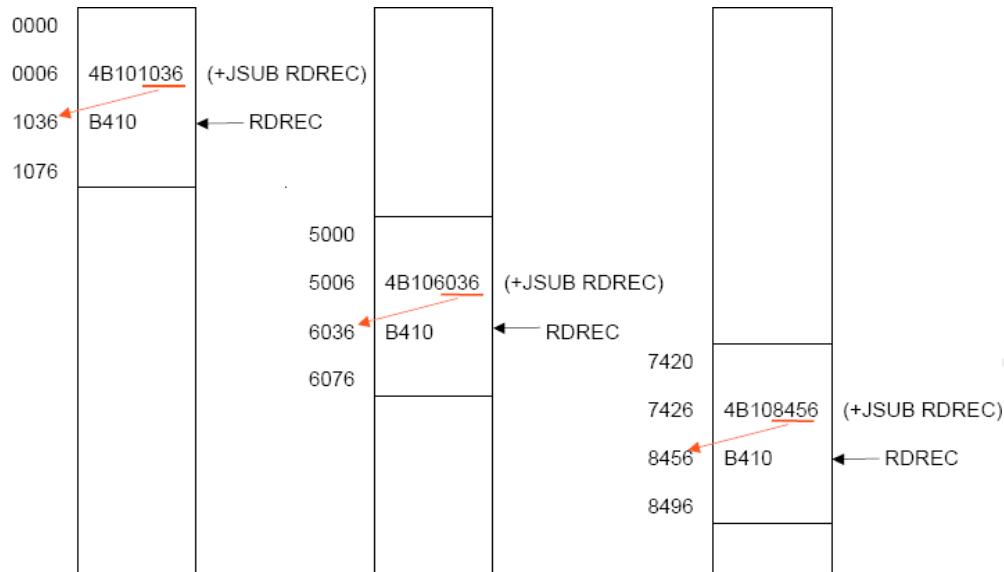
Eg: Consider the instruction:

101B LDA THREE 00102D

- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.
- Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.
- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler

identifies for the loader those parts of the program which need modification.

- An object program that has the information necessary to perform this kind of modification is called the **relocatable program**.



- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.
- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses(format 4 instructions). The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of

the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1 M

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

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

One modification record is created for each address to be modified. The length is stored in half-bytes (4 bits). The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

Eg: Consider the instruction

CLOOP +JSUB RDREC 4B101036

where RDREC is at the address 1036. The modification record for this instruction can be written as

M00000705

- There is one modification record for each address field that needs to be changed when the program is relocated(ie. For each format 4 instructions in the program).

Machine-Independent features:

These are the features which do not depend on the architecture of the machine. Such features are more related to software than to machine architecture. These are:

- Literals
- Symbol defining statements
- Expressions
- Program blocks
- Control sections

Literals:

- It is easy for a programmer to write the value of a constant operand as part of the instruction that uses it.
- This avoids defining the constant elsewhere in the program and making a label for it. Such an operand is called a literal because the value is stated literally in the instruction.
- A literal is defined with a prefix = followed by a specification of the literal value.

Example:

001A	ENDFIL	LDA	=C'EOF'	032	010
------	--------	-----	---------	-----	-----

-
-

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

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

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

0020 LDA #03 010003

- All the literal operands used in a program are gathered together into one or more *literal pools*. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values.

Eg: 1076 * =X'05' 05

- In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

LTORG

002D * =C'EOF' 454F46

- Recognizing Duplicate literals** – That is the same literal used in more than one place in a program and store only one copy of the data value. For example, the literal =X'05' is used in different instructions in a program, but only one data area with this value is created.

- Duplicate literals can be identified by comparing character strings. Eg: X'05'
- Otherwise, generated value can be compared. For eg: the literals =C'EOF' and =X'454F46' are identical operand values.

- The value of some literals depends on their location in the program. Literals referring to the current value of the location counter (denoted by the symbol *) . Such literals are useful for loading base registers.

Eg: BASE *
 LDB *

Such literal operands will have different values in different places of the program since they hold the current value of the location counter.

- Handling of literals by the assembler** - A literal table is created for the literals which are used in

the program. The literal table contains the literal name, operand value and length and the address assigned to the operand. The literal table is usually created as a hash table using the literal name or value as the key.

