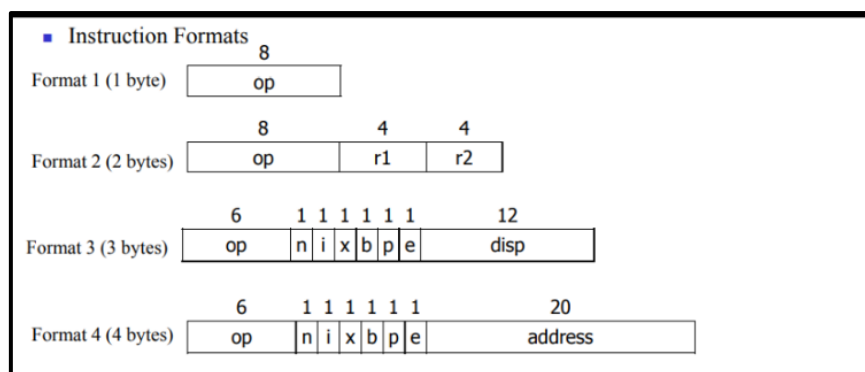


SIC/XE Assembler Project

CSN-252 Tutorial-8

- SIC/XE stands for Simplified Instructional Computer Extra Equipment or Extra Expensive . This computer is an advance version of SIC . Both SIC and SIC/XE are closely related to each other that's why are upward compatible .
- **Memory** : Memory consists of 8 bit-bytes and the memory size is 1 megabytes (220 bytes). Standard SIC memory size is very small. This change in the memory size leads to change in the instruction formats as well as addressing modes. 3 consecutive bytes form a word (24 bits) in SIC/XE architecture. All address are byte addresses and words are addressed by the location of their lowest numbered byte.
- **Registers** : It contains 9 registers (5 SIC+ 4 Additional Register). Four additional registers are :
 1. B : base register
 2. S : General working register
 3. T : General working register
 4. F : Floating point Accumulator
- **Data Formats** : Integers are represented by binary numbers . Characters are represented using ASCII codes . Floating points are represented using 48 bits .
- **Instruction Formats** : In SIC/XE architecture there are 4 types of formats available. The Bit(e) is used to distinguish between Formats 3 and 4 . e = 0 means Format 3 and e = 1 means Format 4 .



- This Assembler also includes Machine – Independent Assembler Features.
 1. Literals
 2. Symbol Defining Statements
 3. Expressions
 4. Control Sections
- The Assembler is implemented our assembler using C++ programming language.
- The Assembler uses C++ library ifstream to read input from a file and write output on another file.

ASSEMBLER DESIGN

Functions.cpp

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

- 1) **getString**: Converts a single character to a string.
- 2) **intToStringHex**: Converts an integer to a hexadecimal string with specified fill width.
- 3) **expandString**: Expands or truncates a string to a specified length, adding or removing characters as necessary.
- 4) **stringHexToInt**: Converts a hexadecimal string to an integer.
- 5) **stringToHexString**: Converts a string to its hexadecimal representation.
- 6) **checkWhiteSpace**: Checks if a character is a white space character.
- 7) **checkCommentLine**: Checks if a line is a comment line.
- 8) **if_all_num**: Checks if all characters in a string are digits.
- 9) **readFirstNonWhiteSpace**: Reads the first non-white space characters from a string.
- 10) **readByteOperand**: Reads a byte operand from a string, handling character or hexadecimal byte format.
- 11) **writeToFile**: Writes data to a file, optionally appending a newline character.
- 12) **getRealOpcode**: Extracts the real opcode from a formatted opcode string.
- 13) **getFlagFormat**: Determines the flag format of a string.
- 14) **EvaluateString** (class): Evaluates arithmetic expressions provided as strings, supporting addition, subtraction, multiplication, division, and parentheses.

Tables.cpp

-It contains all the data structures required for our assembler to run .

