CSN 252

SIC/XE Assembler Implementation

Name: Soham Singh

Enrolment No: 21114099

Email:s_singh2@cs.iitr.ac.in

Batch:O-4, CSE

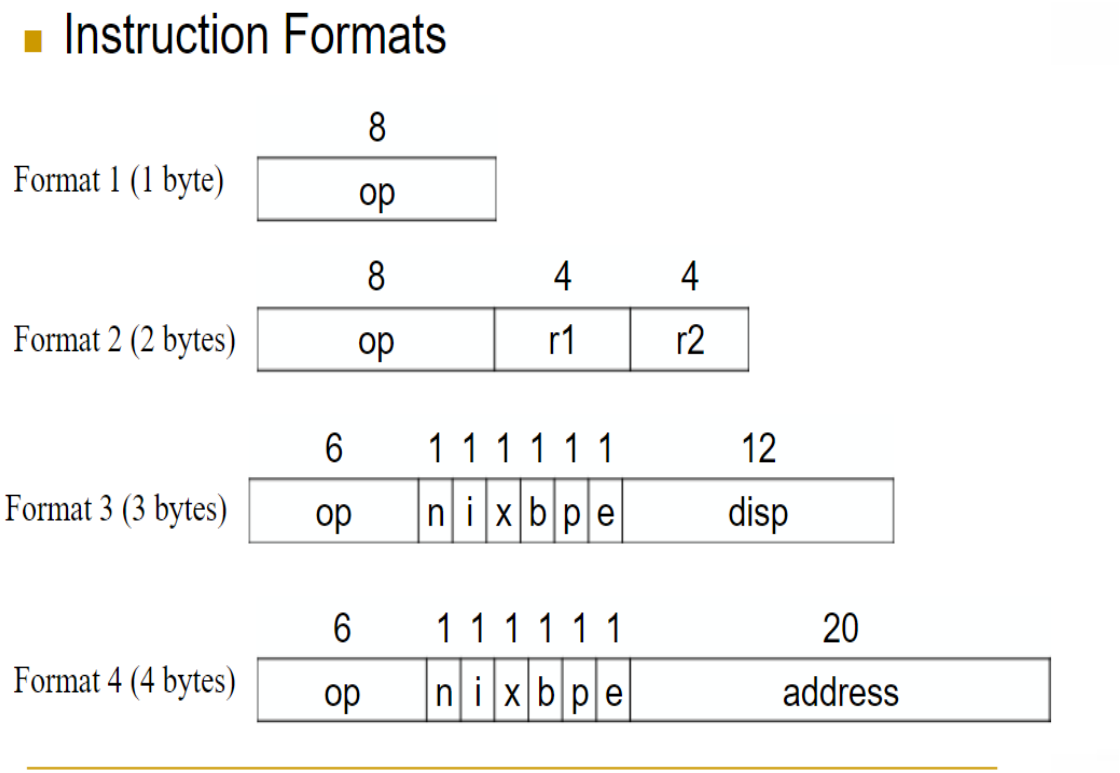
Phone no:8957914217

Introduction :

1. Our task in this project is to implement SIC/XE assembler supporting control section.

2. We have developed an Assembler that boasts comprehensive support for all SIC-XE instructions in all four formats, along with all addressing modes. Our Assembler also includes the crucial feature of program relocation and supports several machine-independent assembler features such as literals, symbol defining statements, expressions, control sections, and program linking.

Instruction formats of SIC/XE:



Execution flow :

INPUT : Assembler source program using the 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

