UNIT - 1

MACHINE ARCHITECTURE

1.1 Introduction:

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.

1.2 System Software and Machine Architecture:

One characteristic in which most system software differs from application software is machine dependency.

System software supports operation and use of computer. Application software provides solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly,

Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

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:

Opcode(8) x Address (15)

All machine instructions on the standard version of SIC have the 24-bit format as shown above

- 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 register 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

1.3.2Example Programs (SIC):

Example 1: Simple data and character movement operation

```
        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

```
        LDA  ALPHA
        ADD  INCR
        SUB  ONE
        STA  BETA
        ……..
        ……..
        ……..
ONE     WORD  1
ALPHA   RESW  1
BEETA   RESW  1
INCR    RESW  1
```

1.3.3SIC/XE Machine Architecture:

- Memory

Maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} 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)

- Floating-point data type:

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

1	11	36
s	exponent	fraction

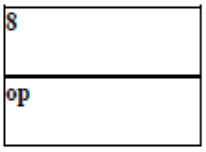
- Instruction Formats:

The new set of instruction formats fro SIC/XE machine architecture are as follows.

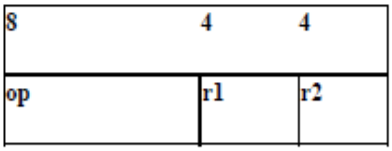
- Format 1 (1 byte): contains only operation code (straight from table).

- 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).
- 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).
- 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.

Format 1 (1 byte)

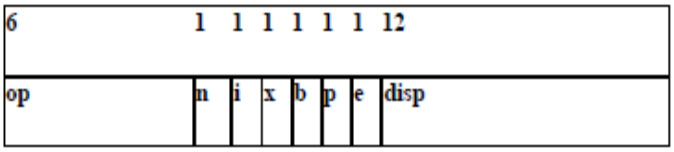


Format 2 (2 bytes)

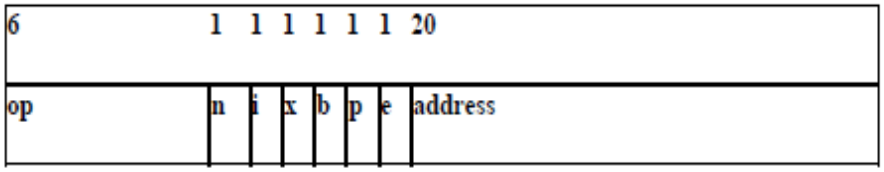


Formats 1 and 2 are instructions do not reference memory at all

Format 3 (3 bytes)



Format 4 (4 bytes)



- 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

Bits x,b,p : Used to calculate the target address using relative, direct, and indexed addressing Modes.

Bits i and n: Says, how to use the target address

b and p - both set to 0, disp field from format 3 instruction is taken to be the target address. For a format 4 bits b and p are normally set to 0, 20 bit address is the target address

x - x is set to 1, X register value is added for target address calculation

i=1, n=0 Immediate addressing, **TA**: TA is used as the operand value, no memory reference

i=0, n=1 Indirect addressing, **((TA))**: The word at the TA is fetched. Value of TA is taken as the address of the operand value

i=0, n=0 or i=1, n=1 Simple addressing, **(TA)**:TA is taken as the address of the operand value

Two new relative addressing modes are available for use with instructions assembled using format 3.

Mode	**Indication**	**Target address calculation**
Base relative	b=1,p=0	TA=(B)+ disp (0≤disp ≤4095)
Program-counter relative	b=0,p=1	TA=(PC)+ disp (-2048≤disp ≤2047)

- 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

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

Example 2: Arithmetic operations

