K S INSTITUTE OF TECHNOLOGY

#14,Raghuvanahalli, Kanakapura Main Road,Bengaluru-560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SYSTEM SOFTWARE & COMPILER DESIGN

15CS63

DEEPA.S.R

ASSOCIATE PROFESSOR

DEPARTMENT OF CSE

Assembler Design

Assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to the assembler is a source code written in assembly language (using mnemonics) and the output is the object code. The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

1. Basic Assembler Functions:

The basic assembler functions are:

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

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

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

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

- SIC Assembler Directive:
  - START: Specify name & starting address.
  - END: End of the program, specify the first execution instruction.

- BYTE, WORD, RESB, RESW
- End of record: a null char(00)
  End of file: a zero length record

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

Single-pass Assembler:

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

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

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

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

Pass-1

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

Pass-2

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

Assembler Design:

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

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

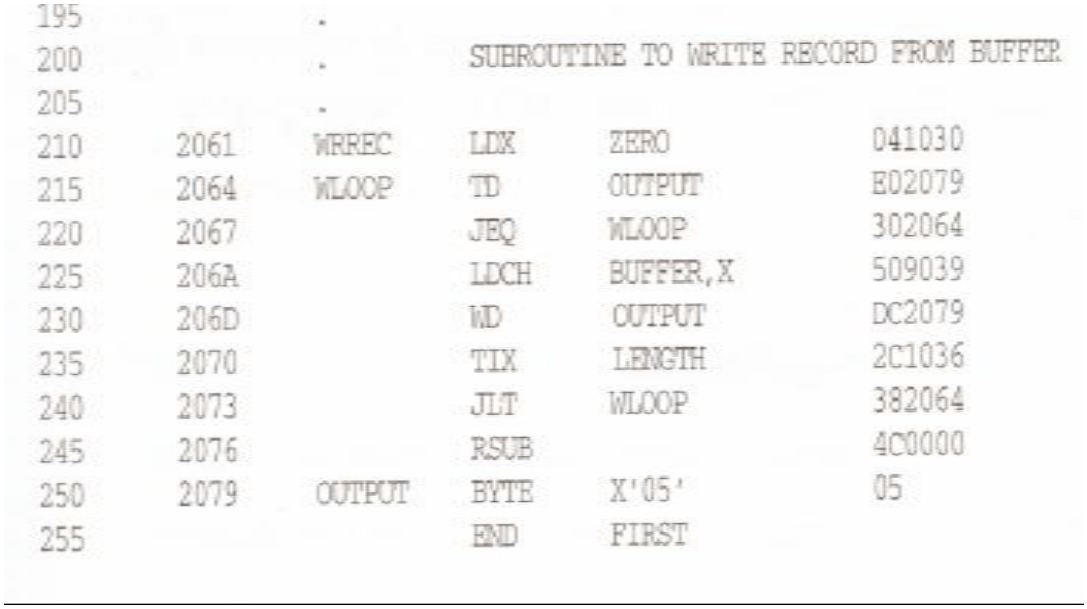
Example Program:

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

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

Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			

110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	2039	RDREC	LDX	ZERO	041030
130	203C		LDA	ZERO	001030
135	203F	RLOOP	TD	INPUT	E0205D
140	2042		JEQ	RLOOP	30203F
145	2045		RD	INPUT	D8205D
150	2048		COMP	ZERO	281030
155	204B		JEQ	EXIT	302057
160	204E		STCH	BUFFER,X	549039
165	2051		TIX	MAXLEN	2C205E
170	2054		JLT	RLOOP	38203F
175	2057	EXIT	STX	LENGTH	101036
180	205A		RSUB		4C0000
185	205D	INPUT	BYTE	X'F1'	F1
190	205E	MAXLEN	WORD	4096	001000
195					



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

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

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

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

Object code for the example program:

Session –II

1. Simple SIC Assembler

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

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

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

10 1000 LDA ALPHA 00-----

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

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

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

The format of each record is as given below.

Header record:

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

Text record:

Col. 1	T

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

End record:

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

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

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

2. Algorithms and Data structure

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

OPTAB: It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length. In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR. In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table-that is, entries are not normally added to or deleted from it. In such cases it is possible to

design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB: This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).

During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.

SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2. A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

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

The Algorithm for Pass 1:

```
Begin
  read first input line
   if OPCODE = ‘START’ then begin
      save #[Operand] as starting addr
      initialize LOCCTR to starting address
      write line to intermediate file
       read next line
     end( if START)
    else
      initialize LOCCTR to 0
```

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

The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line. If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value. If the symbol already exists that indicates an

entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol. It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction. If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes. If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

The Algorithm for Pass 2:

```
Begin
  read 1st input  line
    if OPCODE =  'START' then
      begin
        write listing line
         read next input line
       end
    write Header record to object program
    initialize 1st Text record
while OPCODE != 'END' do
     begin
       if this is not comment line then
         begin
           search OPTAB for OPCODE
            if found then
              begin
                if there is a symbol in OPERAND field then
                    begin
                       search SYMTAB for OPERAND field then
                       if found then
                       begin
                              store symbol value as operand address
                       else
                         begin
                              store 0 as operand address
                              set error flag (undefined symbol)
                         end
                   end (if symbol)
                 else store 0 as operand address
                       assemble the object code instruction
                 else if OPCODE = 'BYTE' or 'WORD" then
                      convert constant to object code
                  if object code doesn't fit into current Text record then
                      begin
```

```
                    Write text record to object code
                    initialize new Text record

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

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code. If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

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

Design and Implementation Issues

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

3. Machine-Dependent Features:

- Instruction formats and addressing modes
- Program relocation

3.1 Instruction formats and Addressing Modes

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

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

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

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

During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass 2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2. Translation involving Register-Memory instructions:

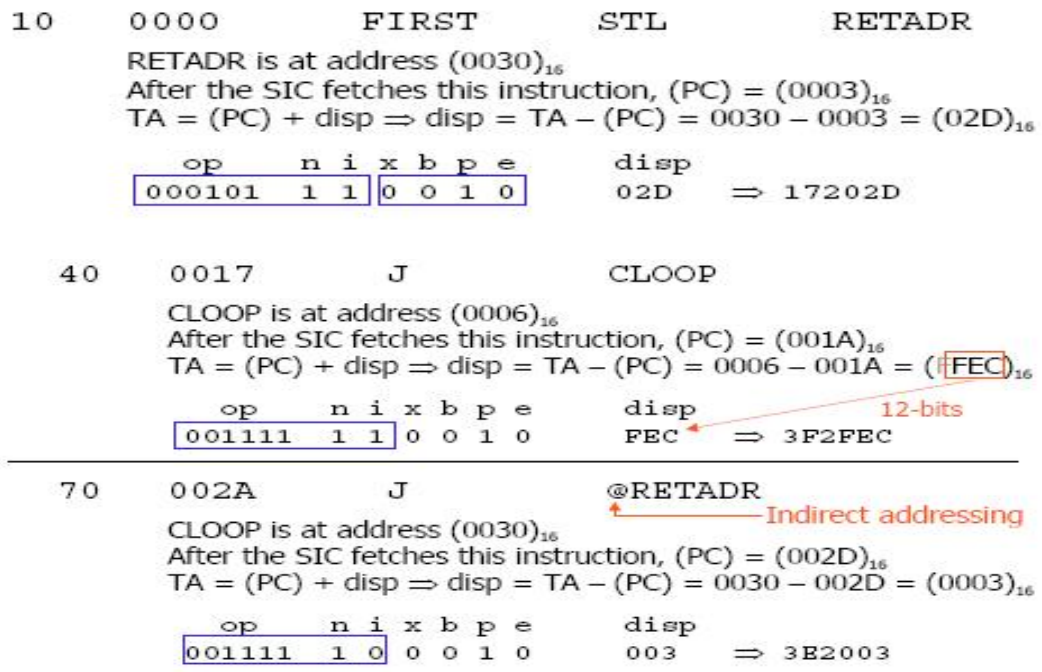
In SIC/XE machine there are four instruction formats and five addressing modes. For formats and addressing modes refer chapter 1.

Among the instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other

operand is in the memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

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

2. Program-Counter Relative: In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction. This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:



3. Base-Relative Addressing Mode: in this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE. The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.

When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

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

For example:

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

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

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

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

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

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

Consider Line 175. If we use PC-relative

Disp = TA – (PC) = 0033 –1059 = EFDA

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

4. Immediate Addressing Mode

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

