Implementation of SIC/XE Two Pass Assembler

Akshay kumar Gyara, Sheshank Makkapati, Kaushik kumar Kamala, Shashank Reddy Gopireddy
Computer Engineering Department
San José State University (SJSU)
San José, CA, USA
Email: {akshaykumar.gyara, sheshank.makkapati, koushikkumar.kamal, shashankreddy.gopireddy}@sjsu.edu

***Abstract*—Simplified Instructional Computer Extra Equipment or Extra Expensive (SIC- XE) that is an extended version of SIC. It is a hypothetical computer that includes the hardware features most often found on real machines. In this project, we are going to implement a SIC-XE two-pass assembler using C and C++ language. The input to the assembler is an assembly code with a minimum of 50 lines code and we will be generating SYMTAB, Intermediate file, Object code, and Object Program as output files. We will be adding some enhancements to the assembler by adding error handling. This includes errors such as unable to read or open a file, invalid OPCODE, and duplication Labels already present in the SYMTAB. Finally, we will be comparing both the assembler on basic of time complexity, space complexity, and data structure used.**

***Index Terms*—SIC/XE, processor, assembler, linker, simulator**

I. INTRODUCTION

SIC/XE stands for Simplified Instructional computer/ extra equipment which is a upgraded version of the Simplified Instructional computer. All the instructions that are supported by SIC are supported by SIC/XE. The characteristics of the SIC/XE architecture are:
Memory: The memory is same as of that in the SIC architecture. But the maximum memory in the SIC architecture only supports 32,768 bytes, the maximum memory for the SIC/XE architecture supports 1 Mb.
Registers : There are 4 registers in the SIC/XE architecture. They are:
B: Base register
S: General purpose register
T : General purpose register
F : Floating point register
Data formats; The data formats supported by the SIC/Xe assembler are integers,alphabets,negative numbers and also floating point numbers which is not supported by SIC architecture.
Assembler is used to convert the assembly level language to machine level language. Assembly level language is the language understood by the humans and the machine level language is only understood by the computer. In the SIC/XE assembler the SIC/XE assembly code is converted to machine level code. The steps involved in the process are:
1) Take the program and assigning address to all the instructions
2) Object code generation using address and value of opcodes.
3) Object program generation.
The object program consists of the header record, text record and the end records. The addressing modes supported by SIC/XE assembler are:
a) Indexed Addressing mode: This type of addressing mode is represented by using a value X.
b) Direct Addressing mode : This type of addressing mode doesn't have the value X in it.

II. ASSEMBLER ARCHITECTURE

In this section, we will be discussing about the architecture of the two pass assembler.

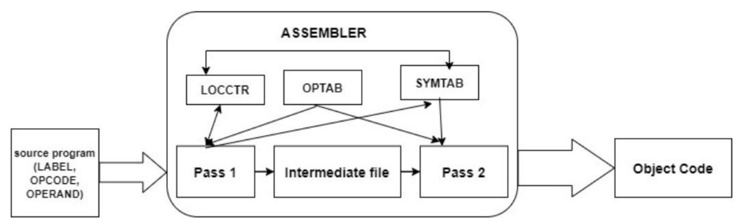


Fig. 1. Assembler Architecture.

In two pass assembler, there are two passes. In pass 1 assembler, we are converting mnemonic operation codes to their equivalent machine languages. Some examples of the mnemonic operation codes are MOVE R, SUB etc. Next convert symbolic operands to machine address i.e we are assigning address to the symbols which are called symbolic operands. Then build machine instructions in a proper format such that we can easily load it into the loader. Next we convert the data constants to internal machine representation. Data constants are formed by assigning address to the literals. Finally, writing object program, and the assembly listing which converts the assembly program to machine program.

As shown in fig 1 the source program , OPTAB are passed as inputs to pass 1 which is used to generate intermediate file and is further passed as input to pass2. We also pass SYMTAB generated by pass 1, which helps in generating the object code.

