SIC – XE

ASSEMBLER

CSN 252

-21114032

-Tushar Chinchole

SIC – XE Assembler

CSN 252

- SIC/XE stands for Simplified Instructional Computer Extra Equipment or Extra Expensive . This computer is an advance version of SIC . Both SIC and SIC/XE are closely related to each other that's why are upward compatible .

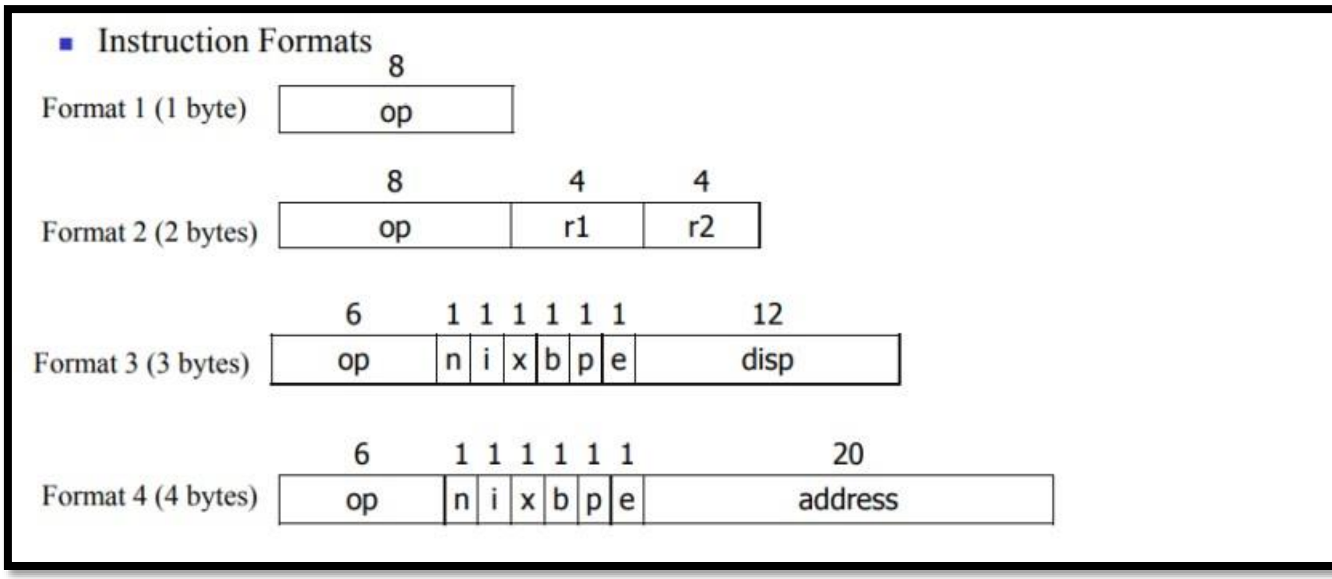
- **Memory** : Memory consists of 8 bit-bytes and the memory size is 1 megabytes (2^{20} bytes). Standard SIC memory size is very small. This change in the memory size leads to change in the instruction formats as well as addressing modes. 3 consecutive bytes form a word (24 bits) in SIC/XE architecture. All address are byte addresses and words are addressed by the location of their lowest numbered byte.

- **Registers** : It contains 9 registers (5 SIC + 4 Additional Register) . Four additional registers are :

  1. B : base register
  2. S : General working register
  3. T : General working register
  4. F : Floating point Accumulator

- **Data Formats** : Integers are represented by binart numbers . Characters are represented using ASCII codes . Floating points are represented using 48 bits .

- **Instruction Formats** : In SIC/XE architecture there are 4 types of formats available. The Bit(e) is used to distinguish between Formats 3 and 4 . e = 0 means Format 3 and e = 1 means Format 4 .



Addressing modes

- Base relative (n=1, i=1, b=1, p=0)
- Program-counter relative (n=1, i=1, b=0, p=1)
- Direct (n=1, i=1, b=0, p=0)
- Immediate (n=0, i=1, x=0)
- Indirect (n=1, i=0, x=0)
- Indexing (both n & i = 0 or 1, x=1)
- Extended (e=1 for format 4, e=0 for format 3)

- This Assembler also includes Machine – Independent Assembler Features.
    1. Literals
    2. Symbol Defining Statements
    3. Expressions
    4. Program Blocks

- The Assembler is implemented our assembler using C++ programming language.

- The Assembler uses C++ library fstream to read input from a file and write output om another file.

- The Assembler includes a shell script to compile the .cpp files.

