CSN-252:   
System Software  
Project Documentation  
SIC/XE Assembler

short lineSubmitted By:

## Ashutosh Kumar

## 21114021

## Batch-O1

# Introduction

SIC/XE stands for "Simplified Instructional Computer Extra Equipment". It is an advanced version of the SIC architecture that includes additional features and capabilities. SIC/XE is designed to be upward-compatible with SIC, which means that programs written for the SIC architecture can be run on the SIC/XE architecture without modification. This makes it easier for developers to migrate their programs to the new architecture.

# Assembler

The Assembler is implemented using C++ programming language.

This Assembler implements following Machine-Independent Features.

1. Literals

2. Symbol Defining Statements

3. Expressions

4. Control Sections

# Assembler : Design

## tables.cpp

This file contains the data structures required:

* The info\_op, info\_reg, info\_sym, info\_literal, and info\_mod structs are used to store information about opcodes, registers, symbols, literals, and modifications respectively.
* OPTAB map contains information about opcode mnemonics and their corresponding opcode values and formats as it is mapped to the info\_op struct. Similarly, the REGISTER map contains information about register mnemonics and their corresponding register numbers as it is mapped to the info\_op struct..
* The reg\_num function returns the corresponding register number given the register mnemonic.
* The init\_tables function initializes the OPTAB and REGISTER maps with their respective values.

## utility.cpp

It contains useful functions that will be required by other files .

* ***bool isNumber(string str)***: This function takes a string as input and returns a boolean indicating whether the string represents a decimal number or not. The function checks if each character in the string is a digit or not and returns false if any non-digit character is found.
* ***int strToDec(string s)***: This function takes a string representing a decimal number as input and returns the corresponding integer value. The function works by iterating over the characters in the string from the right end and multiplying each digit by a power of 10 and adding it to the result.
* ***int strToHex(string s)***: This function takes a string representing a hexadecimal number as input and returns the corresponding integer value. The function works similarly to the strToDec function, but instead multiplies each digit by a power of 16.
* ***int strToASCII(string s)***: This function takes a string as input and returns the corresponding integer value obtained by treating the string as a sequence of ASCII characters. The function iterates over the characters in the string from the right end and multiplies the ASCII value of each character by a power of 256 and adds it to the result.
* ***bool isHex(string str)***: This function takes a string as input and returns a boolean indicating whether the string represents a valid hexadecimal number or not. The function checks if each character in the string is a digit or a valid hexadecimal character (i.e., A-F or a-f).
* ***vector<string> parseWords(string line)***: This function takes a string representing a line of instructions as input and returns a vector containing three strings, which represent the three parameters of the instruction. The function splits the input string into words using spaces as delimiters and returns the first three words as a vector.
* ***string toHex(int num, int len)***: This function takes an integer value and a length as input and returns a string representing the hexadecimal value of the integer.

These functions are useful for processing strings of numbers and characters and converting them to their corresponding integer or hexadecimal representations. These operations are commonly used in computer programming and are essential for working with assembly language and machine code.

## csect.cpp

The given section contains two structs: instruction and csect.

* The instruction struct is used to represent an assembly instruction. It contains the following members:
  + address: An integer representing the location counter (locctr) of the instruction.
  + label: A string representing the label of the instruction.
  + mnemonic: A string representing the mnemonic of the instruction (e.g. "LDA", "ADD", etc.).
  + operand: A string representing the operand of the instruction (e.g. "X'0A'", "BUFFER", etc.).
  + objCode: A string representing the object code of the instruction, which will be generated during the assembly process.
  + length: An integer representing the length of the instruction in bytes.
* The csect struct is used to represent a control section, which is a portion of the program that is loaded into memory as a single unit. It contains the following members:
  + name: A string representing the name of the control section.
  + start: An integer representing the starting address of the control section.
  + end: An integer representing the ending address of the control section.
  + valid\_base: A boolean indicating whether a base register has been set for this control section.
  + def: A map that stores the defined symbols and their corresponding addresses.
  + ref: A vector that stores the names of the external symbols that are referenced in this control section.
  + litTab: A map that stores the literals used in the program and their corresponding information, such as the literal value and its length.
  + mod\_record: A vector that stores the modification records for this control section. A modification record indicates that a particular memory location needs to be modified at link-edit time.
  + instructions: A vector that stores the instructions of the program in this control section.
  + symTab: A map that stores the symbols used in the program and their corresponding information, such as their addresses and whether they are relative or absolute.

