# Module 2

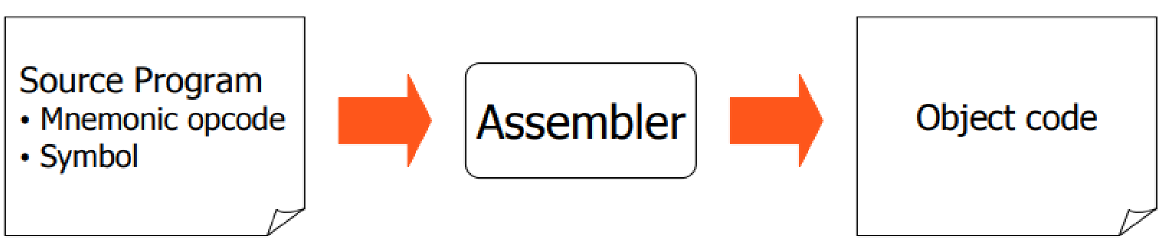
Assemblers

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

## Assemblers

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

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




## Basic Assembler functions

- Convert **symbolic operands** to their equivalent **machine addresses** (eg: RETADR to 1033)

- Convert **mnemonic operation codes** to their **machine language equivalents** (eg: STL to 14)
- Convert the **data constants** specified in the source program into their **internal machine representations** (eg: EOF to 454F46)
- Write the object program and the assembly listing
- Build the machine instruction into proper format

**Assembler Directive**

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



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

   eg: COPY START 1000

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

   eg: END FIRST

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

1. BYTE : Generate **character or hexadecimal constant**, occupying as many bytes as needed to represent the constant
   - Eg: CHARZ BYTE C'Z'

2. WORD : Generate one-word **integer constant**
   - Eg: FIVE WORD 5

3. RESB : Reserve the indicated number of bytes for a data area
   – eg: C1 RESB 1

4. RESW : Reserve the indicated number of words for a data area
   – eg: ALPHA RESW 1

**Example of a SIC Assembler language program**

```
Source statement

COPY     START   1000          COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR        SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC         READ INPUT RECORD
         LDA     LENGTH        TEST FOR EOF (LENGTH = 0)
         COMP    ZERO
         JEQ     ENDFIL        EXIT IF EOF FOUND
         JSUB    WRREC         WRITE OUTPUT RECORD
         J       CLOOP         LOOP
ENDFIL   LDA     EOF           INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     THREE         SET LENGTH = 3
         STA     LENGTH
         JSUB    WRREC         WRITE EOF
         LDL     RETADR        GET RETURN ADDRESS
         RSUB                  RETURN TO CALLER
EOF      BYTE    C'EOF'
THREE    WORD    3
ZERO     WORD    0
RETADR   RESW    1
LENGTH   RESW    1             LENGTH OF RECORD
BUFFER   RESB    4096          4096-BYTE BUFFER AREA
.
.        SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC    LDX     ZERO          CLEAR LOOP COUNTER
         LDA     ZERO          CLEAR A TO ZERO
RLOOP    TD      INPUT         TEST INPUT DEVICE
         JEQ     RLOOP         LOOP UNTIL READY
         RD      INPUT         READ CHARACTER INTO REGISTER A
         COMP    ZERO          TEST FOR END OF RECORD (X'00')
         JEQ     EXIT          EXIT LOOP IF EOR
         STCH    BUFFER,X      STORE CHARACTER IN BUFFER
         TIX     MAXLEN        LOOP UNLESS MAX LENGTH
         JLT     RLOOP           HAS BEEN REACHED
EXIT     STX     LENGTH        SAVE RECORD LENGTH
         RSUB                  RETURN TO CALLER
INPUT    BYTE    X'F1'         CODE FOR INPUT DEVICE
MAXLEN   WORD    4096
.
.        SUBROUTINE TO WRITE RECORD FROM BUFFER
.
WRREC    LDX     ZERO          CLEAR LOOP COUNTER
WLOOP    TD      OUTPUT        TEST OUTPUT DEVICE
         JEQ     WLOOP         LOOP UNTIL READY
         LDCH    BUFFER,X      GET CHARACTER FROM BUFFER
         WD      OUTPUT        WRITE CHARACTER
         TIX     LENGTH        LOOP UNTIL ALL CHARACTERS
         JLT     WLOOP           HAVE BEEN WRITTEN
         RSUB                  RETURN TO CALLER
OUTPUT   BYTE    X'05'         CODE FOR OUTPUT DEVICE
         END     FIRST
```

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

The program contains a main routine that reads records from an input device, identified by device code F 1 and copies them to an output device 05. Main routine calls subroutine RDREC to read a record into buffer and another subroutine WRREC to write the record from the buffer to the output device. Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WD. The buffer is necessary: because the I/O rates for two devices, such as a disk and a slow printing terminal, may be different. The end of each record is marked with a null character (hexadecimal 00).