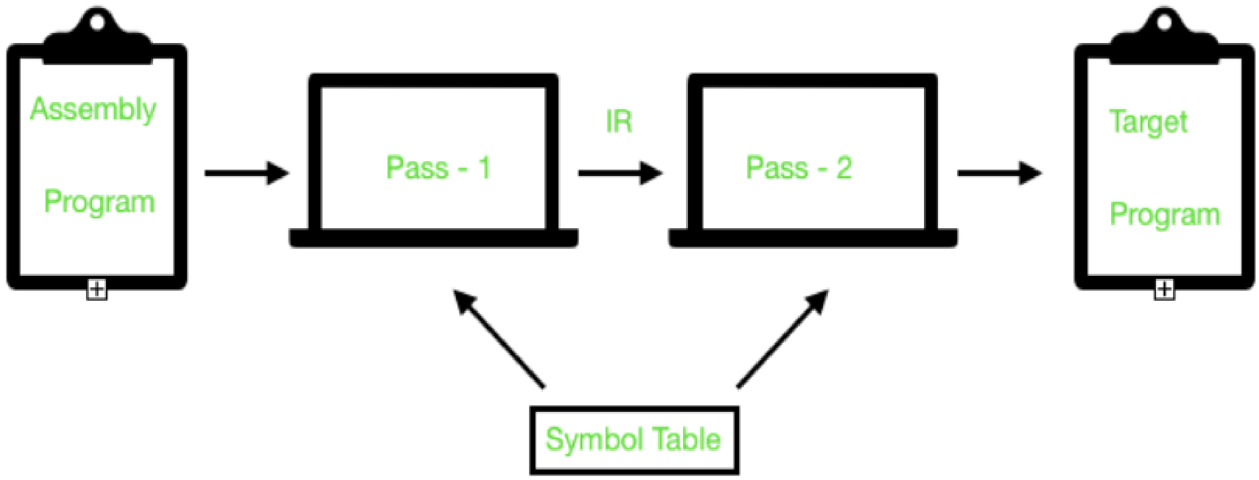
Assembler

- Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.

- It generates instructions by evaluating the mnemonics in operation field and find the value of symbol and literals to produce machine code .

- **Pass 1 :**
    1. Define symbols and literals and remember them in symbol table and literal table respectively.
    2. Keep track of location counter .
    3. Process pseudo – operations .

- **Pass 2 :**
    1. Generate object code by converting symbolic op code into respective numeric op-code.
    2. Generate data for literals and look for values of symbols .



Assembler : Design

- ***tables.cpp*** *: It contains all the data structures required for our assembler to run . It contains the structs for labels , opcode , literal , blocks. Maps are defined for various tables with their indices as strings with the names of labels or opcodes as required .*

- ***utility.cpp :*** *It contains useful functions that will be required by other files .*
    1. *convert_int_to_string_hexadecimal () : converts the input number into a string which is the hexadecimal representation of that number .*
    2. *convert_string_hex_to_int() : Converts the hexadecimal string to integer and returns the integer value of that hexadecimal number.*
    3. *convert_string_to_hexadecimal_string() : Takes in string as input and then converts the string into its hexadecimal equivalent and then returns the equivalent as string .*
    4. *expandString() : To transform the input string to the given size . It takes string to be expanded as parameter and length of output string and character to be inserted in order to expand the string.*
    5. *If_all_num() : Checks if the given input string has all the elements digits only !*
    6. *REAL_OPCODE() : for opcode of format 4 , it returns the opcode leaving the first flag bit.*
    7. *FLAG_FORMAT() : If there is a flag bit in the string then return true otherwise returns null string .*

8. *Check_comment_line() : Checks the comment by looking at the first character of the input string , and then accordingly returns true if comment or else false*
9. *read_first_non_white_space() : Iterates until it gets the first non-spaced character in the input string .*
10. *check_white_space() : if blanks are present , return true else false ;*
11. *Class AkshatEvaluateString contains the functions peek() , get() and number();*

- ***pass1.cpp :*** *We update the error file and the intermediate file using source file, If we are unable to find the source file or else if the intermediate file doesn't open , we write the corresponding error in the error file otherwise we print it to console. We declare many variables that will be required later . Then the assembler will take the first line as input and check if it is a comment line or not . Till the lines are comment , the assembler print them to the intermediate file and accordingly update the line number . Once the line is not comment we check if the opcode is START , if yes we update the line number , LOCCTR and start address if not found , we initialize start address and LOCCTR as 0 . Inside the inner loop, we check if line is a comment. If comment, we print it to our intermediate file, update line number and take in the next input line. If not a comment, we check if there is a label in the line, if present we check if it is present in the SYMTAB, if found we print error saying 'Duplicate symbol' in the error file or else assign name, address and other required values to the symbol and store it in the SYMTAB.*

  Then we verify if the opcodes are present in OPTAB or not , if present we find its other details and accordingly increment LOCCTR . If not found in OPTAB , we check it with other opcodes like WORD,BYTE,RESB etc . For opcodes like USE , we insert a new BLOCK entry in the BLOCK map as defined in utility.cpp file . For LTORG , we calla function in pass1.cpp . For 'ORG', we point out LOCCTR to the operand value given, for EQU , we check if whether the operand is an expression then we check whether the expression is valid by using the manipulateExpression() function, if valid we enter the symbols in the SYMTAB. And if the opcode doesn't match with the above given opcodes, we print an error message in the error file.

  Accordingly , we then update our data which is to be written in the intermediate file. After the loop ends , we store the program length and then go on for printing the SYMTAB , LTTAB and other tables if present .

to_handle_the_LTORG() : This function uses pass by reference . Assembler print the literal pool present till time by taking the arg from pass1 function . Assembler runs an iterator to print all the literals present in LITTAB and then update the line number . If for some reason / some literal we did not find the address , we store the present address in LITTAB and then increment the LOCCTR on basis of literal present .

Manipulate_EXPRESSION() : In this function , we use a while loop to get the symbols from the expression , If the symbol is not found in SYMTAB , we keep the error message in error file . We use a variable paircount which keeps the account of whether the expression is absolute or relative and if the paircount gives some unexpected value , we print error message .

