**TWO PASS ASSEMBLER**

PROBLEM STATEMENT

Write a two-pass assembler for the 12-bit accumulator architecture

SOLUTION

In this project, we have designed an implementation of a two-pass assembler. The assembler takes a text file as an input. The text file contains the assembly code. The assembler translates this assembly code into machine code and saves it as a separate text file.

*Programming Language: Java*

ASSUMPTIONS

The following are the assumptions that we have been given while building the two-pass assembler:

1. There are no Macros and Procedures in the assembly code.
2. All operands are declared at the end of the instructions, sequentially and in the order of their appearance in the code.
3. Each operand occupies one word.

ASSUMPTIONS

The following are the assumptions that we have taken while building the two-pass assembler:

1. The location counter starts with 0 address
2. Location counter is always incremented by 1
3. The symbols can be of maximum 7 characters
4. Comments must begin with “@”
5. All the instructions must be separated only by 1 single space
6. Instructions must be written in capital alphabets
7. Register Number must be between 1 and 10
8. The available memory space is 2^8 bytes
9. Assembly Code should be in a text file named as “Sample”

INPUT FILES

The following files are the input files of the project:

1. TwoPassAssembler.java

Java file containing the program for the two-pass assembler

1. SampleInput.txt

Text file containing the assembly code that needs to converted to machine code

1. InstructionTable.txt

Text file containing the instructions table

1. RegisterTable.txt

Text file containing the register table

OUTPUT FILES

The following files are part of the output file of the project:

1. Output.txt

Text file containing the final machine language code generated using the assembler

1. Pass1.txt

Text file containing the output of the assembler after pass 1 is completed

1. OnlySymbolTable.txt

Text file containing just the name of all the symbols declare in the assembly code

1. SymbolTable.txt

Text file containing the symbols list with their values and addresses

GETTING STARTED

Please follow these instructions to use the assembler:

1. Prepare a valid input file by following the above stated assumptions
2. Make sure to place all the input files in the same folder
3. Execute the program
4. The desired output file is created in the same folder

NOTE: Please keep all the files in the same folder as the directory where your Java program is stored in computer. For Example; Since I am using Eclipse Workspace for Java, hence, I will store the files in C:\\Users\\Piyush Sharma\\eclipse-workspace\\Assembler

ERRORS HANDLED

The following are the errors that the assembler handles after reading the given assembly code:

1. ERROR: The code does not have a stop condition
2. ERROR: The memory is full
3. ERROR: Opcode supplied with too many operands
4. ERROR: Invalid opcode
5. ERROR: Please specify the opcode
6. ERROR: The specified symbol exceed seven characters
7. ERROR: Please specify the operand
8. ERROR: Duplicate labels cannot exist in the table
9. ERROR: No operand required
10. ERROR: Symbol has been defined more than once
11. ERROR: Symbol not declared but used

SUPPORT

If you have any issues regarding this project, feel free to contact us.

Vani Sikka vani19405@iiitd.ac.in

Piyush Sharma piyush19316@iiitd.ac.in

**UNDERSTANDING THE ASSEMBLER PROJECT**

The main objective behind this whole project is to design an assembler that converts assembly language code into machine language code.

So, to thoroughly understand the project, it is important to learn/revise some theory related to the project.

**ASSEMBLY LANGUAGE**

*Assembly language is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. In simple words,* ***Assembly code****is plain-text and (somewhat) human read-able source code that mostly has a direct 1:1 analog with machine instructions. This is accomplished using mnemonics for the actual instructions, registers, or other resources. Examples include JMP and MULT for the CPU's jump and multiplication instructions.*

