Ho Chi Minh City University of Technology

FACULTY OF COMPUTER SCIENCE & ENGINEERING

Assignment Report

Course: Computer Architecture Lab - CO2008

Subject: Calculator using MIPS

Name: Dương Gia Bảo - 2252061

Lecturer: Nguyễn Thiên Ân

Ho Chi Minh City, April 2024

I. Introduction

A calculator is typically a portable electronic device used to perform calculations, ranging from basic arithmetic to complex mathematics [1].

The calculator idea starts from long ago where people would use items in their possession to count. After some time, when civilization became more and more advanced, mechanical calculators emerged and came to be one of the fundamental tools for a person. At the time, such tools can only offer very few functionalities, including addition and subtraction. After the invention of the first solid-state electronic calculator in the 1960s, calculators began to have memory storing capabilities. Therefore, complex instructions can be calculated. Nowadays, calculators are inseparable from the modern society. From elementary schools to university and marketplaces, these pocket-size devices can be found supporting people daily calculations anywhere in the world.

A simple modern calculator should be able to solve basic arithmetic such as addition, subtraction, multiplication, division, and utilizing memory. Some other frequently used functions include factorization, finding Lowest Common Multiple (LCM), Greatest Common Divisor (GCD), square roots, exponents, and converting bases.

I will implement the best calculator using MIPS assembly, with full function listed.

II. Implement

1. Theory and the basic idea:

Because the compiler scans the expression either from left to right or from right to left, it can not use to calculate infix expression. So we have to convert the infix to a **post-fix** or prefix to make the computer understand. Here, we will use **post-fix**

To convert from infix to **post-fix** expression, we use the stack data structure. Scan the infix expression from left to right. Whenever we get an operand, add it to the **post-fix** expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.

Below are the steps to implement the above idea:

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, put it in the **post-fix** expression.

3. Otherwise, do the following

- If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '('], then push it in the stack. ['^' operator is right associative and other operators like '+','–','*' and '/' are left-associative].

  + Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.

  + In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

  + After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is a '(', push it to the stack.

5. If the scanned character is a ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.

6. Repeat steps 2-5 until the infix expression is scanned.

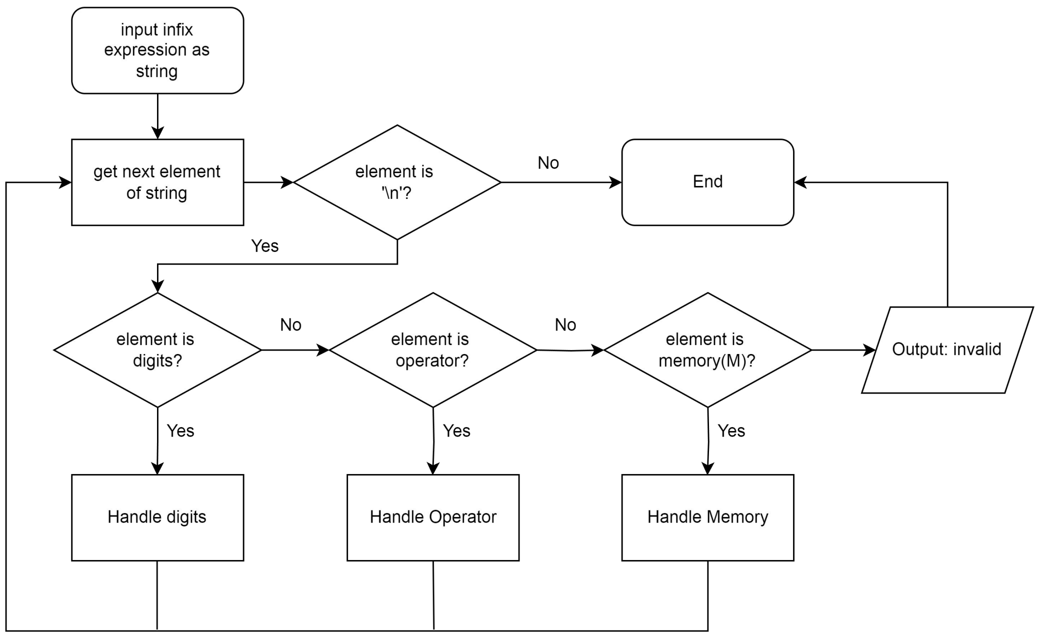
7. Once the scanning is over, Pop the stack and add the operators in the **post-fix** expression until it is not empty.

