VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER ARCHITECTURE PROJECT REPORT

MIPS Assembly Calculator Implementation

Teacher in charge: Nguyen Thien An

Full name:	Luu Bao Long
ID number:	2252442

1. Introduction

1. Overview:

In this report, we delve into the development of a calculator using MIPS assembly language, completed as part of the Computer Architecture Lab course at Ho Chi Minh City University of Technology. The assignment challenged us to create a functional calculator capable of basic and complex arithmetic operations, all implemented within the MARS MIPS simulator environment.

The significance of a calculator extends beyond simple arithmetic tasks; it represents a fundamental tool in both educational and professional fields. Historically, the evolution from mechanical to electronic calculators marked a significant technological leap. Today, calculators are integrated into various devices, demonstrating the ongoing relevance and adaptation of calculating tools in technology.

The primary objective of this project was to apply theoretical knowledge of MIPS assembly language to a practical scenario—building a calculator. This not only tests our understanding of assembly language programming but also enhances our problem-solving skills in creating efficient, reliable software.

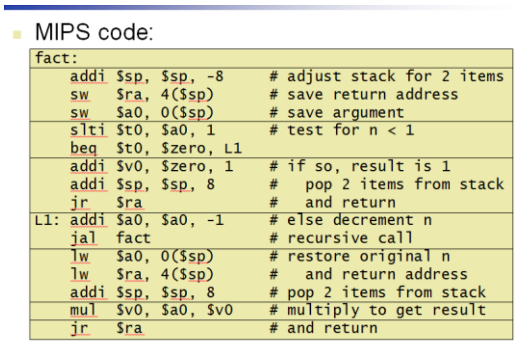
This report outlines the project's requirements, our implementation strategies, the challenges encountered, and the solutions devised to overcome them. Additionally, we discuss the design decisions made to facilitate user interaction with the calculator, ensuring a friendly and intuitive user interface. Through rigorous testing, we demonstrate the calculator's functionality and reliability, ultimately reflecting the successful application of MIPS assembly language in solving real-world problems.

By the conclusion of this report, the reader will gain insights into the complexities of assembly language programming and the meticulous attention to detail required in software development, especially when dealing with low-level programming languages like MIPS assembly.

2. Assignment Outcomes

1. Understanding of MIPS Assembly Language: Students will gain hands-on experience in coding using MIPS assembly language, which is pivotal for understanding the lower-level workings of modern computers. This includes familiarity with syntax, operations, and the environment of MIPS programming.

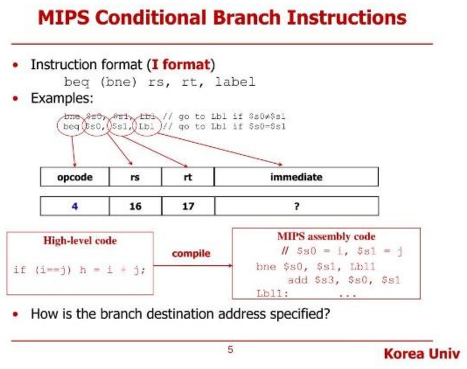
2. Application of Arithmetic and Data Transfer Instructions: The project requires the application of various MIPS instructions for performing arithmetic calculations and data transfers. Students should be able to effectively utilize these instructions to implement the basic functionalities of a calculator.



- MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

3. Proficiency in Handling Conditional Branches and Unconditional Jumps: Through the development of the calculator, students will learn to use conditional and unconditional jump instructions to control the flow of execution within the program, an essential skill for any assembly language programmer.



MIPS Conditional Branch Instructions

- Instruction format (**I format**)
  `beq (bne) rs, rt, label`
- Examples:

```
bne $s0, $s1, Lbl // go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl // go to Lbl if $s0=$s1
```

opcode	rs	rt	immediate
4	16	17	?