These structs are used to store information about the program being assembled, including its instructions, symbols, and control sections. The information stored in these structs is used during the assembly process to generate the object code for the program.

## pass1.cpp

It takes an input file that contains assembly code as input and generates a symbol table, literal table, and an intermediate code file. This is the first step in the process of assembling code for execution on a computer.

The code starts by opening the input file and error file. If the input file is not found, an error message is displayed and the program exits. The input file is read line by line and each line is stored in a vector.

The program then initializes several variables, including a location counter, vectors for instructions and literals, and a vector for control sections. A control section is defined by the presence of a label, and all instructions within that section are assigned to that label.

The program then begins processing each line of the input file. The line is parsed into its label, mnemonic, and operand. If a control section has not yet been defined, the program creates one based on the label of the first instruction. The instruction is then added to the instruction vector for the current control section.

The program then checks the mnemonic to determine how to handle the instruction. If the mnemonic is "START", the location counter is set to the value of the operand (converted from hexadecimal if necessary), and the start address for the current control section is set to the new location counter value.

Similarly, It processes all the different instructions and assembler directives.

After processing all instructions in the input file, the program writes the symbol and literal tables to the intermediate file, along with the instruction addresses and lengths.

## pass2.cpp

Since the addresses have been assigned in Pass1, in Pass 2 we are first writing the complete intermediate file by calculating the object codes of respective instructions and storing in the structs.

Coming to the major function of the pass2 in this assembler whose code is in the function createObjProg() which is explained below:

The function starts by opening an output file named object\_program.txt. It then iterates through all the control sections in the source code. For each control section, it writes the header record in the object file. The header record contains the section name, starting address, and length of the section.

After that, the function writes definition records for all the symbols defined in the control section. The definition record contains the symbol name and its address. Similarly, it writes reference records for all the symbols referred to in the control section.

Then the function writes the text records for the instructions in the control section. It iterates through all the instructions and checks whether the instruction has a corresponding object code or not. If it has, it calculates the current size of the object code and checks if the object code can fit in the current text record or not. If it can, it adds the object code to the current text record. Otherwise, it writes the current text record to the object file and starts a new text record for the current instruction.

If the current instruction is a reserve word (RESW or RESB), it writes the current text record (if it has any object code) to the object file and starts a new text record for the next instruction.

After writing all the text records, the function writes the modification records for all the instructions with immediate addressing.

Finally, the function writes the end record in the object file. If it is the last control section, it writes the end record with the starting address of the first control section. Otherwise, it writes the end record without any address.

At the end of the function, the output file and intermediate file are closed, and a message is printed to indicate that the object program file has been written.

# Assembler : Data Structures

## REGTAB :

Contains information of the registers like its numeric equivalent , character representing , whether such register exists or not .