8. Finally, print the **post-fix** expression.

For the illustration, you can review it here: Convert infix to postfix

2. Implement:

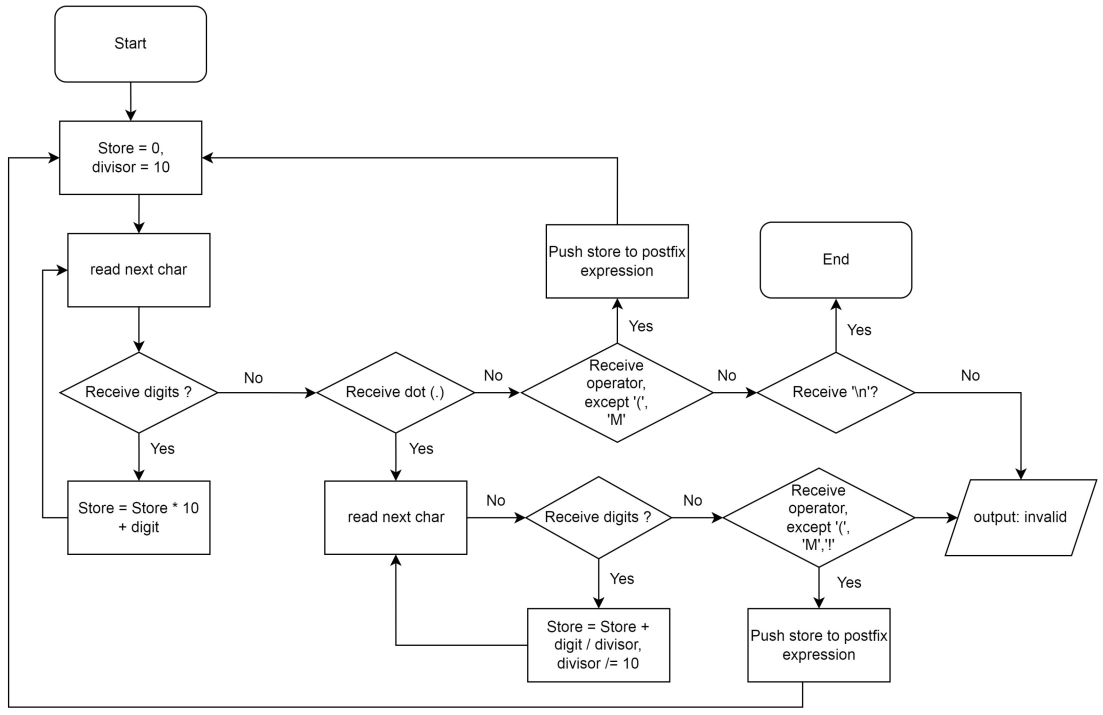
2.1) First we read and iterate the input string (infix expression), my algorithm is describe as follow:



2.2) Next, we handle operands:

- To read 1 number, we do as below. The idea is simple, first we assign 0 to a register ($f20), when we read a char that is a 'digit', we multiply $f20 by 10 (store in $f4) and add the value to $f20. For example: with "26", first iterate is 0 *10 + 2 = 2 and second iterate is 2 * 10 + 6.

- To read the value after a decimal point (.). We do as follow, the first number after the dot will divide by 10, the second will divide by 100, third is 1000,… And we add each to $f20 (store the int value). If before the decimal point is not a number then it is invalid input. For example, we have 26.34, with each iterator, we will have "26 + 3/10 = 26.3" and next is "26.3 + 4/100 = 26.34".

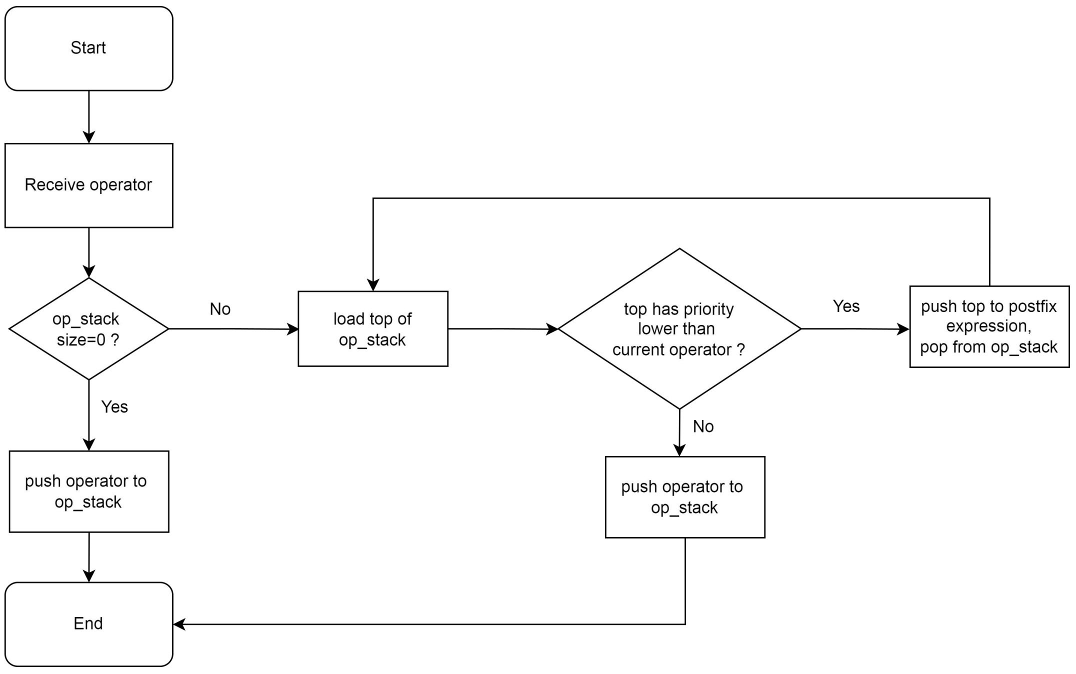

Start
Store = 0, divisor = 10
read next char
Receive digits ?
No
Yes
Store = Store * 10 + digit
Receive dot (.)
No
Yes
read next char
No
Receive digits ?
Yes
Store = Store + digit / divisor, divisor /= 10
No
Receive operator, except '(', 'M','!'
Yes
Push store to postfix expression
output: invalid
Receive operator, except '(', 'M'
Yes
Push store to postfix expression
No
Receive '\n'?
Yes
End
No

2.3) Handle operators:

- Next is the operator, we push back the operator to the operator stack if it has nothing, else we pop from the operator stack to the **post-fix** expression until it is empty or the top of stack has lower priority than the operator we have (like + - lower than * and /) and then we push back. In addition, because before a operator can be a number or a '(', so we check if it is a number, we push it to the **post-fix** expression too. We do the same with all operator, the following table is the priority table of each operator.