High-level code

```
if (i==j) h = i + j;
```

compile

MIPS assembly code

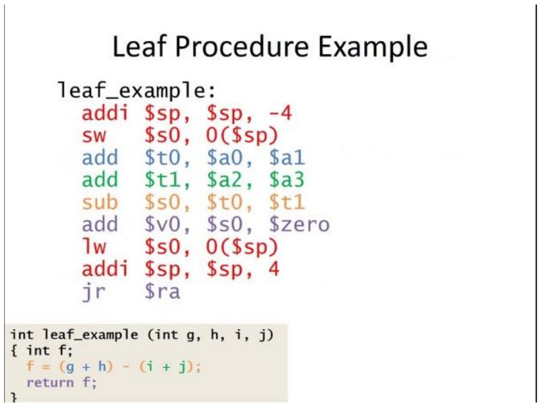
```
    // $s0 = i, $s1 = j
bne $s0, $s1, Lbl1
    add $s3, $s0, $s1
Lbl1:     ...
```

- How is the branch destination address specified?

5

Korea Univ

4. Development of Procedures: Students will develop and use procedures within MIPS to handle repetitive tasks and enhance the modularity of the code. This is crucial for writing clean, maintainable, and reusable code.



Leaf Procedure Example

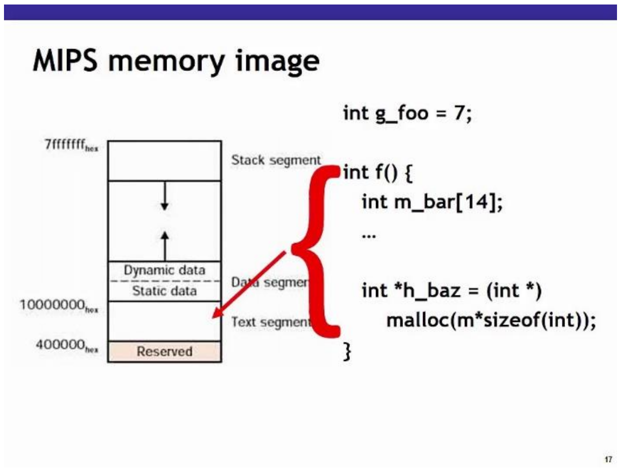
```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

5. Implementation of Complex Arithmetic Functions: The assignment goes beyond basic arithmetic to include functions such as factorization, exponentiation, and operations involving parentheses, thus pushing students to implement more complex algorithms in assembly language.

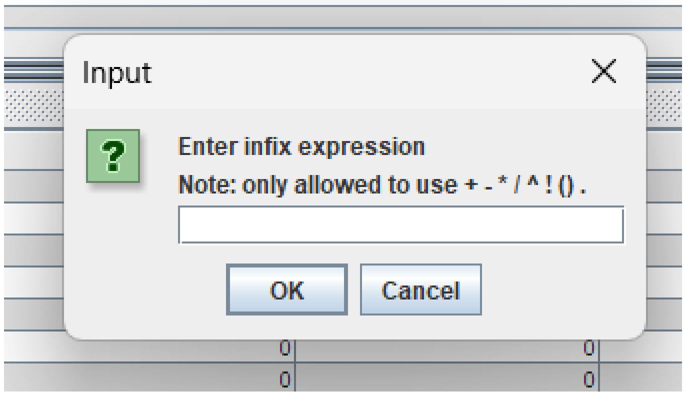
6. Error Handling: Students will learn to anticipate and manage potential errors in user input, providing robustness in the calculator's functionality.

7. Memory Management: Through implementing features like memory storage of calculations, students will handle dynamic data storage within MIPS, an important aspect of programming and computer architecture.



```
int g_foo = 7;

int f() {
  int m_bar[14];
  ...

