

CSN-252

System Software

SIC/XE Assembler Design

Name: Piyush Arya

Enrollment No.: 21114074

Batch: O3(CSE)

NOTE - This code works with proper spacing in the input file(as written in book) otherwise it will not work. I have written some sample programs in the folder "other inputs to test" with proper spacing. They are taken from LL Beck and the output of this code can be verified from the book. Also for default program blocks you have to write "USE DEFAULT" not just "USE". Thanks.

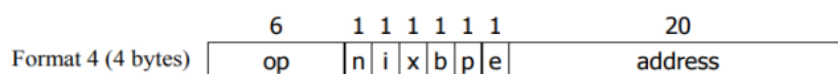
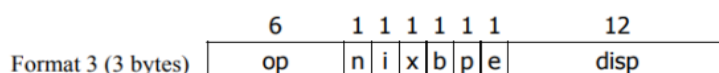
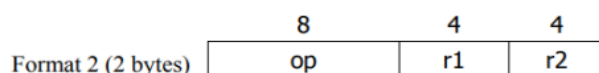
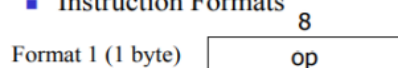
Introduction

The Objective of this project is to implement a two-pass SIC/XE assembler with *program blocks*. The Assembler that I have implemented includes all the SIC/XE instructions and supports all four formats-1, 2, 3, 4, addressing modes and program relocation.

Let us have a brief recap of the instruction formats and addressing modes:

Instruction formats

■ Instruction Formats



Formats 1 and 2 are instructions that do not reference memory at all

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=1)
- 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)

The assembler also includes the following machine-independent features:

- Literals
- Symbol defining statements
- Expressions
- Program blocks

Input

input.txt file is provided as input to assembler. This file contains the machine instructions which the assembler converts into object code.

Output

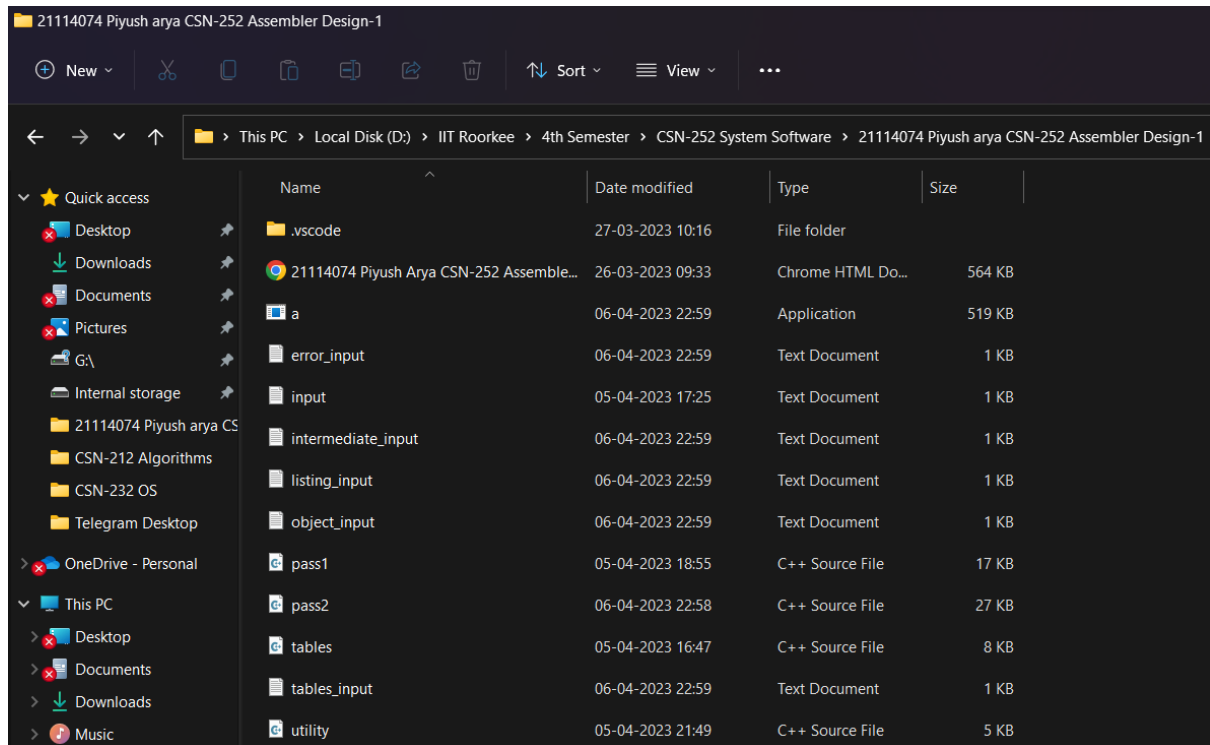
The assembler will generate the following files as output:

1. Pass 1 will generate a Symbol Table and generate an intermediate file for Pass 2.
2. Pass 1 will also generate an Intermediate File for Pass 2.
3. Pass 2 will generate a listing file containing the input assembly code and address, block number, object code of each instruction.
4. Pass 2 will also generate an object program including the following type of record: H, T, M and E types.
5. An error file is also generated displaying the errors in the assembly program (if any).

