Two pass Assembler for SIC/XE

Project Domain – System Software Internals

Submitted by,

CT - S Batch (2014-2018)

Computer Technology Department

Madras Institute Of Technology

Anna University

DOCUMENTATION

ABSTRACT:

The project is to design an assembler to optimize the present functions of the SIC/XE and also handle various real time requirements. The newly designed assembler is the modification of the current one which provides extra equipments and *extra optimization*. The various functions included individually produce a performance gain or add an extra feature to the assembler.

This *proposal eliminates the bottlenecks* in the present assembler.The project is also proposed to be an *open source*. The methods implemented to reduce the bottlenecks is clearly explained in this document.

Two pass Assembler for SIC/XE

Table of Contents

INTRODUCTION:

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term assembly language. The programmer can write a program using a sequence of these assembler instructions. This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started.

The assembler program takes each program statement in the source program and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length). The output of the assembler program is called the object code or object program relative to the input source program. The sequence of 0's and 1's that constitute the object program is sometimes called machine code. The object program can then be run (or executed) whenever desired.

OVERVIEW:

a) CURRENT SYSTEM:

The current assembler design converts mnemonic operation codes to their machine language equivalents, converts symbolic operands to their equivalent machine addresses. It builds the machine instructions in the proper format, converts the data constants to internal machine representations and writes the object program and the assembly listing.

In the current two pass assembler, the Pass 1 assigns addresses to all statements in the program, saves the values assigned to all labels for use in the Pass 2 and also performs some processing of the assembler directives. Pass 2 assembles instructions, generate data values defined by BYTE, WORD and also performs processing of assembler directives not done in Pass1. It finally writes the object program and assembly listing.

b) PROPOSED WORK:

This newly proposed assembler is designed to *handle the features which are not supported by current assembler design*. The features handled are listed below:

- **Literals**
- **Duplicate Labels**
- **Multiple Control Sections**
- **Different Addressing Modes**
- **Program Relocation**

HANDLING LITERALS:

A literal is a constant value in an assembly instruction directly as an operand without a need of a label. It is denoted by '='. With '=', the assembler generates the specified value as a constant in another location of the memory and include the address of the constant as TA in the machine instruction. In the current assembler design, this literal handling is not supported. Hence we have designed this assembler to handle this feature.

DUPLICATE LABELS:

In case of a duplicate symbol, the current assembler design would give an error message only for the second definition. The newly designed assembler would give error messages for all the definitions and references of a doubly defined symbol.

MULTIPLE CONTROL SECTIONS:

Control sections are segments that are translated into independent object program units. In case of multiple control sections we use **EXTDEF** and **EXTREF** type. Any reference to a symbol not defined within a control section is an external reference (EXTREF). The external definition (EXTDEF) means the symbol defined in this control section is to be made visible outside it.

DIFFERENT ADDRESSING MODES:

In the newly proposed design, all four formats of addressing modes are supported. Format 1 and Format doesn't use memory. Format 3 supports immediate, indirect, program relative and base relative addressing modes. Format 4 supports only immediate and indirect addressing modes.

PROGRAM RELOCATION:

Relocation is the process of assigning load addresses to various parts of a program and adjusting the code and data in the program to reflect the assigned addresses. A modification record is used to describe each part of the object code that must be changed when the program is relocated.

REQUIREMENTS:

DATA STRUCTURES USED:

- **Operand table(OPTAB)**

Name

- **Symbol table(SYMTAB)**

Name	Address	Count

- **Literal table(LITTAB)**

Name	Address	Count

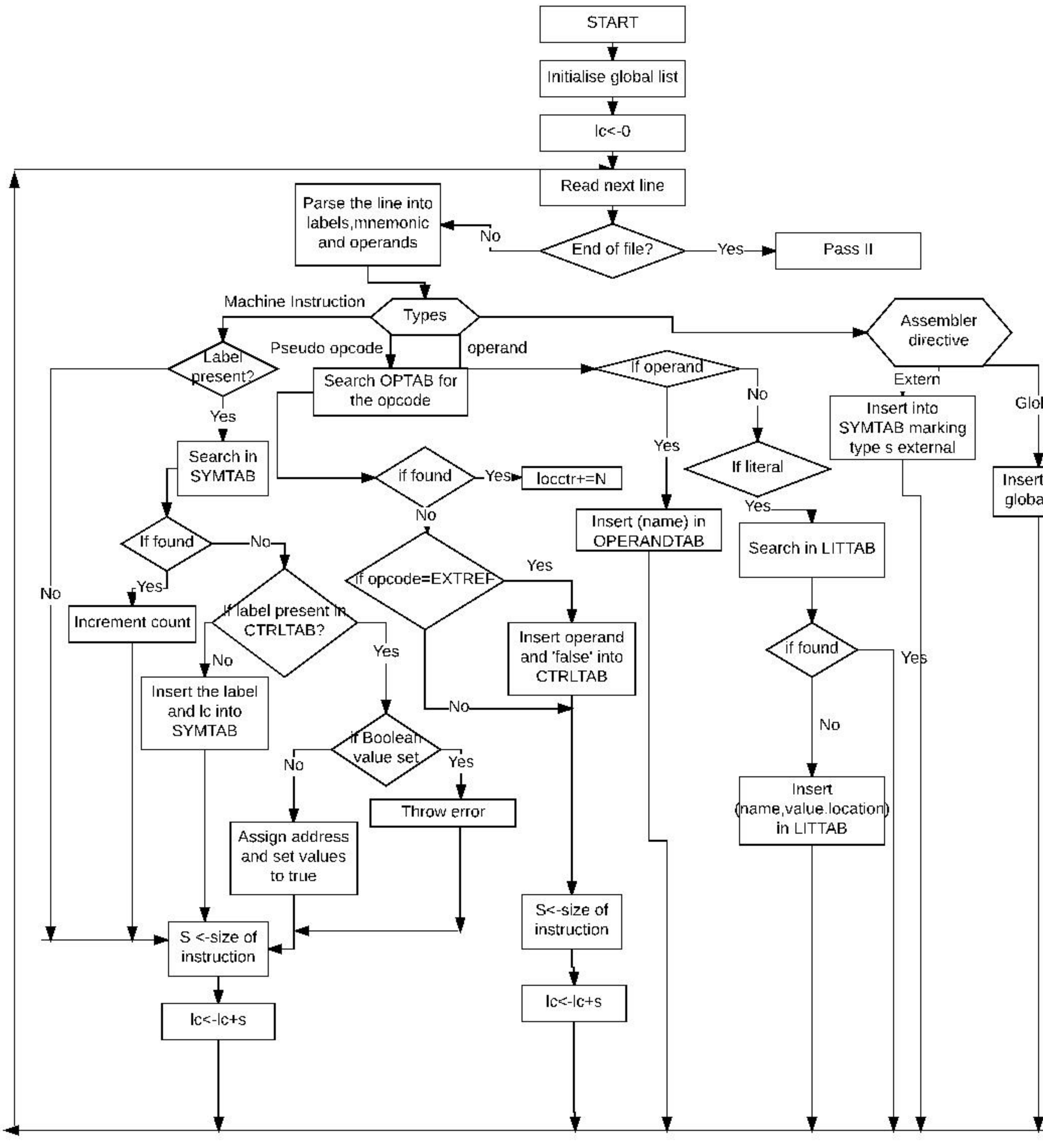
- **Control table(CTRLTAB)**

Name	Boolean	Address

- **Location counter(LOCCTR)**

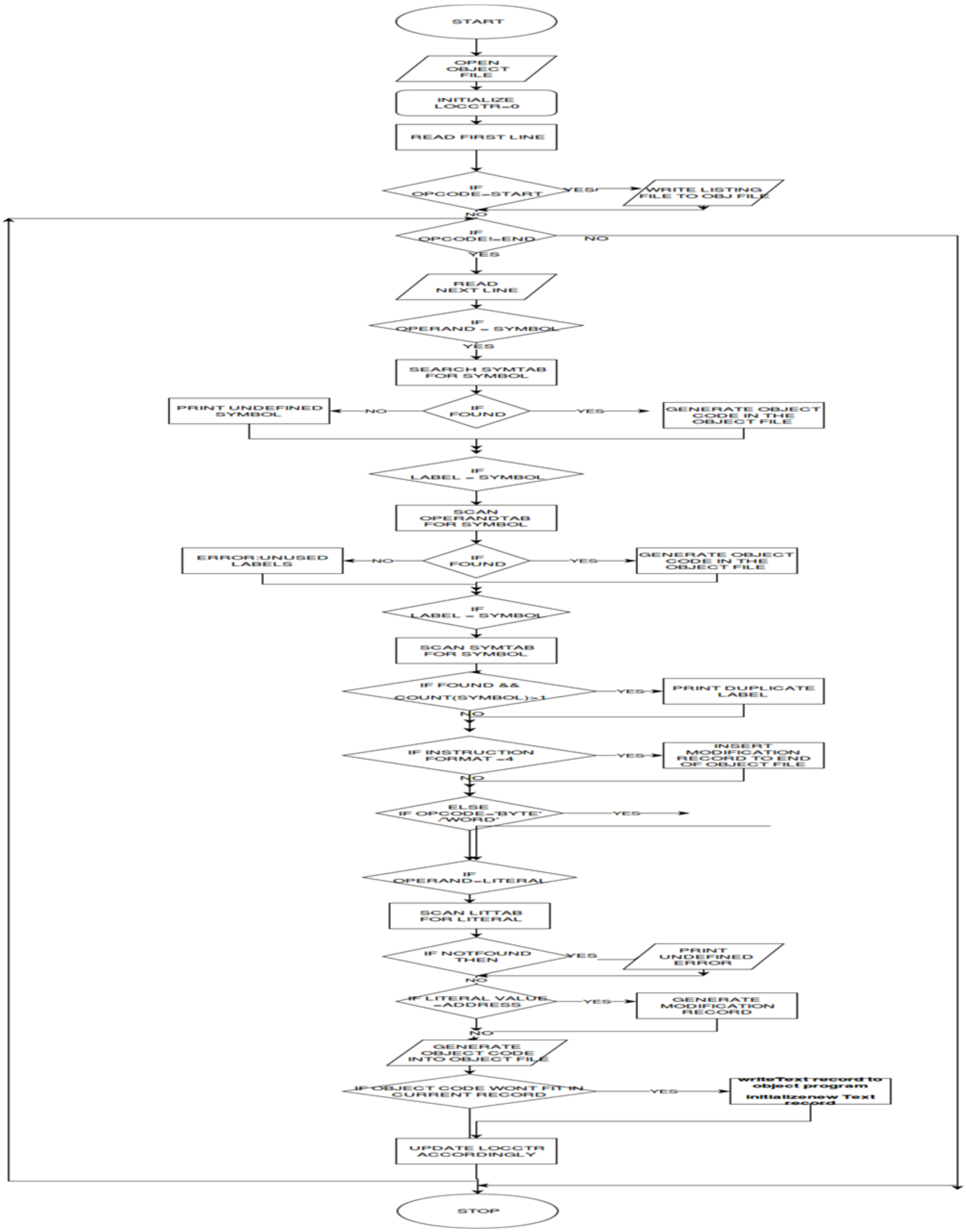
DESIGN

PASS 1



START
Initialise global list
lc<-0
Read next line
End of file?
Yes
Pass II
No
Parse the line into labels,mnemonic and operands
Types
Machine Instruction
Pseudo opcode
operand
Assembler directive
Label present?
No
Yes
Search OPTAB for the opcode
If operand
Yes
No
Extern
Global
Insert into SYMTAB marking type s external
Insert into global list
Search in SYMTAB
if found
Yes
No
locctr+=N
If literal
Yes
Insert (name) in OPERANDTAB
Search in LITTAB
If found
No
Yes
if opcode=EXTREF
Yes
No
Increment count
If label present in CTRLTAB?
No
Yes
Insert operand and 'false' into CTRLTAB
if found
Yes
No
Insert the label and lc into SYMTAB
If Boolean value set
No
Yes
Insert (name,value.location) in LITTAB
Throw error
Assign address and set values to true
S <-size of instruction
S<-size of instruction
lc<-lc+s
lc<-lc+s

