CSN-252

SYSTEM SOFTWARE

TUTORIAL-8

DESIGN OF SIC/XE ASSEMBLER

Name-Jagrati Kaushik

Enrollment no-22114040

Batch – O2

INTRODUCTION

The SIC/XE assembler, crafted using C++ programming, operates through two passes, each serving a distinct purpose.

During Pass 1- the assembler constructs a symbol table and generates an intermediate file pivotal for the subsequent Pass 2.

In Pass 2 - the assembler produces a comprehensive listing file encompassing the original assembly code, alongside pertinent details such as addresses, block numbers, and corresponding object codes for every instruction.

Additionally, it generates an object program and an error file, which meticulously pinpoints any discrepancies within the input assembly program. This sophisticated assembler accommodates the entire spectrum of SIC/XE instructions, alongside embracing fundamental assembler functionalities like literals, program blocks, expressions, and symbol-defining statements.

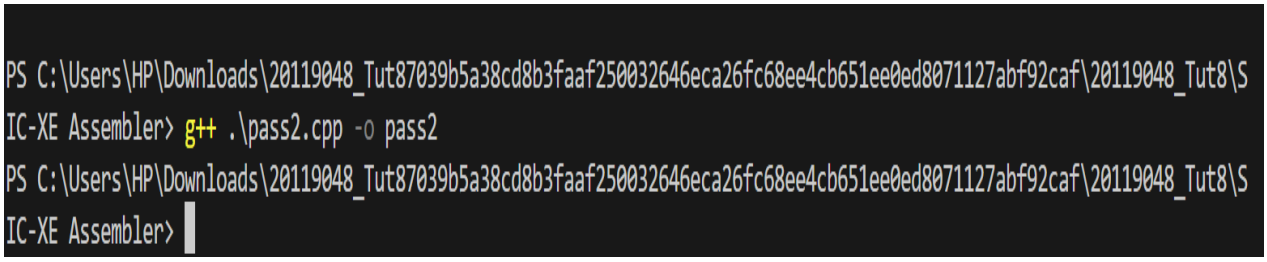
STEPS TO COMPILE

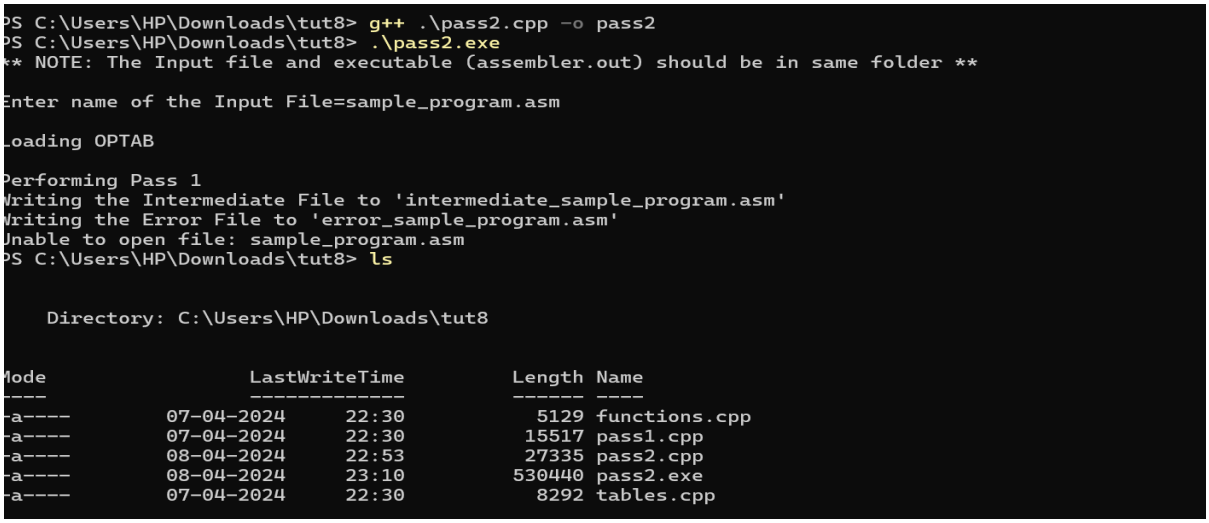
1. Open terminal in folder having the C++ files and the input txt file.

2. Check for the C++ files and compile them to make the executable file.

3. Run the file according to the following images.

4. Check the newly generated files in the folder i.e. error file, intermediate file and assembled instructions file.





ARCHITECTURE AND WORKING

PASS-1

During Pass 1, the assembler initiates the creation of intermediate and error files utilizing the input source file. Should the intermediate file fail to open or if the source file is inaccessible, corresponding errors are logged in the error file. In the initial stages of Pass 1, the assembler scrutinizes the first line of the input to determine if it constitutes a comment. If identified as such, the comment is appended to the intermediate file, and the line number is updated accordingly. This iterative process persists until a non-comment line emerges, signaling the assembler to commence opcode validation.