  int *h_baz = (int *)
     malloc(m*sizeof(int));
}
```

8. Interactive User Interface Design: Designing an interface that guides the user clearly through their interactions with the calculator is another crucial learning outcome, ensuring that the software is user-friendly and functional.



Enter infix expression
Note: only allowed to use + - * / ^ ! () .

3. Requirements Analysis

This section outlines the detailed functional requirements for the MIPS assembly calculator designed in this project. The calculator is expected to handle various types of arithmetic operations, ensure correct input format, and display results accurately.

3.1. Input

- Format and Constraints: The program accepts a string of arithmetic expressions from the user, which should not exceed 100 characters. This allows users to input complex calculations within a reasonable length.
- Validation: The input must be validated to ensure that only permissible characters are used. Valid characters include digits (0-9), basic arithmetic operators (+, -, *, /), factorials (!),

exponents (^), parentheses (), and the memory recall variable (M). Any input containing invalid characters triggers a prompt to the user indicating the error.

3.2. Output

- Display of Results: After processing the input expression, the output, which is the result of the calculation, is displayed to the user. This includes results from basic arithmetic operations, as well as more complex functions like factorials and exponents.
- Handling Infinite Decimals: In cases where the result is an infinite decimal, the calculator will display the result up to 16 decimal places, ensuring precision without overwhelming the user with digits.

3.3. Operations

The calculator must support the following operations, each with specific requirements:

- Addition (+): Handles two operands and adds them together. It is a binary operation requiring the presence of two numeric values on either side of the '+' operator.
- Subtraction (-): Can act as a binary operator (subtracting one number from another) or a unary operator (denoting a negative number). Proper handling of these cases is essential for accurate calculations.
- Multiplication (*): Multiplies two numbers. This operation is straightforward but must be efficiently handled to manage potentially large numbers.
- Division (/): Divides one number by another. The program must handle division by zero errors gracefully, either by prompting an error message or managing an exception in some manner.
- Factorial (!): Calculates the factorial of a number. This is a unary operation that requires the program to handle only positive integers as valid inputs for this function.
- Exponent (^): Raises a number to the power of another. This binary operation requires careful implementation to handle large exponents without performance degradation.

4. Implementation

The implementation of the MIPS assembly calculator was centered on the efficient conversion of user-inputted arithmetic expressions from infix to postfix notation, followed by the evaluation of these postfix expressions using a stack-based method. This two-step approach ensures that the order of operations is correctly handled and simplifies the computational process.

4.1 Handling Input Strings: Conversion to Numeric and Operator Stacks

Pseudocode:

```
function StringToNumber(string)
    Initialize integerPart to 0
    Initialize decimalPart to 0
    Initialize isDecimal to false
```

```
   Initialize decimalPlace to 1

   for each character in string
      if character is '.'
         isDecimal = true
         continue
      if isDecimal is false
         integerPart = integerPart * 10 + (character - '0')
      else
         decimalPlace = decimalPlace * 10
         decimalPart = decimalPart * 10 + (character - '0')

   if isDecimal
      totalNumber = integerPart + (decimalPart / decimalPlace)
   else
      totalNumber = integerPart

   return totalNumber
```

4.2 Infix to Postfix Conversion

Pseudocode:

```
function InfixToPostfix(infixExpression)
   Initialize an empty stack for operators
   Initialize an empty list for output (postfix expression)
   for each token in infixExpression
      if token is a number
         add token to output list
      else if token is an operator
         while there is an operator at the top of the stack with higher precedence
            pop operators from the stack to the output list
         push the token to the stack
      else if token is '('
         push token to the stack
      else if token is ')'
         while the operator at the top of the stack is not '('
            pop operators from the stack to the output list
         pop '(' from the stack
   while the stack is not empty
      pop operators from the stack to the output list
   return output list
```

- Example:
    - Input String: "23 + 42 * 3"
    - Parsing Result: Number Stack = [23, 42, 3], Operator Stack = [+, *]
    - Postfix notation: "23 42 3 * +"

4.3 Evaluation of Postfix Expressions

- Stack-Based Evaluation: Once the expression is in postfix notation, a second stack is used for the evaluation. Numbers are pushed onto the stack as they are read, and when an operator is read, the requisite number of operands (two for binary operations like addition, and one for unary operations like factorial) are popped from the stack, the operation is performed, and the result is pushed back onto the stack.
- Handling Different Operations: Each operator triggers a different computation strategy. For example, addition pops two operands and pushes their sum, while exponentiation pops the base and exponent, computes the power, and pushes the result.
- Efficiency Considerations: Using a stack for the evaluation of postfix expressions is efficient in terms of both time and space complexity. Operations are performed as soon as they are read, and no backtracking is necessary, which optimizes the calculation process.

Pseudocode:

```
function EvaluatePostfix(postfixExpression)
    Initialize an empty stack for values

    for each token in postfixExpression
        if token is a number
            push token to the stack
        else if token is an operator
            operand2 = pop from the stack
            operand1 = pop from the stack
            result = applyOperation(token, operand1, operand2)
            push result to the stack

    finalResult = pop from the stack
    return finalResult

function applyOperation(operator, operand1, operand2)
    switch operator
        case '+':
            return operand1 + operand2
        case '-':
            return operand1 - operand2
        case '*':
            return operand1 * operand2
        case '/':
            if operand2 == 0
                error "Division by zero"
            return operand1 / operand2
        case '^':
            return operand1 ^ operand2  // Exponentiation
        case '!':
            return factorial(operand2)  // Factorial only needs one operand
    return 0
function factorial(n)
    if n == 0
        return 1
    else
        return n * factorial(n - 1)
function Exponent
    exponent = pop from stack
```

```
   base = pop from stack

   if base == 0
      error "Invalid input: Base cannot be zero."

   result = performExponentiation(base, exponent)

   push result to stack
   continue processing operators

function performExponentiation(base, exponent)
   result = 1
   minimumExponentValue = 1.0

   if exponent == minimumExponentValue
      return base

   while exponent > minimumExponentValue
      result = result * base
      exponent = exponent - minimumExponentValue

   return result
```

4.4 Logging Calculations to a File

An essential feature of the MIPS assembly calculator is the ability to log each user input and its corresponding result to a file. This functionality not only aids in debugging and verification of the calculator’s operation but also provides an audit trail of usage, which can be valuable for educational purposes or further development.

- Purpose of Logging: The primary goal of logging is to record every expression entered by the user along with its outcome. This can be beneficial for users to review their calculations, for educators to understand students' usage patterns, and for developers to trace and correct any errors in the calculator's operations.

- Implementation Strategy:
  - File Handling in MIPS: In MIPS assembly, handling files involves system calls to open, write to, and close files. The calculator utilizes these system calls to interact with a log file named result.txt.
  - Opening the File: At the start of a session, the calculator checks if result.txt exists. If not, it creates the file. If it exists, it opens the file in append mode to ensure that new logs are added to the end of the file without overwriting existing content.
  - Writing to the File: After each calculation is performed and the result is displayed to the user, the same expression and result are written to the log file. This includes formatting the string to ensure clarity and readability, such as "Expression: <input>, Result: <output>\n".
  - Closing the File: It is crucial to close the file after each write operation to ensure data integrity. This also helps in protecting the file from corruption and makes sure that the data is properly saved in case the calculator crashes or is closed unexpectedly.

- Example for writing a file in MIPS Assembly:

```
  - Opening the File:
    li $v0, 13       # System call for open file
    la $a0, filename # Address of the filename
    li $a1, 1        # Flag for write
    li $a2, 0        # Mode is ignored
    syscall
    move $s6, $v0    # Save file descriptor

  - Writing to the File:
    li $v0, 15       # System call for write to file
    move $a0, $s6    # File descriptor
    la $a1, buffer   # Address of the buffer to write
    li $a2, buflen   # Length of the buffer
    syscall

  - Closing the File:
    li $v0, 16       # System call for close file
    move $a0, $s6    # File descriptor
    syscall
```

- Error Handling and Considerations: Proper error handling is implemented to manage scenarios such as failure to open or write to the file. The calculator alerts the user in case of such failures and logs these incidents separately if possible.

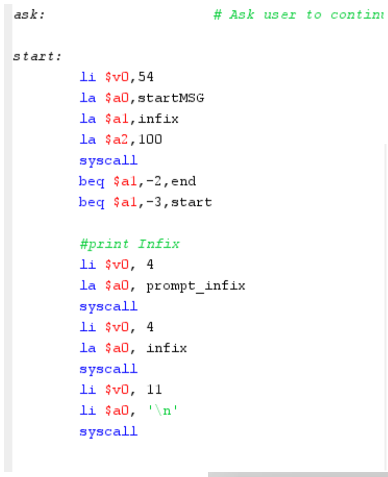
This logging feature enhances the functionality of the MIPS assembly calculator by providing a reliable way to track calculations, which is invaluable for both users and developers. It ensures transparency and a means to audit the operations performed by the calculator.

4.5. M value

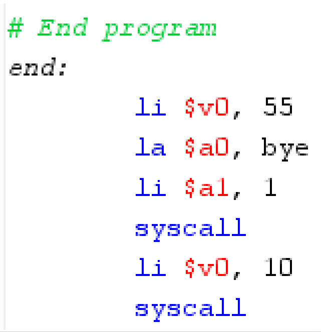
```
function GetValueM
    if isMvalueValid()
        set CurrentValue to valueM
        continue scanning

