


CSN-252: **System Software** **Project** **Documentation** **SIC/XE Assembler**



Submitted By:

Ashutosh Kumar

21114021

Batch-O1

Introduction

SIC/XE stands for "Simplified Instructional Computer Extra Equipment". It is an advanced version of the SIC architecture that includes additional features and capabilities. SIC/XE is designed to be upward-compatible with SIC, which means that programs written for the SIC architecture can be run on the SIC/XE architecture without modification. This makes it easier for developers to migrate their programs to the new architecture.

Assembler

The Assembler is implemented using C++ programming language.

This Assembler implements following Machine-Independent Features.

1. Literals
2. Symbol Defining Statements
3. Expressions
4. Control Sections

Assembler : Design

`tables.cpp`

This file contains the data structures required:


- The `info_op`, `info_reg`, `info_sym`, `info_literal`, and `info_mod` structs are used to store information about opcodes, registers, symbols, literals, and modifications respectively.

- OPTAB map contains information about opcode mnemonics and their corresponding opcode values and formats as it is mapped to the `info_op` struct. Similarly, the REGISTER map contains information about register mnemonics and their corresponding register numbers as it is mapped to the `info_op` struct..
- The `reg_num` function returns the corresponding register number given the register mnemonic.
- The `init_tables` function initializes the OPTAB and REGISTER maps with their respective values.

utility.cpp

It contains useful functions that will be required by other files .

- ***bool isNumber(string str)***: This function takes a string as input and returns a boolean indicating whether the string represents a decimal number or not. The function checks if each character in the string is a digit or not and returns false if any non-digit character is found.
- ***int strToDec(string s)***: This function takes a string representing a decimal number as input and returns the corresponding integer value. The function works by iterating over the characters in the string from the right end and multiplying each digit by a power of 10 and adding it to the result.
- ***int strToHex(string s)***: This function takes a string representing a hexadecimal number as input and returns the corresponding integer value. The function works similarly to the `strToDec` function, but instead multiplies each digit by a power of 16.
- ***int strToASCII(string s)***: This function takes a string as input and returns the corresponding integer value obtained by treating the string as a sequence of ASCII characters. The function iterates over the



characters in the string from the right end and multiplies the ASCII value of each character by a power of 256 and adds it to the result.

- ***bool isHex(string str)***: This function takes a string as input and returns a boolean indicating whether the string represents a valid hexadecimal number or not. The function checks if each character in the string is a digit or a valid hexadecimal character (i.e., A-F or a-f).
- ***vector<string> parseWords(string line)***: This function takes a string representing a line of instructions as input and returns a vector containing three strings, which represent the three parameters of the instruction. The function splits the input string into words using spaces as delimiters and returns the first three words as a vector.
- ***string toHex(int num, int len)***: This function takes an integer value and a length as input and returns a string representing the hexadecimal value of the integer.

These functions are useful for processing strings of numbers and characters and converting them to their corresponding integer or hexadecimal representations. These operations are commonly used in computer programming and are essential for working with assembly language and machine code.

csect.cpp

The given section contains two structs: instruction and csect.

- The instruction struct is used to represent an assembly instruction. It contains the following members:
 - address: An integer representing the location counter (locctr) of the instruction.
 - label: A string representing the label of the instruction.
 - mnemonic: A string representing the mnemonic of the instruction (e.g. "LDA", "ADD", etc.).

- operand: A string representing the operand of the instruction (e.g. "X'0A", "BUFFER", etc.).
 - objCode: A string representing the object code of the instruction, which will be generated during the assembly process.
 - length: An integer representing the length of the instruction in bytes.
- The csect struct is used to represent a control section, which is a portion of the program that is loaded into memory as a single unit. It contains the following members:
 - name: A string representing the name of the control section.
 - start: An integer representing the starting address of the control section.
 - end: An integer representing the ending address of the control section.
 - valid_base: A boolean indicating whether a base register has been set for this control section.
 - def: A map that stores the defined symbols and their corresponding addresses.
 - ref: A vector that stores the names of the external symbols that are referenced in this control section.
 - litTab: A map that stores the literals used in the program and their corresponding information, such as the literal value and its length.
 - mod_record: A vector that stores the modification records for this control section. A modification record indicates that a particular memory location needs to be modified at link-edit time.
 - instructions: A vector that stores the instructions of the program in this control section.

- symTab: A map that stores the symbols used in the program and their corresponding information, such as their addresses and whether they are relative or absolute.