Upon encountering the 'START' opcode, the assembler updates line numbers, LOCCTR (Location Counter), and the Start Address. Alternatively, if no 'START' opcode is present, the Start Address and LOCCTR are set to 0. A singular while loop governs the pass, halting only upon encountering the 'END' opcode. Within this loop, the assembler continues to analyze each line, distinguishing between comment and non-comment lines. Comment lines are logged and line numbers are adjusted, while non-comment lines undergo further scrutiny.

In the case of a non-comment line, the assembler checks for the presence of a label. Should a label exist, it is cross-referenced with the SYMTAB (Symbol Table). If a duplicate symbol is identified, an error message ('Duplicate symbol') is recorded in the error file. Conversely, if the label is unique, it is assigned an address and associated attributes before being stored in the SYMTAB.

Subsequently, the assembler verifies if the opcode is listed in the OPTAB (Opcode Table). If present, the opcode's format is determined, and LOCCTR is incremented accordingly. If the opcode is absent from the OPTAB, alternative opcodes such as 'RESW', 'BYTE', 'RESBYTE', 'WORD', 'LTORG', 'ORG', 'BASE', 'USE', and 'EQU' are explored. Based on the opcode encountered, the assembler populates the appropriate entries in the tables file map. For instance, encountering 'USE' triggers the insertion of a new BLOCK entry in the BLOCK map, while 'ORG' redirects the LOCCTR to the specified operand value.

Special handling is allocated for 'LTORG' and 'EQU' opcodes. The 'LTORG' operation invokes the handle_LTORG() function defined in pass1.cpp, whereas 'EQU' operations undergo validation to determine if the operand constitutes a valid expression before being entered into the SYMTAB.

Failure to match the opcode against any valid entries results in the assembler logging an error message in the error file. Upon loop completion, the assembler computes the program length and outputs the LITTAB (Literal Table) and SYMTAB. Subsequently, it proceeds to Pass 2 with the intermediate file.

The evaluateExpression() function plays a pivotal role in evaluating expressions. Utilizing pass-by-reference, this function iteratively extracts symbols from expressions. Symbols not found in the SYMTAB trigger error messages in the error file. The variable 'pairCount' tracks the nature of the expression, distinguishing between absolute and relative expressions.

Unexpected values of 'pairCount' prompt error message generation, ensuring expression integrity.

PASS-2

In Pass 2, the assembler harnesses the intermediate file generated by Pass 1, utilizing pass2.cpp to produce a listing file and an object program through the readIntermediateFile() function. Similarly to Pass 1, if either the intermediate file or the object file fails to open, an error message is logged in the error file.

Upon successful file access, the assembler initiates by reading the first line of the intermediate file. Comment lines are identified and transcribed to the intermediate file. Conversely, if an opcode is detected, the assembler proceeds with opcode processing. Upon encountering 'START', the start address is initialized within LOCCTR, and the line is inscribed into the listing file. Subsequently, the header record is generated.

The assembler continues parsing the intermediate file until it encounters the 'END' opcode. Utilizing the textrecord() function, it composes the object program and updates the listing file. Object code composition is contingent upon the instruction formats utilized, with format 3 and 4 instructions invoking the createObjectCodeFormat34() function. Upon completion of all text records, the assembler appends the end record.

The readIntermediateFile() function serves as a crucial component, accepting parameters such as line number, LOCCTR, opcode, operand, label, and input/output files. This function navigates various opcode scenarios, taking operand and half-byte considerations into account while calculating object codes for instructions. Additionally, it generates modification records if deemed necessary.

The WriteEndRecord() function fulfills the role of crafting the end record for the program. Following the execution of pass1.cpp, the tables (SYMTAB, LITTAB, etc.) are printed in a distinct file, after which pass2.cpp is executed to finalize the assembly process.

FUNCTIONS

In the functions.cpp file, a collection of essential functions is housed, each serving a distinct purpose pivotal to the operation of both Pass 1 and Pass 2 in the assembler:-

- **readFirstNonWhiteSpace():** This function scans the current line to retrieve the first non-whitespace character. Employing pass by reference, it updates the index of the string where the non-whitespace character is encountered.
- **getString():** Accepting a character as input, this function returns a string.
- **checkWhiteSpace():** With a Boolean return type, this function verifies if whitespace is present in the input string.
- **expandString():** This function expands the input string to a specified length, inserting a designated character as required. Parameters include the string to be expanded, the desired output string length, and the character for expansion.

- **checkCommentLine():** Determines whether the current line is a comment line. It accepts a string input and returns a Boolean value based on the outcome.
- **writeToFile():** Writes a provided string onto a specified file. Parameters include the file and the string to be written.
- **getFlagFormat():** This function retrieves the flag bit from the input string, returning either the flag bit or a null string if not present.
- **intToStringHex():** Converts an integer input into its hexadecimal equivalent, returning the result as a string.
- **stringHextoInt():** Converts a hexadecimal string input into an integer, with the return type being an integer.