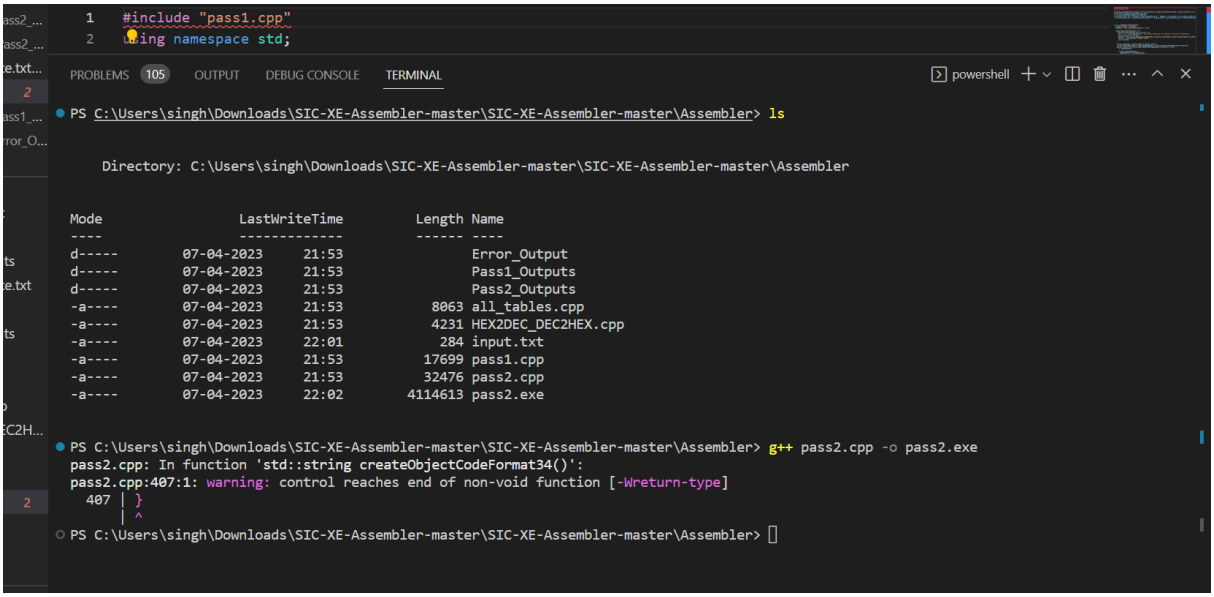
ERROR :

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

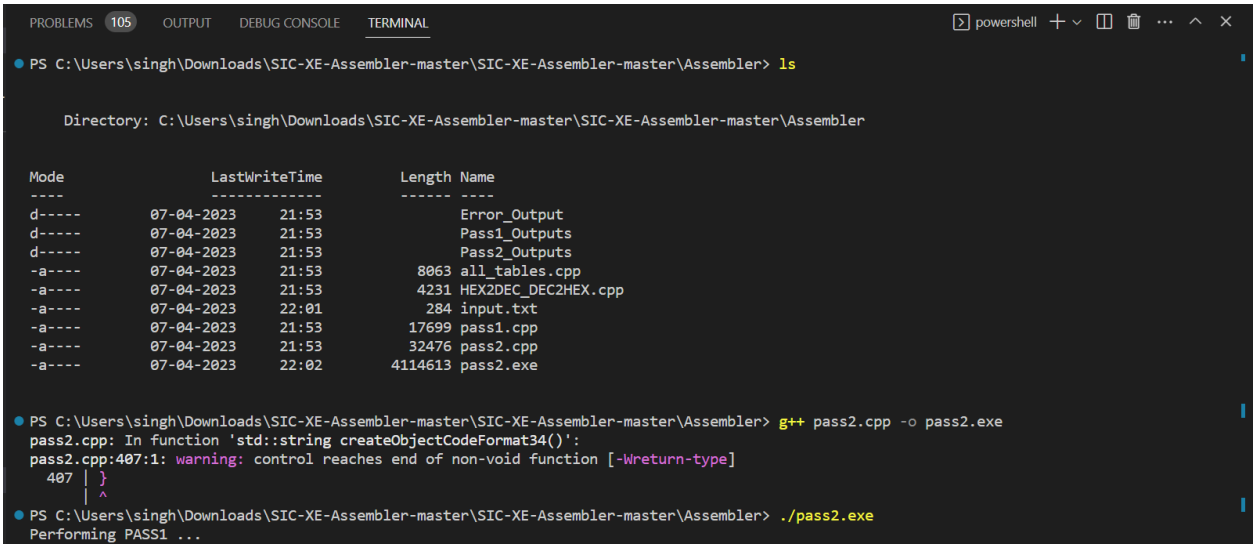
Steps to Compile and Execute Assembler :