These structs are used to store information about the program being assembled, including its instructions, symbols, and control sections. The information stored in these structs is used during the assembly process to generate the object code for the program.

pass1.cpp


It takes an input file that contains assembly code as input and generates a symbol table, literal table, and an intermediate code file. This is the first step in the process of assembling code for execution on a computer.

The code starts by opening the input file and error file. If the input file is not found, an error message is displayed and the program exits. The input file is read line by line and each line is stored in a vector.

The program then initializes several variables, including a location counter, vectors for instructions and literals, and a vector for control sections. A control section is defined by the presence of a label, and all instructions within that section are assigned to that label.

The program then begins processing each line of the input file. The line is parsed into its label, mnemonic, and operand. If a control section has not yet been defined, the program creates one based on the label of the first instruction. The instruction is then added to the instruction vector for the current control section.

The program then checks the mnemonic to determine how to handle the instruction. If the mnemonic is "START", the location counter is set to the value of the operand (converted from hexadecimal if necessary), and the



start address for the current control section is set to the new location counter value.

Similarly, It processes all the different instructions and assembler directives.

After processing all instructions in the input file, the program writes the symbol and literal tables to the intermediate file, along with the instruction addresses and lengths.

pass2.cpp


Since the addresses have been assigned in Pass1, in Pass 2 we are first writing the complete intermediate file by calculating the object codes of respective instructions and storing in the structs.

Coming to the major function of the pass2 in this assembler whose code is in the function createObjProg() which is explained below:

The function starts by opening an output file named object_program.txt. It then iterates through all the control sections in the source code. For each control section, it writes the header record in the object file. The header record contains the section name, starting address, and length of the section.

After that, the function writes definition records for all the symbols defined in the control section. The definition record contains the symbol name and its address. Similarly, it writes reference records for all the symbols referred to in the control section.

Then the function writes the text records for the instructions in the control section. It iterates through all the instructions and checks whether the instruction has a corresponding object code or not. If it has, it calculates the current size of the object code and checks if the object code can fit in the current text record or not. If it can, it adds the object code to the current text



record. Otherwise, it writes the current text record to the object file and starts a new text record for the current instruction.

If the current instruction is a reserve word (RESW or RESB), it writes the current text record (if it has any object code) to the object file and starts a new text record for the next instruction.

After writing all the text records, the function writes the modification records for all the instructions with immediate addressing.

Finally, the function writes the end record in the object file. If it is the last control section, it writes the end record with the starting address of the first control section. Otherwise, it writes the end record without any address.

At the end of the function, the output file and intermediate file are closed, and a message is printed to indicate that the object program file has been written.

Assembler : Data Structures

- **REGTAB :**

Contains information of the registers like its numeric equivalent , character representing , whether such register exists or not .

- **LITTAB :**

Contains information of literals like its value , address , block number , a character representing whether the literal exists in literal table or not . It has also been shown in “*lit_tab.txt*”

- **SYMTAB :**

Contains information of labels like , name , address , block number , character representing whether the label exists in the symbol table or not , an integer representing whether label is relative or not . It has also been shown in “***symtab.txt***”

