KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Module 3

Assembler design options

Machine Independent assembler features – program blocks, Control sections, Assembler design options- Algorithm for Single Pass assembler, Multi pass assembler, Implementation example of MASM Assembler

Machine-Independent Assembler Features

- Literals
- Symbol-Defining Statements
- Expressions
- Program Blocks
- Control Sections and Program Linking

Literals

It is convenient if a programmer can 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 stated literally in the instruction.In our assembler language notation, a literal is identified with the prefix =, which is followed by a specification of the literal value, using the same notation as in the BYTE statementeg1 :

```
45        001A    ENDFIL      LDA   =C'EOF'       032010
```

Thus the literal in the statement specifies a 3-byte operand whose value is the character string EOF.

SIC/XE Program

```
 5     0000    COPY     START   0
10     0000    FIRST    STL     RETADR          17202D
12     0003             LDB     #LENGTH         69202D
13                      BASE    LENGTH
15     0006    CLOOP    +JSUB   RDREC           4B101036
20     000A             LDA     LENGTH          032026
25     000D             COMP    #0              290000
30     0010             JEQ     ENDFIL          332007
35     0013             +JSUB   WRREC           4B10105D
40     0017             J       CLOOP           3F2FEC
45     001A    ENDFIL   LDA     EOF             032010
50     001D             STA     BUFFER          0F2016
55     0020             LDA     #3              010003
60     0023             STA     LENGTH          0F200D
65     0026             +JSUB   WRREC           4B10105D
70     002A             J       @RETADR         3E2003
80     002D    EOF      BYTE    C'EOF'          454F46
95     0030    RETADR   RESW    1
100    0033    LENGTH   RESW    1
105    0036    BUFFER   RESB    4096
```

LITERAL PROGRAM

```
 5     COPY     START   0               COPY FILE FROM INPUT TO OUTPUT
10     FIRST    STL     RETADR          SAVE RETURN ADDRESS
13              LDB     #LENGTH         ESTABLISH BASE REGISTER
14              BASE    LENGTH
15     CLOOP    +JSUB   RDREC           READ INPUT RECORD
20              LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
25              COMP    #0
30              JEQ     ENDFIL          EXIT IF EOF FOUND
35              +JSUB   WRREC           WRITE OUTPUT RECORD
40              J       CLOOP           LOOP
45     ENDFIL   LDA     =C'EOF'         INSERT END OF FILE MARKER
50              STA     BUFFER
55              LDA     #3              SET LENGTH = 3
60              STA     LENGTH
65              +JSUB   WRREC           WRITE EOF
70              J       @RETADR         RETURN TO CALLER
93              LTORG
95     RETADR   RESW    1
100    LENGTH   RESW    1               LENGTH OF RECORD
105    BUFFER   RESB    4096            4096-BYTE BUFFER AREA
106    BUFEND   EQU     *
107    MAXLEN   EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
```

Use = to represent a literal

Literal Program with Object Code

```
 5     0000    COPY     START   0
10     0000    FIRST    STL     RETADR          17202D
13     0003             LDB     #LENGTH         69202D
14                      BASE    LENGTH
15     0006    CLOOP    +JSUB   RDREC           4B101036
20     000A             LDA     LENGTH          032026
25     000D             COMP    #0              290000
30     0010             JEQ     ENDFIL          332007
35     0013             +JSUB   WRREC           4B10105D
40     0017             J       CLOOP           3F2FEC
45     001A    ENDFIL   LDA     =C'EOF'         032010
50     001D             STA     BUFFER          0F2016
55     0020             LDA     #3              010003
60     0023             STA     LENGTH          0F200D
65     0026             +JSUB   WRREC           4B10105D
70     002A             J       @RETADR         3E2003
93                      LTORG
       002D     *       =C'EOF'                 454F46
95     0030    RETADR   RESW    1
```

Notice that the object code is the same as the previous one.

- Eg 2 :

215 1062 WLOOP TD =X'05' E32011

This statement specifies a 1-byte literal with the hexadecimal value 05

```
195     .
200     .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC   CLEAR   X               CLEAR LOOP COUNTER
212             LDT     LENGTH
215     WLOOP   TD      =X'05'          TEST OUTPUT DEVICE
220             JEQ     WLOOP           LOOP UNTIL READY
225             LDCH    BUFFER,X        GET CHARACTER FROM BUFFER
230             WD      =X'05'          WRITE CHARACTER
235             TIXR    T               LOOP UNTIL ALL CHARACTERS
240             JLT     WLOOP             HAVE BEEN WRITTEN
245             RSUB                    RETURN TO CALLER
255             END     FIRST
```

Literal Program with Object Code

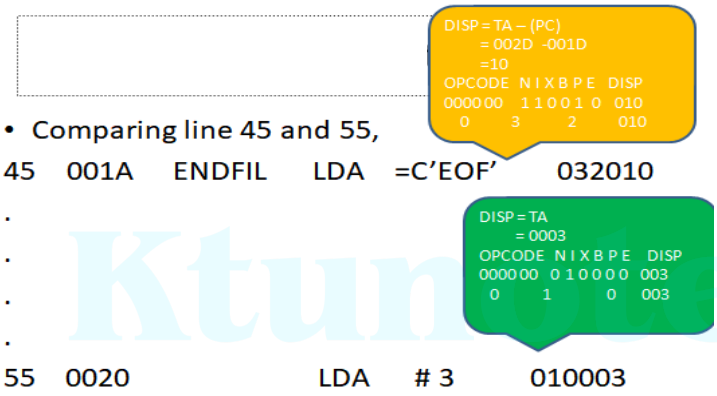

```
205     .
210     105D    WRREC   CLEAR   X               B410
212     105F            LDT     LENGTH          774000
215     1062    WLOOP   TD      =X'05'          E32011
220     1065            JEQ     WLOOP           332FFA
225     1068            LDCH    BUFFER,X        53C003
230     106B            WD      =X'05'          DF2008
235     106E            TIXR    T               B850
240     1070            JLT     WLOOP           3B2FEF
245     1073            RSUB                    4F0000
255                     END     FIRST
        1076    *       =X'05'                  05
```

Notice that the object code is the same as the previous one.

The notation used for literals varies from assembler to assembler, however most assemblers use some symbols (as we have used =) to make literal identification easier.

Difference between Literal and Immediate Operand

- Immediate addressing
  - The operand value is assembled as part of the machine instruction.
- Literal
  - The assembler generates the specified value as a constant at some other memory location.
  - The address of this generated constant is used as the target address for the machine instruction.
  - The effect of using a literal is exactly the same as if the programming had defined the constant explicitly and used the label assigned to the constant as the instruction operand.



Literal Pools

All of the literal operands used in the program are gathered together into one or more literal pools. Normally literals are placed into a pool at the end of the program. The assembly listing of the program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. Such literal pool listing is shown in the figure immediately following the END statement. Here in this case the pool consists of the single literal = X'05'.

Literal Program (2)'s Object Code

```
210     105D    WRREC   CLEAR   X               B410
212     105F            LDT     LENGTH          774000
215     1062    WLOOP   TD      =X'05'          E32011
220     1065            JEQ     WLOOP           332FFA
225     1068            LDCH    BUFFER,X        53C003
230     106B            WD      =X'05'          DF2008
235     106E            TIXR    T               B850
240     1070            JLT     WLOOP           3B2FEF
245     1073            RSUB                    4F0000
255                     END     FIRST
        1076    *       =X'05'                  05
```

Notice that the object code is the same as the previous one.

In some cases, however, it is desirable to place literals into a pool at some other location in the object program. For this an assembler directive LTORG (line 93).

LITERAL PROGRAM

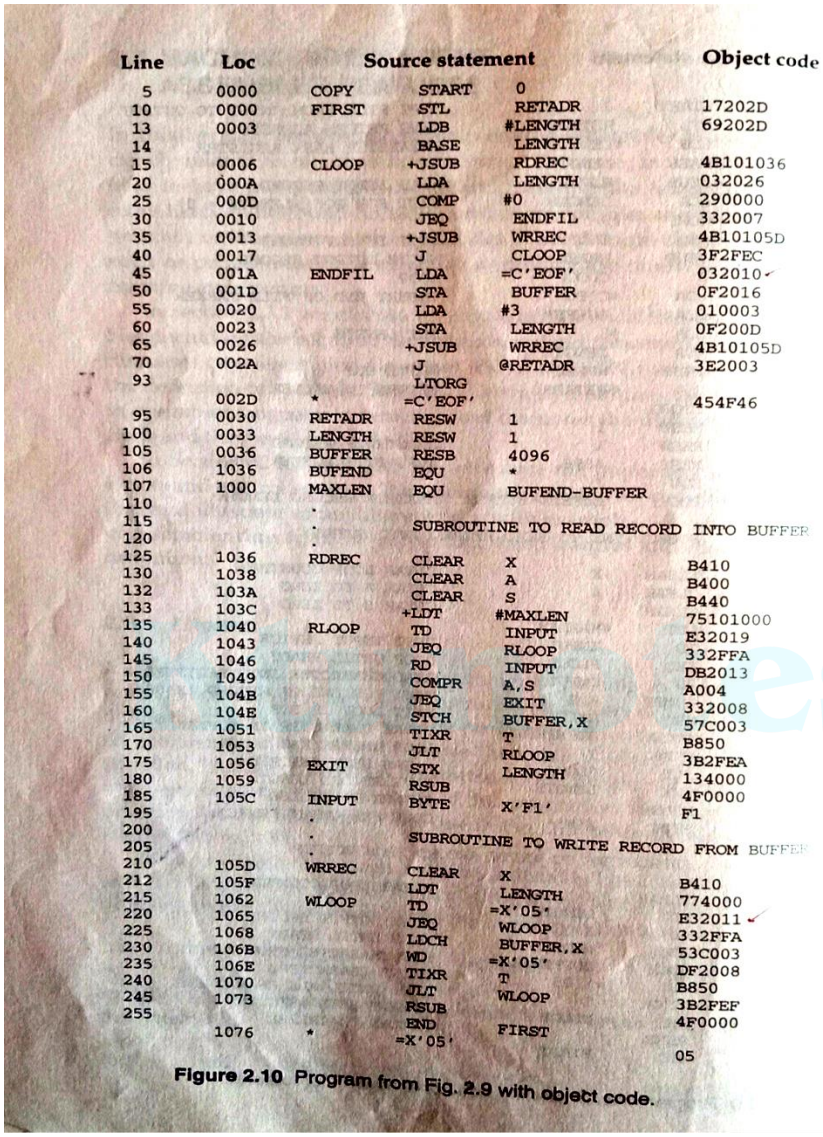
```
 5     COPY    START   0               COPY FILE FROM INPUT TO OUTPUT
10     FIRST   STL     RETADR          SAVE RETURN ADDRESS
13             LDB     #LENGTH         ESTABLISH BASE REGISTER
14             BASE    LENGTH
15     CLOOP   +JSUB   RDREC           READ INPUT RECORD
20             LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
25             COMP    #0
30             JEQ     ENDFIL          EXIT IF EOF FOUND
35             +JSUB   WRREC           WRITE OUTPUT RECORD
40             J       CLOOP           LOOP
45     ENDFIL  LDA     =C'EOF'         INSERT END OF FILE MARKER
50             STA     BUFFER
55             LDA     #3              SET LENGTH = 3
60             STA     LENGTH
65             +JSUB   WRREC           WRITE EOF
70             J       @RETADR         RETURN TO CALLER
93             LTORG
95     RETADR  RESW    1
100    LENGTH  RESW    1               LENGTH OF RECORD
105    BUFFER  RESB    4096            4096-BYTE BUFFER AREA
106    BUFEND  EQU     *
107    MAXLEN  EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
```

Use = to represent a literal

LTORG

When the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the program). This literal pool is placed in the object program at the location where the LTORG directive was encountered. Note that, the literals placed in a pool by LTORG will not be repeated in the pool at the end of the program. If we had not used the LTORG statement on line 93, the literal =C'EOF'

would be placed in the pool at the end of the program. This literal pool would begin at address 1073.This means that the literal operand would be placed too far away from the instruction referencing it to allow program – counter relative addressing.



Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
13	0003		LDB	#LENGTH	69202D
14			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	=C'EOF'	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
93			LTORG		
	002D	*	=C'EOF'		454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
106	1036	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#MAXLEN	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	=X'05'	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	=X'05'	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
255			END	FIRST	
	1076	*	=X'05'		05

Figure 2.10 Program from Fig. 2.9 with object code.

The problem is the large amount of storage reserved for BUFFER.

DISP = TA – (PC)
= 1073 -001D
= $(1056)_{16}$

DISP = TA - (B)
= 1073 - 0033
= $(1040)_{16}$

By placing the literal pool before this buffer, avoids the need to use extended format instructions when referring to literals.The need for an assembler directive LTORG arises when it is desirable

to keep the literal operand close to the instruction that uses it.Duplicate Literals – same literal in more than one place in the program and store only one copy of the specified data value. Most assemblers recognize duplicate literals. For eg: the literal =X'05' is used in the program in line 215 and 230.However, only one data area with this value is generated.Both instructions refer to the same address in the literal pool for their operand.

Literal Program (2)'s Object Code

```
210     105D    WRREC   CLEAR   X               B410
212     105F            LDT     LENGTH          774000
215     1062    WLOOP   TD      =X'05'          E32011
220     1065            JEQ     WLOOP           332FFA
225     1068            LDCH    BUFFER,X        53C003
230     106B            WD      =X'05'          DF2008
235     106E            TIXR    T               B850
240     1070            JLT     WLOOP           3B2FEF
245     1073            RSUB                    4F0000
255                     END     FIRST
        1076    *       =X'05'                  05
```

Notice that the object code is the same as the previous one.

The easiest way to recognize duplicate literals is by comparison of the character strings defining them eg: =X'05'. More intelligent way is to look at the generated value instead of the defining expression.eg: =C'EOF' and =X'454F46' would specify identical operand values.The assembler might avoid storing both literals if it recognised this equivalence. But the complexity of the assembler increases.If used character string defining a literal to recognize the duplicates we must be very careful of literals whose value depends upon their location in the program.For example, there are literals that refer to the current value of the location counter (often used by the symbol *). Such literals are sometimes useful for loading base registers.

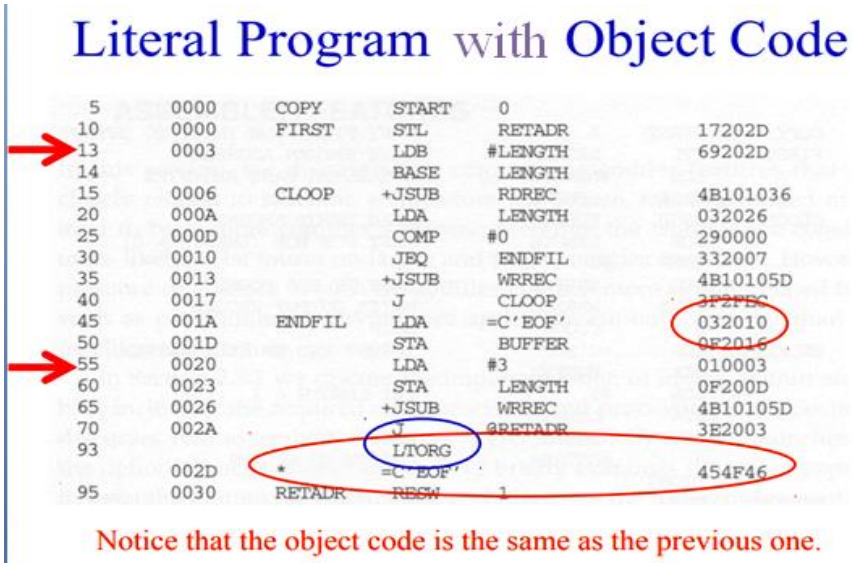
- Consider the statements:

```
BASE    *
LDB     =*
```

as the first lines of a program

This will load the beginning address of the program into register B. So that this value would be available for base relative addressing.In detecting duplicate literals such a notation causes a problem. For eg:

- If a literal =* appeared on line 13 in the program, it would specify an operand with value 0003.
- If the same literal appeared on line 55, it would specify an operand with value 0020.



In such case, the literal operands have identical names; however they have different values and both must appear in the pool. Another problem arises if a literal refers to any other item whose value changes between one point in the program and another.

How is a literal handled by an assembler?

The basic data structure needed is a literal table (LITTAB). In LITTAB for each literal contains the literal name, the operand value and length and the address assigned to the operand when it is placed in a literal pool.LITTAB is organised as hash table, using the literal name or value as the key.

- **In Pass 1**, each literal operand is recognised
  - The assembler searches LITTAB for the specified literal name (or value)
  - If the literal is already present in the LITTAB: no action is required
  - If the literal not present: the literal is added to 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 (unless such an address has already been filled in)
- As these addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal

- **In Pass 2**, the operand address for using in generating object code is obtained by searching LITTAB for each literal operand is encountered
  - The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD statements.
  - If a literal value represents an address in the program (eg: location counter value), the assembler must also generate the appropriate Modification Record

LITTAB

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

Symbol –Defining Statements

Till now we deal only with user-defined symbols appear in the assembly program 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 programmer to define symbols and specify their values.

EQU

EQU (for "equate") is the assembler directive generally used to define symbols and specify their values. General form:

symbol EQU value

This statement defines the given symbol (ie., it enters into SYMTAB) and assigns to it the value specified.The value may be given as a constant or as any expression involving constants and previously defined symbols.**One common use of EQU** is to establish symbolic names that can be used for improved readability in place of numeric values. For eg: on line 133 of the program 2.5 (*as per the text*)

\+ LDT # 4096

This statement loads the value 4096 into register T. This value represents the maximum length record we could read with subroutine RDREC.

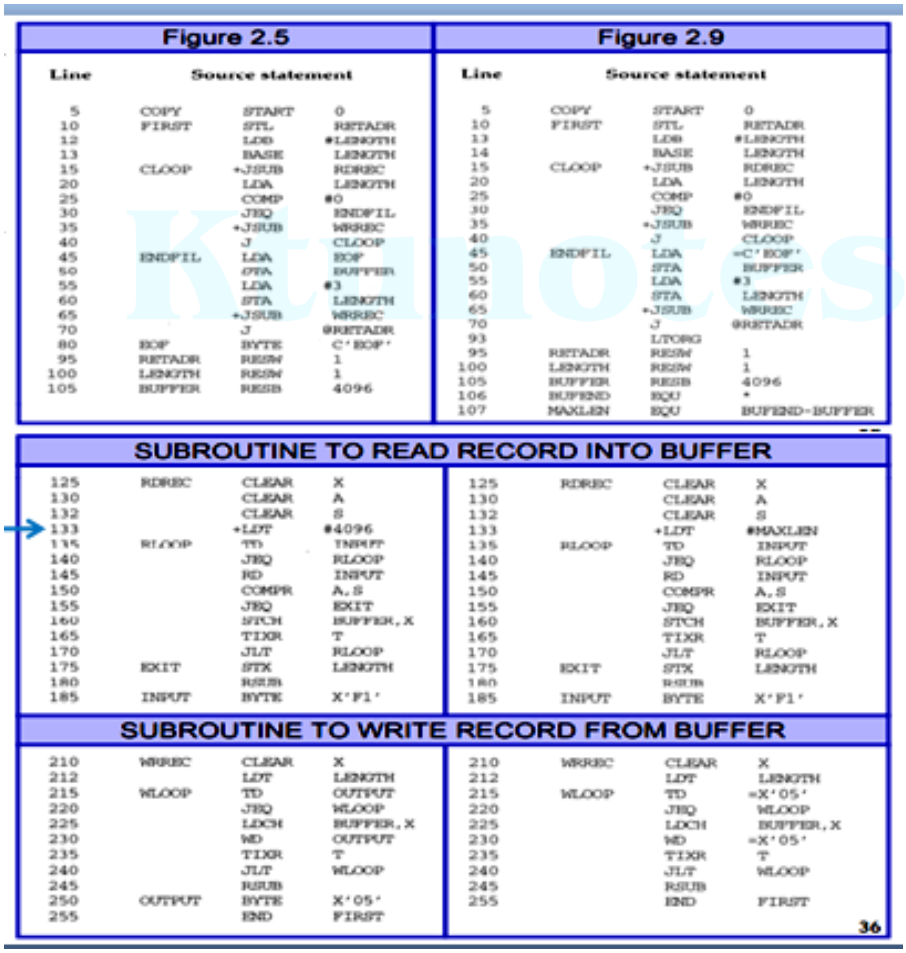


Figure 2.5

Line	Source statement		
5	COPY	START	0
10	FIRST	STL	RETADR
12		LDB	#LENGTH
13		BASE	LENGTH
15	CLOOP	+JSUB	RDREC
20		LDA	LENGTH
25		COMP	#0
30		JEQ	ENDFIL
35		+JSUB	WRREC
40		J	CLOOP
45	ENDFIL	LDA	EOF
50		STA	BUFFER
55		LDA	#3
60		STA	LENGTH
65		+JSUB	WRREC
70		J	@RETADR
80	EOF	BYTE	C'EOF'
95	RETADR	RESW	1
100	LENGTH	RESW	1
105	BUFFER	RESB	4096

Figure 2.9

Line	Source statement		
5	COPY	START	0
10	FIRST	STL	RETADR
13		LDB	#LENGTH
14		BASE	LENGTH
15	CLOOP	+JSUB	RDREC
20		LDA	LENGTH
25		COMP	#0
30		JEQ	ENDFIL
35		+JSUB	WRREC
40		J	CLOOP
45	ENDFIL	LDA	=C'EOF'
50		STA	BUFFER
55		LDA	#3
60		STA	LENGTH
65		+JSUB	WRREC
70		J	@RETADR
93		LTORG	
95	RETADR	RESW	1
100	LENGTH	RESW	1
105	BUFFER	RESB	4096
106	BUFFEND	EQU	*
107	MAXLEN	EQU	BUFFEND-BUFFER

SUBROUTINE TO READ RECORD INTO BUFFER

125	RDREC	CLEAR	X
130		CLEAR	A
132		CLEAR	S
133		+LDT	#4096
135	RLOOP	TD	INPUT
140		JEQ	RLOOP
145		RD	INPUT
150		COMPR	A,S
155		JEQ	EXIT
160		STCH	BUFFER,X
165		TIXR	T
170		JLT	RLOOP
175	EXIT	STX	LENGTH
180		RSUB	
185	INPUT	BYTE	X'F1'