(We must make sure that g++ compiler is installed before executing below commands)

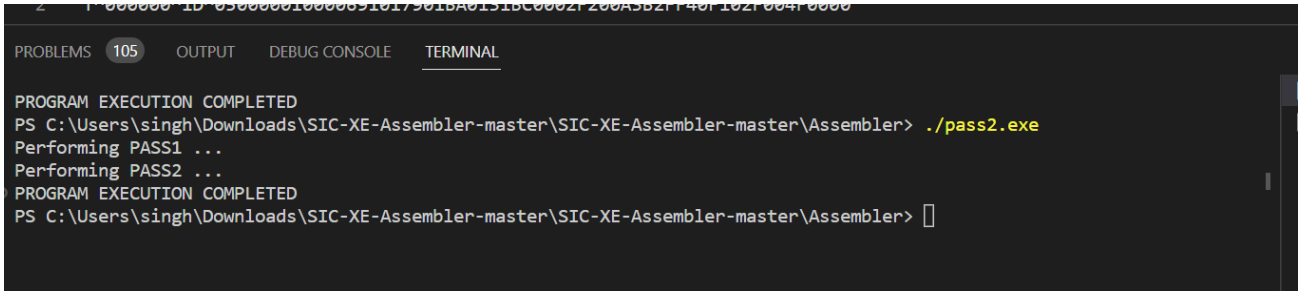
- Initially we have to **compile "pass2.cpp"** file by "**g++ pass2.cpp -o pass2.exe**" command:



- Then we have to execute the **object program "pass2.exe"** generated by **compilation** of "pass2.cpp" file:



- Successful execution of program produces below output in console :



- During execution of program, "assembly source program goes through Pass-1 and Pass-2 stages and generates "**intermediate.txt, tables.txt**" and "**listing.txt,object.txt**" files in **Pass1** and **Pass2** respectively
- An error file "errors.txt" is also generated displaying the errors in the assembly program (if any)

To Execute Any other Assembler code :
Replace the code in "input.txt" file with code which we want to execute and follow above commands for compilation and execution.

IMPLEMENTATION TECH :

- Our assembler has been developed using the C++ programming language. We have also used a variety of C++ libraries, such as ifstream and ofstream, to read data from files and write results to other files.

MODULES AND THEIR IMPLEMENTATIONS :

1. **Module "all_tables.cpp" :**
   1. This module contains all the data structures required to design our assembler.
   2. It contains the classes for labels, opcode, literal, blocks, extdef, extref, and control sections.
   3. It also contains definition of representation of many tables such as symbol_table, opcode_table, etc :- using unordered_map(STL)

2. **Module "HEX2DEC_DEC2HEX.cpp"** :

- This module contains many functions that are required by the other files.

Functions :

1. **int_to_string_hexadecimal**() :
   - takes in input as int and then converts it into its hexadecimal equivalent with string data type.2.
2. **string_expansion**() :
   - expands the input string to the given input size.
3. **hexadecimal_string_to_int**() :
   - converts the hexadecimal string to integer and returns the integer value.
4. **stringToHexString**() :

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

5. **is_white_space**() :
   - checks if blanks are present. If present, returns true or else false.
6. **Is_comment_line**() :
   - check the comment by looking at the first character of the input string, and then accordingly returns true if comment or else false.
7. **Ifall_num**() :
   - checks if all the elements of the string of the input string are number digits.
8. **rd_first_nonwhitespace**() :
   - 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.
9. **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.

10. **get_original_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.
11. **get_flag_format**():
    - returns the flag bit if present in the input string or else it returns null string.
12. **Class EvaStr** :
    - **Class methods** :
    - **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.

3. Module "pass1.cpp" :

1. This module is responsible for generating "tables.txt" and "intermediate.txt" files which are required for pass2
2. This module also writes errors encountered during execution of pass1 stage to "errors.txt" file
3. In this module, we declare the variables which are 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 OPTAB, if present we find out its format and then accordingly increment the LOCCTR. If not found in OPTAB, 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 expression_evaluation() 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, startaddress 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().