Steps to execute the assembler-

1. Unzip the file and save all the four files : pass1.cpp, pass2.cpp, tables.cpp, utilities.cpp into one directory.

2. Create the following text files in the directory - input.txt, intermediate_input.txt, listing_input.txt, object_input.txt, tables_input.txt and error_input.txt.
3. Copy the SIC/XE instructions which you want to run into input.txt.



```

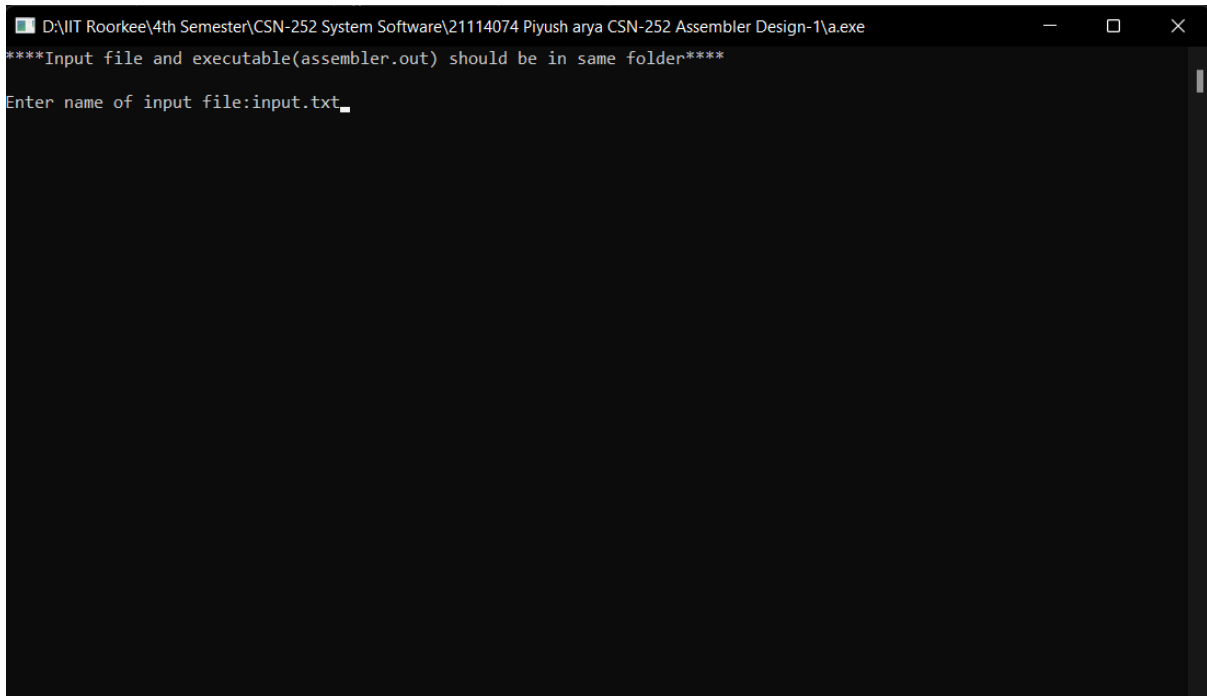
input
File Edit View

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
        USE  CDATA
COUNT RESW 1
TABLE  RESW 2000
TABLE2 RESW 2000
TOTAL  RESW 1
        END  FIRST