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

**Forward Reference**

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

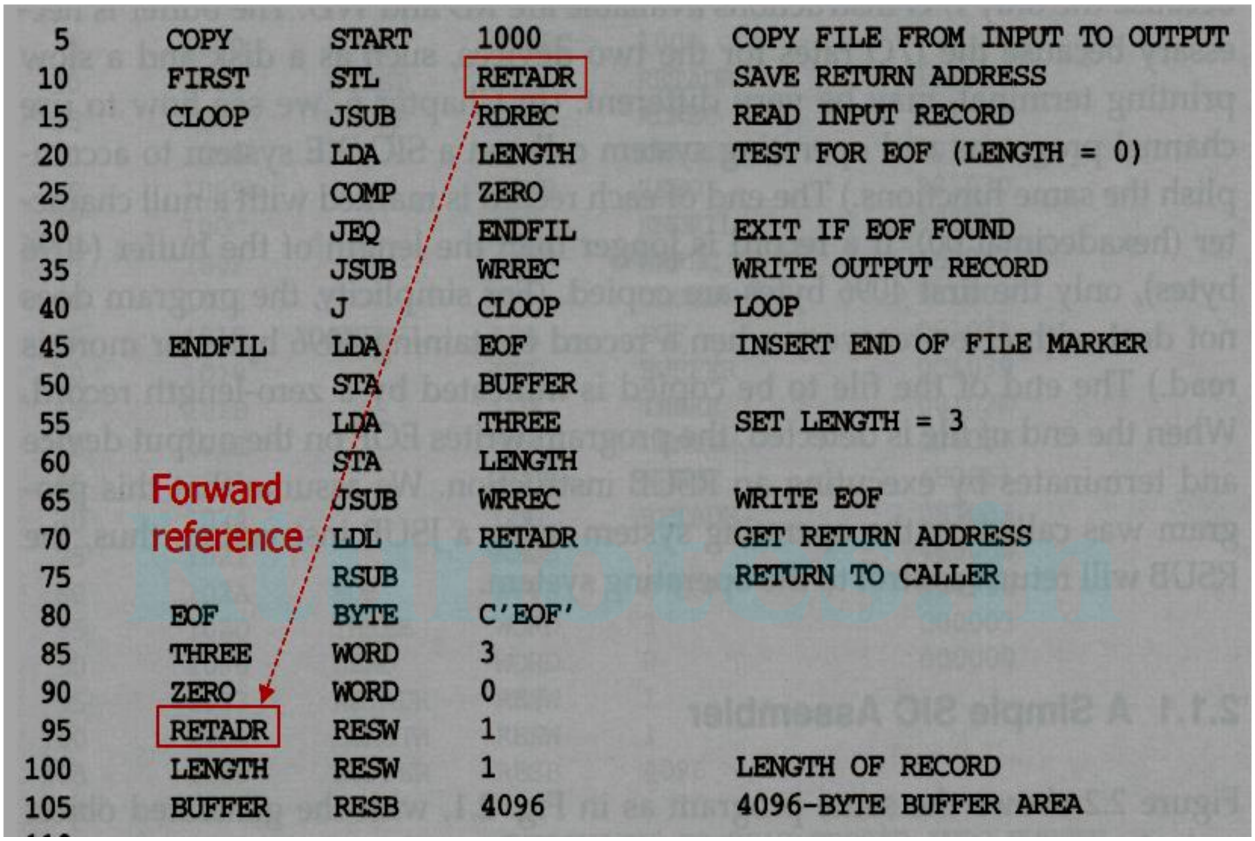
**Forward Reference** – a reference to label (RETADR) that is defined later in the program. If we attempt to translate the program line by line, we will unable to process this statement because we do not know the address that will be assigned to RETADR. Because of this, most assemblers make two passes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

- **PASS 1:**

– Scan the source program for label definitions and assign addresses (such as the Loc column)

- **PASS 2:**

  – Performs the actual translation



```
COPY    START   1000     COPY FILE FROM INPUT TO OUTPUT
FIRST   STL     RETADR   SAVE RETURN ADDRESS
CLOOP   JSUB    RDREC    READ INPUT RECORD
        LDA     LENGTH   TEST FOR EOF (LENGTH = 0)
        COMP    ZERO
        JEQ     ENDFIL   EXIT IF EOF FOUND
        JSUB    WRREC    WRITE OUTPUT RECORD
        J       CLOOP    LOOP
ENDFIL  LDA     EOF      INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     THREE    SET LENGTH = 3
        STA     LENGTH
        JSUB    WRREC    WRITE EOF
        LDL     RETADR   GET RETURN ADDRESS
        RSUB             RETURN TO CALLER
EOF     BYTE    C'EOF'
THREE   WORD    3
ZERO    WORD    0
RETADR  RESW    1
LENGTH  RESW    1        LENGTH OF RECORD
BUFFER  RESB    4096     4096-BYTE BUFFER AREA
```



**Assembler Output Format**

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

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

- **End record** marks the end of the object program and specifies the address in the program where execution is to begin.

- **Header record**

  Col. 1 H

  Col. 2~7 Program name

  Col. 8~13 Starting address of object program (hex)

  Col. 14-19 Length of object program in bytes (hex)

- **Text record**

  Col. 1 T

  Col. 2~7 Starting address for object code in this record (hex)

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

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

- **End record**

  Col.1 E

  Col.2~7 Address of first executable instruction in object program (hex)