Operator	**Priority**
(	0
+ , -	1
* , /	2
^	3

- If before the operator is '(', 'another operator', 'nothing', '.') so it will be invalid input. In addition, we check if before it is a ')' so we don't need to push the number store in $f20 to **post-fix** again (to avoid push number 0).


Start
Receive operator
op_stack size=0 ?
No
Yes
load top of op_stack
top has priority lower than current operator ?
Yes
No
push top to postfix expression, pop from op_stack
push operator to op_stack
push operator to op_stack
End

- With ‘!’ (Factorization), because we can not factorize the decimal value, so we have to check the number before ! if it <0 or not, and then check whether it is a int or double, and if it is int, we factorize it and store to $f20, otherwise it will be invalid_input.

- To deal with ‘(‘ ‘)’, the idea is very simple, if we meet ‘(’, push it back to the operator stack, else if we meet ‘)’, we pop the operator stack and push into the **post-fix** expression until we meet ‘(‘, and then we remove the ‘(’ out of the operator stack.

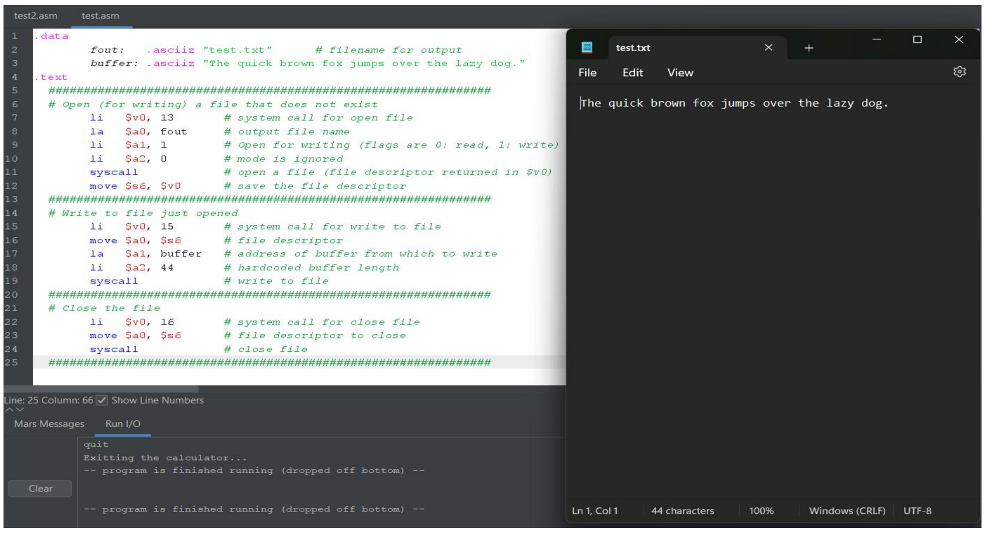
- So we have cover all steps to convert infix expression to **post-fix**, now we calculate base on the **post-fix**. We will first initialize a stack, we iterate the expression from left to right and keep on storing the numbers into a stack. Once an operator is received, pop the two topmost numbers and evaluate them based on that operator and push the result in the stack again.

- About the memory after each run ‘M’, we store the result to a register $fi and in the next run, if we meet character ‘M’ ,we assign that $fi to $f20.

- For the illustration of evaluating the **post-fix** expression, please refer it here: Link

2.4) Print the result to output file:

- After all, we will print the result to a log file called " calc_log.txt". To do this, we will use some function of system call, that is $v0 = 13 (to open file) and $v0 = 15 (to write). The hardest thing is print the result number, to do that, we have to iterate through the result and convert it back to character (because the syscall can only print character). Below is an example:



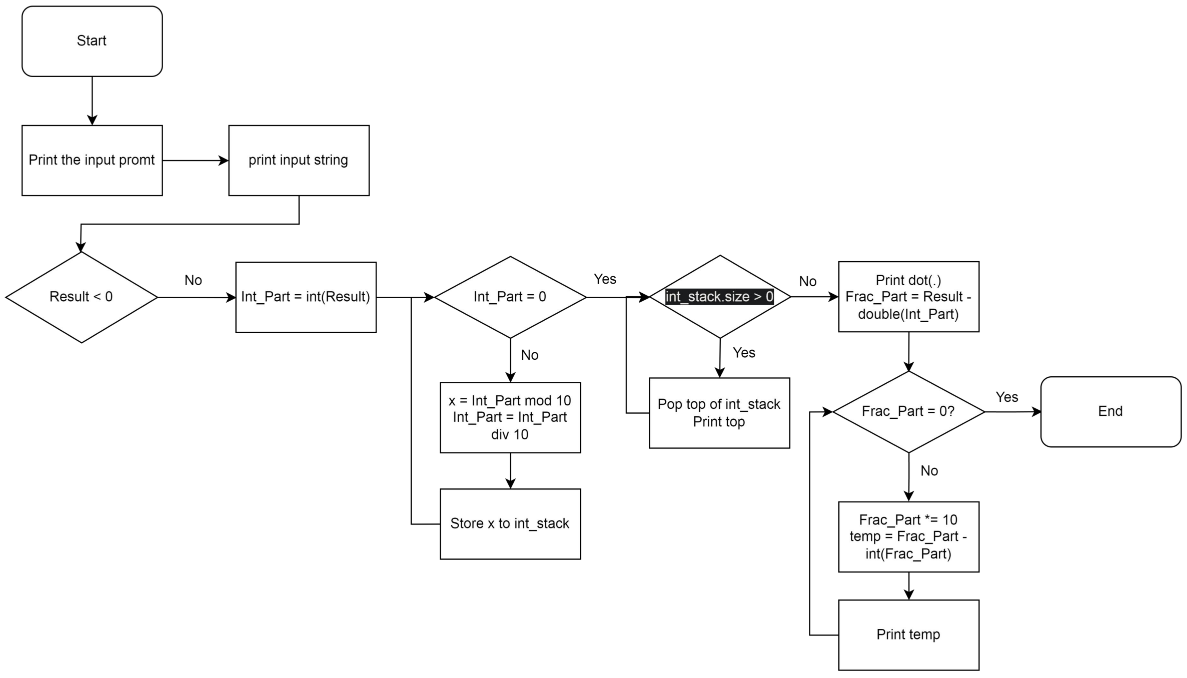
```
.data
        fout:   .asciiz "test.txt"      # filename for output
        buffer: .asciiz "The quick brown fox jumps over the lazy dog."
.text
  ################################################################
  # Open (for writing) a file that does not exist
        li   $v0, 13        # system call for open file
        la   $a0, fout      # output file name
        li   $a1, 1         # Open for writing (flags are 0: read, 1: write)
        li   $a2, 0         # mode is ignored
        syscall             # open a file (file descriptor returned in $v0)
        move $s6, $v0       # save the file descriptor
  ################################################################
  # Write to file just opened
        li   $v0, 15        # system call for write to file
        move $a0, $s6       # file descriptor
        la   $a1, buffer    # address of buffer from which to write
        li   $a2, 44        # hardcoded buffer length
        syscall             # write to file
  ################################################################
  # Close the file
        li   $v0, 16        # system call for close file
        move $a0, $s6       # file descriptor to close
        syscall             # close file
  ################################################################
```