```

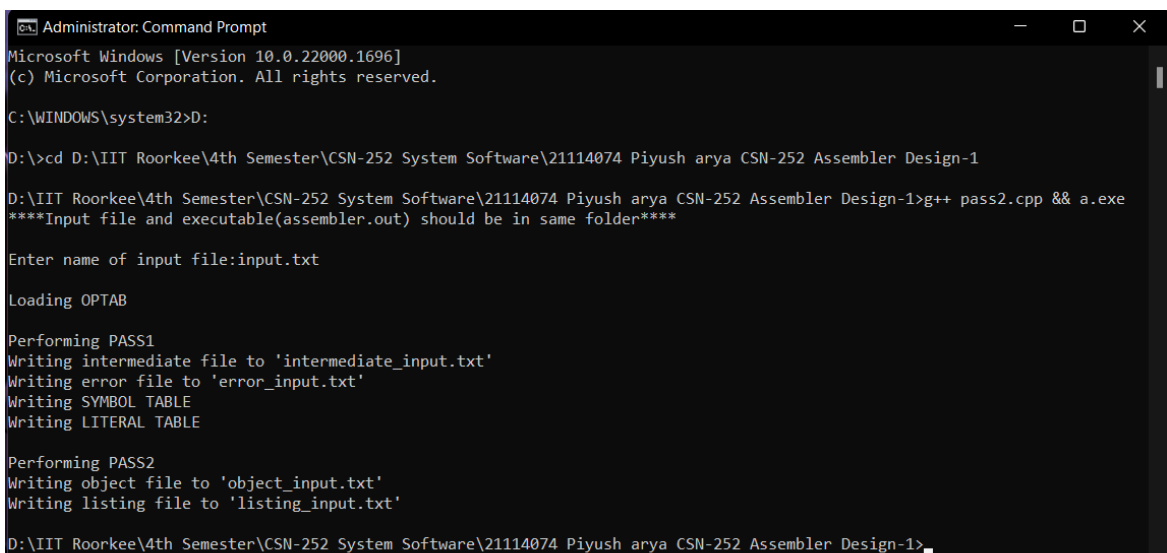
➤ Now you can execute the assembler in 1 of 2 ways -

1. Open the a.exe file which i have created in the directory and write the name of the input file when prompted and press enter.



```
D:\IIT Roorkee\4th Semester\CSN-252 System Software\21114074 Piyush arya CSN-252 Assembler Design-1\a.exe
****Input file and executable(assembler.out) should be in same folder****
Enter name of input file:input.txt
```

2. (i) Run the command prompt as administrator in windows and change the directory to where you have unzipped the file.
(ii) Write the command - "g++ pass2.cpp && a.exe" and press enter.
(iii) Write the name of the input file when prompted and press enter.



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.22000.1696]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>D:

D:\>cd D:\IIT Roorkee\4th Semester\CSN-252 System Software\21114074 Piyush arya CSN-252 Assembler Design-1

D:\IIT Roorkee\4th Semester\CSN-252 System Software\21114074 Piyush arya CSN-252 Assembler Design-1>g++ pass2.cpp && a.exe
****Input file and executable(assembler.out) should be in same folder****

Enter name of input file:input.txt

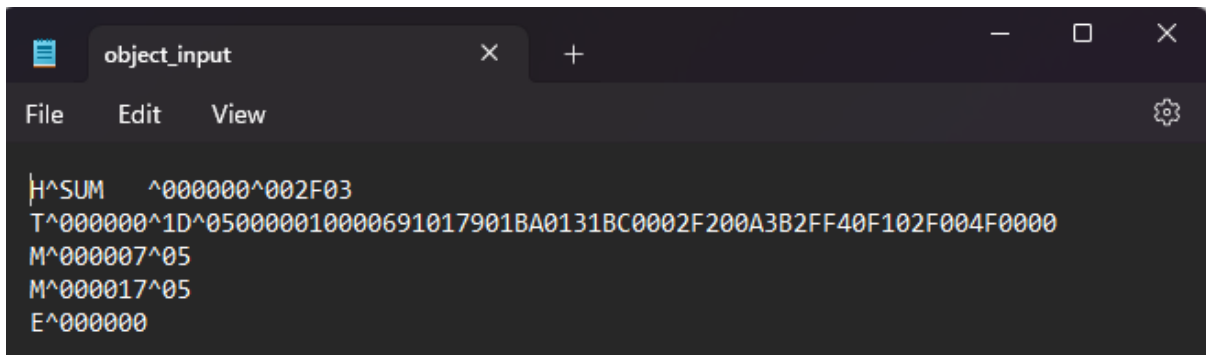
Loading OPTAB

Performing PASS1
Writing intermediate file to 'intermediate_input.txt'
Writing error file to 'error_input.txt'
Writing SYMBOL TABLE
Writing LITERAL TABLE

Performing PASS2
Writing object file to 'object_input.txt'
Writing listing file to 'listing_input.txt'