PASS 2


START
OPEN OBJECT FILE
INITIALIZE LOCCTR=0
READ FIRST LINE
IF OPCODE=START
YES
WRITE LISTING FILE TO OBJ FILE
NO
IF OPCODE!=END
NO
YES
READ NEXT LINE
IF OPERAND = SYMBOL
YES
SEARCH SYMTAB FOR SYMBOL
IF FOUND
NO
PRINT UNDEFINED SYMBOL
YES
GENERATE OBJECT CODE IN THE OBJECT FILE
IF LABEL = SYMBOL
SCAN OPERANDTAB FOR SYMBOL
IF FOUND
NO
ERROR:UNUSED LABELS
YES
GENERATE OBJECT CODE IN THE OBJECT FILE
IF LABEL = SYMBOL
SCAN SYMTAB FOR SYMBOL
IF FOUND && COUNT(SYMBOL)>1
YES
PRINT DUPLICATE LABEL
NO
IF INSTRUCTION FORMAT =4
YES
INSERT MODIFICATION RECORD TO END OF OBJECT FILE
NO
ELSE IF OPCODE='BYTE' /'WORD'
YES
IF OPERAND=LITERAL
SCAN LITTAB FOR LITERAL
IF NOTFOUND THEN
YES
PRINT UNDEFINED ERROR
NO
IF LITERAL VALUE =ADDRESS
YES
GENERATE MODIFICATION RECORD
NO
GENERATE OBJECT CODE INTO OBJECT FILE
IF OBJECT CODE WONT FIT IN CURRENT RECORD
YES
writeText record to object program
initializenew Text record
UPDATE LOCCTR ACCORDINGLY
STOP

IMPLEMENTATION:

PASS 1:

Algorithm:

```
Begin

        if starting address is given
                LOCCTR = starting address;
         else
                LOCCTR = 0;
         while OPCODE != END do              ;; or EOF
           begin
                read a line from the code

        parse the line into label, mnemonic and operands

/*   The SYMTAB uses an additional field known as counter to track the number
of  instances of the label */
                if there is a label
                        if this label is in SYMTAB, then increment count for the
label in SYMTAB
                        else if  label is present in CTRLTAB

        Begin

                    If Boolean value is false

                        Assign the address and make Boolean value as true .

                        Else

                            Throw error of a control variable at two places

                        End(if)

            End (else if)

            else insert (label, LOCCTR,1) into SYMTAB
            end (if)
```

```
        search OPTAB for the op code for the corresponding mnemonic
        if found
        LOCCTR += N          ;; N is the length of this instruction (4 for
        MIPS)
        else if this is an assembly directive
                update LOCCTR as directed
```

else if OPCODE=EXTREF then

begin

while for each variable

insert operand and 'false' into CTRLTAB

end(while)

end (else if)

else if literal

search in LITTAB

if found

insert (label, LOCCTR,1) into LITTAB

else if this label is in LITTAB ,then increment count for the label in LITTAB

end(if)

else

Throw error

// OPERANDTAB is used to store operands used in the instruction for latter use

if there are operands in the parsed string

then insert the operands into the OPERANDTAB

write line to intermediate file

end

program size = LOCCTR - starting address;

end

Explanation:

If Label is a variable then it is inserted into *Symbol table*. If Operand is variable or value, then it is inserted into *Operand table*. If literal, then insert it into *literal table*. If label is a variable then first check in the *Symbol table*. If the variable already exists, increment the count. We use this technique to identify if there is **duplicate label** present.

If the variable does not exist in the *Symbol table*, check in the *Control table*. We use this technique for **forward reference**. If *Boolean* is given as false for the corresponding *Label*, then change it to TRUE and update its address. If the Label doesn't exist in *Symbol table* as well as in *Control table*, update the *Symbol table*. If operand is available in *Operand table*, then increment *Location counter*.

If not, the operand is an **Assembler Directive**. In case of operand is an assembler directive, *location counter* changes accordingly. Else if operand is a mnemonic then *location counter* is increased to the size of the instruction.

If operand is **externally referenced**, then add it to *Control table* and change Boolean value to FALSE. Other operands are updated in operand table. Finally write it into **intermediate file.**

PASS 2:

Algorithm:

```
            Open Object file
LOCCTR=0
begin
        read first line
        if OPCODE ='START' then
                write listing line to object file
end {if START}
while OPCODE!=END  do
        begin
        read next line
```

//Undefined Symbols

```
        if OPERAND='symbol ' then
                begin
                scan SYMTAB  for symbol
                if FOUND  then
                        generate object code in the object file
                else
                        print "error:undefined symbol"
                end{symbol}
```

```
//Unused Labels
        if LABEL='symbol' then
                begin
                scan OPERANDTAB for symbol
                if FOUND then
                        generate object code in the object file
                else
                        print "error: unused labels"

//Duplicate Labels
        if LABEL='symbol' then
        begin
                scan SYMTAB for symbol
                if FOUND && count(symbol) >1 then
                        print "error:Duplicate label"
        end

        //Program relocation handling
        if the INSTRUCTION is of EXTENDED format
        begin
                insert modification record from the intermediate file to the end of
Object file
        end {if instruction is of extended format}
        else if OPCODE ='BYTE' or 'WORD' then
                convert constant to object code  and insert into object file
```

```
//Literal Handling
if OPERAND='literal' then
    scan LITTAB for literal
    if FOUND then
        scan OPERATIONTAB for literal
        if NOTFOUND then
            print "error: unused literal"
        end{if NOTFOUND}
    else
        print "error: undefined literal"
    end{if FOUND}
generate object code into object file
if object code will not fit into the current Text record then
begin

write Text record to object program
    initialize new Text record
end{if text record}
end{if literal}
end {while}
```

Explanation:

While **handling undefined symbols,** see the symbol in the *operand column*, scan SYMTAB for that symbol. If it is present generate the object code and put it into the object file. Otherwise print as error.

While **handling unused labels**, see the symbol in the *label column*, scan the OPTAB for that symbol. If it is present generate the object code and put it into the object file. Otherwise print as error.

