System Software

SIC/XE assembler

NAME: ARYA ANAND

ENROLLMENT NO: 20114017

SUB BATCH: O1-CSE

CONTACT NO: 9328034486

EMAIL ID: arya\_a@cs.iitr.ac.in

C++ Implementation for Two pass Assembler

Assembler includes all the SIC/XE instructions and supports all four formats 1, 2, 3, 4, addressing modes and program relocation.It also includes all Machine-Independent Assembler Features-

* Literals
* Symbol Defining Statements
* Expressions
* Program Blocks
* Control Sections

Input to assembler- Assembler source program using the instruction set of SIC/XE.

Output- Assembler will generate the following files as output-

Pass 1 will generate a **Symbol** Table.

Pass 1 will also generate **Intermediate File** for the Pass 2.

Pass 2 will generate **a listing file** containing the input assembly code and address, block number, object code of each instruction.

Pass 2 will also generate an **object program**.

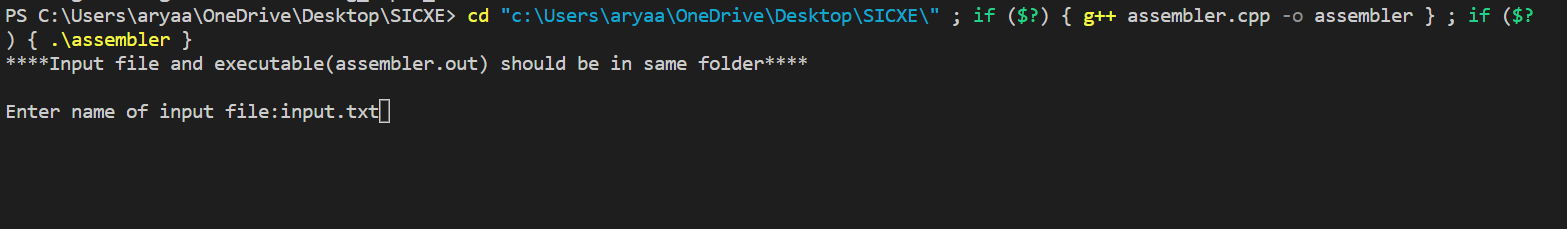
An **error file** is also generated displaying the errors in the assembly program (if any).

Execution Steps-

Run assembler.cpp ONLY.

All the files including PASS1, PASS2, opcode table and other functions of assembler are present in this one file.

While running in CMD



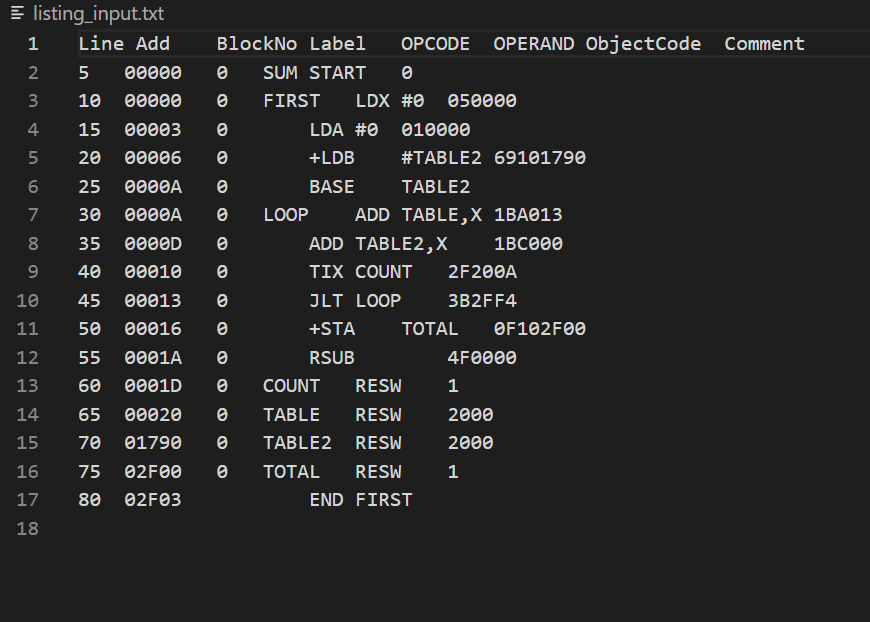
It asks for the filename of input file.

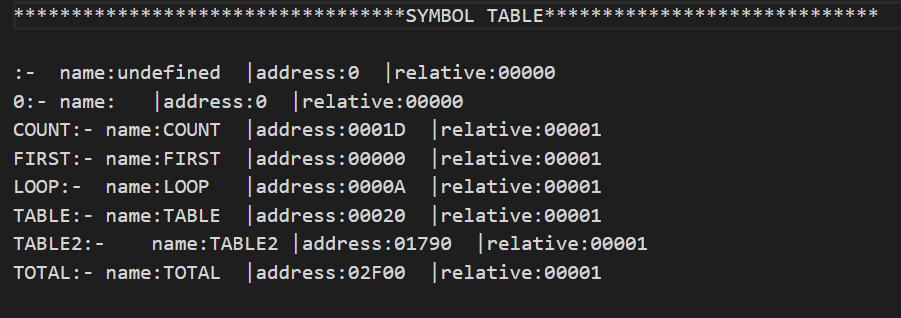
The input should be given as- "input.txt" with proper txt extension.

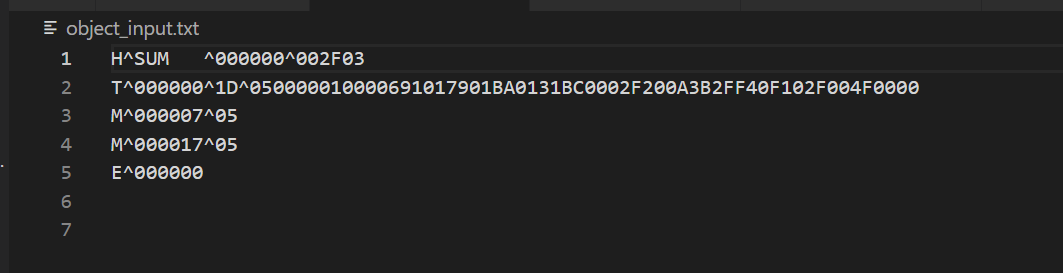
We get -

* Object file
* Listing file
* And Symbol Table

**For the Given sample code asked in Tut8 for SICXE i.e Q3 of section2.2 , the output is as follow-**







Implementation 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 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 utility.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 evaluateExpression() 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().

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.

evaluateExpression()- 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.

**TABLES** –It contains all the data structures required for our assembler to run. It contains the structs for labels, opcode, literal, blocks, extdef, extref, and control sections. 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.

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

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

intToStringHex()- 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 :

-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 readIntermediateFile() 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 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 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.

readTillTab()- takes in the string as input and reads the string until tab(‘\t’) occurs.

readIntermediateFile()- 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 readTillTab() 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 SYTAB, LITTAB, etc., in a separate file and then execute the pass2.cpp.