Module 2 (Assembly Language Programming and Assemblers)

SIC/XE Programming, Basic functions of Assembler, Assembler output format – Header, Text and End Records. Assembler data structures, Two pass assembler algorithm.

Assemblers

At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of 1's and 0's (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions.

Programmers found it difficult to write or read programs in machine language. In their quest for a more convenient language they began to use a mnemonic (symbol) for each machine instructions, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as Assemblers were written to automate the translation of assembly language into machine language. The input to an assembler program is called source program. The output is a machine language translation called object program.

Basic Assembler functions

- Convert **symbolic operands** to their equivalent **machine addresses** (eg: RETADR to 1033)
- Convert **mnemonic operation codes** to their **machine language equivalents** (eg: STL to 14)
- Convert the **data constants** specified in the source program into their **internal machine representations** (eg: EOF to 454F46)

- Write the object program and the assembly listing
- Build the machine instruction into proper format

Assembler Directive

An assembler directive is a set of instructions used to instruct the assembler to perform certain actions during the assembly of the program. This statement neither represents machine instructions that are to be included in the object program nor indicate the storage allocation of constants or variables. It directs the assembler to take certain actions during the process of assembling a program. START, END, BYTE, WORD, RESW, RESB, BASE, EQU, ORG, LTORG are some of the assembler directives.

1. START : Specify name and starting address for the program

   eg: COPY START 1000

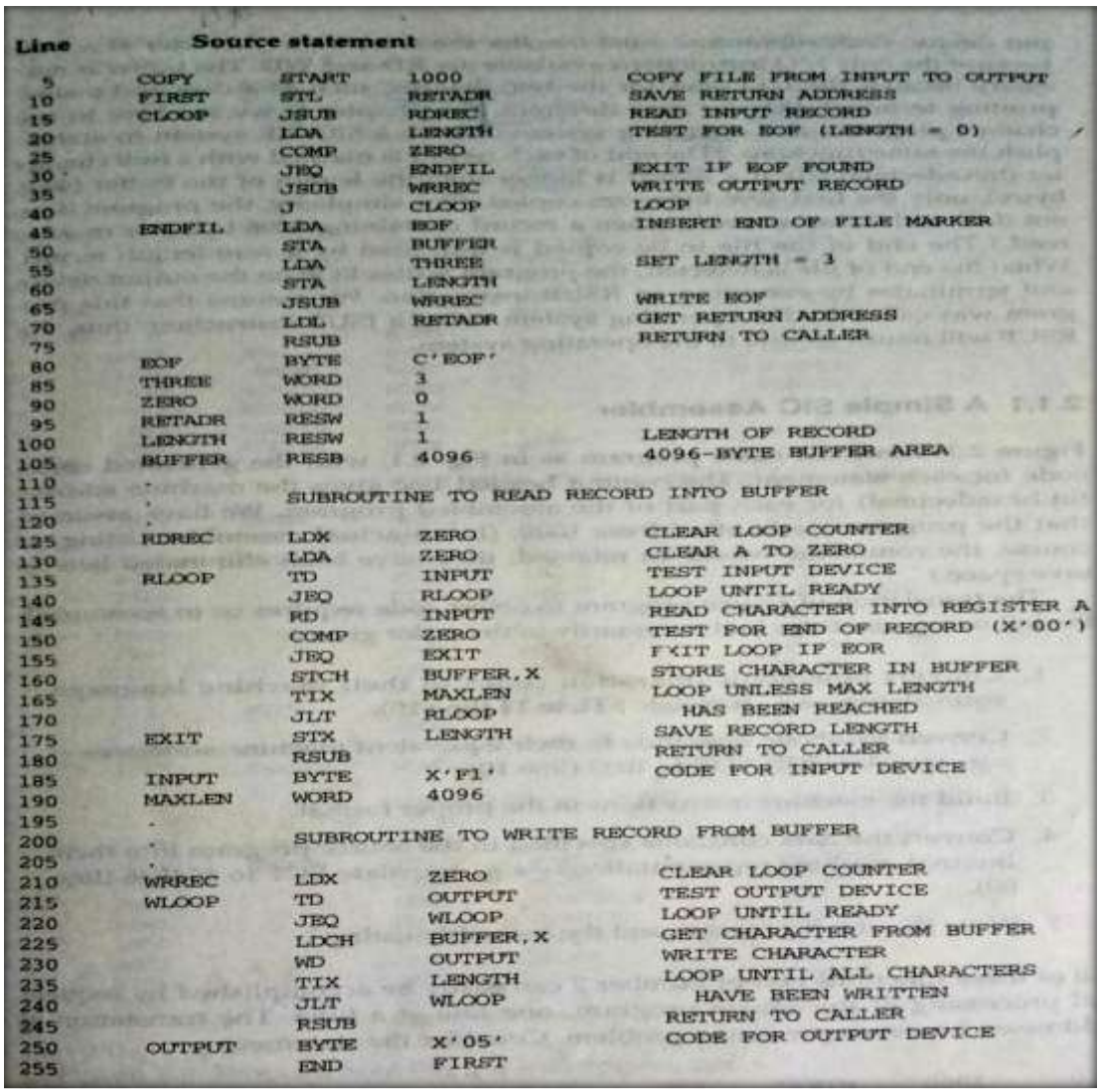
2. END : Indicate the end of the source program and (optionally) specify the first executable instruction in the program

