CSN-252

TUTORIAL – 8

NAME : MURTHATI MAHIBABU.

ROLL NO: 20114058.

SUB-BATCH : O3.

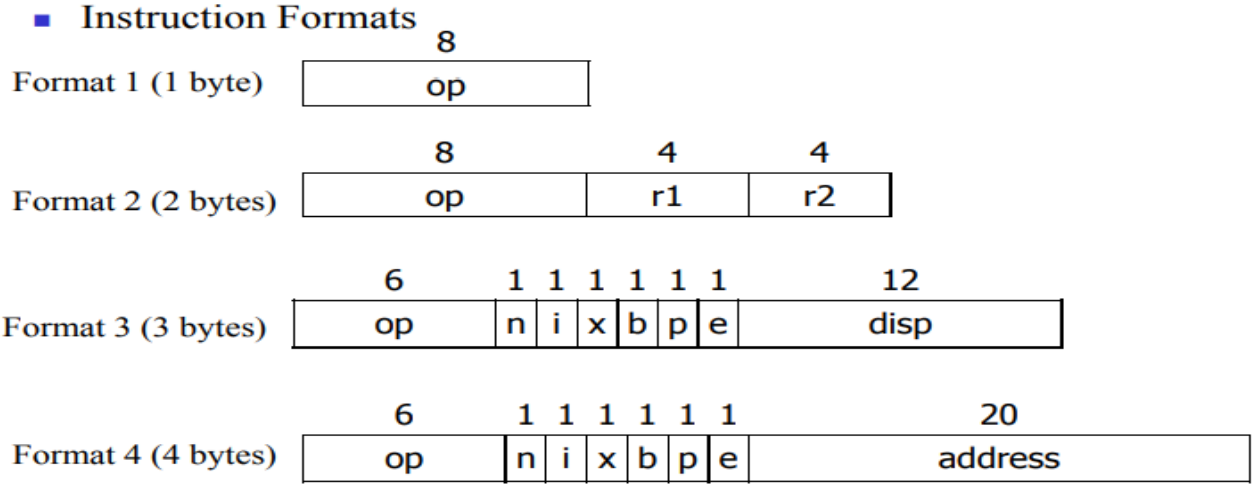
CSE- DEPT.

INTRODUCTION :

The Objective of the project is to implement a version of two-pass SIC/XE assembler: Pass 1 and Pass 2.

The Assembler we implemented includes all the SIC/XE instructions and supports all four formats 1, 2, 3, 4, addressing modes and program relocation.

- Instruction Formats



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

Addressing modes

- Base relative (n=1, i=1, b=1, p=0)
- Program-counter relative (n=1, i=1, b=0, p=1)
- Direct (n=1, i=1, b=0, p=0)
- Immediate (n=0, i=1, x=0)
- Indirect (n=1, i=0, x=0)
- Indexing (both n & i = 0 or 1, x=1)
- Extended (e=1 for format 4, e=0 for format 3)

It also includes all Machine-Independent Assembler Features-

1. Literals
2. Symbol Defining Statements
3. Expressions
4. Program Blocks
5. Control Sections and Program Linking

Sequential flow of Execution :

Input : Assembler source program following SIC-XE instruction set

Output :

PASS 1 :

- In Pass-1, assembler generates a Symbol table and intermediate file for Pass-2.

PASS 2:

- Pass 2 will generate a listing file containing the input assembly code and address, block number, object code of each instruction and also it will generate an object program including following type of record: H, D, R, T, M and E types