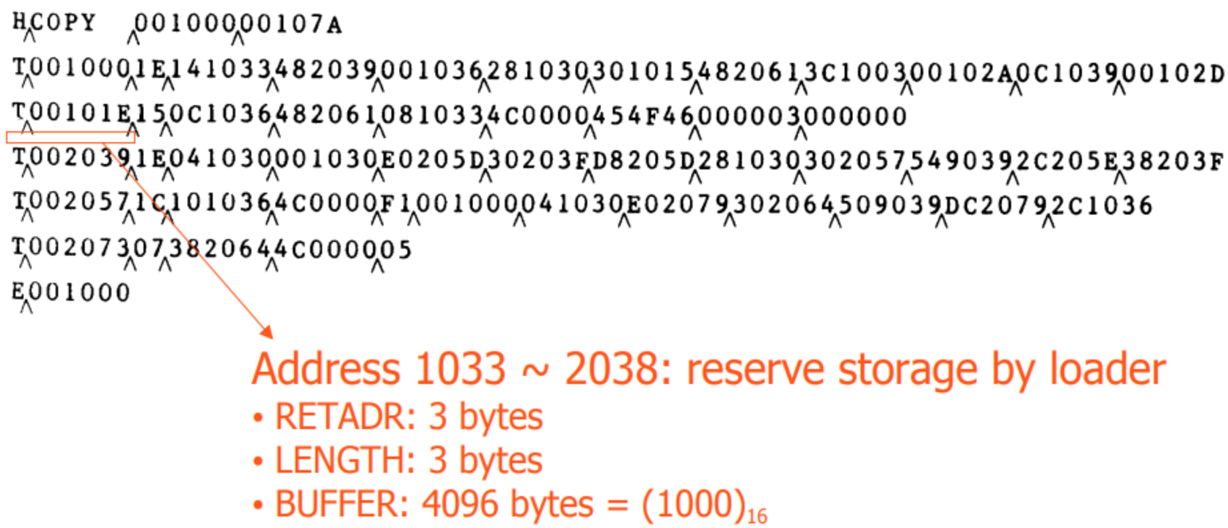
| Line | Loc | Source statement | | | Object code |
|---|---|---|---|---|---|
| 5 | 1000 | COPY | START | 1000 | |
| 10 | 1000 | FIRST | STL | RETADR | 141033 |
| 15 | 1003 | CLOOP | JSUB | RDREC | 482039 |
| 20 | 1006 | | LDA | LENGTH | 001036 |
| 25 | 1009 | | COMP | ZERO | 281030 |
| 30 | 100C | | JEQ | ENDFIL | 301015 |
| 35 | 100F | | JSUB | WRREC | 482061 |
| 40 | 1012 | | J | CLOOP | 3C1003 |
| 45 | 1015 | ENDFIL | LDA | EOF | 00102A |
| 50 | 1018 | | STA | BUFFER | 0C1039 |
| 55 | 101B | | LDA | THREE | 00102D |
| 60 | 101E | | STA | LENGTH | 0C1036 |
| 65 | 1021 | | JSUB | WRREC | 482061 |
| 70 | 1024 | | LDL | RETADR | 081033 |
| 75 | 1027 | | RSUB | | 4C0000 |
| 80 | 102A | EOF | BYTE | C'EOF' | 454F46 |
| 85 | 102D | THREE | WORD | 3 | 000003 |
| 90 | 1030 | ZERO | WORD | 0 | 000000 |
| 95 | 1033 | RETADR | RESW | 1 | |
| 100 | 1036 | LENGTH | RESW | 1 | |
| 105 | 1039 | BUFFER | RESB | 4096 | |
| 110 | | . | | | |

```
                        .
                        .       SUBROUTINE TO READ RECORD INTO BUFFER
                        .
        2039    RDREC   LDX     ZERO                041030
        203C            LDA     ZERO                001030
        203F    RLOOP   TD      INPUT               E0205D
        2042            JEQ     RLOOP               30203F
        2045            RD      INPUT               D8205D
        2048            COMP    ZERO                281030
        204B            JEQ     EXIT                302057
        204E            STCH    BUFFER,X            549039
        2051            TIX     MAXLEN              2C205E
        2054            JLT     RLOOP               38203F
        2057    EXIT    STX     LENGTH              101036
        205A            RSUB                        4C0000
        205D    INPUT   BYTE    X'F1'               F1
        205E    MAXLEN  WORD    4096                001000
```



```
                        .
                        .       SUBROUTINE TO WRITE RECORD FROM BUFFER
                        .
        2061    WRREC   LDX     ZERO                041030
        2064    WLOOP   TD      OUTPUT              E02079
        2067            JEQ     WLOOP               302064
        206A            LDCH    BUFFER,X            509039
        206D            WD      OUTPUT              DC2079
        2070            TIX     LENGTH              2C1036
        2073            JLT     WLOOP               382064
        2076            RSUB                        4C0000
        2079    OUTPUT  BYTE    X'05'               05
                        END     FIRST
```

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




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

**Passes of Assembler**

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

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

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

**Functions of Two Passes of Assembler**

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



**Assembler Data Structures**

Simple Assembler uses two major internal data structures:

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

Also need a variable Location Counter (LOCCTR).

**OPTAB**

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

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

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

**SYMTAB**

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

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

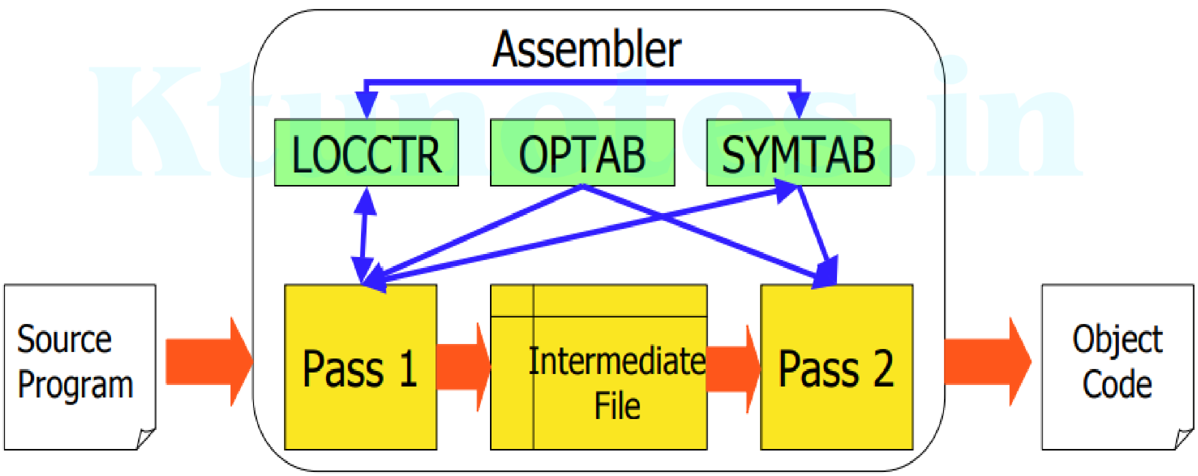
**LOCCTR**

LOCCTR is a variable that is used to help in the assignment of addresses. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to

LOCCTR. Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

**Assembler Algorithm**

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




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

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

LABEL OPCODE OPERAND

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

**Pass 1:**

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

**Pass 2:**

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



### Machine - Dependent Assembler Features

Here considering the design and implementation of an assembler for more complex SIC/XE. So that it is easy to examine the effect of the extended hardware on the structure and functions of assembler. Many real machines have certain architectural features that are similar to those we consider here.