## LITTAB :

Contains information of literals like its value , address , block number , a character representing whether the literal exists in literal table or not . It has also been shown in “***lit\_tab.txt***”

## SYMTAB :

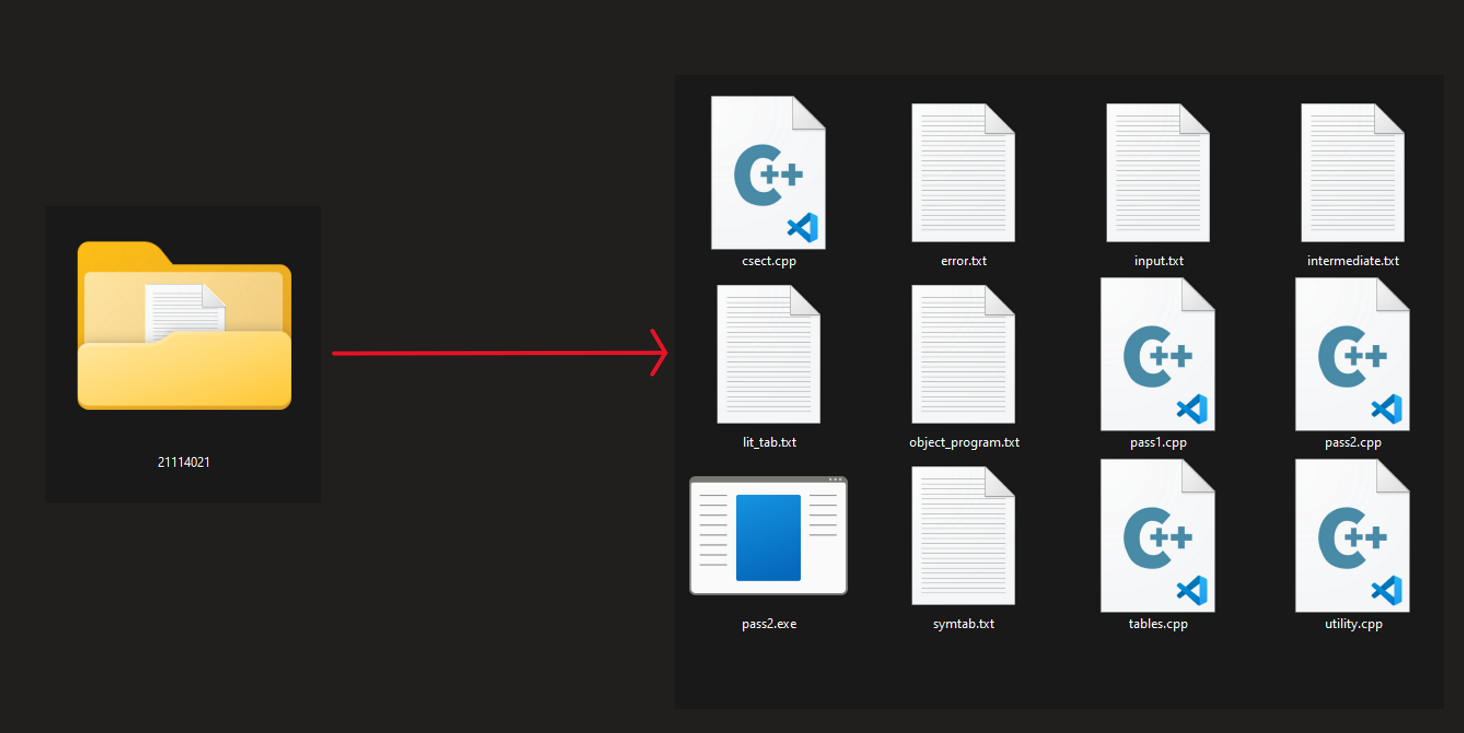
Contains information of labels like , name , address , block number , character representing whether the label exists in the symbol table or not , an integer representing whether label is relative or not . It has also been shown in “***symtab.txt***”

## OPTAB :

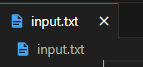
Contains information of opcode like name , format , a bool representing whether the opcode is valid or not .