1. LOCCTR(Location Counter): Helps in the assigning the addresses and is initialized to the beginning address specified in the START statement. After each statement is processed by pass 1,the length of assembled instruction is added to LOCCTR

2. OPTAB(OPCODE Table): Consists of OPCODES, Machine Language Equivalents, length of the instruction and it's format. During Pass1, the Opcodes are validated and during Pass2 the instruction format is determined.
3. SYMTAB: Consists of Label Name, Label Address, Flags and Data Type or length. During Pass1 it stores label name and assigned address from SYMTAB and during Pass2 symbols used as operands are looked up in SYMTAB. A dynamic Hash Table is used for efficient insertion and retrieval.
4. INTERMEDIATE FILE: It contains each source statement with its assigned address, error indicators etc It is used as input to pass2.

Functions of Pass 1:
1) We separate the literals,symbols and mnemonic fields.
2) Next we build the symbol table and the literal table with proper data structures.
3) We do the location counter processing(LC) for every instruction.
4) Finally we will be generating a intermediate file for the pass 1 assembler. .

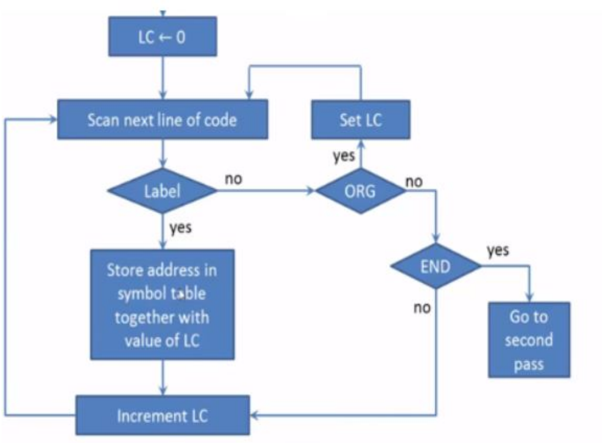


Fig. 2. Pass1 Flowchart

Functions of Pass2:
1) With the help of intermediate file,symbol table and literal table we will be generating the object code

Fig. 3. Pass2 Flowchart

III. Implementation using C++

The C++ implementation uses several data structures to form the OPTAB, SYMTAB and also the literal table.

Data Structures : The data structures used for this implementation are mainly the maps in C++. The map data structure in C++ is basically a kind of Hash map which has key value pairs. So, this data structure suits our functionality to implement the Two pass assembler. All the data structures are implemented in a file named 'tables.cpp' where the opcode table, symbol table. register table and the literal table are formed. The labels, symbols, literals and the registers are stored as structure data structures.

A. Pass 1 Implementation :

The first step of the pass 1 algorithm checks if the first line of the input is START, if the condition satisfies then it saves the operand as starting address and then the location counter is assigned to the starting address and the formed line is written to the intermediate file. If the condition is not satisfied then the LOCCTR is initialised to 0. Then the while loop is ran for all the lines in the assembly code except for the commented lines. In this loop, we check if there is a symbol in the label field, then we search the symtab for the label which we found as a part of the instruction. If the label is found in the symtab, we set the error flag. If the label is not found, we insert the label and the location counter into symtab.
Then we start searching for opcode in the optab. If we found it, we add the instruction length to the location counter. Now, we compare the opcode which is found in the optab. If the optab is equal to the 'WORD', then we add the instruction length to the location counter. If the optab value is 'RESW', then we add 3 times of the number of operands to the location counter(LOCCTR).
If the value of the opcode is equal to the 'RESB' the we just add the operand number to the LOCCTR. If the value is 'BYTE', we add the length of constant in bytes to the LOCCTR. We write this to the intermediate file continuously until we reach the end of the file. Finally, we save the starting address of the location counter as program length.

B. Code implementation for pass 1:

Firstly the Filename is considered to be a string, such that it can manipulated and separate files such as the intermediate files and the error files can be formed. Then we'll start traversing through the entire file using a separate function created named pass1. We first check is the input file format is given correctly otherwise we output an error that the file cannot be opened. Then we start checking if the value of opcode is "START". If so we do the operation as described in the algorithm and then we write to the intermediate file. Similarly as described in the algorithm, each comparison operation is done and the suitable operation is done. At the end of the Pass 1, the intermediate file is complete with the necessary addresses assigned.