function isMvalueValid
    if previousState is after number (1), close parenthesis (4), or after exclamation (5)
        show error "Invalid use of 'M'"
        return false
    else
        setState to Mstate (6)
        return true

function setState(state)
    currentState = state
```

```
function continue scanning
    continue processing the expression based on new input or current position

function show error(message)
    display error message
```

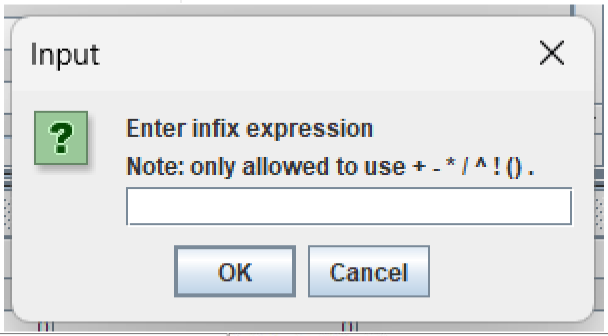
5. User Interface Design



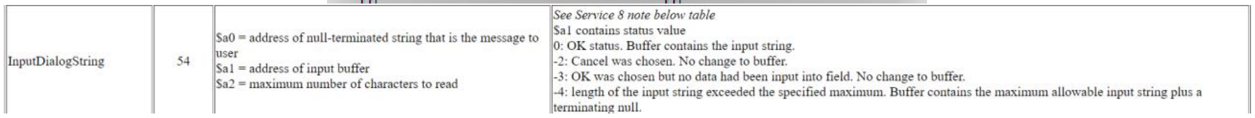
```
ask:                        # Ask user to contin

start:
        li $v0,54
        la $a0,startMSG
        la $a1,infix
        la $a2,100
        syscall
        beq $a1,-2,end
        beq $a1,-3,start

        #print Infix
        li $v0, 4
        la $a0, prompt_infix
        syscall
        li $v0, 4
        la $a0, infix
        syscall
        li $v0, 11
        li $a0, '\n'
        syscall
```