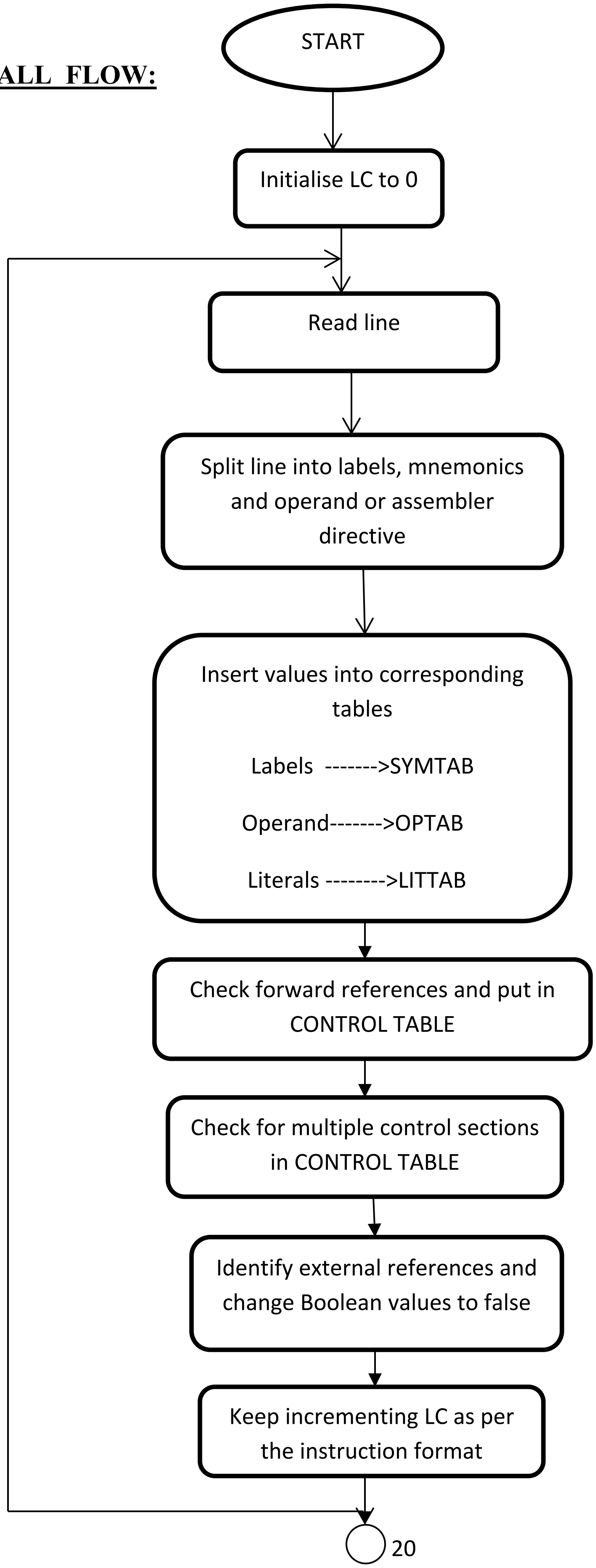
While **handling duplicate labels**, see the symbol in the *label column*, scan the SYMTAB for that symbol. If it is present check the count of that symbol which was generated in PASS 1. If the count is greater that '1' , print error.

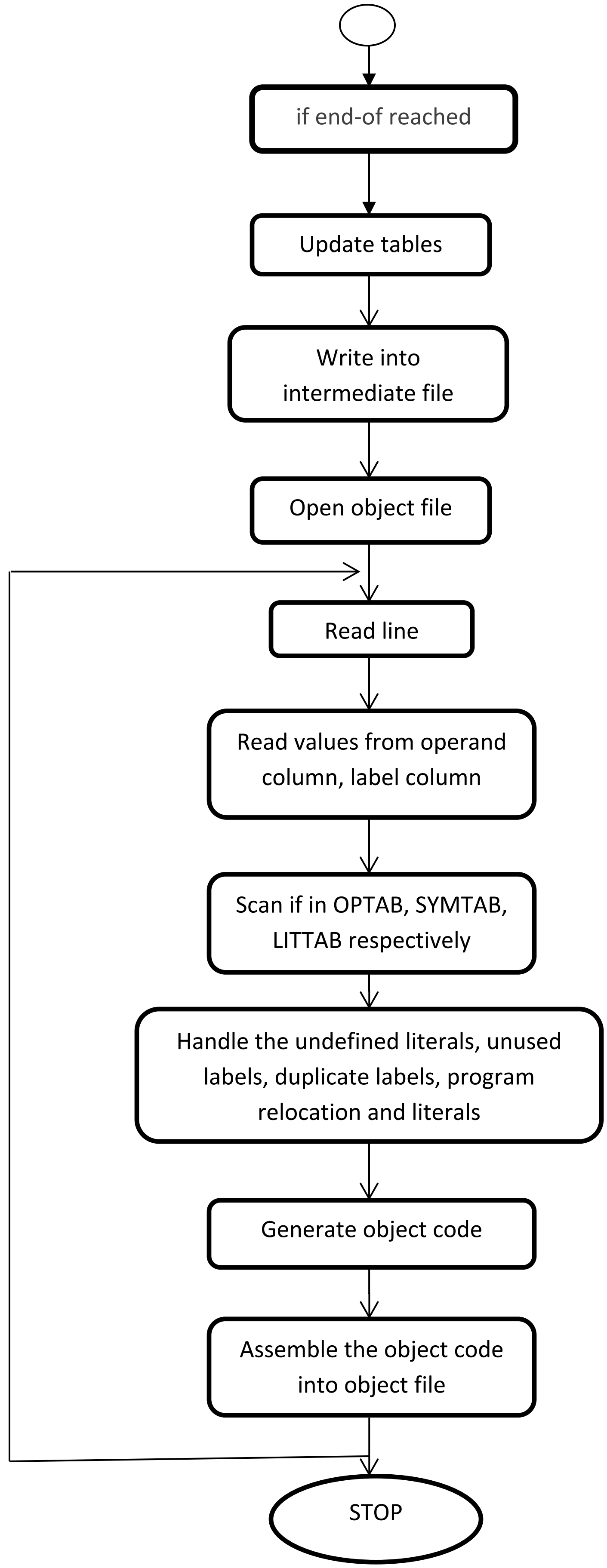
While **handling program relocation,** which occurs only in format 4, add modification record which was already created in PASS 1 at the end of the object code. If it is 'BYTE' or 'WORD' declarations that is assigning constants to that variable, convert it as such to object code

While **handling literals**, if the literal is found at operand column, scan the LITTAB for that literal. If it is not found then print error. If the literal value is an address, then generate the modification record.

Finally , generate the *object code* and insert it into the *object file*. If the object code doesn't fit into the object record, initialize a new record and fill the remaining object code into the text record. Finally write the record into the object file.

