MODULE II
ASSEMBLERS

SYLLABUS:

Basic Functions of Assembler. Assembler output format – Header, Text and End Records- Assembler data structures, Two pass assembler algorithm, Hand assembly of SIC/XE program, Machine dependent assembler features.

2.1 Basic Functions of Assembler

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

- Functions of an assembler includes:
  - Translating mnemonic operation codes to their machine language equivalents.
  - Assigning machine addresses to symbolic labels used by the programmer.
- The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

2.2 A simple SIC Assembler

- The translation of source program to object code requires the following functions:
  1. Convert mnemonic operation codes to their machine language equivalents. Eg: In the program given on next page Translate STL to 14 (line 10).
  2. Convert symbolic operands to their equivalent machine addresses. Eg:Translate RETADR to 1033 (line 10).
  3. Build the machine instructions in the proper format.
  4. Convert the data constants specified in the source program into their internal machine representations. Eg: Translate EOF to 454F46(line 80).
  5. Write the object program and the assembly listing.

Consider the following assembly language program for SIC. This program contains a main routine that calls the subroutine RDREC which reads records from an input device(code F1) and WRREC which copies them to an output device(code 05).
The main routine calls subroutines:

- RDREC – To read a record into a buffer.
- WRREC – To write the record from the buffer to the output device.

At the end of the file it writes EOF on the output device.(The end of each record is marked with a null character (hexadecimal 00)).

The line numbers are for reference only. Indexed addressing is indicated by adding the modifier ”X” following the operand. Lines beginning with ”.” contain comments only.

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		.			
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		.			

```
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 2.1 –Example of a SIC assembler language program

Explanation of above program(No need to study, just to understand the program):

```
Program copy {
                    save return address;
  cloop:    call subroutine RDREC to read one record;
                if length(record)=0 {
                call subroutine WRREC to write EOF;
                    }  else {
               call subroutine WRREC to write one record;
                    goto cloop;
                }
                load return address


                 return to caller
    }

 Subroutine RDREC {
                      clear A, X register to 0;
   rloop:     read character from input device to A register
                  if not EOR {  //EOR is character 00
                    store character into buffer[X];
                    X++;
                    if X < maximum length
                                  goto rloop;
                      }
                  store X to length(record);
                  return
  }
```

```
Subroutine WDREC {
                    clear X register to 0;
  wloop:    get character from buffer[X]
                write character from X to output device
                X++;
                if X < length(record)
                        goto wloop;
                return
   }
```

Data transfer (RD, WD)

A buffer (BUFFER) is used to store record. The end of each record is marked with a null character (0016). Buffer length is 4096 Bytes The end of the file is indicated by a zero-length record(EOF). When the end of file is detected, the program writes EOF on the output device and terminates by RSUB.

Subroutines (JSUB, RSUB)

RDREC is the subroutine for reading records, WRREC is the subroutine for writing the records to output device. The contens of the linkage register (L) is saved into RETADR variable before jumping to subroutine.

Figure 2.1 shows a sample SIC assembler language program along with the generated object code for each statement. Assembler directives, START,END, RESW, RESB, WORD, BYTE etc do not generate the object code but directs the assembler to perform certain operation.. Assume the program starting at address 1000. 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 object program contains three types of records as explained below.

2.3 Assembler output format - Header, Text and End Records

The object program 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). So a maximum of 30 bytes can be stored in each text record.

- **End record**

Col.1 E

Col.2 7 Address of first executable instruction in object program (hex). (”^” is only for separation only)