4. **Functions** :
    - **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 lit_tab and then update the line number. If for some literal, we did not find the address, we store the present address in the lit_tab and then increment the LOCCTR on the basis of literal present.
    - **expression_evaluation** () : 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 paircount which keeps the account of whether the expression is absolute or relative and if the paircount gives some unexpected value, we print an error message.

4. **Module "pass2.cpp"** :
    1. We take in the intermediate file as input using the read_Intermediate_file() function and generate the listing file and the object program. 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 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. 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 writeTextRecord() 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 createObjectCodeFormat34() function in the pass2.cpp. For writing the end record, we use the writeEndRecord() function. If control sections are present, we will use the writeRRecord() and writeDRecord() 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.

2. **Functions** :
   - **read_until_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_until_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.
   - **createObjectCodeFormat34**() : 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.
   - **writeDRecord**() : It writes in the D record after the H record is written if the control sections are present.
   - **writeRRecord**() : It writes in the R record for the control section.
   - **writeEndRecord**() :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 :

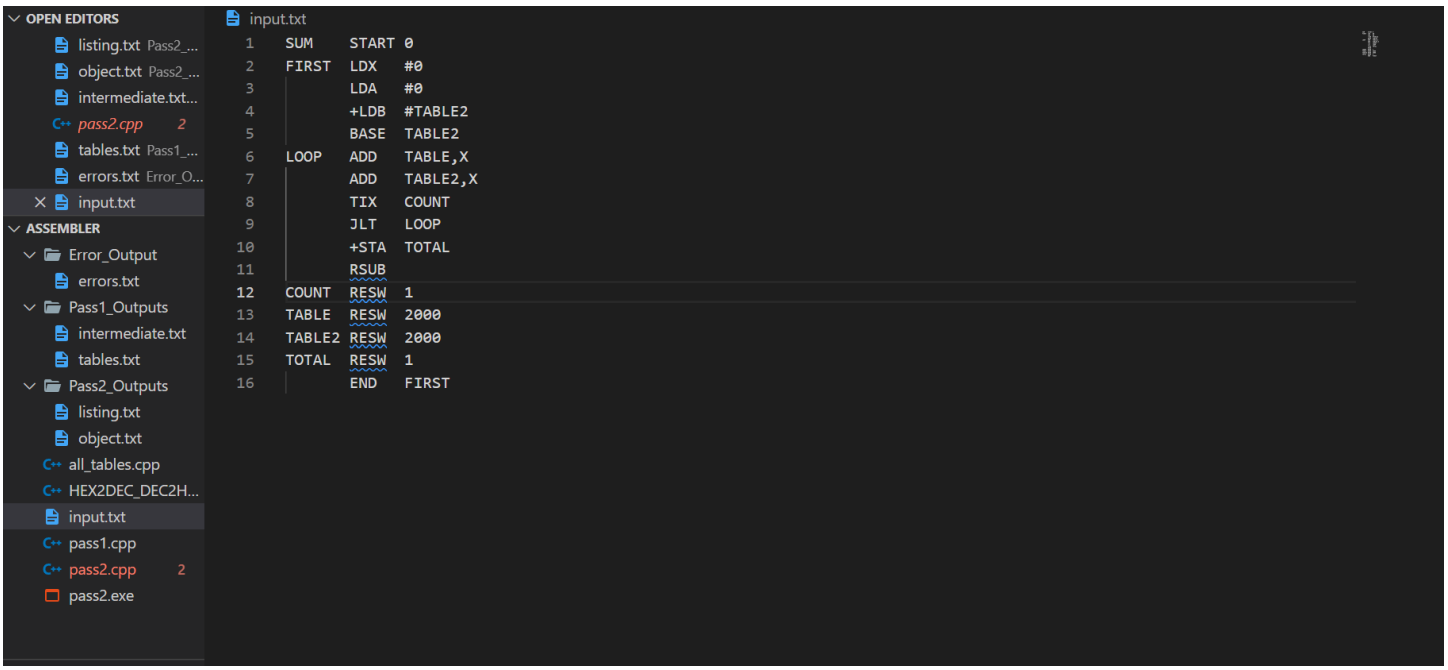
1. **unordered_map** - unordered_map is used to store the SYMBOL TABLE, OPCODE TABLE, REGISTER TABLE, LITERAL TABLE, BLOCK TABLE, CONTROL SECTIONS. Each map of these tables contains a key in the form of string(data type) which represent an element of the table and the mapped value is a class object which stores the information of that element.
2. **Class**

