VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING


# COMPUTER ARCHITECTURE (CO2007)

---

**Computer Architecture Assignment - Semester 232**

## CALCULATOR

---

Advisor: MSc. Nguyễn Thiên Ân
Students: Hà Tường Nguyên 2250013

HO CHI MINH CITY, MAY 2024

# Contents

# List of Figures



# List of Tables



# List of Custom Code Environments

# 0 Abstraction

This report encapsulates the design, development, and evaluation of the MIPS32 assembly program that emulates a versatile calculator[1]. The program's core functionalities encompass basic arithmetic operations, error handling, memory management, and log file generation. Users can input mathematical expressions through a user-friendly interface, receiving accurate results promptly. The calculator supports arithmetic operations such as *addition*, *subtraction*, *multiplication*, and *division*, along with advanced functions like *factorization* and *exponentiation*. Error detection mechanisms ensure the integrity of user input, while *memory functionality* stores the last result for reuse. Decimal numbers are handled with precision, and a comprehensive *log file* captures user interactions and calculated outcomes for future reference.


**Figure 1:** *Casio calculator*

The report delves into the **algorithmic strategies** employed, the **methodologies of implementation**, and the **limitations** inherent to this program. Visual aids such as flowcharts and test case outcomes supplement the narrative, providing clarity and insight into the program's inner workings. The submission adheres rigorously to the evaluation rubric, emphasizing interface friendliness, application functionality, and the quality of the accompanying report. Plagiarism concerns are addressed through originality and adherence to academic integrity standards, ensuring the integrity of the submitted work.

# 1 User Guide and Application Interface

## 1.1 Run On Terminal

This program uses ANSII color codes for display, so it is recommended to run it on Terminal rather than `Mars45` software [1.2] for the best experience. To run it on the terminal, use the following command:

```
>>> spim mar.asm
```

**Listing 1:** *Running the program on Terminal*

After using the above command, there will be a vibrant and user-friendly colored frame providing usage instructions [2]. Information in the frame includes the author, last updated date, input instructions, and some notes.

```
└ spim mar.asm
Loaded: /usr/local/Cellar/spim/9.1.24/share/exceptions.s
#########################################################################################################
#   Author: Ha Tuong Nguyen a.k.a nguyenpanda                                                           #
#   ID:     2250013                                                                                     #
#   Date:   2024-05-09                                                                                  #
#   Ho Chi Minh University of Technology                                                                #
#                                                                                                       #
#    WELCOME TO MY CALCULATOR WRITTEN IN MIPS32 ASSEMBLY LANGUAGE                                       #
#      A simple calculator that can evaluate infix expression by converting it to postfix expression    #
#                                                                                                       #
#   Expression can contain:                                                                             #
#     - Number: 0-9                                                                                     #
#     - Operator: +, -, *, /, ^, !                                                                     #
#     - Parentheses: (, )                                                                               #
#     - Space: ' ' (this is optional becase the calculator will remove all spaces before evaluating)    #
#     - M: store the result of the last expression                                                      #
#     - `quit`: quit the program                                                                        #
#                                                                                                       #
#   Example:                                                                                            #
#     >>> (((-10.2-3)*8-2/7)*2-M!*2^7)*(-1)                                                           #
#     >>> 1 + 2 * 3 - 4 / 5                                                                           #
#     >>> 1 + 2 * 3 - 4 / 5 ^ 6                                                                       #
#                                                                                                       #
#   Note:                                                                                               #
#     - This program contains a lot of comments to help you understand the code                         #
#########################################################################################################
#########################################################################################################
#                                                                                                       #
#   Remember expression only only contain: 0-9, +, -, *, /, ^, !, (, ), space, `M`, `quit`             #
#   Type `quit` to END the program                                                                      #
#   Result will be display on terminal and log out `calc_log.txt`                                       #
#                                                                                                       #
#########################################################################################################
Please insert your expression:
```

**Figure 2:** *Starting the program*

When entering an expression and pressing enter, the software will calculate and display the result of the operation. Simultaneously, it will display a red instruction frame for the next steps for the user. Meanwhile, the value of that operation will be saved in 'M' and written to the file 'calc_log.txt' with *16 decimal places* after the decimal point.