```
HCOPY  00^1000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^3820644C000005
E^001000
```

Fig 2.2 - Object code for the above example program:

We have two columns per byte for object code. Each machine instruction is 3 bytes that is it occupies 6 columns. In the first text record we are saving 10 machine instructions each of 3 bytes size. So we are storing a total of 30 bytes (60 columns) which is 1E in decimal.(IE marked in a circle in the example given).

2.4 Design of a two pass assembler

2.4.1 Necessity of two passes and Forward reference:

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

Loc	Label	Operator	Operand	
1000	FIRST	STL	RETADR	
1003	CLOOP	JSUB	RDREC	
...	...	...	...	...
1012		J	CLOOP	
...	...	...	...	...
1033	RETADR	RESW	1	

In the above example in line number 1000 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 1033.

To generate the object code for the instruction at 1000 we need the opcode for STL and the value for the symbol RETADR. But the value or address of RETADR is not available until 1033. **This reference of RETADR before it is defined is called forward referencing**.

So generating the object code by scanning the entire program only once becomes difficult. Due to this reason usually the **design is done in two passes**. A two pass assembler resolves the forward references with the help of a SYMBOL TABLE and then converts the program into the object code.

Functions of the two passes of assembler:

Pass 1 (Define symbols)

1. Assign addresses to all statements in the program.
2. Save the addresses assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives.

Pass 2 (Assemble instructions and generate object programs)

1. Assemble instructions (translating operation codes and looking up addresses).
2. Generate data values defined by BYTE,WORD etc.
3. Perform processing of assembler directives not done in Pass 1.
4. Write the object program and the assembly listing.

2.4.2 Data Structures Used

The data structures used in the design of 2 pass algorithm are:

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)
- Location Counter(LOCCTR)

Operation Code Table (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 and to find the instruction length for incrementing LOCCTR. In pass 2, it is used to translate the operation codes to machine language.

(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.)

Symbol Table (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. A sample SYMTAB is shown below.

COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039

Location Counter (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.

(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.)

2.4.3 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)
                end(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}
```

Explanation of Pass 1 Algorithm:

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 then written to the intermediate file. 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.

Next it 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 assembler 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 (each word is of size 3bytes so 3*no of words). If it is BYTE it adds the length of the constant to the LOCCTR, if RESB it adds number of bytes reserved. If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR minus the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

2.4.2 The Algorithm for Pass 2

```
Begin
  read 1st 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 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
                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 doesnot 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{while NOT END}
 Write last Text reocrd to object program
 write End record to object program
 write last listing line
End {Pass 2}
```

Explanation of Pass 2 Algorithm:

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the listing file(output 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. OPTAB is searched to find the object code of an opcode. If there is a symbol in the operand field, the symbol table is searched to get the address value for this which gets appended to the object code of the opcode. If the address is not found then zero value is stored as operand's 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.

Machine Dependent Assembler Features

The features which are closely related(dependent) to machine architecture are called machine dependent assembler features.The machine dependent assembler features includes

1. **Instruction Formats and Addressing Modes**
2. **Program Relocation**

Instruction Formats and Addressing Modes

Study the instruction formats and addressing modes of SIC/XE from first module.

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 address is not known until the load time.

In an absolute program the starting address to which the program has to be loaded is mentioned in the program itself using the START directive. So the address of every instruction and labels are known while assembling itself. This is called **absolute addressing.** Consider an example

```
55      101B    LDA     THREE
::
75      102D    THREE   RESW    1
```

This statement says that the register A is loaded with the value stored at location 102D(which is the address of THREE). Suppose we need to load and execute the program at location 3000 instead of location 1000. Since program is loaded into location 3000, at address 102D (address of THREE) the required value which needs to be loaded in the

register A is no more available. The address of the symbols also get 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 3000.