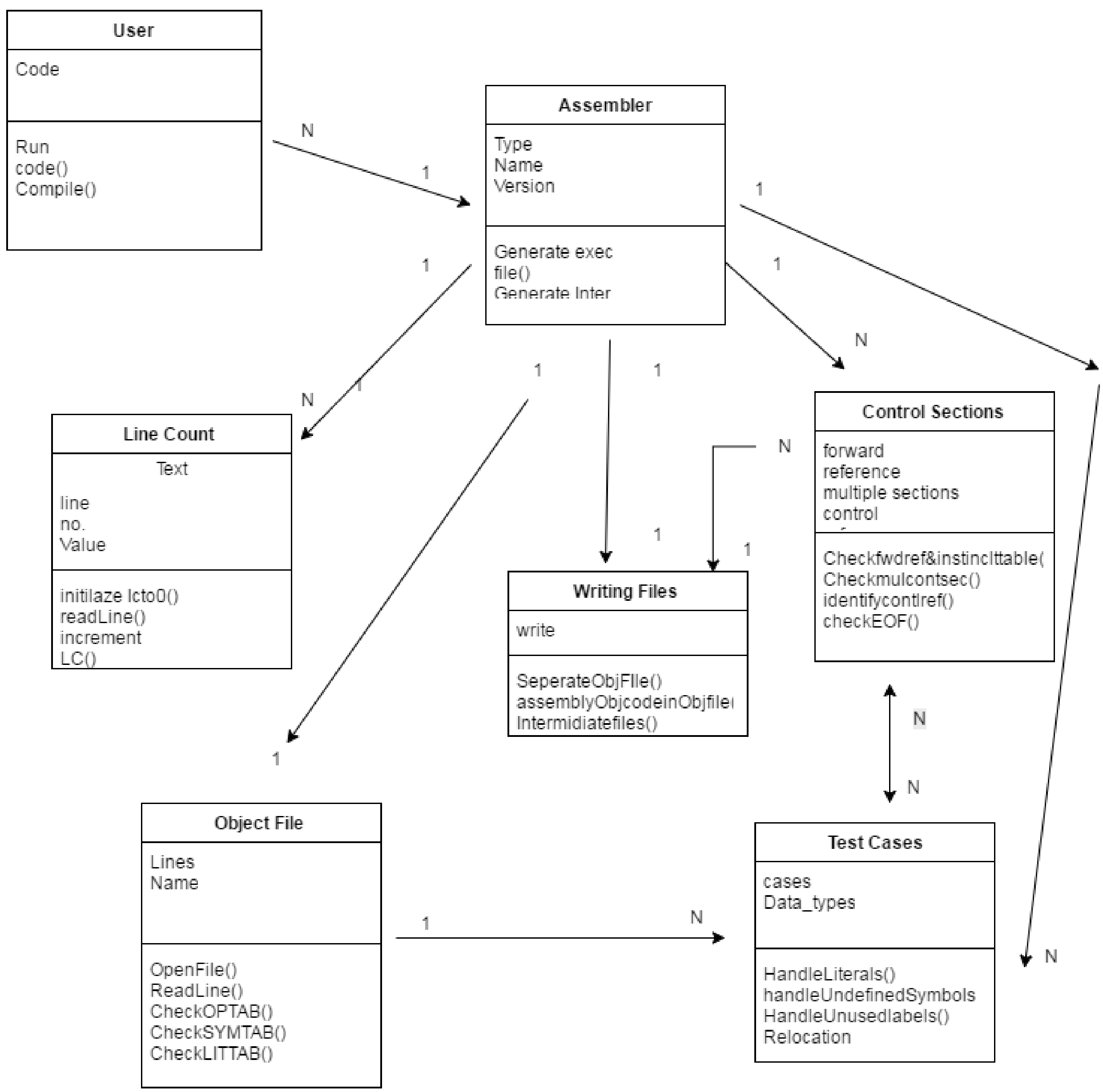
OVERALL FLOW:



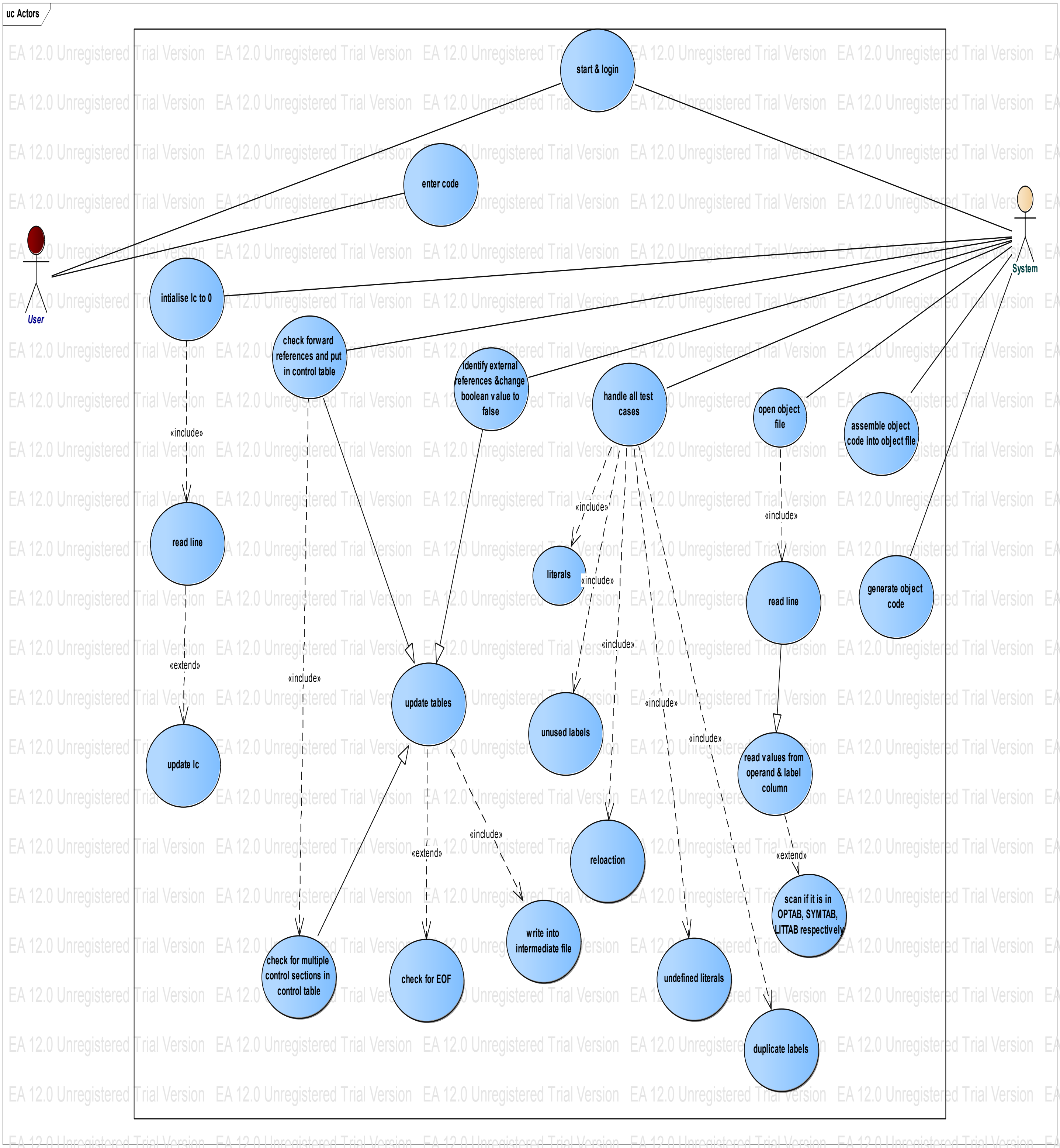

if end-of reached
Update tables
Write into intermediate file
Open object file
Read line
Read values from operand column, label column
Scan if in OPTAB, SYMTAB, LITTAB respectively
Handle the undefined literals, unused labels, duplicate labels, program relocation and literals
Generate object code
Assemble the object code into object file
STOP

Necessary diagram

Class Diagram



Usecase Diagram



TEST CASES

TEST CASE 1: To test the addressing modes and the four formats of the instructions and the literals and the expressions and the EQU.

```
COPY    START  0
FIRST   STL    RETADR
        LDB    #LENGTH
        BASE   LENGTH
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
        LTORG
        * =C'EOF'
RETADR         RESW   135
LENGTH         RESW    1
BUFFER         RESB   4096
BUFEND         EQU     *
MAXLEN         EQU    BUFEND-BUFFER
RDREC   CLEAR  X
        CLEAR  A
        CLEAR  S
        +LDT   #4096
RLOOP   TD     INPUT
        JEQ    RLOOP
        RD     INPUT
        COMPR  A,S
        JEQ    EXIT
        STCH   BUFFER,X
        TIXR   T
        JLT    RLOOP
EXIT    STX    LENGTH
        RSUB
INPUT   BYTE   X'F1'
WRREC          CLEAR   X
        LDT    LENGTH
WLOOP          TD     =X'05'
        JEQ    WLOOP
        LDCH   BUFFER,X
        WD     =X'05'
        TIXR   T
        JLT    WLOOP
        RSUB
        END    FIRST
        * =X'05'
```

NOTE : WE WILL HAVE 3 MODIFICATION RECORDS

EXPECTED OUTPUT:

```
0       COPY    START   0
0       FIRST   STL     RETADR          17202D
3               LDB     #LENGTH 69202D
                BASE    LENGTH
6       CLOOP   +JSUB   RDREC   4B101036
A               LDA     LENGTH          032026
D               COMP    #0      290000
10              JEQ     ENDFIL          332007
13              +JSUB   WRREC           4B10105D
17              J       CLOOP   3F2FEC
1A      ENDFIL          LDA     =C'EOF' 032010
1D              STA     BUFFER          0F2016
20              LDA     #3      010003
23              STA     LENGTH          0F200D
26              +JSUB   WRREC           4B10105D
2A              J       @RETADR 3E2003
2D              LTORG
2D              * =C'EOF'       454F46
30      RETADR          RESW    1
33      LENGTH          RESW    1
36      BUFFER          RESB    4096
1036    BUFEND          EQU     *
1000    MAXLEN          EQU     BUFEND-BUFFER
1036    RDREC   CLEAR   X       B410
1038            CLEAR   A       B400
103A            CLEAR   S       B440
103C            +LDT    #4096   75101000
1040 RLOOP TD INPUT     E32019
1043            JEQ     RLOOP   332FFA
1046            RD      INPUT   DB2013
1049            COMPR   A,S     A004
104B            JEQ     EXIT    332008
104E            STCH    BUFFER,X  57C003
1051            TIXR    T       B850
1053            JLT     RLOOP   3B2FEA
1056    EXIT    STX     LENGTH          134000
1059            RSUB            4F0000
105C    INPUT   BYTE    X'F1'   F139
105D    WRREC           CLEAR   X       B410
105F            LDT     LENGTH          774000
1062    WLOOP           TD      =X'05'  E32011
1065            JEQ     WLOOP           332FFA
1068            LDCH    BUFFER,X        53C003
106B            WD      =X'05'  DF2008
106E            TIXR    T       B850
1070            JLT     WLOOP           3B2FEF
1073            RSUB            4F0000
1076            END     FIRST
1076            *       =X'05'  05
```

TEST CASE 2: To test the BYTE and WORD and RESB and RESW and the addressing modes and the 4 formats of the instructions:

```
COPY    START  0
FIRST   STL    RETADR
        LDB    #LENGTH
        BASE   LENGTH
CLOOP   +JSUB  RDREC
        LDA    LENGTH
        COMP    #0
        JEQ    ENDFIL
        +JSUB  WRREC
        J      CLOOP
ENDFIL         LDA     EOF
        STA    BUFFER
        LDA    #3
        STA    LENGTH
        +JSUB  WRREC
        J      @RETADR
EOF     BYTE   C'EOF'
RETADR         RESW    1
LENGTH         RESW    1
BUFFER         RESB    4096
RDREC   CLEAR   X
        CLEAR   A
        CLEAR  S
        +LDT   #4096
RLOOP   TD     INPUT
        JEQ    RLOOP
        RD     INPUT
        COMPR A,S
        JEQ    EXIT
        STCH   BUFFER,X
        TIXR   T42
        JLT    RLOOP
EXIT    STX    LENGTH
        RSUB
INPUT   BYTE   X'F1'
WRREC          CLEAR  X
        LDT    LENGTH
WLOOP          TD      OUTPUT
        JEQ    WLOOP
        LDCH   BUFFER,X
        WD     OUTPUT
        TIXR   T
        JLT    WLOOP
        RSUB
OUTPUT         BYTE    X'05'
        END    FIRST
```

NOTE:This program does not use relative addressing. Thus the addresses in all the instructions except RSUB must be modified. This would require 31 Modification records.