125	RDREC	CLEAR	X
130		CLEAR	A
132		CLEAR	S
133		+LDT	#MAXLEN
135	RLOOP	TD	INPUT
140		JEQ	RLOOP
145		RD	INPUT
150		COMPR	A,S
155		JEQ	EXIT
160		STCH	BUFFER,X
165		TIXR	T
170		JLT	RLOOP
175	EXIT	STX	LENGTH
180		RSUB	
185	INPUT	BYTE	X'F1'

SUBROUTINE TO WRITE RECORD FROM BUFFER

210	WRREC	CLEAR	X
212		LDT	LENGTH
215	WLOOP	TD	OUTPUT
220		JEQ	WLOOP
225		LDCH	BUFFER,X
230		WD	OUTPUT
235		TIXR	T
240		JLT	WLOOP
245		RSUB	
250	OUTPUT	BYTE	X'05'
255		END	FIRST

210	WRREC	CLEAR	X
212		LDT	LENGTH
215	WLOOP	TD	=X'05'
220		JEQ	WLOOP
225		LDCH	BUFFER,X
230		WD	=X'05'
235		TIXR	T
240		JLT	WLOOP
245		RSUB	
255		END	FIRST

36

If we include a statement

MAXLEN EQU 4096

in the program, then you can include the line 133 as in program 2.9 *(as per text book)*

```
+  LDT     # MAXLEN
```

When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB (with value 4096). During assembly of LDT instruction, the assembler searches SYMTAB for the symbol MAXLEN, using its value as operand in the instruction.**Another common use of EQU** is defining mnemonic names for registers.Assembler recognizes standard mnemonics for registers – A, X, L, etc.Assembler expects register numbers instead of names in an instruction like RMO.This instructs the programmer to write RMO 0,1 instead of RMO A, X

In such case the programmer could include a sequence of EQU statements like:

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

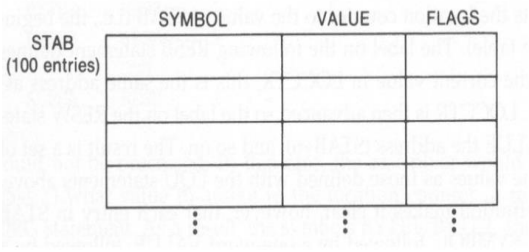
These statements cause the symbols A, X, L... To be entered into SYMTAB with their corresponding values 0,1,2 ...The programmer can establish and use names that reflect the logical function of the registers in the program.

ORG

ORG (for "origin") is an assembler directive used to indirectly assign values to symbols.It is of the form:

```
ORG     value
```

Where value is a constant or an expression involving constants and previously defined symbols.When this statement is encountered during the assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value.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.We know that location counter is used to control assignment of storage in the object program. In most cases, if we alter the location counter value it will result in incorrect assembly. However, the ORG is useful in label definition. Suppose that we are defining a symbol table with the following structure:



- SYMBOL: 6 bytes
- VALUE: 3 bytes (one word)
- FLAGS: 2 bytes

- In this table,
  - the SYMBOL field contains a 6-byte user-defined symbol
  - VALUE is a one-word representation of the value assigned to the symbol
  - FLAGS is a 2-byte field that specifies symbol type and other information

- Consider the statements:

6 + 3 +2 =11 *100 = 1100

```
STAB    RESB    1100
SYMBOL  EQU     STAB
VALUE   EQU     STAB+6
FLAGS   EQU     STAB+9
```

Show offsets, less readable

- First line → we could reserve space for the table STAB

We want to refer to entries in the table using indexed addressing (placing in the index register the offset of the desired entry from beginning of the table). We want to refer the fields SYMBOL, VALUE, and FLAGS individually, so we define these labels.

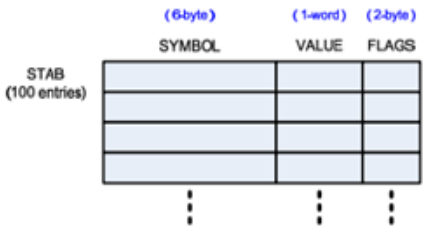
However, this method of definition simply defines labels. It does not make clear the structure of the table

- Use **EQU** assembler directives

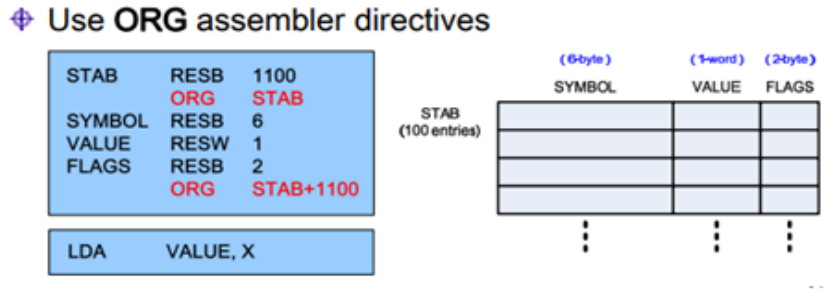
```
STAB    RESB    1100

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

LDA     VALUE, X
```



LDA instruction fetches the VALUE field from the table entry indicated by the contents of register X. Same Symbol definition using ORG directive.



The first ORG statement resets the location counter value to the value of STAB (ie. The beginning address of the table.)

```
STAB        RESB  1100
            ORG   STAB      ← Set LOCCTR to STAB
SYMBOL      RESB  6
VALUE       RESW  1         ← Size of each field
FLAGS       RESB  2
            ORG   STAB+1100 ← Restore LOCCTR
```

The label on the following RESB statement defines SYMBOL to have the current value in the LOCCTR, this is the same address assigned to SYMTAB. LOCCTR is then advanced so the label on the RESW statement assigns to VALUE the address (STAB +6) and so on.The result is a set of labels with the same values as those defined with EQU statements above.So, using ORG the definition becomes clear that each entry in STAB consists of a 6-byte SYMBOL, followed by a one-word VALUE, followed by a 2-byte FLAGS.Last ORG statement is very important because:

- it sets the LOCCTR back to its previous value – the address of the next unassigned byte of memory after the table STAB.
- any labels on subsequent statements, which do not represent the part of STAB, are assigned proper addresses

In some assemblers, the previous value of LOCCTR is automatically remembered, so simply can write

ORG

With no value specified to return to the normal use of LOCCTR.

Restrictions for EQU and ORG

There are restrictions that are common to all symbol defining assembler directives. In case of EQU, all symbols used on right hand side of the statement – ie., all terms used to specify the value of the new symbol – must have been defined previously in the program.

- Consider the sequence,

ALLOWED

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

- Consider another sequence,

NOT ALLOWED

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

The reason for this is the symbol definition process. Here in the second example, BETA cannot be assigned a value when it is encountered during Pass1 of the assembly, because ALPHA does not yet have a value. However, two-pass assembler design requires that all symbols be defined during Pass 1. In case of ORG, all symbols used to specify the new location counter value must have been previously defined. Consider the sequence:


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

NOT ALLOWED

The sequence cannot be processed. In this case, the assembler would not know (during Pass1)what value to assign to the location counter in response to the first ORG statement. As a result, the symbols BYTE 1, BYTE 2, and BYTE 3 could not be assigned addresses during Pass 1.

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

ALLOWED

This restriction is a result of particular way in which we defined the two passes of our assembler.The forward reference problem cannot be resolved by an ordinary two pass assembler regardless of how the work is divided between the passes. Consider the sequence:

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

It cannot be resolved by an ordinary two-pass assembler.We need complex assembler structures.

Expressions

The assembly language statements have used single terms (labels, literals, etc.) as instruction operands. Most assemblers allow the use of expressions as instruction operands.Each such expression must be evaluated by the assembler to produce a single operand address or value.

- Assemblers generally allow arithmetic expressions formed using the operators +, -, *, /
- Division is usually defined to produce an integer result.
- Individual terms in the expression may be:
    - Constants
    - User-defined symbols
    - Special terms
- Most commonly the special term is the current value of the location counter (*). This * represents the value of the next unassigned memory location.

```
106     BUFEND      EQU     *
```

This statement gives BUFFEND a value that is the address of the next after the buffer area.

```
        002D    *         =C'EOF'                    454F46
 95     0030    RETADR    RESW    1
100     0033    LENGTH    RESW    1
105     0036    BUFFER    RESB    4096
106     1036    BUFEND    EQU     *
107     1000    MAXLEN    EQU     BUFEND-BUFFER
110             .
```

- We discussed the problem of program relocation:

- Some values in the object program are relative to the beginning of the program,Some are absolute (independent of program location).

Similarly, the values of the terms and expressions are either relative or absolute.Constant is an absolute term.Labels on the instructions and data areas, references to the location counter value are relative terms. A symbol whose value is given by EQU (or any other similar directive) may be either an absolute term or a relative term depending upon the expression used to define its value.

- Expressions can be:
  - Absolute expression
  - Relative expression

Depending upon the type of value they produce.

- **Absolute expression**: Expression contain only absolute terms.
- It may contain relative terms provided that relative terms occur in pairs and the terms in such pair have opposite signs


```
Only absolute terms
    MAXLEN    EQU    1000
Relative terms in pairs with opposite signs for each
pair
    MAXLEN    EQU    BUFEND-BUFFER
```

- **Relative Expression**: All of the relative terms except one can be paired and the remaining unpaired relative terms must have a positive sign. No relative terms can enter into a multiplication or division operation no matter in absolute or relative expression.

Errors: **(represent neither absolute values nor locations within the program)**

```
BUFEND+BUFFER    // not opposite terms
100-BUFFER       // not in pair
3*BUFFER         // multiplication
```

Assemblers should determine the type of an expression. Keep track of the types of all symbols defined in the program in the symbol table. Generate Modification records in the object program for relative values.We need a “flag” in the SYMTAB for indication.

