Implementation of SIC/XE 2 Pass Assembler

Akansha Jajodia, Harika Andugula, Manasi Gupta, Shravanthi Bhoopalam Sudharshan
Computer Engineering Department
San José State University (SJSU)
San José, CA, USA
Email: akansha.jajodia@sjsu.edu,harika.andugula@sjsu.edu,
manasi.gupta@sjsu.edu, shravanthi.bhoopalamsudharshan@sjsu.edu

***Abstract*— Most software developers are concerned with designing applications to solve real-life problems. The underlying mechanism for running this program has always been overlooked and less acknowledged. There is a need to concentrate on improving the processes that operate behind the scenes.The purpose of the project is to create a SIC/XE assembler using C and C++ for a comparative performance analysis. The regular SIC/XE assembly system has minimal capabilities. The concept is to introduce a SIC/XE assembler with some modifications that could enhance the assembler's ability to handle a number of scenarios.**

***Index Terms*: SIC, SIC/XE, Assembler.**

I. INTRODUCTION

Many of the software applications we see today have their target set to produce performance. These applications are designed with the goal of obtaining the inspiration to develop them, for example, in order to solve a real-life scenario. The machine is only becoming a tool to run these applications. System software is an application that facilitates the use of computers. One such example is the creation of an assembler.Assemblers are executable programs that translate code to machine-readable text. This is why the developer is encouraged to concentrate on the hardware that is being used. Most of today's processors have specific capabilities, leaving device software research limited to hardware. There is a need to generalize the outcome.The solution is believed to be a theoretical machine whose functions follow the hardware features found on computer systems. This helps the developer to build system software with no need to know the functionality of the actual system.

There are a variety of approaches to implement assembly code in system software and several more architectures to apply to such as SIC, CISC machines, RISC machines. Our project focuses on the implementation of the SIC/XE two-pass assembler. All the different terms related to this architecture will be discussed in detail in this report. This report will further describe the the SIC/XE architecture and its specification along with the improvements introduced in the implementation, output developed and the conclusion.

II. BACKGROUND

A. Simplified Instructional Computer

SIC is a theoretical computer. Many modern computers have very complex functionalities and so it becomes very difficult to understand machine programming using these specific applications. The solution is SIC, which is built to provide hardware features that are often found on a real computer, but prevents any uncommon or insignificant complexities. With this , we can grasp the core principles of system software by distinguishing them from the specifics of the machine's implementation.

There are two types of SIC. The first is the regular model and the second is the XE edition where the XE is short for 'extra expensive' or 'extra equipment' .These two versions are built in a forward compatible manner, where an object program written for the standard SIC would also run correctly in the SIC/XE edition.

SIC Architecture

- **Memory**:

  The memory comprises of 8-bit bytes, and a word is created by each of the 3 successive bytes (i.e. 24 bits). All addresses are addressable by byte and the words are addressed by their lowest numbered byte position. The computer's memory has a capacity of 32,768 bytes.

- **Register**:

  A total of 5 special use registers with a length of 24 bits are available. The use of these registers, their numbers and their mnemonics are listed in the following table.

- **Data Formats**:

  The integers are saved using 24-bit binary numbers. 2's complement representation is used for negative values. On this unit, floating-point hardware is not present. For characters, storage uses their 8-bit ASCII codes.

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

Fig. 1. Special Use Registers

- **Instruction Formats**:

  For instructions, there is a 24-bit format that is illustrated below. To indicate the indexed-addressing mode, the flag bit, x, is used.

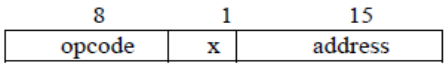

Fig. 2. Instruction Formats

- **Addressing Modes**:

  Addressing modes is of 2 types- Direct and Indirect. They are indicated in the instruction, using the x. The two modes are explained in the table below. X reflects the contents of the X register.

Indication	Mode	Target Address (TA) Calculation
x = 0	Direct	TA = address
x = 1	Indirect	TA = address + X

Fig. 3. Addressing Modes

- **Instruction Set**:

  A basic collection of instructions that are necessary to perform the simplest tasks is given by the SIC machine. The instructions that are used for loading and storing as well as integer arithmetic operations involve these tasks.

- **Input and Output**:

  The I/O operations are performed on the standard SIC machine by moving 1-byte data value to and from the right-most 8 bits of the register. There is a specific 8-bit code allocated to every unit. There are three I/O instructions, each defining the device code as an operand.

SIC/XE Architecture

- **Memory**:

  For the SIC/XE computer, the memory configuration is the same as it was for the SIC computer. For this computer, however the maximum memory size is 1 MB.

