

Project Report

On

SIC/XE Assembler that supports Program Blocks



CSN 252

System Software

Under

Prof. Manoj Misra

Date: 11/04/2024

BY

Name: Aditya Mahesh Mundada

Enrollment no: 22114058

Summary:

I have successfully implemented an assembler for SIC/XE that supports program blocks, literals, expressions as well as other basic functionalities.

It is a two pass assembler. It does the following:

Pass 1:

- It identifies all the labels in the program and stores it in a map LABELS that stores the Label name with its relative location in the given block as well as the block number.
- It also identifies Literals and stores them in LITTAB. If we do not find an LORG statement then all literals are stored with -1 signifying they are yet to be placed. When we encounter an LORG or there are still literals at the end of LITTAB then they are stored below them and their blocks, LOCCTR are then updated in pass 1.

```
}
else if (opcode == "LORG")
{
    for (auto it : LITTAB)
    {
        if (it.second.first == -1)
        {
            operand = it.first;
            LITTAB[it.first] = {pc, currentblk}; // stores location of literal;
            if (operand[1] == 'x')
            {
                string opr = operand.substr(3, operand.length() - 4);
                intermed.push_back({{pc, currentblk}, "=x'" + opr + "'"});
                int curr = pc;
                pc += opr.length() / 2;
                inst.push_back({{pc, currentblk}, {"BYTE", opr}});
                block_length[currentblk] = pc;
                pl[{pc, currentblk}] = curr;
            }
            else if (operand[1] == 'c')
            {
                string ops = operand.substr(3, operand.length() - 4);
                string opr = charTOAsc(ops);
                int curr = pc;
                intermed.push_back({{pc, currentblk}, "=c'" + opr + "'"});
                pc += opr.length() / 2;
                inst.push_back({{pc, currentblk}, {"BYTE", opr}});
                block_length[currentblk] = pc;
                pl[{pc, currentblk}] = curr;
            }
        }
    }
}
```

- The block name is mapped to the corresponding block number using the map block ,the block length and block starting address is also stored in the map block_length and the blockstart maps. Together they can be thought as making the BLOCK table.
- The **INST** vector of pair<pair<int,int>,<string,string> >stores the program counter, block address, instruction opcode and operand respectively of each instruction to be executed.
- Similarly, the **INTERMED** vector stores information in a format suitable to print the intermediate file.

```

if (opcode == "USE")
{
    if (tokens.size() == 1)
    {
        currentblk = 0;
        pc = block_length[currentblk];
    }
    else
    {
        if (block.find(operand) != block.end())
        {
            currentblk = block[operand];
            pc = block_length[currentblk];
        }
        else
        {
            no_of_blocks++;
            block[operand] = no_of_blocks - 1;
            pc = 0;
            currentblk = block[operand];
        }
    }
    inst.push_back({{-3, currentblk}, {"USE", ""}});
}

```

This is how program blocks are handled in pass 1.

We identify whether the use statement makes a new block or uses a previously defined block and work accordingly.

Pass 2:

In pass 2 the instructions written in **INST** vector are executed one by one and assembled to the corresponding object code.

We check the opcode of each instruction as well as operands to find what operation is to be performed, what are the type and length of instructions, whether to perform PC relative, BASE relative assembly, whether operand is to be used as immediate, indirect or direct addressing types. We assemble each instruction one by one. If some line in code is not to be executed i.e it is just an

assembler directive then it is treated so and no object code is made corresponding to it.

Finally, the assembled instructions are stored in vector **TEXTREC** and printed following all the conditions of printing text records.

Also, other records and their lengths are printed as per required formats by previously stored information.