**Figure 3:** *After entering the first expression*

If the user enters 'quit', the program will terminate and notify the user that they have entered 'quit', ending the program.

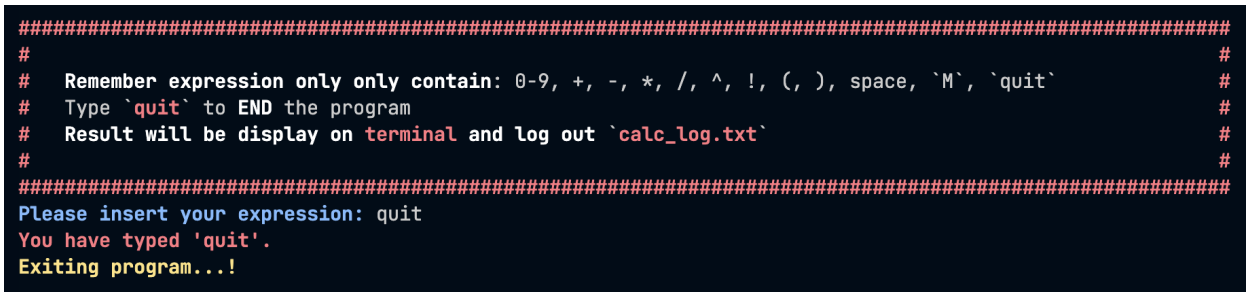


**Figure 4:** *User enter 'quit'*

## 1.2 Run On Mars45

If you prefer not to run it on the terminal, you can run it on *Mars45*. This is done very simply by downloading the *Mars45* software from this link [Missouri State University]. After downloading, open the file and then click `Assemble` → `Run`.

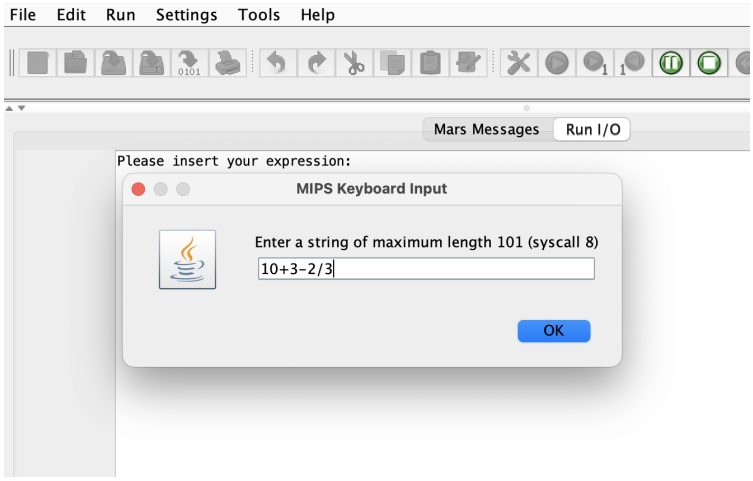


**Figure 5:** *Starting the program on Mars45*

However, using *Mars45* will not display the vibrant colored instruction frame because *Mars45* does not support ANSII color-coded character output. This can be easily overcome by going to Settings and enabling '`Popup dialog for input syscalls`'. When running, you will see a dialog box and the result on '`Run I/O`'.

