System
Software

Design of SIC-XE Assembler

Name: Ojjas Madare

Design of SIC-XE Assembler

INTRODUCTION

The developed SIC/XE assembler is a meticulously crafted two-pass assembler implemented using the C++ programming language. Its primary function is to process input files written in assembly language, conforming to the instruction set of SIC/XE architecture, and subsequently generate the corresponding object code file. Additionally, it produces auxiliary files such as symbol tables, intermediate files, immediate files, and error files, ensuring comprehensive documentation and error handling throughout the assembly process.

Operating in two distinct passes, the assembler's workflow is as follows:

1. Pass 1: During this initial pass, the assembler meticulously constructs a symbol table and an intermediate file, which serve as vital references for the subsequent pass.
2. Pass 2: Building upon the data gathered in Pass 1, this phase of the assembly process entails the generation of a comprehensive listing file. This listing file contains the original assembly code alongside pertinent details such as addresses, block numbers, and the object code for each instruction. Furthermore, it facilitates the creation of the object program and the identification and documentation of any errors present within the input assembly program.

Remarkably, the assembler encompasses support for the entirety of SIC/XE instructions, as well as incorporates machine-independent assembler features including but not limited to literals, program blocks, expressions, and symbol-defining statements. This holistic approach ensures robustness and versatility in handling diverse assembly requirements, thereby enhancing the efficiency and reliability of the assembly process.

STEPS TO COMPILE

1) Open terminal in SIC-XE Assembler folder
2) Compile the pass2.cpp file
3) Execute pass2.exe by typing ./pass2 in terminal
4) Then type the input file name. The desired output will be written in respective files.

Pass 1:

During the initial phase, the assembler undergoes a meticulous process of parsing the input source file to generate intermediate and error files. Any failure in accessing the intermediate file or locating the source file prompts the recording of corresponding errors in the designated error file.

Within Pass 1, the assembler sequentially reads each line of the input, scrutinizing whether it constitutes a comment line. Comment lines are promptly recorded in the intermediate file, while line numbering is updated. This iterative process continues until a non-comment line is encountered. Subsequently, the assembler commences the validation of opcodes. If the initial opcode is 'START', the assembler proceeds to update line numbers, LOCCTR, and Start Address accordingly. Otherwise, the start address and LOCCTR are initialized to 0.

A singular while loop governs the progression of Pass 1, terminating upon encountering the 'END' opcode. Within this loop, the assembler differentiates between comment and non-comment lines. Comment lines are transcribed into the intermediate file, with subsequent line number updates. Conversely, upon encountering a non-comment line, the assembler verifies the presence of a label. If a label is identified, it is cross-referenced with the SYMTAB. If a duplicate symbol is detected within the SYMTAB, an error message 'Duplicate symbol' is dispatched to the error file. Otherwise, the symbol is endowed with a name, address, and other requisite attributes before being stored in the SYMTAB.

Following symbol processing, the assembler checks for opcode presence within the OPTAB. If an opcode is identified, its format is ascertained, consequently incrementing the LOCCTR. In cases where an opcode is not found in the OPTAB, alternative opcode tables such as 'RESW', 'BYTE', 'RESBYTE', 'WORD', 'LTORG', 'ORG', 'BASE', 'USE', and 'EQU' are consulted. Accordingly, the assembler updates symbol mappings within the defined tables. For instance, encountering 'USE' beyond the default scenario triggers the insertion of a new BLOCK entry in the BLOCK map. Similarly, for 'ORG', the assembler repositions the LOCCTR to the specified operand value, and for 'LTORG', the handle_LTORG() function is invoked.

In instances where opcode matching fails even after exhaustive checks, the assembler emits an error message to the error file. Upon conclusion of the while loop, the program length is determined and subsequently printed alongside the LITTAB and SYMTAB. The assembler then transitions to Pass 2, utilizing the intermediate file as input.

evaluateExpression() function:
The evaluateExpression() function, leveraging pass by reference, iteratively retrieves symbols from expressions. If any retrieved symbols are absent from the SYMTAB, an error message is dispatched to the error file. Additionally, a variable named pairCount monitors the nature of the expression, distinguishing between absolute and relative expressions. Unexpected pairCount values prompt the emission of error messages.

Pass 2:

Pass 2 commences with the consumption of the intermediate file generated by Pass 1, facilitated by pass2.cpp. Utilizing the readIntermediateFile() function, the assembler generates a listing file and object program. Analogous to Pass 1, failure to open the intermediate or object file results in error messages being printed in the error file.