Structures of each are as follows :

- **SYMB_TAB-**The class contains information of labels like name, address, block number, a character representing whether the label exits in the symbol table or not, an integer representing whether label is relative or not.
- **OP_TAB-**The class contains information of opcode like name, format, a character representing whether the opcode is valid or not.
- **LIT_TAB-**The class contains information of literals like its value, address, block number, a character representing whether the literal exits in the literal table or not.
- **REG_TAB-**The class contains information of registers like its numeric equivalent, a character 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 character representing whether the block exits or not.
- **CSECT_TAB-**The class contains information of different control section like its name, start address, section number ,length, location counter value for end address of section. It also contains two maps for extref and extdef of particular section.

SAMPLE PROGRAM :

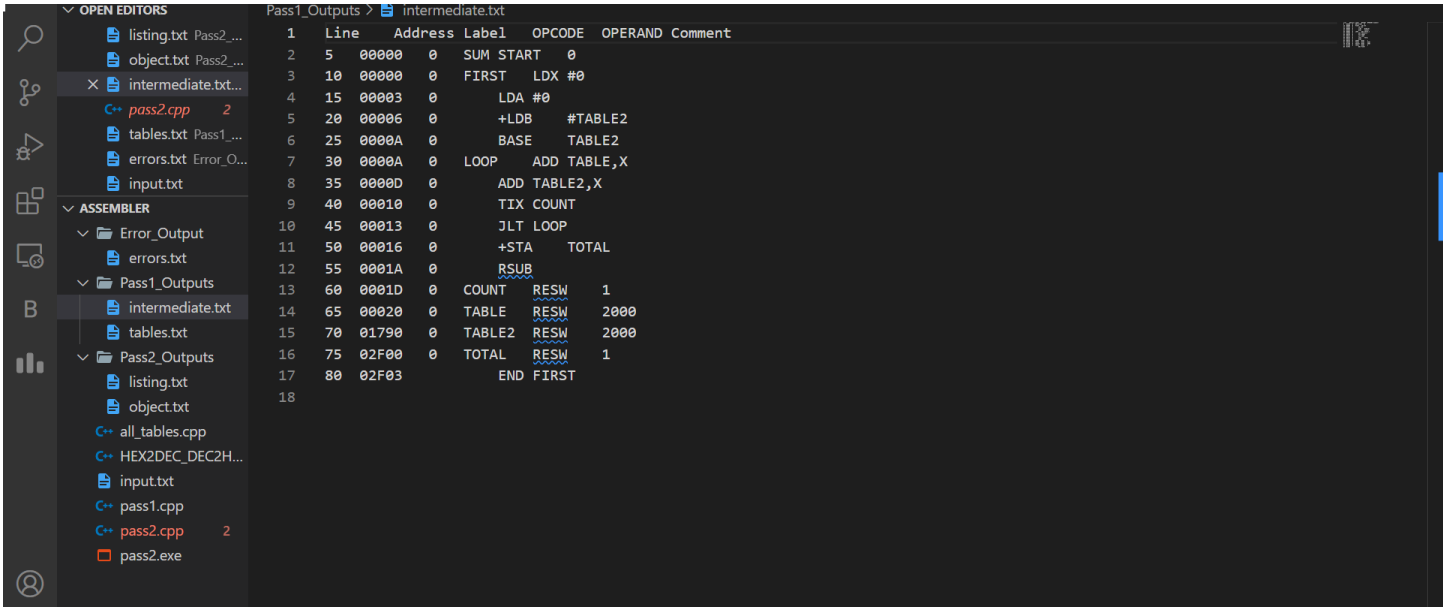
INPUT :



