SICXE-Assembler

Tharuniswer R

20114100

The SIC-XE assembler designed in C++ is capable of converting SIC-XE assembly code into object code, as well as produce an intermediate file with the addresses of all instructions and variables specified.

Like other SIC-XE assemblers this assembler is capable of implementing and creating the following:

The machine dependent features the assembler implements are:

- It supports pc-relative, indexed, immediate, base relative, extended, indirect addressing modes.
- It also supports various instruction formats
    1. 1-byte (opcode only)
    2. 2-byte (register formats)
    3. 3-byte instructions
    4. 4-byte instructions
- It creates modification records too thereby facilitating relocatable programs.

The machine independent features the assembler supports are:

- The use of literals through the creation of literal table and LTORG
- The use of symbol defining statement EQU.
- It also supports ORG statements.
- It also supports expressions.
- Program blocks are also supported.
- This assembler doesn't support control sections

Steps to run the assembler:

I designed and ran the code in VScode. So steps to run it in VSscode are:

- First open the new_sicxeassembler folder in VScode. Bring the test file also into the same folder.
- Next type "g++ Pass2.cpp" onto the VScode terminal.
- After compliation has completed run the executable file which is newly created. In my case it was a.exe. So I ran "./a.exe".
- Now enter the name of the test file which needs to be assembled into object code.
- Then the listing file, object file, intermediate file, tables file will now be created along with the error file.

The design of the assembler:

There are four files Pass1, Pass2, Tables, Functions which are all code files in C++. The descriptions of what each file does is given below:

PASS 1-

In pass-1 the error file and the intermediate file is created. Source file or intermediate file not opening is treated as errors and are printed in error file while error file not opening is printed in console itself.

The variables required are declared.

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 it is then 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'. Inside the inner loop, the line is again checked if it's a comment. If it's a comment, it is printed onto the intermediate file, the line number is updated and next input line is taken in. If it's not a comment, checking is done to find if there is a label in the line, if present in the SYMTAB, error is printed saying 'Duplicate symbol' in the error file or else it is stored onto 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 in the SYMTAB for use in pass2.

For opcodes like USE, we insert a new BLOCK entry in the BLOCK map as defined in the sicxe_utility.cpp file, for LTORG we call the LTORG_put() 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_E() function, if it is valid then we enter the symbols in the SYMTAB. And if the opcode doesn't match with the above given opcodes, an error message is printed in the error file.

Accordingly, the data is updated which is to be written in the intermediate file. After the loop ends, we store the program length and then go on for printing the SYMTAB and LITTAB. After that we move on to the pass2().

LTORG_put() - It uses pass by reference. All the literals till that time is printed by taking the arguments from the pass1() function. An iterator is run 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_E() - It also uses pass by reference. A while loop is used to get the symbols from the expression. If the symbol is not found in the SYMTAB, an error message is kept in the error file. A variable numPairs is used which keeps an account of whether the expression is absolute or relative and if the numPairs gives some unexpected value, an error message is printed.

TABLES –

All the data structures required for the assembler to run is kept in this file. It contains the structs for labels, opcode, literal, blocks.

FUNCTIONS -

The useful functions required to run other assembler files is contained in the FUNCTIONS file

- intToHexString ()- takes in input as int and then converts it into its hexadecimal equivalent with string data type.
- expandString()- 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.
- stringHexToInt()- converts the hexadecimal string to integer and returns the integer value.
- stringToHexString()- takes in string as input and then converts the string into its hexadecimal equivalent and then returns the equivalent as string.
- checkWhiteSpace()- checks if blanks are present. If present, returns true or else false.
- checkCommentLine()- 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.
- readFirstNonWhiteSpace()- 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.
- writeToFile()- 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.
- getRealOpcode()- 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.
- getFlagFormat()- returns the flag bit if present in the input string or else it returns null string.
- Class EvaluateString – contains the functions :
    1. -peek()- returns the value at the present index.
    2. -get()- returns the value at the given index and then increments the index by one.
    3. -number()- returns the value of the input string in integer format.

PASS 2-

- The intermediate file is taken as input using the rIntFile() function which generates the listing file and the object program.