C. Pass 2 Implementation :

The intermediate file is given as the input for the pass 2 function which is implemented. First we read the intermediate file and check if the value of the opcode is 'START'. If it is START then, we start writing to the listing file which consists of the object code. After the writing to the listing file has started, the next line of the intermediate file is read. Then we start writing the object program to a file. The object program consists of the header record, text record and the end record. Firstly , the header record to the object program is written and then the first text record is initialized. Now, we start traversing through the file and we search for the opcode in the OPTAB. If the operand is found, we search if there is a symbol in the operand field, if there is a symbol, then we search the SYMTAB for the operand.If it is found, then we store the symbol value as the operand address. Then we check if the value of OPCODE is BYTE or WORD, if it is satisfied we convert the constant to object code. Also we check if the object code doesn't fit the text record, then we write text record to the object program and initialize new text record. After this is done, we add the object code to the text record. We write to the listing file and move on to the next line . After we reach to the end of file, we finally write last text record to the object program and end record to the object program.

IV. Implementation using C

We have implemented a two pass SIC-XE assembler using C language. For generating SYMTAB we have used hash tables data structures in which data can be stored as array format and each data value has its own unique index value. We have chosen hash tables over other data structures as data access is very fast. Using Hash tables data insertion and search operation are fast irrespective of data size. We use a "hash" function which is used to map symbols to corresponding integers ranging from 0 - K-1 . Its function is to add the ASCII codes of characters in symbols by ignoring overflow and taking modulo K of the result. And finally symbols are stored using a table that consists of K buckets ranged from 0 to k-1.

A. Code implementation for pass 1:

In pass 1 we have added some improvements to the existing SIC-XE assembler by adding error handling features.This includes errors such as unable to read or open a file, invalid OPCODE, and duplication Labels already present in the SYMTAB. We use a function named CheckError() which takes file as an input and using IF condition we will do string comparison if Opcode is word or byte or RESW or RESB . If it is none of them them we will catch this error and write it to error file.
We use increment-LOCCTR() to increment the LC (location counter) value. This function takes Opcode, Operand and Location value as input and verifies if the operand is a symbol, Word, Byte,, RESW or RESB and increments the LC value according . If the operand is none of the above we will increment the LC value by 3.
Finally in the main function we will use file to open the input file that as assembly code in read mode. Simultaneously we also open SYMTAB and Intermediate file on write mode. Then we parse through the input file line by line and call the increment-LOCCTR() function and also copy the labels to SYMTAB file. And at the end of Pass1 execution we have our SYMTAB and Intermediate files ready.

B. Code implementation for pass 2:

For pass 2 we will use SYMTAB and Intermediate files generated by Pass1 program as inputs. We use a function get-mnemonic() to search if there are any opcode in the OPTAB. The main task of this function is to convert char variables to its corresponding ASCII values.
We have another function search-symtab() that is used to parse the SYMTAB file for the operands. If an operand is found we stores the symbol value as operand address.
Finally we have main function and here we implement error handling for invalid OPCODE, and duplication Labels already present in the SYMTAB. We call the get-mnemonic() by passing Opcode file as input . We assign this function to a variable to keep a record of error. Similarly we search-symtab() by passing SYMTAB as a parameter. We assign a constant variable to this function to keep record of error. This is used to print label reference type errors.
By the end of Pass 1 and Pass 2 we have SYMTAB , Intermediate File , Object code and Object Program files as output.

V. Results

We have used assembly language as input which is 53 lines of code. And the Optab have 59 Operands. Input Assembly Code:

A. Code implementation using C for pass 1 and Pass2:

```
. TEST PROGRAM
COPY    START   0 Comment here
FIRST   STL     RETADR  Comment here
CLOOP   JSUB    RDREC  Comment here
        LDA     LENGTH  Comment here
        COMP    #0  Comment here
        JEQ     ENDFIL  Comment here
        JSUB    WRREC  Comment here
        J       CLOOP Comment here
ENDFIL  LDA     =C'EOF' Comment here
        STA     BUFFER Comment here
        LDA     #3  Comment here
        STA     LENGTH  Comment here
        JSUB    WRREC  Comment here
        J       @RETADR Comment here
        USE     CDATA
RETADR  RESW    1 Comment here
LENGTH  RESW    1 Comment here
        USE     CBLKS
BUFFER  RESB    4096 Comment here
BUFEND  EQU     * Comment here to test * literal
MAXLEN  EQU     BUFEND-BUFFER
.
.   RDREC SUBROUTINE
.
        USE     DEFAULT
RDREC   CLEAR   X Comment here
        CLEAR   A Comment here
        CLEAR   S Comment here
        +LDT    #MAXLEN Comment here
RLOOP   TD      INPUT Comment here
        JEQ     RLOOP Comment here
        RD      INPUT Comment here
        COMPR   A,S Comment here
        JEQ     EXIT Comment here
        STCH    BUFFER,X Comment here
        TIXR    T Comment here
        JLT     RLOOP Comment here
EXIT    STX     LENGTH Comment here
        RSUB    Comment here
        USE     CDATA
INPUT   BYTE    X'F1' Comment here
.
.   WRREC SUBROUTINE
.
        USE     DEFAULT
WRREC   CLEAR   X Comment here
        LDT     LENGTH Comment here
WLOOP   TD      =X'05' Comment here
        JEQ     WLOOP Comment here
        LDCH    BUFFER,X Comment here
        WD      =X'05' Comment here
        TIXR    T Comment here
        JLT     WLOOP Comment here
        RSUB        Comment here
        USE     CDATA
        LTORG
        END     FIRST Comment here
```

Fig. 4. Input assembly code

B. Code implementation using C++

```
[Running] cd "e:\Assembler-master_runwell\Assembler-master\" && gcc assembler_pass1.c
FIRST   1000
CLOOP   1003
ENDFIL  1015
EOF 102A
THREE   102D
ZERO    1030
RETADR  1033
LENGTH  1036
BUFFER  1039
RDREC   2039
RLOOP   203F
EXIT    2054
INPUT   205A
MAXLEN  205C
WRREC   205F
WLOOP   2062
OUTPUT  2074

[Done] exited with code=0 in 2.239 seconds
```

Fig. 5. Output of Pass1 without Error Handling

```
[Running] cd "e:\Assembler-master_runwell\Assembler-master\" && gcc assembler_pass1.c
There is a error in the mnemonic: +STA.
Aborting this line..

FIRST   1000
CLOOP   1003
ENDFIL  1015
EOF 1027
THREE   102A
ZERO    102D
RETADR  1030
LENGTH  1033
BUFFER  1036
RDREC   2036
RLOOP   203C
RLOOP   2051
EXIT    2066
INPUT   206C
MAXLEN  206E
WRREC   2071
WLOOP   2074
OUTPUT  2086
```

Fig. 6. Output of Pass1 without Error Handling

The SIC/XE Two pass assembler was successfully developed using two approaches, one using C++ and the other using C. The C++ implementation clearly has the best performance when compared to the implementation using C. The C++ implementation uses the hash map data structure which has

```
[Running] cd "e:\Assembler-master_runwell\Assembler-master\" && gcc assembler_pass2.c -o assembler_pass2
Incorrect Mnemonic error.
FIRST   STL RETADR      141033
CLOOP   JSUB    RDREC       482039
-   LDA LENGTH      01036
-   COMP    ZERO        281030
-   JEQ ENDFIL      301015
-   JSUB    WRREC       48205F
-   J   CLOOP       3c1003
ENDFIL  LDA EOF     0102A
-   STA BUFFER      c1039
-   LDA THREE       0102D
-   STA LENGTH      c1036
-   JSUB    WRREC       48205F
-   LDL RETADR      81033
-   RSUB    -       4c0000
EOF BYTE    C'EOF'      454F46
THREE   WORD    3       0000003
ZERO    WORD    0       0000000
RETADR  RESW    1
LENGTH  RESW    1
BUFFER  RESB    4096
RDREC   LDX ZERO        41030
-   LDA ZERO        01030
RLOOP   TD  INPUT       e0205A
-   JEQ RLOOP       30203F
-   RD  INPUT       d8205A
-   COMP    ZERO        281030
-   JEQ EXIT        302054
-   TIX MAXLEN      2c205C
-   JLT RLOOP       38203F
EXIT    STX LENGTH      101036
```