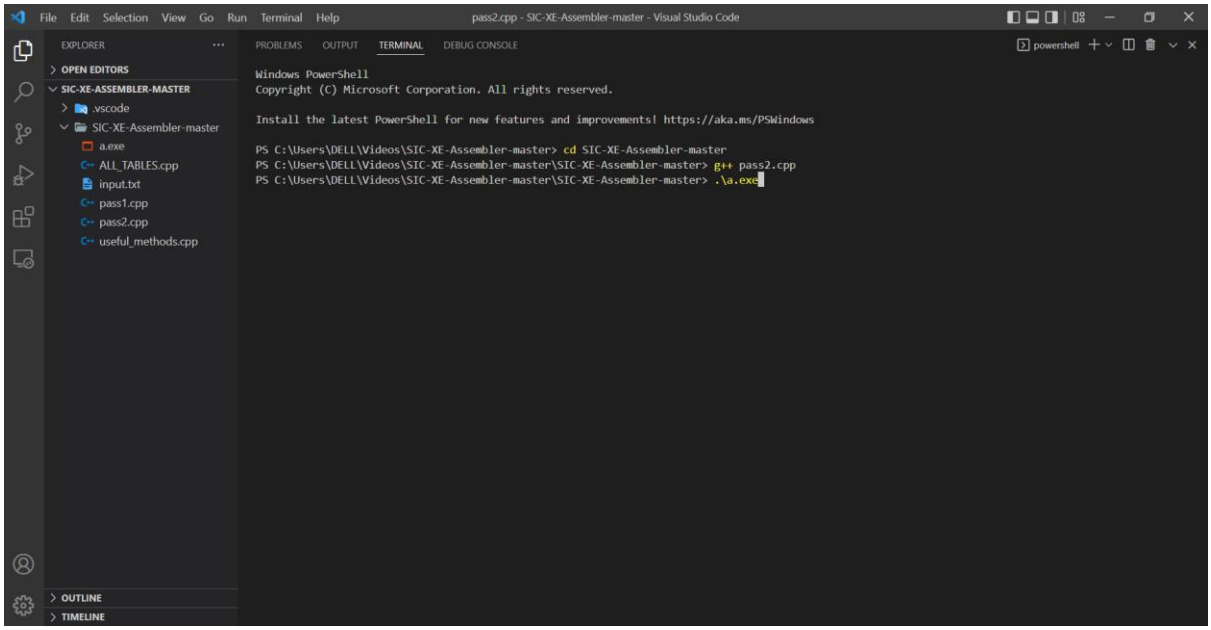
ERROR :

An error file is also generated displaying the errors in the assembly program (if any)

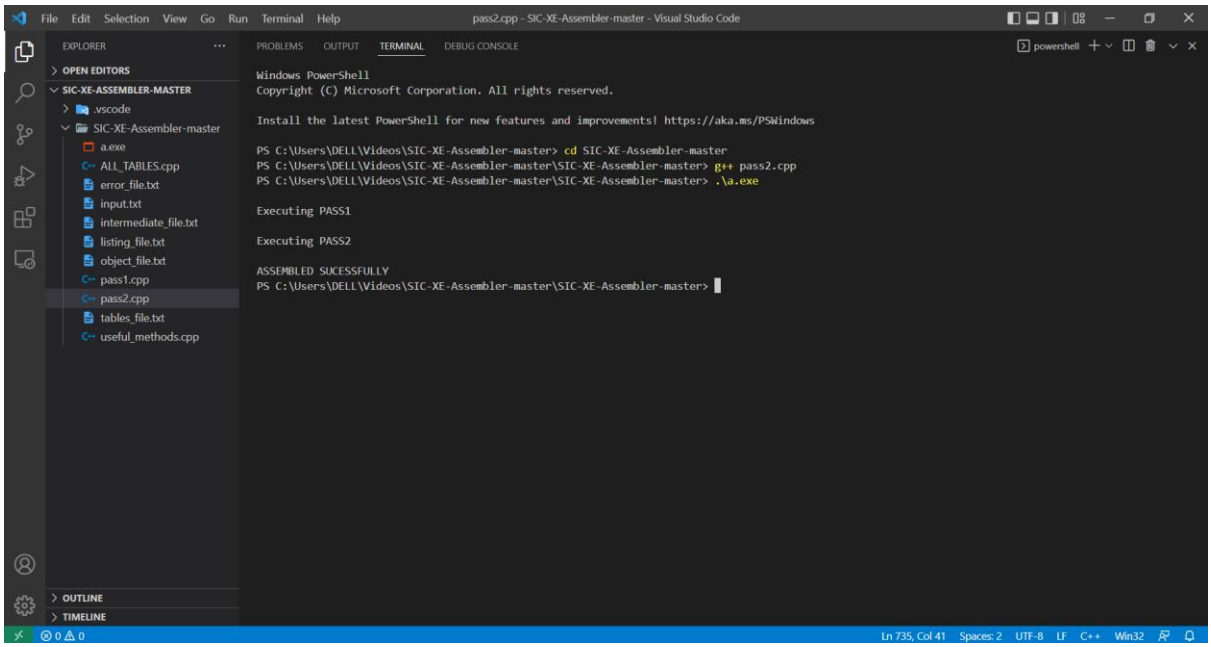
Steps to Compile and Execute Assembler :

(We must make sure that g++ compiler is installed)

- Initially we have to compile “pass2.cpp” file by “g++ pass2.cpp” command:



- Then we have to execute the object program “a.exe” generated by compilation of “pass2.cpp” file:



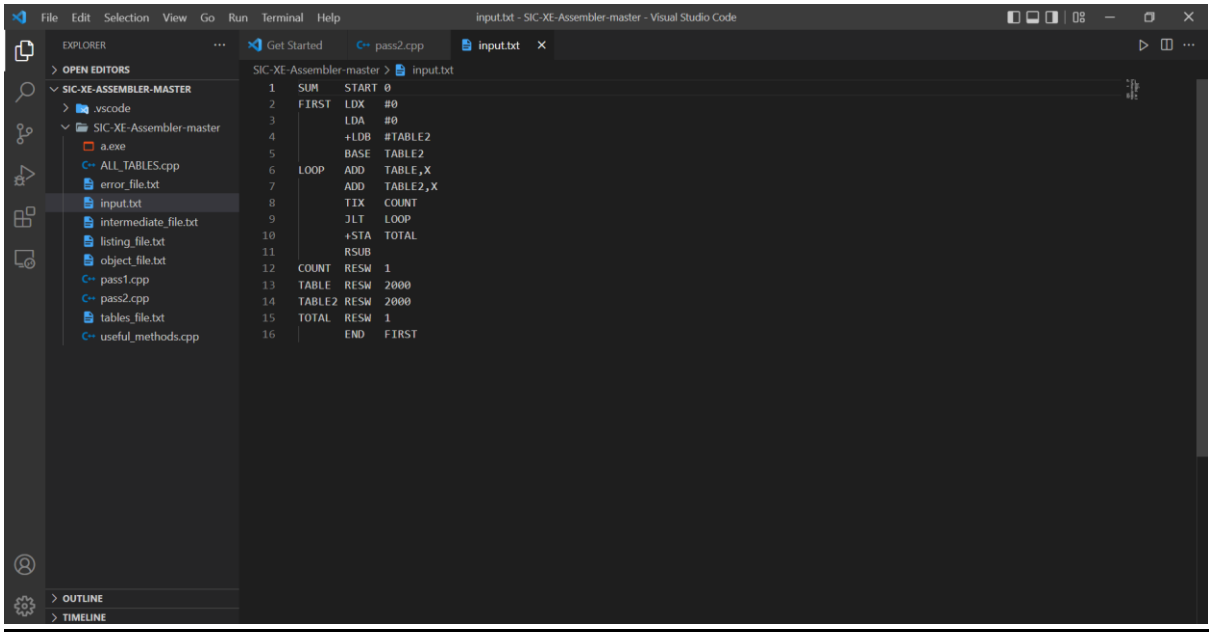
- During execution of program ,“errors, intermediate, listing files and object programs” were Generated automatically

IMPLEMENTATION TECH :

- We have implemented our assembler using C++ programming language. We also have used c++ library fstream to read input from a file and a write output in another file.

EXAMPLE PROGRAM :

- We have provided an example assembly program in file “input.txt” to test our assembler.



Architecture of the software

Functions:

PASS1 :

pass1()–

we update the intermediate file and error file using source file. If we are unable to find the source file or else if the intermediate file doesn’t open, we write the corresponding error in the error file and if error file doesn’t open, we print it to console. We declare the variables required. Then we take the first line as input, check if it is a comment line. Until the lines are comments, we take them as input and print them to our intermediate file and update our line number. Once, the line is not a comment we check if the opcode is ‘START’, if found, we update the line number, LOCCTR and start address if not

found, we initialize start address and LOCCTR as 0. Then, we use two nested while() loops, in which the outer loop iterates till opcode equals 'END' and the inner loop iterates until, we get our opcode as **'END' or 'CSECT'**. Inside the inner loop, we check if line is a comment. If comment, we print it to our intermediate file, update 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 error saying 'Duplicate symbol' in the error file or else assign name, address and other required values to the symbol and store it in the SYMTAB. Then, we check if opcode is present in the **OP_TAB**, if present we find out its format and then accordingly increment the LOCCTR. If not found in **OP_TAB**, we check it with other opcodes like **'WORD', 'RESW', 'BYTE', 'RESBYTE', 'LTORG', 'ORG, 'BASE', 'USE', 'EQU', 'EXTREF' or 'EXTDEF'**. Accordingly, we insert the symbols, external references and external definitions in the SYMTAB or the map for the control section which we created. For instance, for opcodes like USE, we insert a new BLOCK entry in the BLOCK map as defined in the **ALL_TABLES.cpp** file, for LTORG we call the **handle_LTORG()** function defined in pass1.cpp, for 'ORG', we point out LOCCTR to the operand value given, for EQU , we check if whether the operand is an expression then we check whether the expression is valid by using the **evaluate_expression()** function, if valid we enter the symbols in the SYMTAB. And if the opcode doesn't match with the above given opcodes, we print an error message in the error file. Accordingly, we then update our data which is to be written in the intermediate file. After the ending of the while loop for control section, we update our **CSECT_TAB**,the values for labels, LOCCTR, start_addr and length, and head on for the next control section until the outer loop ends. After the loop ends, we store the program length and then go on for printing the **SYMTAB, LITTAB** and other tables for control sections if present. After that we move on to the pass2().

handle_LTORG()- It uses pass by reference. We print the literal pool present till time by taking the arguments from the pass1() function. We run an iterator to print all the literals present in the LITTAB and then update the line number. If for some literal, we did not find the address, we store the present address in the LITTAB and then increment the LOCCTR on the basis of literal present.

evaluate_expression()- It uses pass by reference. We use a while loop to get the symbols from the expression. If the symbol is not found in the SYMTAB, we keep the error message in the error file. We use a variable pair_count which keeps the account of whether the expression is absolute or relative and if the pair_count gives some unexpected value, we print an error message.