- **Register**:

  The five SIC computer registers are the same for SIC/XE computers. There are also an additional 4 registers. They are listed below.

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

Fig. 4. Additional Registers

- **Data Formats**:

  The format of the data is the same as that of a regular SIC computer. Besides that a 48-bit floating-point data type format is defined below.

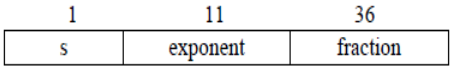

Fig. 5. Data Format

The fraction is a value varying from 0 to 1. Between 0 and 2047, the exponent is perceived to be an unsigned binary number. The (s) sign denotes whether the floating points number is positive (0) or negative (1).

- **Instruction Formats**:

  The addresses would not fit into the 15-bit field as there is a wider memory available on the SIC/XE computer. Therefore, new kinds of formats for instruction are available here. For distinguishing between Formats 3 and 4, the bit (e) is used, where e = 0 indicates Format 3 and e = 1 indicates Format 4. Below table explains the formats.

- **Addressing Modes**:

  There are two new relative addressing modes available for use with instructions assembles using Format 3. Below table describes them.

- **Instruction Set**:

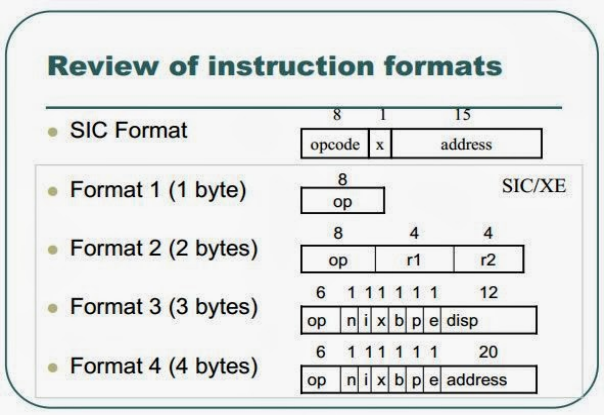

Fig. 6. Instruction Formats

The SIC/XE architecture provides floating point arithmetic functions in addition to the other basic instructions that are provided by SIC. The floating-point arithmetic operations are briefly explained below.

1) ADDF is used for adding floating points.
2) SUBF is used for subtracting floating points.
3) MULF is used for Multiplying floating points.
4) DIVF is used for Dividing floating points.
5) In order to handle interrupts SIC/XE architecture provides Supervisor Call (SVC).

- **Input / Output**:

The SIC/XE architecture is more efficient as it allows allow overlapping of computing and I/O by providing I/O channels that allows to perform I/O operations while CPU is executing other tasks. Instructions such as SIO, TIO, HIO are used to start, test, and halt the operation I/O channels.

B. Assemblers

Assembler is responsible to convert the assembly language to executable machine code and this conversion process is often referred to as assembling the code or assembly. The assembler converts the low-level assembly code to machine relocatable code. The assembler goes through the program one line at a time and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. The instructions are converted into object code which is then interpreted by the machine. It's basically a translator that translates an assembler program into a conventional machine language program.

There are various directives for assemblers that are nothing but pseudo instructions. They can not be converted into machine instructions, but they only provide the assembler with instruction data. There are a few simple directives for assemblers, namely,

1) _START: Specifies the program's name and starting address.
2) END: The end of the program is indicated.
3) _EQU: This directive is used to substitute a symbol for a number.

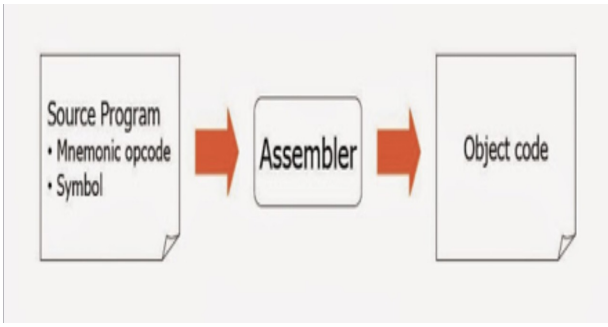

Fig. 7. Assembler Block Diagram

There are 2 distinct assembler forms.

1) One Assembler Transfer.
2) Two Assemblers Move.

One Pass Assembler:

Only once can a one-pass assembler pass over the source program or source code. It gathers all the labels in that one pass, resolves the potential references, and conducts the actual assembly. The difficult part here is to overcome the future references to the mark, which is also known as the problem of forward referencing and assembling the code in one stage.