- 1) **struct_extdef and struct_extref**: These structures represent external definitions and references, respectively. They store the name, address, and existence status of external symbols.
- 2) **struct_csect**: This structure represents a control section. It holds the name, location counter (LOCCTR), section number, length, and tables for external definitions (EXTDEF_TAB) and external references (EXTREF_TAB) within the control section.
- 3) **struct_opcode**: This structure represents an opcode. It contains the opcode value, format (e.g., format 1, format 2, or format 3), and existence status.

4)**struct_literal**: This structure represents a literal value. It stores the value, address, existence status, and block number.

5)**struct_label**: This structure represents a label. It stores the address, name, relative flag, block number, and existence status of a label.

6)**struct_blocks**: This structure represents a block. It includes the start address, name, location counter (LOCCTR), block number, and existence status.

7)**struct_register**: This structure represents a register. It stores the register number and existence status.

8)The code also defines several typedefs for different table types:

SYMBOL_TABLE_TYPE: A map from symbol names to struct_label.

OPCODE_TABLE_TYPE: A map from opcode names to struct_opcode.

REG_TABLE_TYPE: A map from register names to struct_register.

LIT_TABLE_TYPE: A map from literal values to struct_literal.

BLOCK_TABLE_TYPE: A map from block names to struct_blocks.

CSECT_TABLE_TYPE: A map from control section names to struct_csect.

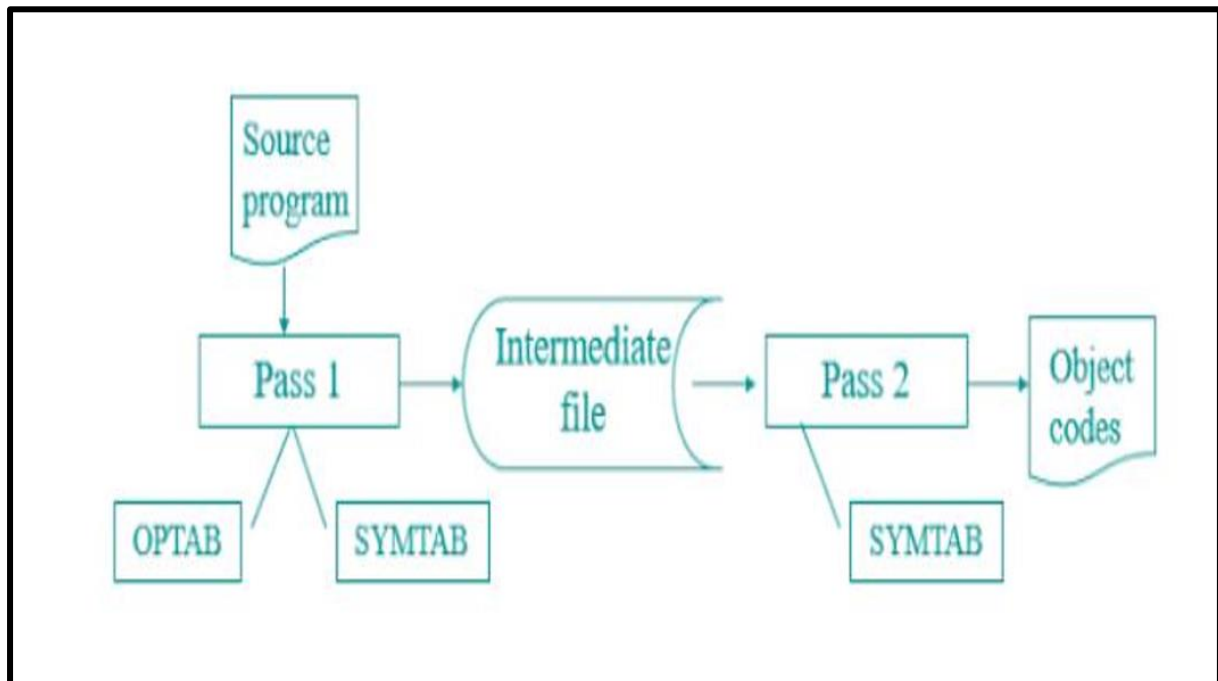
Finally, the functions load_REGTAB(), load_OPTAB(), load_BLOCKS(), and load_tables() initialize the opcode table (OPTAB), register table (REGTAB), block table (BLOCKS), and other tables respectively, with predefined values.