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

Program Blocks

We have seen so far the program being assembled was treated as a unit. Even the source program logically contains subroutines, data areas etc.The assembler handles it as one entity, resulting in a single block of object code.Within this object program, the generated machine instructions and data appeared in the same order as they were written in the source program. Many assemblers provide features that allow more flexible handling of the source and object programs:

- Allow the generated machine instructions and data to appear in the object program in a different order from the corresponding source statements
- Allowed the creation of several independent parts of the object program. These parts maintain their identity and are handled separately by the loader

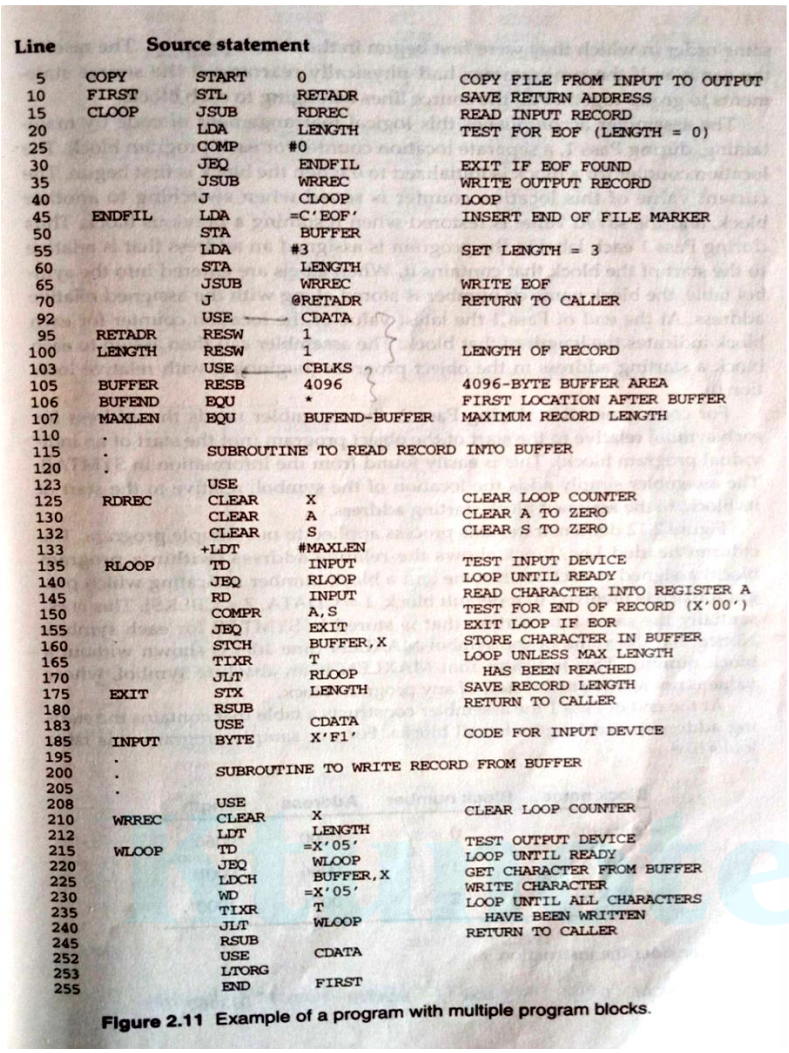
Program Block: segments of code that are rearranged within a single object program unit

Control Section: segments that are translated into independent object program units

Consider the program in the figure which is written using program blocks.

In this case there are three blocks:

1. (unnamed) program block contains the executable instructions of the program
2. (named CDATA) contains all data areas that are a few words or less in length
3. (named CBLKS) contains all data areas that consist of larger blocks of memory



```
Line    Source statement
  5     COPY      START   0               COPY FILE FROM INPUT TO OUTPUT
 10     FIRST     STL     RETADR          SAVE RETURN ADDRESS
 15     CLOOP     JSUB    RDREC           READ INPUT RECORD
 20               LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
 25               COMP    #0
 30               JEQ     ENDFIL          EXIT IF EOF FOUND
 35               JSUB    WRREC           WRITE OUTPUT RECORD
 40               J       CLOOP           LOOP
 45     ENDFIL    LDA     =C'EOF'         INSERT END OF FILE MARKER
 50               STA     BUFFER
 55               LDA     #3              SET LENGTH = 3
 60               STA     LENGTH
 65               JSUB    WRREC           WRITE EOF
 70               J       @RETADR         RETURN TO CALLER
 92               USE     CDATA
 95     RETADR    RESW    1
100     LENGTH    RESW    1               LENGTH OF RECORD
103               USE     CBLKS
105     BUFFER    RESB    4096            4096-BYTE BUFFER AREA
106     BUFEND    EQU     *               FIRST LOCATION AFTER BUFFER
107     MAXLEN    EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
110     .
115     .         SUBROUTINE TO READ RECORD INTO BUFFER
120     .
123               USE
125     RDREC     CLEAR   X               CLEAR LOOP COUNTER
130               CLEAR   A               CLEAR A TO ZERO
132               CLEAR   S               CLEAR S TO ZERO
133               +LDT    #MAXLEN
135     RLOOP     TD      INPUT           TEST INPUT DEVICE
140               JEQ     RLOOP           LOOP UNTIL READY
145               RD      INPUT           READ CHARACTER INTO REGISTER A
150               COMPR   A,S             TEST FOR END OF RECORD (X'00')
155               JEQ     EXIT            EXIT LOOP IF EOR
160               STCH    BUFFER,X        STORE CHARACTER IN BUFFER
165               TIXR    T               LOOP UNLESS MAX LENGTH
170               JLT     RLOOP             HAS BEEN REACHED
175     EXIT      STX     LENGTH          SAVE RECORD LENGTH
180               RSUB                    RETURN TO CALLER
183               USE     CDATA
185     INPUT     BYTE    X'F1'           CODE FOR INPUT DEVICE
195     .
200     .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
208               USE
210     WRREC     CLEAR   X               CLEAR LOOP COUNTER
212               LDT     LENGTH
215     WLOOP     TD      =X'05'          TEST OUTPUT DEVICE
220               JEQ     WLOOP           LOOP UNTIL READY
225               LDCH    BUFFER,X        GET CHARACTER FROM BUFFER
230               WD      =X'05'          WRITE CHARACTER
235               TIXR    T               LOOP UNTIL ALL CHARACTERS
240               JLT     WLOOP             HAVE BEEN WRITTEN
245               RSUB                    RETURN TO CALLER
252               USE     CDATA
253               LTORG
255               END     FIRST
```

Figure 2.11 Example of a program with multiple program blocks.

The assembler directive **USE** indicates which portions of source program belong to 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. The USE statement in line 92 signals the beginning of the block named CDATA. Source statements are associated with this block until the USE statement on line 103, which begins the block named CBLKS. The USE statement may also indicate the continuation of a previously begun block.Thus the line 123 resumes the default block, and the statement on line 183 resumes the block named CDATA.

- Three blocks
  - First: unnamed, i.e., default block
    - Line 5~ Line 70 + Line 123 ~ Line 180 + Line 208 ~ Line 245
  - Second: CDATA
    - Line 92 ~ Line 100 + Line 183 ~ Line 185 + Line 252 ~ Line 255
  - Third: CBLKS
    - Line 103 ~ Line 107

Each program block may actually contain several separate segments of the source program. Assembler rearrange these segments to gather together the pieces of each block. These blocks will then be assigned addresses in the object program, with the blocks appearing in the same order in which they were first begun in the source program.

How assembler handles Program Blocks?

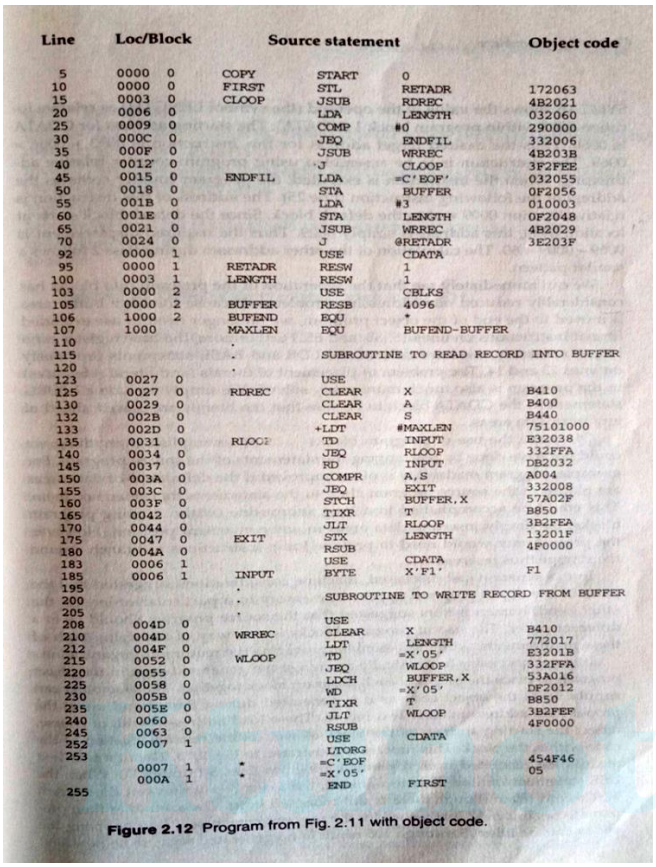
During Pass 1:

- Assembler accomplishes the logical rearrangement of code by maintaining a separate location counter for each program block.
- When the block begins first, the location counter for a bloc is initialized to 0
- When switching to another block, the current value of the location counter is saved and when resuming a previous block, the saved value is restored
- Each label in the program is assigned an address that is relative to the start of the block that contains it.
- When labels are entered into the symbol table, the block name or number is stored along with the assigned relative address.
- At the end of Pass1 the latest value of location counter for each block indicates the length of that block
- The assembler can then assign to each block a starting address in the object program (beginning with relative location 0)

During Pass 2:

- For code generation, the assembler needs the address for each symbol relative to the start of the object program (not the start of an individual program block) which is easily found from the information in SYMTAB.

- The assembler simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address



Line	Loc/Block		Source statement			Object code
5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	
110			.			
115			.	SUBROUTINE TO READ RECORD INTO BUFFER		
120			.			
123	0027	0		USE		
125	0027	0	RDREC	CLEAR	X	B410
130	0029	0		CLEAR	A	B400
132	002B	0		CLEAR	S	B440
133	002D	0		+LDT	#MAXLEN	75101000
135	0031	0	RLOOP	TD	INPUT	E32038
140	0034	0		JEQ	RLOOP	332FFA
145	0037	0		RD	INPUT	DB2032
150	003A	0		COMPR	A,S	A004
155	003C	0		JEQ	EXIT	332008
160	003F	0		STCH	BUFFER,X	57A02F
165	0042	0		TIXR	T	B850
170	0044	0		JLT	RLOOP	3B2FEA
175	0047	0	EXIT	STX	LENGTH	13201F
180	004A	0		RSUB		4F0000
183	0006	1		USE	CDATA	
185	0006	1	INPUT	BYTE	X'F1'	F1
195			.			
200			.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205			.			
208	004D	0		USE		
210	004D	0	WRREC	CLEAR	X	B410
212	004F	0		LDT	LENGTH	772017
215	0052	0	WLOOP	TD	=X'05'	E3201B
220	0055	0		JEQ	WLOOP	332FFA
225	0058	0		LDCH	BUFFER,X	53A016
230	005B	0		WD	=X'05'	DF2012
235	005E	0		TIXR	T	B850
240	0060	0		JLT	WLOOP	3B2FEF
245	0063	0		RSUB		4F0000
252	0007	1		USE	CDATA	
253				LTORG		
	0007	1	*	=C'EOF'		454F46
	000A	1	*	=X'05'		05
255				END	FIRST	

Figure 2.12 Program from Fig. 2.11 with object code.

Consider the program: The column headed Loc/Block shows the relative address within a program block assigned to each source line and a block number indicating which program block is involved. This information gets stored in the SYMTAB for each symbol. The value of MAXLEN in line 107, shown without a block number indicates that it is an absolute symbol, whose value is not relative to the start of the program. At the end of Pass1 assembler constructs a table that contains the starting addresses and lengths for all blocks.