- **During Pass-1:** The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB (leaving the address unassigned). When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.
- **During Pass-2:** The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value.

Symbol-Defining Statements:

EQU Statement:

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

Symbol	EQU	value
--------	-----	-------

- This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example , instead of +LDT #4096 we can write

MAXLEN	EQU	4096
--------	-----	------

+LDT	#MAXLEN
------	---------

- When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During the assembly of LDT instruction the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).
- Another common usage of EQU statement is **for defining values for the general-purpose registers**. The assembler can use the mnemonics for register usage like a-register A , X – index register and so on. But there are some instructions which requires numbers in place of names in the instructions. For example in the instruction RMO 0,1 instead of RMO A,X. The programmer can assign the numerical values to these registers using EQU directive.

A EQU 0

X EQU 1 and so on

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

BASE EQU R1

INDEX EQU R2

COUNT EQU R3

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

BETA EQU ALPHA

ALPHA RESW 1

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

ORG Statement:

- This directive can be used to indirectly assign values to the symbols. This assembler directive changes the value in the location counter. The directive is usually called ORG (for origin). Its general format is:

ORG value

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

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

SYMBOL 6 Bytes

VALUE 3 Bytes

FLAG 2 Bytes

The table looks like the one given below.

STAB (100 entries)	SYMBOL	VALUE	FLAGS
⋮	⋮	⋮	⋮

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

STAB RESB 1100

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

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

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

LDA VALUE, X

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

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

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

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

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

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

Expressions:

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

```
BUFFEND EQU *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants).

- Expressions are classified as either absolute expression or relative expressions , neither absolute nor relative depending on the type of value they produce.
 - Absolute Expressions:** The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. None of the relative terms enter into multiplication or division. Example:

```
MAXLEN EQU BUFEND-BUFFER
```

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