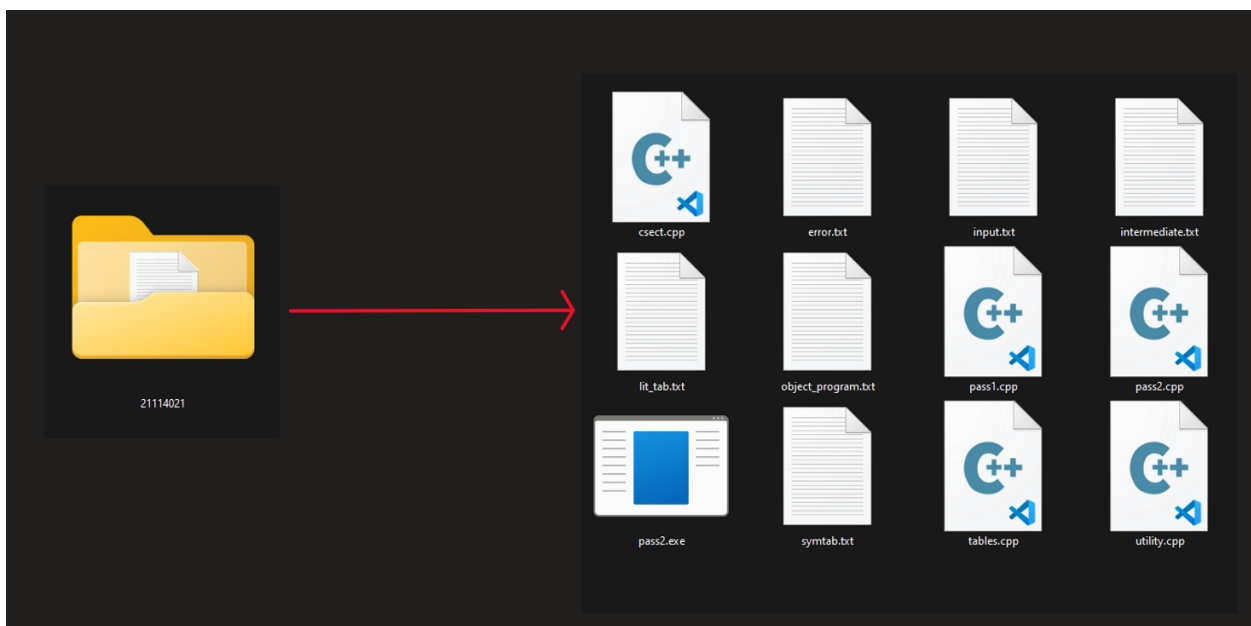
- **OPTAB :**

Contains information of opcode like name , format , a bool representing whether the opcode is valid or not .

Compilation and Execution

To compile and execute the SIC/XE Assembler, follow these steps:

1. Unzip the Attached File “**21114021.zip**”.
2. Upon unzipping the above folder will contain the following files.



3. Open Terminal at that location and type “**code .**” to open the directory in VS Code.

```
PS D:\IIT Roorkee CSE\2-2\CSN 252\21114021> code .
```

4. In VS Code, If a user wants to change the input then can change in “**input.txt**”. The Current input is:

```
1 COPY START 0
2 EXTDEF BUFFER, BUFEND, LENGTH
3 EXTREF RDREC, WRREC
4 BASE LENGTH
5 LDB LENGTH
6 FIRST STL RETADR
7 CLOOP +JSUB RDREC
8 LDA LENGTH
9 COMP #0
10 JEQ ENDFIL
11 +JSUB WRREC
12 J CLOOP
13 ENDFIL LDA =C'EOF'
14 STA BUFFER
15 LDA #3
16 STA LENGTH
17 +JSUB WRREC
18 J @RETADR
19 RETADR RESW 1
20 LENGTH RESW 1
21 LTORG
22 BUFFER RESB 4096
23 BUFEND EQU *
24 MAXLEN EQU BUFEND-BUFFER
25
26 RDREC CSECT
27
28 SUBROUTINE TO READ RECORD INTO BUFFER
29
30
31
32
33 EXTREF BUFFER, LENGTH, BUFEND
34 CLEAR X
35 CLEAR A
36 CLEAR S
37 LDT MAXLEN
38 RLOOP TD INPUT
39 JEQ RLOOP
40 RD INPUT
41 COMPR A, S
42 JEQ EXIT
43 +STCH BUFFER, X
44 TIXR T
45 JLT RLOOP
46 EXIT +STX LENGTH
47 RSUB
48 INPUT BYTE X'F1'
49 MAXLEN WORD BUFEND-BUFFER
50
51 WRREC CSECT
52
53 SUBROUTINE TO WRITE RECORD FROM BUFFER
54
55
56
57 EXTREF LENGTH, BUFFER
58 CLEAR X
59 +LDT LENGTH
60 WLOOP TD =X'05'
61 JEQ WLOOP
62 +LDCH BUFFER, X
63 WD =X'05'
64 TIXR T
65 JLT WLOOP
66 RSUB
67 END FIRST
```

5. To run the code type the following command in Terminal.

```
PS D:\IIT Roorkee CSE\2-2\CSN 252\21114021> g++ pass2.cpp -o pass2
PS D:\IIT Roorkee CSE\2-2\CSN 252\21114021> .\pass2
Initializing Tables
PASS 1 Executed successfully
Intermediate File Written.
Object Program File Written.
```

Terminal Outputs shows that the programs have run successfully.

6. This shows that the corresponding intermediate file and Object Program file has been generated in “**intermediate.txt**” and “**object_program.cpp**” respectively. It also generates “**error.txt**” if there are any errors.

```

intermediate.txt X
intermediate.txt
1
2 COPY
3 000000 COPY START 0
4 000000 EXTDEF BUFFER, BUFEND, LENGTH
5 000000 EXTREF RDREC, WRREC
6 000000 BASE LENGTH
7 000000 LDB LENGTH 6B202D
8 000003 FIRST STL RETADR 172027
9 000006 CLOOP +JSUB RDREC 4B100000
10 00000A LDA LENGTH 032023
11 00000D COMP #0 290000
12 000010 JEQ ENDFIL 332007
13 000013 +JSUB WRREC 4B100000
14 000017 J CLOOP 3F2FEC
15 00001A ENDFIL LDA =C'EOF' 032016
16 00001D STA BUFFER 0F2016
17 000020 LDA #3 010003
18 000023 STA LENGTH 0F200A
19 000026 +JSUB WRREC 4B100000
20 00002A J @RETADR 3E2000
21 00002D RETADR RESW 1
22 000030 LENGTH RESW 1
23 000030 LTORG
24 000033 * =C'EOF' 454F46
25 000036 BUFFER RESB 4096
26 000036 BUFEND EQU *
27 000036 MAXLEN EQU BUFEND-BUFFER
28 000036 RDREC CSECT
29