Two Pass Assemblers:

Two passes are carried out by the two-pass assembler over the source program. The following describes a list of high-level steps performed in both passes

1) Pass One:

1) Assign addresses to all program statements.
2) Save values(addresses) assigned to all labels(including label and variable names) for use in Pass 2 (address reference handling)
3) Perform any assembler directive processing (e.g., BYTE, RESW, which can influence the assignment of addresses)
4) All the labels identified in the program should be included in the Symbol Table at the end. The assembly maintains a Position Counter (LC) to assign addresses to labels.

2) Pass Two:

1) Assemble (generate opcode and look up addresses) instructions.
2) Generate values of data specified by BYTE, WORD.
3) Processing assembler instructions that are not done in Pass 1.
4) Write the object programs and the listing of addresses.

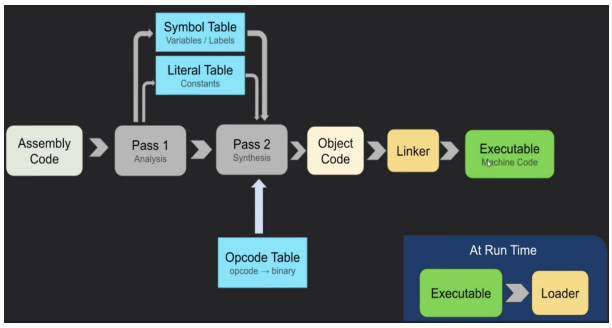

Fig. 8. 2 Pass Assembler Architecture

III. Implementation

SIC-XE supports the Process Code Table (OPTAB), Symbol Table (SYMTAB) and Position Counter (LOCCTR) data structures. To look up mnemonic opcodes and translate them to the corresponding machine language code, OPTAB is used. The addresses/values which are assigned to labels are stored by SYMTAB. In assigning an address, the variable LOCCTR is used. The starting address that is specified in the START statement is always initialized. The current LOCCTR value includes an address associated with the name. In general, OPTAB provides the opcode and its machine code. OPCODE implementation can be accomplished using a hash table with the key being the mnemonic opcode. In the source assembly software, SYMTAB includes the name and value for each mark, along with error flags.

In both C and C++, the Two Pass SIC-XE Assembler has been implemented. An assembly code is the input file. There are 52 instructions in it. Instructions are read and information such as mnemonics, format and opcode is mapped to the operation table (OPTAB) using a hash table for various opcode types. Pass 1 is executed where all lines are parsed and the position counter is changed to search for literals. Error messages are shown in the event of any exception.

An intermediate file is created as the results of pass 1 algorithm. The results of pass 1 also contains SYMTAB. These results are input to the pass 2 algorithm. In pass 2 algorithm "Object code" is produced. Thereafter, SYMTAB is updated and the results are inputted in a text file which represents the output of pass 2. The output text file generated contains the header, define, text, modification and end records which is achieved by calling out their respective functions. In the end, Assembling is done.

A. Language: C++

For the implementation of our project, we have chosen C++ as the coding language. The main purpose of using C++ as the main coding language is because it is an object-oriented language with various options of data structures which can be used for the implementation of different functionalities. Vectors and Hashmaps have been used in numerous places which is readily available. The solution that we have implemented supports all the above-mentioned opcodes. Hence, our implementation code can be utilized to convert any assembly level language code which contains opcodes from Figure [4]. The SIC/XE implementation steps are followed in 2 phases, the steps are given below.
Pseudo Code:
Read input file from cmd line. Read entire source program.

1) Run 1st pass:
   Separate the Symbols, Literals and Mnemonics. Perform Location Counter Processing.
2) Update Symbol Table data structure with labels and their addresses.
3) Write result to an intermediate file.

4) Run 2nd pass:
   Read intermediate file from current working directory.
5) Read instructions and assemble them using Symbol Table. Compute correct address for every label.
6) Translate operands name to actual register. Translate immediate value to 1's and 0's
7) Write resultant object code to output

B. Language: C

C programming language has many advantages like portability. It is the lowest high-level programming language. Many important concepts must be studied carefully to implement SIC/XE assembler in C programming language. These concepts include pointers, file input and output, strings, memory management, and all the data structures necessary to implement the assembler and handle the conversion to machine code. The pseudo code of the implementation of SIC/XE assembler in C is given below: Read input file name from command line.