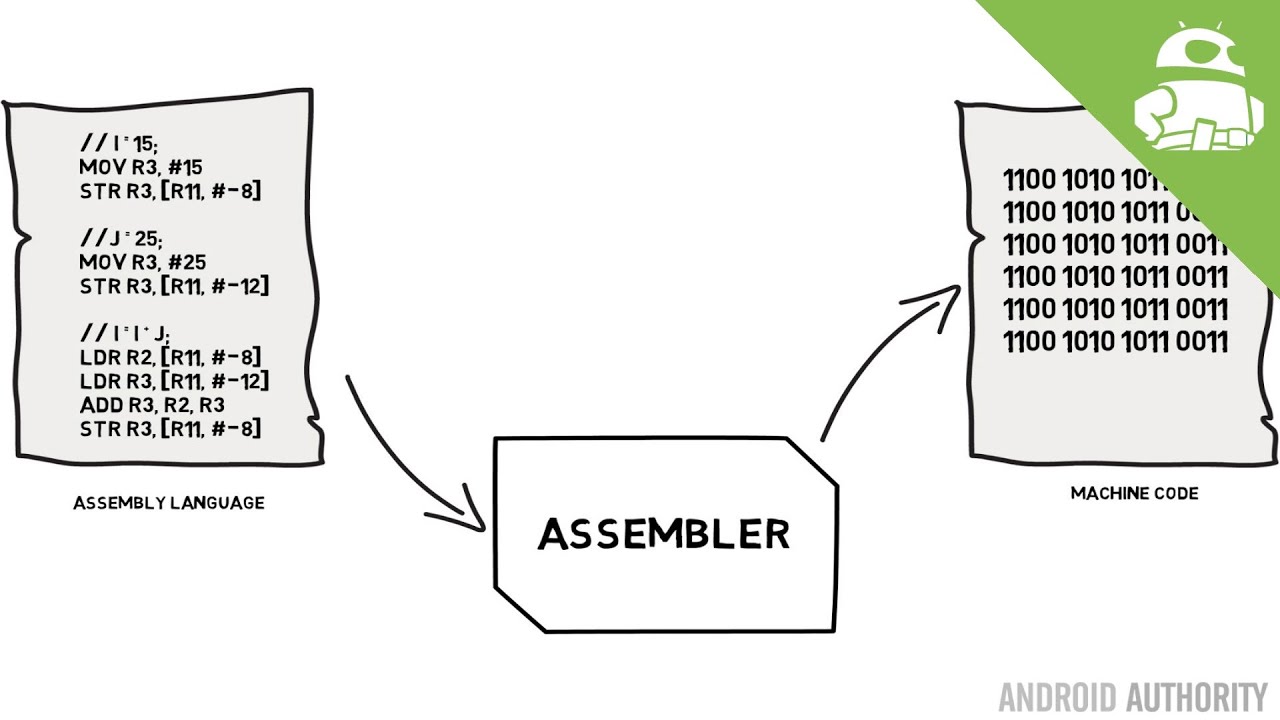
**MACHINE LANGUAGE**

*Machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding.*

*In simple words, Machine code is binary (1's and 0's) code that can be executed directly by the CPU. If you were to open a machine code file in a text editor you would see garbage, including unprintable characters.*

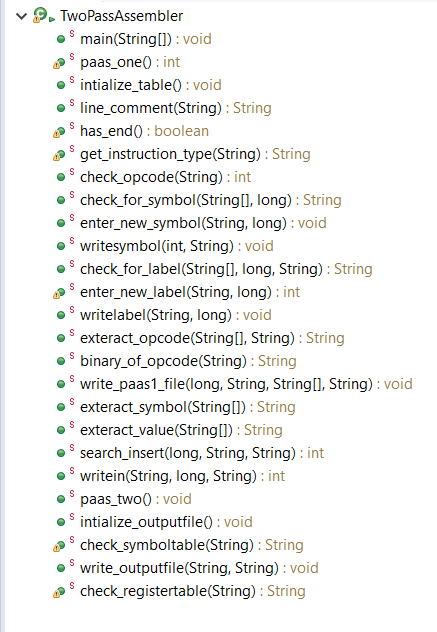
**ASSEMBLER**

*Unlike machine code, the CPU does not understand assembly code. You convert assembly code to machine with the use of an****Assembler.*** *An assembler is a*[*program*](https://techterms.com/definition/program)*that converts*[*assembly language*](https://techterms.com/definition/assembly_language)*into machine code. It takes the basic commands and operations from assembly code and converts them into*[*binary*](https://techterms.com/definition/binary)*code that can be recognized by a specific type of*[*processor*](https://techterms.com/definition/processor)*.*



**LIST OF FUNCTIONS**

Herby, follows a list of all the functions used in the Java program for the assembler:



FUNCTIONS AND THEIR EXPLAINATION

Hereby, follows the list of functions used in the program with a quick explaination about them.   
The list is sorted in the order of the appearance of the functions in the code.