```
MAXLEN EQU 1000
```

- Relative Expressions:** All the relative terms except one can be paired . The remaining unpaired relative term must have a positive sign. None of the relative terms must enter into multiplication or division. A relative term represents some location within the program. Example:

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

- **Neither absolute nor relative:** Expressions that are legal are those expressions whose value remains meaningful when the program is relocated. Expressions that do not meet the conditions for either absolute or relative are neither absolute nor relative. They are considered as errors.

Eg: BUFEND + BUFFER, 100-BUFFER, 3*BUFFER

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

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

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.
- Program blocks refer to segments of code that are rearranged within a single object program unit.
- **Assembler Directive USE:** indicates which portion of the program belong to the various blocks.

USE [blockname]

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

the statements that reference them. In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

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

Example Code

		(default) block		Block number			
0000	0	COPY	START	0			
0000	0	FIRST	STL	RETADR	172063		
0003	0	CLOOP	JSUB	RDREC	4B2021		
0006	0		LDA	LENGTH	032060		
0009	0		COMP	#0	290000		
000C	0		JEQ	ENDFIL	332006		
000F	0		JSUB	WRREC	4B203B		
0012	0		J	CLOOP	3F2FEE		
0015	0	ENDFIL	LDA	=C'EOF'	032055		
0018	0		STA	BUFFER	0F2056		
001B	0		LDA	#3	010003		
001E	0		STA	LENGTH	0F2048		
0021	0		JSUB	WRREC	4B2029		
0024	0		J	@RETADR	3E203F		
0000	1		USE	CDATA	CDATA block		
0000	1	RETADR	RESW	1			
0003	1		RESW	1			
0000	2		USE	CBLKS	CBLKS block		
0000	2	BUFFER	RESB	4096			
1000	2	BUFEND	EQU	*			
1000		MAXLEN	EQU	BUFEND-BUFFER			
(default) block							
0027	0	RDREC	USE				
0027	0		CLEAR	X	B410		
0029	0		CLEAR	A	B400		
002B	0		CLEAR	S	B440		
002D	0		+LDT	#MAXLEN	75101000		
0031	0	RLOOP	TD	INPUT	E32038		
0034	0		JEQ	RLOOP	332FFA		
0037	0		RD	INPUT	DB2032		
003A	0		COMPR	A,S	A004		
003C	0		JEQ	EXIT	332008		
003F	0		STCH	BUFFER,X	57A02F		
0042	0		TIXR	T	B850		
0044	0		JLT	RLOOP	3B2FEA		
0047	0	EXIT	STX	LENGTH	13201F		
004A	0		RSUB		4F0000		
0006	1	INPUT	USE	CDATA	CDATA block		
0006	1		BYTE	X'F1'	F1		

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

- How the assembler handles program blocks –

Pass 1

- A separate location counter for each block is maintained.
- The location counter for a block is initialized to zero when the block is first started.
- The current value of the location counter is saved when switching to another block.
- The saved value is continued when resuming previous block.
- After pass 1 the symbol table will be having labels with block no along with address.(For absolute symbol there is no block number.)
- At the end of pass 1 latest value of location counter of each block gives the length of that block.
- Assembler constructs a block table that contains starting addresses and lengths of all blocks

Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

Pass 2

- Code generation during pass2 the assembler needs the address relative to the start of the

program. (not the start of the individual program block). Assembler adds the label address with its block starting address.

Pass1 algorithm of Program blocks

Pass2 algorithm for program blocks

- **Advantage-** Separation of programs into blocks has reduced the addressing problem. Since the larger buffer area is moved to the end of the object program extended format instructions need not be used. The use of program blocks has achieved the effect of rearranging the source statements without actually rearranging them. The loader will load the object program at the indicated address.

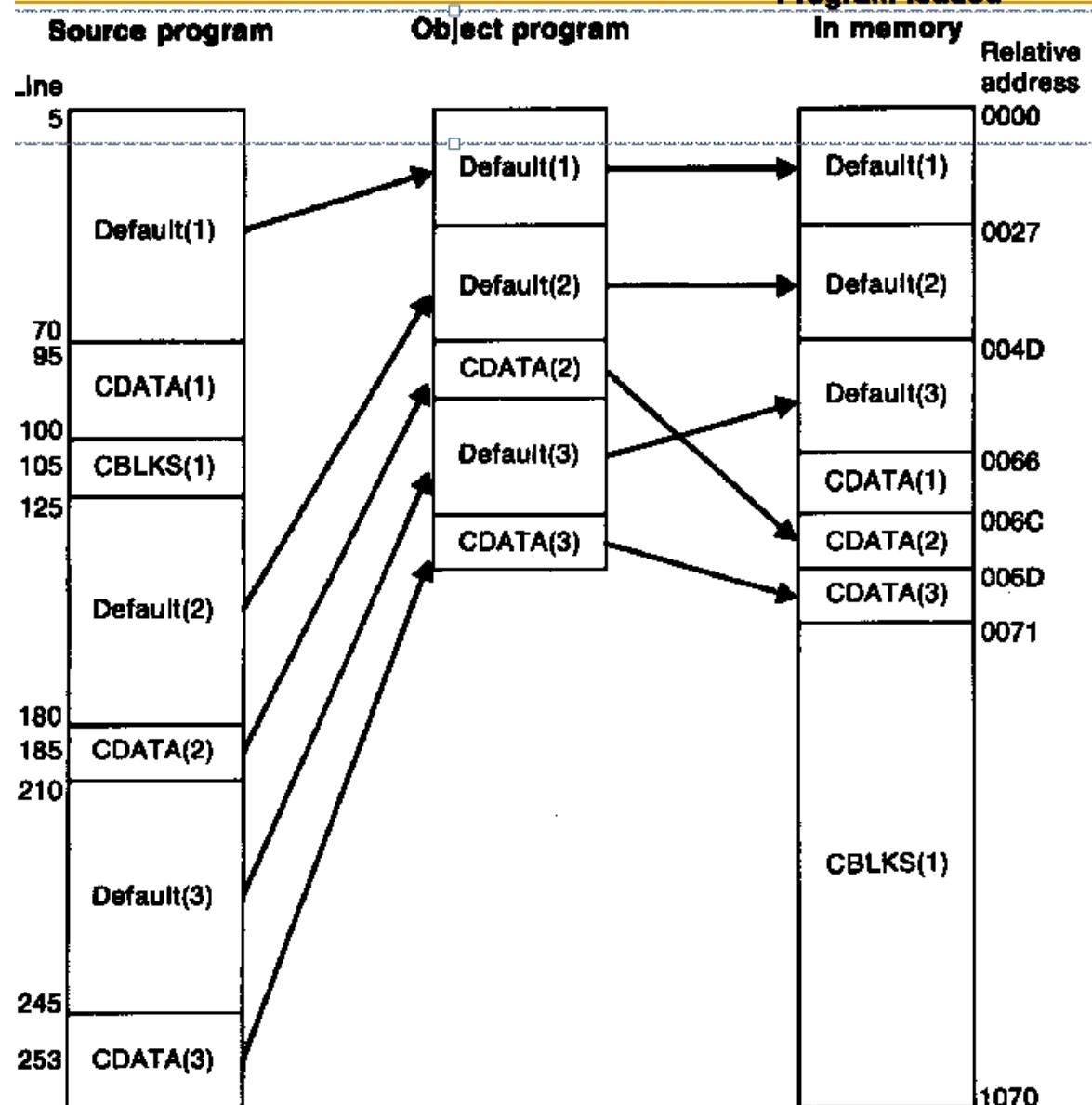


Fig:Program blocks traced through the assembly and loading processes

Pass1 of program blocks

