MODULE -2

ASSEMBLERS-1

2.1 Basic Assembler Functions:

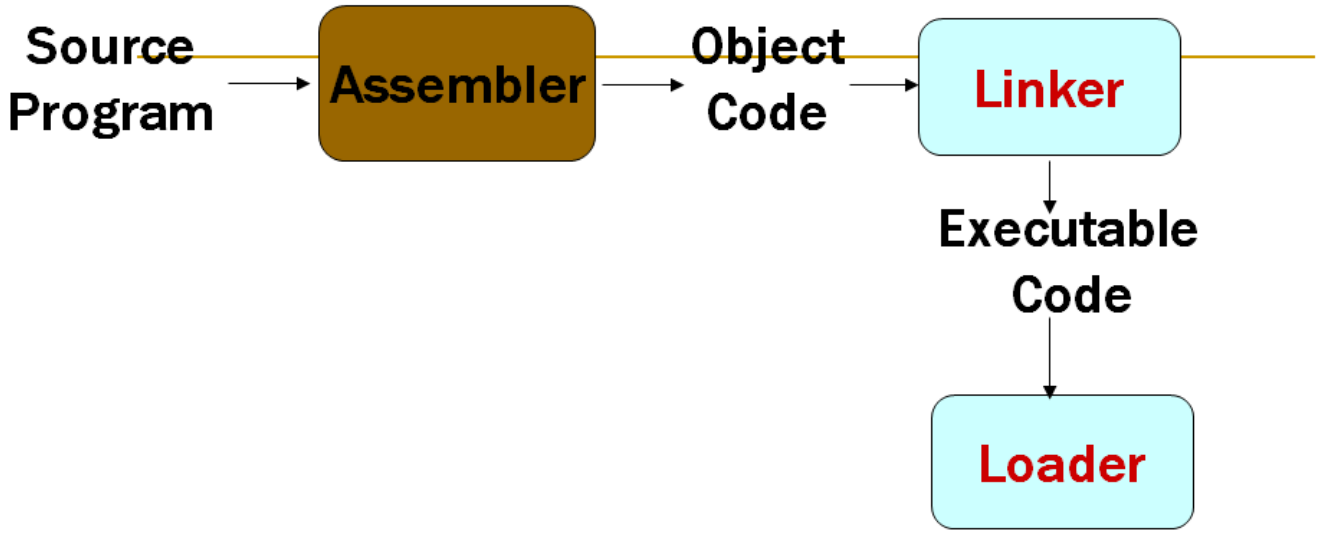


Figure 1

- Figure 2 shows SIC program which contains a main routine that reads records from an input device (F1) and copies that to an output device (05) . This main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from the buffer to the output device. Each subroutine must transfer one byte at a time. The end of each record is marked with a null character(hexa decimal 00). The end of the file to be copied is indicated by a zero length record. When the end of the file is detected the program writes EOF on the output device. And terminates by executing RSUB instruction and returns to the OS. Length of the buffer is 4096 bytes.

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 2.1 Example of a SIC assembler language program.

SIC Assembler Directive:

- In addition to the machine instructions assembler directives are also used in programs. Assembler directives are pseudo instructions. They provide instructions to the assembler itself. They are not translated into machine code.

START – Specify name and starting address for the program.
END – Indicate the end o the source program and(optionally) specify the first executable instruction in the program.
BYTE – Generate character or hexadecimal constant , occupying as many bytes as needed to represent the constant.
WORD- Generate one word integer constant.
RESB- Reserve the indicated number of bytes for a data area.
RESW- Reserve the indicated number of words for a data area.

A Simple SIC Assembler

- Figure 3 shows the same program as in figure 2 with the generated object code for each statement.

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 2.2 Program from Fig. 2.1 with object code.

- The translation of source program to object code requires to accomplish the following **basic functions:**

1. Convert mnemonic operation codes to their machine language equivalents. Eg: translate STL to 14.
2. Convert symbolic operands to their equivalent machine addresses. Eg: translate RETADR to 1033
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
5. Write the object program and assembly listing.

- All these functions except the second one can be easily accomplished by sequential processing of the source program, one line at a time.
- Consider the following:

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

The instruction(line 10) contains a forward reference, that is a reference to a label that is defined later. So can not process the statement . So most of the assemblers makes two passes. The first pass scans the program for labels and assign addresses. The second pass performs the actual translation.