- ***pass2.cpp :*** *Assembler takes in the intermediate file as input and generate the intermediate file and the object program . Similar to pass1 , if the Assembler is unable to open the file, it will print an error message in error file. We then read the first line of the intermediate file. Until the lines are comments, we take them as input and print them to our intermediate file and update our line number. If we get opcode as 'START', we initialize out start address as the LOCCTR, and write the line into the listing file. Then we check that whether the number of sections in our intermediate file was greater than one, if so, then we update our program length as the length of the first control section or else we keep the program length unchanged. For writing the end record we have function . For instructions with immediate addressing , we will write the modification record.*

  function_for_reading_till_TAB() : takes in the string as input and reads the string until tab('\t') occurs.

  function_for_reading_the_intermediate_file() : Takes in location counter, opcode , operand , label . If the line is comment retuns true and takes in the next input line .

Assembler : Data Structures

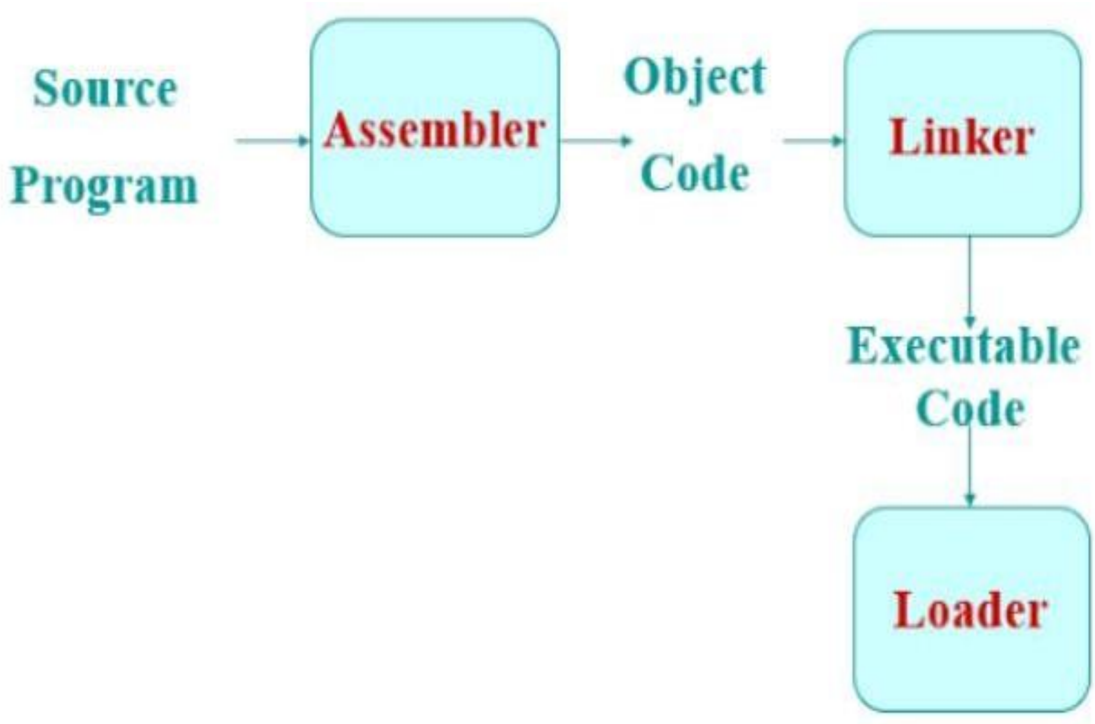
BLOCKS : *Contains information of blocks(name , start address , block number , location counter , character representing whether the block exists or not) .*

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

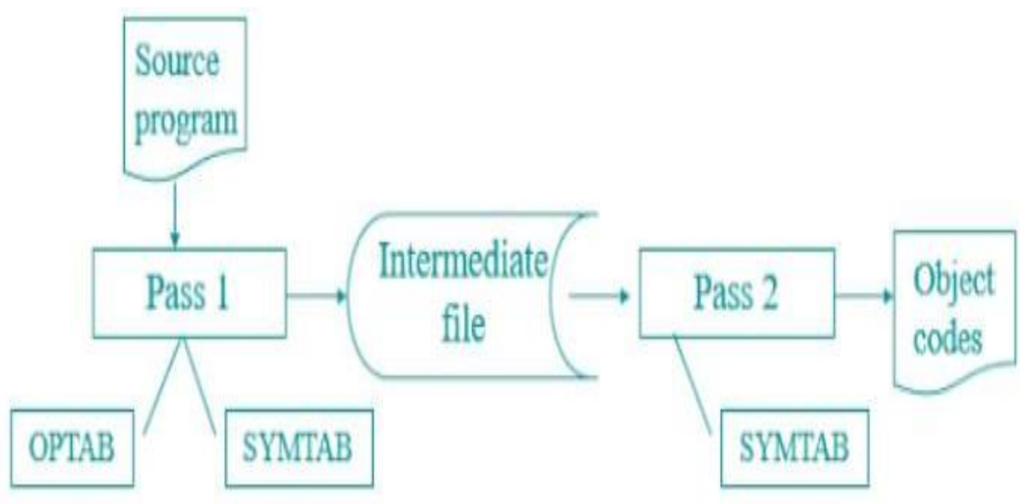
LITTAB : *Contains information of literals like its value , address , block number , a character representing whether the literal exists in literal table or not .*