**public** **static** **void** main(String[] args) **throws** Exception

The execution of the program begins with main(). This is the first function to be executed when the assembler is loaded. It contains the function call to Pass1 and Pass2 of the assembler.

**public** **static** **int** paas\_one() **throws** Exception

This is the Pass1 of the Two-Pass Assembler. It is called by the main() function. This main objective of this function is to prepare the SymbolTable, OnlySymbolTable and Paas1 text files. If there are error in the assembly code, then they are detected by this function.

**public** **static** **void** intialize\_table() **throws** Exception

This function is used to create the output text files.

SymbolTable, Pass1 and OnlySymbolTable text files are created here if they don’t already exist in the system. But if such files exist in the system, then these files are cleared i.e, all the previous data gets erased.

**public** **static** String line\_comment(String line)

This function is to tell whether the passed string is a comment or not. If the whole string is a comment ,i.e, it’s first character is “@”, then this function will return Null. But if that is not the case, then this returns the paased string after removening all part of the string which is after “@” (if there is character after “@”).

**public** **static** **boolean** has\_end() **throws** Exception

This function returns true if there is an end statement in the assembly program and returns false if there is no end statement in the assembly program.

**public** **static** String get\_instruction\_type(String l) **throws** Exception

In this function, we will search each opcode in the instruction table.

And find which opcode is used in the instruction using find substring function and then we will return the respective type value mentioned in the text file of instruction table.

**public** **static** **int** check\_opcode(String ops) **throws** Exception

This function checks each line of instruction in the assembly code to know whether the opcode is present in the instruction or not. If the opcode is present, then the function returns the opcode and if it is not present, then the function returns Null.

**public** **static** String check\_for\_symbol(String[] l, **long** loc)

This function returns Null if the passed array does not contain the symbol.

But if the array contains the symbol then it returns the symbol used in the instruction.

**public** **static** **void** enter\_new\_symbol(String symbol, **long** loc) **throws** Exception

This function enters the passed symbol in the OnlySymbolTable if it is already not present in the OnlySymbolTable file.

**public** **static** **void** writesymbol(**int** f,String s) **throws** Exception

This just a helper function to enter\_new\_symbol.

**public** **static** String check\_for\_label(String[] l, **long** loc, String t)

This function is to check if the passed line of instruction contains a label or not. If it contains the label, then it returns the label and if it does not contain the label, then it returns Null.

**public** **static** **int** enter\_new\_label(String label, **long** loc) **throws** Exception

In this function, we will first check whether the given label exists in the symbol table or not. If the given label is not present in the table, then add the label in the symbol table and it's location as the value of the location\_counter.

If the label already exists in the table then throw an error as duplicate label

names can not exist in the code.

NOTE: Labels and Symbols both are stored in the same file called the Symbol Table file

**public** **static** **void** writelabel(String l,**long** loc) **throws** Exception

This just a helper function to enter\_new\_label.

**public** **static** String exteract\_opcode(String[] l, String t)

This function is used to return the opcode in the given instruction. But if the opcode is not present in the given instruction, then it returns Null.

**public** **static** String binary\_of\_opcode(String opcode) **throws** Exception

This function searches the given opcode in the instruction table and returns

the binary value mentioned in the instruction file against the specified

opcode. But if the given opcode does not exists in the instruction table, then it

returns null and reports an error as it is an invalid opcode.

**public** **static** **void** write\_paas1\_file(**long** loc, String opcode\_binary, String[] line, String type) **throws** Exception

This function writes the (location counter value): (binary value of the opcode): (operand as it is (not in binary terms)) in the paas1 text file.

If there is label in the code file, it will not be written in the paas1 file as type 2 and type 3 instructions are imperative instruction which will be written in this way only.

NOTE: Since the operands are not in binary terms, hence they will be converted to the binary addresses in the second pass.

**public** **static** String exteract\_symbol(String[] l)

This function is simply used to extract the symbol from the string.

**public** **static** String exteract\_value(String[] l)

This function is simply used to extract the value from the string.

**public** **static** **int** search\_insert(**long** loc, String symbol, String val) **throws** Exception

This function searches the variable in the symbol table and on finding it. The function assigns the variable an 8 bit address against it , i.e, the address is nothing but 8 bit long location counter value and the function also assigns the value of the symbol against the symbol after the address.