Firstpass.cpp

We update the error file and the intermediate file using the source file. If we are unable to find the source file, or if the intermediate file doesn't open, we write the corresponding error in the error file; otherwise, we print it to the console. We declare many variables that will be required later. Then, the assembler will take the first line as input and check if it is a comment line or not. As long as the lines are comments, the assembler prints them to the intermediate file and accordingly updates the line number. Once the line is not a comment, we check if the opcode is START. If yes, we update the line number, LOCCTR, and start address. If not found, we initialize the start address and LOCCTR as 0. Inside the inner loop, we check if the line is a comment. If it is a comment, we print it to our intermediate file, update the line number, and take in the next input line. If not a comment, we check if there is a label in the line. If present, we check if it is present in the SYMTAB. If found, we print an error saying 'Duplicate symbol' in the error file; otherwise, we assign the name, address, and other required values to the symbol and store it in the SYMTAB.

Second.cpp

Assembler takes in the intermediate file as input and generate the intermediate file and the object program . Similar to pass1 , if the Assembler is unable to open the file, it will print an error message in error file. We then read the first line of the intermediate file. Until the lines are comments, we take them as input and print them to our intermediate file and update our line number. If we get opcode as 'START', we initialize our start address as the LOCCTR, and write the line into the listing file. Then we check that whether the number of sections in our intermediate file was greater than one, if so, then we update our program length as the length of the first control section or else we keep the program length unchanged. For writing the end record we have function . For instructions with immediate addressing , we will write the modification record. function_for_reading_till_TAB() : takes in the string as input and reads the string until tab('\t') occurs. function_for_reading_the_intermediate_file() : Takes in location counter, opcode , operand , label . If the line is comment returns true and takes in the next input line .

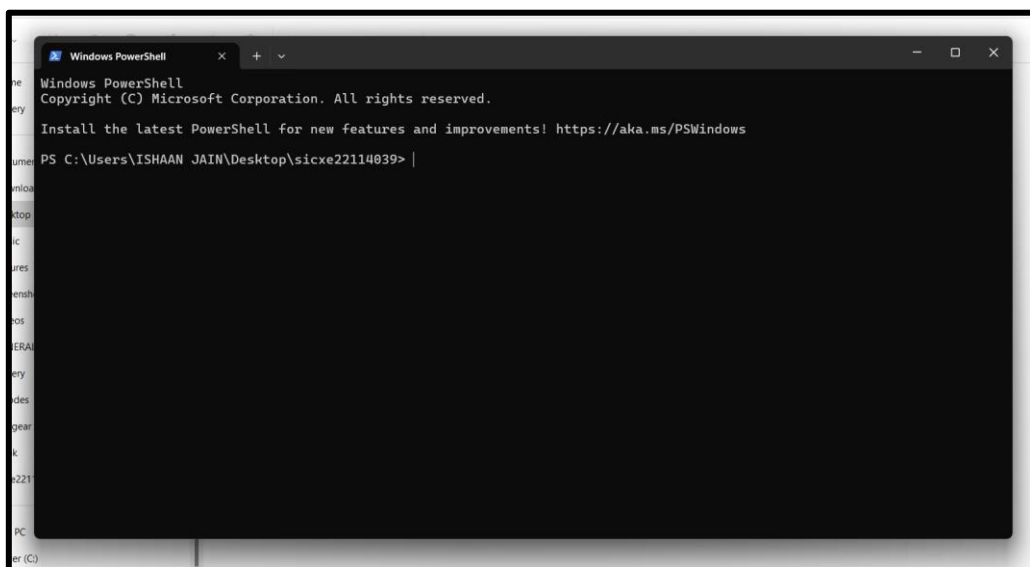
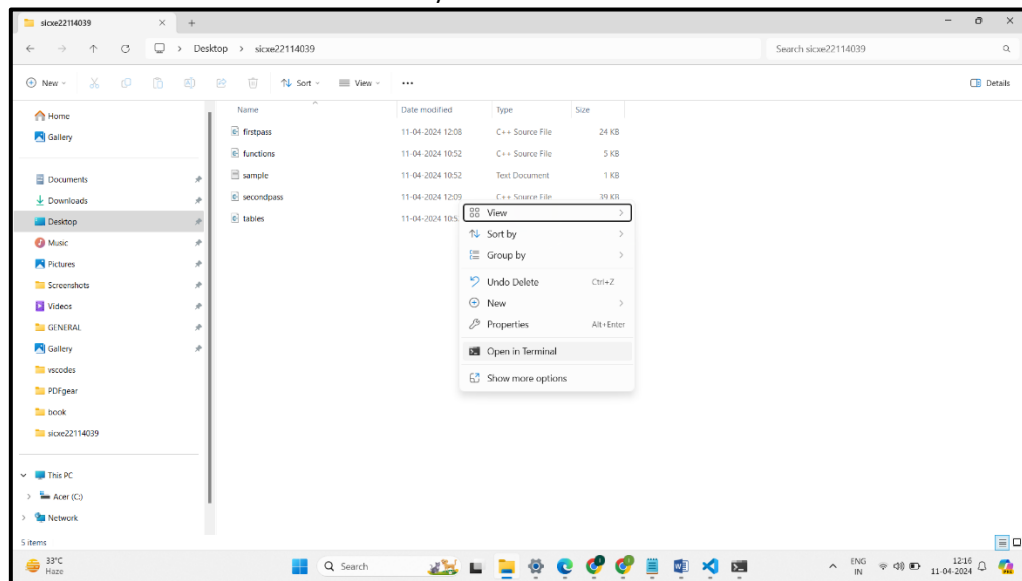