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 .*

OPTAB : *Contains information of opcode like name , format , a character representing whether the opcode is valid or not .*



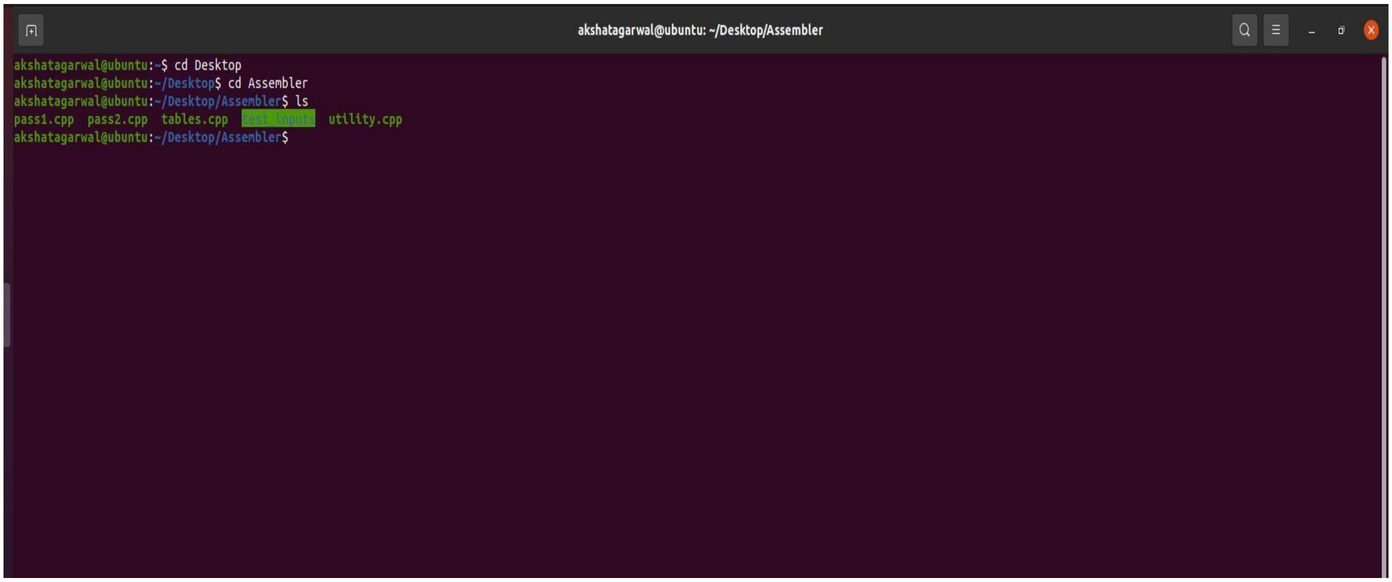
Source
Program
Assembler
Object
Code
Linker
Executable
Code
Loader



Source
program
Pass 1
Intermediate
file
Pass 2
Object
codes
OPTAB
SYMTAB
SYMTAB

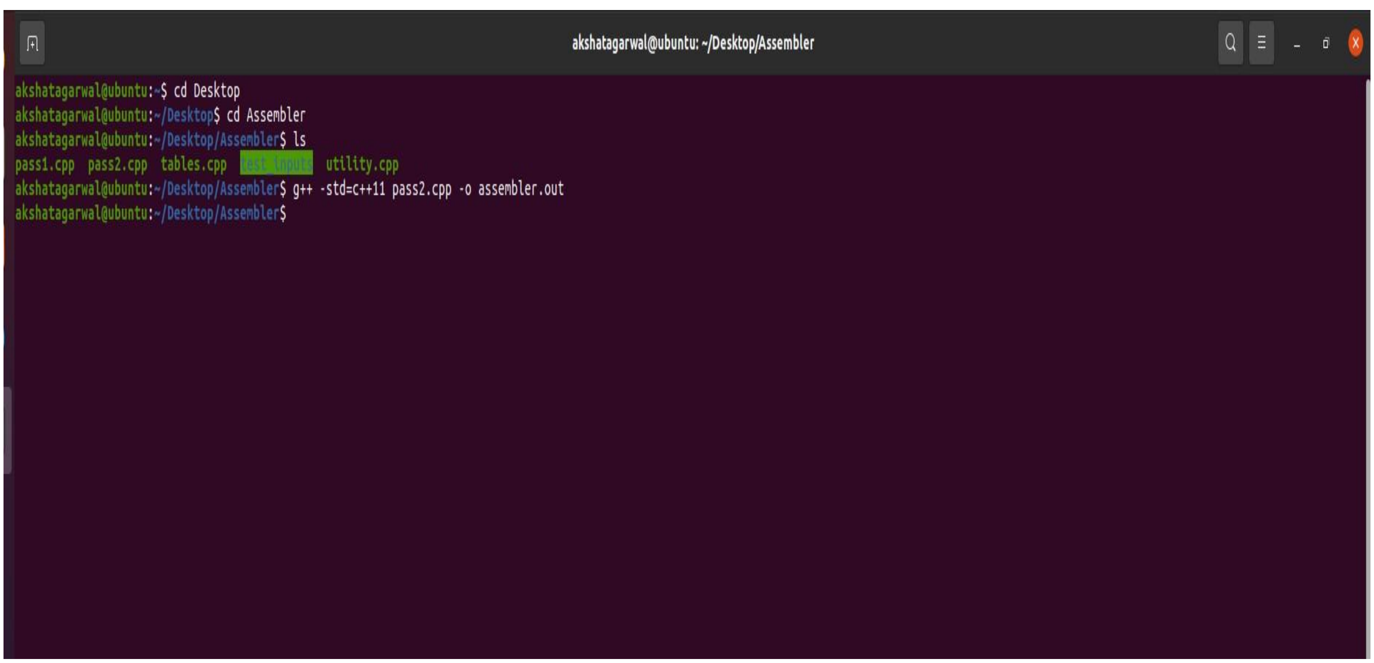
Steps to COMPILE and EXECUTE the ASSEMBLER