ALL_TABLES :

It contains all the data structures required for our assembler to run. It contains the Classes for **info_label, info_op**, **literal**, **blocks**, **extdef**, **extref** and **csect**. The **CSECT_Tab** contains Maps are defined for various tables with their indices as strings with the names of the labels or opcodes as required.

Useful_methods :

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

get_str()- takes in input as a character and returns a string.

int_to_strHex()- takes in input as int and then converts it into its hexadecimal equivalent with string data type.

expand_str()- expands the input string to the given input size. It takes in the string to be expanded as parameter and length of output string and the character to be inserted in order to expand that string.

str_hex_to_int()- converts the hexadecimal string to integer and returns the integer value.

str_to_hex_str()- takes in string as input and then converts the string into its hexadecimal equivalent and then returns the equivalent as string.

Is_space ()- checks if blanks are present. If present, returns true or else false.

Is_comment ()- check the comment by looking at the first character of the input string, and then accordingly returns true if comment or else false.

if_all_num()- checks if all the elements of the string of the input string are number digits.

read_first_non_space ()- takes in the string and iterates until it gets the first non-spaced character. It is a pass by reference function which updates the index of the input string until the blank space characters end and returns void.

write_to_file()- takes in the name of the file and the string to be written on to the file. Then writes the input string onto the new line of the file.

get_real_opcode()-for opcodes of format 4, for example +JSUB the function will see whether if the opcode contains some additional bit like '+' or some other flag bits, then it returns the opcode leaving the first flag bit.

get_flag_format()- returns the flag bit if present in the input string or else it returns null string.

Class Eval_str – contains the functions :

-peek()- returns the value at the present index.

-get()- returns the value at the given index and then increments the index by one.

-number()- returns the value of the input string in integer format

PASS2 :

-pass2()-

We take in the **intermediate_file** as input using the **read_intermediate_file()** function and generate the **listing_file** and the **object_file**. Similar to pass1, if the intermediate_file is unable to open, we will print the error message in the error file. Same with the object_file if unable to open. 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 out start_addr 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. We then write the first header record in the object program. Then until the opcode comes as **'END'** or **'CSECT'** if the control sections are present, we take in the input lines from the intermediate_file and then update the listing_file and then write the object program in the text record using the textrecord() function. We will write the object code on the basis of the types of formats used in the instruction. Based on different types of opcodes such as **'BYTE','WORD','BASE','NOBASE','EXTDEF','EXTREF','CSECT'**, we will generate different types of object codes. For the format 3 and format 4 instruction format, we will use the **create_obj_code_format_34()** function in the **pass2.cpp**. For writing the end record, we use the **write_end_rec()** function. If control sections are present, we will use the **write_R_rec()** and **write_D_rec()** to write the external references and the external definitions. For the instructions with immediate addressing, we will write the modification record. When the inner loop for the control

section finishes, we will again loop to print the next section until the last opcode for **'END'** occurs.

read_till_tab()- takes in the string as input and reads the string until tab('\t') occurs.

read_intermediate_file()- takes in line number, LOCCTR, opcode, operand, label and input output files. If the line is comment returns true and takes in the next input line. Then using the **read_till_tab()** function, it reads the label, opcode, operand and the comment. Based on the different types of opcodes, it will count in the necessary conditions to take in the operand.

create_obj_code_format_34() - When we get our format for the opcode as 3 or 4, we call this function. It checks the various situations in which the opcode can be and then taking into consideration the operand and the number of half bytes calculates the object code for the instruction. It also modifies the modification record when there is a need to do so.

write_D_rec()- It writes in the D record after the H record is written if the control sections are present.

write_R_rec()- It writes in the R record for the control section.

write_end_rec()- It will write the end record for the program.

After the execution of the pass1.cpp, we will print the Tables like SYMTAB, LITTAB, etc., in a separate file and then execute the pass2.cpp.

Data Structures used in the implementation-

1. Map
2. Class

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. Class is a collection of variables of different data types under a single name. It

is similar to a struct in that, both holds a collection of data of different data types.