Mnemonic	Format	Opcode	Mnemonic	Format	Opcode	Mnemonic	Format	Opcode
ADD	3/4	18	LDB	3/4	68	STA	3/4	0C
ADDF	3/4	58	LDCH	3/4	50	STB	3/4	78
ADDR	2	90	LDF	3/4	70	STCH	3/4	54
AND	3/4	40	LDL	3/4	08	STF	3/4	80
CLEAR	2	B4	LDS	3/4	6C	STI	3/4	D4
COMP	3/4	28	LDT	3/4	74	STL	3/4	14
COMPF	3/4	88	LDX	3/4	04	STS	3/4	7C
COMPR	2	A0	LPS	3/4	D0	STSW	3/4	E8
DIV	3/4	24	MUL	3/4	20	STT	3/4	84
DIVF	3/4	64	MULF	3/4	60	STX	3/4	10
DIVR	2	9C	MULR	2	98	SUB	3/4	1C
FIX	1	4C	NORM	1	C8	SUBF	3/4	5C
FLOAT	1	C0	OR	3/4	44	SUBR	2	94
HIO	1	F4	RD	3/4	D8	SVC	2	B0
J	3/4	3C	RMO	2	AC	TD	3/4	E0
JEQ	3/4	30	RSUB	3/4	4C	TIO	1	F8
JGT	3/4	34	SHIFTL	2	A4	TIX	3/4	2C
JLT	3/4	38	SHIFTR	2	A8	TIXR	2	B8
JSUB	3/4	48	SIO	1	F0	WD	3/4	DC
LDA	3/4	00	SSK	3/4	EC			

Fig. 9. Opcodes supported in the implementation

1) Run 1st pass:
   Read one line at a time: Validate input line.
2) Fill OpTab with code and object code Fill SymTab with symbol and address
3) Append above info to input line and write to intermediate file.
4) Run 2nd pass:
   Read intermediate file one line at a time till END: Write H record to the output file.
5) Compute address for every label convert operand to opcodes convert immediate values to binary Prepare a record with above info Write a T record to output file.
6) Write an E record as END is reached Close output file.

IV. ENHANCEMENTS

Numerous enhancements been listed to the basic implementation of SIC/XE assembler implementation to make more efficient and robust in order to improve the performance.

1) **A. Implementation using Hash Table:**

The opcode table has several values for respective opcodes which needs to be accessed multiple times. In our implementation we have made use of Hashmaps to store the values present in opcode table. The reason for choosing Hashmap is because the time complexity of finding or searching an element from hashmap is O(1). This ensures efficient and fast performance.

2) **B. Exception Handling:**

Robustness of SIC/XE assembler implementation is given by the capacity to withstand erroneous scenarios. All the situations must be handled by the assembler efficiently. These exceptions must be taken care of which implementation of the code. Our implementation has included many such scenarios where the code can break and has ensured a feasible solution for the same. Various exceptions that can occur while working on assembler are listed below:

a) Illegal Byte
b) Invalid operand
c) Invalid opcode
d) Illegal Label
e) Illegal Hexadecimal
f) Illegal format (EXTREF WITHOUT '+')
g) Program without END Clause
h) Illegal Use of Reserved Labels Below

Each of these exceptions and their solutions have been explained in detail.

3) **Illegal Byte:**

The value of Bytes can go beyond the addressing limits. There is a chance that it may violate the addressing parameters as well. The below figure shows the test-case outputs.

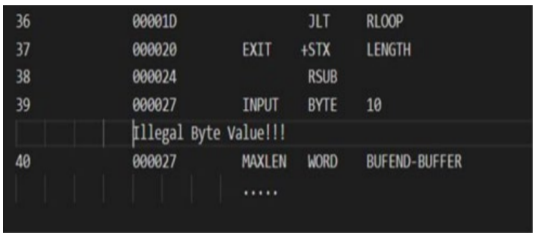

Fig. 10. Illegal Byte

4) **Invalid Operand:**

There could be a scenario wherein the programmer uses one of the operands which has not been configured into the assembler. Under such exceptions, our implementation of SIC/XE assembler executes instructions till the last instruction before the line causing the exception and then throws an illegal instrument exception which is required in such cases. The same case is illustrated in figure 6. The output of such case can be seen in figure 7 where this type of exception is handled by assembler and it passes the test. The message "***Invalid operands for literal" indicates the exception handling provided under such

circumstances in our code.

5) **Invalid Opcode:**

The opcodes are hard-coded to have an opcode as strings in a hash-map and a wrapper class containing their corresponding machine as their values. There might be an example where an opcode that is not visible on the map is defined by the programmer. This can lead to an error happening. In order to prevent this, key miss, an error is shown in that instance that can serve as a debugger. The test case, which is shown to have passed in Figure 11, is shown in Figure 12.