Step1 : *Open the terminal and open the folder in which you have stored all the files of the assembler.*

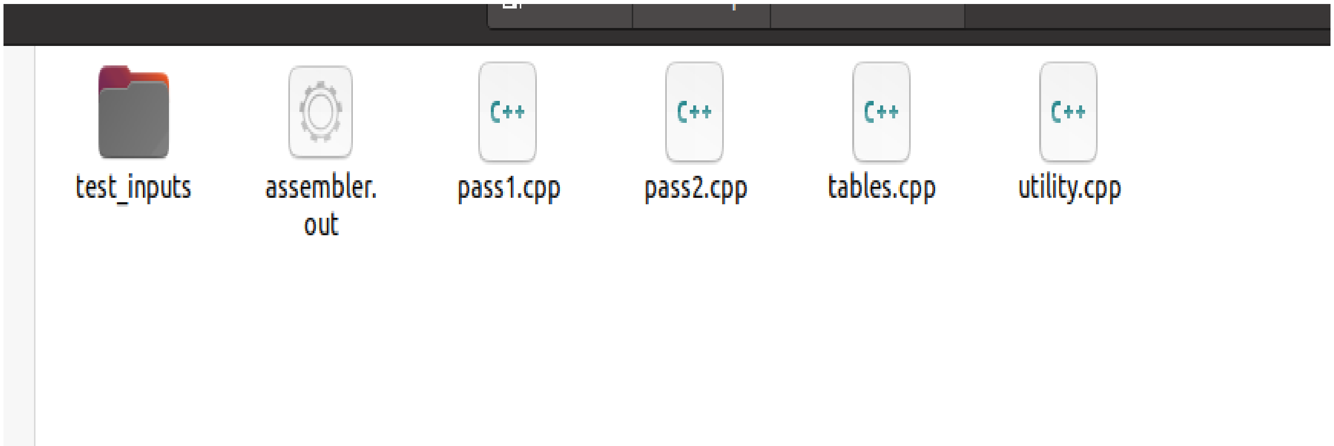


Step2 : *Run the command :*

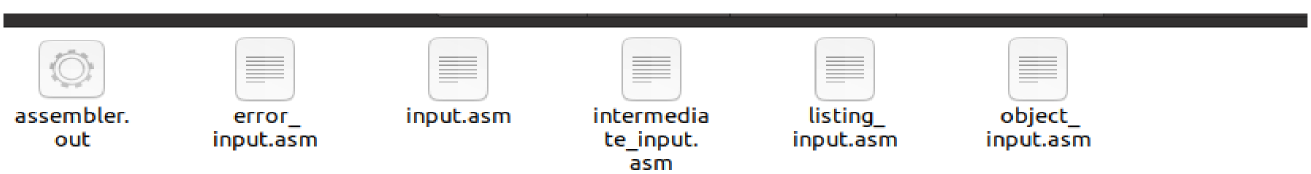
g++ -std=c++11 pass2.cpp -o assembler.out



Step3 : *After the Step2 , a file named* ***assembler.out*** *will be created in the folder. Copy the file and paste it in the* ***test_inputs*** *folder .*