**Source statement**

```
COPY      START   0            COPY FILE FROM INPUT TO OUTPUT
FIRST     STL     RETADR       SAVE RETURN ADDRESS
          LDB     #LENGTH      ESTABLISH BASE REGISTER
          BASE    LENGTH
CLOOP    +JSUB    RDREC        READ INPUT RECORD
          LDA     LENGTH       TEST FOR EOF (LENGTH = 0)
          COMP    #0
          JEQ     ENDFIL       EXIT IF EOF FOUND
         +JSUB    WRREC        WRITE OUTPUT RECORD
          J       CLOOP        LOOP
ENDFIL    LDA     EOF          INSERT END OF FILE MARKER
          STA     BUFFER
          LDA     #3           SET LENGTH = 3
          STA     LENGTH
         +JSUB    WRREC        WRITE EOF
          J       @RETADR      RETURN TO CALLER
EOF       BYTE    C'EOF'
RETADR    RESW    1
LENGTH    RESW    1            LENGTH OF RECORD
BUFFER    RESB    4096         4096-BYTE BUFFER AREA
.
.         SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC     CLEAR   X            CLEAR LOOP COUNTER
          CLEAR   A            CLEAR A TO ZERO
          CLEAR   S            CLEAR S TO ZERO
         +LDT     #4096
RLOOP     TD      INPUT        TEST INPUT DEVICE
          JEQ     RLOOP        LOOP UNTIL READY
          RD      INPUT        READ CHARACTER INTO REGISTER A
          COMPR   A,S          TEST FOR END OF RECORD (X'00')
          JEQ     EXIT         EXIT LOOP IF EOR
          STCH    BUFFER,X     STORE CHARACTER IN BUFFER
          TIXR    T            LOOP UNLESS MAX LENGTH
          JLT     RLOOP          HAS BEEN REACHED
EXIT      STX     LENGTH       SAVE RECORD LENGTH
          RSUB                 RETURN TO CALLER
INPUT     BYTE    X'F1'        CODE FOR INPUT DEVICE
.
.         SUBROUTINE TO WRITE RECORD FROM BUFFER
.
WRREC     CLEAR   X            CLEAR LOOP COUNTER
          LDT     LENGTH
WLOOP     TD      OUTPUT       TEST OUTPUT DEVICE
          JEQ     WLOOP        LOOP UNTIL READY
          LDCH    BUFFER,X     GET CHARACTER FROM BUFFER
          WD      OUTPUT       WRITE CHARACTER
          TIXR    T            LOOP UNTIL ALL CHARACTERS
          JLT     WLOOP          HAVE BEEN WRITTEN
          RSUB                 RETURN TO CALLER
OUTPUT    BYTE    X'05'        CODE FOR OUTPUT DEVICE
          END     FIRST
```

**Figure 2.5 Example of a SIC/XE program.**

| Line | Loc | Source statement | | | Object code |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 17202D |
| 12 | 0003 | | LDB | #LENGTH | 69202D |
| 13 | | | BASE | LENGTH | |
| 15 | 0006 | CLOOP | +JSUB | RDREC | 4B101036 |
| 20 | 000A | | LDA | LENGTH | 032026 |
| 25 | 000D | | COMP | #0 | 290000 |
| 30 | 0010 | | JEQ | ENDFIL | 332007 |
| 35 | 0013 | | +JSUB | WRREC | 4B10105D |
| 40 | 0017 | | J | CLOOP | 3F2FEC |
| 45 | 001A | ENDFIL | LDA | EOF | 032010 |
| 50 | 001D | | STA | BUFFER | 0F2016 |
| 55 | 0020 | | LDA | #3 | 010003 |
| 60 | 0023 | | STA | LENGTH | 0F200D |
| 65 | 0026 | | +JSUB | WRREC | 4B10105D |
| 70 | 002A | | J | @RETADR | 3E2003 |
| 80 | 002D | EOF | BYTE | C'EOF' | 454F46 |
| 95 | 0030 | RETADR | RESW | 1 | |
| 100 | 0033 | LENGTH | RESW | 1 | |
| 105 | 0036 | BUFFER | RESB | 4096 | |
| 110 | | . | | | |
| 115 | | . | SUBROUTINE TO READ RECORD INTO BUFFER | | |
| 120 | | . | | | |
| 125 | 1036 | RDREC | CLEAR | X | B410 |
| 130 | 1038 | | CLEAR | A | B400 |
| 132 | 103A | | CLEAR | S | B440 |
| 133 | 103C | | +LDT | #4096 | 75101000 |
| 135 | 1040 | RLOOP | TD | INPUT | E32019 |
| 140 | 1043 | | JEQ | RLOOP | 332FFA |
| 145 | 1046 | | RD | INPUT | DB2013 |
| 150 | 1049 | | COMPR | A,S | A004 |
| 155 | 104B | | JEQ | EXIT | 332008 |
| 160 | 104E | | STCH | BUFFER,X | 57C003 |
| 165 | 1051 | | TIXR | T | B850 |
| 170 | 1053 | | JLT | RLOOP | 3B2FEA |
| 175 | 1056 | EXIT | STX | LENGTH | 134000 |
| 180 | 1059 | | RSUB | | 4F0000 |
| 185 | 105C | INPUT | BYTE | X'F1' | F1 |

```
195                   .
200                   .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                   .
210     105D    WRREC      CLEAR      X                  B410
212     105F               LDT        LENGTH             774000
215     1062    WLOOP      TD         OUTPUT             E32011
220     1065               JEQ        WLOOP              332FFA
225     1068               LDCH       BUFFER,X           53C003
230     106B               WD         OUTPUT             DF2008
235     106E               TIXR       T                  B850
240     1070               JLT        WLOOP              3B2FEF
245     1073               RSUB                          4F0000
250     1076    OUTPUT     BYTE       X'05'              05
255                        END        FIRST
```

Indirect addressing mode is indicated by adding the prefix @ to the operand. Immediate addressing mode is denoted by adding the prefix # to the operand. Instructions that refer to memory are normally assembled using either the program-counter relative or base relative mode. BASE (line 70) is an assembler directive used in conjunction with base relative addressing. If the displacements required for both program-counter relative and base relative addressing are too large to fit into a 3-byte instruction, then 4-byte extended format (format 4) must be used. The extended instruction format is specified with the prefix + added to the operation code in the source statement (line 15,35,65). Programmer has to specify this addressing when it is required.