HOW TO RUN(README)

My Enrollment No. is odd and hence, my assembler implements all the required SIC/XE functionalities with Control Section. The source code file contains `secondpass.cpp`, `firstpass.cpp`, `functions.cpp` and `tables.cpp`.

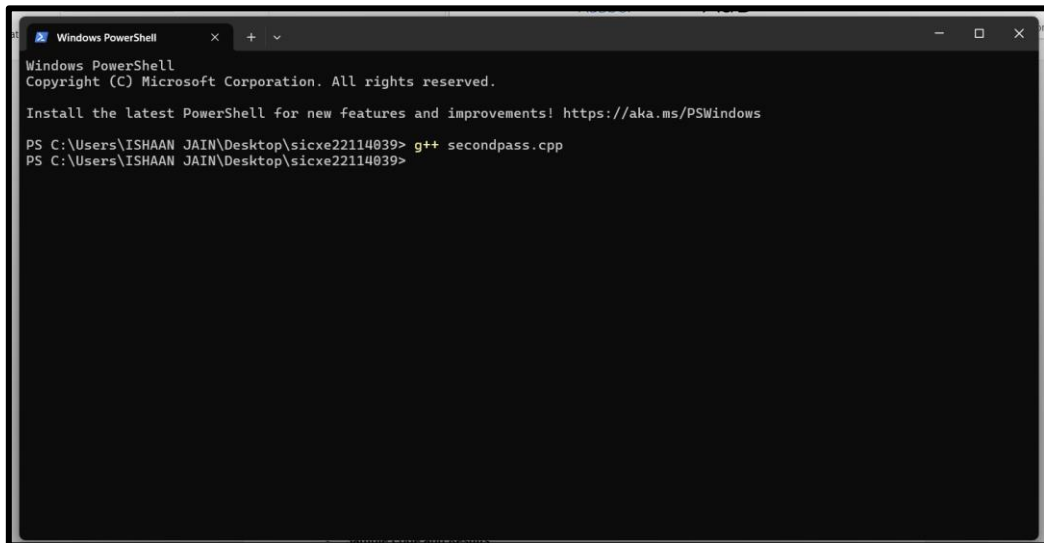
- How to Run the Code

1. Extract the `sicxe22114039.zip` file. (right click on downloaded file and tap extract all.)
2. Open terminal in the `sicxe22114039` directory.