**public** **static** **int** writein(String s,**long** loc,String val) **throws** Exception

This just a helper function to search\_insert().

**public** **static** **void** paas\_two() **throws** Exception

This function’s main objective is to make the output file by using data in the SymbolTable file, Pass1 file and RegisterTable file. When this fuction gets finished the machine code of the passed assembly code is available in the Output file.

**public** **static** **void** intialize\_outputfile() **throws** Exception

This function creates the file where the output i.e the machine language code of the given assembly language code will be written. It creates the file in the same directory where the other files related to this code are saved for easy access.

**public** **static** String check\_symboltable(String s) **throws** Exception

This function checks the given symbol (s) in the symbol table. If the symbol

is present in the symbol table, then it would return the string stored beside

the the name of the symbol; as each line in the symbol table has either

three parts (symbol name: symbol address: symbol value) or two parts (symbol

name: symbol address). The purpose of this function is to return the symbol

address if the symbol is present in the symbol table or return null if the

symbol is not not present in the symbol table.

**public** **static** **void** write\_outputfile(String opcode, String symbol) **throws** Exception

This function writes the final machine code for the present instruction in

the output file. The main task is to write the opcode symbol space separated

as it is in the output file when symbol is not null. But if the symbol is null,

then you have to write the opcode only in the output file.

**public** **static** String check\_registertable(String l) **throws** Exception

This function returns the register location of the particular register

specified in the instruction by searching the register table file (which

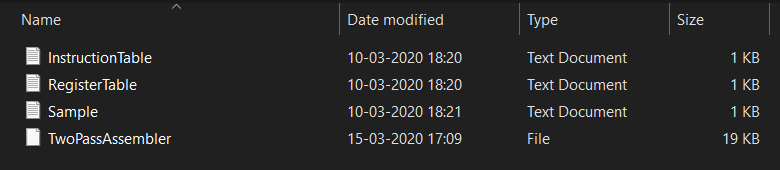
contains the location of the register numbers i.e contains address of R1, R2

etc) But if the String (l) is not found in the register file then it will return

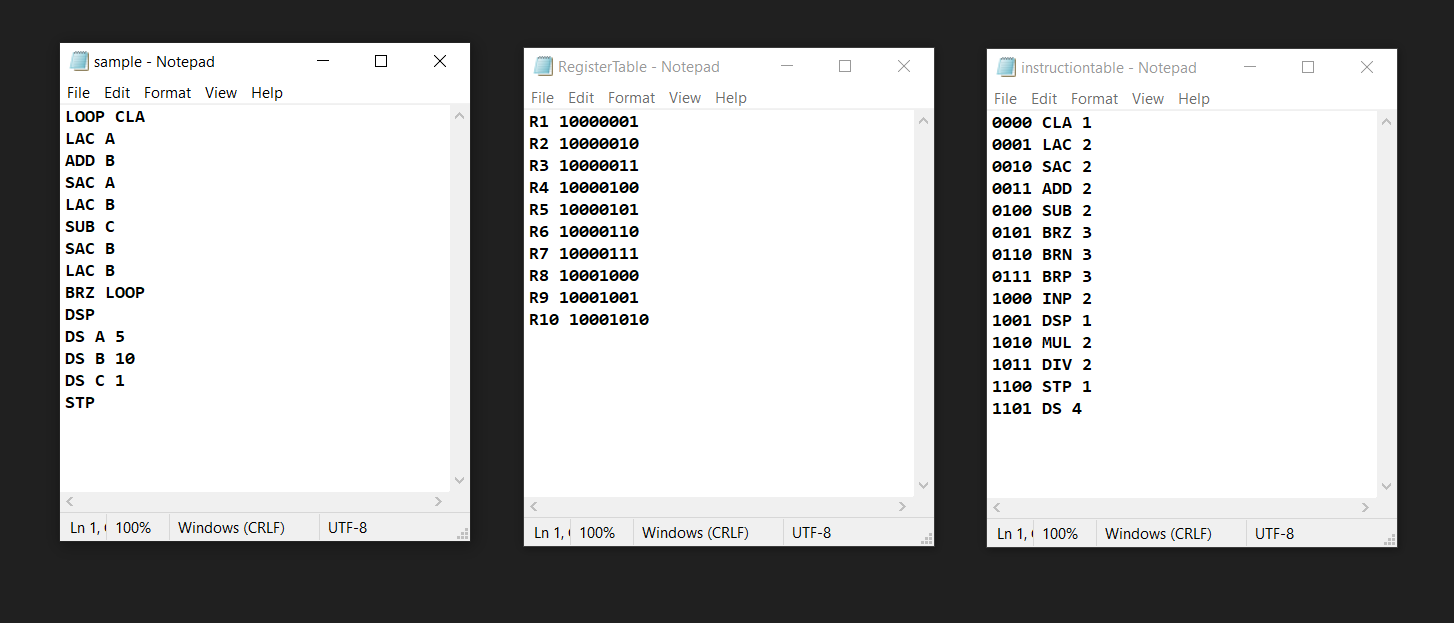
null.