- Main difference in SIC and SIC/XE program:
  - Register –to – register instructions (in place of register – to –memory instructions) wherever possible
  - Eg: in line 150 : COMP ZERO is changed to

    COMPR A,S

  - Similarly, in line 165: TIX MAXLEN

    TIXR T

- Register –to – register instructions are faster than the corresponding register – to – memory instructions because they are shorter and more importantly, they do not require another memory reference

– Fetching an operand from a register is much faster than retrieving it from the main memory

– When using immediate addressing, the operand is already present as part of the instruction and need not be fetched from anywhere (line 55, 70)

– Addition of some instructions:

  – Changing COMP to COMPR (line 150) forces to use CLEAR (line 132)

  – Improvement in execution speed

  – CLEAR is executed only once for each record read

  – COMPR is executed for every byte of data transferred

**Hand Assembly of SIC/XE**

## Instruction Format and Addressing Mode



SIC/XE

- Immediate addressing: op #c
- Indirect addressing: op @m
- PC-relative or Base-relative addressing: op m
  - The assembler directive **BASE** is used with base-relative addressing
  - If displacements are too large to fit into a 3-byte instruction, then 4-byte extended format is used
- Extended format: +op m
- Indexed addressing: op m, **x**
- Register-to-register instructions
- Large memory
  - Support multiprogramming and need *program reallocation* capability

## Translation

- START now specifies a beginning program address of 0
- **Register translation**
  - Register name (A, X, L, B, S, T, F, PC, SW) translated to their number (0,1, 2, 3, 4, 5, 6, 8, 9)
  - May be preloaded in SYMTAB

```
                                   X
125   CLEAR      X            B4|1|0
150   COMPR      A,  S        A0|0|4
                                  A  S
```



## Address Translation

- Most register-to-memory instructions are assembled using *PC relative* or *base relative* addressing
  - Assembler must calculate a *displacement* as part of the object instruction
  - If displacement can be fit into 12-bit field, format 3 is used.
  - Format 3: 12-bit address field
    - Base-relative: 0~4095
    - PC-relative: -2048~2047
  - Assembler attempts to translate using PC-relative first, then base-relative
    - If displacement in PC-relative is out of range, then try base-relative

# Address Translation (Cont.)

- If displacement can not be fit into 12-bit field in the object instruction, format 4 must be used.
  - Format 4: 20-bit address field
  - No displacement need to be calculated.
    - 20-bit is large enough to contain the full memory address
  - Programmer must specify extended format: +op m
  - For example: *+JSUB RDREC => 4B101036*
    - *LOC(RDREC) = 1036, get it from SYMTAB*

# Choice of Addressing Modes



1. Programmer must specify the extended format (4-byte) by using the prefix +
2. If not, assembler first attempts PC-relative
3. If the required displacement is out of range, use base relative addressing can be use
4. Otherwise, generate an error message

## Program counter relative

- Calculate displacement
  - Displacement must be small enough to fit in a 12-bit field (-2048..2047 )
  - In SIC, PC is advanced *after each instruction is fetched* and *before it is executed*; *i.e.*, PC contains the address of the next instruction.

SIC/XE

```
10     0000        FIRST       STL          RETADR
```

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

*10 0000 FIRST STL RETADR 17202D*



- Opcode (6 bits) $=14_{16}=00010100_2$
- nixbpe=110010
  - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
  - p = 1: indicate *PC-relative* addressing

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0001 01 | 1 | 1 | 0 | 0 | 1 | 0 | $(02D)_{16}$ |

Object Code = 17202D

```
40     0017        J           CLOOP
```

CLOOP is at address $(0006)_{16}$

After the SIC fetches this instruction, (PC) = $(001A)_{16}$

TA = (PC) + disp ⇒ disp = TA – (PC) = 0006 – 001A = $(F\boxed{FEC})_{16}$

- Opcode=$3C_{16} = 00111100_2$
- nixbpe=110010

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0011 11 | 1 | 1 | 0 | 0 | 1 | 0 | $(FEC)_{16}$ |

Object Code = 3F2FEC

## Base relative

In PC relative assembler knows what the content of PC will be at execution time

- 12 bits displacement (0 ~ 4095)
- Base register is under the control of the programmer.
  - The programmer must tell the assembler what the base register will contain during execution of program.
- Assembler directive
  - **BASE**: tell the assembler what the base register will contain
  - **NOBASE**: tell the assembler that the contents of the base register can no longer be used for addressing.
  - When based register can be relied upon, the assembler can use base relative, otherwise only the PC-relative can be used
  - The assembler first choose PC-relative;
    if displacement is not enough, choose base relative

```
LDB       #LENGTH     (instruction)
BASE      LENGTH      (directive)
          :
NOBASE
```

```
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
```

TA =(PC)+DISP
DISP =TA – (PC)
DISP =0036 – 1051
=EFE5

- PC-relative is no longer applicable
  - $(0036)_{16} - (1051)_{16} = (-1015)_{16} < (-0800)_{16} = (-2048)_{10}$
- LDB loads the address of LENGTH into base register during execution
- BASE directive explicitly informs the assembler that the base register will contain the address of LENGTH