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

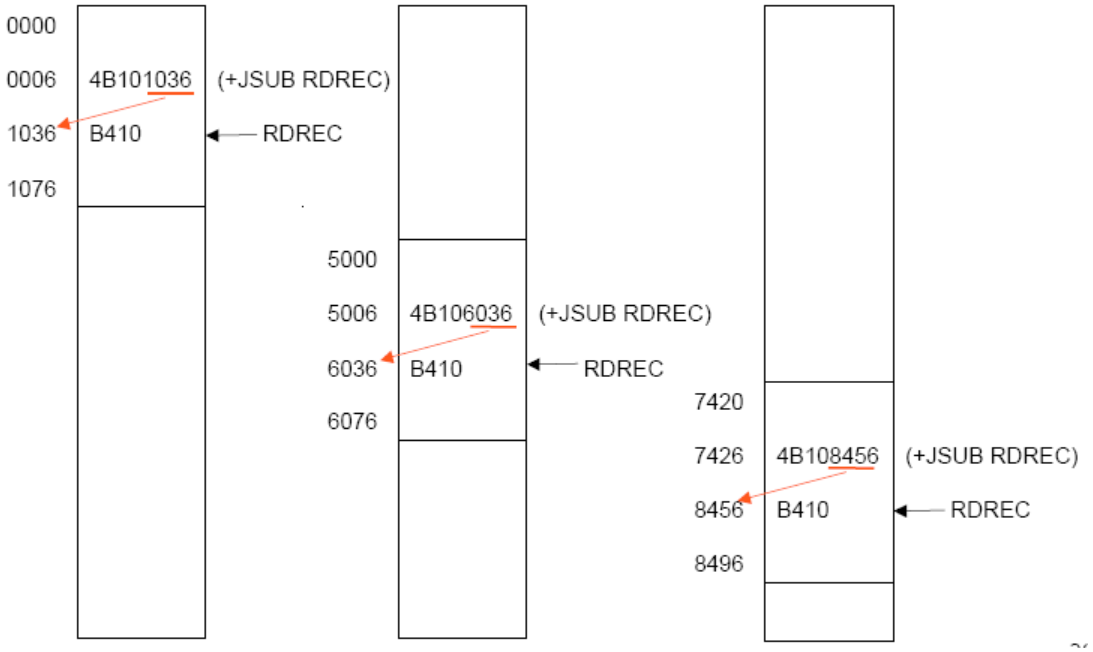
Calculate based on the starting address 1000

Reload the program starting at 3000

```
55     101B          LDA    THREE        00302D
```

The absolute address should be modified

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 can identifies and informs 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.

It is not possible for the loader to distinguish the address and constant from the object program. So the assembler must keep some information to tell the loader which part of the object program need to be modified. For this the concept of **modification record** is record.

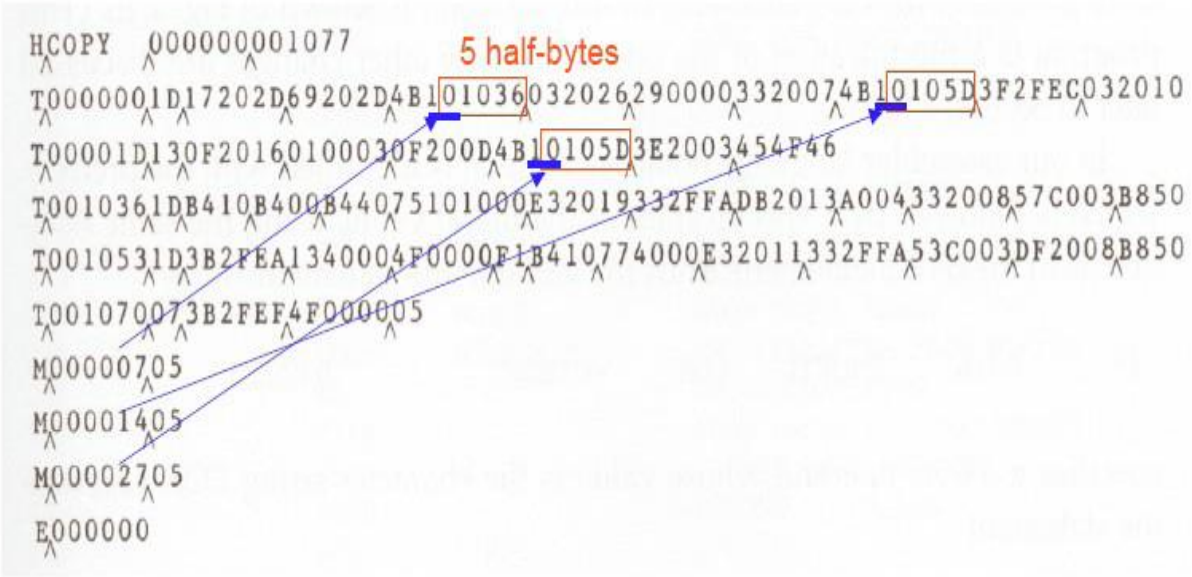
Modification record is a type of record which is added to the object program. One modification record is created for each address to be modified. The assembler produces a modification record to store the starting location and the length of the address field to be modified.

The Modification record 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)

- 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.

Example for a relocatable object program:



- The object code lines at the end starting with M 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 for the remaining modification records.