```

begin
    block number = 0 LOCCTR[i] = 0 for all i
    read the first input line .
    if OPCODE = 'START' then
        begin
            write line to intermediate file
            read next input line
        end {if START}
    while OPCODE ≠ 'END' do
        if OPCODE = 'USE'
            begin
                if there is no OPEREND name then
                    set block name as default
                else block name as OPERAND name
                if there is no entry for block name then
                    insert (block name, block number++) in block table
                    i = block number for block name
                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[i]) into SYMTAB
                            end {if symbol}
                        Search OPTAB for OPCODE
                        if found then
                            add 3 instruction length to LOCCTR[i]
                        else if OPCODE = 'WORD' then
                            add 3 to LOCCTR[i]
                        else if OPCODE = 'RESW' then
                            add 3 * #[OPERAND] to LOCCTR[i]
                        else if OPCODE = 'RESB' then
                            add #[OPERAND] to LOCCTR[i]
                        else if OPCODE = 'BYTE' then
                            begin
                                find length of constant in bytes
                                add length to LOCCTR[i]
                            end {if byte}
                        else

```

```

Set error flag
end {if not a comment}
write line to intermediate file
read Text input line
end {while not END}
write last line to intermediate file
save Length[i] as LOCCTR[i] for all i
Address[0] = starting address
Address[i] = address(i - 1) + Length(i - 1)
    [for i = 1 to max(block number)]
insert (address[i], Length[i]) in block table for all i
end {Pass 1}

```

Pass2 of Program blocks