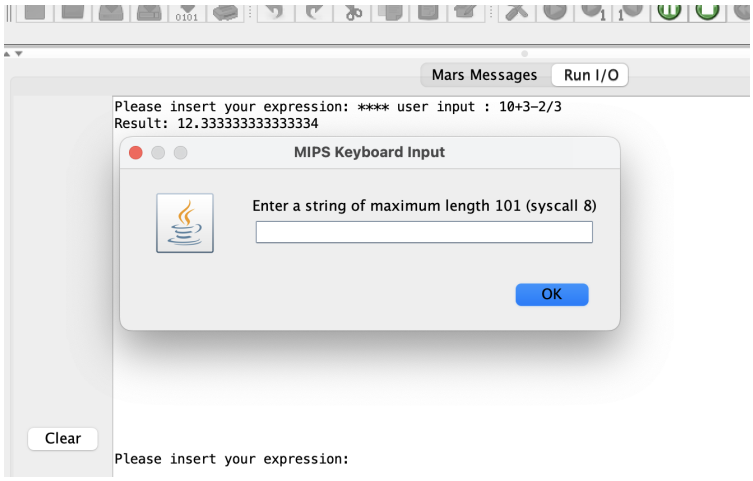


**Figure 6:** *After entering the expression on Mars45*

# 2 Algorithm and Pseudo-code

I utilized Python code as a '*pseudo-code*' for both MIPS32 Assembly and to validate algorithms. The following code section will be divided into three parts.

## 2.1 Support Functions

The support functions below serve essential roles in facilitating the functionality of the main calculator algorithm.

- **fact(n: int) → int**: Calculating the factorial of $n$. If $n < 0$ or $n > 16$, the function will raise an exception.
- **isOperand(char: str) → bool**: Checks if a character is an operand (numeric digit '0–9', decimal point '.', or memory indicator 'M').
- **isOperator(char: str) → bool**: Checks if a character is an operator ($+$, $-$, $*$, $/$, $\hat{}$, !).
- **precedence(char: str) → int**: Determines the precedence of an operator.

These functions collectively contribute to properly parsing and evaluating mathematical expressions within the calculator program. The `fact` function computes factorials, ensuring the input is within acceptable bounds to prevent overflow. `isOperand` and `isOperator` ascertain whether a given character is an operand or an operator, respectively. Finally, `precedence` determines the precedence level of operators, aiding in correct expression evaluation.

```python
def fact(n: int) -> int:
    if n < 0:
        raise ValueError(f'Factorial of negative number, got n {n}')
    if n > 15:
        raise ValueError(f'Factorial of {n} is overflow in 32-bits')
    return 1 if n == 0 else n * fact(n - 1)

def isOperand(char: str) -> bool:
    return char.isdigit() or char == '.' or char == 'M'

def isOperator(char: str) -> bool:
    return char in ('+', '-', '*', '/', '^', '!')

def precedence(char: str) -> int:
    if char == '+' or char == '-': return 1
```

```
    if char == '*' or char == '/': return 2
    if char == '^': return 3
    if char == '!': return 4
    return -1
```

**Listing 2:** *Support function in Python*

## 2.2 Converting Infix to Postfix Algorithm

The following Python code implements an algorithm to convert infix expressions to postfix notation This algorithm scans the input expression character by character and applies specific rules to convert it to postfix notation. It handles operands, operators, and parentheses, and ensures the correct order of operations. Additionally, it accounts for negative numbers, unary minus, unary plus, and invalid characters, raising appropriate exceptions when encountered.

```
def infixToPostfix(string: str, M: int):
    if len(string) == 0 or len(string) > 100:
        raise ValueError('Expression length must in range [0, 100], got
        length', len(string))

    stack: list[str] = []
    result: str = ''

    for char in string:
        if (isOperand(char)): result += char
        elif char == ' ': continue
        elif char == '(': stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(': result += f' {stack.pop()}'
            if stack and stack[-1] == '(': stack.pop()
        elif isOperator(char):
            if char == '!':
                result += ' !'
                continue

            if (char == '-'
                and (not result or result[-1] in ('(', ' ', '*', '/', '^'))
                and (not stack or stack[-1] in ('(', ' ', '*', '/', '^'))
```

```
                ):
                result += char
                continue

            while (stack
                   and stack[-1] != '('
                   and precedence(stack[-1]) >= precedence(char)):
                result += f' {stack.pop()}'

            stack.append(char)
            result += ' '
        else:
            raise Exception('Invalid character')

    while stack: result += f' {stack.pop()}'

    return result
```