EXPECTED OUTPUT:

```
0       COPY    START           0
0       FIRST   STL     RETADR          17202D
```

```
3                   LDB     #LENGTH         69202D
                    BASE    LENGTH
6         CLOOP    +JSUB    RDREC  4B101036
A                   LDA     LENGTH          032026
D                   COMP    #0      290000
10                  JEQ     ENDFIL          332007
13                 +JSUB    WRREC           4B10105D
17                  J       CLOOP  3F2FEC
1A        ENDFIL            LDA     EOF     032010
1D                  STA     BUFFER          0F2016
20                  LDA     #3      010003
23                  STA     LENGTH          0F200D
26                 +JSUB    WRREC           4B10105D
2A                  J       @RETADR  3E2003
2D        EOF       BYTE    C'EOF'  454F46
30        RETADR            RESW    1
33        LENGTH            RESW    1
36        BUFFER            RESB    4096
1036      RDREC    CLEAR    X       B410
1038               CLEAR    A       B400
103A               CLEAR    S       B440
103C               +LDT     #4096   75101000
1040      RLOOP     TD      INPUT   E32019
1043                JEQ     RLOOP   332FFA
1046                RD      INPUT   DB2013
1049               COMPR    A,S     A004
104B                JEQ     EXIT    332008
104E                STCH    BUFFER,X        57C003
1051                TIXR    T        B850
1053                JLT     RLOOP   3B2FEA
1056      EXIT      STX     LENGTH          134000
1059                RSUB            4F0000
105C      INPUT     BYTE    X'F1'   F1
105D      WRREC             CLEAR   X       B410
105F                LDT     LENGTH          774000
1062      WLOOP             TD      OUTPUT          E32011
1065                JEQ     WLOOP           332FFA
1068                LDCH    BUFFER,X 53C003
106B                WD      OUTPUT          DF2008
106E                 TIXR   T       B850
1070                JLT     WLOOP           3B2FEF
1073                RSUB            4F0000
1076      OUTPUT            BYTE    X'05'   05
1077                END     FIRST
```

TEST CASE 3: Test case to check the palindrome program:

```
PALIND          START   0
FIRST   LDB     #1
CLOOP           +JSUB   RDREC
        LDA     LENGTH
        COMP            #0
        JEQ     CLOOP
        LDT     #LENGTH
        SUBR    B,T
        +JSUB   CMPREC
LENGTH          RESW    1
.
```

```
. SUBROUTINE READ RECORD
.
RDREC          CLEAR          X
        CLEAR          A
        CLEAR          S
        +LDT    #4096
RLOOP          TD      INPUT
        JEQ     RLOOP
        RD      INPUT
        COMPR           A,S
        JEQ     EXIT1
        STCH    BUFFER,X
        TIXR    T
        JLT     RLOOP
EXIT1   STX     LENGTH
        RSUB
INPUT          BYTE    X'F1'
.
. SUBROUTINE COMPARE RECORD
.
CMPREC  CLEAR          A
        COMPR  X,T
        JEQ     EXIT2
        JGT     EXIT2
        LDA     BUFFER,X
        LDS     BUFFER,T
        SUBR    B,T
        ADDR           B,X
        COMPR          A,S
        JEQ     CMPREC
        JLT     ERROR
        JGT     ERROR
EXIT2   LDA     #1
        STA     RESULT
ERROR          RSUB
RESULT         RESW    1
BUFFER         RESB    4096
        END     FIRST.
```

EXPECTED OUTPUT:

```
Line    Loc   Block    Source Statement                          Object Code
--------------------------------------------------------------------------------------------------------------------------------------

5       0000  0        FIBO        START       0
10      0000  0        FIRST       LDT         #1                750001
15      0003  0                    LDS         #10               6D000A
20      0006  0                    +JSUB       RDREC             4B100011
25      000A  0                    +JSUB       WRREC             4B10002D
30      000E  0        LENGTH              RESW          1
35                .
40                . SUBROUTINE RDREC
45                .
50      0011  0        RDREC       CLEAR       A                 B400
55      0013  0        RLOOP       TD          INPUT             E32016
60      0016  0                    JEQ         RLOOP             332FFA
65      0019  0                    RD          INPUT             DB2010
```

```
70      001C  0                      COMPR       A,T          A005
75      001E  0                      JLT         RLOOP        3B2FF2
80      0021  0                      COMPR       A,S          A004
85      0023  0                      JGT         RLOOP        372FED
90      0026  0                      STA         LENGTH               0F2FE5
95      0029  0                      RSUB                     4F0000
100     002C  0         INPUT        BYTE        X'F1'
105                 .
110                 .SUBROUTINE WDREC
115                 .
120     002D  0         WRREC        LDX         #0           050000
125     0030  0                      LDS         #1           6D0001
130     0033  0                      LDA         #0           010000
135     0036  0                      LDT         LENGTH               772FD5
140     0039  0         WLOOP        TD          OUTPUT               E32019
145     003C  0                      JEQ         WLOOP        332FFA
150     003F  0                      WD          OUTPUT               DF2013
155     0042  0                      ADDR        S,A          9040
160     0044  0                      STA         VALUE1               0F200F
165     0047  0                      STS         VALUE2               7F200F
170     004A  0                      LDA         VALUE2               03200C
175     004D  0                      LDS         VALUE1               6F2006
180     0050  0                      TIXR        T            B850
185     0052  0                      JLT         WLOOP        3B2FE4
190     0055  0         OUTPUT             BYTE         X'05'
195     0056  0         VALUE1             RESW         1
200     0059  0         VALUE2             RESW         1
205                                  END         FIRST
```

ADDITIONAL TEST CASES:

TEST CASE 4: To verify the value of the location counter (LOCCTR) :

100 START
103 LDA THREE
....
115 THREE EQU 3

The START directive specifies the value 100 (decimal).
The LC value of the above instruction is 103 (decimal).
The symbol THREE has LC value = 115 (decimal).
The assembled form of the instruction (in hex) is 000073.

TEST CASE 5: To verify the entry in the modification record:

Consider line 15 in Test case 1:

0006 CLOOP +JSUB   RDREC 4B101036

The address is 1036 from beginning address 0. Suppose the program were loaded at 1000,

Addr = 1000 - 0 + 1036, yields the correct location.

The modification record is:
M0000705   modify starting at address 7 from load of 0 5 half bytes.

TEST CASE 6: To verify the basic arithmetic operations between two user defined symbols:

```
NREC EQU 200
RSIZE EQU 15
.
.
LOC RESW NREC*RSIZE
```

The above code should multiply the value of two symbols and assign the value to LOC.

TEST CASE 7: To verify the usage of an address increment:

```
START
STA START+2
```

The above code should store the location counter value specified in the address that points to START+2.