   eg: END FIRST

There are four different ways of defining storage for data items in the SIC Assembler language:

1. BYTE : Generate **character or hexadecimal constant**, occupying as many bytes as needed to represent the constant
   - Eg: CHARZ BYTE C'Z'
2. WORD : Generate one-word **integer constant**
   - Eg: FIVE WORD 5
3. RESB : Reserve the indicated number of bytes for a data area
   - eg: C1 RESB 1
4. RESW : Reserve the indicated number of words for a data area
   - eg: ALPHA RESW 1

Example of a SIC Assembler language program



```
Line        Source statement
   5    COPY      START   1000          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    ZERO
  30              JEQ     ENDFIL        EXIT IF EOF FOUND
  35              JSUB    WRREC         WRITE OUTPUT RECORD
  40              J       CLOOP         LOOP
  45    ENDFIL    LDA     EOF           INSERT END OF FILE MARKER
  50              STA     BUFFER
  55              LDA     THREE         SET LENGTH = 3
  60              STA     LENGTH
  65              JSUB    WRREC         WRITE EOF
  70              LDL     RETADR        GET RETURN ADDRESS
  75              RSUB                  RETURN TO CALLER
  80    EOF       BYTE    C'EOF'
  85    THREE     WORD    3
  90    ZERO      WORD    0
  95    RETADR    RESW    1
 100    LENGTH    RESW    1             LENGTH OF RECORD
 105    BUFFER    RESB    4096          4096-BYTE BUFFER AREA
 110    .
 115    .         SUBROUTINE TO READ RECORD INTO BUFFER
 120    .
 125    RDREC     LDX     ZERO          CLEAR LOOP COUNTER
 130              LDA     ZERO          CLEAR A TO ZERO
 135    RLOOP     TD      INPUT         TEST INPUT DEVICE
 140              JEQ     RLOOP         LOOP UNTIL READY
 145              RD      INPUT         READ CHARACTER INTO REGISTER A
 150              COMP    ZERO          TEST FOR END OF RECORD (X'00')
 155              JEQ     EXIT          EXIT LOOP IF EOR
 160              STCH    BUFFER,X      STORE CHARACTER IN BUFFER
 165              TIX     MAXLEN        LOOP UNLESS MAX LENGTH
 170              JLT     RLOOP           HAS BEEN REACHED
 175    EXIT      STX     LENGTH        SAVE RECORD LENGTH
 180              RSUB                  RETURN TO CALLER
 185    INPUT     BYTE    X'F1'         CODE FOR INPUT DEVICE
 190    MAXLEN    WORD    4096
 195    .
 200    .         SUBROUTINE TO WRITE RECORD FROM BUFFER
 205    .
 210    WRREC     LDX     ZERO          CLEAR LOOP COUNTER
 215    WLOOP     TD      OUTPUT        TEST OUTPUT DEVICE
 220              JEQ     WLOOP         LOOP UNTIL READY
 225              LDCH    BUFFER,X      GET CHARACTER FROM BUFFER
 230              WD      OUTPUT        WRITE CHARACTER
 235              TIX     LENGTH        LOOP UNTIL ALL CHARACTERS
 240              JLT     WLOOP           HAVE BEEN WRITTEN
 245              RSUB                  RETURN TO CALLER
 250    OUTPUT    BYTE    X'05'         CODE FOR OUTPUT DEVICE
 255              END     FIRST
```

Figure shows an assembler language program for the basic version of SIC. Line numbers are given only for reference and are not the part of the program. Then there are labels defined by the programmer. Then mnemonic instructions (opcode) eg: STL, JSUB. Indexing addressing is indicated by adding modifier "X" (line 160). Then comments are represented by "."