Ishaan Jain (22114039)
CSE 2nd Year

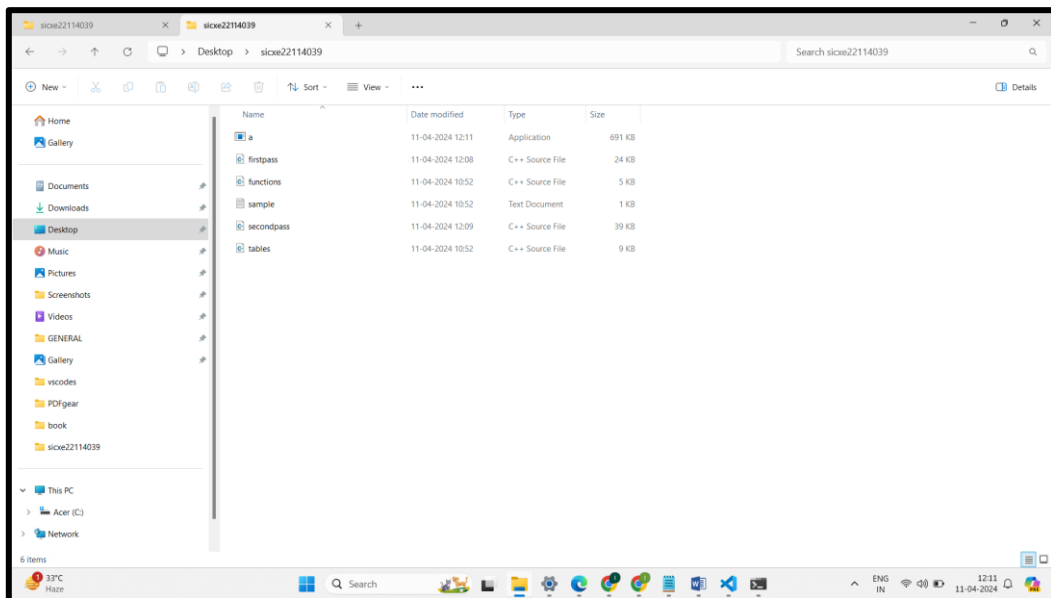
3. Compile the code using the command: `g++ secondpass.cpp`, the compiled program will be written to `a.exe` or `a.out` depending on the operating system.