**Listing 3:** *Converting infix to postfix expression*

## 2.3 Evaluating Postfix Algorithm

The following Python code implements an algorithm to evaluate postfix expressions. This algorithm iterates through the postfix expression, performing arithmetic operations and utilizing a stack to store intermediate results. It supports operands, operators, and memory variable 'M'. Additionally, it handles negative numbers and ensures proper expression evaluation. If the expression is invalid or cannot be evaluated, appropriate exceptions are raised.

```
def evaluate_postfix(expression: str, M: int):
    stack: list[float] = []
    number: str = ''

    for char in expression:
        if char == 'M':
            number += str(M)
        elif isOperand(char) or (char == '-' and len(stack) < 2):
            number += char
        elif char == ' ':
```

```
            if number:
                if number == '-':
                    stack.append(stack.pop() - operand2)
                    number = ''
                else:
                    stack.append(float(number))
                    number = ''
        else:
            if number:
                stack.append(float(number))
                number = ''

            operand2 = stack.pop()
            if char == '+':
                stack.append(stack.pop() + operand2)
            elif char == '-':
                stack.append(stack.pop() - operand2)
            elif char == '*':
                stack.append(stack.pop() * operand2)
            elif char == '/':
                stack.append(stack.pop() / operand2)
            elif char == '!':
                stack.append(fact(int(operand2)))
            elif char == '^':
                operand2 = stack.pop()
                stack.append(stack.pop() ** int(operand2))
            else:
                raise ValueError("Invalid token in expression")

    if number:
        stack.append(float(number))

    if len(stack) != 1:
        raise ValueError('Invalid postfix expression')

    return stack.pop()
```

**Listing 4:** *Evaluate Postfix*

# 3 Description of Support Procedures

All procedures follow the convention of using $a0-$a3 as arguments and $v0-$v1 as return registers, taken from registers with smaller to larger values. All registers used by the procedures are stored in main memory, so when encountering the `jr $ra` instruction, all registers are restored to their values before the subprocedure calculation.

For functions that need to use registers in Coprocessor 1 (`$f<x>`), here is my convention for register management:

- `Return` : $f0 - $f3
- `Temporary`: $f4 - $f11
- `Argument` : $f12 - $f15
- `Temporary`: $f16 - $f19
- `Constant` : $f24 - $f31

All privates prefixed with double underscores are indications of private procedures, only to be used within their own procedures or when called. The coder should not actively call them out.

## 3.1 Description of character and string handling

The procedures below aid two main procedures, `INFIX_TO_POSTFIX` [2.2] and `EVALUATE_POSTFIX` [2.3], `STACK` [4], as well as other related procedures. They include character checks and string-handling functions

| **Procedure** | **Args** | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| IS_ OPERAND | char = $a0 | bool = $v0 | X | Checking if a character is an operand. |
| IS_ OPERATOR | char = $a0 | bool = $v0 | X | Checking if a character is in (+, -, *, /, ^, !). |
| OPERATOR_ PRECEDENCE | char = $a0 | bool = $v0 | X | Getting the precedence of an operator. |

| | | | | |
|---|---|---|---|---|
| STRING_ LENGTH | str = $a0 | int = $v0 | X | Getting the length of a string by looping through the string until it encounters the null or newline character. This procedure has a time complexity of $O(n)$. |
| COMPARE_ STRING | str = $a0<br>str = $a1 | bool = $v0 | | Comparing two strings by looping through both strings, return true if both strings are the same length and have identical contents. This operation has a time complexity of $O(n)$. |
| RESET_ STRING | str = $a0<br>str = $a0 | void | X | Resetting the space containing the string by looping through and replacing each character with a null character. This operation has a time complexity of $O(n)$. |