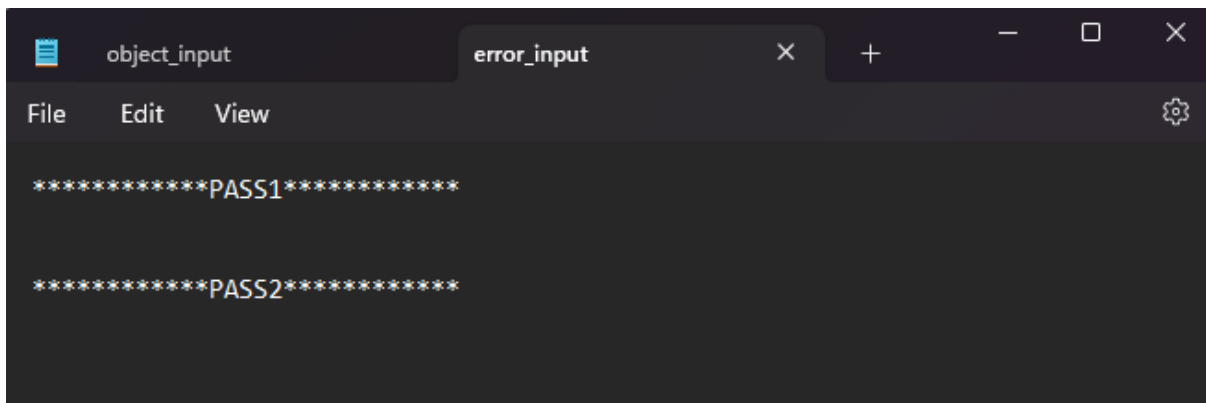
D:\IIT Roorkee\4th Semester\CSN-252 System Software\21114074 Piyush arya CSN-252 Assembler Design-1>
```

4. The object code is generated and saved in the file object_input.txt



```
H^SUM ^000000^002F03
T^000000^1D^050000010000691017901BA0131BC0002F200A3B2FF40F102F004F0000
M^000007^05
M^000017^05
E^000000
```

5. The file error_input.txt is also generated which reports any irregularity which may have been present in the instruction contained in the input file.



```
*****PASS1*****

*****PASS2*****
```

Architecture of the assembler

Functions:

Tables

It contains all the data structures required for our assembler to run. It contains the structs for labels, opcode, literal, register and program blocks. The opcode table and register table are also loaded in this file.

Utility

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

toInt()- Converts the hexadecimal string to integer and returns the integer value.

getString()- Takes in input as a character and returns a string.

expandString()- Expands the input string to the given input size. It takes in the string to be expanded as parameter and length of the output string and the character to be inserted in order to expand that string.

toHex()- Takes in input as int and then converts it into its hexadecimal equivalent with string data type.

ifAllNum()- Checks if all the elements of the string of the input string are number digits.

stringToHexString()- Takes in string as input and then converts the string into its hexadecimal equivalent and then returns the equivalent as string.

checkWhiteSpace()- checks if blanks are present. If present, returns true or else false.

checkCommentLine()- check the comment by looking at the first character of the input string, and then accordingly return true if comment or else false.

readFirstNonWhiteSpace()- takes in the string and iterates until it gets the first non-space character. It is a pass by reference function which updates the index of the input string until the blank space characters end and returns void.

writeToFile()- takes in the name of the file and the string to be written on to the file. Then writes the input string onto the new line of the file.

getRealOpcode()- for opcodes of format 4, for example +JSUB the function will see whether if the opcode contains some additional bit like '+' or some other flag bits, then it returns the opcode leaving the first flag bit.

getFlagFormat()- returns the flag bit if present in the input string or else it returns null string.

Class EvaluateString - contains the functions :

-**peek()** - returns the value at the present index.

-**get()** - returns the value at the given index and then increments the index by one.

-**number()** - returns the value of the input string in integer format.

Pass1

pass1()

In pass1() we update the intermediate file and error file using the 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 and if the error file doesn't open, we print it to console. We declare the variables required. Then we take the first line as input, check if it is a comment line. Until the lines are comments, we take them as input and print them to our intermediate file and update our line number. Once the line is not a comment we check if the opcode is 'START', if found, we update the line number, LOCCTR and start address if not found, we initialise start address and LOCCTR as 0. Then, we use a while() loop, which iterates until the opcode equals 'END'. Inside the loop, we check if the line is a comment. If it is a 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 check if opcode is present in the OPTAB, if present we find out its format and then accordingly increment the LOCCTR. If not found in OPTAB, we check it with other opcodes like 'WORD', 'RESW', 'BYTE', 'RESBYTE', 'LTORG', 'ORG', 'BASE', 'USE', or 'EQU'. Accordingly, we insert the symbols in the SYMTAB which we created. For instance, for opcodes like USE, we insert a new BLOCK entry in the BLOCK map as defined in the utility.cpp file, for LTORG we call the handleLTORG() function defined 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 evaluateExpression() 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 to printing the SYMTAB, LITAB and other tables. After that we move on to the pass2().

evaluateExpression()-

It uses pass by reference. We use a while loop to get the symbols from the expression. If the symbol is not found in the SYMTAB, we keep the error message in the 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 an error message.

handleLTORG()-

It uses pass by reference. We print the literal pool present till time by taking the arguments from the pass1() function. We run an iterator to print all the literals present

in the LITAB and then update the line number. If for some literal, we did not find the address, we store the present address in the LITAB and then increment the LOCCTR on the basis of the literal present.

Algorithm for Pass1:

Pass 1:

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR to starting address
      write line to intermediate file
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
    begin
      if this is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin
              search SYMTAB for LABEL
              if found then
                set error flag (duplicate symbol)
              else
                insert (LABEL,LOCCTR) into SYMTAB
              end {if symbol}
            end {if not a comment}
          search OPTAB for OPCODE
          if found then
            add 3 {instruction length} to LOCCTR
          else if OPCODE = 'WORD' then
            add 3 to LOCCTR
          else if OPCODE = 'RESW' then
            add 3 * #[OPERAND] to LOCCTR
          else if OPCODE = 'RESB' then
            add #[OPERAND] to LOCCTR
          else if OPCODE = 'BYTE' then
            begin
              find length of constant in bytes
              add length to LOCCTR
            end {if BYTE}
          else
            set error flag (invalid operation code)
          end {if not a comment}
        end {if not a comment}
        write line to intermediate file
        read next input line
      end {while not END}
      write last line to intermediate file
      save (LOCCTR - starting address) as program length
    end {Pass 1}
```