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

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

disp = TA – (B) ➔ disp = 0036 – 0033 = $(0003)_{16}$

- Opcode=54=01010100
- nixbpe=111100
  - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
  - x = 1: *indexed* addressing
  - b = 1: *base-relative* addressing

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0101 01 | 1 | 1 | 1 | 1 | 0 | 0 | $(003)_{16}$ |

Object Code = 57C003

# Base relative

In PC relative:
DISP = TA –(PC)
DISP = 0033 – 000D =26

```
20   000A         LDA       LENGTH    032026
                   :           :
175  1056 EXIT    STX       LENGTH    134000
```

- Line 20, using PC-relative
- Consider Line 175
  - If we use PC-relative
    - LENGTH at address 0033
    - Disp = TA – (PC) = 0033 -1059 = EFDA
    - PC relative is no longer applicable, try to use BASE relative addressing

Line 20,
If Base relative:
DISP = TA - (B)
DISP = 0033 – 0033 =00

Assembler attempts to translate using *PC-relative* first, then *base-relative*

- e.g. 175   1056   STX   LENGTH   134000

- Try base-relative next
  - displacement= LENGTH – (B) = 0033 – 0033 =0
  - Opcode=10
  - nixbpe=110100
    - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
    - b = 1: *base-relative* addressing

If you calculate program – counter relative displacement that would be required for the statement on line 175, you will see that it is too large to fit into the 12 bit displacement field. Line 20 can be used with base relative mode. In our assembler, however we have arbitrarily chosen to attempt program – counter relative assembly first.

## Immediate addressing

* Convert the immediate operand to its internal representation and insert into the instruction

- No memory reference is involved
- If immediate mode is specified, the target address becomes the operand

```
55     0020        LDA         #3
```

Immediate operand

TA = $(0003)_{16}$

- Opcode=00
- nixbpe=010000
  - i = 1: *immediate addressing*

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0000 00 | 0 | 1 | 0 | 0 | 0 | 0 | $(003)_{16}$ |

Object Code = 010003

•



```
133   103C        +LDT        #4096
```

Extended instruction format

TA = $(01000)_{16}$

- Opcode=74=01110100
- nixbpe=010001
  - i = 1: *immediate addressing*
  - e = 1: *extended instruction format* since 4096 is too large to fit into the 12-bit displacement field

•Even the operand is too large for 20 bit address field, immediate addressing could not be used

| OPCODE | n | i | x | b | p | e | Address |
|---|---|---|---|---|---|---|---|
| 0111 01 | 0 | 1 | 0 | 0 | 0 | 1 | $(01000)_{16}$ |

Object Code = 75101000

•

## Immediate & PC-relative addressing

12 0003 LDB #LENGTH 69202D

- The immediate operand is the symbol LENGTH
  - The address of LENGTH is loaded into register B
- Displacement=LENGTH – (PC) = 0033 – 0006 = 02D
- Opcode=$68_{16}$ = $01101000_2$
- nixbpe=010010
  - Combined *PC relative* (p=1) with *immediate addressing* (i=1)

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0110 10 | 0 | 1 | 0 | 0 | 1 | 0 | $(02D)_{16}$ |

- 69202D

In this line 12, the immediate operand is the symbol LENGTH. Since, the value of symbol is the address assigned to it, the immediate instruction has the effect of loading register B with the address of LENGTH. Note that here we combined program counter relative addressing with immediate addressing. In general, target address calculation is performed, then, if immediate mode is specified, the target address (*not the contents stored at that address)* becomes the operand.

## Indirect Address Translation

- Indirect addressing
  - The contents stored at the location represent the *address* of the operand, not the operand itself
  - Target addressing is computed as usual (PC-relative or BASE-relative)
  - *n* bit is set to 1

- Line 70, shows a statement that combines program – counter relative and indirect addressing .

```
70     002A       J          @RETADR
```

Indirect addressing

CLOOP is at address $(0030)_{16}$

After the SIC fetches this instruction, (PC) = $(002D)_{16}$

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

- Opcode= 3C=00111100
- nixbpe=100010
  - n = 1: *indirect addressing*
  - p = 1: *PC-relative addressing*

| OPCODE | n | i | x | b | p | e | disp (12 bit) |
|---|---|---|---|---|---|---|---|
| 0011 11 | 1 | 0 | 0 | 0 | 1 | 0 | $(003)_{16}$ |

3E2003

**Program Relocation**

- The larger main memory of SIC/XE
  - Several programs can be loaded and run at the same time.
  - This kind of sharing of the machine between programs is called ***multiprogramming***
- To take full advantage
  - Load programs into memory wherever there is room
  - Not specifying a fixed address at assembly time
  - Called ***program relocation***

If we knew in advance exactly which programs were to be executed concurrently in this way, we could assign addresses when the programs were assembled so that they would fit together without overlap or waste space. It is impractical to plan program execution this closely. We do not know exactly when jobs will be submitted, exactly how long they will run etc. Desirable to load programs into memory wherever there is room for it. In such a situation, the actual starting address of the program is not known until load time.

## *Absolute program* (or *absolute assembly*)

- Program must be loaded at the address specified *at assembly time.*
- E.g. Fig. 2.1

program loading starting address 1000

```
COPY   START   1000
FIRST  STL     RETADR
                 :
                 :
```

The address may be invalid if the program is loaded into somewhere else.