Fig. 7. Output of Pass2 without Error Handling

```
[Running] cd "e:\Assembler-master_runwell\Assembler-master\" && gcc assembler_pass2.c -
Incorrect Mnemonic error.
FIRST   STL RETADR      141030
CLOOP   JSUB    RDREC       482036
-   LDA LENGTH      01033
-   COMP    ZERO        28102D
-   JEQ ENDFIL      301015
-   JSUB    WRREC       482071
Label referencing error.
ENDFIL  LDA EOF     01027
-   STA BUFFER      c1036
-   LDA THREE       0102A
Incorrect Mnemonic error.
-   JSUB    WRREC       482071
-   LDL RETADR      81030
-   RSUB    -       4c0000
EOF BYTE    C'EOF'      454F46
THREE   WORD    3       0000003
ZERO    WORD    0       0000000
RETADR  RESW    1
LENGTH  RESW    1
BUFFER  RESB    4096
RDREC   LDX ZERO        4102D
```

Fig. 8. Output of Pass2 without Error Handling

```
test_inputs > intermediate_input.asm
Line    Address Label   OPCODE  OPERAND Comment
5   . TEST PROGRAM
10  00000   0   COPY    START   0   Comment here
15  00000   0   FIRST   STL RETADR  Comment here
20  00003   0   CLOOP   JSUB    RDREC   Comment here
25  00006   0       LDA LENGTH  Comment here
30  00009   0       COMP    #0  Comment here
35  0000C   0       JEQ ENDFIL  Comment here
40  0000F   0       JSUB    WRREC   Comment here
45  00012   0       J   CLOOP   Comment here
50  00015   0   ENDFIL  LDA =C'EOF' Comment here
55  00018   0       STA BUFFER  Comment here
60  0001B   0       LDA #3  Comment here
65  0001E   0       STA LENGTH  Comment here
70  00021   0       JSUB    WRREC   Comment here
75  00024   0       J   @RETADR Comment here
80  00000   1       USE CDATA
85  00000   1   RETADR  RESW    1   Comment here
90  00003   1   LENGTH  RESW    1   Comment here
95  00000   2       USE CBLKS
100 00000   2   BUFFER  RESB    4096    Comment here
105 01000   2   BUFEND  EQU *   Comment here to test * literal
110 01000       MAXLEN  EQU BUFEND-BUFFER
```

Fig. 9. Intermediate File

```
test_inputs > object_input.asm
H^COPY  ^000000^001071
T^000000^1E^1720634B2021032060290000332006 4B203B3F2FEE0320550F2056010003
T^00001E^09^0F20484B20293E203F
T^000027^1D^B410B400B44075101000E32038332FFADB2032A00433200857A02FB850
T^000044^09^3B2FEA13201F4F0000
T^00006C^01^F1
T^00004D^19^B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000
T^00006D^04^454F4605
E^000000
```

Fig. 10. Object Program

```
************PASS1************
Line: 25 : Invalid OPCODE. Found LDAA
Line: 45 : Invalid OPCODE. Found JB


************PASS2************
Line 15 : Symbol doesn't exists. Found RETDR
Line 195 : Symbol doesn't exists. Found LENGTH+
```

Fig. 11. Error Handling

Fig. 12. Performance comparision between C and C++ code

the time complexity of O(1).The intermediate file, Symbol table, Object code and the object program are successfully generated using our implementation. The C++ implementation also includes proper error handling, which writes to an error file if there is any discrepancy in the input assembly file. The below graph show the comparison between C and C++ for 3 types of assembly codes : small , medium and large. We can see that C++ performs better than C in terms of execution time.

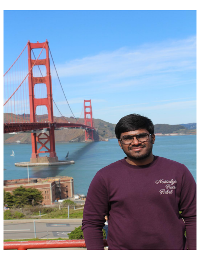
REFERENCES

1. System Software an introduction to systems programming by Leland beck third edition .

FirstName LastName : Akshay kumar Gyara
SID : 013836277
Ph no : 408-646-7191

FirstName LastName : Sheshank Makkapati
SID : 013824148
Ph no : 669-220-9961

FirstName LastName : Kaushik kumar Kamala
SID : 013766571
Ph no : 408-210-2991



FirstName LastName : Shashank Reddy Gopireddy
SID : 013853931
Ph no : 408-480-8831