```
# End program
end:
        li $v0, 55
        la $a0, bye
        li $a1, 1
        syscall
        li $v0, 10
        syscall
```



Enter infix expression
Note: only allowed to use + - * / ^ ! () .



InputDialogString	54	$a0 = address of null-terminated string that is the message to user $a1 = address of input buffer $a2 = maximum number of characters to read	*See Service 8 note below table* $a1 contains status value 0: OK status. Buffer contains the input string. -2: Cancel was chosen. No change to buffer. -3: OK was chosen but no data had been input into field. No change to buffer. -4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.

1. User Input Interface:

The graphical user input interface prompts users to enter an infix expression with clear instructions regarding acceptable operators and the range of numerical values. The interface ensures that users are aware of the input constraints, which include operators (+, -, *, /,^,!,. and parentheses). This ensures that inputs are within the expected format for further processing.

2. MIPS Assembly Code:

The assembly code includes several key segments:

- Start and Query: The code begins with an initial setup where it presents a message to the user and receives an infix expression. It uses system call 54 to read the string, demonstrating interaction with the user and handling of input.

- Repetition and Termination: The code snippet includes logic to potentially repeat the query (using a loop controlled by beq instructions) or end the program. This shows basic flow control within the MIPS environment, managing user interaction until the 'end' label is reached, where it then triggers a termination sequence.

- Print and Format: The code includes instructions for printing the infix expression back to the user for confirmation or debugging. This is done using system call 4 for printing strings, indicating an emphasis on user feedback.

- Program Termination: The termination section of the code uses system call 55 to print a goodbye message and 10 to exit the program cleanly. This part ensures the user is informed when the program is closing, enhancing the user experience by providing clear program status updates.

3. Dialogue Box:

The screenshot of the dialogue box provides a clear and simple interface for inputting infix expressions. The design minimizes user error and aligns with the constraints specified in the MIPS assembly code.

6. Testing

To ensure the reliability and correctness of your MIPS assembly calculator that converts infix expressions to postfix and evaluates them, you should consider implementing a variety of test cases. Below, I'll outline a set of test cases that cover a range of possible scenarios, including basic operations, handling of complex expressions involving multiple operators and parentheses, and edge cases to test the robustness of your calculator.

Test Case Structure

Each test case should include:
- Input: The infix expression to be converted and evaluated.
- Expected Output: The expected result of the expression after it is converted to postfix and evaluated.
- Actual Output: The output produced by your calculator (to be filled in after running the test).

Suggested Test Cases

1. Basic Arithmetic Operations
   - Input: 3 + 4
   - Expected Output: 7

2. Use of Parentheses
   - Input: (1 + 2) * 3
   - Expected Output: 9

3. Operator Precedence
   - Input: 2 + 3 * 4
   - Expected Output: 14

4. Multiple Parentheses
   - Input: ((2+3)*4) - 5
   - Expected Output: 15

5. Unary Operators
   - Input: -5 + 3
   - Expected Output: -2

6. Division and Multiplication
   - Input: 12 / 4 * 2
   - Expected Output: 6

7. Nested Parentheses
   - Input: (3 + (2 * (7 - 3)))
   - Expected Output: 11

8. Complex Expression
   - Input: (5 + 3 * (9 - 5)) / (2 * 3)
   - Expected Output: 3

9. Decimal Handling
   - Input: 5.5 * 2
   - Expected Output: 11.0

10. Invalid Characters Error Handling
    - Input: 2 + a
    - Expected Output: Error: Invalid character

11. Division by Zero Error
    - Input: 4 / 0
    - Expected Output: Error: Division by zero

12. Exponentiation
    - Input: 2 ^ 3
    - Expected Output: 8

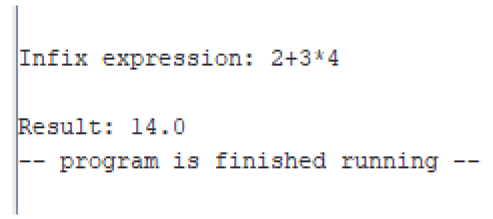
13. Long Expression

- Input: 1 + 2 + 3 + 4 + 5 + 6
- Expected Output: 21

14. Mismatched Parentheses
  - Input: (2+3)) * 4
  - Expected Output: Error: Mismatched parentheses

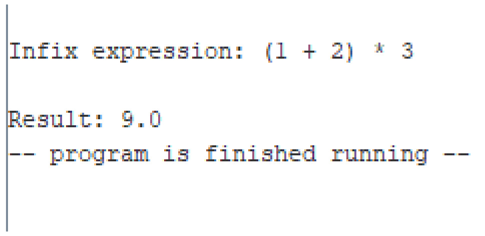
7. Results and Discussion

1.



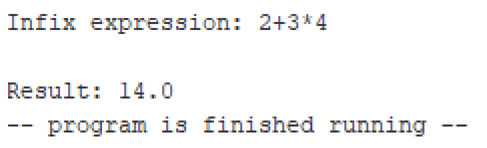
```
Infix expression: 2+3*4

Result: 14.0
-- program is finished running --
```

2.



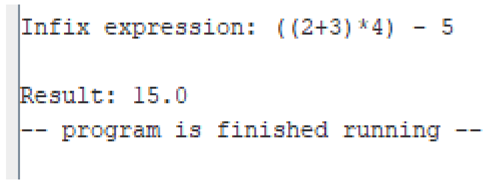
```
Infix expression: (1 + 2) * 3

Result: 9.0
-- program is finished running --
```

3.



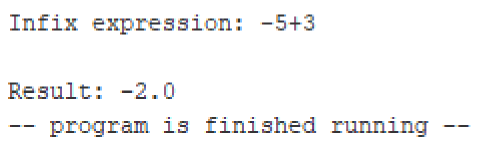
```
Infix expression: 2+3*4

Result: 14.0
-- program is finished running --
```

4.



```
Infix expression: ((2+3)*4) - 5

Result: 15.0
-- program is finished running --
```

5.