Similar to pass1, if the intermediate file is unable to be opened, an error message is printed in the error file. Same with the object file if unable to open. The first line of the intermediate file is then read. Until the lines are comments, we take them as input and print them to our listing file and update our line number.

If opcode is 'START', the start address is initialized as the LOCCTR, and write the line into the listing file.

. We then write the first header record in the object program. Then until the opcode becomes 'END' 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. The object codes are written on the basis of the types of formats used in the instruction. Based on different types of opcodes such as 'BYTE','WORD','BASE','NOBASE', different types of object codes are generated. For the format 3 and format 4 instruction format, the createObjectCodeFormat34() function is used in the pass2.cpp file .

For writing the end record, the writeEndRecord() function is used. For the instructions with immediate addressing, we will write the modification record.

rdTillTab()- Input is read until tab occurs

rIntFile()- 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 rdTillTab() 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.

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

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

Data Structures in C++used-

1. Map data structure
2. Struct data structure

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. Structure(struct) is a collection of variables of different data types under a single name. It is similar to a class. The difference is that one's data is by default public while other's is by default private.

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

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 struct which stores the information of that element.

Structures of each are as follows-

SYMTAB

The struct 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.

OPTAB

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

LITTAB

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

REGTAB

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

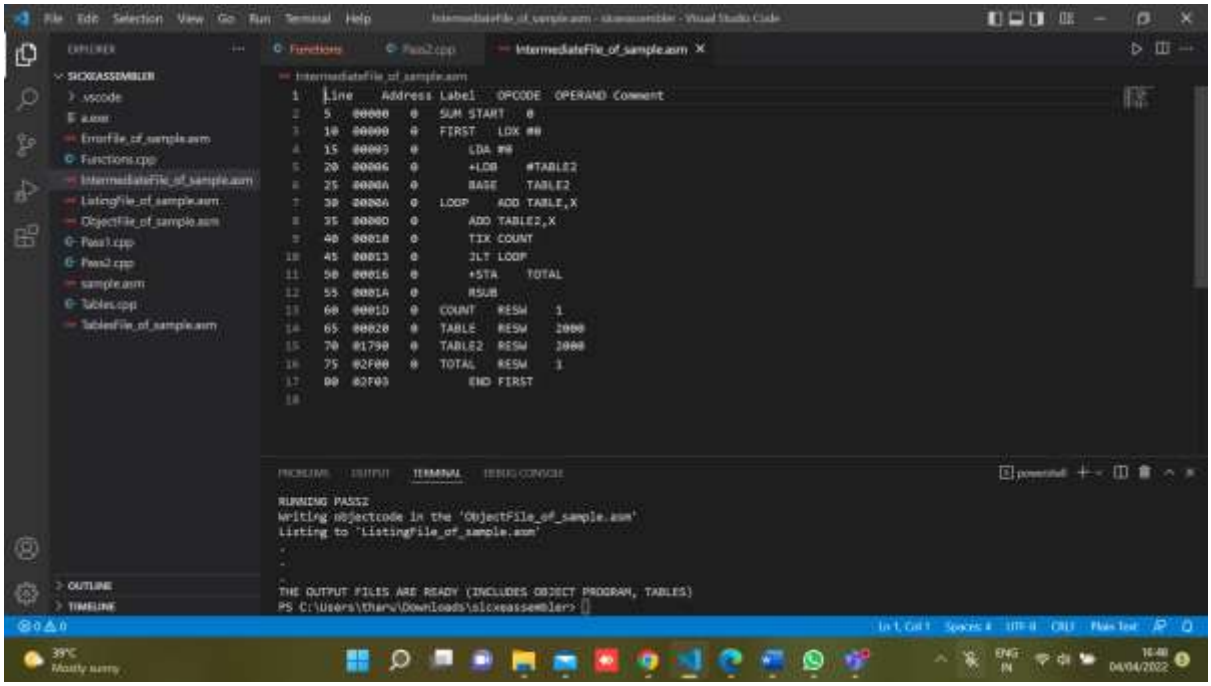
BLOCKTAB

The struct 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.