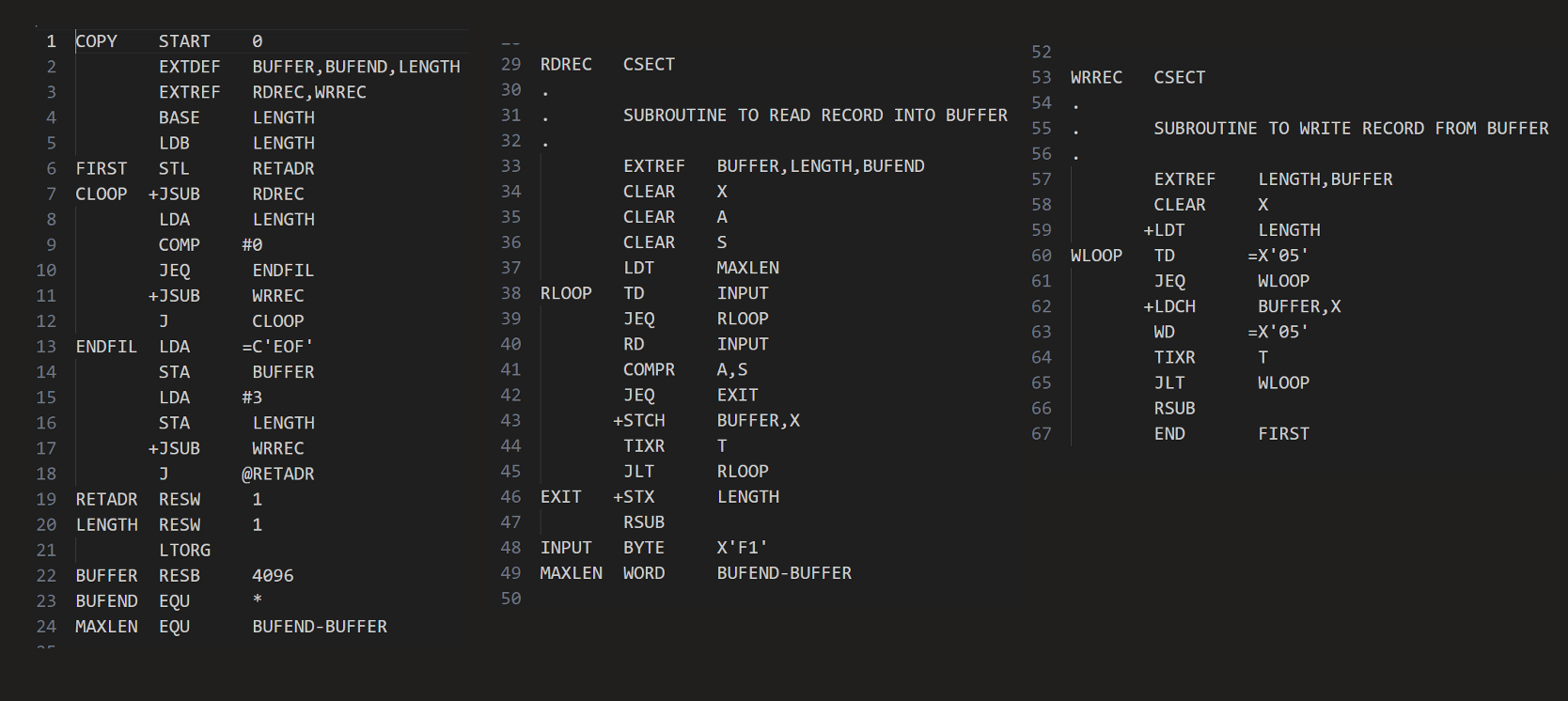
# Compilation and Execution

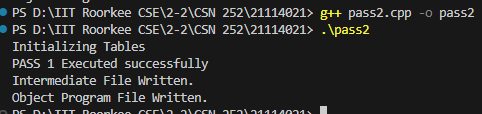
## To compile and execute the SIC/XE Assembler, follow these steps:

1. Unzip the Attached File “**21114021.zip**”.
2. Upon unzipping the above folder will contain the following files.
3. Open Terminal at that location and type “**code .**” to open the directory in VS Code.