```

If OPCODE = 'USE' then
    set block number for block name with OPERAND field
    search SYMTAB for OPERAND
    store symbol value + address [block number] as operand address
end {Pass 2}

```

Control Sections:

- A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others.
- Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.
- Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections.
- Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references*.
- The assembler generates the information about each of the external references that will allow the loader to perform the required linking.
- When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive
 - assembler directive: **CSECT**

The syntax

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

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

**Implicitly defined as an external symbol
third control section**

WRREC	CSECT	
-------	-------	--

:

SUBROUTINE TO WRITE RECORD FROM BUFFER

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

Handling External Reference

Case 1

15 0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference.
 - The assembler has no idea where RDREC is
 - inserts an address of zero
 - can only use extended 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 loader to perform the required linking.

Case 2

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

Case 3

- 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

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
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	Case 2

<u>0000</u>	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	
0027	INPUT	BYTE	X'F1' F1
0028	MAXLEN	WORD	BUFFEND-BUFFER 000000 Case 3

<u>0000</u>	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	
		END	FIRST 4F0000
001B	*		=X'05' 05

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

Define record (EXTDEF)

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

Refer record (EXTREF)

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

Modification record

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

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF. A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification 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.

COPY

HCOPY 000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B100000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000

RDREC

HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F000QF1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER } BUFEND - BUFFER
E

WRREC

HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E

- 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*.
- **Handling Expressions in Multiple Control Sections:** The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).
- When it comes in a program having multiple control sections then we have an extended restriction that:
 - Both terms in each pair of an expression must be within the same control section
If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located).

Legal: BUFEND-BUFFER (both are in the same control section)

- If the terms are located in different control sections, their difference has a value that is unpredictable.
- Illegal:** RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.

- **How to enforce this restriction**

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

Assembler Design Options

- There are two design options or the assembler.
 - One pass assembler: is used when it is necessary to avoid a second pass over the source program.
 - Multipass Assembler: allows an assembler to handle forward references.

One-Pass Assembler

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

- One that produces object code directly in memory for immediate execution (**Load-and-go assemblers**).
- The other type produces **the usual kind of object code** for later execution.

Load-and-Go Assembler

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.

- It is useful in a system with frequent program development and testing
 - The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

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

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

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.

- For Load-and-Go assembler
 - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

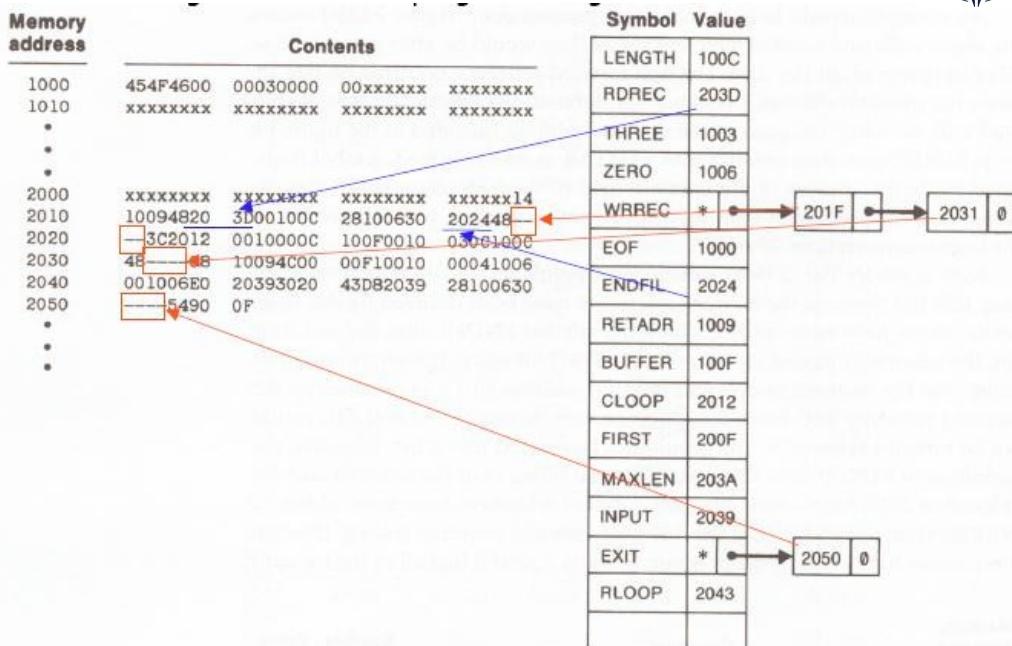
After Scanning line 40 of the program:

40 2021 J CLOOP 302012

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

Memory address	Contents				Symbol	Value
1000	454F4600	00030000	00xxxxxx	xxxxxxxx	LENGTH	100C
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	RDREC	* → 2013 0
•					THREE	1003
•					ZERO	1006
•					WRREC	* → 201F 0
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14	EOF	1000
2010	100948	--00100C	28100630	---48--	ENDFIL	* → 201C 0
2020	3C2012				RETADR	1009
•					BUFFER	100F
•					CLOOP	2012
•					FIRST	200F

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



One-Pass Assembler that generates object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

```

HCOPY 00100000107A
T00100009454F46000003000000
T00200F1514100948000000100C2810063000004800003C2012
T00201C022024
T002024190010000C100F0010030C100C4800000810094C0000F1001000
T00201302203D
T00203D1E041006001006E02039302043D8203928100630000054900F2C203A382043
T00205002205B
T00201F022062
T002031022062
T00206218041006E0206130206550900FDC20612C100C3820654C0000
E00200F

```

Algorithm for one pass assembler

