Module-1 *Introduction to System Software, Machine Architecture of SIC and SIC/XE. Assemblers: Basic assembler functions, machine dependent assembler features, machine independent assembler features, assembler design options. Basic Loader Functions*

Text book 1: Chapter 1: 1.1,1.2,1.3.1,1.3.2, Chapter2 : 2.1 to 2.4, Chapter 3 ,3.1

RBT: L1, L2, L3

MACHINE ARCHITECTURE

The Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer.

Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

Difference between System Software and Application Software

System Software	Application Software
System Software intended to support the operation and use of computer	Application Software is primarily concerned with the solution of some problem using computer as a tool
Related to Machine Architecture	Not related to machine architecture
Machine Dependent Example: Compilers, Assemblers, OS etc	Machine Independent Example: Payroll System, Games etc

The Simplified Instructional Computer (SIC):

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. There are two versions of SIC, they are, standard model (SIC), and, extension version (SIC/XE) (extra equipment or extra expensive).

SIC Machine Architecture:

We discuss here the SIC machine architecture with respect to its Memory and Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output

• **Memory:** There are a total of 32,768(215)bytes in the computer memory. It uses Little Endian format to store the numbers, 3 consecutive bytes form a word, and each location in memory contains 8-bit bytes.

•**Registers:** There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

•**Data Formats:** Integers are stored as 24-bit binary numbers. 2's complement representation is used for negative values; characters are stored using their 8-bit ASCII codes. No floating-point hardware on the standard version of SIC.

•**Instruction Formats:** All machine instructions on the standard version of SIC have the 24-bit format as shown above

Opcode	x	Address

•**Addressing Modes**:

Mode	Indication	Target address calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x)

There are two addressing modes available, which are as shown in the above table.
Parentheses are used to indicate the contents of a register or a memory location.

•**Instruction Set :**

1. SIC provides, load and store instructions (LDA, LDX, STA, STX, etc.). Integer arithmetic operations: (ADD, SUB, MUL, DIV, etc.).

2. All arithmetic operations involve registers A and a word in memory, with the result being left in the register. Two instructions are provided for subroutine linkage.
3. COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to indicate the result. There are conditional jump instructions: (JLT, JEQ, JGT), these instructions test the setting of CC and jump accordingly.
4. JSUB jumps to the subroutine placing the return address in register L, RSUB returns by jumping to the address contained in register L.

•**Input and Output**:
Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

Data movement and Storage Definition
LDA, STA, LDL, STL, LDX, STX (A- Accumulator, L – Linkage Register, X – Index Register), all uses3-byte word. LDCH, STCH associated with characters uses 1-byte. There are no memory-memory move instructions.
Storage definitions are
- WORD - ONE-WORD CONSTANT
- RESW - ONE-WORD VARIABLE
- BYTE - ONE-BYTE CONSTANT
- RESB - ONE-BYTE VARIABLE

Example Programs (SIC):

<u>Example 1: Simple data and character movement operation</u>
To store the value 5 in a variable ALPHA and character Z in a variable C1

```
          LDA FIVE
          STA ALPHA
          LDCH    CHARZ
          STCH    C1
ALPHA     RESW    1
FIVE      WORD    5
CHARZ     BYTE    C'Z'
C1        RESB    1
```

Example 2: Arithmetic operations

```
        BETA=ALPHA+INCR+1
        LDA         ALPHA
        ADD         INCR
        SUB         ONE
        STA         BETA
……..

……..
……..
        ONE         WORD        1
        ALPHA       RESW        1
        BEETA       RESW        1
        INCR        RESW        1
```

Example 3: Looping and Indexing operation

To perform STR2=STR1 where STR1 is a string of 11 characters.

```
LDX     ZERO                    ;       X = 0
MOVECH      LDCH STR1, X        ;       LOAD A FROM STR1
STCH    STR2, X                 ;       STORE A TO STR2
TIX     ELEVEN                  ;       ADD 1 TO X, TEST
JLT     MOVECH
.

.

.

STR1        BYTE  C 'HELLO WORLD'
STR2        RESB        11
ZERO        WORD        0
ELEVEN      WORD        11
```