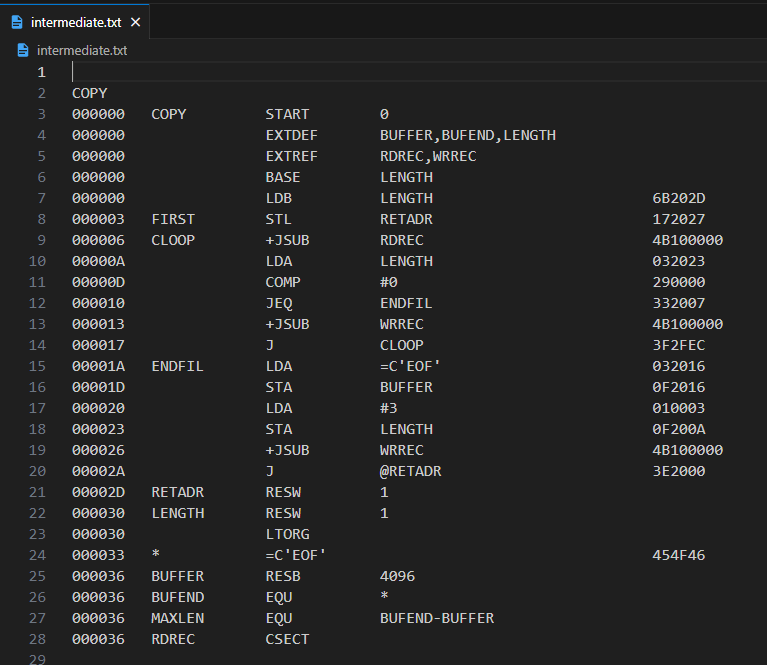
1. In VS Code, If a user wants to change the input then can change in “**input.txt**”. The Current input is:



1. To run the code type the following command in Terminal.

Terminal Outputs shows that the programs have run successfully.

1. This shows that the corresponding intermediate file and Object Program file has been generated in “**intermediate.txt**” and “**object\_program.cpp**” respectively. It also generates “**error.txt**” if there are any errors.