```

begin
    read first input line
    if OPCODE = 'START' then
        begin
            save #[OPERAND] as starting address
            initialize LOCCTR as starting address
            read next input line
        end {if START}
    else
        initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
        if there 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
                            begin
                                if symbol value as null
                                set symbol value as LOCCTR and search
                                    the linked list with the corresponding
                                    operand
                                    PTR addresses and generate operand
                                    addresses as corresponding symbol
                                    values
                                set symbol value as LOCCTR in symbol
                                    table and delete the linked list
                            end
                        else
                            insert (LABEL, LOCCTR) into SYMTAB
                    end
                end
                search OPTAB for OPCODE
                if found then
                    begin
                        search SYMTAB for OPERAND address
                    if found then
                        if symbol value not equal to null then
                            store symbol value as OPERAND address
                        else
                            insert at the end of the linked list
                            with a node with address as LOCCTR
                        else
                            insert (symbol name, null)
                    end
                end
            end
        end
    end

```

```
        add 3 to LOCCTR
    end
else if OPCODE = 'WORD' then
    add 3 to LOCCTR & convert comment to
    object code
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
    convert constant to object code
end
if object code will not fit into current
text record then
begin
    write text record to object program
    initialize new text record
end
add object code to Text record
end
write listing line
read next input line
end
write last Text record to object program
write End record to object program
write last listing line
end {Pass 1}
```

MultiPass Assembler:

- For a two pass assembler, in EQU assembler directive we required that any symbol on the right hand side be defined previously in the program. This is because of the two pass. If multipass is possible this restriction can be avoided. Eg:

```
ALPHA EQU BETA
```

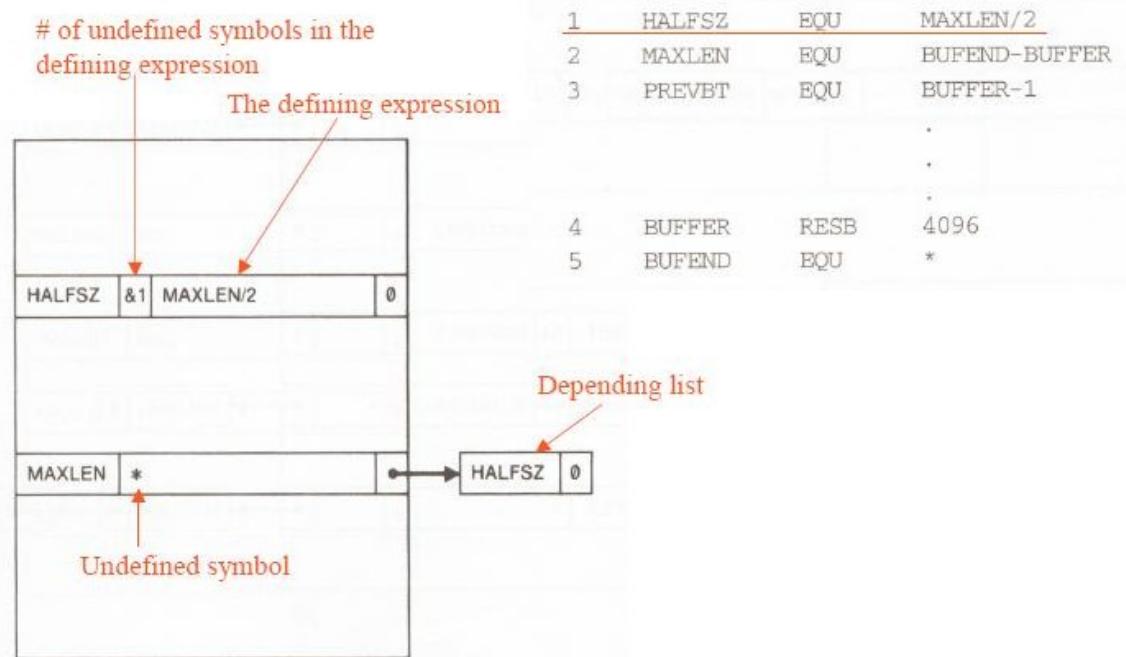
```
BETA EQU DELTA
```