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

```
20  0006  0     LDA    LENGTH     032060
```

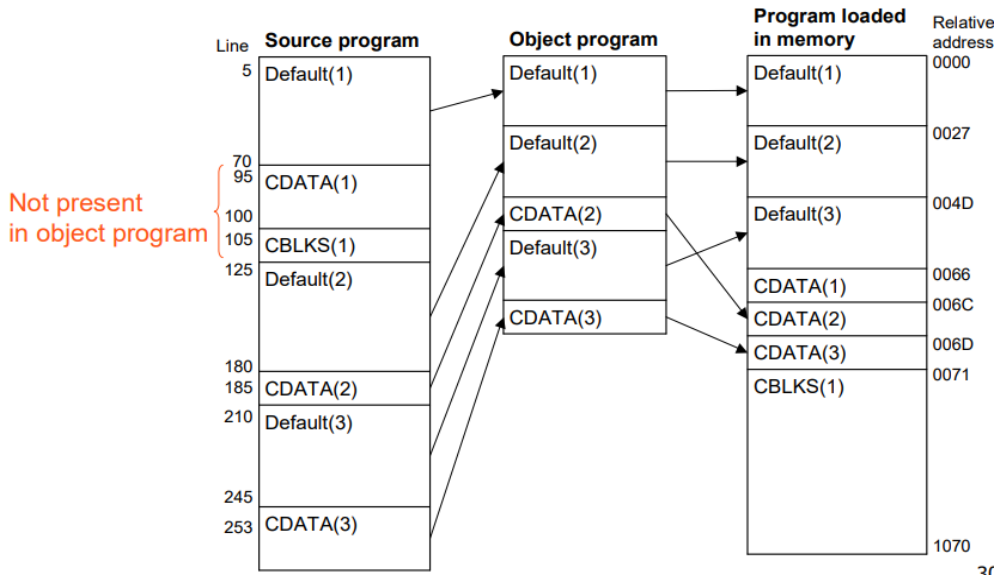
- The value of the operand (LENGTH)
  - Address 0003 relative to Block 1 (CDATA)
  - Address 0003+0066=0069 relative to program
  - When this instruction is executed
    - PC = 0009
  - disp = 0069 – 0009 = 0060
  -

```
    op       nixbpe      disp
  000000     110010      060    => 032060
```

Separation of programs into blocks has considerably reduced the addressing problem. Because the large buffer area is moved to the end of the object program. No longer needed to use extended format instruction on line number 15,35,65. The base register is no longer necessary. The problem of placing literal is also solved: by placing LTORG statement in CDATA block to be sure that literals are placed ahead of any large data area. It is not necessary to physically rearrange the generated code in the object program to place the pieces of each program block together. The assembler simply writes the object code as it is generated during Pass 2and insert the proper loader address in each Text record.These load addresses will reflect the starting address of the block as well as the relative location of the code within the block. For example: in figure: The first two Text records are generated from line 5~70. When the USE statement is recognized. Assembler writes out the current Text record, even if there still room left in it. Begin a new Text record for the new program block.


```
HCOPY  000000001071
T0000001E1720634B2021032060290000332006 4B203B3F2FEE032055 0F2056010003
T00001E090F20484B20293E203F                                          Default(1)
T0000271DB410B400B44075101000E32038332FFADB2032A00433200857A02FB850
T000044093B2FEA13201F4F0000                                          Default(2)
T00006C01F1                                                          CDATA(2)
T00004D19B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000          Default(3)
T00006D04454F4605                                                    CDATA(3)
E000000
```

Line 95~105 no generated code, so no text records are created. Next Text record lines 125~180 → statements that belong to the next program block generate the object code. Fifth text record contains single byte of data from line 185. Sixth record resumes the default program block. Does not matter that the text records of the object program are not in sequence by address. Loader will load the object code from each record at the indicated addresses.



Control Sections and Program Linking

A control section is a part of the program that maintains its identity after the assembly. It can be loaded and relocated independently of the other control sections. Different control sectio... are most often used for subroutines or other logical subdivisions of a program. The programmer can assemble, load, and manipulate each of these control sections separately. The resulting flexibility is the major benefit of using control sections. When control sections form logically related parts of the program, there should be some means for linking control sections together. For example, the instructions in one control section might need to refer to instructions or data located in another section. The control sections are independently loaded and located, so the assembler is unable to process these reference in a usual way. The assembler has no idea where any other control section will be located at execution time. The references between the control sections are called external references. The assembler generates information for each external references that will allow the loader to perform the required linking. Consider an example program in the next slide that might be written using multiple control sections. In this case there are three control sections:

- One for the main program
- Second for the subroutine RDREC
- Third for the subroutine WRREC

The start statement identifies the beginning of the assembly and gives a name COPY the first control section. The first section continues until the CSECT statement on line 109. **CSECT** is an assembler directive signals the start of a new control section named RDREC.

```
secname        CSECT
```

Similarly, the CSECT statement on the line 193 begins the control section named WRREC. The assembler establishes a separate location counter for each control section as done for program blocks.

Implicitly defined as an external symbol

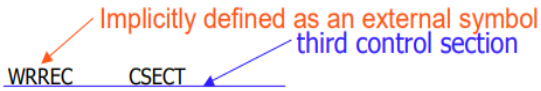
first control section

```
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
```




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

Assembler handles program blocs and control sections in a different way:

Difference:

It is not necessary for all control sections in a program to be assembled at the same time. Symbols that are defined in one control section may not be used directly by another control section. They must be identified as external references for the loader to handle. There are two assembler directives to identify such references:

- External definition
  EXTDEF name [, name]
  - EXTDEF names symbols that are defined in this control section and may be used by other sections
  - Ex: EXTDEF BUFFER, BUFEND, LENGTH
- External reference
  EXTREF name [,name]
  - EXTREF names symbols that are used in this control section and are defined elsewhere
  - Ex: EXTREF RDREC, WRREC

Control section names in this case COPY, RDREC,WRREC do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols. The order in which symbols are listed in EXTDEF and EXTREF statements is not significant. Control section names in this case COPY, RDREC,WRREC do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols. The order in which symbols are listed in EXTDEF and EXTREF statements is not significant.

```
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 1

```
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
```

How external references handled by the assembler?

CASE 1: Consider the instruction:

```
15      0003        CLOOP       +JSUB     RDREC         4B100000
```

The operand RDREC is named EXTREF statement for the control section, so this is an external reference. The assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address for this instruction. Instead the assembler inserts an address of zero and passes information to the loader, which will cause the proper address to be inserted at the load time. The address of RDREC will have no predictable relationship to anything in this control section; therefore, relative addressing is not possible. Thus an extended format instruction must be used to provide room for the actual address to be inserted. This is true of any instruction whose operand involves an external reference.

CASE 2: Consider the instruction:

```
190  0028    MAXLEN    WORD    BUFEND-BUFFER    000000
```

Here the value of data word to be generated is specified by an expression involving two external references: BUFFEND and BUFFER. The assembler stores this value as zero. When the program is loaded, the loader will add to this data area the address of BUFFEND and subtract from it the address of BUFFER, which results in the desired value. Note the difference between handling of expression on line 190 and the similar expression on line 107. The symbols BUFEND and BUFFER are defined in the same control section with EQU statement on line 107. Thus the value of the expression can be calculated immediately, by the assembler. This could not be done for line 190, BUFFEND and BUFFER are defined in another control section, so their values are unknown at assembly time.

CASE 3: Consider the instruction:

```
160   0017          +STCH   BUFFER,X        57900000
```

This makes an external reference to BUFFER. The instruction is assembled using extended format with an address of zero. The x bit is set to 1 to indicate indexed addressing as specified by the instruction. The assembler must remember via entries in SYMTAB, that in which control section a symbol is defined. Any attempt to refer to a symbol is identified using EXTREF as an external reference. The assembler must allow the same symbol to be used in different control sections. There is a conflicting definitions of MAXLEN on line 107 and 190 should cause no

problem. A reference to MAXLEN in the control section COPY would use the definition on line 107, whereas a reference to MAXLEN in RDREC would use the definition on line 190. It is clear that assembler leaves room in the object code for the values of external symbols. The assembler must also include information in the object program that will cause the loader to insert the proper values where they are required. For that two new record types in the object program:

- Define Record
- Refer Record

Define Record: gives information about external symbols that are defined in this control section ie, the symbols named by EXTDEF.

Refer Record: lists symbols that are used as external references by the control section-ie, symbols named EXTREF

- 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 (hexadeccimal)
  - 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

The other information needed for program linking is added to the Modification record type.

Modification record (revised)

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified, relative to the beginning of the control section (hex)
- Col. 8-9 Length of the field to be modified, in half-bytes (hex)
- Col. 10 Modification flag (+ or -)

- Col.11-16 External symbol whose value is to be added to or subtracted from the Indicated field.

For modification record the first three items are the same as we studied earlier. The two new items specify the modification to be performed:

- Adding or subtracting the values of some external symbol
- The symbol used for modification may be defined either in thus control section or in another one

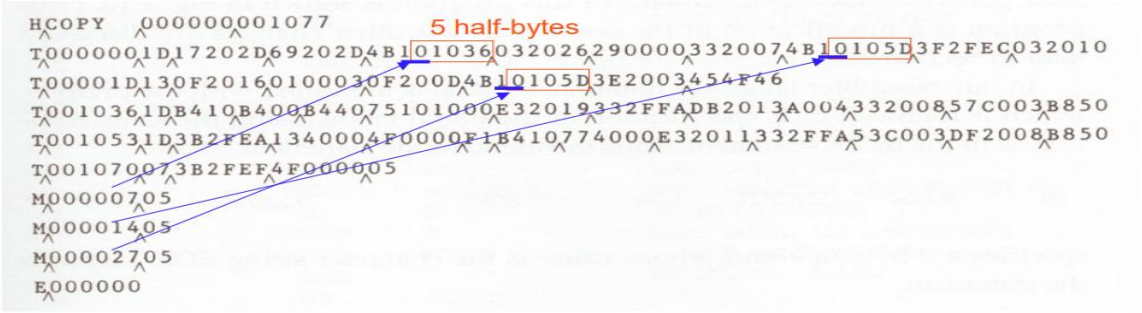
Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	

109	0000	RDREC	CSECT		
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
122			EXTREF	BUFFER,LENGTH,BUFEND	
125	0000		CLEAR	X	B410
130	0002		CLEAR	A	B400
132	0004		CLEAR	S	B440
133	0006		LDT	MAXLEN	77201F
135	0009	RLOOP	TD	INPUT	E3201B
140	000C		JEQ	RLOOP	332FFA
145	000F		RD	INPUT	DB2015
150	0012		COMPR	A,S	A004
155	0014		JEQ	EXIT	332009
160	0017		+STCH	BUFFER,X	57900000
165	001B		TIXR	T	B850
170	001D		JLT	RLOOP	3B2FE9
175	0020	EXIT	+STX	LENGTH	13100000
180	0024		RSUB		4F0000
185	0027	INPUT	BYTE	X'F1'	F1
190	0028	MAXLEN	WORD	BUFEND-BUFFER	000000

The figure shows the object program corresponding to the source program written using control sections. Note that there is separate set of object program records (from Header through End) for each control section. The records for each control section are exactly the same as they would be if the sections are assembled separately. The Define and Refer records for each control section include the symbols named in EXTDEF and EXTREF statements. In case of Define, the record also indicates the relative address for each external symbol within the control section. For EXTREF symbols, no address information is available. These symbols are simply named in the Refer record.

Modification record :

- M00000405+RDREC
- M00000705+COPY
- M00001405+COPY
- M00002705+COPY



The existence of multiple control sections that can be relocated independently of one another makes the handling expressions slightly more complicated. If the two terms represent relative locations in the same control section, their difference is an absolute value regardless of whether the control section is loaded. On the other hand, if they are in different control sections, their difference has a value that is unpredictable.

Assembler Design Options

- One – pass Assembler
- Multi - pass Assembler

One – Pass Assembler

It is used when it is necessary or desirable to avoid a second pass over the source program. But the main problem in trying to assemble a program in one pass involves forward reference. Operand parts of the instructions are often symbols that have not yet been defined in the source program.

- Main problem - Forward references
  - Data items
  - Labels on instructions
- Solution
  - Require that all areas be defined before they are referenced.
  - It is possible, although inconvenient, to do so for data items.
  - Forward jump to instruction items cannot be easily eliminated.
    - Insert (label, *address_to_be_modified*) to SYMTAB
    - Usually, *address_to_be_modified* is stored in a linked-list

- There are two main types of one pass assembler

1. Produces object code directly in memory for immediate execution

2. Produces the object program for later execution

Consider the program:

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