```
55    0020        LDA         #3
                                  <-- Immediate operand
      TA = (0003)16
        op     n i x b p e      disp
      000000   0 1 0 0 0 0      003    ⇒ 010003

133   103C        +LDT        #4096
                  <-- Extended instruction format
      TA = (01000)16
        op     n i x b p e      disp(20 bits)
      011101   0 1 0 0 0 1      01000 ⇒ 75101000
```

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

```
12    0003        LDB         #LENGTH

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

***5. Indirect and PC-relative mode*:**

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

```
70    002A        J           @RETADR
                    :             :
95    0030 RETADR RESW        1

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

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

3.2 Program Relocation

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

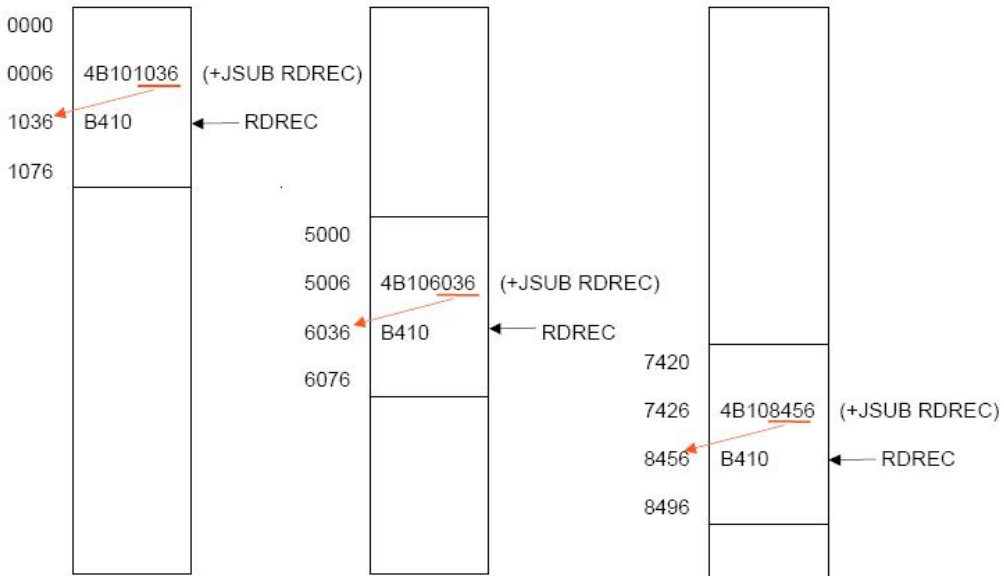
Absolute Program

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

```
55      101B                LDA         THREE       00102D
```

This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000. Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000. Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.

Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification. An object program that has the information necessary to perform this kind of modification is called the relocatable program.



The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006. The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000. The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.

The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions. From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.

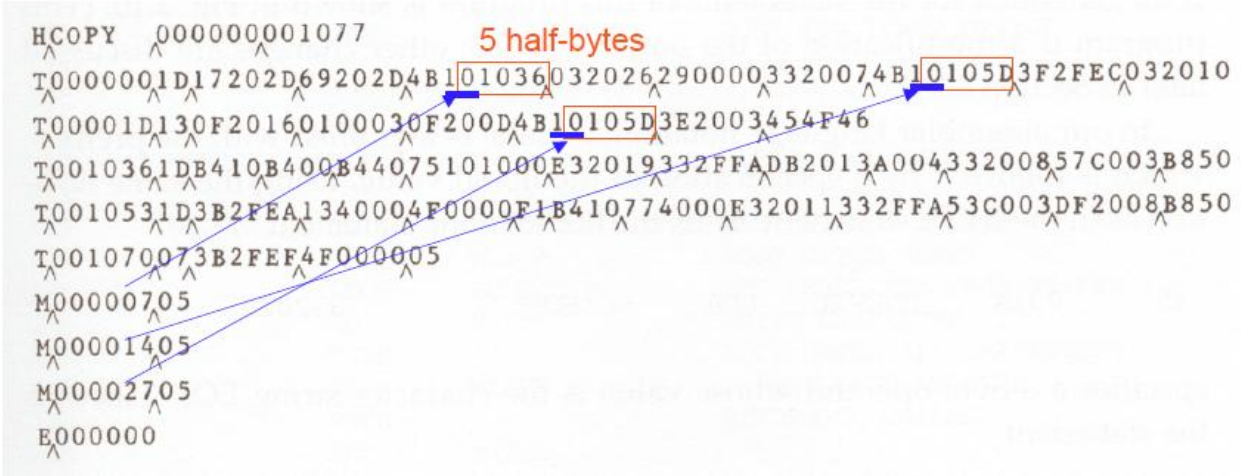
For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes (4 bits) The starting location is the location of the byte containing the

leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.



In the above object code the red boxes indicate the addresses that need modifications. The object code lines at the end are the descriptions of the modification records for those instructions which need change if relocation occurs. M00000705 is the modification suggested for the statement at location 0007 and requires modification 5-half bytes. Similarly the remaining instructions indicate.

Assembler Design Session 4 & 5

3.2 Machine-Independent features:

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

- Literals
- Expressions
- Program blocks
- Control sections

3.2.1 Literals:

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

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

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative

displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

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

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

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

All the literal operands used in a program are gathered together into one or more *literal pool*s. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

A literal table is created for the literals which are used in the program. The literal table contains the *literal name, operand value and length.* The literal table is usually created as a hash table on the literal name.

Implementation of Literals:

During Pass-1:

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG for literal definition. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITTAB

SYMTAB

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

LITTAB

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

3.2.2 Symbol-Defining Statements:

EQU Statement:

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

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example