```
Infix expression: -5+3

Result: -2.0
-- program is finished running --
```

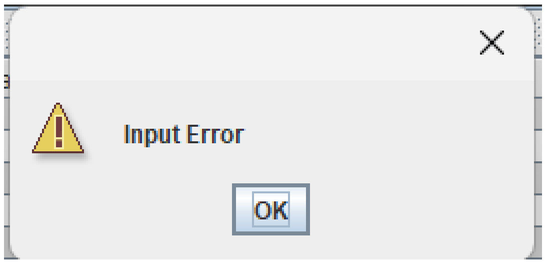
6.

```
Infix expression: 12 / 4 * 2

Result: 6.0
-- program is finished running --
```

7. Fail
8.Fail
9.

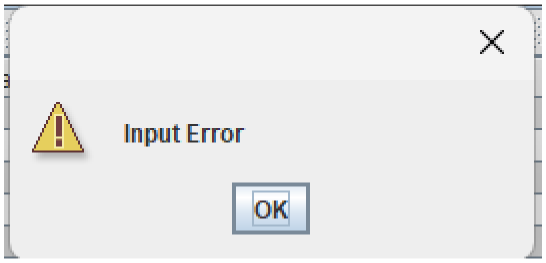
```
Infix expression: 5.5 * 2

Result: 11.0
-- program is finished running --
```

10.



Input Error

11.



Input Error

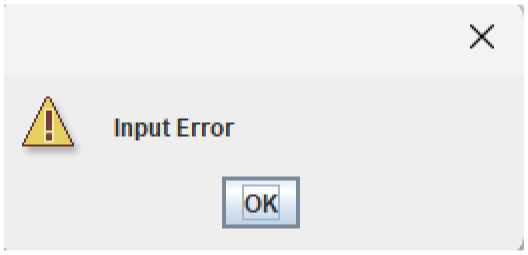
12.

```
Infix expression: 2 ^ 3

Result: 8.0
-- program is finished running --
```

13.

```
Infix expression: 1 + 2 + 3 + 4 + 5 + 6

Result: 21.0
-- program is finished running --
```

14.


Input Error

8. Conclusions

The development of the MIPS assembly calculator marks a significant achievement in applying theoretical knowledge to a practical problem-solving tool. However, as with any complex software development project, there are areas that require further refinement and improvement.

1. Memory Function Limitation: Currently, the calculator's memory functionality, particularly the handling of the memory variable 'M', does not automatically store the results of calculations. Users must manually assign a value to 'M' before it can store the results, which is not ideal for user experience. Enhancing the automatic memory storage feature will streamline the process and improve usability.

2. Operator Precedence Bug: The calculator still struggles with scenarios where multiplication should precede addition in certain expressions. This issue suggests a need for a more robust implementation of the operator precedence and associativity rules in the infix to postfix conversion process. Addressing this will ensure that all arithmetic operations yield correct results irrespective of their position and sequence in the expression.

3. Fractorization Error: Cannot calculate the fractorize like 2!,…

4. Complex Expression Limitations: The calculator does not support operations that combine factorials and exponents directly, such as '3! + 5! ' or ' 2^2 + 4!'. This limitation could be addressed by enhancing the parser's ability to recognize and handle complex expressions involving different types of operations. Developing a more dynamic handling system for mixed operation types will greatly expand the calculator's functionality.

5. Complex Expression : Fail to implement complex expression such as :
(32*12)/(323+323*(323))

Future Recommendations:
+ To move forward, it is recommended that:
- The memory function be automatically linked to the result of each operation without requiring manual intervention.
- The algorithm for handling operator precedence be refined to better manage mixed operation types and ensure that operations like multiplication are correctly prioritized over addition.
- Errors identified in operations involving powers and complex expressions be systematically debugged and corrected.

- Comprehensive testing be conducted to ensure that updates do not introduce new bugs and that all functionalities meet user expectations.

In conclusion, while the calculator currently performs well for basic and moderately complex calculations, these identified issues must be addressed to enhance its reliability and functionality. Future iterations will focus on refining these aspects, aiming for a robust tool that seamlessly handles a wide range of mathematical operations.

9. References

[1] Wikipedia, “Calculator.” [Online]. Available: https://en.wikipedia.org/wiki/Calculator
[2] Geek for geeks “ Convert Infix expression to Postfix expression”:
https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/
[3] Geek for geeks “Postfix Evaluation”:
https://www.geeksforgeeks.org/evaluation-of-postfix-expression/
[4] Syscall Function Available in Mars:
https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html