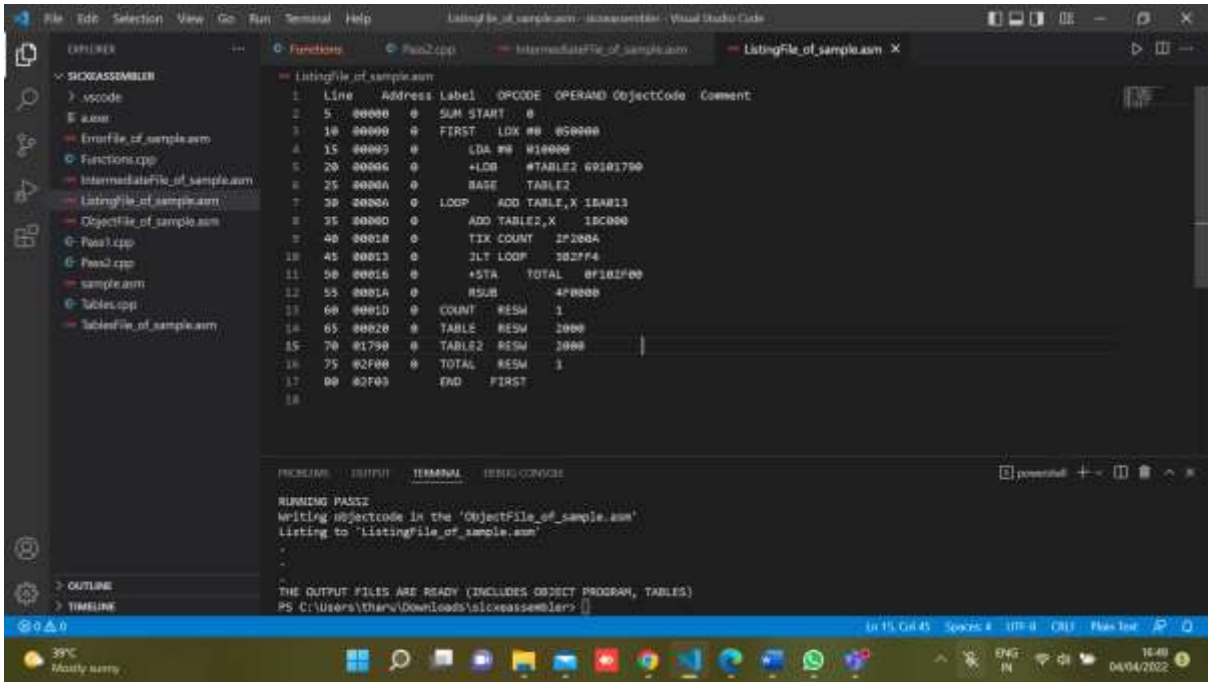
Sample program:

```
SUM     START  0
FIRST   LDX    #0
        LDA    #0
        +LDB   #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
```