```
+LDT    #4096
```

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

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

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

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

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

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

```
BASE    EQU     R1
INDEX   EQU     R2
COUNT   EQU     R3
```

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

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

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

ORG Statement:

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

```
ORG     value
```

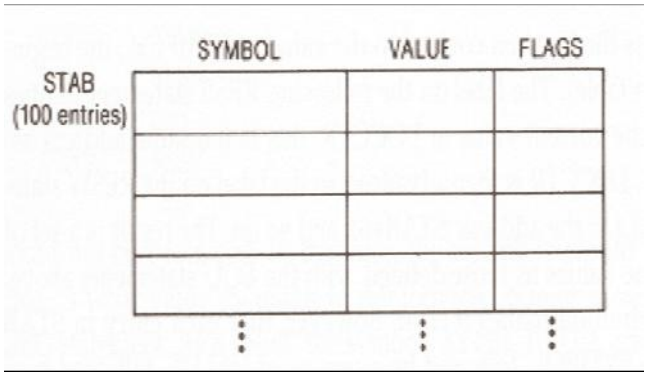
Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR,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

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

```
SYMBOL    6 Bytes
VALUE     3 Bytes
```

FLAG 2 Bytes

The table looks like the one given below.



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

```
STAB    RESB    1100
```

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

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

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

```
LDA     VALUE, X
```

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

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

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

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

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

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

3.2.3 Expressions:

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

```
BUFFEND     EQU         *
```

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

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

```
MAXLEN      EQU         BUFEND-BUFFER
```

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

```
MAXLEN    EQU       1000
```

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

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

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

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

3.2.4 Program Blocks:

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

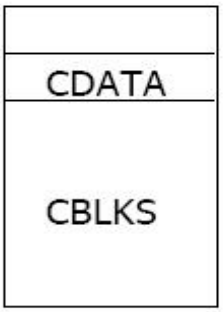
Assembler Directive USE:

```
USE    [blockname]
```

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

In the example below three blocks are used :

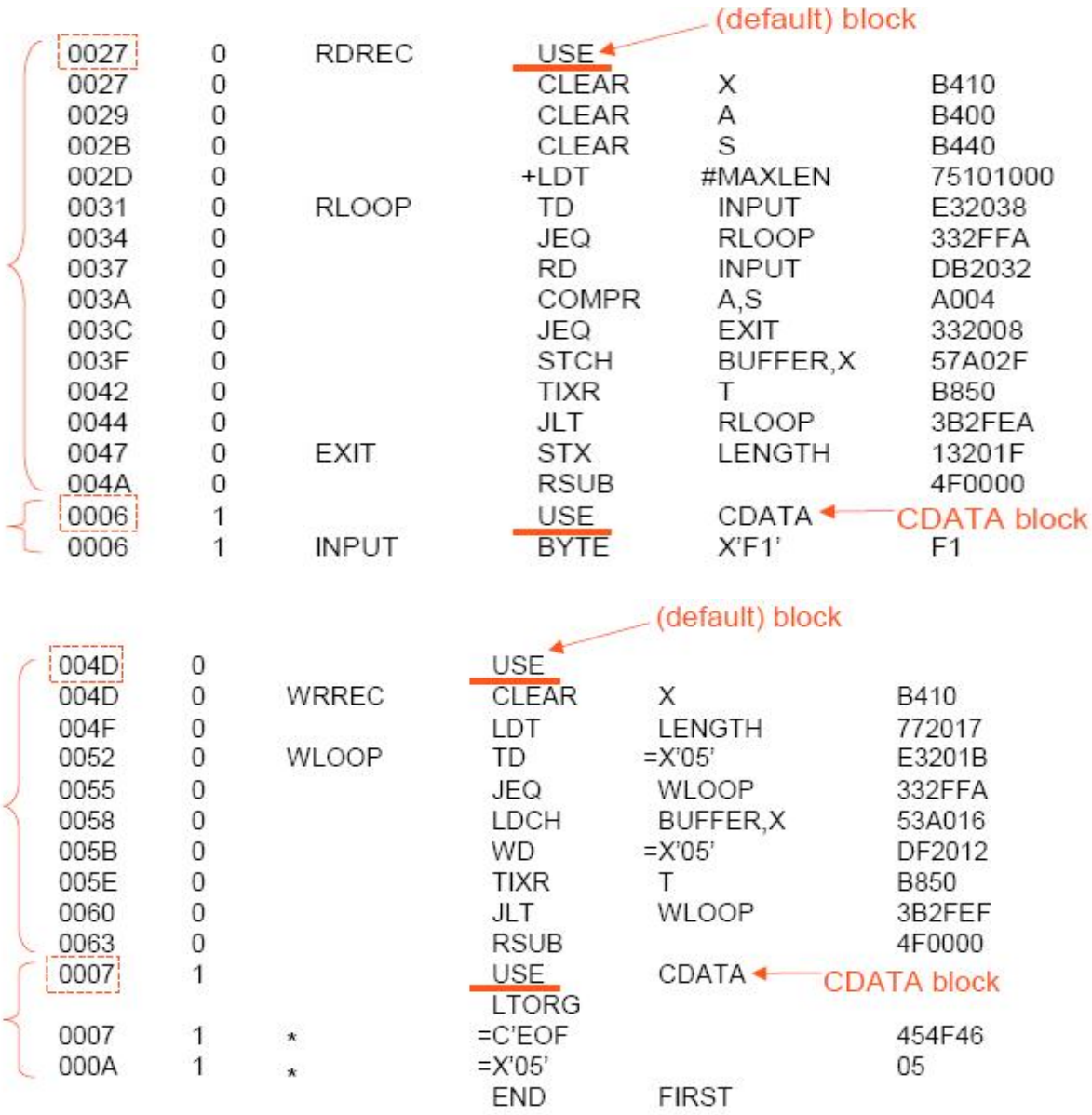
Default: executable instructions
CDATA: all data areas that are less in length
CBLKS: all data areas that consists of larger blocks of memory



Example Code

(default) block — Block number

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



Arranging code into program blocks:

Pass 1

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

3.2.5 Control Sections:

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

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

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

- assembler directive: **CSECT**

The syntax

secname CSECT

- separate location counter for each control section

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

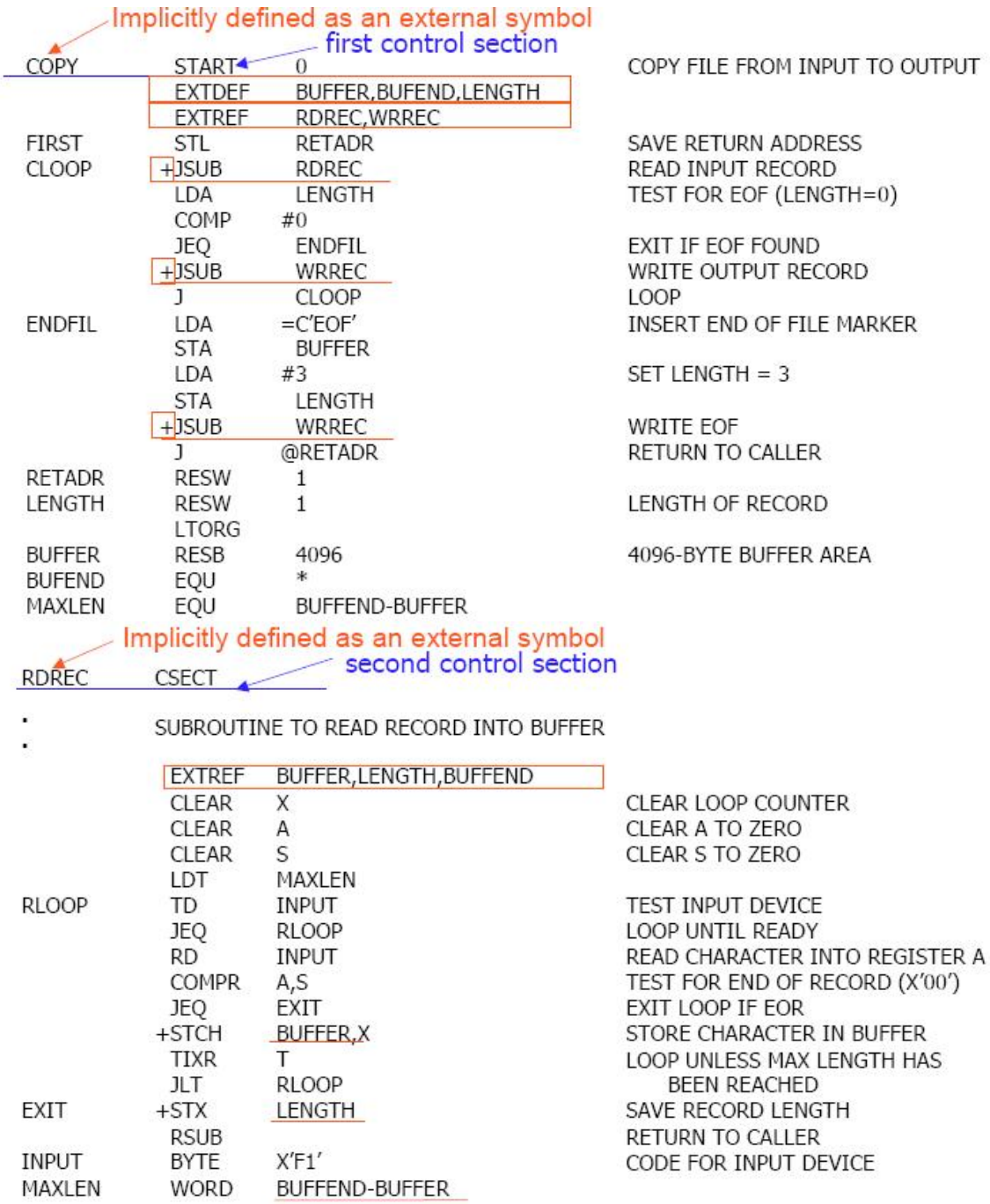
EXTDEF (external Definition):

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

EXTREF (external Reference):

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

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




```
.
.           SUBROUTINE TO WRITE RECORD FROM BUFFER

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

Handling External Reference

Case 1

15 0003 CLOOP +JSUB RDREC 4B100000

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

Case 2

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

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

Case 3

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

```
107     1000 MAXLEN     EQU         BUFEND-BUFFER
```

Object Code for the example program:

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

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

```
0000    WRREC   CSECT

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

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

Define record (EXTDEF)

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

Refer record (EXTREF)

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

Modification record

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

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

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

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification my be defined either in this control section or in another section.

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

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

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

COPY

```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC  WRREC
T000000 1D 1720274B100000 032023290000 3320074B100000 3F2FEC 032016 0F2016
T00001D 0D 010003 0F200A 4B1000003E2000
T000030 03 454F46
M000004 05 +RDREC
M000011 05 +WRREC
M000024 05 +WRREC
E000000
```

RDREC

```
HRDREC  000000 00002B
RBUFFER LENGTH BUFEND
T000000 1D B410 B400 B440 77201F E3201B 332FFA DB2015 A00433 200957 900000 B850
T00001D 0E 3B2FE9 13100000 4F0000 F1 000000
M000018 05 +BUFFER
M000021 05 +LENGTH
M000028 06 +BUFEND
M000028 06 -BUFFER
E
```

BUFEND - BUFFER

WRREC