- The assembler must process assembler directives. They are not translated into machine language. But they provide instructions to assembler itself.
- Finally the assembler must write the generated object code to some output device. The object program will later be loaded into memory for execution.

Object Program format

- 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)

- Figure 2.3 shows the object program corresponding to figure 2.2. The ^symbol is used to separate the fields.

```
HCOPY ^001000^00107A
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^002073^07^3820^644C000005
E^001000
```

Figure 2.3 Object program corresponding to Fig. 2.2.

- 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 some processing of assembler directives, including those for address assignment, such as BYTE and RESW etc.
    - 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

Assembler 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.

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:

- 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.

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}
```

- 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 the length of the constant in bytes 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:

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}
```

- 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 attached 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.

Machine-Dependent Assembler Features:

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

- Instruction formats and addressing modes
- Program relocation.

Instruction formats and Addressing Modes

1. **Translation of Register to Register instructions**

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

   Eg:

   COMPR A, S A004

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

2. **Translation of Format 4 instructions**

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

   Eg:

   CLOOP +JSUB RDREC 4B101036

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

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

3. **Translation PC relative instructions**

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

disp = TA –[PC]

Eg:1

0000 FIRST STL RETADR 17202D

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

☞ displacement= RETADR - PC = 30-3 = 2D

Eg: 2

0017 J CLOOP 3F2FEC

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

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

4. **Translation of Base relative instructions**

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

disp = TA –[B]

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

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

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

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

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

displacement= BUFFER - B = 0036 - 0033 = 3

5. **Translation of Immediate addressing**

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

Eg:

- 0020 LDA #3 010003

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

- 103C +LDT #4096 75101000

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

6. **Translation involving indirect addressing**

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

Eg:

002A J @RETADR 3E2003

$(3C)_{16}$ 1 0 0 0 1 0 $(003)_{16}$

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

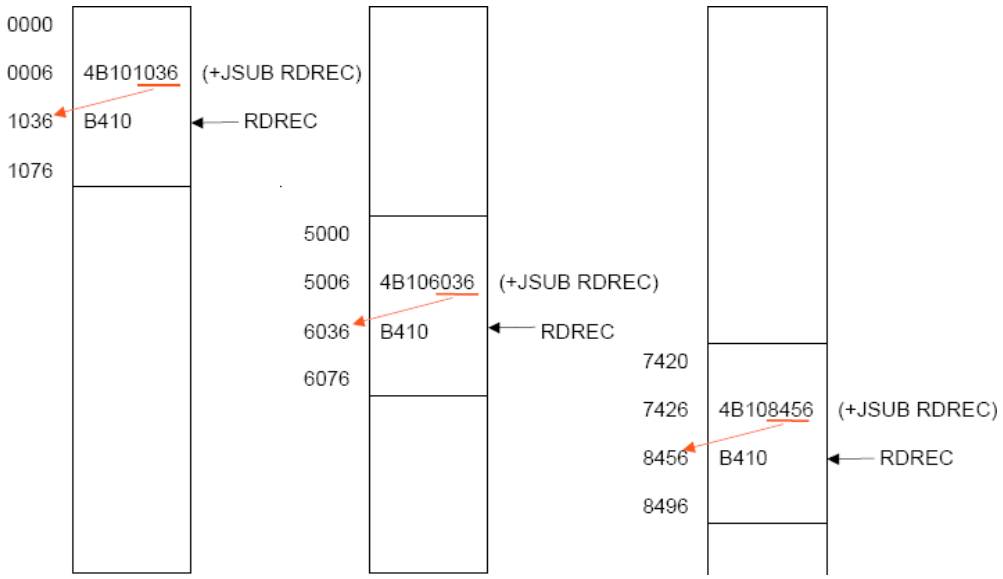
Program Relocation

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

  Eg: Consider the instruction:

  101B LDA THREE 00102D

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



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

Modification record

Col. 1 M

Col. 2-7 Starting location of the address field to be modified, relative to

the beginning of the program (Hex)

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

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

Eg: Consider the instruction

CLOOP +JSUB RDREC 4B101036

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

M00000705

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