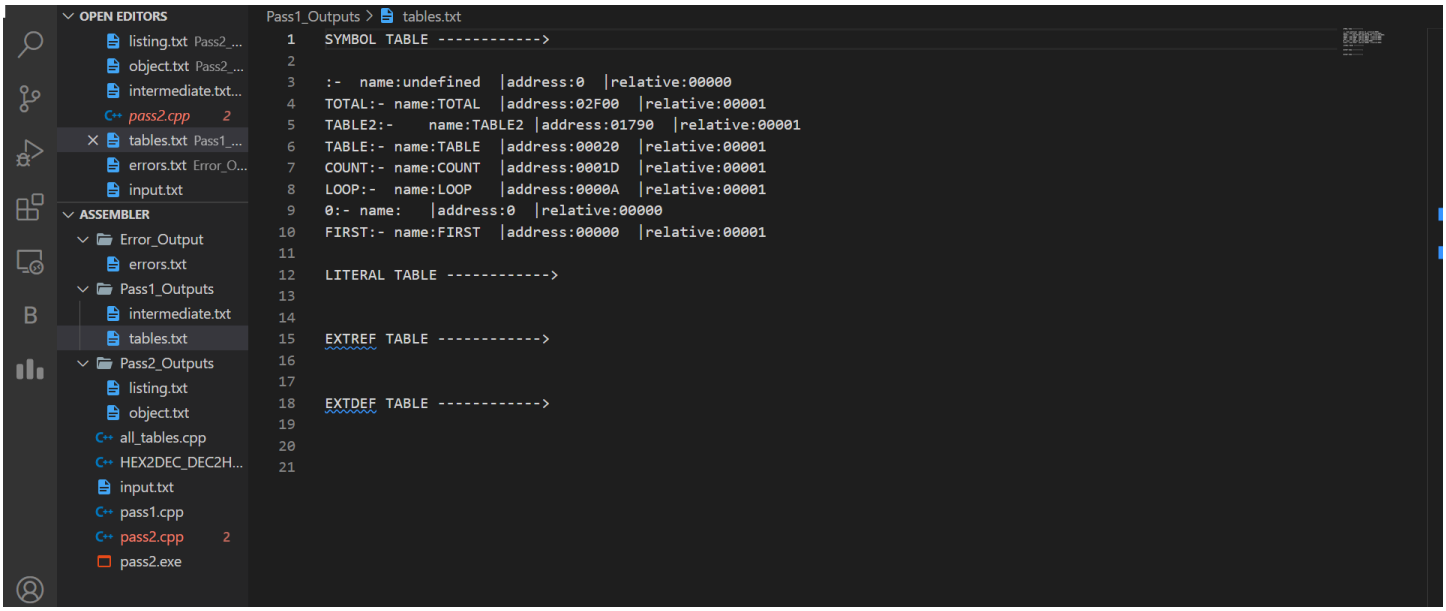
OUTPUT :

PASS-1 Outputs :