**Table 1:** *Description of Support Procedures*

### 3.2 Mathematics procedures

| **Procedure** | **Args** | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| FACTORIAL | int = $a0 | int = $v0 | | Calculating the factorial of an integer using recursion. If the input is greater than 0 and less than 16, raise an exception and end the program. This operation has a time complexity of $O(n)$. |

| | | | | |
|---|---|---|---|---|
| EXPONENT | double = $f12<br>double = $f14 | double = $f0 | X | Calculating the exponent of $f12 ^ $f14 by looping. It automatically floors $f14 into an integer and can calculate if $f14 $< 0$. This operation has a time complexity of $O(\|int(\$f14)\|)$. Notice that if $f12 = $f14 = 0, raise an exception |

**Table 2:** *Description of Mathematics Procedures*

### 3.3 Utility procedures

In this subsection, there will be utility procedures that do not contribute significantly to the algorithms of the `INFIX_TO_POSTFIX` [2.2] and `EVALUATE_POSTFIX` [2.3] procedures. They are used solely for display and assignment requirements.

| **Procedure** | **Args** | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| END_PROGRAM | - | void | X | The exit prompt will be displayed, and the system call $v0 = 10 will be invoked to end the program. |
| TYPE_QUIT | - | void | - | The quit prompt will be displayed, and the `END_PROGRAM` procedure will be called. |
| WRITE_<br>TO_FILE | message = $a0<br>filename = $a1<br>mode = $a2 | void | - | This procedure is used to *write* only with create ($a2=1) or *append* ($a2=9) strings to a file. It automatically measures the length of the string to write to the file by calling the `STRING_LENGTH` procedure and closes the file after writing to it. |

| **Procedure** | **Args** | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| new_line | - | void | X | When printing newline characters to the screen, note that this function will be convenient for code writers but will significantly slow down the program due to unnecessary procedures. |

**Table 3:** *Description of Utility Procedures*

### 3.3.1 Print Procedures

The procedures below are considered leaf procedures supporting screen printing by calling `syscall`.

- `PRINT_DOUBLE`: double = $f12
- `PRINT_CHAR` : char = $a0
- `PRINT_STRING`: string = $a0
- `PRINT_INT` : int = $a0

Example:

```
# Integer
li $a0, 2024
jal PRINT_INT

# Character
li $a0, 97
jal PRINT_CHAR

# Double (The index of Coprocessor must be even)
mov.d $f12, $f8 # Assume $f8 = 09052024.1259
jal PRINT_DOUBLE
```

**Listing 5:** *Color procedures*

### 3.3.2 Color Procedures

All procedures below are leaf procedures, simply using the `jal <COLOR LABEL>` command before and after a print command will suffice.

- `RED`
- `YELLOW`
- `MAGENTA`
- `RESET`
- `GREEN`
- `BLUE`
- `CYAN`

Example:

```
jal CYAN
la $a0, ascii_string # ascii_string: .asciiz "I'm studying CA at BKU\n"
jal PRINT_STRING
jal RESET
```

**Listing 6:** *Color procedures*

# 4 Stack Procedures

The following sub-section only discusses the design architecture of the Stack "*class*", detailed explanations of the procedures will be covered further in the subsequent sub-sections.

## 4.1 Stack Design Architecture

The Stack "*class*" is designed by dividing a memory area `.space` into 3 parts. The first part holds the address of the word after the top of the stack, the next word is an integer indicating the number of elements inserted into the stack, and the remaining part is divided into word-sized regions containing the values of the elements pushed onto the stack [7]. If a Push operation exceeds the memory area, the system will report a stack overflow error, and if a `TOP` or `POP` operation is performed when the stack is empty, the system will report a stack underflow error.