The program contains a main routine that reads records from an input device, identified by device code F 1 and copies them to an output device 05. Main routine calls subroutine RDREC to read a record into buffer and another subroutine WRREC to write the record from the buffer to the output device. Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WD. The buffer is necessary: because the I/O rates for

two devices, such as a disk and a slow printing terminal, may be different. The end of each record is marked with a null character (hexadecimal 00).

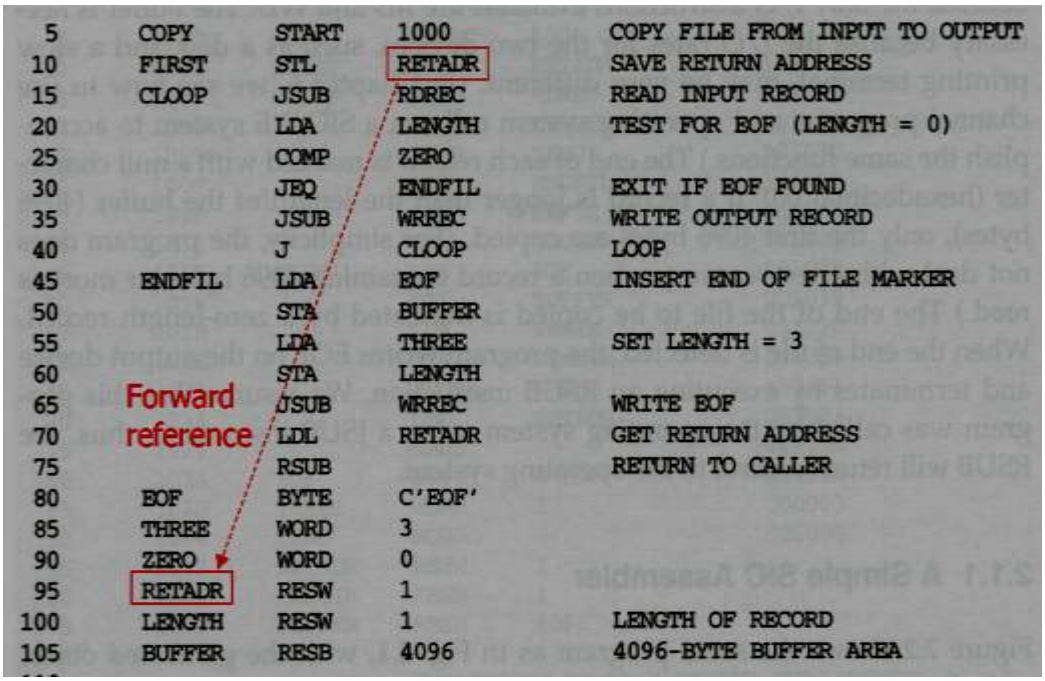
If a record is longer than the length of a buffer (4096 bytes), only the first 4096 bytes are copied.(*for simplicity, the program does not deal with the error recovery when a record containing 4096 bytes or more is read.* The end of the file to be copied is indicated by a zero-length record. When the end of file is detected, the program writes EOF on the output device and terminates by executing an RSUB instruction. *Assumed that the this program was called by the operating system using a JSUB instruction and thus the RSUB will return the control to the operating system*

Forward Reference

Convert symbolic operands to their equivalent machine addresses (eg: RETADR to 1033). This cannot be achieved in the sequential processing of the source program, one line at a time. This poses a problem : Forward Reference

Forward Reference – a reference to label (RETADR) that is defined later in the program. If we attempt to translate the program line by line, we will unable to process this statement because we do not know the address that will be assigned to RETADR. Because of this, most assemblers make two passes over the source program.