```
        LDS   INCR
        LDA   ALPHA
```

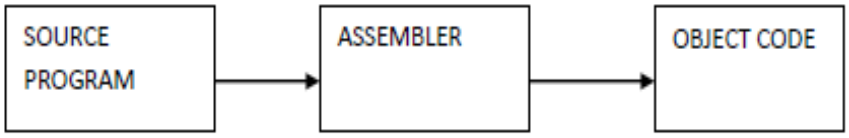
CHAPTER -2

ASSEMBLERS-1

2.1 Basic Assembler Functions:

The basic assembler functions are:

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



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

- SIC Assembler Directive:
  - START: Specify name & starting address.
  - END: End of the program, specify the first execution instruction.
  - BYTE, WORD, RESB, RESW
  - End of record: a null char(00)

End of file: a zero length record

- The design of assembler in other words:
  - Convert mnemonic operation codes to their machine language equivalents

- Convert symbolic operands to their equivalent machine addresses
- Decide the proper instruction format Convert the data constants to internal machine representations
- Write the object program and the assembly listing

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

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

Single-pass Assembler:

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

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

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

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

Pass-1

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

Pass-2

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

Assembler Design:

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

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

Example Program:

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

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

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

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

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

- Header record
  - Col. 1 H
  - Col. 2~7 Program name
  - Col. 8~13 Starting address of object program (hex)
  - Col. 14~19 Length of object program in bytes (hex)
- Text record
  - Col. 1 T
  - Col. 2~7 Starting address for object code in this record (hex)

- Col. 8~9 Length of object code in this record in bytes (hex)

- Col. 10~69 Object code, represented in hex (2 col. per byte)

- End record
  - Col.1 E

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

2.1.1 Simple SIC Assembler

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

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

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

10 1000 LDA ALPHA 00-----

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

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

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

The format of each record is as given below.

Header record:

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

Text record:

Col. 1	T
Col 2-7.	Starting address for object code in this record (hexadecimal)
Col 8-9	Length off object code in this record in bytes (hexadecimal)
Col 10-69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

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

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

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

2.1.2. Algorithms and Data structure

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

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

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

LOCCTR:

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

The Algorithm for Pass 1:

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

```
else
 initialize LOCCTR to 0
 While OPCODE != 'END' do
  begin
    if this is not a comment line then
      begin
        if there is a symbol in the LABEL field then
          begin
            search SYMTAB for LABEL
            if found then
              set error flag (duplicate symbol)
            else
              (if symbol)
        search OPTAB for OPCODE
        if found then
          add 3 (instr length) to LOCCTR
        else if OPCODE = 'WORD' then
          add 3 to LOCCTR
        else if OPCODE = 'RESW' then
      add 3 * #[OPERAND] to LOCCTR
        else if OPCODE = 'RESB' then
```

```
        add #[OPERAND] to LOCCTR
            else if OPCODE = 'BYTE' then
        begin
                find length of constant in bytes
                add length to LOCCTR
         end
            else
        set error flag (invalid operation code)
        end (if not a comment)
        write line to intermediate file
         read next input line
    end { while not END}
    write last line to intermediate file
    Save (LOCCTR – starting address) as program length
End {pass 1}
```

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

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

The Algorithm for Pass 2:

```
Begin
 read 1st input  line
  if OPCODE =  'START' then
   begin
     write listing line
      read next input line
    end
  write Header record to object program
  initialize 1st Text record
while OPCODE != 'END' do
   begin
     if this is not comment line then
       begin
```

```
search OPTAB for OPCODE
if found then
begin
if there is a symbol in OPERAND field then
begin
search SYMTAB for OPERAND field then
if found then
begin
store symbol value as operand address
else
begin
store 0 as operand address
set error flag (undefined symbol)
end
end (if symbol)
else store 0 as operand address
assemble the object code instruction
else if OPCODE = 'BYTE' or 'WORD" then
convert constant to object code
if object code doesn't fit into current Text record then
begin
```

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

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

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.

If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

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

Design and Implementation Issues

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

2.2. Machine-Dependent Assembler Features:

- Instruction formats and addressing modes
- Program relocation.

2.2.1 .Instruction formats and Addressing Modes

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

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

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

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

During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2. Translation involving Register-Memory instructions:

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

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

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

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

Program-Counter Relative:

In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value.

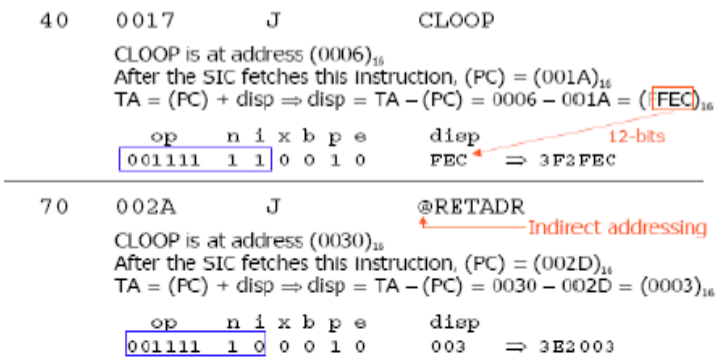
The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction.

This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:

```
10    0000        FIRST       STL         RETADR
```

RETADR is at address $(0030)_{16}$
After the SIC fetches this instruction, (PC) = $(0003)_{16}$
TA = (PC) + disp $\Rightarrow$ disp = TA – (PC) = 0030 – 0003 = $(02D)_{16}$