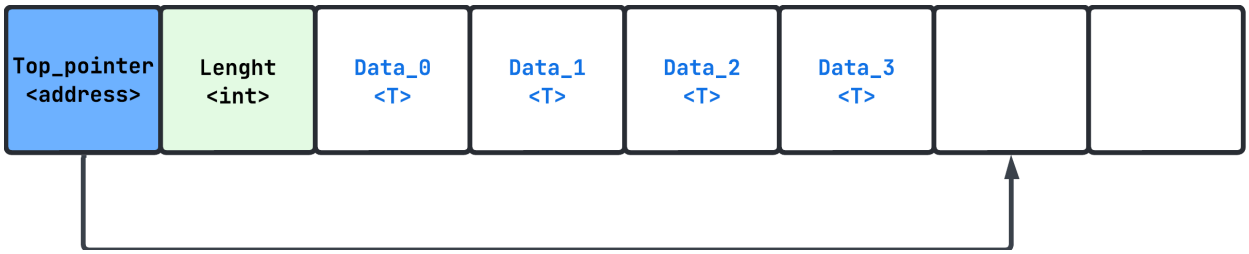


**Figure 7:** *Stack implementation architecture*

An example of how data is organized in a stack:

Let's push the each char *HCMUT* onto the stack sequentially. The result [8] shows that the first word at address `0x10010020` points to the address `0x1001003c`, meaning the word after the `TOP`.

```
la $a0, stack   # Load address of .space

# STACK_PUSH($a0 = address of memory block, $a1 = value) => void
li $a1, H
jal STACK_PUSH
li $a1, C
jal STACK_PUSH
li $a1, M
jal STACK_PUSH
li $a1, U
jal STACK_PUSH
li $a1, T
jal STACK_PUSH
```

```
14
15  la $a1, PRINT_CHAR
16  jal PRINT_STACK
```

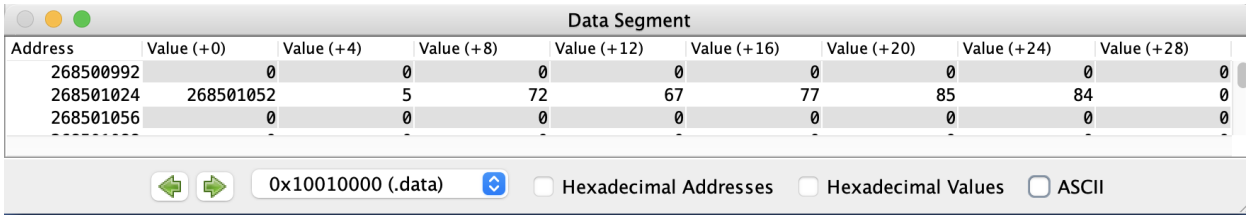


**Figure 8:** *Stack memory architecture*

## 4.2 Stack Procedures API

This sub-section contains all the MIPS32 procedures API for the Stack "*class*". All procedures below use register `$a0` to pass the address of the memory block containing the address of the `Top_pointer` (the blue block in figure [7]). All procedures have time complexity of $O(1)$, except `STACK_RESET` and `STACK_PRINT` have complexity of $O(n)$.

| **Procedure** | **Args** (`$a0` = *memory block address*) | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| STACK_INIT | - | void | X | This procedure **MUST** be called before using a block of memory as a stack in other to set up the `Top_pointer` |
| STACK_ LENGTH | - | int = $v0 | X | Return the number of elements in the stack. |
| STACK_PUSH | <T> = $a1 | void | | Adding the element at the end of the stack. |
| STACK_TOP | - | <T> = $v0 | | Return the value of the last element of the stack |
| STACK_POP | - | <T> = $v0 | | Removing and returning the last element of the stack. |

| **Procedure** | **Args** (`$a0` = *memory block address*) | **Return** | **Leaf fucnt** | **Description** |
|---|---|---|---|---|
| STACK_ PUSH_DOUBLE | double = $f12 | void | | Adding the double values $f12 and $f13 in this order into the stack. |
| STACK_POP_ DOUBLE | - | double = $f0 = $v0 | | Removing and returning the last double value of the stack. |
| STACK_ LENGTH_ DOUBLE | - | int = $v0 | | Return the length of the stack containing double by calling `STACK_LENGTH` and dividing by 2 the values it returns. |
| STACK_ RESET | - | void | X | Pop all elements in stack and reset `Top_pointer` and `Length`. |
| PRINT_STACK | print func = $a1 [3.3.1] | void | | Print all stack properties. The format for displaying on the screen will be: `\|X\|X\|X\|<-TOP (Length=Y)` |