Pass2

pass2()

We take in the intermediate file as input using the `readIntermediateFile()` function and generate the listing file and the object program. Similar to `pass1`, if the intermediate file is unable to open, we will print the error message in the error file. Same with the object file if unable to open. 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 an opcode as 'START', we initialise our start address as the LOCCTR, and write the line into the listing file. We then write the first header record in the object program. Then until the opcode comes as 'END' we take in the input lines from the intermediate file and then update the listing file and then write the object program in the text record using the `textrecord()` function. We will write the object code on the basis of the types of formats used in the instruction. Based on different types of opcodes such as 'BYTE', 'WORD', 'BASE', 'NOBASE', etc. we will generate different types of object codes. For the format 3 and format 4 instruction format, we will use the `createObjectCodeFormat34()` function in the `pass2.cpp`. For writing the end record, we use the `writeEndRecord()` function.

For the instructions with immediate addressing, we will write the modification record.

readIntermediateFile()-

Takes in line number, LOCCTR, opcode, operand, label and input output files. If the line is commented, it returns true and takes in the next input line. Then using the `readTillTab()` function, it reads the label, opcode, operand and the comment. Based on the different types of opcodes, it will count in the necessary conditions to take in the operand.

readTillTab()-

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

createObjectCodeFormat34()-

When we get our format for the opcode as 3 or 4, we call this function. It checks the various situations in which the opcode can be and then taking into consideration the operand and the number of half bytes calculates the object code for the instruction. It also modifies the modification record when there is a need to do so.

writeEndRecord()-

It will write the end record for the program. After the execution of the `pass1.cpp`, we will print the Tables like SYTAB, LITAB, etc., in a separate file and then execute the `pass2.cpp`

Algorithm for pass 2:

Pass 2:

```
begin
  read first input line {from intermediate file}
  if OPCODE = 'START' then
    begin
      write listing line
      read next input line
    end {if START}
  write Header record to object program
  initialize first Text record
  while OPCODE ≠ 'END' do
    begin
      if this is not a comment line then
        begin
          search OPTAB for OPCODE
          if found then
            begin
              if there is a symbol in OPERAND field then
                begin
                  search SYMTAB for OPERAND
                  if found then
                    store symbol value as operand address
                  else
                    begin
                      store 0 as operand address
                      set error flag (undefined symbol)
                    end
                  end {if symbol}
                else
                  store 0 as operand address
                  assemble the object code instruction
                end {if opcode found}
              else if OPCODE = 'BYTE' or 'WORD' then
                convert constant to object code
              if object code will not fit into the current Text record then
                begin
                  write Text record to object program
                  initialize new Text record
                end
              add object code to Text record
            end {if not comment}
          write listing line
          read next input line
        end {while not END}
      write last Text record to object program
      write End record to object program
      write last listing line
    end {Pass 2}
```