1. **“intermediate.txt”:**

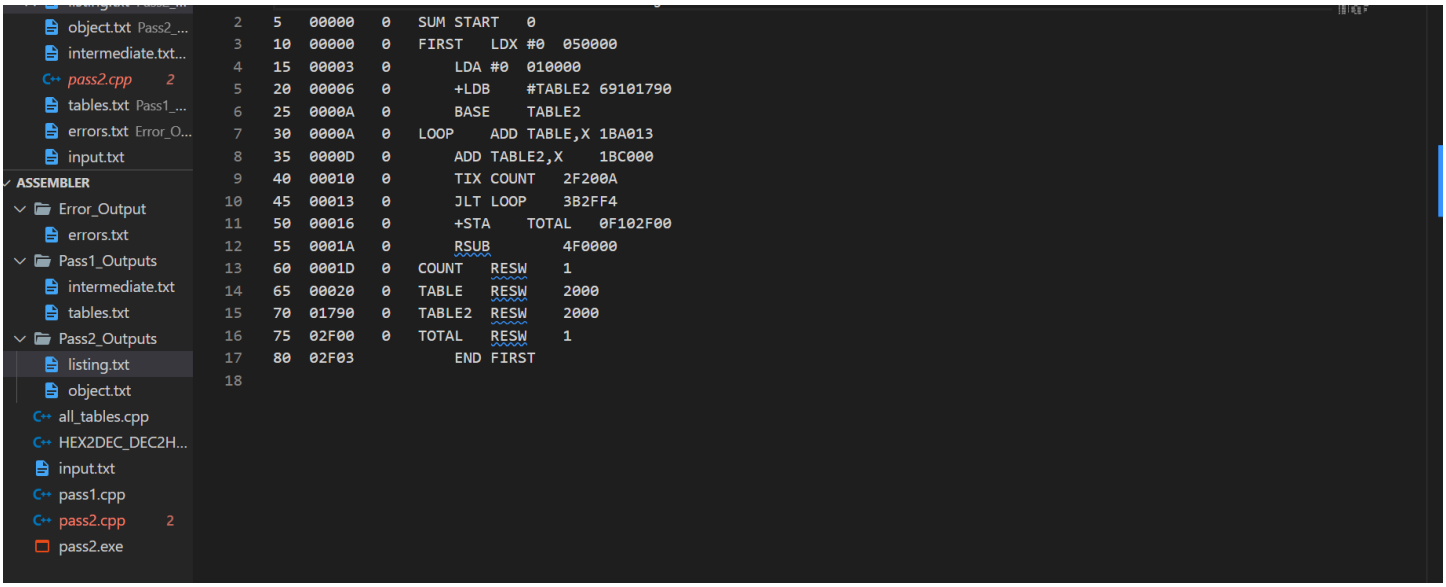


2. **“tables.txt”** :

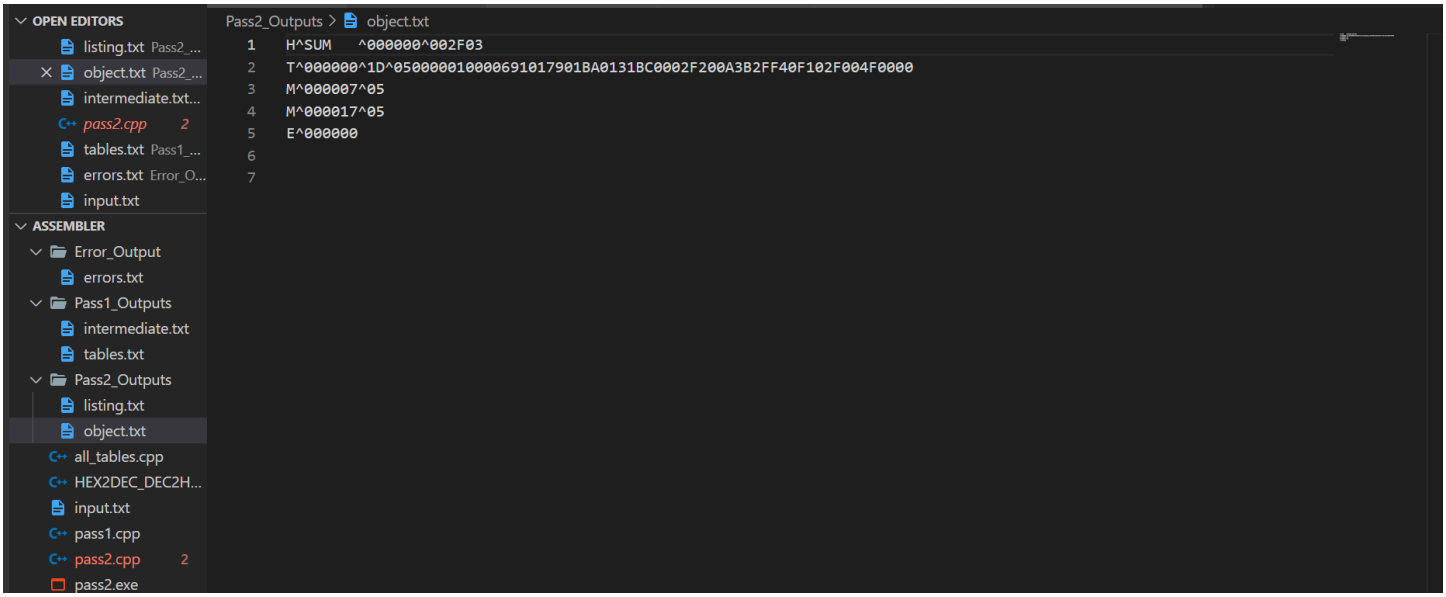


PASS-2 Outputs :