Map is used to store the **SYMBOL TABLE, OPCODE TABLE, REGISTER TABLE, LITERAL TABLE, BLOCK TABLE, CONTROL SECTIONS.**

Each map of these tables contains a key in the form of string (data type) which represent an element of the table and the mapped value is a class which stores the information of that element.

Classes of each are as follows-

SYMTAB

The class contains information of labels like name, address, block number, a bool representing whether the label exits in the symbol table or not, an integer representing whether label is relative or not.

Info_op

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

LITTAB

The class contains information of literals like its value, address, block number, a bool representing whether the literal exits in the literal table or not.

Info_reg

The class contains information of registers like its numeric equivalent, a bool representing whether the registers exits or not.

BLOCKS

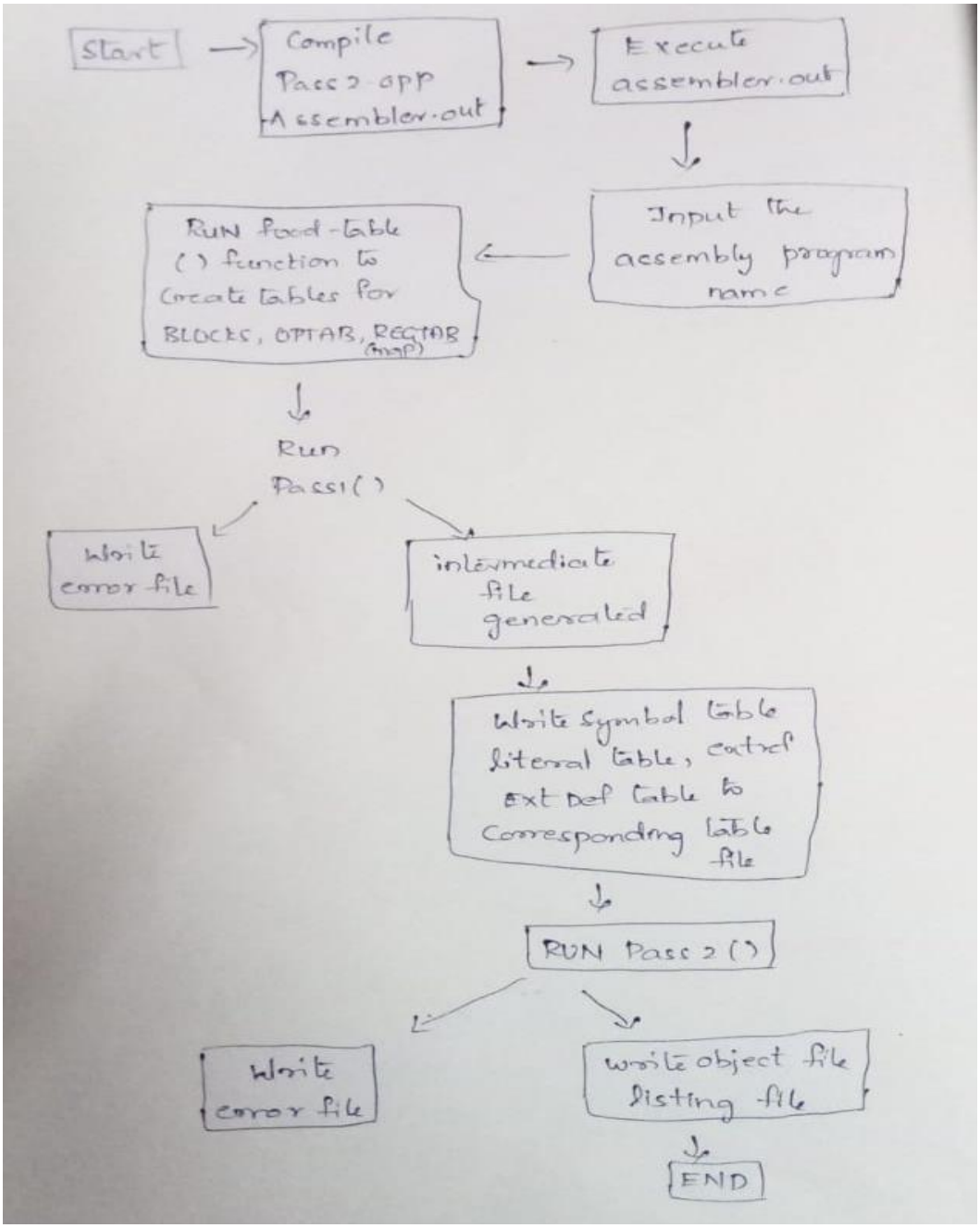
The class contains information of blocks like its name, start address, block number, location counter value for end address of block, a bool representing whether the block exits or not.