All of this occurs in

```
string singleIns(int lc, string opcode, string operand1, string operand2 = "")
```

The lc is the pc of next instruction used in pc relative assembly(blocks considered), the operand 2 is set only when indexing comes in play.

```
else if (f == 3)
{
    string ans = hexToBinaryString(f34[opcode]);
    if (operand1[0] == '#')
        ans[7] = '1';
    else if (operand1[0] == '@')
        ans[6] = '1';
    else
    {
        ans[6] = '1';
        ans[7] = '1';
    }

    if (operand1[0] == '#' || operand1[0] == '@')
        operand1 = operand1.substr(1);
    if (operand2 == "X")
        ans.push_back('1');
    else
        ans.push_back('0');
    // cout<<operand1<<"<<endl;
    if (isabs(operand1))
    {
        ans += "000";
        ans += intToBinaryString(absolute[operand1], 12);
    }
    else if (islabel(operand1))
    {
        // cout<<operand1<<endl;
        int diff = LABELS[operand1].first + blockstart[LABELS[operand1].second] - lc;
        if (diff < 2048 && diff >= -2048)
        {
            ans += "010";
            ans += intToBinaryString(diff, 12);
        }
    }
}
```

```

else if (base_rel)
{
    int disp = LABELS[operand1].first + blockstart[LABELS[operand1].second] - base;
    if (disp >= 0 && disp <= 4095)
    {
        ans += "100";
        ans += intToBinaryString(disp, 12);
    }
}
else{
    if(base_rel==0) err<<"Displacement too large but directive BASE not FOUND so base relative not possible";
    else err<<"Displacement too large to be handled";
}
}

else if (islit(operand1))
{
    int diff = LITTAB[operand1].first + blockstart[LITTAB[operand1].second] - 1c;
    // cout<<operand1<<"in lit"<<endl;
    if (diff < 2048 && diff >= -2048)
    {
        ans += "010";
        ans += intToBinaryString(diff, 12);
    }
    else if (base_rel)
    {
        int disp = LITTAB[operand1].first + blockstart[LITTAB[operand1].second] - base;
        if (disp >= 0 && disp <= 4095)
        {
            ans += "100";
            ans += intToBinaryString(disp, 12);
        }
    }
}
else{
    if(base_rel==0) err<<"Displacement too large but directive BASE not FOUND so base relative not possible";
    else err<<"Displacement too large to be handled";
}
}

```

This is just a small snippet of code used to manage format 3 assembly.

Error handling:

Various errors are handled and printed in error.txt file:

The errors handled include:

Checking whether we try to define a label twice.

Check whether we have an invalid expression(very basic errors)

Checking if a displacement of more than 2047 is tried to be done even when programmer does not write a BASE assembler directive flagging that base is not available to use.

Trying to use indexing with immediate/indirect mode of assembly.

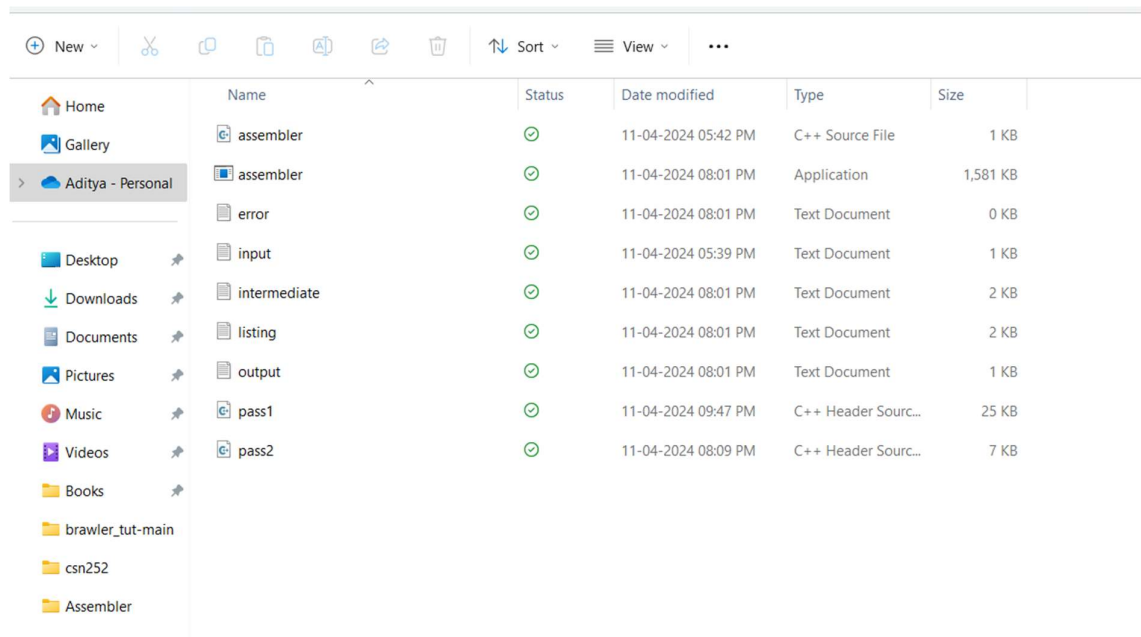
Trying to use invalid registers, etc.