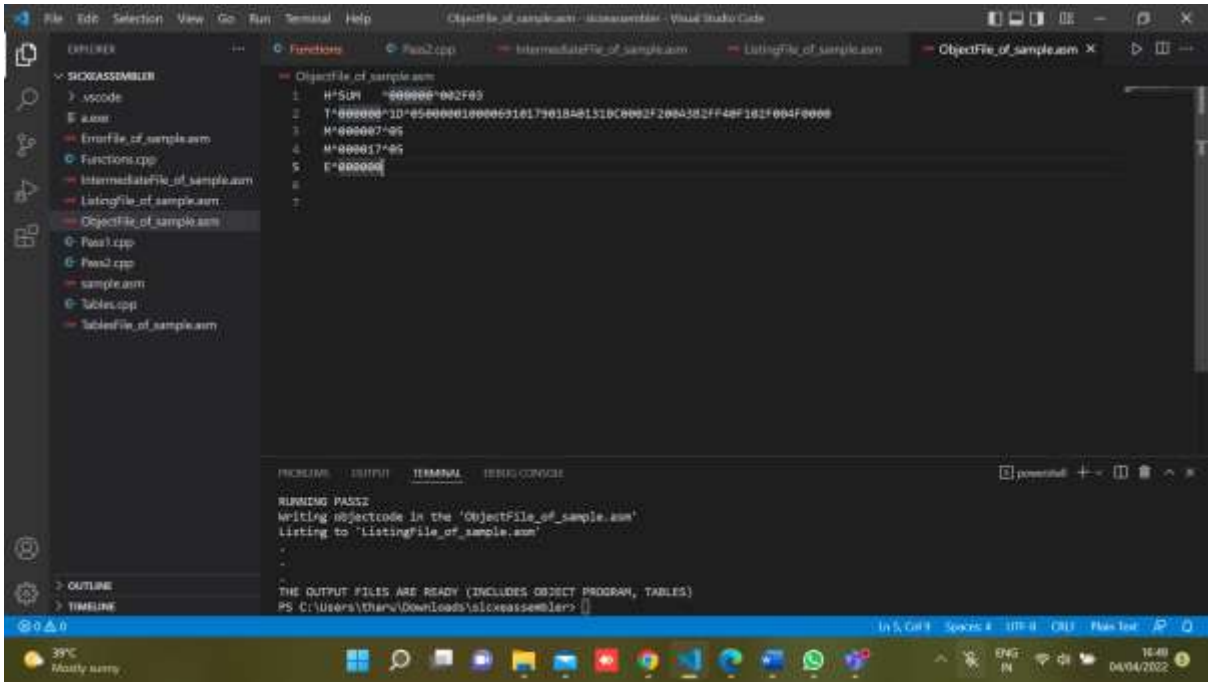
The intermediate file produced is,



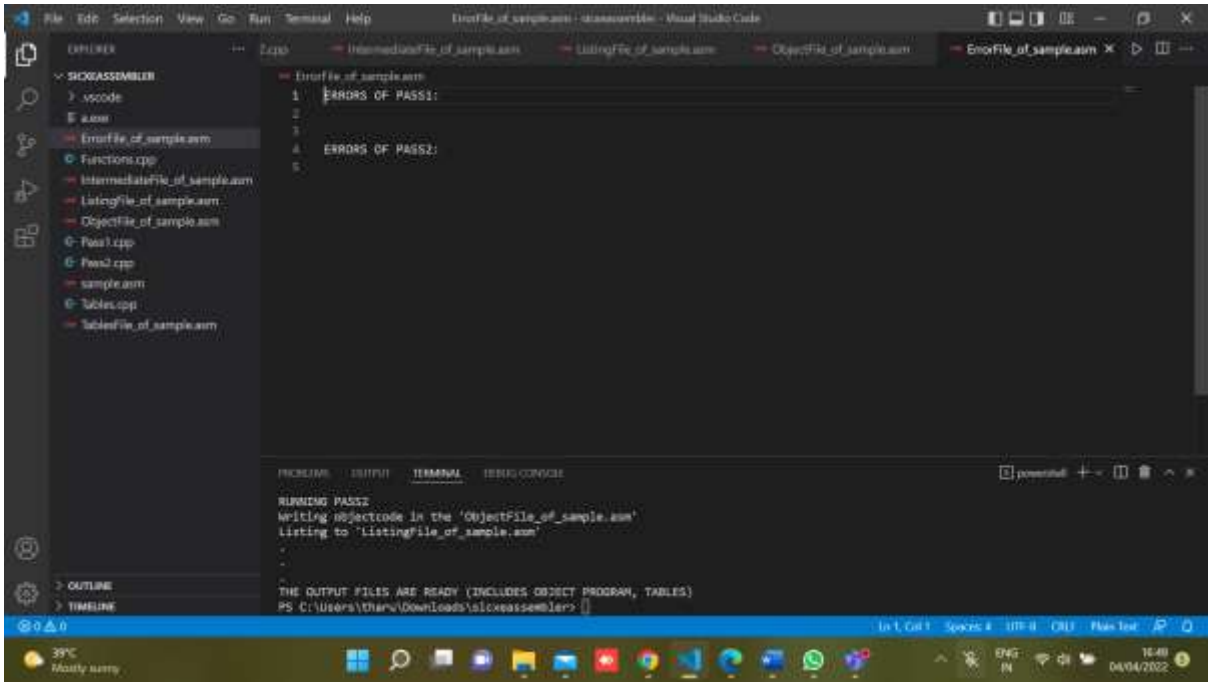
The Listing file produced is,



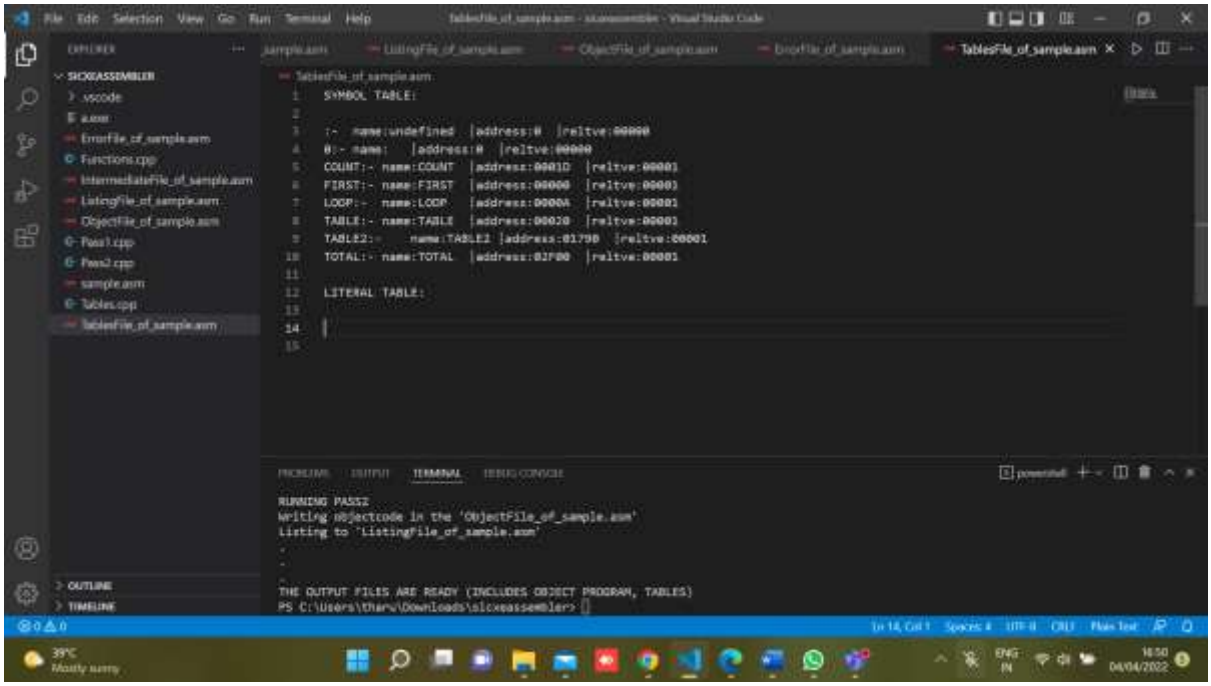
The object code is,



The error file is,



The tables file is,


SICXEASSEMBLER
vscode
a.exe
Errorfile_of_sample.asm
Functions.cpp
IntermediateFile_of_sample.asm
ListingFile_of_sample.asm
ObjectFile_of_sample.asm
Pass1.cpp
Pass2.cpp
sample.asm
Tables.cpp
TablesFile_of_sample.asm
TablesFile_of_sample.asm
SYMBOL TABLE:
:- name:undefined |address:0 |reltve:00000
0:- name: |address:0 |reltve:00000
COUNT:- name:COUNT |address:0001D |reltve:00001
FIRST:- name:FIRST |address:00000 |reltve:00001
LOOP:- name:LOOP |address:00004 |reltve:00001
TABLE:- name:TABLE |address:00020 |reltve:00001
TABLE2:- name:TABLE2 |address:01790 |reltve:00001
TOTAL:- name:TOTAL |address:02F00 |reltve:00001
LITERAL TABLE:
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
powershell
RUNNING PASS2
Writing objectcode in the 'ObjectFile_of_sample.asm'
Listing to 'ListingFile_of_sample.asm'
.
.
.
THE OUTPUT FILES ARE READY (INCLUDES OBJECT PROGRAM, TABLES)
PS C:\Users\Dhar\Downloads\slcxeassembler>
OUTLINE
TIMELINE
39°C
Mostly sunny
ENG
IN
16:50
04/04/2022