csect

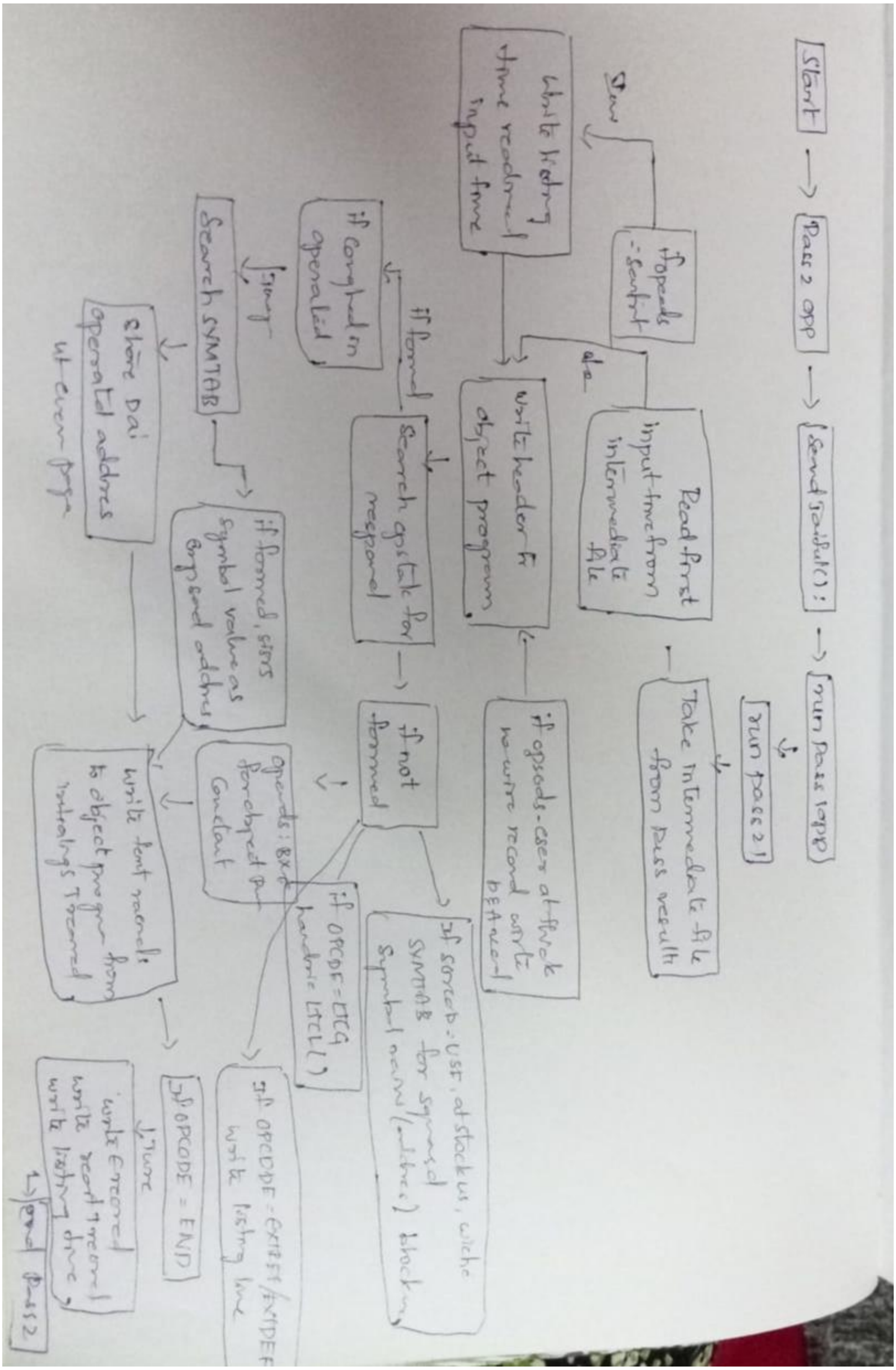
The class contains information of different control section like its name, start_addr,section number,length, location counter value for end address of section.It also contains two maps for extref and extdef of particular section.

Design Results (Flow Chart) :