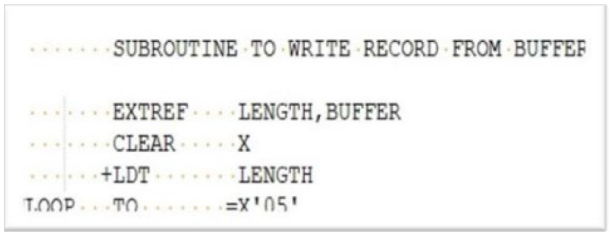

Fig. 11. Invalid Opcode - TQ changed to TD

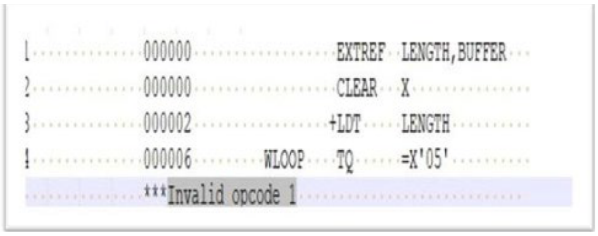

Fig. 12. Invalid Opcode Execution

6) **Illegal Labels:**

When writing a program, it is not uncommon for a programmer to make spelling mistakes. The assembler is used to deal with these situations if the labels are not correct, as shown in Figure 13.

Fig. 13. Illegal Label Exception

7) **Illegal used of Reserve Labels:**

Registers A and S were wrongly used in Fig 11. The COMPR A,S syntax was incorrectly written as COMPR A S. This can not be translated into a valid opcode; as seen in the figure, it is therefore captured as an exception. The test case below illustrates the assembler's response to experiencing such an incident. Represented in the figure.

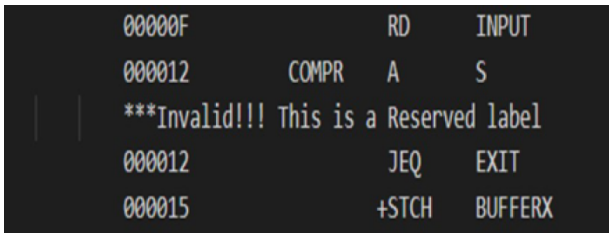

Fig. 14. Illegal use of Reserve Labels

8) **Illegal hexadecimal value:**

Valid Instant values are in the 0-9 and A-F ranges. The immediate value given is invalid in figure 15 and it can not be interpreted. The assembler catches this exception in such an instance and gives the error output as shown in the figure.

Fig. 15. Illegal hexadecimal value

9) **Illegal Format:**

There are four formatting modes on the SIC/XE and each mode has a particular representation system. '+' does not precede the STCH opcode in figure , on line 34, resulting in an exception. The Assembler allows an exception that guides the programmer to use ExtRef in the correct way.

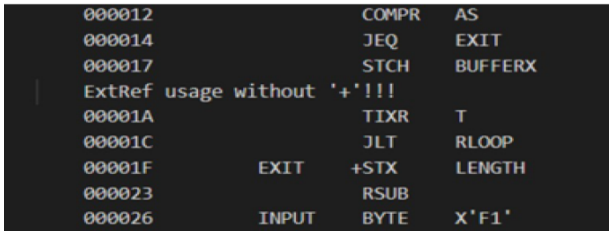

Fig. 16. Illegal Format

10) **Program without End Clause:**

The instructions given to the assembler are of a definite length. The "END" sentence reflects the end of the assembly program. There may be cases where the "END" clause is absent from the programmer. The action of the assembler on that occasion is shown in Figure.

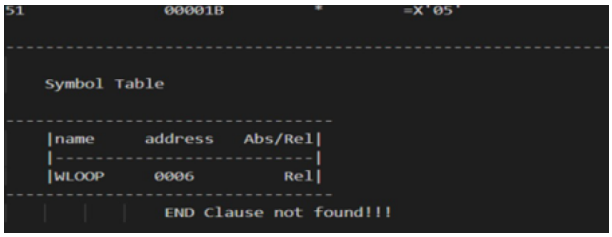

Fig. 17. Missing End Clause

V. RESULTS

This section contains the results of our project. For that we will first see what results are expected as an output of an assembler. We know that the assembler gives the object code as an output which is a series or pattern of 0's and 1's of a given length. This object program can then be executed whenever required. As mentioned in the previous sections, the data structures used for implementation of the SIC/XE assembler in our project are OPTAB and SYMTAB. We have implemented the assembler in C as well as C++ programming language.