Example 4: Input and Output operation

To read a character from the input device and to write a character to the output device.

```
INLOOP      TD     INDEV       : TEST INPUT DEVICE

            JEQ    INLOOP      : LOOP UNTIL DEVICE IS READY
            RD     INDEV       : READ ONE BYTE INTO A
            STCH DATA          : STORE A TO DATA
              .
              .
OUTLOOP     TD     OUTDEV      : TEST OUTPUT DEVICE

            JEQ    OUTLP       : LOOP UNTIL DEVICE IS READY

            LDCH    DATA       : LOAD DATA INTO A

            WD  OUTDEV         : WRITE A TO OUTPUT DEVICE

            .
            .
INDEV       BYTE  X ‘F5’ : INPUT DEVICE NUMBER
OUTDEV      BYTE  X ‘08’ : OUTPUT DEVICE NUMBER
```

Example 5: To transfer two hundred bytes of data from input device to memory

```
            LDX    ZERO
CLOOP       TD     INDEV
            JEQ    CLOOP

            RD     INDEV
            STCH RECORD, X
            TIX    B200
            JLT    CLOOP

            .
```

.

```
INDEV       BYTE      X 'F5'
RECORD      RESB       200
ZERO        WORD      0
B200        WORD      200
```

SIC/XE Machine Architecture:

• **Memory**: Maximum memory available on a SIC/XE system is 1 Megabyte (220 bytes).

• **Registers** : Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC.

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

• **Data Formats**: There is a 48-bit floating-point data type, $F*2^{(e-1024)}$

1	11	36
s	exponent	fraction

• **Instruction Formats**: The new set of instruction formats fro SIC/XE machine architecture are as follows.

• *Format 1 (1 byte):* contains only operation code (straight from table).

8
op

•*Format 2 (2 bytes):* first eight bits for operation code, next four for register 1 and following four for register 2. The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).

8	**4**	**4**
op	**r1**	**r2**

• *Format 3 (3 bytes):* First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

• *Format 4 (4 bytes):* same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

•Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

1. Direct (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.
2. Relative (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)
3. Immediate(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)
4. Indirect(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.
5. Indexed (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings e - > e = 0 means format 3, e = 1 means format 4.

•Instruction Set:

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating- point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction: RMO
Register-to-register arithmetic operations, ADDR, SUBR, MULR, DIVR and, Supervisor call instruction : SVC.

•Input and Output:

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

Example Programs (SIC/XE)

Example 1: Simple data and character movement operation

To store the value 5 in a variable ALPHA and character Z in a variable C1

```
                LDA    #5
                STA    ALPHA
                LDA    #90
                STCH   C1
                  .
                  .
ALPHA           RESW   1
C1              RESB   1
```

Example 2:   Arithmetic operations

```
        BETA=ALPHA+INCR+1
        LDS     INCR
        LDA     ALPHA
        ADDR S,A
        SUB     1
        STA     BETA
        .
        .
        ALPHA           RESW            1
        INCR            RESW            1
        BETA            RESW            1
```

Example 3: Looping and Indexing operation

To perform STR2=STR1 where STR1 is a string of 11 characters.

```
                LDT             #11
                LDX             #0
MOVECH          LDCH            STR1, X         :       LOAD A FROM STR1
                STCH            STR2, X         :       STORE A TO STR2
                TIXR            T               :       ADD 1 TO X, TEST (T)
                JLT     MOVECH
                ……….

STR1            BYTE  C ‘HELLO WORLD’
STR2            RESB  11
```

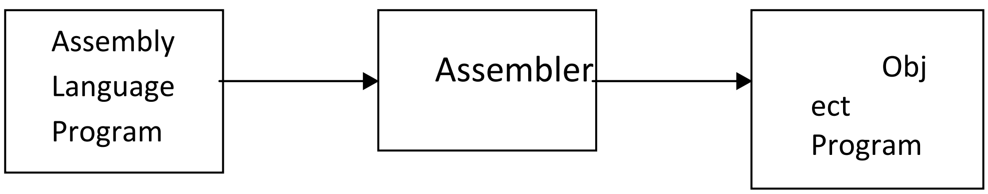
Difference between SIC and SIC/XE

	SIC	SIC/XE
Memory	2^{15} bytes	2^{20} bytes
Registers	5 (A,X,L,PC & SW)	9(A,X,L,B,S,T,F,PC & SW)
Data Formats	No Floating Point Hardware	Supports Floating Point Hardware
Instruction Format	One	Four
Addressing Mode	Two	Five and its combination