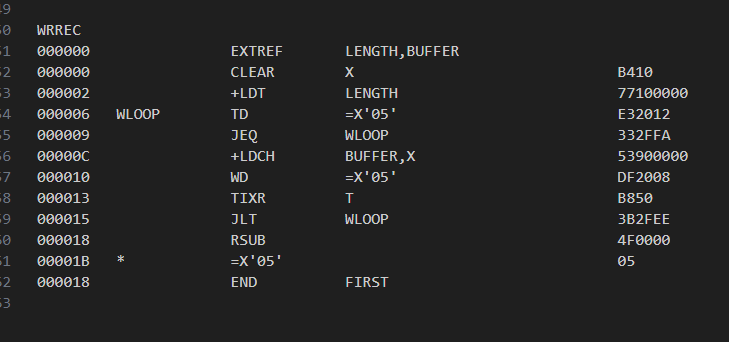


Fig.: intermediate.txt

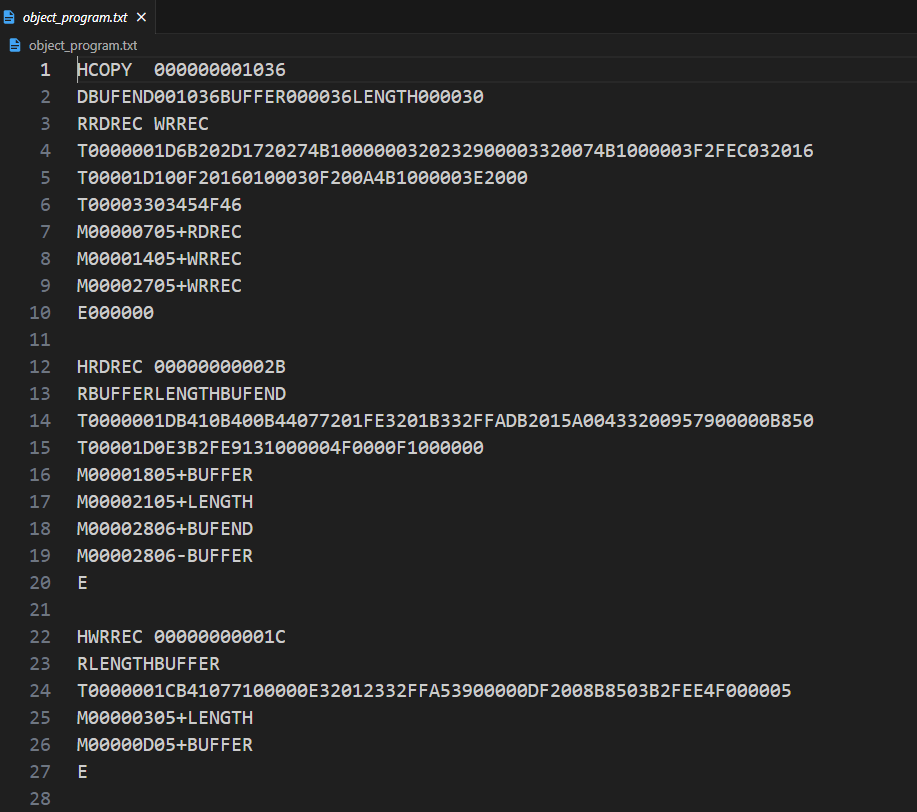


Fig.: object\_program.txt

1. This Assembler also writes LITTABs and SYMTABs of respective control section.

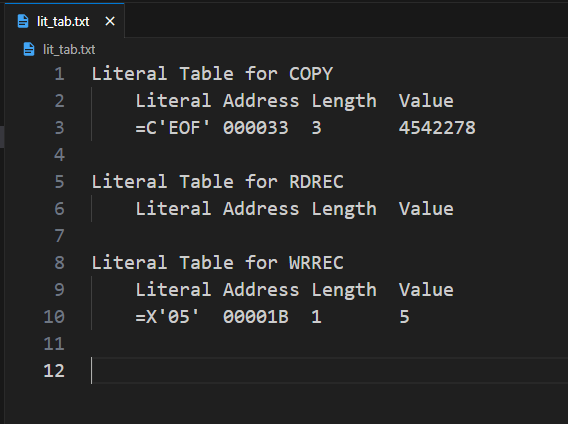


Fig.: lit\_tab.txt

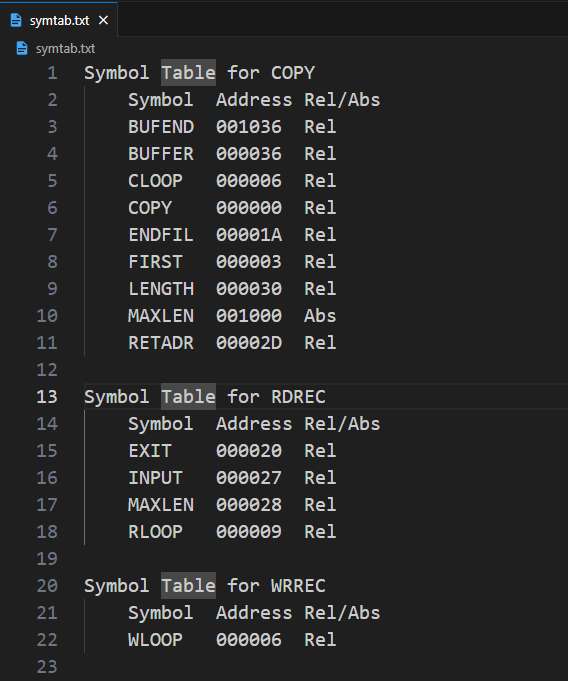


Fig.: symtab.txt

1. This completes the execution of our assembler.

short dash