```
   op     n i x b p e      disp
000101    1 1 0 0 1 0      02D    ⇒ 17202D
```



Base-Relative Addressing Mode:

In this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

- This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE.
- The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.
- When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

LDB #LENGTH (*instruction*)

BASE LENGTH (*directive*)

:

NOBASE

For example:

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

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

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

BUFFER is at location $(0036)_{16}$

$(B) = (0033)_{16}$

$disp = 0036 - 0033 = (0003)_{16}$

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

```
20    000A            LDA       LENGTH          032026
```

: :

175 1056 EXIT STX LENGTH 134000

Consider Line 175. If we use PC-relative

Disp = TA – (PC) = 0033 –1059 = EFDA

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

Immediate Addressing Mode

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

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

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

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

```
12    0003        LDB         #LENGTH

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

Indirect and PC-relative mode:

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

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

RETADR is at address 0030

TA = (PC) + disp ⇒ disp = 0030 – 002D = $(0003)_{16}$

op	n	i	x	b	p	e	disp	
001111	1	0	0	0	1	0	003	⇒ 3E2003

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

2.2.2Program Relocation

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

Absolute Program

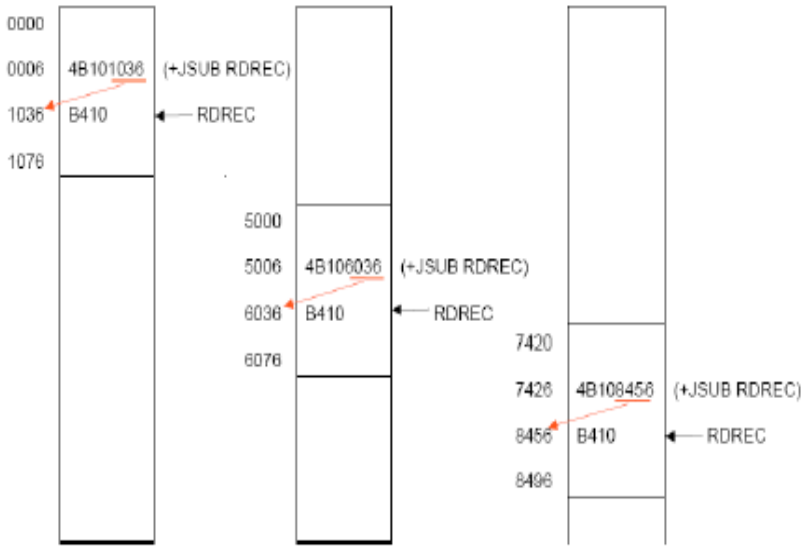
In this the address is mentioned during assembling itself. This is called *Absolute Assembly*. Consider the instruction:

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

- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.
- Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement

of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.

- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification.
- An object program that has the information necessary to perform this kind of modification is called the relocatable program.



- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.

- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.
- From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader.The object program that contains the modification record is called a relocatable program.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record*to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

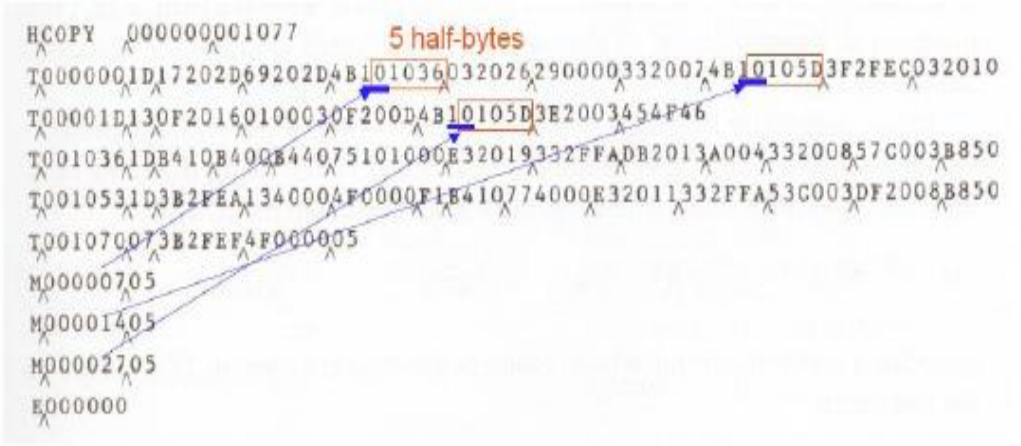
Modification record

Col. 1 M

Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program (Hex)

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

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



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