Line: 25 Column: 66 ✓ Show Line Numbers

```
quit
Exitting the calculator...
-- program is finished running (dropped off bottom) --

-- program is finished running (dropped off bottom) --
```

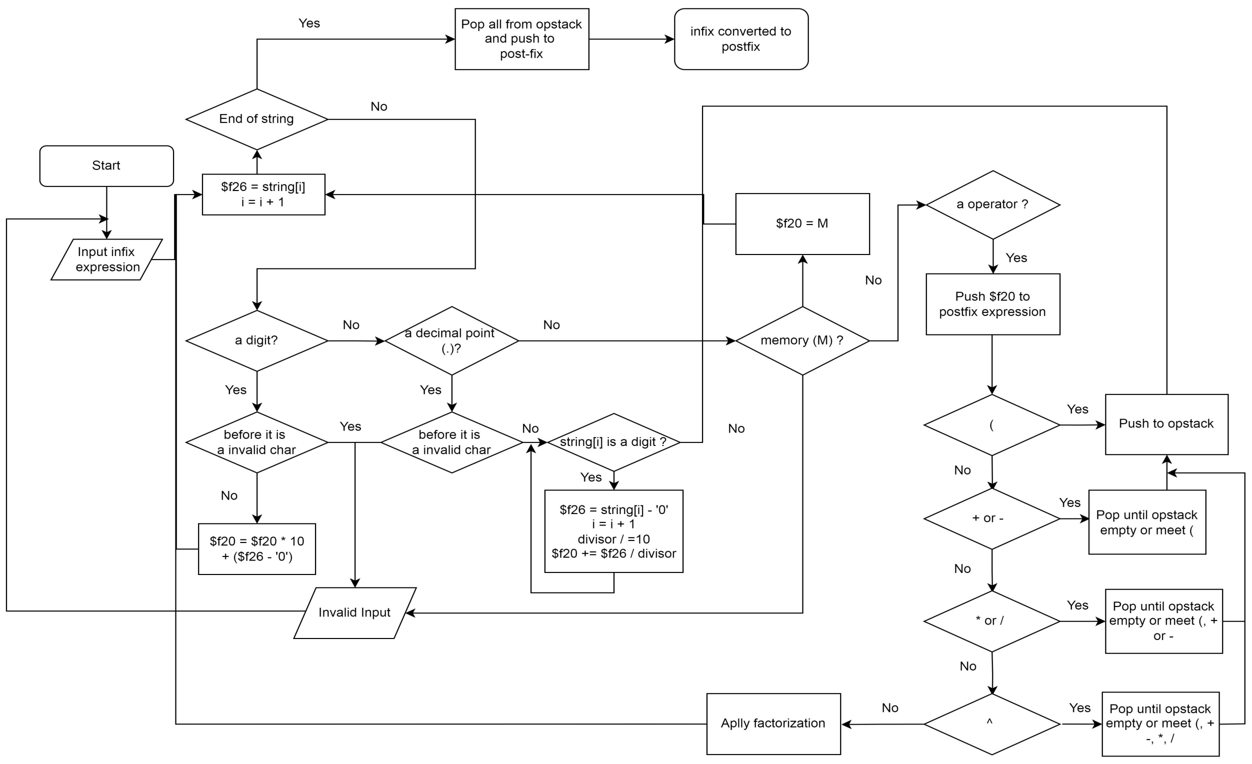
```
The quick brown fox jumps over the lazy dog.
```