```
DELTA RESW 1
```

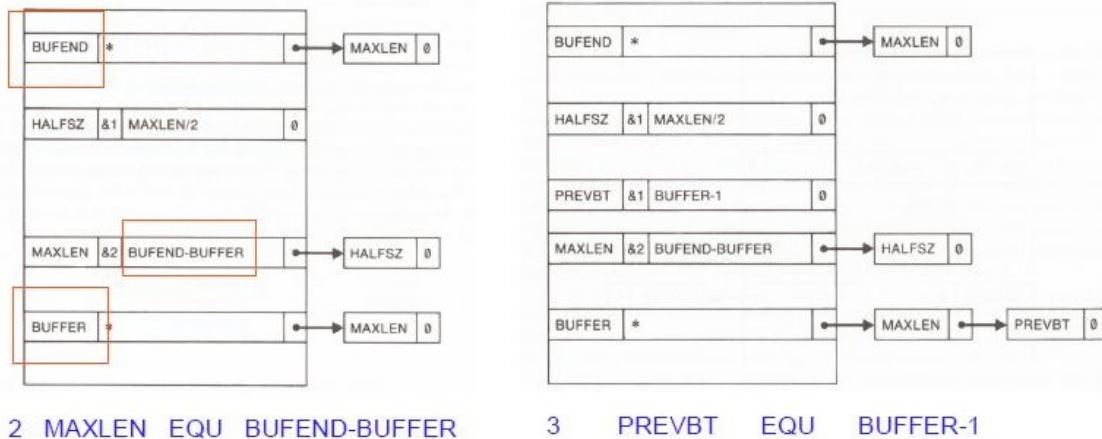
Working of Multipass Assembler:

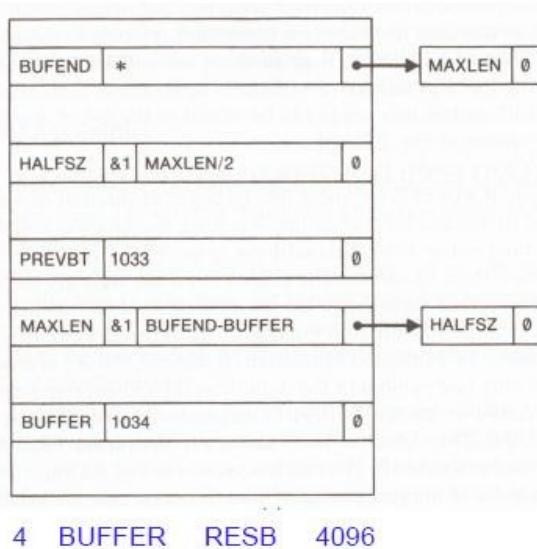
- A multipass assembler can make as many passes as needed to process the definition of symbols.
- For a forward reference in symbol definition, we store in the SYMTAB:
 - The symbol name
 - The defining expression
 - The number of undefined symbols in the defining expression
- The undefined symbol (marked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

Multi-Pass Assembler Example Program



Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:





BUFEND	2034	0
HALFSZ	800	0
PREVBT	1033	0
MAXLEN	1000	0
BUFFER	1034	0

5 BUFEND EQU *

Implementation Example: MASM ASSEMBLER

- Microsoft MASM assembler works for Pentium and other ×86 systems.
 - In this system memory is considered as segments.
 - An MASM assembly language program is written as collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST and STACK
 - During program execution the segments are addressed via the ×86 segment registers. Code segments are addressed using register CS and stack segments are addressed using register SS. These segment registers are automatically set by the system loader when a program is loaded for execution.
 - Register CS is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is to indicate the last stack segment processed by the loader.
 - Data segments (including constant segments) are normally addressed using DS, ES, or GS.
 - By default the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME.
- ASSUME ES: DATASEG2
- Registers DS, ES, FS and GS must be loaded by the program before they can be used to address

data segments. Eg:

MOV AX, DATASEG2

MOV ES, AX

Would set ES to indicate the data segment DATASEG2

- Jump instructions are assembled in two different ways, depending on whether the target of the jump is in the same code segment (near jump) or in a different code segment(far jump).
- The length of the assembled instruction depends on the operands that are used. An operand that specifies a memory location may take varying amounts of space in the instruction depending upon the location of the operand.
- First pass of the ×86 assembler must analyze the operands of an instruction, in addition to looking at the opcode.
- Segments in a MASM source program can be written in more than one place using the assembler directive SEGMENT.
- References between segments that are assembled together are automatically handled by the assembler.
- MASM can also produce an instruction timing listing that shows the number of clock cycles required to execute each machine instruction.

KeralaNotes.com