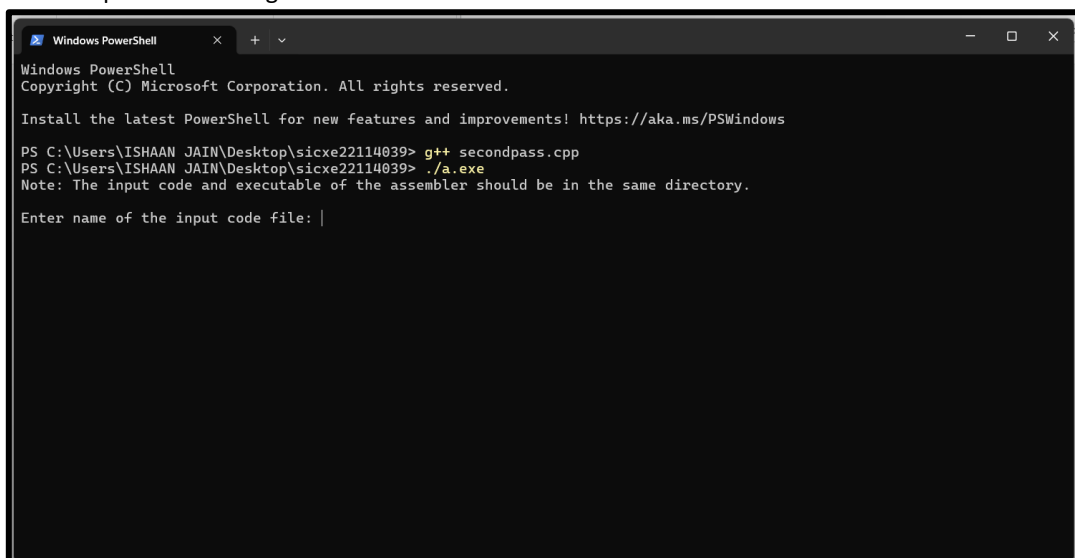
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> g++ secondpass.cpp
PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039>
```



4. Run the compiled code using the command: `./a.exe` or `./a.out`

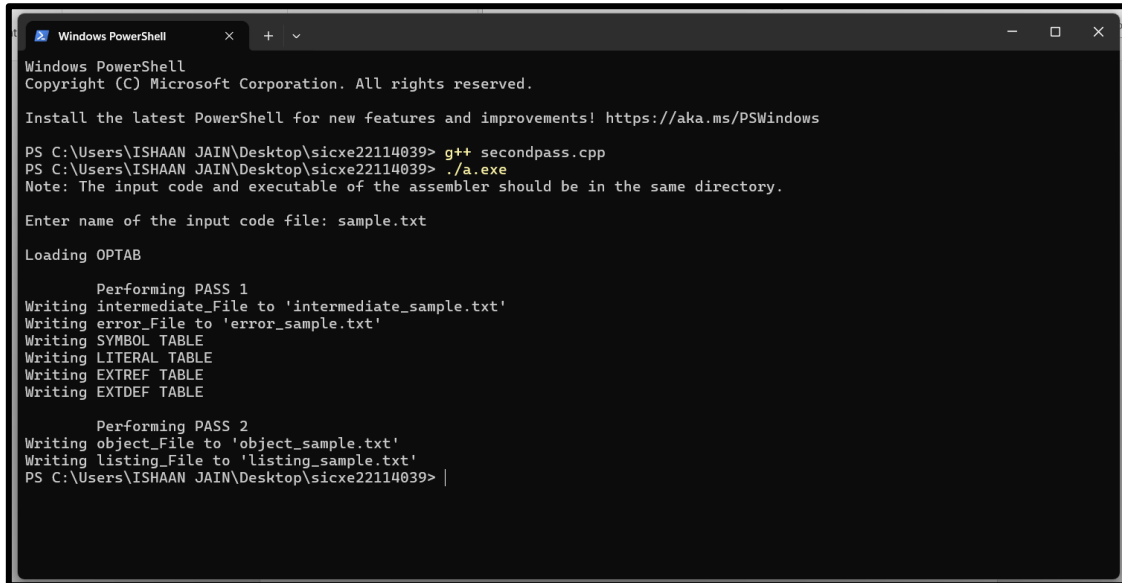


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> g++ secondpass.cpp
PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> ./a.exe
Note: The input code and executable of the assembler should be in the same directory.
Enter name of the input code file: |
```

5. Enter the assembly code file name in the prompt given on running the program.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> g++ secondpass.cpp
PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> ./a.exe
Note: The input code and executable of the assembler should be in the same directory.

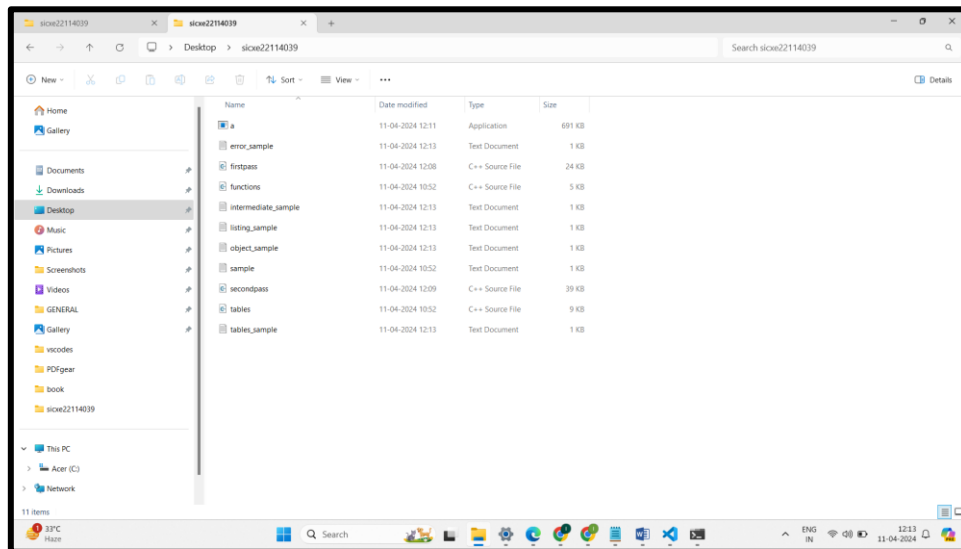
Enter name of the input code file: sample.txt