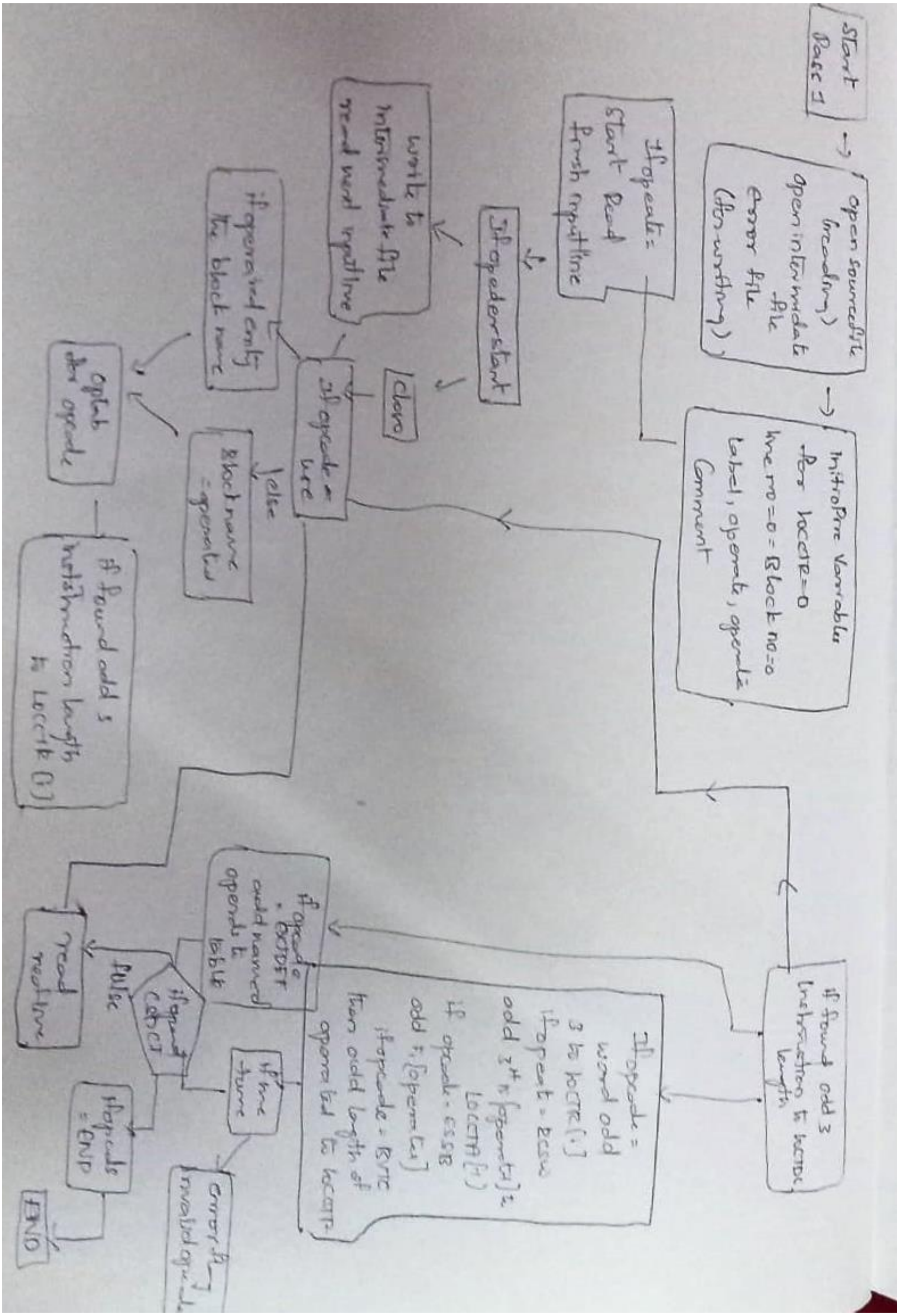
1.control flow:



2.PASS 2 :


Start
run pass 2
Take intermediate file from pass result
Read first input line from intermediate file
write header for object program
if opcode = END
end pass 2

PASS 1:

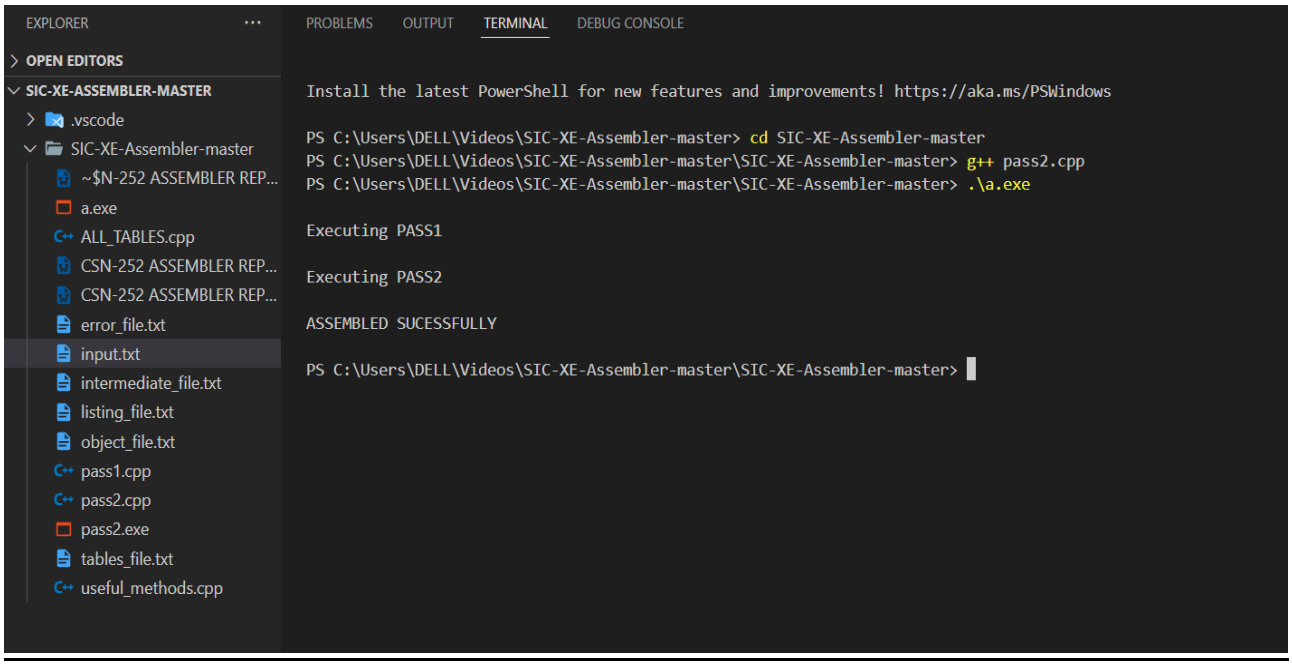


SAMPLECODE : (“input.txt)

SIC-XE-Assembler-master > input.txt

```
1     SUM     START 0
2     FIRST   LDX   #0
3             LDA   #0
4             +LDB  #TABLE2
5             BASE  TABLE2
6     LOOP    ADD   TABLE,X
7             ADD   TABLE2,X
8             TIX   COUNT
9             JLT   LOOP
10            +STA  TOTAL
11            RSUB
12    COUNT   RESW  1
13    TABLE   RESW  2000
14    TABLE2  RESW  2000
15    TOTAL   RESW  1
16            END   FIRST
```

Out put :



Intermediate file : (“intermediate_file.txt”) :

SIC-XE-Assembler-master > intermediate_file.txt

	Line	Address	info_label	OPCODE	OPERAND	Comment
2	5	00000	0	SUM	START	0
3	10	00000	0	FIRST	LDX	#0
4	15	00003	0		LDA	#0
5	20	00006	0		+LDB	#TABLE2
6	25	0000A	0		BASE	TABLE2
7	30	0000A	0	LOOP	ADD	TABLE,X
8	35	0000D	0		ADD	TABLE2,X
9	40	00010	0		TIX	COUNT
10	45	00013	0		JLT	LOOP
11	50	00016	0		+STA	TOTAL
12	55	0001A	0		RSUB	
13	60	0001D	0	COUNT	RESW	1
14	65	00020	0	TABLE	RESW	2000
15	70	01790	0	TABLE2	RESW	2000
16	75	02F00	0	TOTAL	RESW	1
17	80	02F03			END	FIRST

Listing File : (“listing_file.txt)

SIC-XE-Assembler-master > listing_file.txt

```
Line    Address Label   OPCODE  OPERAND ObjectCode  Comment
5   00000   0   SUM START   0
10  00000   0   FIRST   LDX #0  050000
15  00003   0       LDA #0  010000
20  00006   0       +LDB    #TABLE2 69101790
25  0000A   0       BASE    TABLE2
30  0000A   0   LOOP    ADD TABLE,X 1BA013
35  0000D   0       ADD TABLE2,X    1BC000
40  00010   0       TIX COUNT   2F200A
45  00013   0       JLT LOOP    3B2FF4
50  00016   0       +STA    TOTAL   0F102F00
55  0001A   0       RSUB        4F0000
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
```

Error File : (“error_file.txt”)

SIC-XE-Assembler-master > error_file.txt

```
1   **********PASS1**********
2
3
4   **********PASS2**********
5
```

Tables file : (tables_file.txt")

SIC-XE-Assembler-master > tables_file.txt

```
**********************************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*****************************
```

OBJECT CODE : ("object_file.txt")

SIC-XE-Assembler-master > object_file.txt

```
H^SUM    ^000000^002F03
T^000000^1D^0500000100006910179O1BA0131BC0002F200A3B2FF40F102F004F0000
M^000007^05
M^000017^05
E^000000

```

CONCLUSION :

It was an attempt to show how the SIC/XE assembler works and assembles the assembly language code and implements it in the programming language, C++. In this code we tried to show how pass1 and pass2 are interacting with each other using an intermediate file and listing and object program are displayed as output.