Ln 1, Col 1 | 44 characters | 100% | Windows (CRLF) | UTF-8

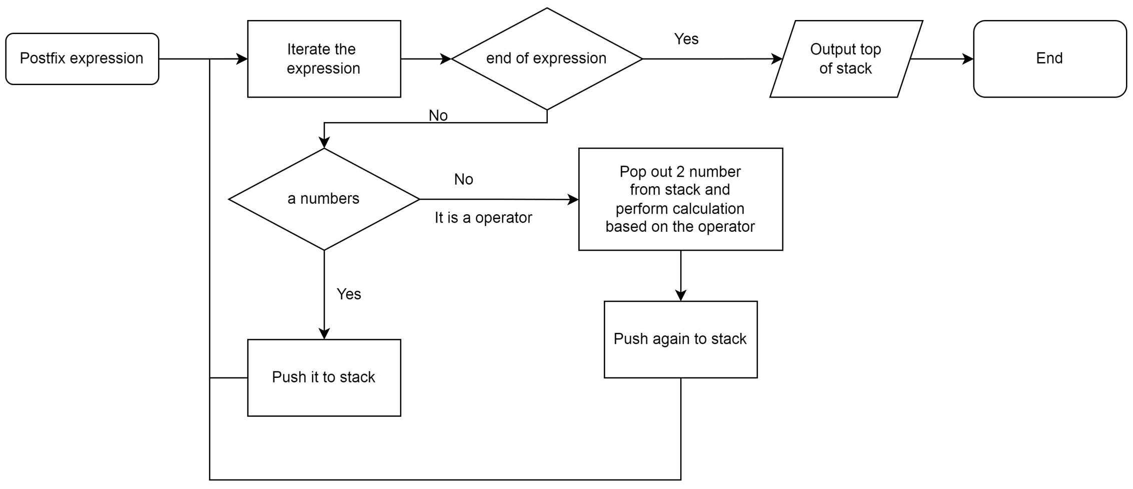


3. Flowchart:

- Convert infix to **post-fix**:

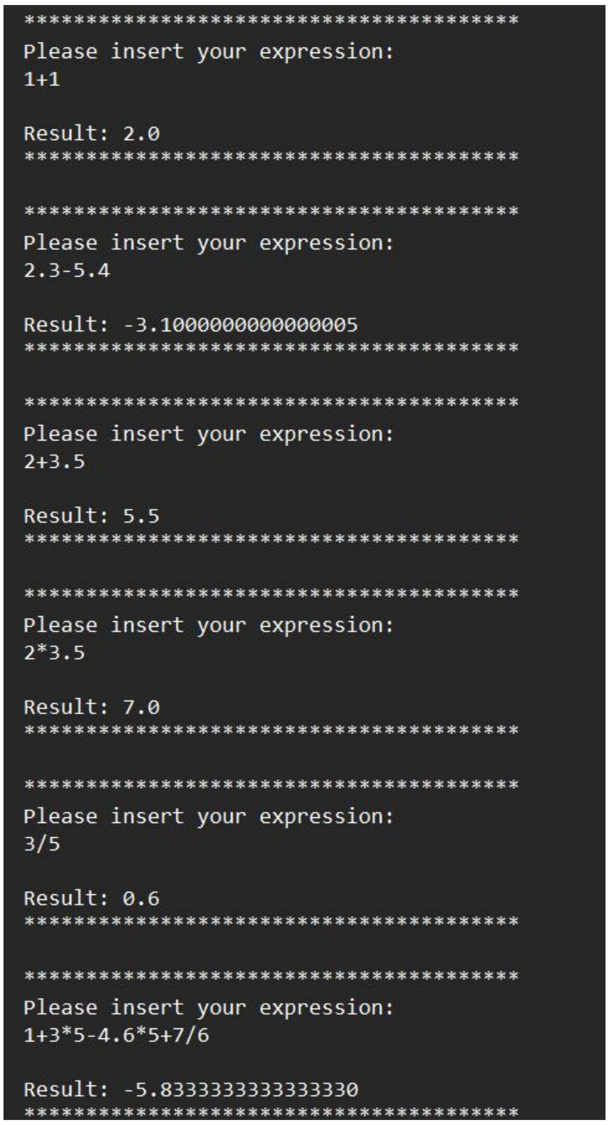


- Calculate **post-fix:**

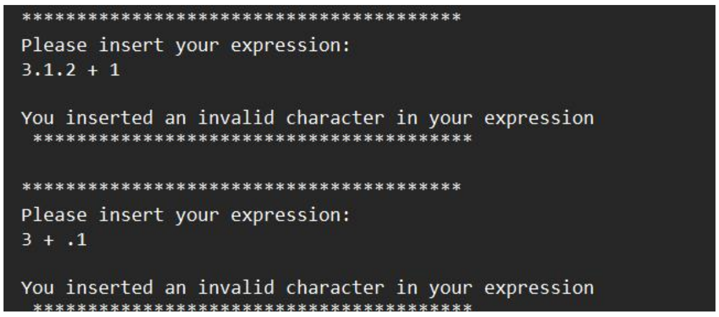


III. Test-cases:

a) Basic math +, -, *, /

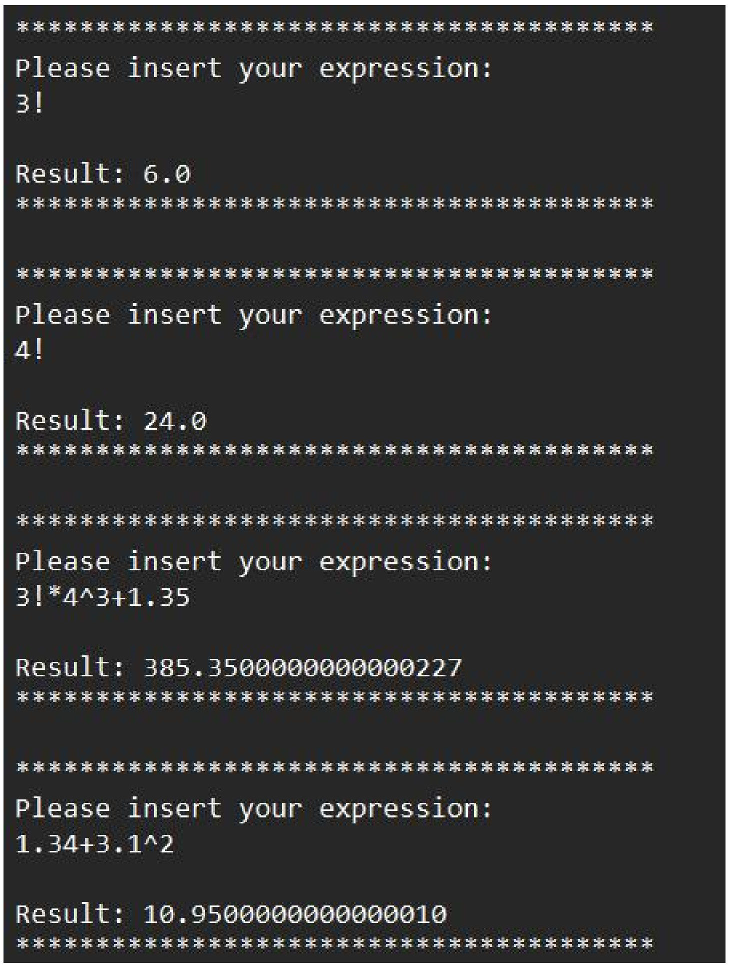


```
*****************************************
Please insert your expression:
1+1

Result: 2.0
*****************************************

*****************************************
Please insert your expression:
2.3-5.4

Result: -3.1000000000000005
*****************************************

*****************************************
Please insert your expression:
2+3.5

Result: 5.5
*****************************************

*****************************************
Please insert your expression:
2*3.5

Result: 7.0
*****************************************

*****************************************
Please insert your expression:
3/5

Result: 0.6
*****************************************

*****************************************
Please insert your expression:
1+3*5-4.6*5+7/6

Result: -5.8333333333333330
*****************************************
```

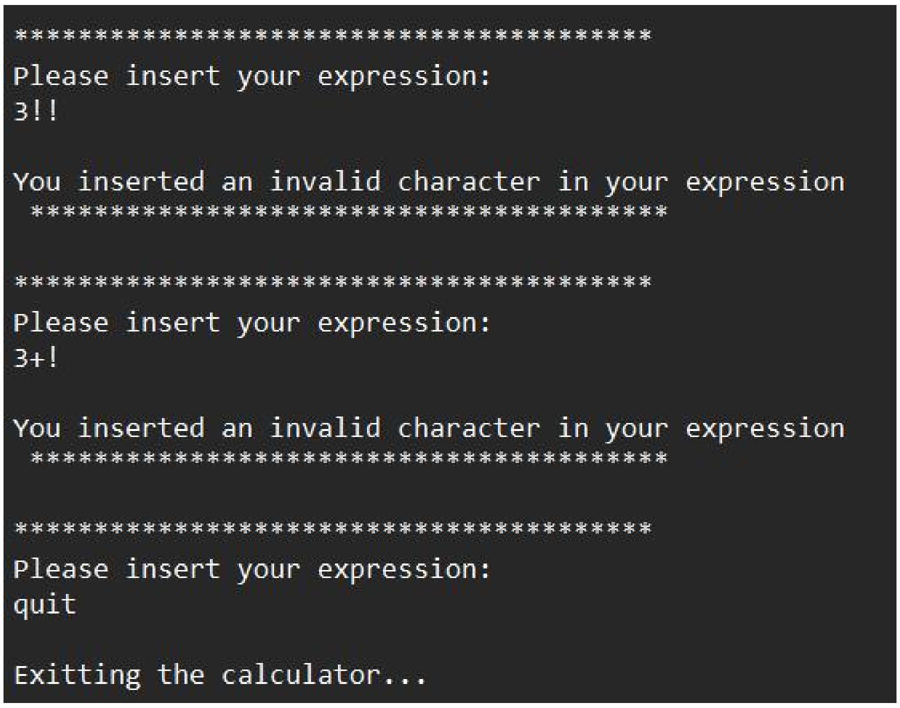


```
*****************************************
Please insert your expression:
3.1.2 + 1

You inserted an invalid character in your expression
 *****************************************

*****************************************
Please insert your expression:
3 + .1

You inserted an invalid character in your expression
 *****************************************
```

b) Add ! and ^ into expression:

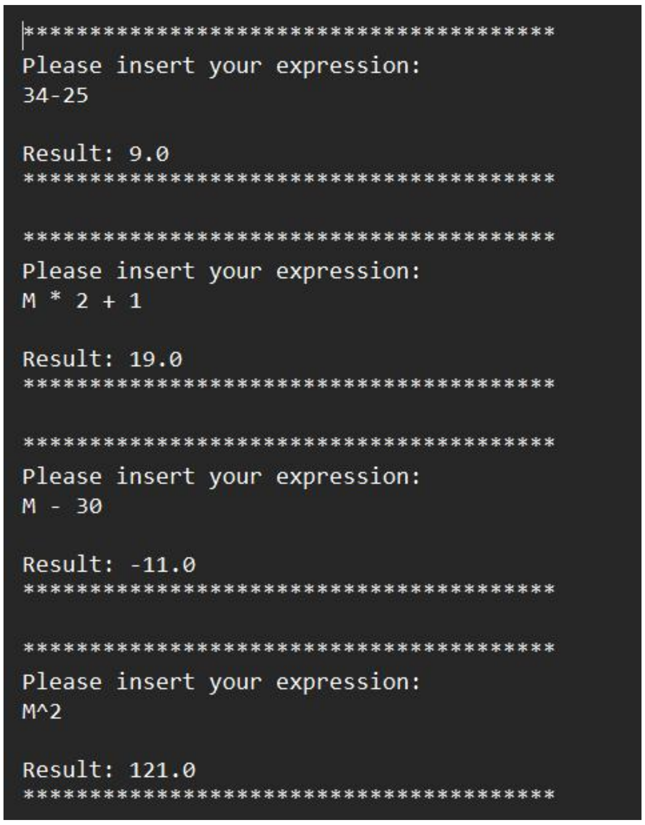


```
****************************************
Please insert your expression:
3!

Result: 6.0
****************************************

****************************************
Please insert your expression:
4!

Result: 24.0
****************************************

****************************************
Please insert your expression:
3!*4^3+1.35

Result: 385.3500000000000227
****************************************

****************************************
Please insert your expression:
1.34+3.1^2

Result: 10.9500000000000010
****************************************
```

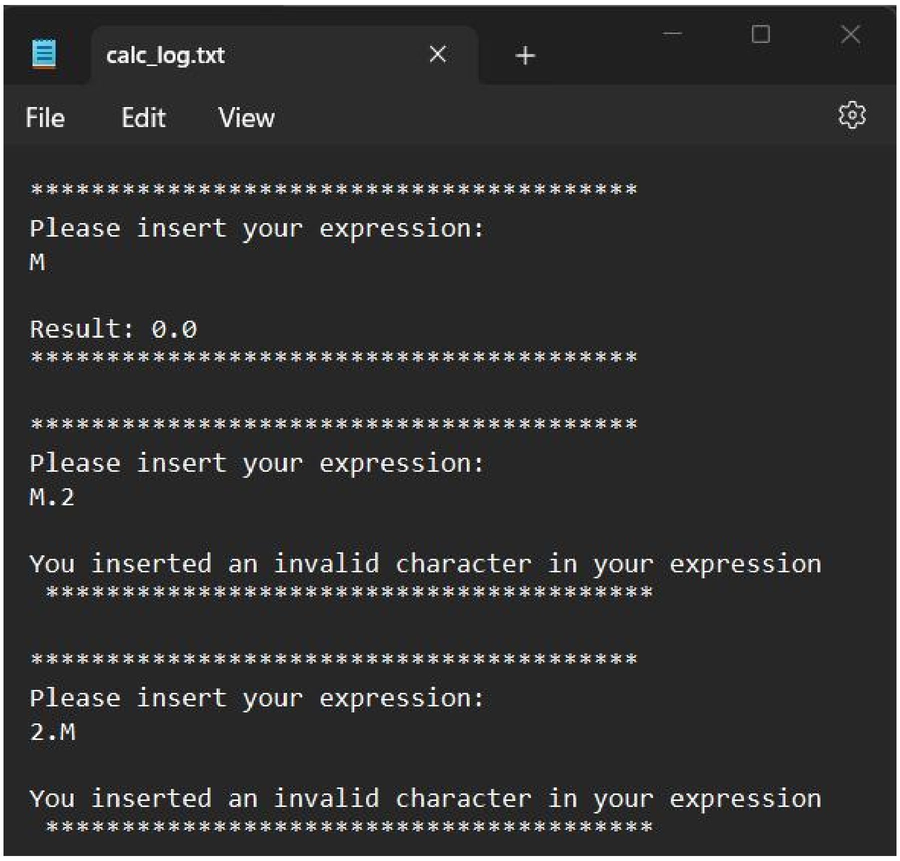


```
****************************************
Please insert your expression:
3!!

You inserted an invalid character in your expression
 ****************************************

****************************************
Please insert your expression:
3+!

You inserted an invalid character in your expression
 ****************************************

****************************************
Please insert your expression:
quit

Exitting the calculator...
```

c) Memory (M):