Loading OPTAB

Performing PASS 1
Writing intermediate_File to 'intermediate_sample.txt'
Writing error_File to 'error_sample.txt'
Writing SYMBOL TABLE
Writing LITERAL TABLE
Writing EXTREF TABLE
Writing EXTDEF TABLE

Performing PASS 2
Writing object_File to 'object_sample.txt'
Writing listing_File to 'listing_sample.txt'
PS C:\Users\ISHAAN JAIN\Desktop\sicxe22114039> |
```

6. The different results; intermediate, listing, object, tables, errors are written suffixed with the input file name.



Sample Code and Results

Sample Code 1- (sample.txt): Question 3 of Section 2.2

```
sample.txt
File Edit View

SUM      START  0
FIRST    LDX    #0
          LDA    #0
          +LDB   #TABLE2
          BASE   TABLE2
LOOP     ADD    TABLE, X
          ADD    TABLE2, X
          TIX    COUNT
          JLT    LOOP
          +STA   TOTAL
          RSUB
COUNT   RESW   1
TABLE    RESW   2000
TABLE2   RESW   2000
TOTAL    RESW   1
          END    FIRST
```

```
intermediate_sample.txt
File Edit View

Line Address Label OPCODE OPERAND Comment
5 00000 0 SUM START 0
10 00000 0 FIRST LDX #0
15 00003 0 LDA #0
20 00006 0 +LDB #TABLE2
25 0000A 0 BASE TABLE2
30 0000A 0 LOOP ADD TABLE,X
35 0000D 0 ADD TABLE2,X
40 00010 0 TIX COUNT
45 00013 0 JLT LOOP
50 00016 0 +STA TOTAL
55 0001A 0 RSUB
60 0001D 0 COUNT RESW 1
65 00020 0 TABLE RESW 2000
70 01790 0 TABLE2 RESW 2000
75 02F00 0 TOTAL RESW 1
80 02F03 END FIRST
```

```
tables_sample.txt
File Edit View

--- SYMBOL TABLE ---

:- name:undefined |address:0 |relative:00000
0:- name: |address:0 |relative:00000
COUNT:- name:COUNT |address:0001D |relative:00001
FIRST:- name:FIRST |address:00000 |relative:00001
LOOP:- name:LOOP |address:0000A |relative:00001
TABLE:- name:TABLE |address:00020 |relative:00001
TABLE2:- name:TABLE2 |address:01790 |relative:00001
TOTAL:- name:TOTAL |address:02F00 |relative:00001

--- LITERAL TABLE ---

--- EXTREF TABLE ---

--- EXTDEF TABLE ---
```

Assembled Results of Sample Code 1:

```
error_sample.txt
File Edit View

--- PASS 1 ---

--- PASS 2 ---
```

```
object_sample.txt
File Edit View

H^SUM ^0000000^002F03
T^0000000^1D^050000010000691017901BA0131BC0002F200A3B2FF40F102F004F0000
M^0000007^05
M^0000017^05
E^0000000
```


Additional Notes

1. The two sample input assembly code files are provided in the folder with names `sample.txt`
 2. The results files of the two sample codes generated by the assembler are saved inside the folder `sample_output`.
 3. The README is provided both in `.docx` and `.pdf` format.
-
-