RUNNING THE ASSEMBLER

Herby are few pictures of the working of the two-pass assembler with a short description followed for each one of them.



These are the three input text files with the Java program stored in the same directory.

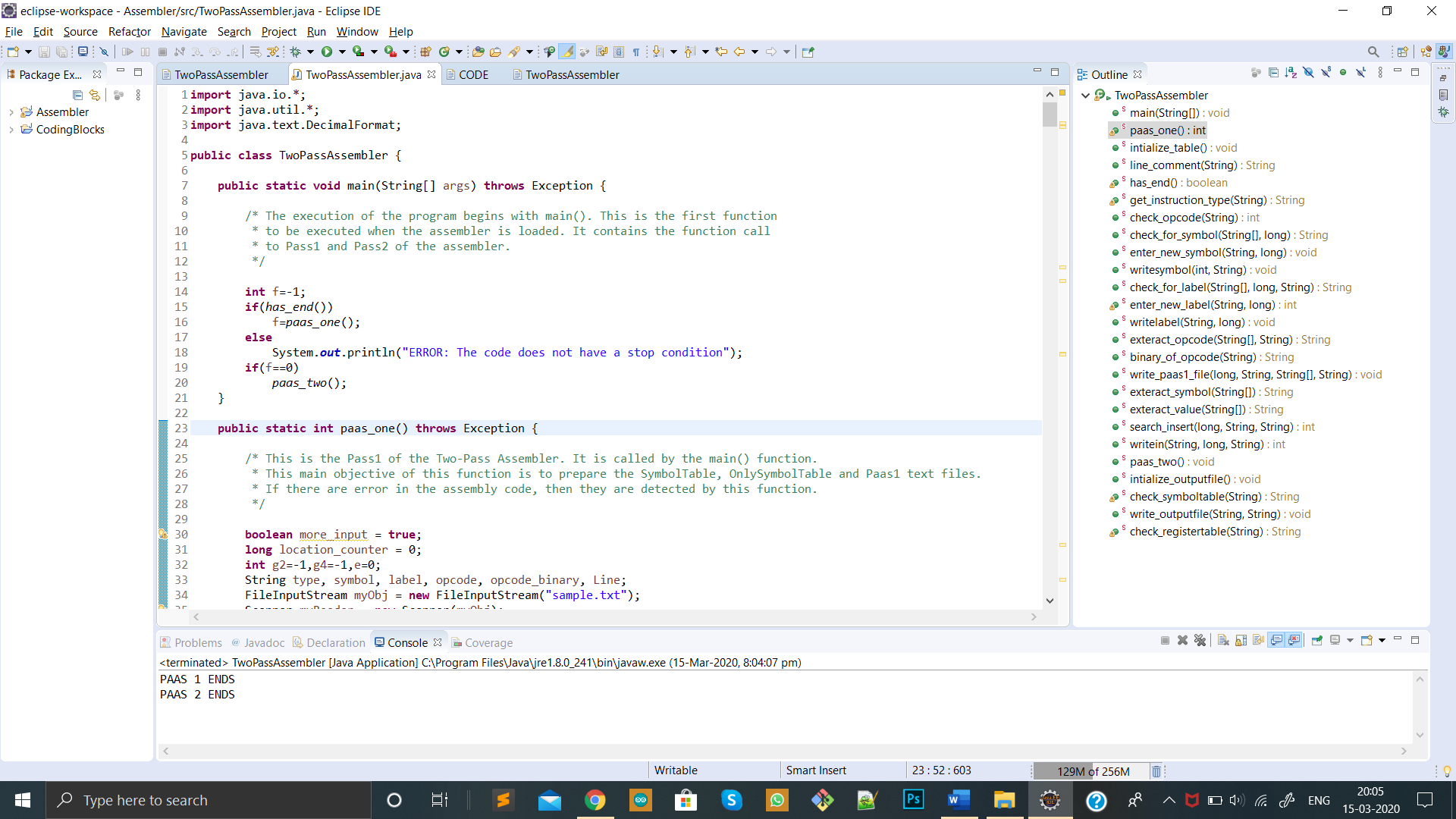


Sample file consists of the actual assembly language code that needs to be fed into the assembler for conversion.