```
HWRREC  000000 00001C
RLENGTH BUFFER
T000000 1C B410 7710000 0E3201 232FFA 53900000 DF2008 B8503B 2FEE4F 000005
M000003 05 +LENGTH
M00000D 05 +BUFFER
E
```

Handling Expressions in Multiple Control Sections:

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

When it comes in a program having multiple control sections then we have 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.

3.5 ASSEMBLER DESIGN

Here we are discussing

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

3.5.1 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'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110					

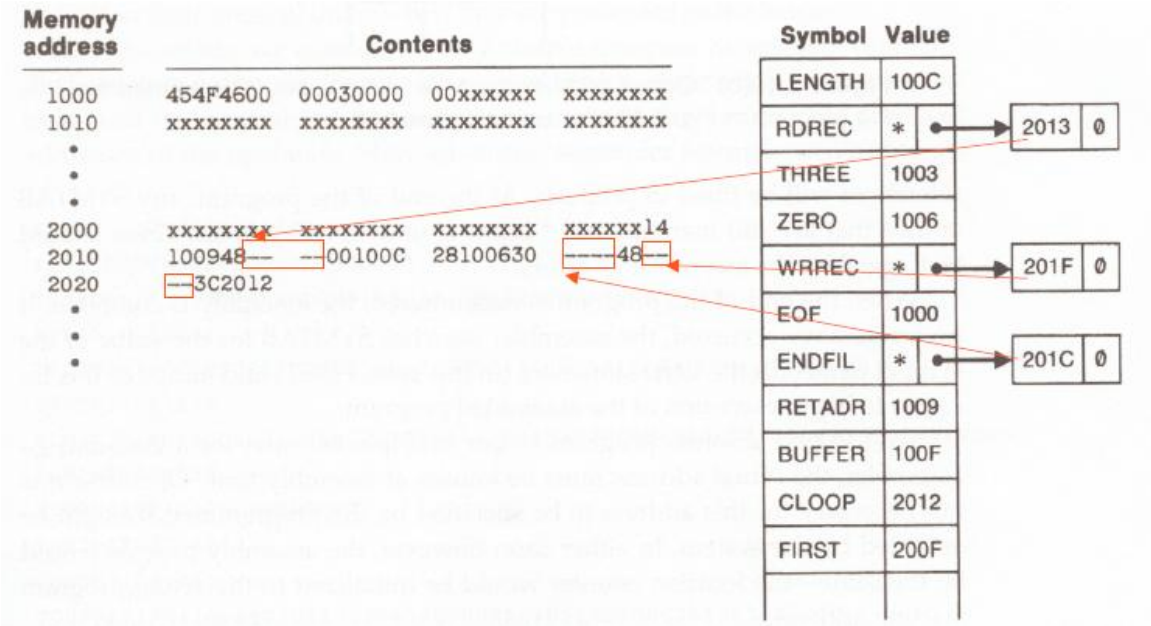
Forward Reference in One-Pass Assemblers: In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler
    - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

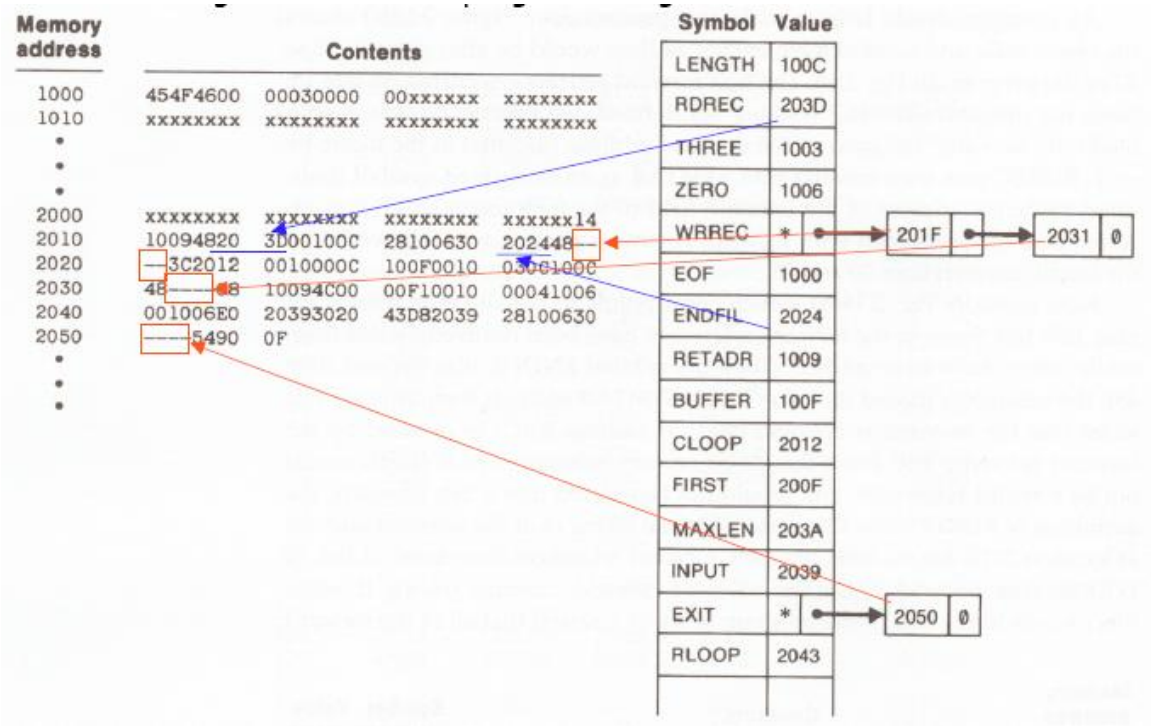
After Scanning line 40 of the program:

40 2021 J` CLOOP 302012

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