1) **Results of C++:**

For the generation of the final object code, there are multiple intermediate stages whose output is important for the final result. All these results are given in detail below:

- OPTAB is generated after Pass 1 wherein the input file is parsed line by line and mapping between mnemonic is present along with instruction format, addressing modes and length of information.
- SYMTAB is generated from the OPTAB where addresses are assigned to all the labels.
- This is given to Pass 2 as an input and the opcode for all the operands is generated
- Also, the object code is generated in a different text file with Header, Define, Refer, Text and End records. In our project, we have divided the input file into 3 blocks: Main operations to be performed (Block 1), subroutine to read record into buffer (Block 2), subroutine to write record from the buffer (Block 3). Hence, the output tables are generated separately for all these 3 blocks.

Output of Pass 2

Output for Pass 1
Operation Table

line number	address	label	op	operands
1	000000	COPY	START	0
2	000000		EXTDEF	BUFFER,BUFEND,LENGTH
3	000000		EXTREF	RDREC,WRREC
4	000000	FIRST	STL	RETADR
5	000003	CLOOP	+JSUB	RDREC
6	000007		LDA	LENGTH
7	00000A		COMP	#0
8	00000D		JEQ	ENDFIL
9	000010		+JSUB	WRREC
10	000014		J	CLOOP
11	000017	ENDFIL	LDA	=C'EOF'
12	00001A		STA	BUFFER
13	00001D		LDA	#3
14	000020		STA	LENGTH
15	000023		+JSUB	WRREC
16	000027		J	@RETADR
17	00002A	RETADR	RESW	1
18	00002D	LENGTH	RESW	1
19	000030		LTORG	
20	000030	*	=C'EOF'	
21	000033	BUFFER	RESB	4096
22	001033	BUFEND	EQU	*
23	001033	MAXLEN	EQU	BUFEND-BUFFER

Fig. 18. OPTAB for Block 1

Symbol Table

name	address	Abs/Rel
BUFEND	1033	Rel
BUFFER	0033	Rel
CLOOP	0003	Rel
COPY	0000	Rel
ENDFIL	0017	Rel
FIRST	0000	Rel
LENGTH	002D	Rel
MAXLEN	1000	Rel
RETADR	002A	Rel

Fig. 19. SYMTAB for Block 1

2) **Results of C code:**

In C, we have implemented a small code with basic functionalities and few input instructions. We got the Opcodes and Object program after its execution as follows:

VI. EXPERIMENTS

We have done two experiments to see the performance of the SIC/XE assembler. For the first experiment, a software

```
                .
                .       SUBROUTINE TO READ RECORD INTO BUFFER
                .
000000                  EXTREF  BUFFER,LENGTH,BUFFEND
000000                  CLEAR   X
000002                  CLEAR   A
000004                  CLEAR   S
000006                  LDT     MAXLEN
000009          RLOOP   TD      INPUT
00000C                  JEQ     RLOOP
00000F                  RD      INPUT
000012                  COMPR   AOATS
000014                  JEQ     EXIT
000017                  +STCH   BUFFEROATX
00001B                  TIXR    T
00001D                  JLT     RLOOP
000020          EXIT    +STX    LENGTH
000024                  RSUB
000027          INPUT   BYTE    X'F1'
000028          MAXLEN  WORD    BUFEND-BUFFER
                .....
```

Fig. 20. OPTAB for Block 2

```
 Symbol Table

-----------------------------
 |name      address    Abs/Rel|
 |---------------------------|
 |EXIT        0020         Rel|
 |INPUT       0027         Rel|
 |MAXLEN      0028         Rel|
 |RLOOP       0009         Rel|
-----------------------------
```

Fig. 21. SYMTAB for Block 2

```
                        .
                        .       SUBROUTINE TO WRITE REC
                        .
41          000000              EXTREF  LENGTH,BUFFER
42          000000              CLEAR   X
43          000002              +LDT    LENGTH
44          000006      WLOOP   TD      =X'05'
45          000009              JEQ     WLOOP
46          00000C              +LDCH   BUFFEROATX
47          000010              WD      =X'05'
48          000013              TIXR    T
49          000015              JLT     WLOOP
50          000018              RSUB
51          00001B              END     FIRST
52          00001B      *       =X'05'
```

Fig. 22. OPTAB for Block 3

tool was run and for second, performance of the C++ and C code was found. The experiments are discussed below:

1) **Simulation:**

We ran the "SICTools" software to simulate the SIC/XE assembler with a sample input.asm file. The software was run by using the command: java -jar sictools.jar The .asm file was loaded into the software. See figure 27. This simulator shows us the addresses of all the registers used (eg, A = 000040). It also gives us the memory usage.

```
 Symbol Table

-----------------------------
 |name      address    Abs/Rel|
 |---------------------------|
 |WLOOP       0006         Rel|
-----------------------------
```

Fig. 23. SYMTAB for Block 3

```
        Output for Pass 2
        Operation Table
address     label    op      operands          n i x b p e    opcode
-------------------------------------------------------------------
000000      COPY     START   0
000000               EXTDEF  BUFFER,BUFEND,LENGTH
000000               EXTREF  RDREC,WRREC
000000      FIRST    STL     RETADR            1 1 0 0 1 0    172027
000003      CLOOP   +JSUB    RDREC             1 1 0 0 0 1    4B100000
000007               LDA     LENGTH            1 1 0 0 1 0    32023
00000A               COMP    #0                0 1 0 0 1 0    292000
00000D               JEQ     ENDFIL            1 1 0 0 1 0    332007
000010              +JSUB    WRREC             1 1 0 0 0 1    4B100000
000014               J       CLOOP             1 1 0 0 1 0    3F2FEC
000017      ENDFIL   LDA     =C'EOF'           1 1 0 0 1 0    32016
00001A               STA     BUFFER            1 1 0 0 1 0    F2016
00001D               LDA     #3                0 1 0 0 1 0    12003
000020               STA     LENGTH            1 1 0 0 1 0    F200A
000023              +JSUB    WRREC             1 1 0 0 0 1    4B100000
000027               J       @RETADR           1 0 0 0 1 0    3E2000
00002A      RETADR   RESW    1
00002D      LENGTH   RESW    1
000030               LTORG
000030      *        =C'EOF'                                  464F45
000033      BUFFER   RESB    4096
001033      BUFEND   EQU     *
000000               EXTREF  BUFFER,LENGTH,BUFFEND
000000               CLEAR   X                 1 1 0 0 0 0    B410
000002               CLEAR   A                 1 1 0 0 0 0    B400
000004               CLEAR   S                 1 1 0 0 0 0    B440
000006               LDT     MAXLEN            1 1 0 0 1 0    77201F
000009      RLOOP    TD      INPUT             1 1 0 0 1 0    E3201B
00000C               JEQ     RLOOP             1 1 0 0 1 0    332FFA
00000F               RD      INPUT             1 1 0 0 1 0    DB2015
000012               COMPR   AOATS             1 1 0 0 0 0    A004
000014               JEQ     EXIT              1 1 0 0 1 0    332009
000017              +STCH    BUFFEROATX        1 1 1 0 0 1    57900000
00001B               TIXR    T                 1 1 0 0 0 0    B850
00001D               JLT     RLOOP             1 1 0 0 1 0    3B2FE9
000020      EXIT    +STX     LENGTH            1 1 0 0 0 1    13100000
000024               RSUB                      1 1 0 0 0 0    4F0000
000027      INPUT    BYTE    X'F1'                            F1
000000               EXTREF  LENGTH,BUFFER
000000               CLEAR   X                 1 1 0 0 0 0    B410
000002              +LDT     LENGTH            1 1 0 0 0 1    77100000
000006      WLOOP    TD      =X'05'            1 1 0 0 1 0    E32012
000009               JEQ     WLOOP             1 1 0 0 1 0    332FFA
00000C              +LDCH    BUFFEROATX        1 1 1 0 0 1    53900000
000010               WD      =X'05'            1 1 0 0 1 0    DF2008
000013               TIXR    T                 1 1 0 0 0 0    B850
000015               JLT     WLOOP             1 1 0 0 1 0    3B2FEE
000018               RSUB                      1 1 0 0 0 0    4F0000
00001B               END     FIRST
```

Fig. 24. OPTAB after Pass 2 (With opcodes)

2) **Performance:**

We found the performance our C as well as C++ codes. But as discussed earlier, the C code has just the basic functionalities and hence was implemented only in a single file. Whereas, the C++ code assembler has all the enhancements too and had a number of files. Hence, the execution time of both the codes differ greatly. We can see the results below:

VII. CONCLUSION AND FUTURE SCOPE

In this project, we have implemented the SIC/XE assembler in two languages: C and C++. We were able

```
H^000000^00001B
  Loading Address: 000000

D^BUFFER^000033^BUFEND^001033^LENGTH^00002D

R^RDREC^WRREC

T^000000^10^172027^4B100000^32023^292000^332007
T^000010^10^4B100000^3F2FEC^32016^F2016^12003
T^000020^0A^F200A^4B100000^3E2000

M^4^05^+RDREC
M^11^05^+WRREC
M^24^05^+WRREC

E^000000
```

```
H^000000^00001B
  Loading Address: 000000

R^BUFFER^LENGTH^BUFFEND

T^000000^0C^B410^B400^B440^77201F^E3201B
T^00000C^0F^332FFA^DB2015^A004^332009^57900000
T^00001B^0C^B850^3B2FE9^13100000^4F0000^F1

M^18^05^+BUFFER
M^21^05^+LENGTH
M^28^06^-BUFFER

E^000000
```

```
H^000000^00001B
  Loading Address: 000000

R^LENGTH^BUFFER

T^000000^10^B410^77100000^E32012^332FFA^53900000
T^000010^0B^DF2008^B850^3B2FEE^4F0000

M^3^05^+LENGTH
M^D^05^+BUFFER

E^000000
```

Fig. 25. Object program after execution of Pass 2

```
LOCATION  LABEL   OPERAND  OPCODE

 1000     MYPGM   START    1000
 1000             STA
 1003     LOOP1   JMP      LOOP2
 1006             LDA
 1009     LOOP2   JMP      LOOP1
 1012             RESB     04
 1016             LDA
 1019             STA
 1022             JMP      LOOP1
 1025             END

Corresponding Object code is..

H^ MYPGM ^ 1000 ^ 25
T^ 1000 ^8 ^^02^011009
T^ 1008 ^8 ^00^011003
T^ 1016 ^6 ^04^00^02
T^ 1022 ^6 011003
E^ 1000
```

Fig. 26. Output with opcode and object program

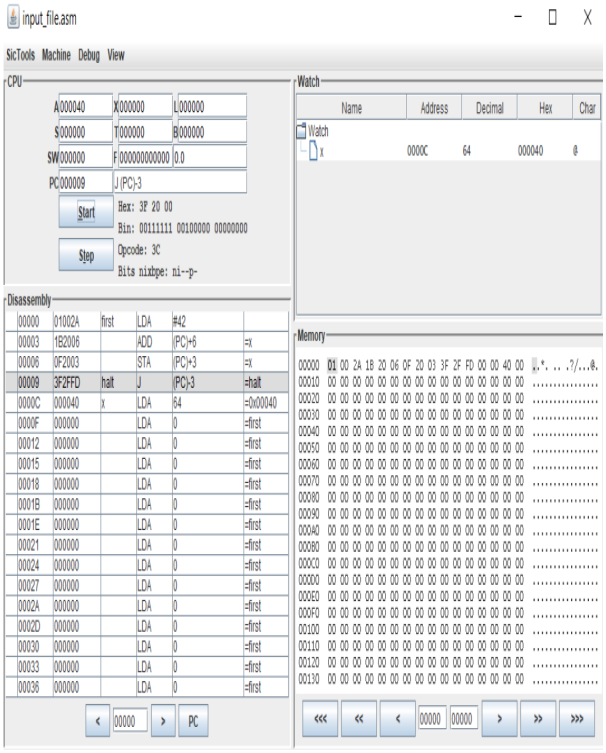

Fig. 27. SIC Tools Simulator

```
$ time g++ main.c

real    0m0.440s
user    0m0.108s
sys     0m0.310s
```

Fig. 28. Execution time of C Code

```
$ time g++ Assembler.cpp Assembler.h

real    0m14.450s
user    0m11.163s
sys     0m2.833s
```

Fig. 29. Execution time of C Code

to generate the Symbol table, Object code and the Object Program in our project successfully. By using the HashMap data structure, we were able to reduce the time complexity of searching in the Operation Table to O(1). Even though implementation through C language offers portability, the C++ implementation gives an overall optimal performance. As an enhancement to the SIC/XE assembler, we have done exception and error handling to achieve better results and

smooth operation. In future, we plan to implement a GUI to see the output of our assembler instead of using the command line. We also plan to do some visualization for comparing the performance using different programming languages and operating systems.

VIII. ACKNOWLEDGEMENT

We would like to thank our Professor Weider Yu, who gave us this exciting opportunity to make a project under him. Without his guidance and constructive criticism, we would not have completed this project.He also encouraged us to implement additional features like exception handling. Also, we would like to thank our peers who supported us in this venture

REFERENCES

[1] System Software, An Introduction to Systems Programming. Leland L. Beck
[2] https://www.cs.virginia.edu/ evans/cs216/guides/x86.html
[3] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Ninth ed., Wiley, 2013.