```
110                 .
115                 .       SUBROUTINE TO READ RECORD INTO BUFFE
120                 .
121     2039    INPUT   BYTE    X'F1'           F1
122     203A    MAXLEN  WORD    4096            001000
124                 .
125     203D    RDREC   LDX     ZERO            041006
130     2040            LDA     ZERO            001006
135     2043    RLOOP   TD      INPUT           E02039
140     2046            JEQ     RLOOP           302043
145     2049            RD      INPUT           D82039
150     204C            COMP    ZERO            281006
155     204F            JEQ     EXIT            30205B
160     2052            STCH    BUFFER,X        54900F
165     2055            TIX     MAXLEN          2C203A
170     2058            JLT     RLOOP           382043
175     205B    EXIT    STX     LENGTH          10100C
180     205E            RSUB                    4C0000
195                 .

195                 .
200                 .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
206     2061    OUTPUT  BYTE    X'05'           05
207                 .
210     2062    WRREC   LDX     ZERO            041006
215     2065    WLOOP   TD      OUTPUT          E02061
220     2068            JEQ     WLOOP           302065
225     206B            LDCH    BUFFER,X        50900F
230     206E            WD      OUTPUT          DC2061
235     2071            TIX     LENGTH          2C100C
240     2074            JLT     WLOOP           382065
245     2077            RSUB                    4C0000
255                     END     FIRST
```

Figure 2.18 Sample program for a one-pass assembler.

Load- and- go Assembler

One pass assembler that generate their object code in memory for execution. No object program is written out, and no loader is needed. This kind of load and-go assemblers are useful in a system.

We first discuss one-pass assemblers that generate their object code in memory for immediate execution. No object program is written out, and no loader is needed. This kind of *load-and-go* assembler is useful in a system that is oriented toward program development and testing. A university computing system for student use is a typical example of such an environment. In such a system, a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration. A load-and-go assembler avoids the overhead of writing the object program out and reading it back in. This can be accomplished with either a one- or a two-pass assembler. However, a one-pass assembler also avoids the overhead of an additional pass over the source program.

Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward references becomes less difficult. The assembler simply generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled. The symbol used as an operand is entered into the symbol table (unless such an entry is already present). This entry is flagged to indicate that the symbol is undefined. The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry. When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists), and the proper address is inserted into any instructions previously generated.

An example should help to make this process clear. Figure 2.19(a) shows the object code and symbol table entries as they would be after scanning line 40 of the program in Fig. 2.18. The first forward reference occurred on line 15. Since the operand (RDREC) was not yet defined, the instruction was assembled with no value assigned as the operand address (denoted in the figure by ----). RDREC was then entered into SYMTAB as an undefined symbol (indicated by *); the address of the operand field of the instruction (2013) was inserted in a list associated with RDREC. A similar process was followed with the instructions on lines 30 and 35.

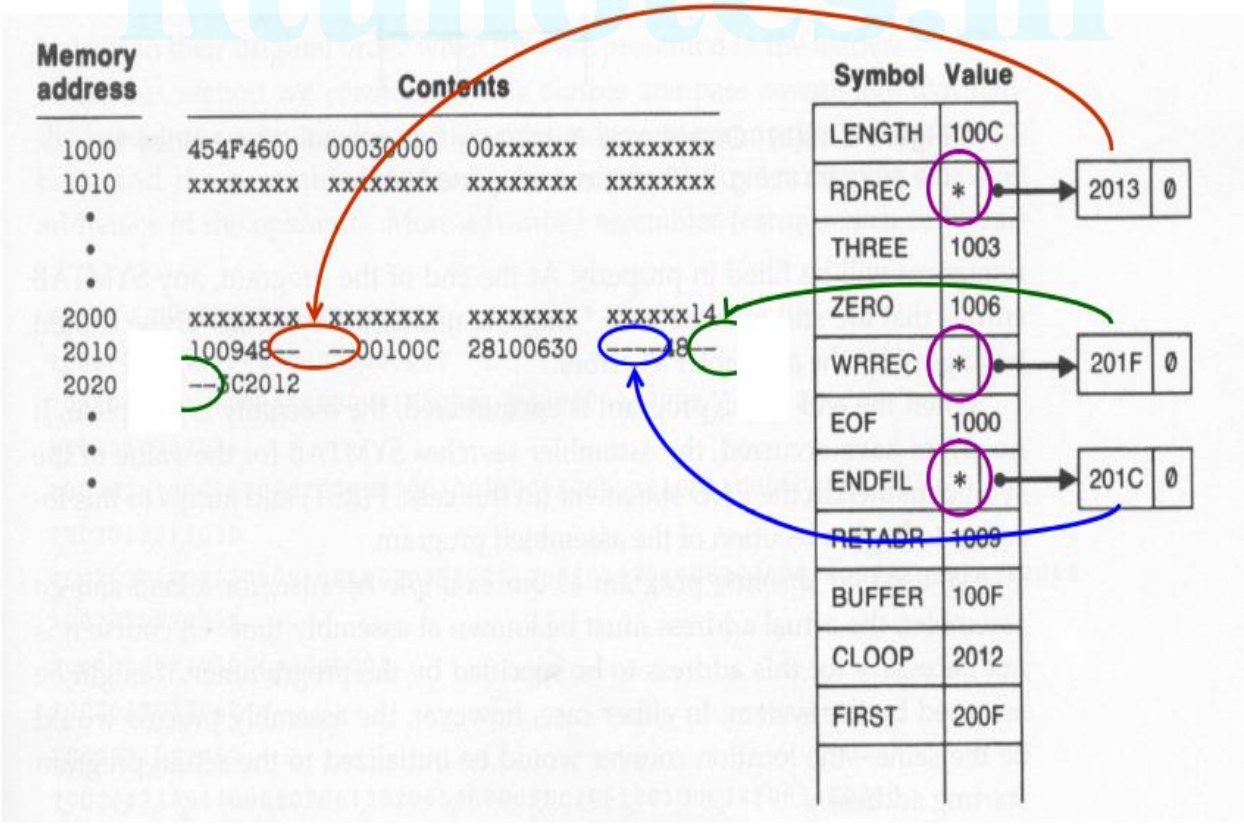
Now consider Fig. 2.19(b), which corresponds to the situation after scanning line 160. Some of the forward references have been resolved by this time, while others have been added. When the symbol ENDFIL was defined (line 45), the assembler placed its value in the SYMTAB entry; it then inserted this value into the instruction operand field (at address 201C) as directed by the forward reference list. From this point on, any references to ENDFIL would not be forward references, and would not be entered into a list. Similarly, the definition of RDREC (line 125) resulted in the filling in of the operand address at location 2013. Meanwhile, two new forward references have been added: to WRREC (line 65) and EXIT (line 155). You should continue tracing through this process to the end of the program to show yourself that all of the forward

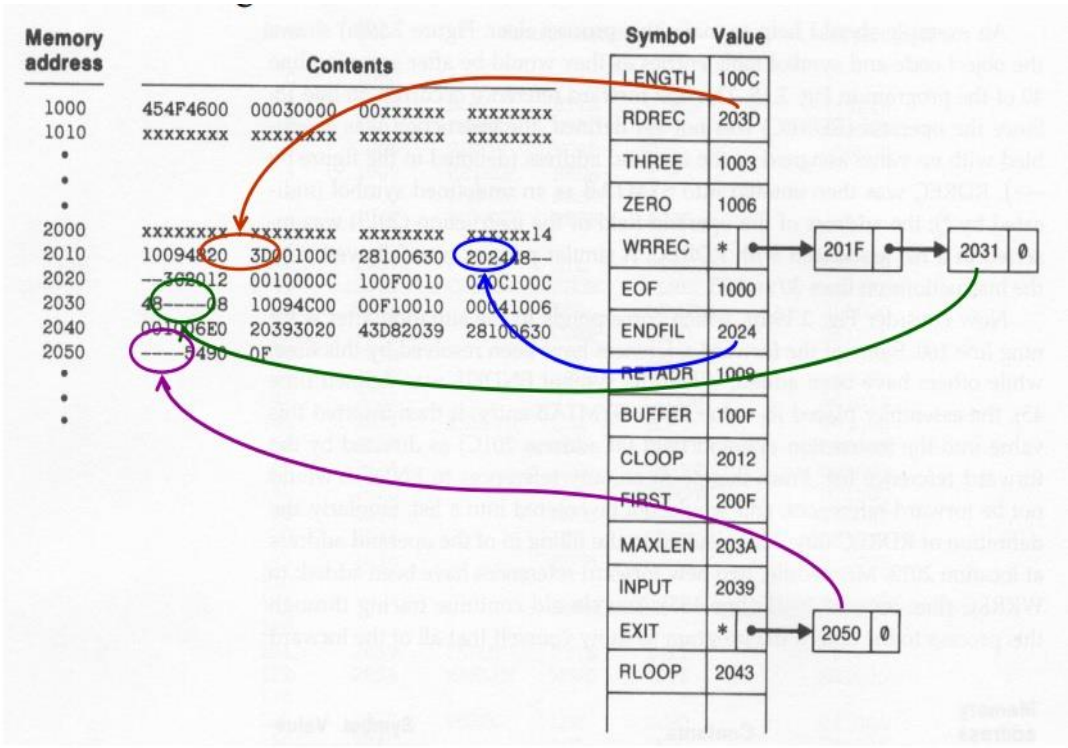
references will be filled in properly. At the end of the program, any SYMTAB entries that are still marked with * indicate undefined symbols. These should be flagged by the assembler as errors.

When the end of the program is encountered, the assembly is complete. If no errors have occurred, the assembler searches SYMTAB for the value of the symbol named in the END statement (in this case, FIRST) and jumps to this location to begin execution of the assembled program.

We used an absolute program as our example because, for a load-and-go assembler, the actual address must be known at assembly time. Of course it is not necessary for this address to be specified by the programmer; it might be assigned by the system. In either case, however, the assembly process would be the same—the location counter would be initialized to the actual program starting address.

One-pass assemblers that produce object programs as output are often used on systems where external working-storage devices (for the intermediate file between the two passes) are not available. Such assemblers may also be





useful when the external storage is slow or is inconvenient to use for some other reason. One-pass assemblers that produce object programs follow a slightly different procedure from that previously described. Forward references are entered into lists as before. Now, however, when the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. In general, they will already have been written out as part of a Text record in the object program. In this case the assembler must generate another Text record with the correct operand address. When the program is loaded, this address will be inserted into the instruction by the action of the loader.

Figure 2.20 illustrates this process. The second Text record contains the object code generated from lines 10 through 40 in Fig. 2.18. The operand addresses for the instructions on lines 15, 30, and 35 have been generated as 0000. When the definition of ENDFIL on line 45 is encountered, the assembler generates the third Text record. This record specifies that the value 2024 (the address of ENDFIL) is to be loaded at location 201C (the operand address field of the JEQ instruction on line 30). When the program is loaded, therefore, the value 2024 will replace the 0000 previously loaded. The other forward references in the program are handled in exactly the same way. In effect, the services of the loader are being used to complete forward references that could not be handled by the assembler. Of course, the object program records must be kept in their original order when they are presented to the loader.