TEST CASE 8: To check the use of macros:

```
COPY    START  0               .COPY FILE FROM INPUT TO OUTPUT
RDBUFF         MACRO           &INDEV,&BUFADR,&RECLTH
.
.          MACRO TO READ RECORD INTO BUFFER
.
        CLEAR  X               .CLEAR LOOP COUNTER
        CLEAR  A
        CLEAR  S
        +LDT   #4096           .SETMAXIMUM RECORD LENGTH
        TD     =X'&INDEV'      .TEST INPUT DEVICE
        JEQ    *-3             .LOOP UNTIL READY
        RD     =X'&INDEV'      .READ CHARACTER INTO REG A
        COMPR  A,S             .TEST FOR END OF RECORD
        JEQ    *+11            .EXIT LOOP IF EOR
        STCH   &BUFADR,X       .STORE CHARACTER IN BUFFER
        TIXR   T               .LOOP UNLESS MAXIMUM LENGTH HAS BEEN REACHED
        JLT    *-19
        STX    &RECLTH                  .SAVE RECORD LENGTH
        MEND
WRBUFF         MACRO           &OUTDEV,&BUFADR,&RECLTH
.
.          MACRO TO WRITE RECORD FROM BUFFER
.
        CLEAR  X               .CLEAR LOOP COUNTER
```

```
          LDT     &RECLTH
          LDCH    &BUFADR,X     .GET CHARACTER FROM BUFFER
          TD      =X'&OUTDEV'   .TEST OUTPUT DEVICE
          JEQ     *-3           .LOOP UNTIL READY
          WD      =X'&OUTDEV'   .WRITE CHARACTER
          TIXR    T             .LOOP UNTIL ALL CHARACTERS HAVE BEEN WRITTEN
          JST     *-14
          MEND
.
.             MAIN PROGRAM
.
FIRST     STL     RETADR                  .SAVE RETURN ADDRESS
CLOOP     RDBUFF          F1,BUFFER,LENGTH     .READ RECORD INTO BUFFER
          LDA     LENGTH                  .TEST FOR END OF FILE
          COMP    #0
          JEQ     ENDFIL        .EXIT IF EOF FOUND
          WRBUFF          05,BUFFER,LENGTH     .WRITE OUTPUT RECORD
          J       CLOOP         .LOOP
ENDFIL    WRBUFF          05,BUFFER,THREE      .WRITE OUTPUT RECORD
          J       @RETADR
EOF       BYTE    C'EOF'
THREE     WORD    3
RETADR            RESW    1
LENGTH            RESW    1               .LENGTH OF RECORD
BUFFER            RESB    4096            .4096-BYTE BUFFER AREA
          END
```

TEST CASES 9: To check the usage of program blocks:

```
COPY      START   0             .COPY FILE FROM INPUT TO OUTPUT
FIRST     STL     RETADR                  .SAVE RETURN ADDRESS
CLOOP     JSUB    RDREC         .READ INPUT RECORD
          LDA     LENGTH                  .TEST FOR EOF (LENGTH = 0)
          COMP    #0
          JEQ     ENDFIL        .EXIT IF EOF FOUND
          JSUB    WRREC         .WRITE OUTPUT RECORD
          J       CLOOP         .LOOP
ENDFIL    LDA     =C'EOF'       .INSERT END OF FILE MARKER
          STA     BUFFER
          LDA     #3
          STA     LENGTH
          JSUB    WRREC
          J       @RETADR       .RETURN TO CALLER
          USE     CDATA
RETADR            RESW    1
LENGTH            RESW    1               .LENGTH OF RECORD
          USE     CBLKS
BUFFER            RESB    4096            .4096-BYTE BUFFER AREA
BUFEND            EQU     *               .FIRST LOCATION AFTER BUFFER
MAXLEN            EQU     BUFEND-BUFFER        .MAXIMUM RECORD LENGTH
.
.            SUBROUTINE TO READ RECORD INTO BUFFER
.
          USE
RDREC     CLEAR   X             .CLEAR LOOP COUNTER
          CLEAR   A             .CLEAR A TO ZERO
```

```
        CLEAR S             .CLEAR S TO ZERO
        +LDT  #MAXLEN
RLOOP   TD    INPUT         .TEST INPUT DEVICE
        JEQ   RLOOP         .LOOP UNTIL READY
        RD    INPUT         .READ CHARACTER INTO REGISTER A
        COMPR A,S           .TEST FOR END OF RECORD (X'00')
        JEQ   EXIT          .EXIT LOOP IF EOR
        STCH  BUFFER,X              .STORE CHARACTER IN BUFFER
        TIXR  T             .LOOP UNLESS MAX LENGTH HAS BEEN REACHED
        JLT   RLOOP
EXIT    STX   LENGTH                .SAVE RECORD LENGTH
        RSUB                .RETURN TO CALLER
        USE   CDATA
INPUT   BYTE  X'F1'         .CODE FOR INPUT DEVICE
.
.         SUBROUTINE TO WRITE RECORD INTO BUFFER
.
        USE
WRRECCLEAR X                .CLEAR LOOP COUNTER
        LDT   LENGTH
WLOOPTD       =X'05'        .TEST INPUT DEVICE
        JEQ   WLOOP         .LOOP UNTIL READY
        LDCH  BUFFER,X              .GET CHARACTER FROM BUFFER
        WD    =X'05'        .WRITE CHARACTER
        TIXR  T             .LOOP UNTIL ALL CHARACTERS HAVE BEEN WRITTEN
        JLT   WLOOP
        RSUB                .RETURN TO CALLER
        USE   CDATA
        LTORG
        END   FIRST
```

TEST CASE 10: To test the usage of literals:

```
       BASE *
0003   LDB  =*   .base gets 3
    ...
0020   LDA  =*   .A gets 20
```

Anything followed by an “=” is said a LITERAL. If an ‘*’ is specified in the instruction, then the value of the current location counter is assigned to the literal.

TEST CASE 11: To test the duplication in literals:

```
        =C'EOF'
        =X'454F46'    .same literal
```

The above two literals are one and the same. They should not have two entries in the LITTAB.

CONCLUSION:

Thus, this newly proposed can be used to handle features such as Literals, Duplication of labels, Multiple Control Sections, Program Relocation, different types of addressing modes. These features cannot be handled in the current assembler. We use different methods to handle different features in an efficient manner which paves way for an optimized solution. Pass1 assigns addresses to all statements in the program, save the values assigned to all labels and processes some assembler directives while Pass2 assembles instructions, generate data values, process assembler directives which are not done in Pass1 and write the object program and assembly listing.

Hence we have presented a better and fully optimised implementation of a two pass SIC/XE assembler.