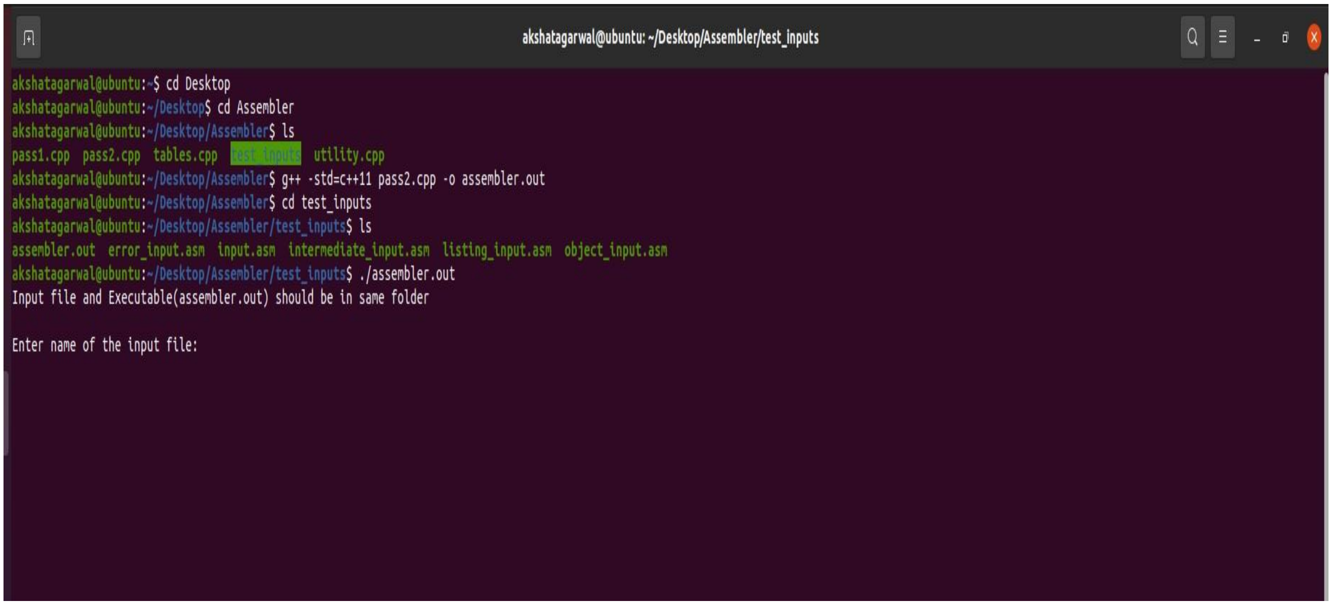


Pasted the file in test_inputs.