Register Table file contains the address of the Registers from R1 to R10

Instruction Table text file is home to the instructions that we have followed while making this assembler. All the asssemly code that is written in sample.txt should be written in accordance with the Instructions Table text file.

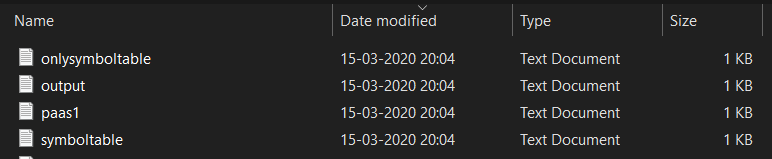
The numeric 1,2,3 and 4 in this file indicate the different types of instructions in the assembly language program.



This is a snapshot after the assembler has loaded and worked the code.  
NOTE: The line “PASS 1 ENDS indicates that the program ran successfully.

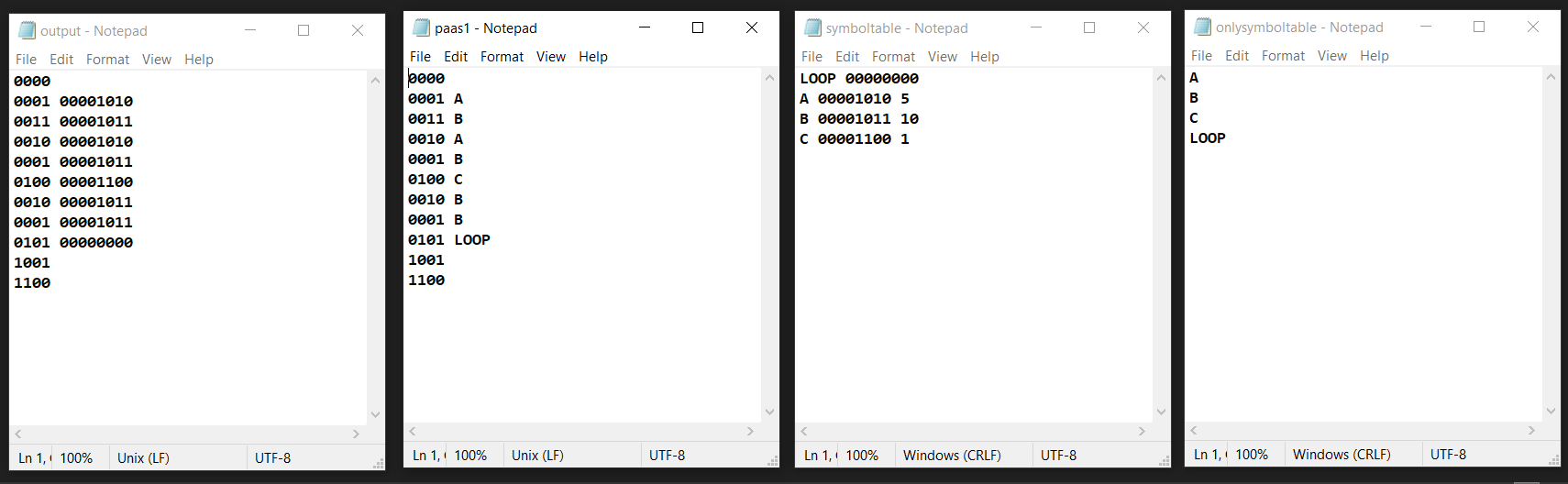
PASS 2 ENDS”

Had there been a problem while execution, it would have thrown an ERROR.



These are the three output text files.

NOTE: They are stored in the same directory.



These are the 4 output files generated after running the assembler.

NOTE: “Output” file contains the final machine level language code generated after conversion with the assembler.