- Example: (Figure 2.2, pp.47)

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

Calculate based on the starting address 1000

**Reload the program starting at 3000**

```
55      3 01B         LDA     THREE          00302D
```

The absolute address should be modified

- What if the program is loaded to 2000

  e.g. 55     101B     LDA     THREE     <u>00202D</u>

  - Each <u>absolute address</u> should be modified

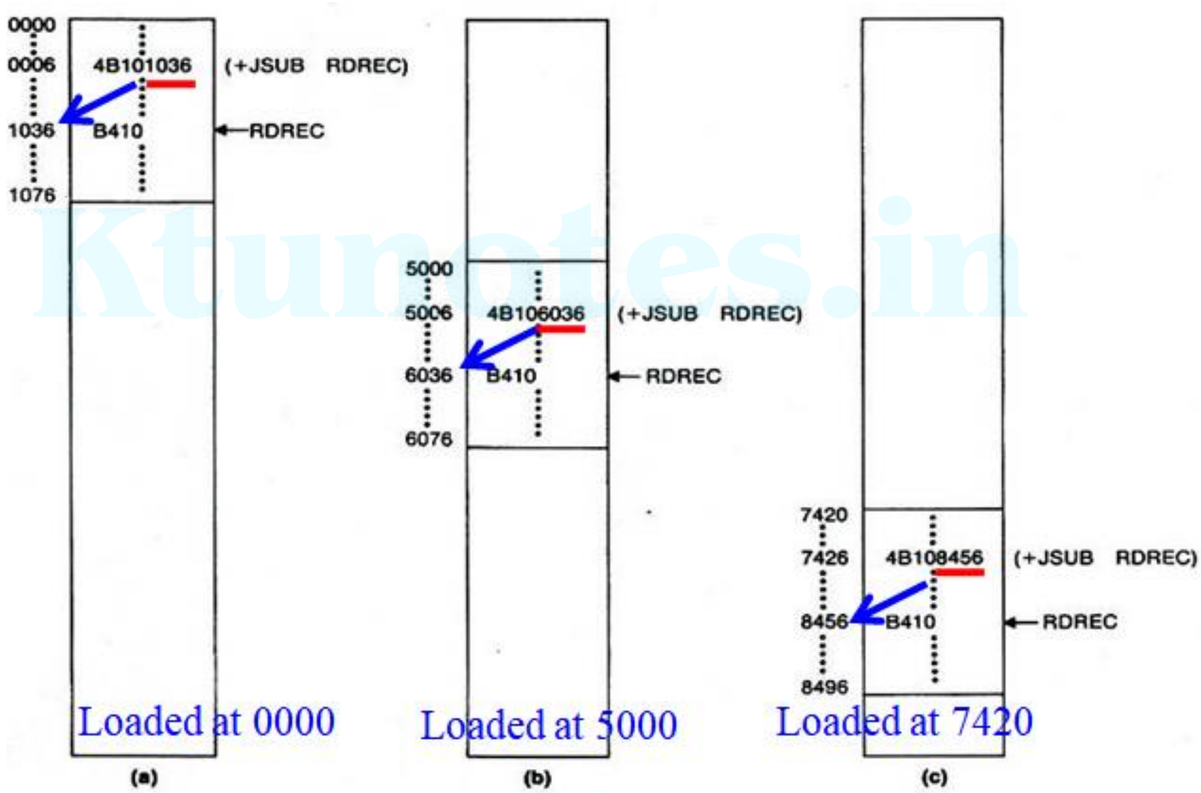
If we do this, the address 102D will not contain the value that we expect – in fact, it will probably be part of some other user's program. We need to make some change in the address portion of this instruction so we can load and execute the program at address 2000.

| Line | Loc | Source statement | | | Object code |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 17202D |
| 12 | 0003 | | LDB | #LENGTH | 69202D |
| 13 | | | BASE | LENGTH | |
| 15 | 0006 | CLOOP | +JSUB | RDREC | 4B101036 |
| 20 | 000A | | LDA | LENGTH | 032026 |
| 25 | 000D | | COMP | #0 | 290000 |
| 30 | 0010 | | JEQ | ENDFIL | 332007 |
| 35 | 0013 | | +JSUB | WRREC | 4B10105D |
| 40 | 0017 | | J | CLOOP | 3F2FEC |
| 45 | 001A | ENDFIL | LDA | EOF | 032010 |
| 50 | 001D | | STA | BUFFER | 0F2016 |
| 55 | 0020 | | LDA | #3 | 010003 |
| 60 | 0023 | | STA | LENGTH | 0F200D |
| 65 | 0026 | | +JSUB | WRREC | 4B10105D |
| 70 | 002A | | J | @RETADR | 3E2003 |
| 80 | 002D | EOF | BYTE | C'EOF' | 454F46 |
| 95 | 0030 | RETADR | RESW | 1 | |
| 100 | 0033 | LENGTH | RESW | 1 | |
| 105 | 0036 | BUFFER | RESB | 4096 | |

The only parts of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified

- Not a memory address (immediate addressing)
- PC-relative, Base-relative

Looking at the object program, it is not possible to distinguish the values which represent addresses and which represent the constant data items. Assembler does not know the actual location where the program will be loaded, it cannot make changes in the addresses used by the program. However, the assembler can identify for the loader those parts of the object program that need modification




Note that no matter where the program is loaded, RDREC is always 1036 bytes past the starting address of the program.

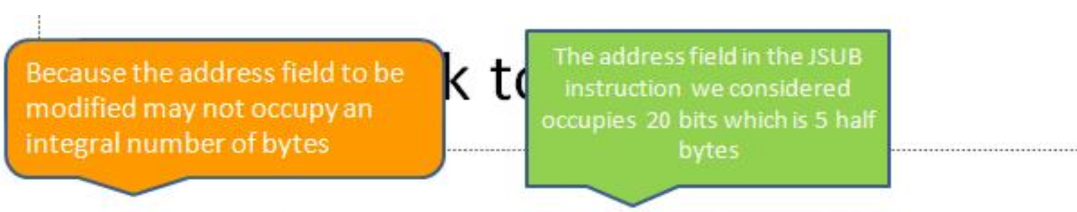
Please check the slide no. 48 and 49 in slide show for better understanding of program relocation in SIC and SIC/XE.