Successful file access initiates the parsing of the first line within the intermediate file. Comment lines are transcribed into the intermediate file, while opcodes are scrutinized. Upon encountering 'START', the assembler initializes the start address in LOCCTR and records the line in the listing file. Subsequently, the header record is written. The assembler proceeds to read from the intermediate file until encountering the 'END' opcode.

The textrecord() function facilitates the writing of the object program and updates the listing file. Object code generation adheres to the instruction format utilized, with format 3 and 4 instructions processed via the createObjectCodeFormat34() function. Upon completion of all text records, the end record is written.

readIntermediateFile() function:
This function, taking line number, LOCCTR, opcode, operand, label, and input/output files as parameters, navigates the various scenarios associated with opcodes. Considering operands and half bytes, it calculates the object code for instructions and generates modification records if necessary.

WriteEndRecord() function:
Responsible for writing the end record for the program, this function concludes Pass 2. Following the execution of pass1.cpp, the tables (SYTAB, LITTAB, etc.) are printed in a separate file before execution of pass2.cpp.

Functions:

The functions.cpp file encompasses various utility functions utilized in Pass 1 and Pass 2 of the assembler. Noteworthy functions include:

- readFirstNonWhiteSpace(): Facilitates the retrieval of the first non-whitespace character in the current line, updating the index of the string where the non-whitespace character is encountered.
- getString(): Converts characters into strings.
- checkWhiteSpace(): Determines the presence of whitespace, returning a Boolean value.
- expandString(): Expands input strings to the specified length by inserting the required character.
- checkCommentLine(): Determines if the current line is a comment line, returning a Boolean value.
- writeToFile(): Writes a string to a file.
- getFlagFormat(): Retrieves flag bits from the input string.
- intToStringHex(): Converts integers to their hexadecimal equivalents, returning a string.
- stringHextoInt(): Converts hexadecimal strings to integers, returning an integer.

Tables:

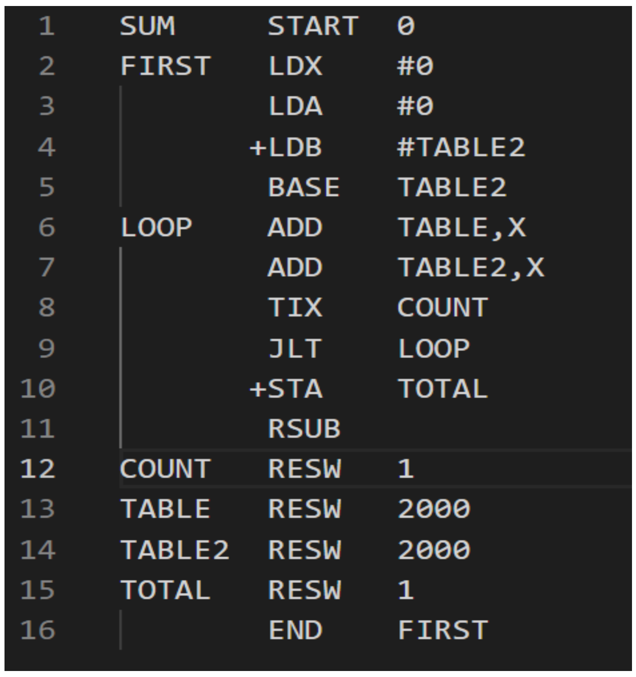
The tables file delineates the structure and description of all tables generated by the assembler. Utilizing the map data structure, tables such as SYMBOL, OPCODE, REG, LIT, and BLOCK are defined. The SYMBOL TABLE houses symbols from the input file, while the OPCODE TABLE comprises descriptions of all SIC/XE opcodes. LIT TABLE defines literals, and the BLOCK TABLE lists all blocks within the input file.

DATA STRUCTURES USED:

For the implementation of Symbol Table, Opcode Table, Register Table, Literal Table, Block Table Maps have been used. Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order. Maps of these tables contain a string as key and a Struct storing information of the key element as mapped values. A structure is a user-defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type. For SYMTAB, the mapped struct value contains address, name, block number, a character exists (represents whether label exists in symbol table or not) and an integer relative (represents whether label is relative or not). For OPTAB, the mapped struct value contains information of opcode like name, format and a character exists (represents whether the opcode is valid or not). For LITTAB, the mapped struct value contains value, address, block number and a character exists (represents whether the literal exits in the literal table or not). For REGTAB, the mapped struct value contains information of registers like it's numeric equivalent and a character exists (represents whether the registers exits or not). For BLOCKS, the mapped struct value contains information of blocks like its name, start address, block number, location counter value for end address of block and a character exists (represents whether the block exits or not).