**Table 4:** *Description of Stack Procedures*

## 4.3 Stack Exception Handling

The procedures below are private and will be automatically called when an error occurs.

| **Private Exception** | **Description** |
|---|---|
| __STACK_OVERFLOW | __STACK_OVERFLOW will be called when `Push` more than 25 doubles or 50 other elements into the stack. This involves printing an error message and exiting the program. |

| **Private Exception** | **Description** |
|---|---|
| __STACK_OVERFLOW | __STACK_OVERFLOW will be called when `Pop` or `Top` an empty stack. This involves printing an error message and exiting the program. |

**Table 5:** *Description of Stack Exception*

# 5 Limitations

In this section, I will outline areas where the program can be further improved in the future. These areas are listed from ***major issues to minor ones***.

## 5.1 Handling Unary Operations

My program still has numerous bugs, especially in handling unary operations such as addition and minus. I've tested several cases like multiple consecutive addition or subtraction signs before a number, or a minus sign before a parenthesis, and the program crashes. This occurs due to flaws in my Python algorithm. The following code is the result of both MIPS32 and Python.

```
>>> Please insert your expression: -1*-(3+1)
>>> Result: 2
>>>
>>> Please insert your expression: (-1)*(-1)-1*-1
>>> Error: Invalid postfix expression
>>> Result: -0
>>>
>>> Please insert your expression: -1--2
>>> Error: Invalid postfix expression
>>> Result: -2
>>>
>>> Please insert your expression: -1+-2
>>> PROGRAM IS CRACKING
>>>
>>> Please insert your expression: 1++++++2
>>> PROGRAM IS CRACKING
>>>
>>> Please insert your expression: -1*(3+1
>>> PROGRAM IS CRACKING
```

**Listing 7:** *Failing testcase*

## 5.2 Error Handling

I haven't implemented error handling yet. The program usually freezes [5.1] instead of automatically resetting and restarting the program from the beginning when an error occurs.

### 5.3 Arithmetic Overflow Causing File Output Error

Although the program still produces approximately correct results when computing with large numbers, when exporting the result to a file, it outputs strange characters. This is because there are still many issues with my `DOUBLE_TO_STRING` procedure. An example for this issue:

```
>>> Please insert your expression: 10^100
>>> Result: 1.0000000000000006e+100
In calc_log.txt:
    -./,),(-*,(../,),(-*,(./,),(-*,(
```

**Listing 8:** *Wrong result in .txt file*

### 5.4 String Class Design

I haven't designed a "class" for strings like I did for the "stack" because I had already written string processing functions. Consequently, these procedures all have a complexity of $O(n)$.

### 5.5 Usage of Load and Store Commands

When coding subprocedures, I always use the `sw` and `lw` commands for all `$a<x>` and `$t<x>` registers to make the code and debugging easier. As the theoretical class mentions, `lw` and `sw` commands consume more time than other commands. This leads to the excessive use of subprocedures for easier coding and debugging but at the cost of reducing the program's speed.

### 5.6 Heap Memory Usage

I don't utilize heap memory for allocation; instead, I only use the main memory (stack) for this purpose. Consequently, my program cannot handle cases with excessively large or numerous numbers while running.

## References

[1] John L. Hennessy David A. Patterson. *Computer Organization and Design - The Hardware Software Interface*. Fifth Edition.

[2] Khoa Khoa Học Kỹ Thuật Máy Tính. Hướng dẫn thực hành môn kiến trúc máy tính. *Trường Đại học Bách Khoa - ĐHQG HCM*, pages 3–7, 2016.

[3] Missouri State University. Syscall functions available in mars, 2014. URL https://courses.missouristate.edu/kenvollmar/mars/Help/SyscallHelp.html. Last accessed August 2014.

[4] Wikipedia. Calculator. URL https://en.wikipedia.org/wiki/Calculator.
refs.bib