**Relocatable Program**

An object program that contains information needed for address modification for loading is called re-locatable program. Here we are considering JSUB instruction, when assembler generates the object code for the JSUB instruction, it will insert the address of RDREC relative to the start of the program (*This is the reason we initialized the location counter to 0 for the assembly. The assembler will also produce a command for the loader, instructing it to add the beginning address of the program to the address field in the JSUB instruction at load time. The command for the loader must also be a part of the object program.*

**Format of Modification Record**

| | |
|---|---|
| Col 1 | M |
| Col 2-7 | Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal) |
| Col 8-9 | length of the address field to be modified, in half-bytes (hexadecimal) |

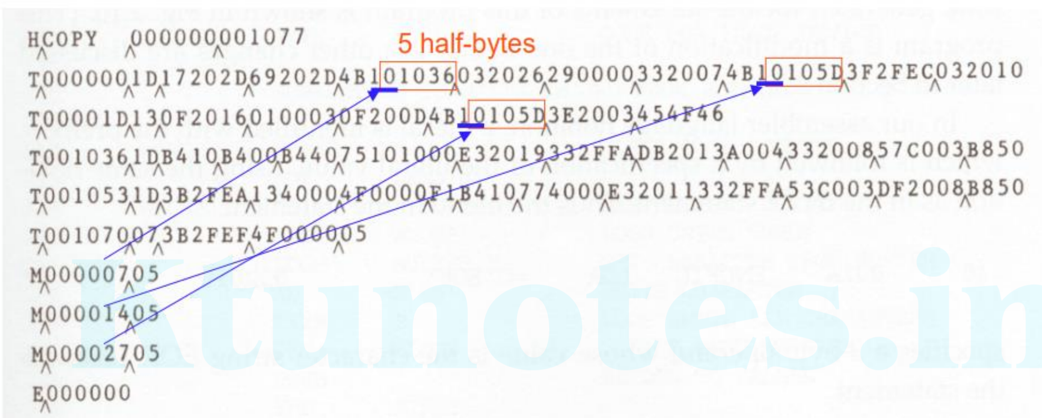


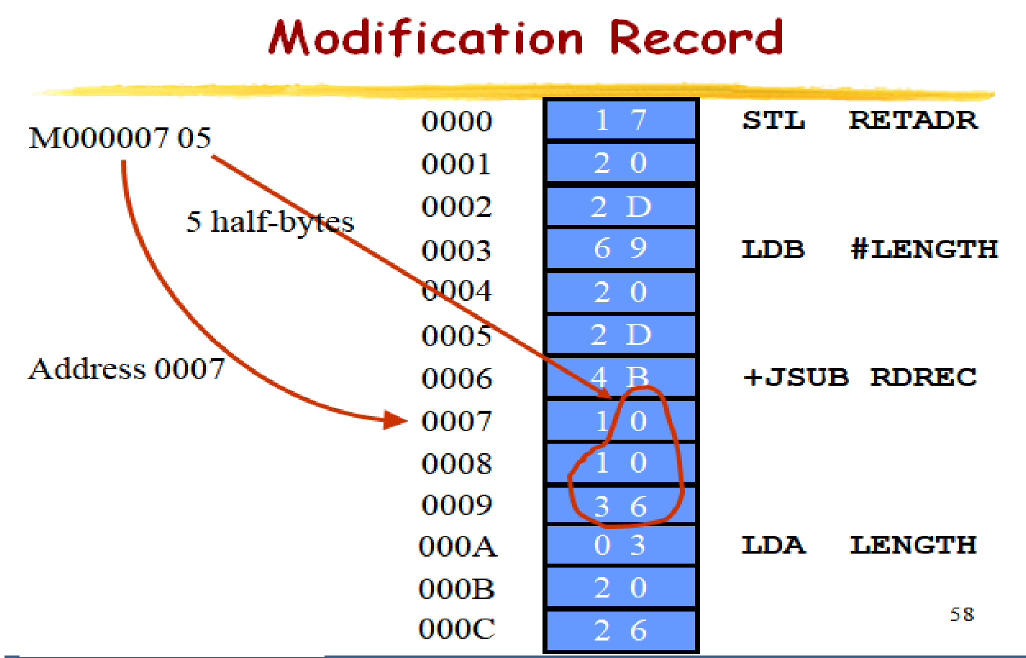

- One modification record for each address to be modified
- The length is stored in half-bytes (4 bits)
- The starting location is the location of the byte containing the leftmost bits of the address field to be modified.
- If the field contains an odd number of half-bytes, the starting location begins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

Half byte approach is closely related to SIC/XE. Other machines it is not appropriate.

```
H^COPY ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

**Figure 2.8** Object program corresponding to Fig. 2.6.



The record specifies that the beginning address of the program is to be added to a field that begins at address 000007( relative to start of the program) and is 5 half bytes in length. Thus in the assembled instruction 4B101036, the first 12 bits (4B1) will remain unchanged. The program load address will be added to the last 20 bits (01036) to produce the correct operand address.

Exactly, the same kind of relocation must be performed for the instructions on lines 35 and 65. The rest of the instructions in the program, however, need not be modified when the program is loaded:

- Some cases operand is not memory address at all( eg: CLEAR S or LDA # 3)
- Some cases operand is specified using program-counter relative or base relative addressing
- In line 10 STL RETADR is assembled using program counter relative addressing with displacement 02D)
- No matter where the program is loaded in memory, the word labelled RETADR will always be 2D bytes away from the STL instruction→ Thus no instruction modification is needed
- When STL is executed, the program counter will contain the (actual ) address of the next instruction
- The target address calculation process will then produce the correct (actual ) operand address corresponding to RETADR

Similarly, the distance between LENGTH and BUFFER will always be 3 bytes.

- Thus displacement in the base relative instruction on line 160 will be correct without modification. (The contents of the base register will, depend upon where the program is loaded. However, this will be taken care of automatically when the program –counter relative instruction LDB # LENGTH is executed)