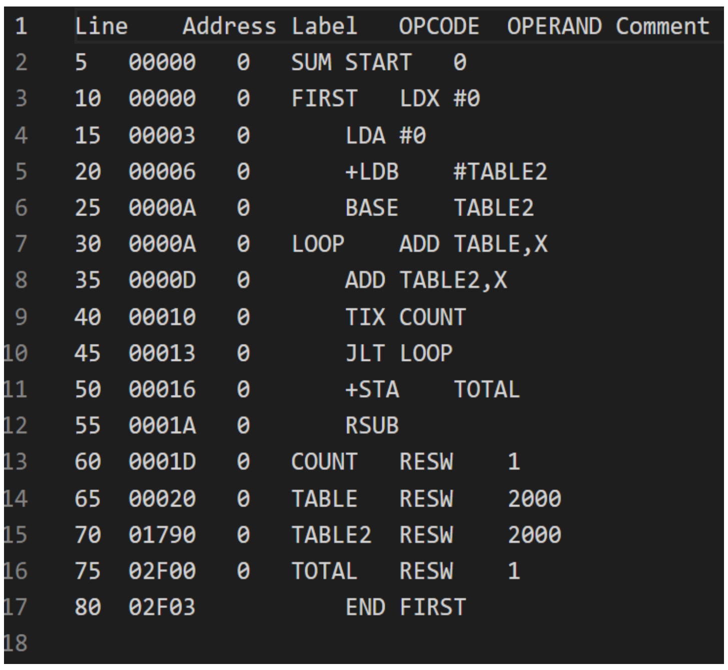
OBJECT PROGRAMS:

Sample_Program:



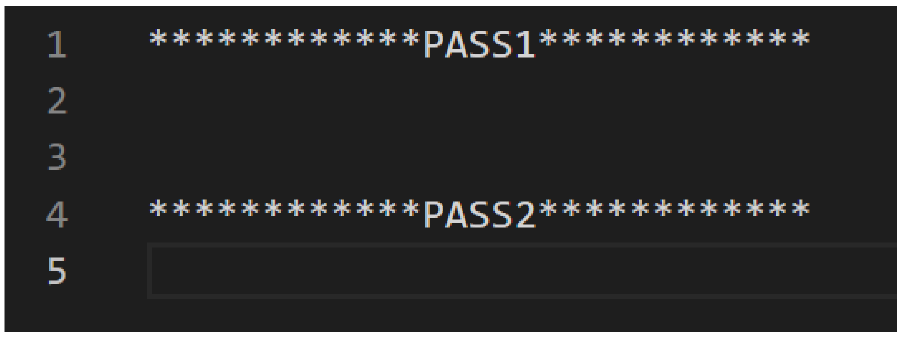
```
SUM      START   0
FIRST    LDX     #0
         LDA     #0
        +LDB     #TABLE2
         BASE    TABLE2
LOOP     ADD     TABLE,X
         ADD     TABLE2,X
         TIX     COUNT
         JLT     LOOP
        +STA     TOTAL
         RSUB
COUNT    RESW    1
TABLE    RESW    2000
TABLE2   RESW 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

Intermediate_Sample:



```
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
```

Error_Sample

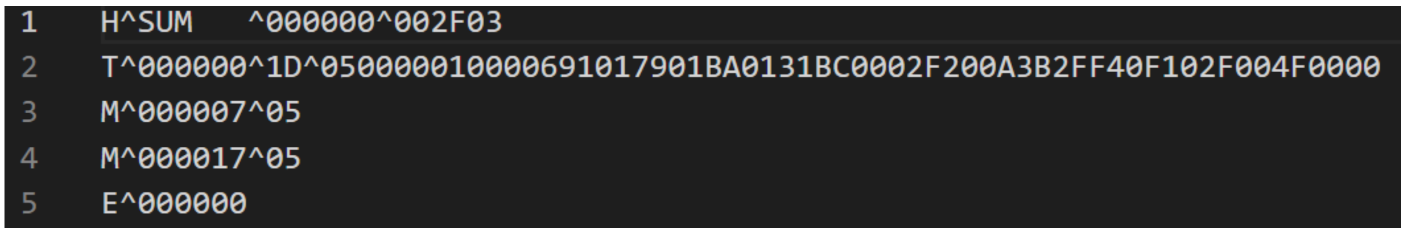


```
************PASS1************


************PASS2************

```

Object_Sample



```
H^SUM   ^000000^002F03
T^000000^1D^050000010000691017901BA0131BC0002F200A3B2FF40F102F004F0000
M^000007^05
M^000017^05
E^000000
```