```

```

29
30  RDREC
31  000000      EXTREF      BUFFER,LENGTH,BUFEND
32  000000      CLEAR      X      B410
33  000002      CLEAR      A      B400
34  000004      CLEAR      S      B440
35  000006      LDT        MAXLEN  77201F
36  000009      RLOOP      TD      INPUT  E3201B
37  00000C      JEQ        RLOOP  332FFA
38  00000F      RD         INPUT  DB2015
39  000012      COMPR      A,S     A004
40  000014      JEQ        EXIT    332009
41  000017      +STCH      BUFFER,X 57900000
42  00001B      TIXR       T      B850
43  00001D      JLT        RLOOP  3B2FE9
44  000020      EXIT      +STX      LENGTH 13100000
45  000024      RSUB
46  000027      INPUT      BYTE     X'F1'  F1
47  000028      MAXLEN     WORD     BUFEND-BUFFER 000000
48  000028      WRREC      CSECT
49

```

```

9
0  WRREC
1  000000      EXTREF      LENGTH,BUFFER
2  000000      CLEAR      X      B410
3  000002      +LDT       LENGTH  77100000
4  000006      WLOOP      TD      =X'05'  E32012
5  000009      JEQ        WLOOP  332FFA
6  00000C      +LDCH      BUFFER,X 53900000
7  000010      WD         =X'05'  DF2008
8  000013      TIXR       T      B850
9  000015      JLT        WLOOP  3B2FEE
0  000018      RSUB
1  00001B      *         =X'05'  05
2  000018      END        FIRST
3

```

Fig.: intermediate.txt

```
object_program.txt X
object_program.txt
1 HCOPY 000000001036
2 DBUFEND001036BUFFER000036LENGTH000030
3 RRDREC WRREC
4 T0000001D6B202D1720274B1000000320232900003320074B1000003F2FEC032016
5 T00001D100F20160100030F200A4B1000003E2000
6 T00003303454F46
7 M00000705+RDREC
8 M00001405+WRREC
9 M00002705+WRREC
10 E0000000
11
12 HRDREC 00000000002B
13 RBUFFERLENGTHBUFEND
14 T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
15 T00001D0E3B2FE9131000004F0000F1000000
16 M00001805+BUFFER
17 M00002105+LENGTH
18 M00002806+BUFEND
19 M00002806-BUFFER
20 E
21
22 HWRREC 00000000001C
23 RLENGTHBUFFER
24 T0000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
25 M00000305+LENGTH
26 M00000D05+BUFFER
27 E
28
```

Fig.: object_program.txt

7. This Assembler also writes LITTABs and SYMTABs of respective control section.

```

lit_tab.txt x
lit_tab.txt
1  Literal Table for COPY
2      Literal Address Length  Value
3      =C'EOF' 000033 3      4542278
4
5  Literal Table for RDREC
6      Literal Address Length  Value
7
8  Literal Table for WRREC
9      Literal Address Length  Value
10     =X'05' 00001B 1      5
11
12

```

Fig.: lit_tab.txt

```

symtab.txt x
symtab.txt
1  Symbol Table for COPY
2      Symbol Address Rel/Abs
3      BUFEND 001036 Rel
4      BUFFER 000036 Rel
5      CLOOP 000006 Rel
6      COPY 000000 Rel
7      ENDFIL 00001A Rel
8      FIRST 000003 Rel
9      LENGTH 000030 Rel
10     MAXLEN 001000 Abs
11     RETADR 00002D Rel
12
13 Symbol Table for RDREC
14     Symbol Address Rel/Abs
15     EXIT 000020 Rel
16     INPUT 000027 Rel
17     MAXLEN 000028 Rel
18     RLOOP 000009 Rel
19
20 Symbol Table for WRREC
21     Symbol Address Rel/Abs
22     WLOOP 000006 Rel
23

```

Fig.: symtab.txt



8. This completes the execution of our assembler.