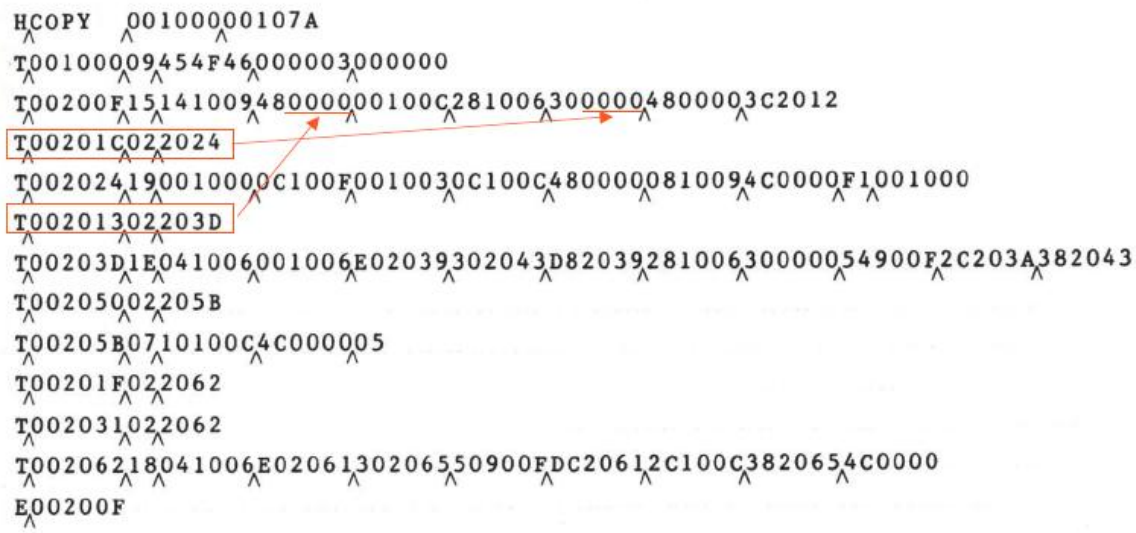
The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:



If One-Pass needs to generate object code:

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

Object Code Generated by One-Pass Assembler:



Multi_Pass Assembler:

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

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

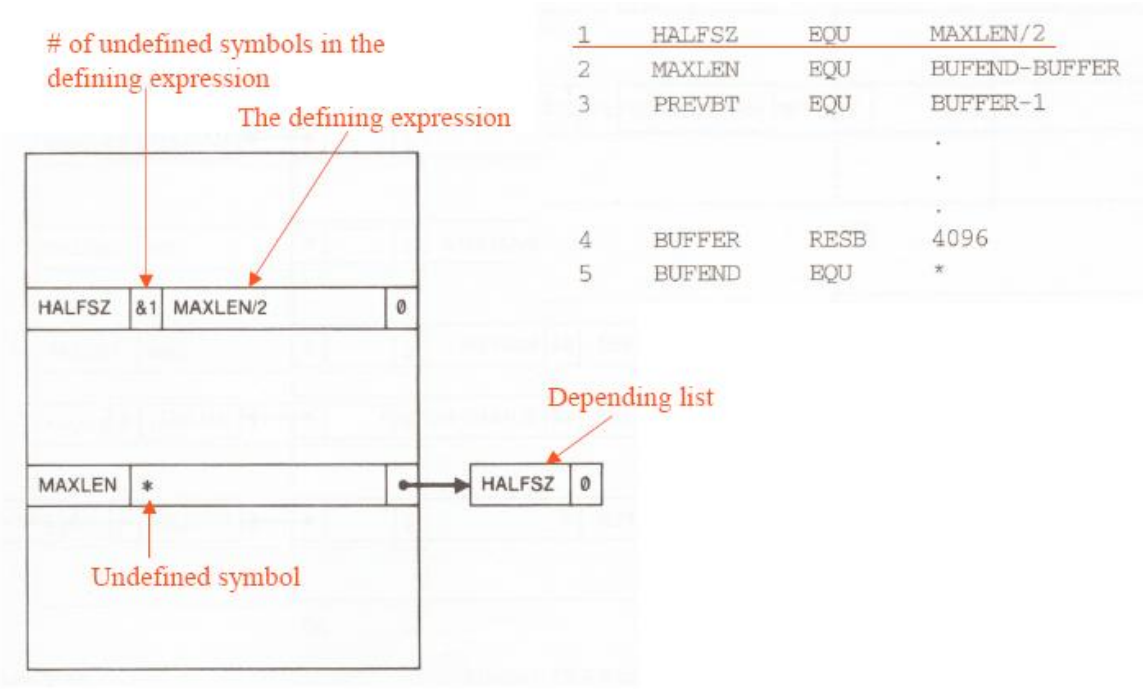
  - Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

Implementation Issues for Modified Two-Pass Assembler:

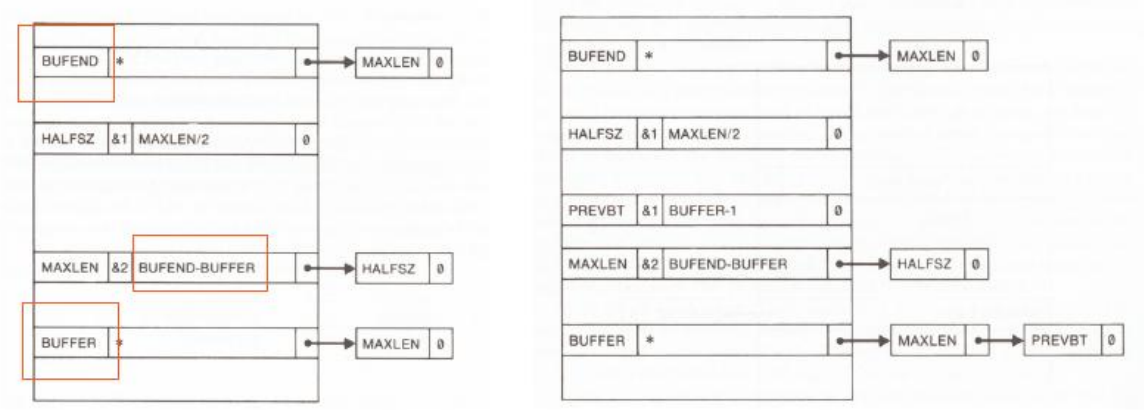
Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

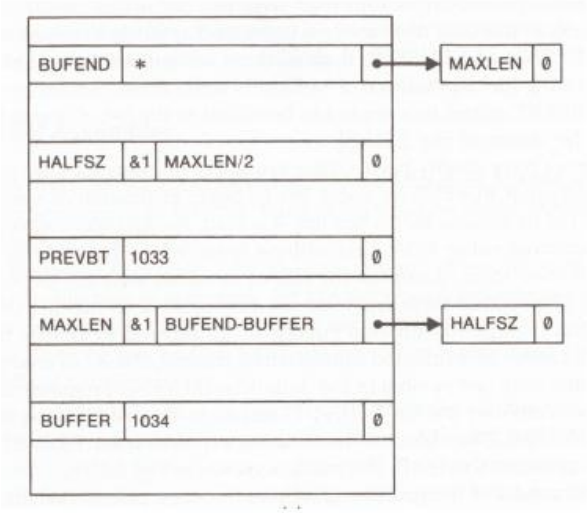
- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- When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

Multi-Pass Assembler Example Program

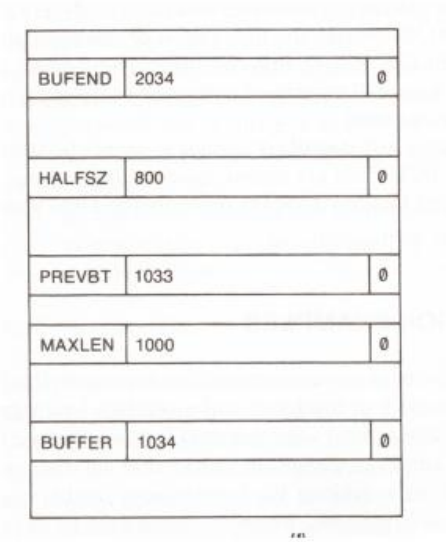


Multi-Pass Assembler (Figure 2.21 of Beck): Example for forward reference in Symbol Defining Statements:






```
4    BUFFER    RESB    4096
```




```
5    BUFEND    EQU    *
```

Chapter 4: Macro Processor

A *Macro* represents a commonly used group of statements in the source programming language.

- A macro instruction (macro) is a notational convenience for the programmer
  - It allows the programmer to write shorthand version of a program (module programming)
- The macro processor replaces each macro instruction with the corresponding group of source language statements (*expanding*)
  - Normally, it performs no analysis of the text it handles.
  - It does not concern the meaning of the involved statements during macro expansion.
- The design of a macro processor generally is *machine independent!*
- Two new assembler directives are used in macro definition
  - **MACRO:** identify the beginning of a macro definition
  - **MEND:** identify the end of a macro definition
- Prototype for the macro
  - Each parameter begins with '&'
    - name MACRO parameters
:
body
:
MEND
  - Body: the statements that will be generated as the expansion of the macro.

4.1 Basic Macro Processor Functions:

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

4.1.1 Macro Definition and Expansion:

The figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.

M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

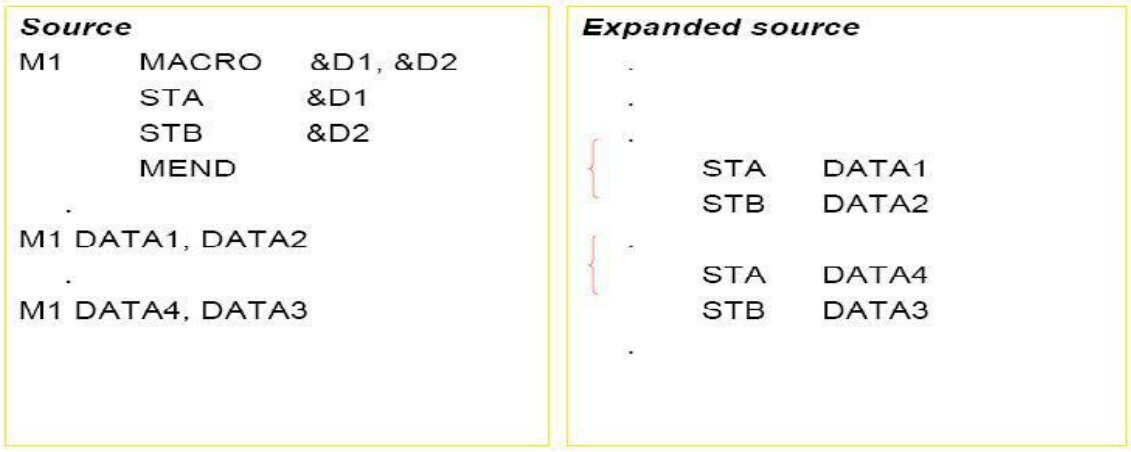


Fig 4.1

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

Macro Expansion:

The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed.

The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

After *macro processing* the expanded file can become the input for the *Assembler*. The *Macro Invocation* statement is considered as comments and the statement generated from expansion is treated exactly as though they had been written directly by the programmer.

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

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

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

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

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

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

Fig 4.2

4.1.2 Macro Processor Algorithm and Data Structure:

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

Two-pass macro processor

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

MACROS for SIC machine

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

Fig 4.3(a)

MACROX for SIC/XE Machine

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

Fig 4.3(b)

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

One-Pass Macro Processor:

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

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

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

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

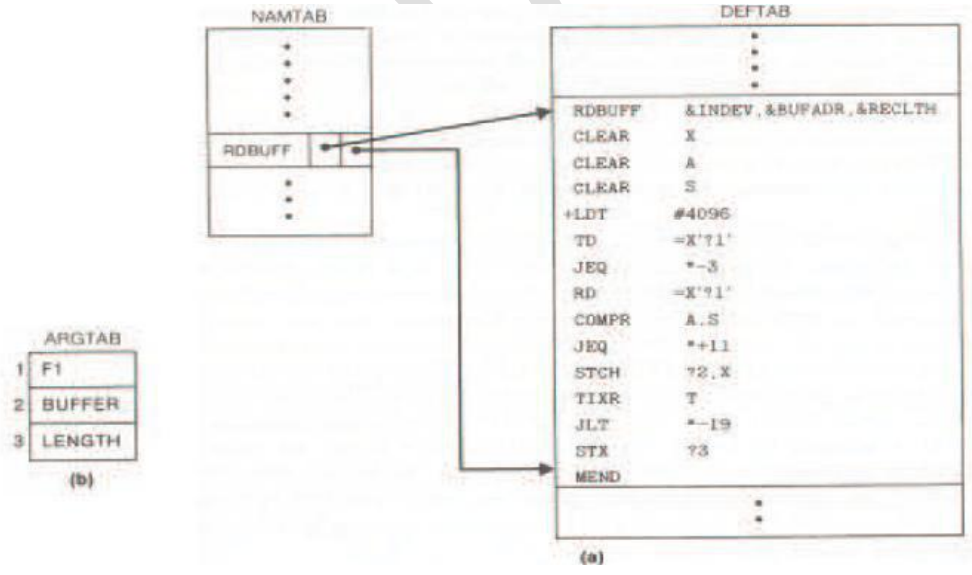


Fig 4.4

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

```
TD      =X'?1'
```

The above instruction is to test the availability of the device whose number is given by the parameter &INDEV. In the instruction this is replaced by its positional value? 1. Figure 4.4(b) shows the ARTAB as it would appear during expansion of the RDBUFF statement as given below:

```
CLOOP       RDBUFF      F1, BUFFER, LENGTH
```

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

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

When a macro definition is encountered it is entered in the DEFTAB. The normal approach is to continue entering till MEND is encountered. If there is a program having a Macro defined within another Macro. While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the difintion of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL. Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

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

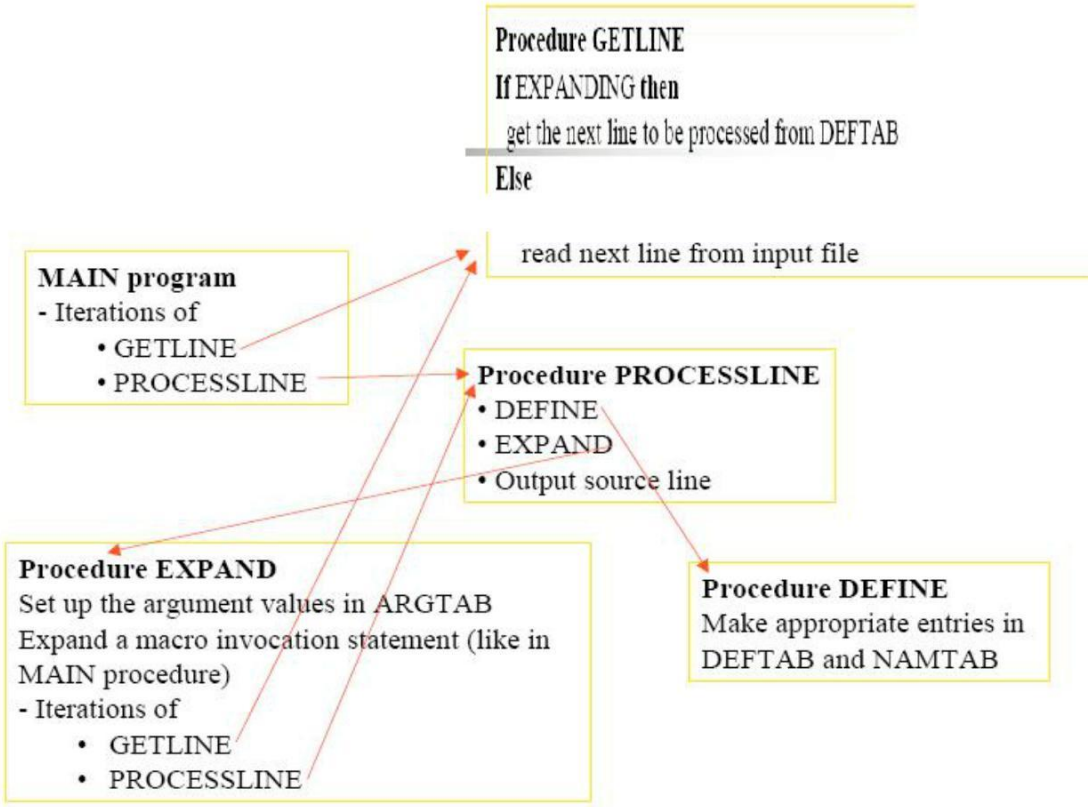

Procedure GETLINE
If EXPANDING then
get the next line to be processed from DEFTAB
Else
read next line from input file
MAIN program
- Iterations of
• GETLINE
• PROCESSLINE
Procedure PROCESSLINE
• DEFINE
• EXPAND
• Output source line
Procedure EXPAND
Set up the argument values in ARGTAB
Expand a macro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

Fig 4.5

Algorithms

```
begin {macro processor}
        EXPANDINF := FALSE
        while OPCODE ≠ 'END' do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
end {macro processor}
```

```
Procedure PROCESSLINE
        begin
          search MAMTAB for OPCODE
          if found then
                EXPAND
          else if OPCODE = 'MACRO' then
                DEFINE
          else write source line to expanded file
        end {PRCOESSOR}
```

```
Procedure DEFINE
        begin
            enter macro name into NAMTAB
            enter macro prototype into DEFTAB
            LEVEL   :- 1
            while LEVEL > do
                begin
                    GETLINE
                    if this is not a comment line then
                      begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL +1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL – 1
                      end {if not comment}
                end {while}
            store in NAMTAB pointers to beginning and end of definition
end {DEFINE}
```

```
Procedure EXPAND
    begin
            EXPANDING := TRUE
            get first line of macro definition {prototype} from DEFTAB
            set up arguments from macro invocation in ARGTAB
            while macro invocation to expanded file as a comment
            while not end of macro definition do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
            EXPANDING := FALSE
    end {EXPAND}

 Procedure GETLINE
     begin
             if EXPANDING then
                begin
                   get next line of macro definition from DEFTAB
                   substitute arguments from ARGTAB for positional notation
                end {if}
             else
                read next line from input file
      end {GETLINE}
```

Fig 4.6

Write the object program

loc	len	label	mnemonic	operand	Object code	op	n i x b p e	**disp**	Disp=TA/ TA-PC/ TA-B
		SUM	START	0					
0000	3	FIRST	LDX	#0	050000	**0**01	0 1 0 0 0 0	**000**	
0003	3		LDA	#0	010000	**0**00	0 1 0 0 0 0	**000**	
0006	4		+LDB	#TABLE2	69101790	**6**10	0 1 0 0 0 1	**01790**	
			BASE	TABLE2					
000A	3	LOOP	ADD	TABLE,X	1BA013	**1**10	1 1 1 0 1 0	**013**	0020-000D=013
000D	3		ADD	TABLE2,X	1BC000	**1**10	1 1 1 1 0 0	**000**	1790-0010=1780=$(6016)_D$ Not in range of PC, so use **BASE 1790-1790=000**
0010	3		TIX	COUNT	2F200A	**2**11	1 1 0 0 1 0	**00A**	1D-13=0A
0013	3		JLT	LOOP	3B2FF4	**3**10	1 1 0 0 1 0	**FF4**	SUBTRACT USING 2S COMPLIMENT
0016	4		+STA	TOTAL	0F102F00	**0**11	1 1 0 0 0 1	**02F00**	
001A	3		RSUB		4F0000				
001D	3	COUNT	RESW	1					
0020	1770	TABLE	RESW	2000					
1790	1770	TABLE2	RESW	2000					
2F00	3	TOTAL	RESW	1					
2F03			END	FIRST					

OBJECT PROGRAM

H^ SUM ^000000^2F03

T^000000^1D^050000^----------^4F0000

M^000007^05

M^000017^05

E^000000

Write the object program

LOCATION	LENGTH	LABEL	MNEMONIC	OPERAND	OBJECT CODE
		SIMPLE	START	1000	
1000	3	FIRST	LDA	FIVE	00102B
1003	3		STA	ALPHA	0C1034
1006	3		LDX	ZERO	04102E
1009	3	MOVECH	LDCH	STR1,X	509015
100C	3		STCH	STR2,X	549020
100F	3		TIX	ELEVEN	2C1031
1012	3		JLT	MOVECH	381009
1015	11	STR1	BYTE	C' TEST STRING	54 45 53 54 20 53 54 52 49 4E 47
1020	11	STR2	RESB	11	
102B	3	FIVE	WORD	5	000005
102E	3	ZERO	WORD	0	000000
1031	3	ELEVEN	WORD	11	00000B

1034	3	ALPHA	RESW	1	

OBJECT PROGRAM

H^SIMPLE^001000^000035

T^001000^15^00102B,0C1034,.....381009

T^001015^1C^544553,542053,.....000000

T^001031^03^00000B

E^001000