```
HCOPY  001000001 07A                    201C
T001000094 54F46000003000000
T00200F151410094800000100C2810063000004800003C2012
T00201C022024
T00202419001000C100F0010030C100C4800000810094C0000F1001000
T002013022 03D
T00203D1E041006001006E02039302043D82039281006300000 54900F2C203A382043
T002050022 05B
T00205B07101 00C4C000005
T00201F022062
T002031022062
T00206218041006E02061302065509 00FDC20612C100C3820654C0000
E00200F
```

Multi-Pass Assembler

In symbol defining statements – using EQU and ORG directive, the symbol or the expression giving the new value which used on the right hand side should be defined previously in the source program

Click to add

- DELTA can be defined in pass 1
- BETA can be defined in pass 2
- ALPHA can be defined in pass 3

- Consider the symbol definition process in two-pass assembler.

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

- In this sequence the symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined.

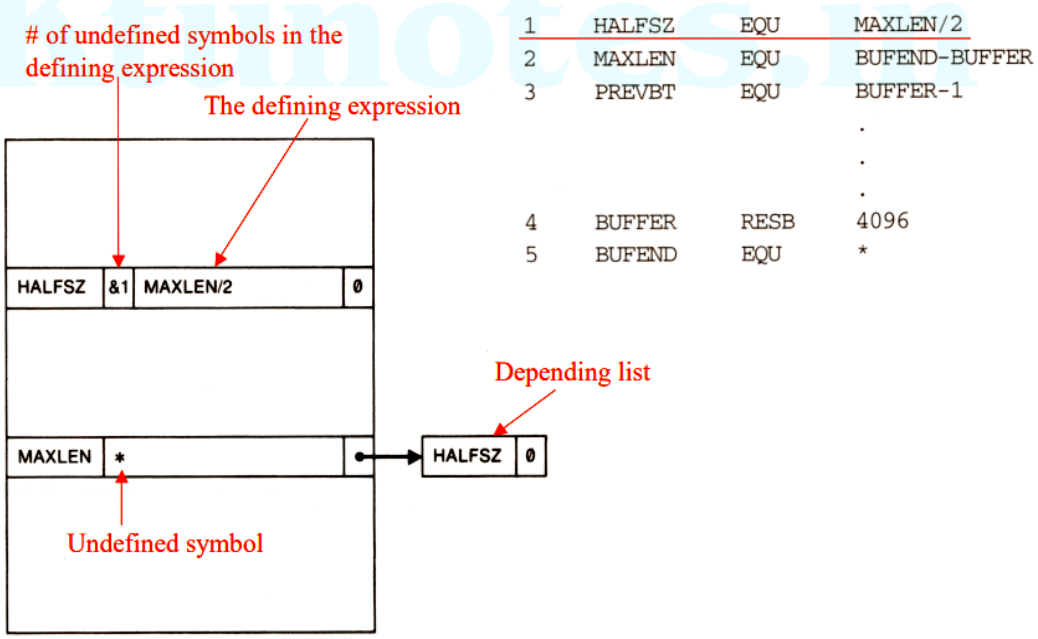
As a result, ALPHA cannot be evaluated during the second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions. The general solution is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbol. It is not necessary for such an assembler to make more than two passes over the entire program. Instead the portions of the program that involve

forward references in symbol definition are saved during pass 1. Additional passes through these stored definitions are made as the assembly progresses. This process is normally followed by a 2-pass assembler. These tasks are accomplished in several ways: Method involves storing of symbol definitions that involve forward references in the symbol table. This table also indicates which symbols are dependent on the values of others, to facilitate symbol evaluations

Consider the sequence of symbol defining statements that involve forward references:

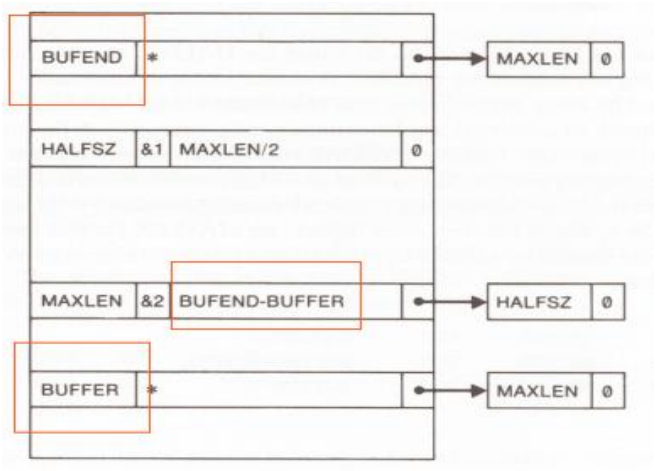
```
1    HALFSZ    EQU    MAXLEN/2
2    MAXLEN    EQU    BUFEND-BUFFER
3    PREVBT    EQU    BUFFER-1
                      .
                      .
                      .
4    BUFFER    RESB   4096
5    BUFEND    EQU    *
```

Symbol Table entries resulting from Pass 1 processing of the statement



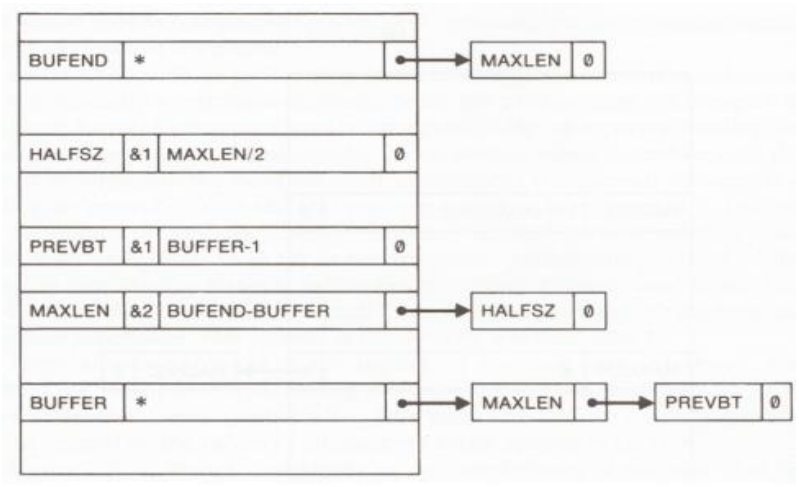
MAXLEN has not yet been defined, so no value for HALFSZ can be computed. The defining expression for HALFSZ is stored in the symbol table in place of its value. The entry &1 indicates that one symbol in the defining expression is undefined. In actual

implementation, this definition might be stored at some other location. SYMTAB would then simply contain a pointer to the defining expression. The symbol MAXLEN is also entered in the symbol table, with the flag * identifying it is undefined. Associated with this entry is a list of the symbols whose values depend on MAXLEN (here HALZSZ)→ similar to one-pass assembler we have seen.



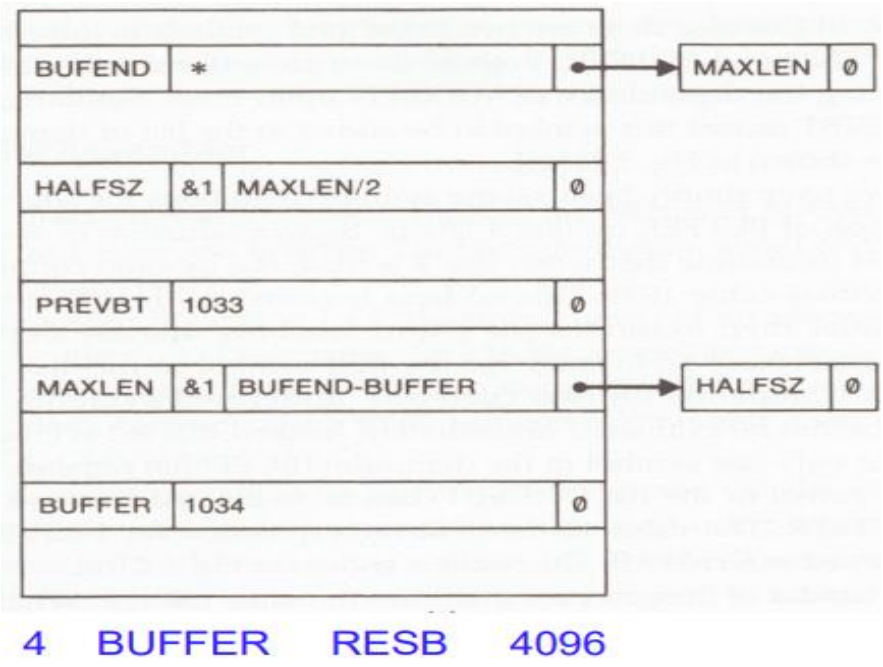

```
2   MAXLEN   EQU   BUFEND-BUFFER
```


In this case there are two undefined symbols: BUFFEND and BUFFER. Both these are entered into SYMTAB with lists indicating the dependence of MAXLEN upon them.

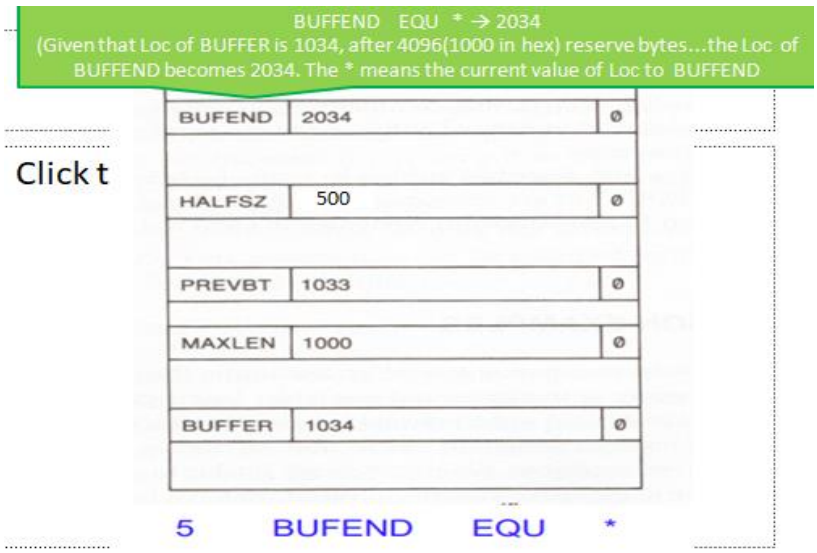