ASSEMBLERS

The basic assembler functions are:

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



SIC Assembler Directive:

- START: Specify name and starting address for the program
- END: Indicate End of the program and (optionally) specify the first execution 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

The design of assembler in other words:

1. Convert mnemonic operation codes to their machine language equivalents.
   Example: Translate LDA to 00.
2. Convert symbolic operands to their equivalent machine addresses.
   Example: Translate GAMMA to 400F
3. Build the machine instructions in the proper format.
4. Convert the data constants to internal machine representations.
   Example: ONE WORD 1 to 000001
5. Write the object program and the assembly listing

Two Pass Assembler

Pass-1

- Assign addresses to all the statements in the program
- 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.

Figure 2.1: Assembly Language Program with object code Object Code for Instruction DELTA=GAMMA + INCR – 1

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	ARTH	START	4000	
4000		LDA	GAMMA	00400F
4003		ADD	INCR	184012
4006		SUB	ONE	1C4015
4009		STA	DELTA	0C400C
400C	DELTA	RESW	1	
400F	GAMMA	RESW	1	
4012	INCR	RESW	1	
4015	ONE	WORD	1	000001
4018		END		

OPTAB	
MNEMONIC	**OPCODE**
LDA	00

ADD	18
SUB	1C
STA	0C

SYMTAB	
LABEL	**ADDRESS**
DELTA	400C
GAMMA	400F
INCR	4012
ONE	4015

LDA GAMMA

Opcode	X	Address
0000 0000	0	100 0000 0000 1111

0 0 4 0 0 F

OBJECT PROGRAM

- 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.

Syntax

- 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)

HARTH 004000000018
T0040000C00400F1840121C40150C400C
T00401503000001
E004000

Fig 2.2 Object program corresponding to Fig 2.1

Write the object program for the ALP given below

STR2 = STR1 where STR1="HELLO"

LOCCTR	**SOURCE STATEMENT**			**OBJECT CODE**
	COPY	START	2000	
2000		LDX	ZERO	042019
2003	MOVECH	LDCH	STR1,X	50A00F
2006		STCH	STR2,X	54A014
2009		TIX	FIVE	2C201C
200C		JLT	MOVECH	382003
200F	STR1	BYTE	C'HELLO'	48454C4C4F
2014	STR2	RESB	5	
2019	ZERO	WORD	0	000000
201C	FIVE	WORD	5	000005
201F		END		

OPTAB	
MNEMONIC	**OPCODE**
LDX	04
LDCH	50
STCH	54
TIX	2C
JLT	38

SYMTAB	
LABEL	**ADDRESS**
MOVECH	2003
STR1	200F
STR2	2014
ZERO	2019
FIVE	201C

Object Code for the instruction

LDCH STR1,X

Opcode	X	Address
0101 0000	1	010 0000 0000 1111
5 0		A 0 0 F

HCOPY 00200000001F
T0020001404201950A00F54A0142C201C38200348454C4C4F
T00201906000000000005
E0020000

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:

LOCCTR is an 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 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}
```

- 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
                  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 doesn't fit into 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 ;ast listing linbe
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 assembled and when the END directive is encountered, the End record is written.

Generate the complete object program for the following assembly level program

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	69101788
		BASE	TOTAL	
0009	LOOP	ADD	TABLE,X	1BA00C
000C		TIX	COUNT	2F2006
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015	COUNT	RESW	1	
0018	TABLE	RESW	2000	
1788	TOTAL	RESW	1	
178B		END	FIRST	

Program Length= LOCCTR – STARTING ADDRESS=178B-0=178BH

OPTAB	
MNEMONIC	**OPCODE**
LDA	00
LDB	68
ADD	18
TIX	2C
JLT	38
STA	0C
CLEAR	B4

SYMTAB	
LABEL	**ADDRESS**
FIRST	0000
LOOP	0009
COUNT	0015
TABLE	0018
TOTAL	1788

The Object code for the instruction

+LDB #TOTAL

Opcode	N	I	X	B	P	E	Address
0110 10	0	1	0	0	0	1	0000 0001 0111 1000 1000

6 9 1 0 1 7 8 8

STA TOTAL

Opcode	N	I	X	B	P	E	Displacement
0000 11	1	1	0	1	0	0	0000 0000 0000

0 F 4 0 0 0

The instruction cannot be assembled by using Program Counter Relative Addressing Mode because the Displacement what we calculate can not fit into 12 bit displacement. So, Base Relative addressing mode is used.

Displacement = TA – (B)
= 1788-1788=0

Object Program

HSUM 00000000178B

T00000015B410010000691017881BA00C2F20063B2FF70F4000

E000000

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.

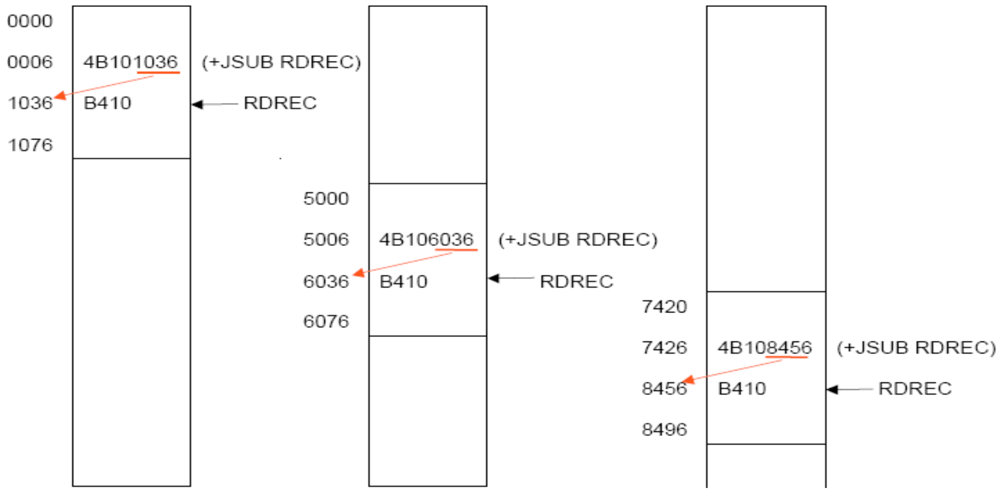


Fig: Examples of Program Relocation

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.

The Modification Record for

+JSUB RDREC

instruction is

M00000705

000007 is the starting location of the address field to be modified by the loader for proper execution of the program.

05 is the length of the address field to be modified , in half bytes.

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.

LOADERS

Introduction

The Source Program written in assembly language or high level language will be translated to object program, which is in the machine language form for execution. This translation is either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

- **Loading** - which allocates memory location and brings the object program into memory for execution-(Loader)
- **Linking**-which combines two or more separate object programs and supplies the information needed to allow references between them -(Linker)
- **Relocation**-which modifies the object program so that it can be loaded at an address different from the location originally specified-(Linking Loader)

Basic Loader Functions:

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 4.1. The assembler generates the object program and later loaded to the memory by the loader for execution.

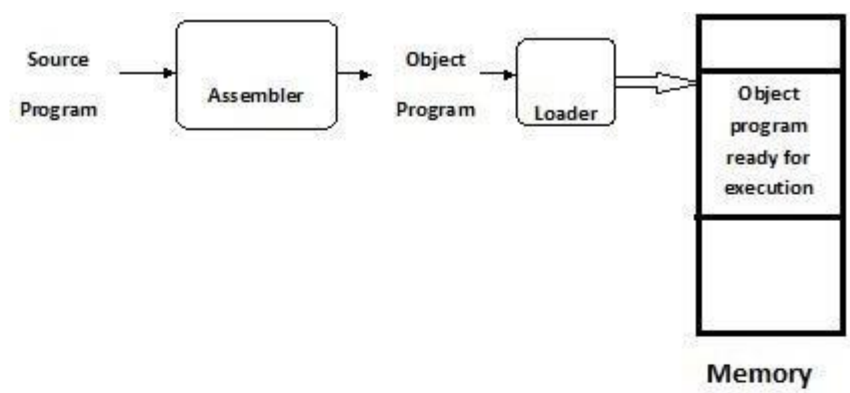


Figure 4 : The Role of Loaderype of Loaders

The different types of loaders are

- Absolute loader
- Bootstrap loader
- Relocating loader (relative loader) and
- Direct linking loader

1. Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure.

The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

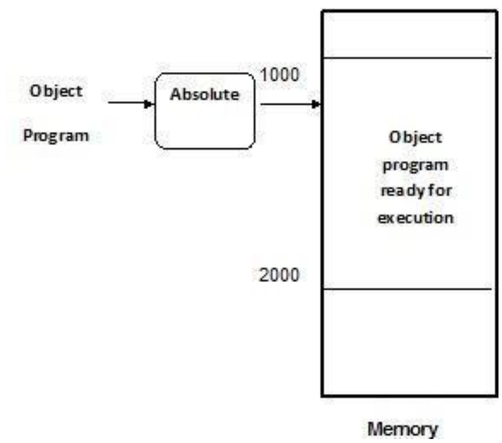


Figure 4.4: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte.