Files and Folders:

The following files are used in the program:

1. pass1.hpp : logic of pass1 of assembler.

2. pass2.hpp : logic of pass2 of assembler.
3. Assembler.cpp : main file which you must run in order to avail the functionality.
4. Input.txt: The input is given in this file.
5. Output.txt: The final object program is written in this file.
6. Intermediate.txt: The intermediate file includes block numbers and LOCCTR information of the instructions. It also includes the symbol table.
7. Listing.txt : It contains the listing of the instructions and object codes.



Name	Status	Date modified	Type	Size
assembler	✓	11-04-2024 05:42 PM	C++ Source File	1 KB
assembler	✓	11-04-2024 08:01 PM	Application	1,581 KB
error	✓	11-04-2024 08:01 PM	Text Document	0 KB
input	✓	11-04-2024 05:39 PM	Text Document	1 KB
intermediate	✓	11-04-2024 08:01 PM	Text Document	2 KB
listing	✓	11-04-2024 08:01 PM	Text Document	2 KB
output	✓	11-04-2024 08:01 PM	Text Document	1 KB
pass1	✓	11-04-2024 09:47 PM	C++ Header Sourc...	25 KB
pass2	✓	11-04-2024 08:09 PM	C++ Header Sourc...	7 KB

Input Program:

```

COPY START 0
FIRST STL RETADR
CLOOP JSUB RDREC
LDA LENGTH
COMP #0
JEQ ENDFIL
JSUB WRREC
J CLOOP
ENDFIL LDA =C'EOF'
STA BUFFER
LDA #3
STA LENGTH

```

```

JSUB WRREC
J @RETADR
USE CDATA
RETADR RESW 1
LENGTH RESW 1
USE CBLKS
BUFFER RESB 4096
BUFFEND EQU *
MAXLEN EQU BUFFEND-BUFFER
USE
RDREC CLEAR X
CLEAR A
CLEAR S
+LDT #MAXLEN
RLOOP TD INPUT
JEQ RLOOP
RD INPUT
COMPR A,S
JEQ EXIT
STCH BUFFER,X
TIXR T
JLT RLOOP
EXIT STX LENGTH
RSUB
USE CDATA
INPUT BYTE X'F1'
USE
WRREC CLEAR X
LDT LENGTH
WLOOP TD =X'05'
JEQ WLOOP
LDCH BUFFER,X
WD =X'05'
TIXR T
JLT WLOOP
RSUB
USE CDATA
LTORG
END FIRST

```

This is the sample input and it contains various program blocks.