```
3       PREVBT      EQU      BUFFER-1
```

Similarly, the definitions of PREVBT cause this symbol to be added to the list of dependencies on BUFFER.



So far we are simply saving symbol definitions for later processing. Line 4 the definition of BUFFER begins the evaluation. Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034. This address is stored as the value of BUFFER. The assembler the examines the lists of symbols that are dependent on BUFFER. The symbol table entry for the first symbol in this list (MAXLEN) shows that it depends on two currently undefined symbols; so MAXLEN can be calculated immediately. Then &2 is changed to &1 to show that only one symbol in the definition (BUFFEND) remains undefined. The symbol PREVBT can be calculated and stored in SYMTAB.



When BUFFEND is defined in line 5, its value is entered into the symbol table. The list associated with BUFFEND then directs the assembler to evaluate MAXLEN, and entering a value for MAXLEN causes the evaluation of the symbol in its list (HALFSZ). This completes the symbol definition process. If any symbols remained undefined at the end of the program, the assembler would flag them as errors.

Traditional CISC Machines

- *CISC: Complex Instruction Set Computers*
- *Have :*
  - *Large and complicated instruction set*
  - *Different instruction formats and lengths*
  - *Many different addressing modes*

Pentium Pro Architecture

Introduced near 1995. Family of Intel x86

Memory

Described in two ways: At physical level memory consists of 8 bit bytes. All addresses used are byte addresses. Two consecutive bytes form a word. Four bytes form a double word (also called a dword). At programmers level the memory of x86 viewed as a collection of segments. So the address consists of two parts – a segment number and an offset that points to a byte within the segment. Segments can be of different sizes and are used for different purposes. Some segments contains executable instructions and other segments may be used to store data. Some data segments may be treated as stacks that can be used to save register contents, pass parameters to subroutines and for other purposes. It is not necessary for all of the segments used by the program to be in physical memory. In some cases a segment can also be divided into pages. Some pages of the segment may be in physical memory, while others may be stored on disk. When an x86 instruction is executed, the hardware and the operating system make sure that the needed byte of the segment is loaded into physical memory. The segment/offset address specified

by the programmer is automatically translated into a physical byte address by the x86 Memory Management Unit(MMU).

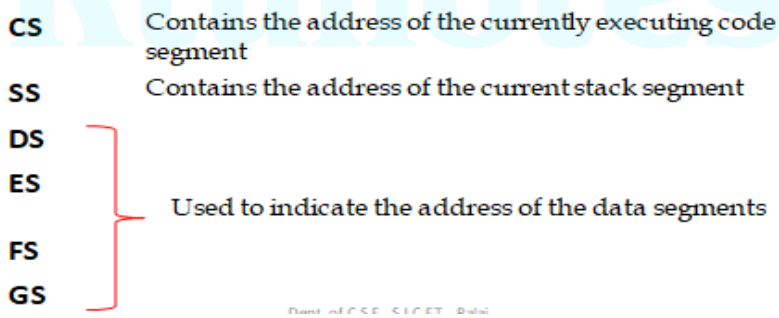
Registers

There are 8 general-purpose registers named as:

EAX, EBX, ECX, EDX, ESI,EDI, EBP and ESP

Each general –purpose registers are 32 bits long (ie. One double word). Registers EAX,EBX,ECX,EDX are generally used for data manipulation. It is possible to access individual works from these registers. These four registers commonly used to hold addresses

- *Special purpose registers:*
  - *EIP – a 32-bit register that contains a pointer to next instruction to be executed*
  - *FLAGS – a 32-bit register that contains many different bit flags(indicate the status of the processor, others to record results of comparisons and arithmetic operations)*
  - *Segment registers: 16 –bit registers that are used to locate segments in the memory*



- *FPU : Floating Point Unit*
  - *Contain eight 80 bit data registers and several other control and status registers*

Data Formats

Integers: are stored as 8-, 16-, 32-bit Binary numbers. Both signed and unsigned numbers (ordinals) are supported. 2's complement representation for negative numbers. Can also be stored in Binary Coded Format (BCD).

- *Packed: each byte represents two decimal digits, with each digit encoded (in binary) using 4 bits of byte*
- *Unpacked: each byte represents one decimal digit. The value of this digit is encoded(in binary) in the low-order 4 bits of the byte; the higher order are normally zero*

***Characters** are stored one per byte, and represented using 8-bit ASCII codes.*

***Strings** may consist of bits, bytes,words or doublewords; there are also special instructions to handle each type of string. FPU can handle 64-bit signed integers.*

***Floating points** are represented using three different formats namely,*

- *single-precision format (32 bit long→24 bit floating point value,7 bit exponent, 1 bit for storing the sign)*
- *double-precision format (64 bit long →53 significant bit, 10 bit exponent)*
- *extended-precision format(80 bit long →64 significant bits, 15 bit exponent)*

Instruction Formats

The format begins with an optional prefixes containing flags that modify the operation of the instruction

- *For example, some prefixes specify a repetition count for an instruction*
- *Others specify a segment register that is to be used for addressing an operand (overriding the normal default assumptions made by the hardware)*

Following the prefixes(if any) is an opcode (1 or 2 bytes). Some operations have different opcodes, each specifying a different variant. Following opcode are the number of bytes that specify the operands and addressing modes used. The opcode is the only element that is always present in every instruction. Other elements may or may not be present, and may be of different length depending on the operations and operand involved. Thus there are large number of potential instruction formats varying in length from 1 byte to 10 bytes or more.

Addressing modes

- *Provides a large number of addressing modes*
- *Immediate mode: Operand value may be specified as part of the instruction itself*
- *Register mode: Operand value may be in register*
- *Operands stored in memory are often specified using variations of the general target address calculation*
- *TA = (base register)+(index register)*(scale factor)+ displacement*
- *Any general purpose register may be used as a base register. Any general purpose register except ESP can be used as an index register.The scale factor may have the value 1,2,4,or . The displacement may be as 8,16,or 32 bit value*
- *The base and index register numbers, scale and displacement are encoded as parts of the operand specifiers in the instruction.*
- *Various combinations of these items may be omitted, resulting in eight different addressing modes*
- *Direct mode : The address of an operand in memory may also be specified as an absolute location*
- *Relative mode: The address of an operand in memory may also be specified as a location relative to the EIP register*

Instruction Set

- *There are more than 400 different instructions. An instruction may have zero, one, two or three operands*
- *There are Register-to-register instructions, register-to-memory instructions, and a few memory-to-memory instructions*
- *In some cases, operands may also be specified in the instruction as immediate value*

- *Most data movement and integer arithmetic instructions can use operands that are 1,2 or 4 bytes long*
- *String manipulation instructions, which uses repetition prefixes, can deal directly with variable-length strings of bytes, words or double words*
- *There are many instructions that perform logical and bit manipulations, and support control of the processor and memory management systems*
- *The x86 architecture also include special purpose instructions to perform operations frequently required in high-level programming languages.For example, entering and leaving procedures and checking subscript values against the bounds of an array*

Input and Output

- *Input is performed by instructions that transfer one byte, word or double word at a time from an I/O port into register EAX*
- *Output instructions transfer one byte, word or double word from EAX to an I/O port*
- *Repetition prefixes allow these instructions to transfer an entire string in a single operation*

Implementation-MASM Assembler

The programmer of an x86 views memory as a collection of segments. MASM assembler language program is written as a collection of segments. Each segment belongs to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST and STACK. During program execution, segments are addressed via the x86 segment registers. In most cases the code segments are addressed using register CS. Stack segments are addressed using the register SS. These segment registers are automatically set by the system loader when a program is loaded for execution. Register is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is set to indicate the last stack segment processed by the loader. Data segments (including constants segment) are normally addressed using DS,ES,FS or GS. The segment register to be used can be explicitly specified by the programmer (by writing it as a part of the assembler language program). If the programmer

does no specify a segment register, one is selected by the assembler. By default, the assembler assumes that all the references to data segments use register DS. This assumption can be changed by the assembler directive **ASSUME**

ASSUME

ASSUME ES: DATASEG2

Tell the assembler that register ES indicate the segment DATASEG2. Thus, any reference to labels are defined in DATASEG2 will be assembled using register ES. It is possible to collect several segments into a group and use ASSUME to associate a segment register with the group. 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. Similar to BASE directive in SIC/XE. BASE tell a SIC/XE assembler the contents of register B; programmer must provide the executable instructions to load this value into the register. Similarly, ASSUME tells MASM the contents of a segment register; the programmer must provide instructions to load this register when the program is executed. Jump instructions are assembled in two ways depending on whether the target of the jump is in the same code segment as the jump instruction

- Near jump: jump to a target address in the same code segment as the jump instruction
- Far jump: : jump to a target address in the different code segment

Near jump is assembled using the current code segment register CS. The assembled machine instruction for a near jump occupies 2 or 3 bytes. A far jump is assembled using a different segment register which is specified in the instruction prefix. The assembled machine instruction for a far jump occupies 5 bytes. Forward references to the labels in the source program can cause problem. For example: consider a jump instruction

JMP TARGET

If the definition of the label TARGET occurs in the program before the JMP instruction, the assembler can tell whether this is a far jump or near jump. If forward reference to TARGET, the assembler does not know how many bytes to reserve for the instruction. By default, the MASM assumes that a forward jump is a near jump. If the target is in another code segment, the programmer must warn the assembler by writing.

JMP FAR PTR TARGET

Programmer can specify the near jump by

JMP SHORT TARGET

If the programmer does not specify FAR PTR a problem occurs: During Pass 1, the assembler reserves 3 bytes for the jump instruction. But the actual assembled instruction requires 5 bytes. Earlier version of MASM causes a phase error. Later version, the assembler can repeat pass1 to generate the correct location counter values. Far jump is similar to forward references in SIC/XE that require the use of extended format instructions. Other situations in which the length of an assembled instruction depends on the operands that are used. Eg: For ADD instruction, the operand may be registers, memory or immediate operands. Immediate operands may occupy from 1 to 4 bytes in the instruction. An operand that specifies a memory location may tae varying amounts of space in the instruction, depending upon the location of the operand. Other situations in which the length of an assembled instruction depends on the operands that are used. Pass1 of x86 assembler is more complex than SIC. Segments in a MASM source program can be written in more than one part. If a **SEGMENT** directive specifies the same name as a previously defined segment, it is considered to be continuation of that segment.

All the parts of a segment are gathered together by the assembly process. References between segments that are assembled together are automatically handled by the assembler. External references between separately assembled modules must be handled by the linker. PUBLIC is used in MASM instead for EXTDEF in SIC/XE. EXTRN is used in MASM instead for EXTREF in SIC/XE. The object program from MASM assembler may be in several different format. MASM can produce instruction timing listing that shows the number of clock cycles required to execute each machine instruction. This allows the programmer to exercise a great deal of control in optimizing time-critical sections of code.