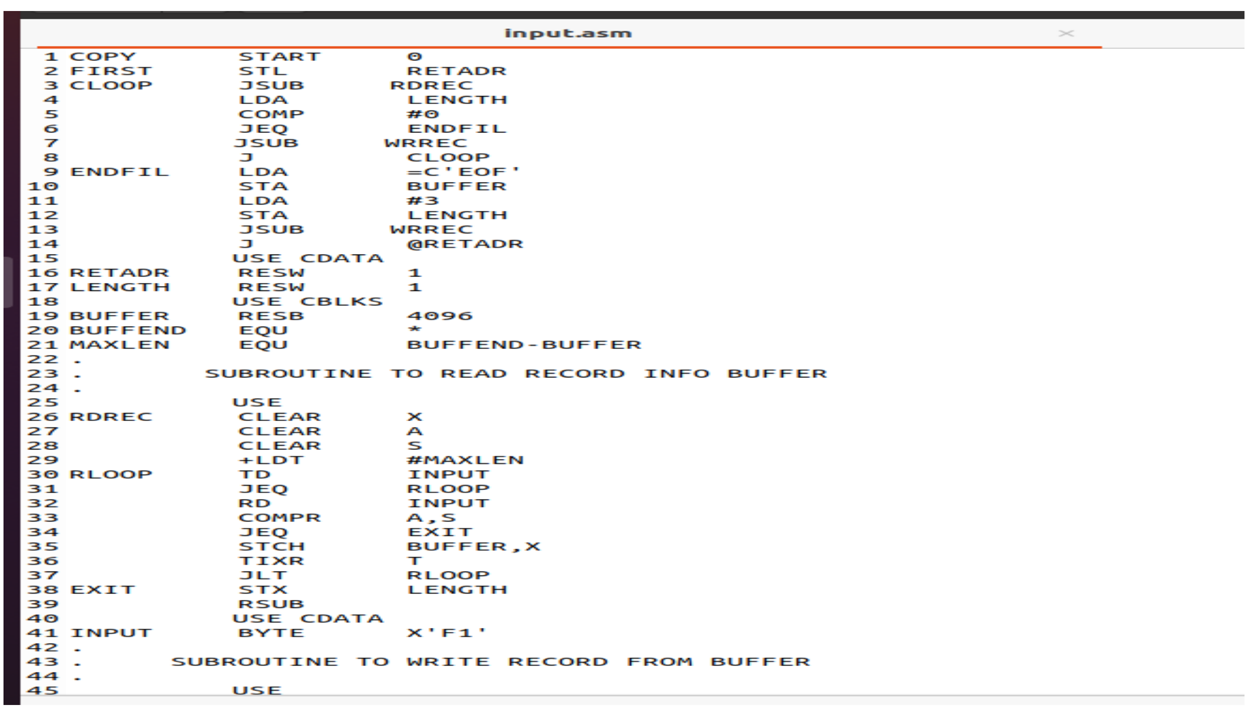


Step4 : *Now open the folder test_inputs in the terminal and run a command :*

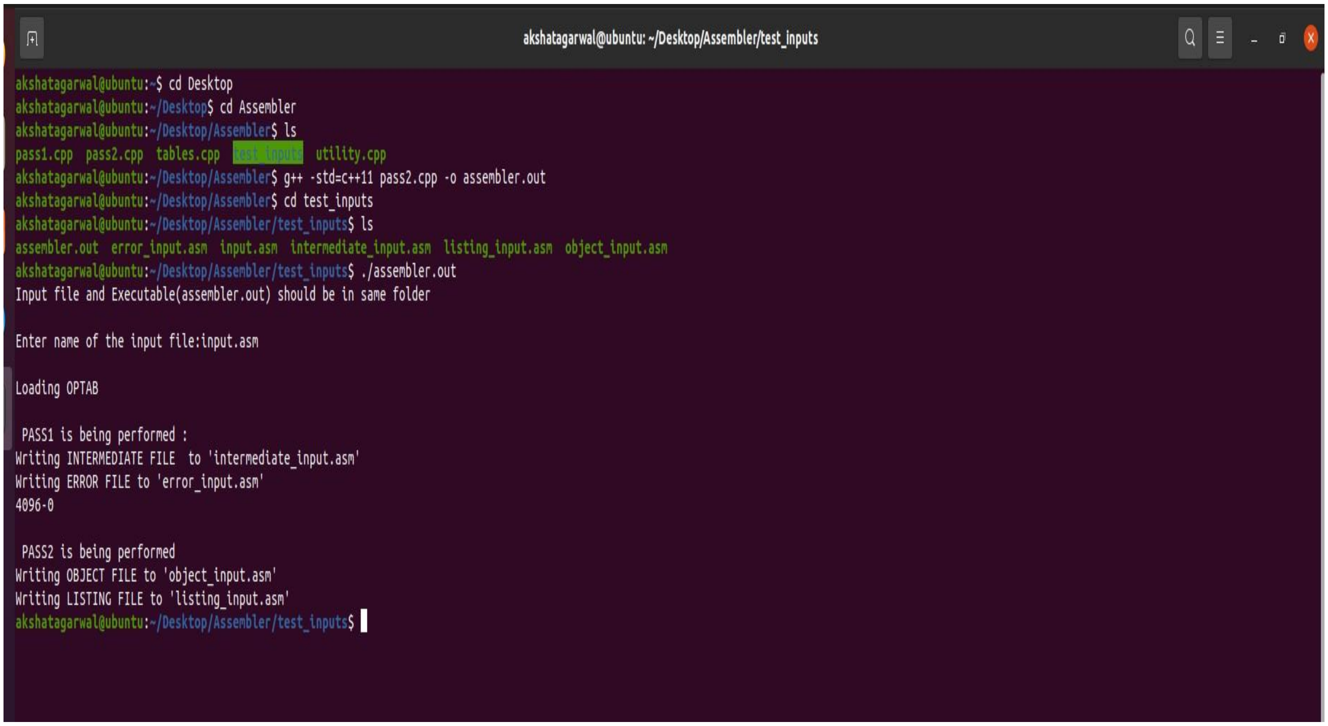
./assembler.out



Step5 : *Now copy the program, you want to run on the assembler and paste in into the file* ***input.asm*** *in folder* ***test_inputs*** *.*



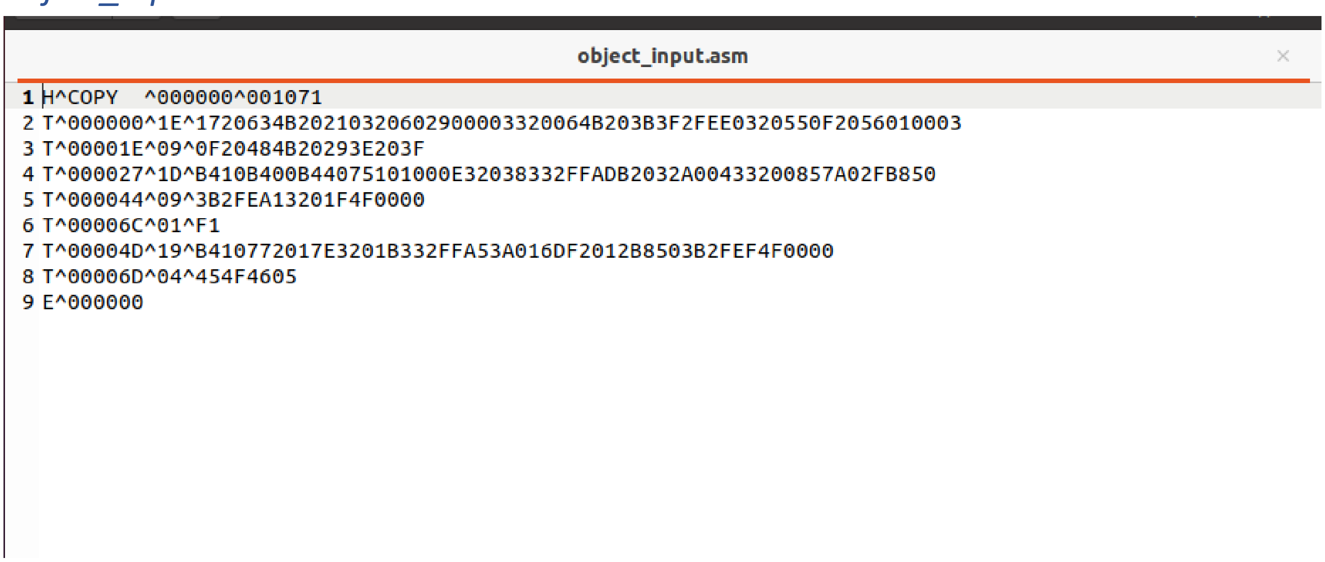
Step6 : *Now type* ***input.asm*** *in the terminal in front of : "Enter name of the input file" and press ENTER .*



```
akshatagarwal@ubuntu: ~/Desktop/Assembler/test_inputs
akshatagarwal@ubuntu:~$ cd Desktop
akshatagarwal@ubuntu:~/Desktop$ cd Assembler
akshatagarwal@ubuntu:~/Desktop/Assembler$ ls
pass1.cpp  pass2.cpp  tables.cpp  test_inputs  utility.cpp
akshatagarwal@ubuntu:~/Desktop/Assembler$ g++ -std=c++11 pass2.cpp -o assembler.out
akshatagarwal@ubuntu:~/Desktop/Assembler$ cd test_inputs
akshatagarwal@ubuntu:~/Desktop/Assembler/test_inputs$ ls
assembler.out  error_input.asm  input.asm  intermediate_input.asm  listing_input.asm  object_input.asm
akshatagarwal@ubuntu:~/Desktop/Assembler/test_inputs$ ./assembler.out
Input file and Executable(assembler.out) should be in same folder

Enter name of the input file:input.asm

Loading OPTAB

 PASS1 is being performed :
Writing INTERMEDIATE FILE  to 'intermediate_input.asm'
Writing ERROR FILE to 'error_input.asm'
4096-0

 PASS2 is being performed
Writing OBJECT FILE to 'object_input.asm'
Writing LISTING FILE to 'listing_input.asm'
akshatagarwal@ubuntu:~/Desktop/Assembler/test_inputs$
```