Snippets of the intermediate file and the listing file are given below:

```
This is an intermediate file
-----
0 0000 COPY START 0
0 0000 FIRST STL RETADR
0 0003 CLOOP JSUB RDREC
0 0006 LDA LENGTH
0 0009 COMP #0
0 000C JEQ ENDFIL
0 000F JSUB WRREC
0 0012 J CLOOP
0 0015 ENDFIL LDA =C'EOF'
0 0018 STA BUFFER
0 001B LDA #3
0 001E STA LENGTH
0 0021 JSUB WRREC
0 0024 J @RETADR
0 0027 USE CDATA
1 0000 RETADR RESW 1
1 0003 LENGTH RESW 1
1 0006 USE CBLKS
2 0000 BUFFER RESB 4096
2 1000 BUFFEND EQU *
2 1000 MAXLEN EQU BUFFEND-BUFFER
2 1000 USE
0 0027 RDREC CLEAR X
0 0029 CLEAR A
0 002B CLEAR S
0 002D +LDT #MAXLEN
0 0031 RLOOP TD INPUT
0 0034 JEQ RLOOP
0 0037 RD INPUT
0 003A COMPR A,S
0 003C JEQ EXIT
0 003F STCH BUFFER,X
0 0042 TIXR T
0 0044 JLT RLOOP
0 0047 EXIT STX LENGTH

Assembler > ≡ intermediate.txt
38 0 004A RSUB
39 0 004D USE CDATA
40 1 0006 INPUT BYTE X'F1'
41 1 0007 USE
42 0 004D WRREC CLEAR X
43 0 004F LDT LENGTH
44 0 0052 WLOOP TD =X'05'
45 0 0055 JEQ WLOOP
46 0 0058 LDCH BUFFER,X
47 0 005B WD =X'05'
48 0 005E TIXR T
49 0 0060 JLT WLOOP
50 0 0063 RSUB
51 0 0066 USE CDATA
52 1 0007 LTORG
53 1 0007 =C'454f46'
54 1 000A =X'05'
55 1 000B END FIRST
56 -----
57 SYMBOL TABLE
58
59 BUFFEND 1071
60 BUFFER 0071
61 CLOOP 0003
62 ENDFIL 0015
63 EXIT 0047
64 FIRST 0000
65 INPUT 006C
66 LENGTH 0069
67 MAXLEN 1071
68 RDREC 0027
69 RETADR 0066
70 RLOOP 0031
71 WLOOP 0052
72 WRREC 004D
73 MAXLEN 1000
```

```
This is the listing file
-----
0 0000 COPY START 0
0 0000 FIRST STL RETADR 172063
0 0003 CLOOP JSUB RDREC 4B2021
0 0006 LDA LENGTH 032060
0 0009 COMP #0 290000
0 000C JEQ ENDFIL 332006
0 000F JSUB WRREC 4B203B
0 0012 J CLOOP 3F2FEE
0 0015 ENDFIL LDA =C'EOF' 032055
0 0018 STA BUFFER 0F2056
0 001B LDA #3 010003
0 001E STA LENGTH 0F2048
0 0021 JSUB WRREC 4B2029
0 0024 J @RETADR 3E203F
0 0027 USE CDATA
1 0000 RETADR RESW 1
1 0003 LENGTH RESW 1
1 0006 USE CBLKS
2 0000 BUFFER RESB 4096
2 1000 BUFFEND EQU *
2 1000 MAXLEN EQU BUFFEND-BUFFER
2 1000 USE
0 0027 RDREC CLEAR X B410
0 0029 CLEAR A B400
0 002B CLEAR S B440
0 002D +LDT #MAXLEN 75101000
0 0031 RLOOP TD INPUT E32038
0 0034 JEQ RLOOP 332FFA
0 0037 RD INPUT DB2032
0 003A COMPR A,S A004
0 003C JEQ EXIT 332008
0 003F STCH BUFFER,X 57A02F
0 0042 TIXR T B850
0 0044 JLT RLOOP 3B2FEA
0 0047 EXIT STX LENGTH 13201F

Assembler > ≡ listing.txt
38 0 004A RSUB 4F0000
39 0 004D USE CDATA
40 1 0006 INPUT BYTE X'F1' F1
41 1 0007 USE
42 0 004D WRREC CLEAR X B410
43 0 004F LDT LENGTH 772017
44 0 0052 WLOOP TD =X'05' E3201B
45 0 0055 JEQ WLOOP 332FFA
46 0 0058 LDCH BUFFER,X 53A016
47 0 005B WD =X'05' DF2012
48 0 005E TIXR T B850
49 0 0060 JLT WLOOP 3B2FEF
50 0 0063 RSUB 4F0000
51 0 0066 USE CDATA
52 1 0007 LTORG
53 1 0007 =C'454f46'454f46
54 1 000A =X'05'05
55 1 000B END FIRST
56
```



```

HCOPY__ 000000 001071
T000000 1E 172063 4B2021 032060 290000 332006 4B203B 3F2FEE 032055 0F2056 010003
T00001E 09 0F2048 4B2029 3E203F
T000027 1D B410 B400 B440 75101000 E32038 332FFA DB2032 A004 332008 57A02F B850
T000044 09 3B2FEA 13201F 4F0000
T00006C 01 F1
T00004D 19 B410 772017 E3201B 332FFA 53A016 DF2012 B850 3B2FEF 4F0000
T00006D 04 454f46 05
E000000