TABLES

In this file, a comprehensive description of all the tables generated by the assembler is provided. Each table's structure is defined utilizing the map data structure. The following tables are outlined:

SYMBOL TABLE: This table houses all symbols extracted from the input file. Each symbol is associated with relevant attributes such as its address and other necessary values.

OPCODE TABLE: Here, detailed descriptions of all SIC/XE opcodes are stored. For each opcode, pertinent information such as its format and functionality is provided.

REGISTER TABLE: This table contains information regarding SIC/XE registers. It outlines the available registers and their corresponding codes or addresses.

LITERAL TABLE: Within this table, all literals encountered in the input file are defined. Each literal is assigned an appropriate representation and may include additional attributes as necessary.

BLOCK TABLE: This table is utilized to enumerate and organize all program blocks present within the input file. It provides a structured overview of the various blocks and their respective attributes.

Utilizing the map data structure, these tables efficiently organize and manage the diverse elements encountered during the assembly process. They serve as indispensable resources for referencing and processing data throughout both Pass 1 and Pass 2 of the assembler.

DATA STRUCTURES USED

The implementation of various tables in the assembler leverages maps, which are associative containers in C++. Maps store elements consisting of a key value and a corresponding mapped value, maintaining a specific order. Here's a breakdown of the structure and contents of each table:

Symbol Table (SYMTAB):

Address: Represents the memory address associated with the symbol.

Name: Holds the name of the symbol.

Block Number: Indicates the block number to which the symbol belongs.

Exists: Character flag indicating whether the label exists in the symbol table or not.

Relative: Integer flag denoting whether the label is relative or not.

Opcode Table (OPTAB):

Name: Specifies the mnemonic name of the opcode.

Format: Describes the format of the opcode.

Exists: Character flag indicating whether the opcode is valid or not.

Literal Table (LITTAB):

Value: Represents the value of the literal.

Address: Denotes the memory address associated with the literal.

Block Number: Indicates the block number to which the literal belongs.

Exists: Character flag indicating whether the literal exists in the literal table or not.

Register Table (REGTAB):

Numeric Equivalent: Specifies the numeric representation of the register.

Exists: Character flag indicating whether the register exists or not.

Block Table (BLOCKS):

Name: Specifies the name of the block.

Start Address: Denotes the memory address where the block begins.

Block Number: Indicates the block number.

Location Counter Value for End Address: Represents the value of the location counter for the end address of the block.

Exists: Character flag indicating whether the block exists or not.

By utilizing maps with structured mapped values, the assembler efficiently organizes and manages information pertaining to symbols, opcodes, literals, registers, and program blocks, facilitating seamless processing during both Pass 1 and Pass 2. The structured approach enhances clarity and accessibility of data within the assembler.

OBJECT PROGRAMS

A. Sample Program (Question 3 of section 2.2)

Filename: sample_program.asm

ASSEMBLY CODE

```
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
```

OBJECT FILE

```
H^SUM   ^000000^002F03
T^000000^1D^0500000100006910179 01BA0131BC0002F200A3B2FF40F102F004F0000
M^000007^05
M^000017^05
E^000000
```

LISTING FILE

```
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
```

B. Correct Program Filename:

correct_test_program.asm

ASSEMBLY CODE

```
HEAD    START   0
FIRST   LDT     #11
        LDX     #0
MOVECH  LDCH    STR1,X
        STCH    STR2,X
        TIXR    T
        JLT     MOVECH
STR1    BYTE    C'TEST STRING'
STR2    RESB    11
        END     FIRST
```

OBJECT FILE

```
H^HEAD  ^000000^000027
T^000000^1C^75000B050000053A00857A010B8503B2FF5544553542053545249 4E47
E^000000
```

LISTING FILE

```
Line    Address Label   OPCODE  OPERAND ObjectCode  Comment
5   00000   0   HEAD    START   0
10  00000   0   FIRST   LDT #11 75000B
15  00003   0       LDX #0  050000
20  00006   0   MOVECH  LDCH    STR1,X  53A008
25  00009   0       STCH    STR2,X  57A010
30  0000C   0       TIXR    T   B850
35  0000E   0       JLT MOVECH  3B2FF5
40  00011   0   STR1    BYTE    C'TEST STRING'  544553542053545249 4E47
45  0001C   0   STR2    RESB    11
50  00027           END FIRST
```

C. Incorrect Test Program

Filename: incorrect_test_program.asm

ASSEMBLY CODE

```
HEAD    START   0
FIRST   LDT     #11
        LDX     #0
MOVECH  LDC     STR1,X
        STCH    STR1,X
        TIXR    T
        JL      MOVECH
STR1    BYTE    C'TEST STRING'
STR2    RESB    11
        END     FIRST
```

OBJECT FILE

```
**********PASS1**********
Line: 20 : Invalid OPCODE. Found LDC
Line: 35 : Invalid OPCODE. Found JL


**********PASS2**********
```

It is evident that the assembler functions properly. For the two appropriately written input files, it produces the appropriate Object Programs. The faults are accurately displayed in the error file for the erroneous input programs.