1. **“listing.txt”** :

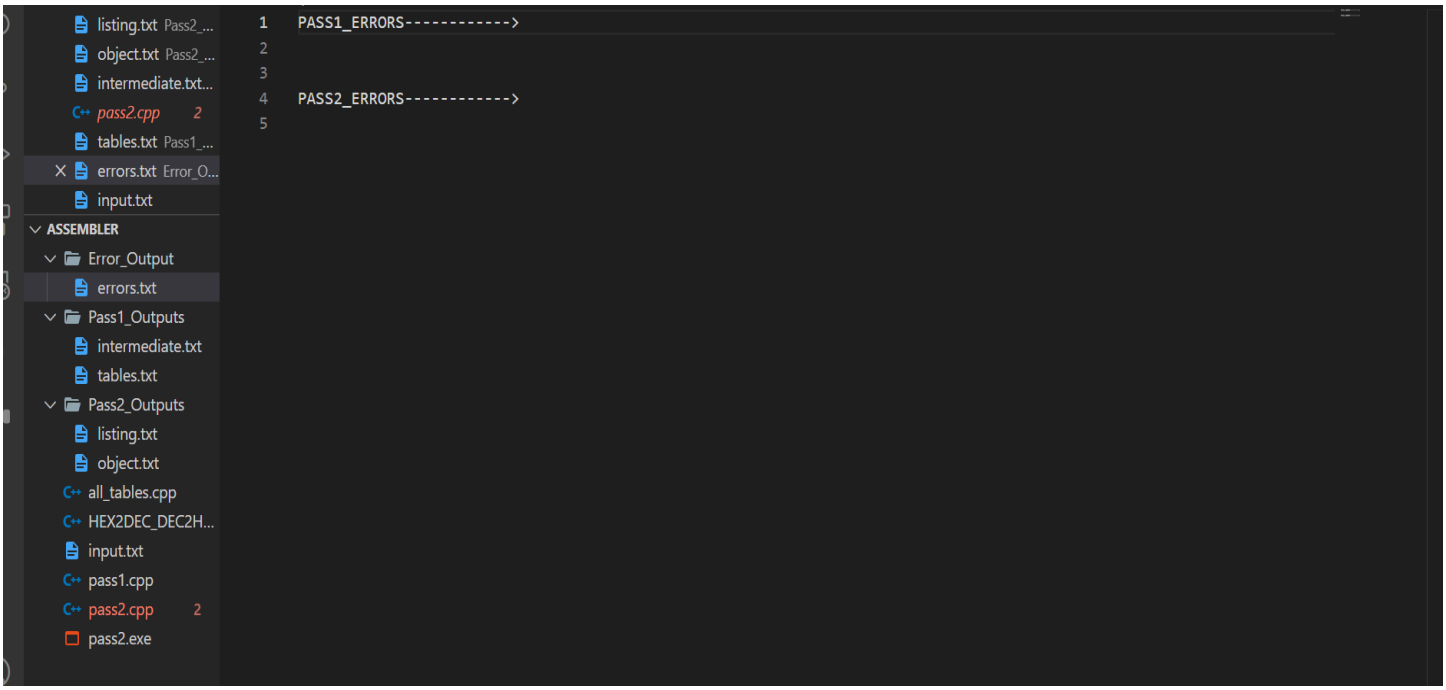


2. **“object.txt”** :



ERRORS :

1. **“errors.txt”** :


listing.txt Pass2_...
object.txt Pass2_...
intermediate.txt...
pass2.cpp 2
tables.txt Pass1_...
errors.txt Error_O...
input.txt
ASSEMBLER
Error_Output
errors.txt
Pass1_Outputs
intermediate.txt
tables.txt
Pass2_Outputs
listing.txt
object.txt
all_tables.cpp
HEX2DEC_DEC2H...
input.txt
pass1.cpp
pass2.cpp 2
pass2.exe
1 PASS1_ERRORS------------>
2
3
4 PASS2_ERRORS------------>
5