```

This is the output file that is generated. It can be verified here:

```

HCOPY 000000001071
T0000001E1720634B20210320602900003320064B203B3F2FEE0320550F2056010003
T00001E090F20484B20293E203F
T0000271DB410B400B44075101000E32038332FFADB2032A00433200857A02FB850
T000044093B2FEA13201F4F0000
T00006C01F1
T00004D19B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000
T00006D04454F4605
E000000

```

Figure 2.13 Object program corresponding to Fig. 2.11.

Sample input 2:

```

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

Generated output:

```
HSUM__ 000000 002F03
T000000 1D 050000 010000 69101790 1BA013 1BC000 2F200A 3B2FF4 0F102F00 4F0000
M00000705
M00001705
E000000
```

This is an intermediate file

```
-----
0 0000 FIRST LDX #0
0 0003 LDA #0
0 0006 +LDB #TABLE2
0 000A BASE TABLE2
0 000A LOOP ADD TABLE,X
0 000D ADD TABLE2,X
0 0010 TIX COUNT
0 0013 JLT LOOP
0 0016 +STA TOTAL
0 001A RSUB
0 001D COUNT RESW 1
0 0020 TABLE RESW 2000
0 1790 TABLE2 RESW 2000
0 2F00 TOTAL RESW 1
0 2F03 END FIRST
-----
```

SYMBOL TABLE

```
COUNT 001D
FIRST 0000
LOOP 000A
TABLE 0020
TABLE2 1790
TOTAL 2F00
```

Test Program 3 with error:

SUM START 0

FIRST LDX #0

LDA #0

+LDB #TABLE2

```
BASE TABLE2
LOOP ADD TABLE,X
ADD TABLE,X
TIX COUNT
JLT LOOP
+STA TOTAL
RSUB
COUNT RESW 1
TABLE RESW 2000
TABLE RESW 2000
TOTAL RESW 1
END FIRST
```

```
Assembler > ≡ error.txt
1  SYMBOL ALREADY DEFINED, YOU CANNOT DUPLICATE SYMBOL: TABLE
2
```

If we try to define TABLE twice the error is generated in error.txt

Instructions:

Please run the assembler.cpp file after putting the necessary input assembly code in input.txt file present in the same working directory.

```
PS C:\Users\Asus\OneDrive\Desktop\csn252\Assembler> g++ assembler.cpp -o assembler
PS C:\Users\Asus\OneDrive\Desktop\csn252\Assembler> ./assembler
PS C:\Users\Asus\OneDrive\Desktop\csn252\Assembler> |
```

Conclusion:

Coding of expressions was particularly tough for me. However coding the assembler and implementing these features helped me to clarify my concepts to a great extent.

Overall it was a great experience completing this project.