```
Begin
read Header record
verify program name and length
read first Text record
while record type is != 'E' do begin

        {if object code is in character form, convert into internal representation}
        move object code to specified location in
        memory read next object program record
        end
  jump to address specified in End record
end
```

Figure: Algorithm for an absolute loader

Example:

Program to find **SUM=ALPHA + BETA**

LOCC	SOURCE	OBJECT

TR	STATEMENTS			CODE
	ART H	STAR T	1000	
1000		LDA	ALPH A	001009
1003		ADD	BETA	18100C
1006		STA	SUM	0C100F
1009	ALPH A	WOR D	4	000004
100C	BETA	WOR D	2	000002
100F	SUM	RES W	1	
1012		END	ART H	

OPTAB	
MNEMONIC	**OPCODE**
LDA	00
ADD	18
STA	0C

SYMTAB	
LABEL	**ADDRESS**
ALPHA	1009
BETA	100C
SUM	100F

MEMORY ADDRESS	CONTENTS
0000	**xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**
………	… …. .

1000	001009 18100C 0C100F 000004 000002
1012	
1015	

Fig: Program loaded in memory at the address 1000H

OBJECT PROGRAM

HARTH 001000000012

T0010000F00100918100C0C100F000004000002

E001000

Figure: Absolute Object Program for the above source program.

2. A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin

X=0x80 (the address of the next memory location to be loaded

LOOP

A←GETC (and convert it from the ASCII character code to the value of the hexadecimal digit) save the value in the high-order 4 bits of S
A←GETC
combine the value to form one byte A← (A+S)
store the value (in A) to the address in register
X
X←X+1

End

It uses a subroutine GETC, which is

GETC A←read one character

if A=0x04 then jump to 0x80 if A<48 then GETC

A ← A-48 (0x30)
if A<10 then return
A ← A-7 return

3. Relocating Loader

The loader that allow program relocation is called relocating loader.

```
BEGIN

  Get PROGADDRfrom operating system
  while not end of input do
    BEGIN
      read next input record

      while record type != 'E' do

        BEGIN
             read next input record
             while record type = 'T' then
               BEGIN
                    move object code from record to location PROGADDR + specified address.
               End
             While record type = 'M'
                    Add PROGADDR at the location PROGADDR + SPECIFIED ADDRESS.
        END
    END
END
```

Figure: SIC/XE relocation loader algorithm

LOCCTR	SOURCE STATEMENTS			OBJECT CODE
	READ	START	0	
0000		+JSUB	TEST	4B10000C
0004		RD	INPUT	DB2003
0007		STCH	DATA	572001
000A	INPUT	BYTE	X'F1'	F1
000B	DATA	RESB	1	
000C	TEST	TD	INPUT	E32FFB
000F		JEQ	*-3	332FFA
0012		RSUB		4C0000
0015		END	READ	

SYMTAB	

LABEL	ADDRESS
INPUT	000A
DATA	000B
TEST	000C

OPTAB	
MNEMONIC	**OPCODE**
JSUB	48
RD	D8
STCH	54
TD	E0
JEQ	30
RSUB	4C

OBJECT PROGRAM

HREAD 000000000015

T0000000B4B10100CDB2003572001F

1 T00000C09E32FFB332FFA4C0000

M00000105

E000000

MEMORY ADDRESS	CONTENTS
0000	**xxxxxxxxxxxxxxxxxxxxxxxxxxx**
………	……..
1000	4B10100C DB2003 572001 F1
100C	E32FFB 332FFA 4C0000
1015	

Fig: Program loaded in memory at the address 1000H because there is no space at address 0000H.

Relocating loader add 1000H at the location 1000H + 000001.