- **PASS 1:**
  - Scan the source program for label definitions and assign addresses (such as the Loc column)
- **PASS 2:**
  - Performs the actual translation



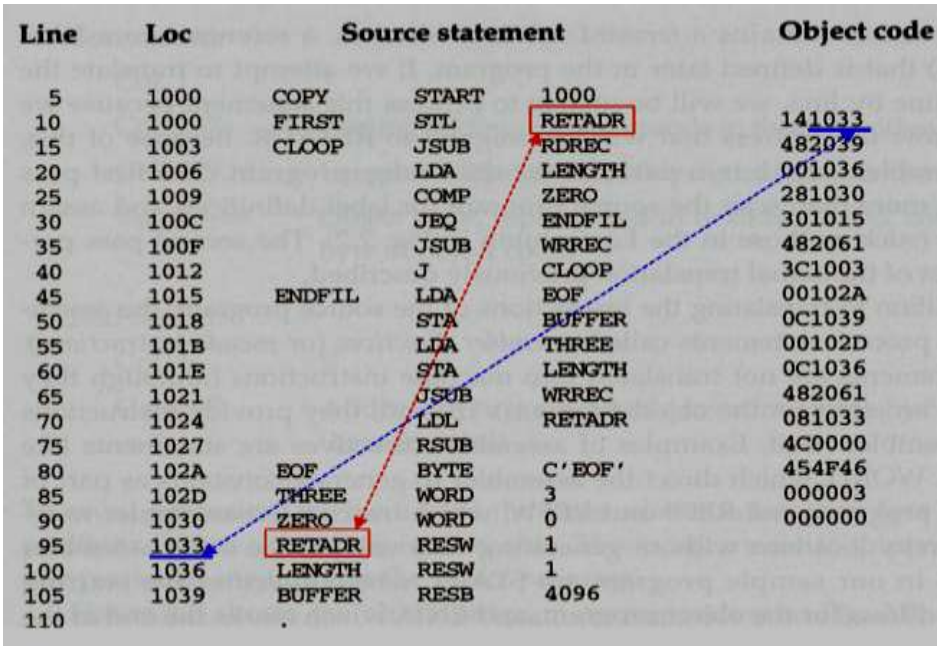

```
 5     COPY      START   1000        COPY FILE FROM INPUT TO OUTPUT
10     FIRST     STL     RETADR      SAVE RETURN ADDRESS
15     CLOOP     JSUB    RDREC       READ INPUT RECORD
20               LDA     LENGTH      TEST FOR EOF (LENGTH = 0)
25               COMP    ZERO
30               JEQ     ENDFIL      EXIT IF EOF FOUND
35               JSUB    WRREC       WRITE OUTPUT RECORD
40               J       CLOOP       LOOP
45     ENDFIL    LDA     EOF         INSERT END OF FILE MARKER
50               STA     BUFFER
55               LDA     THREE       SET LENGTH = 3
60               STA     LENGTH
65               JSUB    WRREC       WRITE EOF
70               LDL     RETADR      GET RETURN ADDRESS
75               RSUB                RETURN TO CALLER
80     EOF       BYTE    C'EOF'
85     THREE     WORD    3
90     ZERO      WORD    0
95     RETADR    RESW    1
100    LENGTH    RESW    1           LENGTH OF RECORD
105    BUFFER    RESB    4096        4096-BYTE BUFFER AREA
```

Assembler Output Format

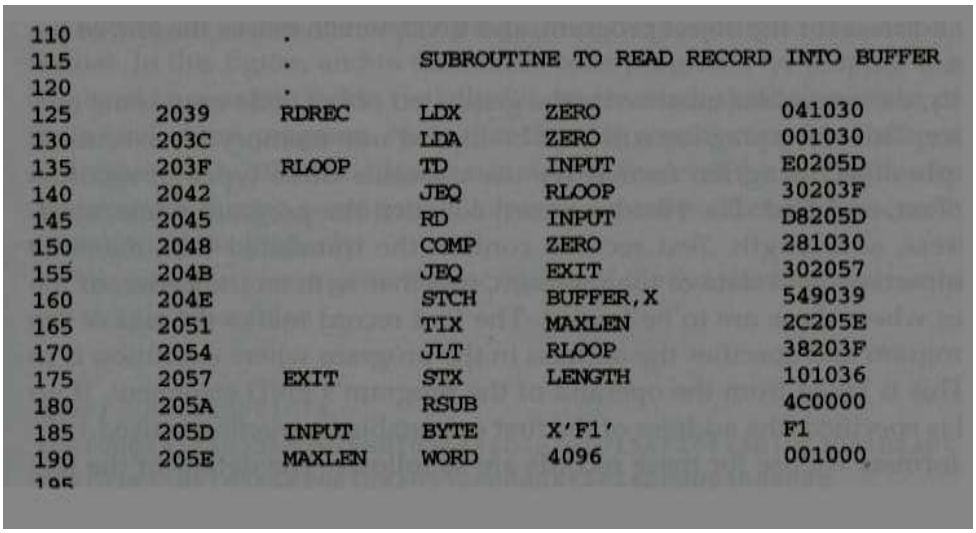
Finally, the assembler must write the generated object code onto some output device. The object program will later be loaded into memory for execution. The simple object program format uses 3 types of records: Header, Text and End.