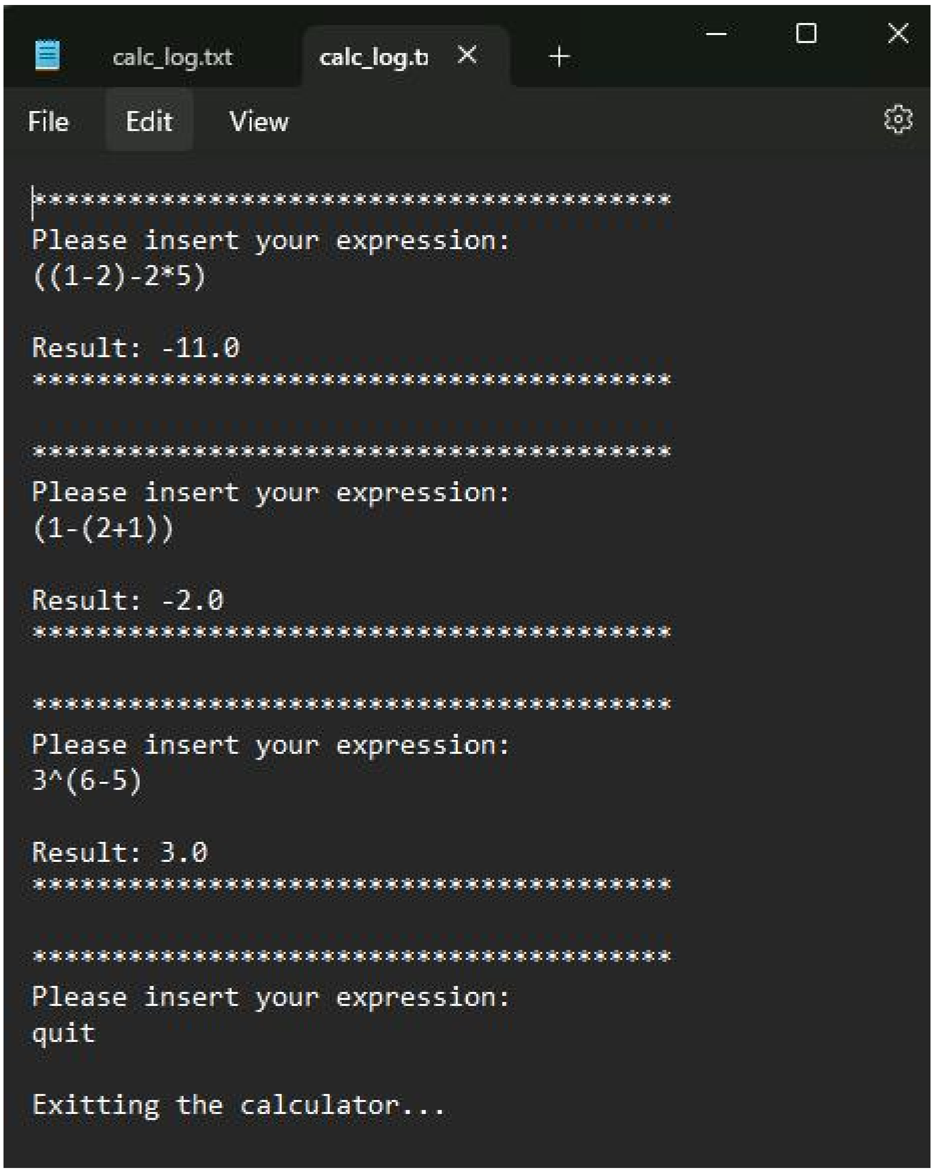


```
****************************************
Please insert your expression:
34-25

Result: 9.0
****************************************

****************************************
Please insert your expression:
M * 2 + 1

Result: 19.0
****************************************

****************************************
Please insert your expression:
M - 30

Result: -11.0
****************************************

****************************************
Please insert your expression:
M^2

Result: 121.0
****************************************
```

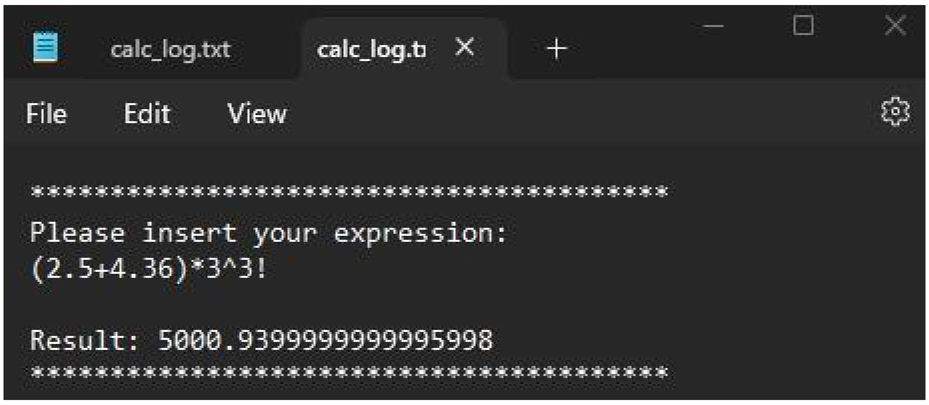


```
****************************************
Please insert your expression:
M

Result: 0.0
****************************************

****************************************
Please insert your expression:
M.2

You inserted an invalid character in your expression
 ****************************************

****************************************
Please insert your expression:
2.M

You inserted an invalid character in your expression
 ****************************************
```

d) Open and close bracket:



```
*****************************************
Please insert your expression:
((1-2)-2*5)

Result: -11.0
*****************************************

*****************************************
Please insert your expression:
(1-(2+1))

Result: -2.0
*****************************************

*****************************************
Please insert your expression:
3^(6-5)

Result: 3.0
*****************************************

*****************************************
Please insert your expression:
quit

Exitting the calculator...
```



```
*****************************************
Please insert your expression:
(2.5+4.36)*3^3!

Result: 5000.9399999999995998
*****************************************
```

IV. Observation and expansion:

- Through the project, I have a good understanding about :

• MARS MIPS simulator.

• Arithmetic & data transfer instructions.

• Conditional branch and unconditional jump instructions.

• Procedures

- Addition, I can understand the way the computer or the compiler read the expression that we input, it just read from left to right or right to left and how to implement the infix to post-fix translator for the computer to perform arithmetic operation.

- How ever, beside that, I also found multiple improvement that I can make to make my project become cleaner. Those as follow:

+ My ability to control the register is not good, I use a lot of register and forgot that some can be reuse, also I don’t arrange those reg in a best way I feel.

+ I can make some function that will do for me, for example, I can make a function to print out to the log file whenever I pass an address of a string to it. Moreover, I could make a function that will return value when I pass 2 numbers and 1 operator code to it, in my project I only just carry all cases.

- Because I didn't have much time, I didn't do enough research to make my project perfect. I apologize for the inconvenience.

V. Reference:

1) https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/

2) https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

3) https://www.geeksforgeeks.org/evaluation-of-postfix-expression/