Step7 : *Now all the data has been written in the designated files in the folder* ***test_inputs .***

object_input.asm



```
object_input.asm
H^COPY   ^000000^001071
T^000000^1E^1720634B20210320602900003320064B203B3F2FEE0320550F2056010003
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
```

listing_input.asm

listing_input.asm

```
Line   Address        Label   OPCODE  OPERAND         ObjectCode      Comment
5      00000    0     COPY    START   0
10     00000    0     FIRST   STL     RETADR  172063
15     00003    0     CLOOP   JSUB    RDREC   4B2021
20     00006    0             LDA     LENGTH  032060
25     00009    0             COMP    #0      290000
30     0000C    0             JEQ     ENDFIL  332006
35     0000F    0             JSUB    WRREC   4B203B
40     00012    0             J       CLOOP   3F2FEE
45     00015    0     ENDFIL  LDA     =C'EOF'         032055
50     00018    0             STA     BUFFER  0F2056
55     0001B    0             LDA     #3      010003
60     0001E    0             STA     LENGTH  0F2048
65     00021    0             JSUB    WRREC   4B2029
70     00024    0             J       @RETADR         3E203F
75     00000    1             USE     CDATA
80     00000    1     RETADR  RESW    1
85     00003    1     LENGTH  RESW    1
90     00000    2             USE     CBLKS
95     00000    2     BUFFER  RESB    4096
100    01000    2     BUFFEND         EQU     *
105    01000          MAXLEN  EQU     BUFFEND-BUFFER
110    .
115    .       SUBROUTINE TO READ RECORD INFO BUFFER
120    .
125    00027    0             USE
130    00027    0     RDREC   CLEAR   X       B410
135    00029    0             CLEAR   A       B400
140    0002B    0             CLEAR   S       B440
145    0002D    0             +LDT    #MAXLEN         75101000
150    00031    0     RLOOP   TD      INPUT   E32038
155    00034    0             JEQ     RLOOP   332FFA
160    00037    0             RD      INPUT   DB2032
165    0003A    0             COMPR   A,S     A004
170    0003C    0             JEQ     EXIT    332008
175    0003F    0             STCH    BUFFER,X        57A02F
180    00042    0             TIXR    T       B850
185    00044    0             JLT     RLOOP   3B2FEA
190    00047    0     EXIT    STX     LENGTH  13201F
195    0004A    0             RSUB            4F0000
200    00006    1             USE     CDATA
205    00006    1     INPUT   BYTE    X'F1'   F1
210    .
215    .      SUBROUTINE TO WRITE RECORD FROM BUFFER
220    .
```

Testing on sample INPUTS

Program 1 : Sample program from book L L Beck section (2.2)

```
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
.
.        SUBROUTINE TO READ RECORD INFO 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'
.
.      SUBROUTINE TO WRITE RECORD FROM BUFFER
.
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
```

Object Code for Program 1 :

object_input.asm

```
H^COPY   ^000000^001077
T^000000^1D^17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T^00001D^13^0F20160100030F200D4B10105D3E2003453046
T^001036^1D^B410B400B44075101000E32019332FFADB2013A00433200857C003B850
T^001053^1D^3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T^001070^07^3B2FEF4F000005
M^000007^05
M^000014^05
M^000027^05
E^000000
```

Program 2 : L L Beck program from fig 2.9 including Literals and Expressions

```
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
RETADR    RESW    1
LENGTH    RESW    1
BUFFER    RESB    4096
BUFFEND   EQU     *
MAXLEN    EQU     BUFFEND-BUFFER
.
.       SUBROUTINE TO READ RECORD INFO BUFFER
.
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
INPUT     BYTE    X'F1'
.
.     SUBROUTINE TO WRITE RECORD FROM BUFFER
.
WRREC     CLEAR   X
          LDT     LENGTH
WLOOP     TD      =X'05'
          JEQ     WLOOP
          LDCH    BUFFER,X
          WD      =X'05'
          TIXR    T
          JLT     WLOOP
          RSUB
          END     FIRST
```

Object code for Proagram 2 :

object_input.asm

```
H^COPY  ^000000^001077
T^000000^1D^17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T^00001D^13^0F20160100030F200D4B10105D3E2003454F46
T^001036^1D^B410B400B44075101000E32019332FFADB2013A00433200857C003B850
T^001053^1D^3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T^001070^07^3B2FEF4F000005
M^000007^05
M^000014^05
M^000027^05
E^000000
```

Program3 : L L Beck program from fig 2.11 involving program blocks

```
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
.
.           SUBROUTINE TO READ RECORD INFO 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'
.
.           SUBROUTINE TO WRITE RECORD FROM BUFFER
.
            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
```

Objectcode for Program3 :

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
```