- **Header record** contains the program name, starting address and length.
- **Text records** contain the translated (ie., machine code) instructions and data of the program, together with an indication of the addresses where these are to be loaded.
- **End record** marks the end of the object program and specifies the address in the program where execution is to begin.

- **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)



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
```

Figure below shows the sample object program generated for the above given simple SIC assembly program.

```
HCOPY  001000001107A
T001000 1E 141033 482039 001036 281030 301015 482061 3C1003 00102A 0C1039 00102D
T00101E 150C1036 482061 081033 4C0000 454F46 000003 000000
T002039 1E 041030 001030 E0205D 30203F D8205D 281030 302057 549039 2C205E 38203F
T002057 1C 101036 4C0000 F1 001000 041030 E02079 302064 509039 DC2079 2C1036
T002073 07 382064 4C0000 05
E001000
```

Address 1033 ~ 2038: reserve storage by loader
- RETADR: 3 bytes
- LENGTH: 3 bytes
- BUFFER: 4096 bytes = $(1000)_{16}$

To avoid confusions, we have used he term column rather than byte to refer to positions within object program records. This is not meant to imply the use of any particular medium for the object program. "^" used to separate fields visually and is not present in the actual object program. Note there is no object code corresponding to the addresses 1033- 2038 → this storage is simply reserved by the loader for use by the program during execution.

Passes of Assembler

A Pass is defined as the processing activity of every single statement in the source code to perform a set of language processing functions. Pass can also be defined as the activity of scanning the assembly language programming.

- **Single pass Assembler:** The assembler scans the entire source program (assembly language program) once and convert into an object code.

- **Multi-pass Assembler:** The translation of assembly language program into object code requiring many passes.

The breaking of the entire assembly process into passes makes design simpler and enables better control over the subtasks and intermediate operations.

Functions of Two Passes of Assembler

- **PASS 1 (*Define symbols)***
  - Assign addresses to all statements in the program
  - Save the values (addresses) assigned to all labels for use in Pass 2
  - Perform some processing of assembler directives. (This includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW, etc.)
- **PASS 2** (*Assemble instructions and generate object program)*
  - Assemble instructions (translating operation codes and looking up addresses)
  - Generate data values defined by BYTE, WORD, etc.
  - Perform processing of assembler directives not done during Pass 1
  - Write the object program and assembly listing

Assembler Data Structures

Simple Assembler uses two major internal data structures:

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)

Also need a variable Location Counter (LOCCTR).

OPTAB

OPTAB is used to look up mnemonic operation codes and translate them to machine language equivalents. This must contain at least mnemonic operation code and its machine language equivalent. In more complex assemblers, this table also contains information about instruction format and length. During Pass 1 OPTAB is used to look up and validate operation codes in the source program. In Pass 2, it is used to translate the operation codes to machine language.

In case of SIC/XE machine that has instruction of different length. We must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR. In second pass, the information from OPTAB tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instructions (typically most real assemblers).

OPTAB is usually organised as a hash table, with mnemonic operation code as the key. This information in OPTAB is predefined when the assembler itself is written, rather than being loaded into the table at the execution time. This hash table organisation provides fast retrieval with a minimum of searching. OPTAB is static table – entries are not normally added to or deleted from it.

SYMTAB

SYMTAB is used to store values(address) assigned to labels. SYMTAB includes the name and value(address) for each label in the source program, together with flags to indicate error conditions(eg: a symbol defined in two different places).The table may contain other information about the data area or instruction labelled (eg: it's type or length). During Pass 1 , the labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses (from LOCCTR). During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions

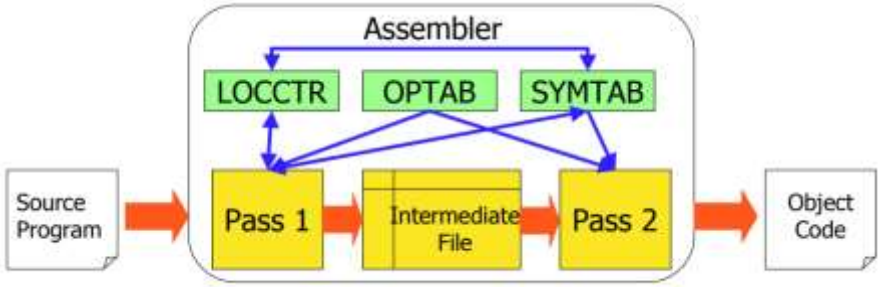
SYMTAB is usually organised as hash table for efficiency in insertion and retrieval. Entries are rarely deleted from this table. Programmers often select many labels that have similar characteristics (eg: label start or end with the same characters , like LOOP1, LOOP2, LOOPA,...or are of same length like A, X, Y, Z). Hashing function selected should perform well with such non random keys. Care should be taken in the selection of hashing function because the SYMTAB is used throughout the assembly. Good option is the selection of hash function which divides the entire key by a prime table length.

LOCCTR

LOCCTR is a variable that is used to help in the assignment of addresses. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR. Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

Assembler Algorithm

Both passes of the assembler reads the original source program as input. However, there is certain information (such as location counter values and error flags for the statements) that can or should be communicated between the two passes. Pass 1 usually writes an *intermediate file* that contains each source statements together with its assigned address, error indicators etc. This file is used as input to Pass 2. Means this working copy of the source program(intermediate file) can also be used to retain the results of certain operations that may be performed during Pass 1 (such as scanning the operand field for symbols and addressing flags), so these need not be performed again during Pass 2. Similarly, pointers into OPTAB and SYMTAB may be retained for each operation code and symbol used.



The intermediate file include each source statement, assigned address and error indicator

Algorithm explains the logic flow of two passes of assembler. Apply the algorithm to source program (assembly language) to generate object program. For simplicity, we assume that source lines are written in the fixed format with fields:

LABEL OPCODE OPERAND

If one of these fields contains a character string that represents a number, we denote its numeric value with a prefix #. (eg: #(OPERAND))

Pass 1:

```
begin
   read first input line
   if OPCODE = 'START' then
      begin
         save #[OPERAND] as starting address
         initialize LOCCTR to starting address
         write line to intermediate file
         read next input line
      end {if START}
   else
      initialize LOCCTR to 0
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               if there is a symbol in the LABEL field then
                  begin
                     search SYMTAB for LABEL
                     if found then
                        set error flag (duplicate symbol)
                     else
                        insert (LABEL,LOCCTR) into SYMTAB
                  end {if symbol}
               search OPTAB for OPCODE
               if found then
                  add 3 {instruction length} to LOCCTR
               else if OPCODE = 'WORD' then
                  add 3 to LOCCTR
               else if OPCODE = 'RESW' then
                  add 3 * #[OPERAND] to LOCCTR
               else if OPCODE = 'RESB' then
                  add #[OPERAND] to LOCCTR
               else if OPCODE  = 'BYTE' then
                  begin
                     find length of constant in bytes
                     add length to LOCCTR
                  end {if BYTE}
               else
                  set error flag (invalid operation code)
            end {if not a comment}
         write line to intermediate file
         read next input line
      end {while not END}
   write last line to intermediate file
   save (LOCCTR - starting address) as program length
end {Pass 1}
```

Pass 2:

```
begin
   read first input line (from intermediate file)
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end (if START)
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               search OPTAB for OPCODE
               if found then
                  begin
                     if there is a symbol in OPERAND field then
                        begin
                           search SYMTAB for OPERAND
                           if found then
                              store symbol value as operand address
                           else
                              begin
                                 store 0 as operand address
                                 set error flag (undefined symbol)
                              end
                        end (if symbol)
                     else
                        store 0 as operand address
                     assemble the object code instruction
                  end (if opcode found)
               else if OPCODE = 'BYTE' or 'WORD' then
                  convert constant to object code
               if object code will not fit into the current Text record then
                  begin
                     write Text record to object program
                     initialize new Text record
                  end
               add object code to Text record
            end (if not comment)
         write listing line
         read next input line
      end (while not END)
   write last Text record to object program
   write End record to object program
   write last listing line
end (Pass 2)
```